Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets

Christof Tinnes[§]
Siemens AG
Garching bei München, Germany
christof.tinnes@siemens.com

Alisa Welter[§]
Saarland University
Saarbrücken, Germany
welter@cs.uni-saarland.de

Sven Apel
Saarland University
Saarbrücken, Germany
apel@cs.uni-saarland.de

***Abstract*—Modeling structure and behavior of software systems plays a crucial role in the industrial practice of software engineering. As with other software engineering artifacts, software models are subject to evolution. Supporting modelers in evolving software models with recommendations for model completions is still an open problem, though. In this paper, we explore the potential of large language models for this task. In particular, we propose an approach, RAMC, leveraging large language models, model histories, and retrieval-augmented generation for model completion. Through experiments on three datasets, including an industrial application, one public open-source community dataset, and one controlled collection of simulated model repositories, we evaluate the potential of large language models for model completion with RAMC. We found that large language models are indeed a promising technology for supporting software model evolution (62.30% semantically correct completions on real-world industrial data and up to 86.19% type-correct completions). The general inference capabilities of large language models are particularly useful when dealing with concepts for which there are few, noisy, or no examples at all.**

I. INTRODUCTION

Models play an important role in modern software and system development [61], software documentation [42, 56], system architecture [57], simulation [21], and industrial automation [34]. In practice, all artifacts in software and system development are subject to evolution, which also applies to *software models*[1]: Software models must evolve because of changing requirements, but they are also subject to bugfixes and refactorings [72].

From the perspective of a modeling tool, we can understand the evolution of a software model as a sequence of *edit operations*: To change or evolve the model, the user executes edit operations (e.g., using mouse clicks and keyboard strokes) provided by the tool. Supporting tool users in accomplishing various software model (evolution) tasks is clearly desirable in practice [23, 71]. For the evolution of software models, modeling tools typically provide an initial set of edit operations (e.g., adding an attribute to a model element). Nevertheless, since the usage of a (domain-specific) language is also subject to evolution and since (project-specific) usage patterns might emerge, this initial set of edit operations is likely not exhaustive. For example, in object-oriented design, design patterns [30] are widely used and are not part of UML [56], but could be provided as edit operations by a UML modeling tool.

For source code, modern integrated development environments already support writing and evolving source code by *(auto-)completion*. Most notably, the use of large language models (LLMs) has become state-of-the-art for the auto-completion of source code [18, 77, 5, 29, 6, 75].

The world of software models seems to be lagging behind, and no general approach for software model auto-completion is ready for industrial application. It has been even argued that the so-called cognification of use cases in model-driven software engineering might turn the difference between (perceived) added value and cost from negative to positive [13].

Problem Statement. Notably, for a few domain-specific languages, rule-based approaches exist that use pre-defined edit operations or patterns for model completion [43, 44, 32, 64]. Using a specification language for defining edit operations poses three challenges, though. First, specifying new edit operations requires knowledge about the specification language and the domain-specific language. Second, domain-specific edit operations are often not explicitly known, that is, they are a form of tacit knowledge [58]. Externalizing the knowledge is hard or even impossible for domain experts. Third, edit operations can change over time, for example, because the metamodel changes. In the light of these challenges, mining approaches that retrieve edit operations are especially appealing, since they do not require any manual specification, no hand-crafting of examples (as in model transformation by example [73, 38]), and they are not limited to well-formedness rules that can be derived out of the metamodel. Unfortunately, existing approaches such as applying frequent subgraph mining to software model repositories are not scalable [71], and mining approaches lack abstraction capabilities [71].

Clearly, from the perspective of software model evolution, it is desirable to have *context-dependent auto-completions*, rather than utilizing a fixed set of edit operations. We posit that generative language models exhibit a deep understanding of language and hold comprehensive knowledge across various domains, which is a result of their training on vast corpora. This

[§] Equal contribution [1] In our work, to avoid confusion, it's crucial to differentiate between software models and machine learning models.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00112

capability enhances their potential to interpret and complete software models effectively, which usually encompass a vast amount of natural language data.

While recent research suggests that LLMs could be utilized for model completion [16], we go beyond and utilize model evolution data from model repositories to capture real-world complexities. It is important to note that, in our work, we explicitly acknowledge the complexity of real-world data, which is due to the close collaboration with our industry partner (who also contributes a case study).

Contributions. By leveraging existing software model histories[2], and by defining an encoding for serializations of model difference graphs, we study to what extent retrieval-augmented generation, (i.e., we provide examples as context in the prompt) can be used for software model completion. We find that RAMC is indeed a promising approach for software model completion, with 62.30% of semantically correct completions. We furthermore propose to use fine-tuning (i.e., the LLM's weights are adapted by training on parts of our data) for software model completion and compare it to our retrieval-based approach, RAMC. LLM's general inference capabilities prove especially helpful in handling noisy and unknown context, and real-time capabilities enabled by LLMs are beneficial for stepwise model completion. We conclude that using LLMs for software model completion is viable in practice (despite various complexities), but further research is necessary to provide more task and domain knowledge to the LLM.

In summary, we make the following contributions:

- As a foundation for applying LLMs, we formalize the concept of software model completion based on change graphs and their serialization.
- We propose a retrieval-augmented generation approach, RAMC, for software model completion.
- We evaluate RAMC qualitatively and quantitatively on three datasets, including an industrial application, one public open-source community dataset, and one controlled collection of simulated model repositories. We compare our approach with the most recent advancements in model completion [16] as well as to the alternative of fine-tuning a pre-trained LLM. We find that, for all three datasets, LLMs are a promising technology for software model completion, with up to 86.19% correct completions (for the synthetic dataset) and 62.30% of semantically correct completions on the industrial dataset. Notably, our approach improves significantly over the state of the art [16]. Furthermore, it appears that fine-tuning can be an alternative to retrieval-augmented generation that is worthwhile investigating.

Source code for the experiments, scripts, public datasets, and results are publicly available (see Section VII).

[2] Note that we use the terms *software model repositories* and *software model histories* interchangeably, and we assume that the repository contains several revisions of a software model.

II. Related Work

Various approaches have been proposed for software model completion, ranging from rule-based approaches to data mining techniques and more sophisticated machine learning approaches. An overview of recommender systems in model-driven engineering is given by Almonte et al. [7]. Some of the previous work studies recommending model completions by utilizing knowledge bases such as pattern catalogs or knowledge graphs [2, 43, 44, 50, 46, 23, 48]. Consequently, these research efforts are often domain-specific, as they require the provision of domain-specific catalogs (a.k.a., the cold start problem), such as for UML [43, 44, 50] or business process modelling [23, 46].

Another common approach is to use already existing model repositories and employ techniques such as frequency-based mining, association rule mining, information retrieval techniques, and clustering to suggest new items to be included in the model [1, 68, 24, 28] or new libraries for use [32]. MemoRec [24] and MORGAN [26] are frameworks that use a graph-based representation of models and a similarity-based information retrieval mechanism to retrieve relevant items (such as classes) from a database of modelling projects. However, their graph-based representation does focus on the relationship between a model element and its attributes, but it does not capture relationships *between* different elements in the model and consequently may not capture the essential semantics and constraints of the model and modelling languages. Repository mining and similarity-based item recommendation techniques are often combined [23, 46]. Kögel et al. [41, 40] identify rule applications in current user updates and find similar ones in the model's history. More generally, one could automatically compute consistency-preserving rules [39] or pattern mining approaches [71, 70, 45] to derive a set of rules to be used in conjunction with a similar association rule mining approach.

Another strategy to generate model completion candidates that comply with the given metamodel and additional constraints involves using search-based techniques [66]. Without knowledge about higher-level semantics, these approaches are more comparable to the application of a catalog of minimal consistency-preserving edit operations [39].

Regarding the application of natural language processing (NLP) [12] and language models [19, 76], Burgueño et al. [12] propose an NLP-based system using word embedding similarity to recommend domain concepts. Weyssow et al. [76] use a transformer-based language model to recommend metamodel concepts without generating full model completions. Di Rocco et al. [25] introduce a recommender system using an encoder-decoder neural network to assist modelers with editing operations. It suggests element types to add, but leaves the specification of details, values, and names of these elements and operations to the human modeler. Gomes et al. [31] use natural language processing to translate user intents, expressed in natural language, into actionable commands for developing and updating a system domain model. While code completion and model completion are closely related, recent research has mainly concentrated on code completion, where LLMs seem

to be the state of the art [18, 36, 20, 65]. Considering the close connection to code and model completion, it's essential for us to explore further how generative approaches, such as LLMs, operate within the context of software model completion of complex real-world models. Most closely to this work, is an approach by Chaaben et al. [16], which utilized the few-shot capabilities of GPT-3 for model completion by providing example concepts of unrelated domains. In contrast, our approach takes a different avenue, leveraging model evolution from model repositories. Cámara et al. [14] further extend on Chaaben et al.'s research by conducting experiments to assess ChatGPT's capability in model generation. Ahmad et al. explore the role of ChatGPT in collaborative architecting through a case study focused on defining Architectural Significant Requirements (ASRs) and their translation into UML [4]. On the supplementary website[3], a table summarizing related work on model completion is provided.

A slightly different but similar research area focuses on model repair [52, 35, 51, 49, 54, 68]. REVISION [54] uses so-called consistency-preserving edit operations to detected inconsistencies and then uses the pre-defined edit operations to recommend repair operations.

III. FORMAL DEFINITIONS

In this section, we describe the fundamental concepts essential for the subsequent approach and analysis.

A. Software Models, Edit Operations and Model Completion

In model-driven engineering, the language for a software model (i.e., its abstract syntax and static semantics) is typically defined by a metamodel TM. We denote by $\mathcal{M}$ the set of all valid models (according to some metamodel). This can be formalized using typed attributed graphs [9, 27].

Definition III.1 (Abstract Syntax Graph)**.** An *abstract syntax graph* G_m of a model $m \in \mathcal{M}$ is a attributed graph, typed over an attributed type graph TG given by metamodel TM.

The idea of typed graphs is to define a graph homomorphism (i.e., a function from the typed graph G to the type graph TG). Details of this formalization are given by Biermann et al. [9]. The abstract syntax graph of a model and its type graph contain all information that a model holds. In this paper, we are concerned with model repositories. We assume that the modelling tool takes care of checking the correct typing of the software models. Furthermore, we work with a simplified graph representation of the models in which the abstract syntax graph is a *labeled directed graph* with node and edge labels equal to a textual representation of corresponding classifiers and relationships of the abstract syntax graph (cf. Definition III.1).

Definition III.2 (Labeled Directed Graph)**.** Given a label alphabet L, a *labeled directed graph* G is a tuple (V, E, λ), where V is a finite set of nodes, E is a subset of $V \times V$, called the edge set, and $\lambda : V \cup E \to L$ is the labeling function, which assigns a label to nodes and edges.

[3] https://github.com/se-sic/icse_model_completion/blob/main/MC_Preprint_and_Appendix.pdf

Rather than working directly on the abstract syntax graph of the models, we will mostly be working with model differences.

Definition III.3 (Structural Model Difference)**.** A *structural model difference* Δ_{mn} of a pair of model versions m and n is obtained by matching corresponding model elements in the model graphs G_m and G_n (using a model matcher [69], e.g., EMFCompare [10] or SiDiff [63]). There are added elements (the ones present in G_n but not in G_m), removed element (the ones present in G_m but not in G_n), and preserved elements which are present in G_m and G_n.

We assume that this matching is deterministic, that is, given two models $m, n \in \mathcal{M}$, we obtain a unique structural model difference Δ_{mn}. The difference can be represented as a *difference graph* $G_{\Delta mn}$ [54]. More concretely, we add the change type ("Add", "Preserve", or "Remove") in the node and edge labels, and matching elements (i.e., the preserved ones) from G_m and G_n are unified (present only once).

We define a *simple change graph* to be the smallest subgraph comprising all changes in the difference graph $G_{\Delta_{mn}}$.

Definition III.4 (Simple Change Graph)**.** Given a difference graph $G_{\Delta_{mn}}$, a *simple change graph* $SCG_{\Delta_{mn}} \subseteq G_{\Delta_{mn}}$ is derived from $G_{\Delta_{mn}}$ by first selecting all the elements in $G_{\Delta_{mn}}$ representing a change (i.e., added, removed nodes and edges) and, second, adding preserved nodes that are adjacent to a changed edge.

Definition III.5 (Endogenous model transformation)**.** An *endogenous model transformation* is a pair $t = (m, n) \in \mathcal{M} \times \mathcal{M}$. We call m the *source model* and n the *target model* of the transformation and $\mathcal{T} \stackrel{\text{def}}{=} \mathcal{M} \times \mathcal{M}$ the space of endogenous model transformations.

Next, we define a function $SCG\colon \mathcal{T} \to \mathcal{G}$ that takes a model transformation (i.e., a pair of models) as input and returns the simple change graph for the corresponding model difference. We can use SCG to define an equivalence relation on $\mathcal{T}$ by

$$t_1 = (m, n) \sim t_2 = (k, l) \iff SCG_{\Delta_{mn}} = SCG_{\Delta_{kl}}.$$

It is straightforward to see that this relation indeed defines an equivalence relation (i.e., the relation is reflexive, symmetric, and transitive). We can therefore define the quotient set $\mathcal{T}/\sim$. By construction there is bijection from the quotient set to the range of SCG. We can therefore use this construction to formally define the concept of an *edit operation*.

Definition III.6. An *edit operation* is an equivalence class in the set $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{T}/\sim$. An edit operation is therefore a set of model transformations that have the same simple change graph.

Remark. The graph labeling function λ allows us do define the scope of the edit operation. For example, if we are interested only in the type of nodes and edges, we can omit the attributes from the label. Likewise, if we are interested in the attributes, or only want to set them during execution time, we can define placeholders for the attribute values in

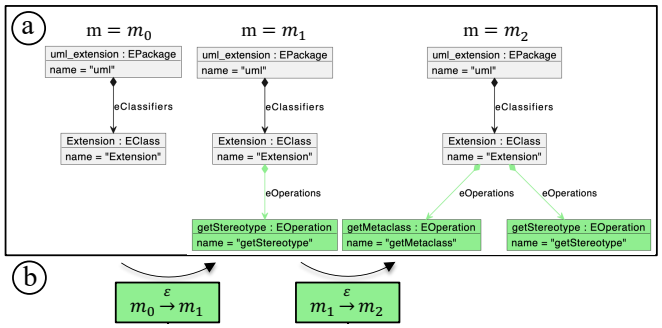

Figure 1: Visual presentation of our example taken from the REPAIRVISION dataset: ⓐ Evolutionary View: User performs edit operations one by one. ⓑ Evolution can be performed by a user or by using a completion approach.

the labels. Therefore, we define edit operations only up to the concrete label representation, which leaves some freedom for templating. In this work, we do make use of placeholders only during the evaluation (e.g., checking for type correctness).

Given an edit operation ε and a model m, one can perform the removal of "Remove" nodes and the gluing of "Add" nodes as defined by the simple change graph corresponding to ε, and then set concrete attributes. This yields the corresponding model n with $(m, n) \in \varepsilon$. This way, an edit operation $\varepsilon \in \mathcal{E}$ can be interpreted as a template for a model transformation, which is in line with previous constructions [9, 37, 71]. We write $m \xrightarrow{\varepsilon} n$ to denote a concrete element (i.e., a model transformation) in the equivalence class $\varepsilon \in \mathcal{E}$. We are interested in completing software models. That is, for an existing evolution $m \xrightarrow{\varepsilon} n$, we want to find a completion $\gamma \in \mathcal{E}$, such that $m \xrightarrow{\varepsilon} n \xrightarrow{\gamma} c$ is a realistic completion, meaning, in some real-world scenarios, it actually will be done by a modeler.

Definition III.7 (Model Completion)**.** Given a set of model transformations $\mathcal{T}$, *model completion* is a computable function $C : \mathcal{T} \to \mathcal{T}$ that, given a model transformation $m \xrightarrow{\varepsilon} n$ from a source model m to a (partial) target model n, computes a model transformation $C(m \xrightarrow{\varepsilon} n) = n \xrightarrow{\gamma} c$. We call the edit operation γ a *software model completion*.

Given a model completion γ, we denote the application of γ to model n by $\pi : \mathcal{M} \times \mathcal{E} \to \mathcal{T}$, where $\pi(m, \gamma \circ \varepsilon) = (n, c)$. In general, for an edit operation ε, there might be zero or more applications to a given model $m \in \mathcal{M}$. Nevertheless, given that the matching in n is fully defined by the application of ε, there is a uniquely defined candidate $(n, c) \in \mathcal{T}$.

B. Language Models

Language models, as *generative models*, have the capability to produce new sequences of text based on their training data.

Definition III.8 (Language Model)**.** A *language model* is a conditional probability distribution $\mathbb{P}(\omega|c)$ for a (sequence of) token(s) ω, given a sequence of context tokens c.

The probability distribution is typically derived from a *corpus* of documents, containing (some of) the tokens. With the success of transformer architecture [74], LLMs have become quite popular now and are used in plenty of domains including software engineering [62, 79, 78]. There are two tactics available to feed domain knowledge or context into a generative language model: fine-tuning and retrieval-augmented generation. Retrieval-augmented generation includes additional knowledge in the context (or prompt). Fine-tuning adjusts the LLM's weights based on additional training data.

IV. APPROACH

In this section, we describe RAMC – our approach of *how* to employ LLMs to (auto-)complete software models.

A. Running Example

Consider the motivating example depicted in Figure 1, which originates from one of our datasets, REPAIRVISION, further explained in Section V-B. In ⓐ, we show the evolution of its abstract syntax graph[4]. In this evolution scenario, a modeller adds the UML Profiles mechanism (cf. UML specification [56], Chapter 12.3) to the ECORE metamodel[5] of UML 2.5.1. Step by step the modeller extends the existing UML metamodel with additional functionality, currently focusing on the EClass extension in the UML package. In a first step, the modeller adds an operation getStereotype (responsible for accessing the Sterotype of the extensions associated with an element in the (meta-)model). As defined in the UML specification [56], every extension has access to the Metaclass it extends, realized in ECORE by the EOperation getMetaclass. This EOperation is implemented by the modeller in a second step. These steps in the evolution of the UML metamodel could be performed via edit operations by a human user, or likewise, recommended in the form of a model completion (as depicted in ⓑ of Figure 1).

B. Overview and Design Choices

Utilizing LLMs for software model completion gives rise to several challenges addressed by RAMC: how to provide context, such as domain knowledge, to the LLM, how to serialize software models, and how to deal with limited context[6]?

Regarding context, we opt for retrieval-augmented generation, and compare the approach to fine-tuning in one of our experiments. The next important design decision is that we do not work on the software models directly but on the simple change graphs, described in Section III. The basic idea is that simple change graph completions can be straight forwardly interpreted as model completions (i.e., generating a new "added" node corresponds to adding a new model element to the model). Working with the concept of a simple change graph has several advantages: First, we do not have to work with the entire software model representation, but we can focus on slices of

[4] Due to obvious space constraints, only a small part of the original model (only one out of 256 classifiers and 2 out of 741 operations) is shown [5] UML, according to the Meta-Object Facility [55], is itself a model according to its meta-metamodel, ECORE, and therefore covered by the present work. [6] Software models can become huge compared to the limited number of tokens that can be given to a LLM.

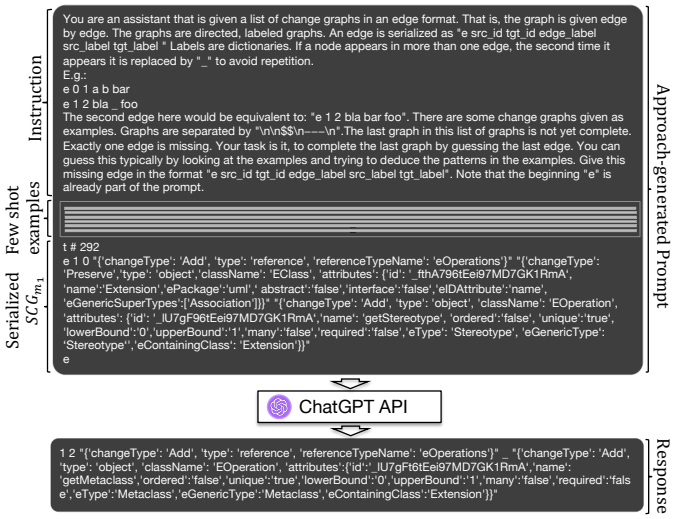

Figure 2: Detailed prompt and simple change graph serialization of the RAMC approach corresponding to the example given in Figure 1, exact few-shot examples are provided in supplementary website due to space constraints.

the models around recently changed elements. This is one tactic of dealing with the common problem of the limited context of a LLM. For example, in our running example, the entire (serialized) UML metamodel is huge and would not fit in the context of contemporary LLMs.

Second, simple change graph completions also include attribute changes and deletions of model elements and are not limited to the creation of new model elements. RAMC is capable of suggesting semantically appropriate changes, such as renaming an attribute or altering the type of an attribute. Additionally, it recommends specific attribute values that are beyond predefined options, for example, values for string type attributes. Although alternative representations besides simple change graph can influence the outcome, choosing simple change graph was a deliberate design decision we made.

An overview of the approach is depicted in Figure 3, the computation of model differences (Figure 3, ①) and simple change graphs (Figure 3, ②) is explained in Section III. Their serialization will be addressed in the next subsection. Based on the terminology in Section III, the formalization of our approach RAMC is given on our supplementary website.

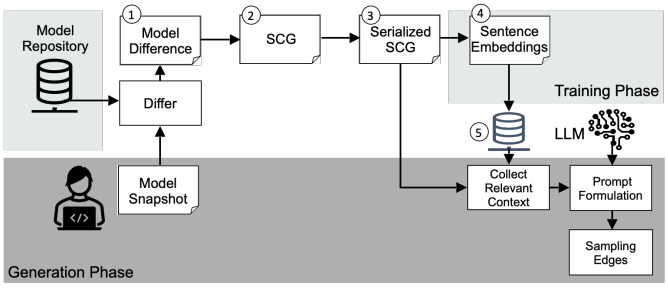

Figure 3: Overview of RAMC.

C. Pre-processing

Both training phase and generation phase work on serializations of simple change graphs. We describe how these serializations are derived based on the example given in Figure 1. Input to this procedure are two (successive) revisions of a model; output is a serialization of their simple change graphs. These revisions can originate either from the model the user is working on (in the generation phase) or from our training data.

In the first step, a model difference is computed for each pair of successive revisions of a model (Figure 3, ①). Regarding our running example in ⓐ of Figure 1, we also highlighted these model differences by color, that is, "added" model elements are depicted in green. From this model difference, we compute a (partial) simple change graph (see Definition III.4 and Figure 3, ②). Finally, the simple change graph is serialized as a list of edges (Figure 3, ③). To this end, we defined a graph serialization, called *EdgeList*, for directed labeled graphs. Figure 2 presents the prompt generated from our approach alongside the corresponding response, which was retrieved via API access to ChatGPT. It also shows an example of this graph serialization (e.g., last part of the prompt), which contains all kinds of attribute information. It can quickly become verbose and noisy in real-world examples. Common formats such as the GraphML[7] are less suitable for LLMs, since they list vertices before edges. This requires guessing all nodes first – added, deleted, and preserved – before generating edges.

D. Training Phase

The *input* to the training phase is a set of serialized simple change graph components. The *output* is a (vector) store of serializations with a key for retrieval (Figure 3, ⑤). We retrieve relevant simple change graphs from model repositories by utilizing a *similarity search* based on sentence embeddings [59]. The serializations are stored in a vector database together with their sentence embedding (Figure 3, ④ and ⑤).

E. Generation Phase

The *input* to the generation phase is a set of serialized simple change graph components capturing the difference of a new model snapshot (i.e., local changes) and the previous model revision ($m_1 \xrightarrow{\varepsilon} m_2$), as well as the vector store from the training phase. The *output* is a (list of) completion(s) in the form of EdgeList serializations, which are suggested to the user after being parsed (an example is given in Figure 2, at the bottom under 'Response').

Retrieval. The vector store is queried for simple change graph serializations via a similarity-based retrieval. Note that, in our case the retrieved context can be interpreted as *few-shot examples*, because we retrieve complete simple change graphs, that is, completed partial simple change graphs from the history. The few-shot samples from Figure 2 are detailed on our supplementary website, due to space limitations. To ensure a diversity of samples, we use a procedure similar to maximum marginal relevance [15], explained in detail on the supplementary website. As few-shot samples, we select

[7] http://graphml.graphdrawing.org

up to 12 serialized simple change graphs; we investigate the dependency on the number of few-shot samples in Section V.

Prompt formulation. The prompt (input to the LLM) used by our approach consists of an instruction at the beginning, followed by the few-shot samples retrieved from the vector store (joined via a separation token), and finally the (partial)-simple change graph serialization is concatenated (see Figure 2).

Sampling new edges. We can sample multiple completion candidates from the LLM by using a beam search or by instructing the LLM to generate multiple edges. Details of the edge sampling are given on the supplementary website.

F. Implementation

We have implemented the computation of model differences and simple change graphs on top of the ECLIPSE MODELING FRAMEWORK [67], using SIDIFF [63] for matching and diffing. The other parts are implemented in PYTHON3, mainly utilizing NETWORKX[8] for handling graphs. We use LANGCHAIN[9] for the handling of language models and retrieval-augmented generation. We use the ALL-MINILM-L6-V2[10] language model for the sentence embeddings since it performed well in preliminary experiments. As vector store, we use CHROMADB[11]. As language model, we use GPT-4 (version 0613), since it performed best in preliminary experiments. We use a dedicated deployment of OpenAI on Microsoft Azure that is certified for the classification level of the industrial data.

V. EVALUATION

We evaluate to what extent our approach is able to derive structurally and semantically correct completion operations from the software model history. This includes, in particular, their applicability in industrial scenarios. We aim at a systematic evaluation of LLMs for model completion in a controlled setting. This allows us to concentrate on the core effectiveness of LLM technology, while controlling for confounding factors such as tool use and human aspects (e.g., UX design facets). This is also the reason why, at this stage, conducting a user study settled in a specific application context would be not opportune (but needs to follow at a later stage). However, by applying our approach to a real-world context at our industry partner, who expressed clear interest in and demand for this technology, we establish a solid methodological and empirical foundation, before considering the development of sophisticated and potentially costly tools.

A. Research Questions

To understand the merits of language models for model completion, we want to answer the following research questions:

RQ 1: *To what extent can pre-trained language models and retrieval-augmented generation be used for the completion of software models?*

Clearly, a general pre-trained language model is typically not aware of the syntax and domain-specific semantics of the simple change graph serializations *per se*. This includes the definition of the graph serialization format, the definition of simple change graphs, the metamodel, and the domain-specific semantics of the software models not already encoded in the metamodel. For example, a generated completion might be invalid according to the metamodel, (e.g., invalid combination of edge, source, and target node labels) or could even result in an invalid directed labeled graph serialization (e.g., they do not adhere to the EdgeList format).

RQ 2: *What influence does semantic retrieval have on the performance of* RAMC*?*

As motivated in Section IV, providing context that is semantically close to a to-be-completed change could improve the correctness of retrieval-augmented generation. We therefore want to understand the influence of the similarity-based retrieval on model completion. That is, we want to compare semantic retrieval and random retrieval of few-shot examples and to analyze the influence of the number of few-shot examples.

RQ 3: *How does* RAMC *compare against the state of the art (Chaaben et al. [16])?*

We evaluate the accuracy of our proposed approach, RAMC, by comparing it to the closely related work of Chaaben et al. [16], which we use as a baseline. Their study focuses on few-shot learning to suggest new model elements, providing the same unrelated, few-shot examples independently of the current model to be completed. Our investigation centers on the prediction improvements that can be realized by providing semantically similar examples from the model history as context to the LLM for the model completion task.

RQ 4: *What are limitations of using LLMs for model completion in a real-world setting?*

While quantitative results provide insights into the merits of LLMs on model completion, we also want to investigate when and why model completion fails. From simple examples and simulated changes it is hardly possible to make assertions for real-world changes. We therefore take a closer look at a sample set from *real-world changes*. From our observations, we will derive research gaps and hypotheses for future research.

RQ 5: *What insights can be gained when comparing domain-specific fine-tuning to our retrieval-based approach* RAMC*?*

An alternative to retrieval-augmented generation is domain-specific fine-tuning. We explore its viability, considering dataset properties and training specifics (e.g., epochs and base LLM).

B. Datasets

To answer our research questions, we make use of three datasets, balancing internal and external validity. Basic statistics about the datasets are given in Table I.

[8] https://networkx.org [9] https://python.langchain.com
[10] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[11] https://www.trychroma.com

Table I: Basic statistics for the datasets. Model size is measured in terms of the number model elements. Changes include added, deleted, and modified model elements.

Dataset	No. Models	No. Revisions	Avg. Model Size	Avg. No. Changes	Public
INDUSTRY	8	159	11 365	50 340	No
REPAIRVISION	42	3 139	685	70	Yes
SYNTHETIC	24	360	5 402	564	Yes

INDUSTRY Dataset. We have extracted this dataset from a repository of SYSML models in MAGICDRAW[12] for a train control software used by a large product line of trains of our industry partner. The dataset stems from an industry collaboration, where we tackle several challenges related to the management of large industrial software product lines. The model for the train control software comprises several submodels, such as drive and brake control, interior lightning, exterior lightning, sanitary facilities, HVAC, etc. In a preprocessing step, we have removed confidential information (e.g., the models contain requirement owner information and other personal information of involved engineers). The models themselves as well as the average number of changes between revisions in this dataset are large (cf. Table I). The large number of changes originates from many attributes changes, such as renamings, and typically long time periods between two revisions.

The INDUSTRY dataset with its domain-specific and project-specific concepts helps to understand to what extent we can use LLMs for software model completion in a complex, real-world setting. It allows us to assess the effectiveness in navigating the noisy, complex, and often irregular nature of real-world data – a critical aspect often overlooked in existing research.

REPAIRVISION Dataset. The REPAIRVISION [54, 53] dataset is a public dataset[13] of real-world open-source models, containing histories of 21 ECORE repositories, such as UML2 or BPMN2. The REPAIRVISION dataset plays a crucial role in our evaluation in assessing how effectively LLMs can be employed for software model completion in real-world settings. Similar to the INDUSTRY dataset, the serialized change graphs in this dataset can become verbose and noisy and reflect the difficulties of real-world model completion (see Figure 2). Its public availability facilitates reproducibility, comparability, and public accessibility, fundamental aspects that ensure our research can be examined and extended by others.

SYNTHETIC Ecore Dataset. With the first two datasets, we aimed at external validity and a real-world setting. At the same time, we had only little control over potentially influential factors of the dataset impairing internal validity. To obtain a dataset for which we can control several properties of the model repositories, we simulated the evolution of a software model similar to Tinnes et al. [71]: We used a metamodel that resembles a simple component model (as used in modelling system architecture) with components, implementations, ports, connectors, and requirements. Some predefined edit operations have been randomly applied to a revision of a software model to obtain a new revision of the software model. This way we were able to control the number of edit operations that are applied per model revision (i.e., 11, 31, 51, 81) and the number of model revisions in one dataset (i.e., 10 or 20). We furthermore randomly applied perturbations. That is, with a certain probability (i.e., 0%, 50%, 100%), we slightly modified the edit operation by a successive application of an additional edit operation that overlaps with the original edit operation. The repositories in this dataset contain only changes at the type level, that is, we do no include attributes or changes thereof. The SYNTHETIC dataset gives us more control over several properties of a model repository, allowing us to specifically understand how fine-tuning is affected by the properties of the model repositories, this way increasing internal validity.

C. Operationalization

We conduct four experiments, one per research question. For all significance tests, we use a significance level of $\alpha = 0.05$.

Experiment 1 (RQ 1): To answer RQ 1, we preprocess all three datasets from Section V-B and generate a collection with training (75%) and testing samples (25%), more specifically simple change graphs, to ensure a systematic evaluation. We then select[14] between 122–221 samples, depending on the dataset from the testing set and, for each, we select between 1 to 12 few-shot samples from the training set. The reason to choose between 122–221 samples is (1) to obtain a sample set of a manageable size that we can manually analyze and that induces acceptable costs for the LLM usage and (2) to obtain a large enough set to draw conclusions.

First, we analyze the correctness of the generated completions with respect to the ground truth. A simple change graph contains a change that actually occurred in the modeling history. From the change graph, we randomly remove edges to obtain a partial change graph, with the full change graph being the corresponding ground truth. This approach improves over previous methods that involves arbitrarily removing elements from a static snapshot. By focusing on model histories, we create a realistic setting, selecting subsets of changes that have actually occurred in real-world scenarios. We consider different levels of correctness: *Structural correctness* ensures that the graph structure is correct, with properly directed, sourced, and targeted nodes. *Change structure correctness* builds on this by additionally requiring correct types of changes to the model, such as whether elements should be modified, added, or removed. Lastly, *type structure correctness* demands further an exactly correct 'type' and 'changetype'. An illustrative example for these types is given in Figure 2 under 'response'. We automatically check the format, structural correctness, change semantics, and type correctness for all datasets.

For the INDUSTRY dataset, we additionally manually evaluate the generated completions to also check for *semantic* correctness. In our manual analysis of *semantic correctness*, a solution was deemed correct if the LLM's proposed completion matched

[12] MAGICDRAW is a modeling tool commonly used in industries for UML and SysML (system modeling). [13] https://repairvision.github.io/evaluation

[14] The selection procedure is explained in detail on the supplementary website.

the ground truth in meaning and purpose. This check cannot be automated due to the extensive use of natural language in our data and application-specific identifiers (e.g., user-chosen attribute names). For example, in Figure 2, naming a new operation 'getExtension' or 'getExt' is a matter of preference, while their semantic meaning is the same. We addressed potential errors and bias in our manual analysis by having two of the authors independently evaluate the proposed solutions. Any mismatches in their evaluations were discussed, and a consensus was reached on the correct interpretation. For the base LLM, we use GPT-4[15] (version 0613) in a dedicated Azure deployment to complete our prompts.

Experiment 2 (RQ 2): In RQ 2, we investigate whether the correctness (from correct format to semantic correctness) depends on the number of few-shot samples. For the INDUSTRY dataset, we have the information on whether a few-shot sample's change is of a similar class as the test simple change graph. We also investigate how this affects correctness, that is, whether the similarity-based retrieval in RAMC affects the correctness of completions. To this end, we compare semantic sampling with few-shot samples that have been randomly retrieved from the training data. We evaluate this for semantic correctness. For this reason, and also to reduce the LLMs usage costs, we perform this analysis only for the INDUSTRY dataset.

Experiment 3 (RQ 3): To address RQ 3, we selected the publicly available REVISION dataset. This selection not only enhances reproducibility but also allows for comparisons with future methodologies, such that ongoing research advancements can be directly compared to our RAMC and the work by Chaaben et al. [16]. Their approach recommends new classes, their associations, and attributes. Accordingly, the present experiment specifically targets these aspects. We excluded samples that did not fall into these categories, resulting in 51 test examples from the REVISION dataset for comparison. To replicate the approach introduced by Chaaben et al., which we denote as a BASELINE, we use their few-shot examples, serialization of concepts, and incorporate the partial models similarly into the prompt. Further details are available on the supplementary website. We query GPT-3 (text-davinci-002) several times and suggest the most frequently occurring concept.

Experiment 4 (RQ 4): We answer RQ 4 by manually investigating completions that have been generated in the first experiment for the INDUSTRY dataset. We go through all prompt and completion pairs and identify common patterns where the model completion works well or does not, and we aim at interfering causes that led to the results. Since this analysis is time-consuming, we focus on the INDUSTRY dataset – a domain- and project-specific, real-world dataset. We report on the identified strengths and weaknesses of the approach – given this real-world scenario – and point to research gaps and formulate hypotheses for future research and improvements.

Experiment 5 (RQ 5): To investigate whether fine-tuning is a viable alternative to few-shot prompting (see Experiment 1), we fine-tune models from the GPT family of language models on the SYNTHETIC dataset. The reasons why we restrict this analysis to the SYNTHETIC datasets are manifold: The main reason is that we want to understand *how* the performance of the fine-tuning approach depends on various properties of the dataset in a controlled setting. Furthermore, we have a limited budget for this experiment, and fine-tuning is costly. We also control for the number of fine-tuning epochs and the base language model used for the fine-tuning. For every repository of the dataset, we split the data into training set (90%) and testing set (10%), and we use the test set to report on the performance of the completion task. The fine-tuning of the models optimizes the average token accuracy[16]. To compare the retrieval-augmented generation to fine-tuning, we run both for the same test samples. For the few-shot training samples, we also use the same training samples used to fine-tune the language models. We assess the correctness with regard to the ground truth. Due to the unique characteristics of the SYNTHETIC dataset, the ground truth correctness is defined by the graph structure, change structure, and type structure.

D. Results

Experiment 1 (RQ 1): Addressing RQ 1, which explores the extent to which pre-trained LLMs and retrieval-augmented generation can be utilized for software model completion, our findings on the correctness of RAMC are detailed in Table II.

We list the different *levels of correctness* for all datasets. We see that more than 90% of the completions have a correct format and even more than 76% of completions are type correct, that is, completed edges have the right source and target nodes, and type and the types of the source and target node are correct. Even at a semantic level, 62% of the generated completions are correct for the INDUSTRY dataset. For the SYNTHETIC dataset type correctness is equivalent to semantic correctness. Consequently 86% of the results are correct for this dataset.

Table II: Different levels of correctness in percent (%) of the entire test set for all three datasets.

Dataset	Format	Structure	Change Structure	Type Structure	Semantic	Total Count
INDUSTRY	92.62	86.89	78.69	76.23	62.30	122
REPAIRVISION	91.86	84.62	84.16	76.92	–	221
SYNTHETIC	99.05	86.19	86.19	86.19	–	210

Experiment 2 (RQ 2): Regarding the relationship between the number of few-shot samples and correctness, we conducted a (one-sided) Mann-Whitney-U test for the overall

[15] Note that we experimented with several LLMs from the GPT family of models and also observed changes in the specific model's performance over time [17]. At the time of execution, GPT-4 using a small introductory prompt that explains the tasks (see supplementary website) was performing best on a small test set, and we therefore fixed the LLM in RAMC to GPT-4.

[16] At the time of experiment execution, evaluating with any self-defined test metrics was not possible using the fine-tuning APIs provided by OpenAI. This metric is not aware of any specifics of the dataset, and even a single wrong token in a serialization can produce a syntactically wrong serialization, while the token accuracy for the incorrect completion would still be high.

and type/semantic correct distributions over the number few-shot samples. For every dataset, we do not find any significant relationship between the number of few-shot samples and correctness (smallest p-value is 0.2 for the type correctness of the REPAIRVISION dataset). Furthermore, we find that test samples where a similar class of changes is among the few-shot samples perform significantly better than overall correctness ($p = 0.0289$ using a Mann-Whitney-U test, $p = 0.0227$ using a binomial test). Finally, we find that similarity-based retrieval performs significantly better than random retrieval for type correctness ($p < 10^{-9}$, using a binomial test) as well as for semantic correctness ($p < 0.0038$ by a binomial test[17]).

Table III: Different levels of correctness in percent (%) of RAMC and random retrieval on the INDUSTRY dataset.

Approach	Format	Structure	Change Structure	Type Structure	Semantic	Total (Count)
RAMC	92.62	86.89	78.69	76.23	62.30	122
RANDOM	84.43	79.51	52.46	50.00	–	122

Experiment 3 (RQ 3): To obtain a clear picture of the pros and cons of RAMC, BASELINE and random retrieval, we independently report the accuracy of the correct concepts (classes) and the correct association. We further split correct concepts in correct type ("Same Class" in Table IV) and correct name (see supplementary website for details). We perform binomial tests (our random baseline against RAMC and Chaaben et al.) to compare the effectiveness of our approach. We found that, in all cases, RAMC performs significantly better than RANDOM, which, in turn, performs even significantly better than BASELINE (Table IV).

Table IV: Different levels of correctness (%) of RAMC, random retrieval, and BASELINE on the REVISION dataset.

Approach	Same Class	Same Name	Same Concept	Same Assoc.
RAMC	94.1^{**}	96.1^{**}	94.1^{**}	80.4^{*}
RANDOM	78.4	80.4	76.5	68.6
BASELINE	21.6^{**}	9.8^{**}	9.8^{**}	7.8^{**}

(**: $p < 0.01$, *: $p < 0.05$)

Experiment 4 (RQ 4): To better understand when and why the retrieval-augmented generation succeeds or fails when completing software models, we separate our analysis here in two parts—successful completions and unsuccessful ones.

Reoccurring patterns (success): Several of the successful completions follow repeating completion patterns. For example, there is a move refactoring, where a package declaration with type definitions is moved from one package to another package. Since this happened quite often in the past repository histories, the correct new parent package could be deduced, even though this package is not yet part of the incomplete test sample.

Complex refactorings (success): Furthermore, more complex refactorings have also been be completed correctly, for example, a redesign of a whole-part decomposition including packages and SYSML block definitions has been correctly performed. Similarly, we find correctly completed refactorings dealing with inheritance (of port types).

Project-specific concepts (success): Even project-specific concepts, such as a special kind of tagging concept to mark software components as "frozen", are correctly inferred from the few-shot examples or co-changes of components are correctly identified, likewise.

No memorization (success): We also observe correct handling of structure in non-trivial cases. For example, correct combinations of source and target node ids are generated can not be observed in the few-shot examples.

Noise (success): We also observe that the language model is able to infer concepts among noise, that is, unrelated changes. For example, there are correctly completed instances of the "add interface block and type reference" concept where similar few-shot samples are only present with lots of entangled changes.

Regarding unsuccessful cases, we observe two main reasons for failure: incorrect structure and incorrect semantic.

Structural conflicts (failure): For incorrect structure, we find examples where conflicts occur because a node with the same node id is already present. Furthermore, sometimes (correct) model elements or packages are added to the incorrect parent package (in most cases, we see a tendency of the LLM to "flatten" hierarchies).

Structure incorrect (failure): There are several instances where correct edge, source, and target node types are generated but their ids, and consequently the structure, is incorrect.

Semantics wrong b/c copy&paste (failure): One cause for incorrect semantic completions is that parts of few-shot samples are incorrectly copied and pasted. This typically occurs when the LLM lacks sufficient context to generate the correct completion, leading it to mistakenly copy and paste segments from the provided examples.

Semantics wrong b/c unknown evolution/missing context (failure): For example, in the case of functional project-specific evolution, it might be hard to "guess" the right completion without further knowledge, or the semantic retrieval might fail to retrieve instances of the correct change pattern. Interestingly, in some of these cases, the LLM is "guessing well but not perfect" (e.g., added subsystem instead of external subsystem).

Conceivable but unobserved evolution (failure): Another interesting instance of incorrect semantic completion is a completion where a comment (in German) should be removed but instead a comment (in English) has been added. In the project, there were many renamings from German to English and, in this case, a future change has been correctly anticipated.

Experiment 5 (RQ 5): To compare our retrieval-augmented generation-based to fine-tuning, we perform an analysis at the token level, and we also compare the completions on a graph-structural and semantic level. At the token level, we find an average token accuracy of 96.9%, with a minimum of 92.1%, and a maximum of 99.0% on our test data sets (10% test ratio). We can observe strong correlation of the average token accuracy with the number of fine-tuning epochs. Also, larger models

[17] For semantic correctness, we rely on the fact that the number of semantically correct samples is smaller than the number of type correct samples. Thus, we are able to compute an upper bound for the p-value using the type correct random retrieval samples.

perform better with respect to the average token accuracy. Regarding the repository properties, we only find significant negative correlations with the perturbation probability. That is, more diverse repositories are typically harder for the model completion using fine-tuning. Exact numbers are given on our supplementary website. When comparing the distributions of the edges removed in the simple change graph for incorrect and correct completions, we see that the average number of removed edges for the incorrect (i.e., no exact match) completions (5.78) is significantly larger than the average number of removed edges for the correct ones (2.94). Similarly, we find a significant relationship for the distributions of the total simple change graph size (14.89 for the incorrect completions, and 6.39 for correct completions). Accuracies of the comparison of our approach to the fine-tuning approach are given in Table V.

Table V: Different levels of correctness in percent (%) for fine-tuned models compared to the retrieval-based approach in multi-edge software model completion on SYNTHETIC.

Dataset	Method	Correct edge(s)	Exact match
BATCH 1	RAMC	88.52	39.34
	text-ada-001	88.33	56.67
BATCH 2	RAMC	86.00	37.00
	text-curie-001	90.05	64.68

We conducted a Mann-Whitney-U test to compare the performance of retrieval-augmented generation and the fine-tuned text-curie-001 and text-ada-001 models from the GPT-3 family. In terms of producing, at least, one correct edge, neither fine-tuning nor retrieval-augmented generation exhibit statistical significance in outperforming the other. In terms of exact matches, text-ada-001 ($p = 0.0290$) and text-curie-001 ($p < 10^{-7}$) outperform retrieval-augmented generation. Regarding exact matches, the impact of different sampling methods used in fine-tuning and RAMC becomes substantial (algorithms are provided in supplementary website). While RAMC often produces more edges than required, the sampling procedure used with the fine-tuning models is more conservative.

E. Discussion

Overall, we find that both RAMC and fine-tuning of LLMs are promising approaches for model completion, and the general inference capabilities of LLMs are useful, can handle noisy contexts, and provide real-time capabilities. We will next discuss the results, outline hypotheses for potential future research, and describe threats to validity in Section V-F.

RQ 1: In the Experiment 1, we observed promising correctness values across all datasets. Not only are more than 90% of completions correct w.r.t. the serialization format, but we also find *more than 62% of semantically correct completions in the real-world industrial setting*. This indicates that retrieval-augmented generation is a promising technique for model completion. Token processing times fall within the millisecond range, and time required for semantic retrieval is negligible, even for larger models. The approach's real-time capability is significant given the stepwise model completion use case.

RQ 2: In the retrieval-augmented generation setting, we do not find any significant relationship between the number of few-shot samples and correctness. We find that similarity-based retrieval boosts the correctness of the approach and that it significantly performs better if a similar relevant change—following a similar pattern—is available in the context. It also worthwhile mentioning that real-world datasets are typically biased with respect to the change pattern, and semantic retrieval can avoid sampling from large but irrelevant change pattern.

RQ 3: We have observed that, in all instances where new elements with associations are recommended, RAMC consistently outperforms random retrieval and Chaaben et al. [16]. These results reinforce our findings from RQ 1, namely that leveraging LLMs with retrieval-augmented generation represents a viable approach for model completion.

RQ 4: We have seen that our approach can be used to provide completions that are correct to a large extent for simple reoccurring patterns but also more complex refactorings. Even project-specific concepts can be deduced from few-shot examples. In many cases, generated edges are also structurally correct. The general inference capabilities of LLMs are useful, for example, in dealing with concepts for which there are few or no similar examples. Furthermore, also with noise retrieval-augmented generation often provides correct completions. Regarding usefulness of the completions, our manual analysis reveals that many of the completions appear useful for the modeler. For example, RAMC was able to perform a translation of several German comments to English, because the engineering language of the project has been changed. Furthermore, RAMC was able to complete project-specific refactorings.For a further investigation of these observations, we formulate the following hypothesis.

Hypothesis 1: LLMs and retrieval-augmented generation are able to handle noisy training examples, leverage (domain) knowledge from pre-training, adapt to project-specific concepts, and provide useful software model completions.

We found completions that are incorrect from a structural viewpoint as well as incorrect from a semantic viewpoint. As for structurally incorrect completions, we identified cases where existing node ids are incorrectly reused, where incorrect (containment) hierarchies would have been created, or where completed edges are correct from a type perspective but do not connect the right nodes. It is worth further investigating how these structural deficiencies could be overcome, in particular, given that LLMs are designed for sequential input, not for graph inputs. This leaves us with the following hypothesis.

Hypothesis 2: Conceivable remedies for the structural deficiencies include fine-tuning of LLMs, combining graph neural networks – designed for graph-like input – with LLMs, providing multiple different graph serialization orders, or a positional encoding that reflects the graph-like nature of the simple change graph serializations.

Regarding semantics, we found incorrect completions that were related to a lack of (domain) knowledge in the pre-trained model or the few-shot examples, respectively. For example, we found cases of functional evolution where the language model is missing (domain) knowledge or requirements, or cases of a refactoring without any relevant few-shot sample. We further identified cases where a conceivable completion has been generated but was not the one from the ground truth.

Hypothesis 3: Conceivable remedies for the semantic deficiencies include strategies to further fuse the approach with context knowledge (e.g., fine-tuning, providing requirements, or task context in the prompt, leveraging other project data in repositories etc.). Furthermore, providing a list of recommendations may cure some identified deficiencies.

RQ 5: We found that a more fine-tuning epochs are beneficial for the average token accuracy. More diverse repositories increase the difficulty for the software model completion. The larger the simple change graph and the more edges we omit for the completion, the higher the probability of an incorrect completion. The reason that fine-tuning has a higher exact match accuracy is more due to the edge sampling algorithm than to the method itself: When analyzing the percentage of correct edges, it becomes clear that we cannot conclude that one approach outperforms the other. Instead, we hypothesize a strong dependency on the edge sampling procedure, which deserves further investigation. While the retrieval-augmented generation often generated more edges than necessary, the sampling procedure used with the fine-tuned models from the GPT-3 family takes a more conservative approach, prioritizing the generation of edges with high confidence.

Comparison to code completion. Note that LLMs for source code completions show similar results to our findings in Experiment 1 and 5, ranging from 29% for perfect prediction of entire code blocks to 69% for a few tokens in a single code statement [20]. Drawing a direct comparison between code and model completion is not straightforward, though.

F. Threats to Validity

With respect to construct validity, we made several design choices that may not be able to leverage the entire potential of LLMs for software model completion, including our definition of simple change graphs, the serialization of the simple change graph, the strategy of how to provide domain knowledge to the language model, and the choice of the base LLM.

To increase internal validity, we incorporated the SYNTHETIC Ecore Dataset into our experiments, controlling for properties of software model repositories. Still, we were not always able to completely isolate every factor in our experiments. For example, fine-tuning and few-shot learning use different edge samplings. This is due to the API that we used to access the language models. In future research, an ablation study for the design choices in the algorithms shall be performed. To address the potential variability that LLMs may exhibit, we checked and confirmed that the completions were stable.

Regarding external validity, we included two real-world datasets (REPAIRVISION and INDUSTRY), and we study real-world change scenarios from the observed history in these repositories. We have chosen our test samples to be small enough to perform manual semantic analysis, but large enough to draw conclusions. To minimize costly manual checks, our semantic analysis was confined to our most challenging dataset, the INDUSTRY dataset. Extending the analysis to the other datasets would enhance validity. However, we are confident that, having analyzed hundreds of samples, we have struck a reasonable compromise. We are therefore certain that our results have an acceptable degree of generalizability for the current state of research. In any case, user studies shall investigate the usefulness of our completions in practice. Investigating merits of LLMs for model completion is an emerging topic, and many questions are open. Still, our results set a lower bound for the potential of LLMs in this area, with promising results, insights, and hypotheses for further research.

VI. CONCLUSION

We presented and investigated an approach to software model completion based on retrieval-augmented generation, RAMC, and compared it to fine-tuning during our evaluation. Our experiments on a simulated, a public, open source ECORE, and an industrial SYSML dataset for a train control software product line show that, indeed, LLMs are a promising technology for software model completion. The real-time capability of our approach is especially beneficial for stepwise model completion, highlighting its practical utility. We achieved a semantic correctness in a real-world industry setting of 62.30%, which is comparable to earlier results with LLMs for source code completion. Further investigation revealed that similarity-based retrieval significantly enhances the correctness of model completions and that fine-tuning is a viable alternative to retrieval-augmented generation. All in all, the general inference capabilities of LLMs are beneficial, particularly in dealing with concepts for which only scarce or even no analogous examples are provided. We have identified concrete causes for the technology to fail and formulated corresponding hypotheses for future research. Of utmost importance for future research is to compare technology, such as graph neural networks, that has been designed for processing graph-like data (e.g., our simple change graphs), especially for structural aspects of software model completion. Also, marrying approaches that are strong for structural aspects, and LLMs, that are typically strong for semantic aspects of model completion is worth further investigation.

VII. DATA AVAILABILITY

We provided all data (excluding the INDUSTRY dataset) and the Python code of our approach on a supplementary website. We cannot include the INDUSTRY dataset, because it contains sensitive data, including intellectual property of products on the market. We provide R scripts and Jupyter Notebooks to replicate our statistical evaluation.

References

[1] Bhisma Adhikari, Eric J Rapos, and Matthew Stephan. Simima: a virtual simulink intelligent modeling assistant: Simulink intelligent modeling assistance through machine learning and model clones. *Software and Systems Modeling*, pages 1–28, 2023.

[2] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. Domore– a recommender system for domain modeling. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development*, volume 1, pages 71–82. Setúbal: SciTePress, 2018.

[3] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. Automated recommendation of related model elements for domain models. In *Model-Driven Engineering and Software Development: 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers 6*, pages 134–158. Springer, 2019.

[4] Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fahmideh, Mst Shamima Aktar, and Tommi Mikkonen. Towards human-bot collaborative software architecting with chatgpt. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 279–285, 2023.

[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint*, 2021.

[6] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 1–5, 2022.

[7] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. Recommender systems in model-driven engineering. *Software and System Modelling*, 21(1):249–280, 2022.

[8] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 121–135. Springer, 2010.

[9] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and Systems Modeling*, 11(2):227–250, 2012.

[10] Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[11] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19:5–13, 2020.

[12] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. An NLP-based architecture for the autocompletion of partial domain models. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, pages 91–106. Springer, 2021.

[13] Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard. Cognifying model-driven software engineering. In *Software Technologies: Applications and Foundations*, pages 154–160. Springer, 2018.

[14] 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*, pages 1–13, 2023.

[15] Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the International Conference on Research and Development in Information Retrieval*, page 335–336, New York, NY, USA, 1998. ACM.

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

[17] Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time? *arXiv*, 2023.

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint*, 2021.

[19] Tsigkanos Christos, Rani Pooja, Müller Sebastian, and Kehrer Timo. Large language models: the next frontier for variable discovery within metamorphic testing? In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2023.

[20] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of transformer models for code completion. *Transactions on Software Engineering*, 48(12):4818–4837, 2022.

[21] James B Dabney and Thomas L Harman. *Mastering simulink*, volume 230. Pearson/Prentice Hall Upper Saddle River, 2004.

[22] Carlos Diego Nascimento Damasceno and Daniel Strüber. Quality guidelines for research artifacts in model-driven engineering. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 285–296. IEEE, 2021.

[23] Shuiguang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. A recommendation system to facilitate business process modeling. *IEEE transactions on cybernetics*, 47(6):1380–1394, 2016.

[24] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Alfonso Pierantonio. Memorec: a recommender system for assisting modelers in specifying metamodels. *Software and Systems Modeling*, 22(1):203–223, 2023.

[25] Juri Di Rocco, Claudio Di Sipio, Phuong T Nguyen, Davide Di Ruscio, and Alfonso Pierantonio. Finding with nemo: a recommender system to forecast the next modeling operations. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pages 154–164, 2022.

[26] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong T Nguyen. Morgan: a modeling recommender system based on graph kernel. *Software and Systems Modeling*, pages 1–23, 2023.

[27] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *International Conference on Graph Transformation (ICGT)*, pages 161–177. Springer, 2004.

[28] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. An uml class recommender system for software design. In *Proceedings of the International Conference of Computer Systems and Applications (AICCSA)*, pages 1–8. IEEE, 2016.

[29] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint*, 2020.

[30] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Prentice Hall, 1995.

[31] Anderson Gomes and Paulo Henrique M Maia. Dome: An architecture for domain model evolution at runtime using nlp. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 186–195, 2023.

[32] Lars Heinemann. Facilitating reuse in model-based development with context-dependent model element recommendations. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 16–20. IEEE, 2012.

[33] Ningyuan Teresa Huang and Soledad Villar. A short tutorial on the weisfeiler-lehman test and its variants. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8533–8537. IEEE, 2021.

[34] IEC. Programmable controllers - part 3: Programming languages. Technical report, DIN/EN/IEC 61131, 2014.

[35] Ludovico Iovino, Angela Barriga Rodriguez, Adrian Rutle, and Rogardt Heldal. Model repair with quality-based reinforcement learning. 2020.

[36] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. Large language models and simple, stupid bugs. *arXiv*, 2023.

[37] Timo Kehrer. *Calculation and Propagation of Model Changes based on User-Level Edit Operations: A Foundation for Version and Variant Management in Model-Driven Engineering*. PhD thesis, University of Siegen, 2015.

[38] Timo Kehrer, Abdullah M Alshanqiti, and Reiko Heckel. Automatic inference of rule-based specifications of complex in-place model transformations. In *Proceedings of the International Conference on Model Transformations (ICMT)*, pages 92–107. Springer, 2017.

[39] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. Automatically deriving the specification of model editing operations from meta-models. In *Proceedings of the International Conference on Model Transformations (ICMT)*, volume 9765, pages 173–188, 2016.

[40] Stefan Kögel. Recommender system for model driven software development. In *Proceedings of the 2017 11th Joint Meeting on Foundations of*

Software Engineering, pages 1026–1029, 2017.

[41] Stefan Kögel, Raffaela Groner, and Matthias Tichy. Automatic change recommendation of models and meta models based on change histories. In *ME@ MoDELS*, pages 14–19, 2016.

[42] Philippe B Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

[43] Tobias Kuschke and Patrick Mäder. Rapmod—in situ auto-completion for graphical models. In *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*, pages 303–304. IEEE, 2017.

[44] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. Recommending auto-completions for software modeling activities. In *International conference on model driven engineering languages and systems*, pages 170–186. Springer, 2013.

[45] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[46] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, Shuiguang Deng, Yuyu Yin, and Zhaohui Wu. An efficient recommendation method for improving business process modeling. *IEEE Transactions on Industrial Informatics*, 10(1):502–513, 2013.

[47] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. Modelset: a dataset for machine learning in model-driven engineering. *Software and Systems Modeling*, pages 1–20, 2022.

[48] Steffen Mazanek and Mark Minas. Business process models as a showcase for syntax-based assistance in diagram editors. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, pages 322–336, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[49] Steffen Mazanek and Mark Minas. Generating correctness-preserving editing operations for diagram editors. *Electronic Communication of the European Association of Software Science and Technology*, 18, 2009.

[50] Patrick Mäder, Tobias Kuschke, and Mario Janke. Reactive auto-completion of modeling activities. *Transactions on Software Engineering*, 47(7):1431–1451, 2021.

[51] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. Rule-based repair of emf models: An automated interactive approach. In *Theory and Practice of Model Transformation: 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings 10*, pages 171–181. Springer, 2017.

[52] Patrick Neubauer, Robert Bill, Tanja Mayerhofer, and Manuel Wimmer. Automated generation of consistency-achieving model editors. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 127–137. IEEE, 2017.

[53] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. History-based model repair recommendations. *Transactions of Software Engineering Methodology (TOSEM)*, 30(2), 2021.

[54] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. ReVision: A tool for history-based model repair recommendations. In *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*, pages 105–108. ACM, 2018.

[55] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. Technical report, Object Management Group, June 2013.

[56] OMG. Unified modeling language (UML) version 2.5.1. Standard, Object Management Group, December 2017.

[57] OMG. Omg sysml v. 1.6. Standard, Object Management Group, December 2019.

[58] Michael Polanyi. *Personal Knowledge: Towards a Post Critical Philosophy*. University of Chicago Press, 1958.

[59] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint*, 2019.

[60] Gregorio Robles, Michel RV Chaudron, Rodi Jolak, and Regina Hebig. A reflection on the impact of model mining from github. *Information and Software Technology*, 164:107317, 2023.

[61] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems and Structures*, 43:139–155, 2015.

[62] Hazem Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 2022.

[63] Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the SiDiff framework. In *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*, pages 947–948. ACM/IEEE, 2008.

[64] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.

[65] Dominik Sobania, Martin Briesch, and Franz Rothlauf. Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. *CoRR*, abs/2111.07875, 2021.

[66] Friedrich Steimann and Bastian Ulke. Generic model assist. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 18–34. Springer Berlin Heidelberg, 2013.

[67] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[68] Matthew Stephan. Towards a cognizant virtual software modeling assistant using model clones. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 21–24. IEEE, 2019.

[69] Matthew Stephan and James R Cordy. A survey of model comparison approaches and applications. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 265–277, 2013.

[70] Christof Tinnes, Timo Kehrer, Mitchell Joblin, Uwe Hohenstein, Andreas Biesdorf, and Sven Apel. Mining domain-specific edit operations from model repositories with applications to semantic lifting of model differences and change profiling. *Automated Software Engineering*, 30(2):17, 2023.

[71] Christof Tinnes, Timo Kehrer, Joblin. Mitchell, Uwe Hohenstein, Andreas Biesdorf, and Sven Apel. Learning domain-specific edit operations from model repositories with frequent subgraph mining. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, 2021.

[72] Arie Van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. *Technical Report Series TUD-SERG-2007-006.*, 2007.

[73] Dániel Varró. Model transformation by example. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 410–424. Springer, 2006.

[74] 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.

[75] 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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 382–394, 2022.

[76] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. Recommending metamodel concepts during modeling activities with pre-trained language models. *Software and Systems Modeling*, 21(3):1071–1089, 2022.

[77] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the International Symposium on Machine Programming*, pages 1–10, 2022.

[78] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the International Symposium on Machine Programming*, page 1–10. ACM, 2022.

[79] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J Letsholo, Muideen A Ajagbe, Erol-Valeriu Chioasca, and Riza T Batista-Navarro. Natural language processing for requirements engineering: a systematic mapping study. *ACM Computing Surveys (CSUR)*, 54(3):1–41, 2021.