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

***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 [60], software documentation [41, 55], system architecture [56], simulation [20], and industrial automation [33]. 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 [71].

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 [22, 70]. 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 [29] are widely used and are not part of UML [55], 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 [17, 76, 5, 28, 6, 74].

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 [12].

Problem Statement. Notably, for a few domain-specific languages, rule-based approaches exist that use pre-defined edit operations or patterns for model completion [42, 43, 31, 63]. 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 [57]. 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 [72, 37]), 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 [70], and mining approaches lack abstraction capabilities [70].

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 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 [15], 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).

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

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 [15] 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 [15]. 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).

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, 42, 43, 49, 45, 22, 47]. 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 [42, 43, 49] or business process modelling [22, 45].

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, 67, 23, 27] or new libraries for use [31]. MemoRec [23] and MORGAN [25] 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 [22, 45]. Kögel et al. [40, 39] 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 [38] or pattern mining approaches [70, 69, 44] 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 [65]. Without knowledge about higher-level semantics, these approaches are more comparable to the application of a catalog of minimal consistency-preserving edit operations [38].

Regarding the application of natural language processing (NLP) [11] and language models [18, 75], Burgueño et al. [11] propose an NLP-based system using word embedding similarity to recommend domain concepts. Weyssow et al. [75] use a transformer-based language model to recommend metamodel concepts without generating full model completions. Di Rocco et al. [24] 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. [30] 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 [17, 35, 19, 64]. 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. [15], 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. [13] 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

[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.

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 [51, 34, 50, 48, 53, 67]. ReVision [53] 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 [8, 26].

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. [8]. 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.

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 [68], e.g., EMFCompare [9] or SiDiff [62]). 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* $G_{\Delta mn}$ [53]. 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 $\mathcal{T} \overset{\text{def}}{=} \mathcal{M} \times \mathcal{M}$ 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 $\mathcal{E} \overset{\text{def}}{=} \mathcal{T}/\sim$. An edit operation is therefore a set of model transformations that have the same simple change graph.

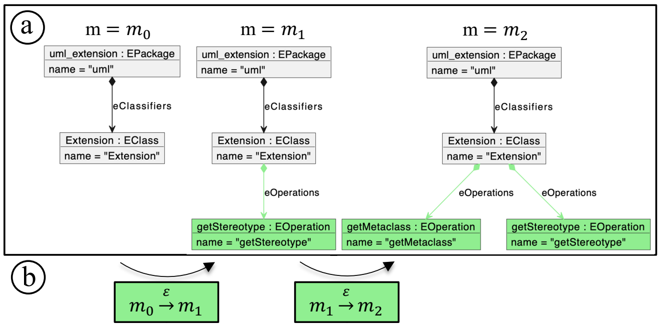


Figure 1: Visual presentation of our example taken from the RepairVision dataset: ⓐ Evolutionary View: User performs edit operations one by one. ⓑ Evolution can be performed by a user or by using a completion approach.

[3] https://anonymous.4open.science/r/icse_model_completion-CBC4/MC_Preprint_and_Appendix.pdf

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 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 [8, 36, 70]. 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 $C\colon \mathcal{T} \to \mathcal{T}$ 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.

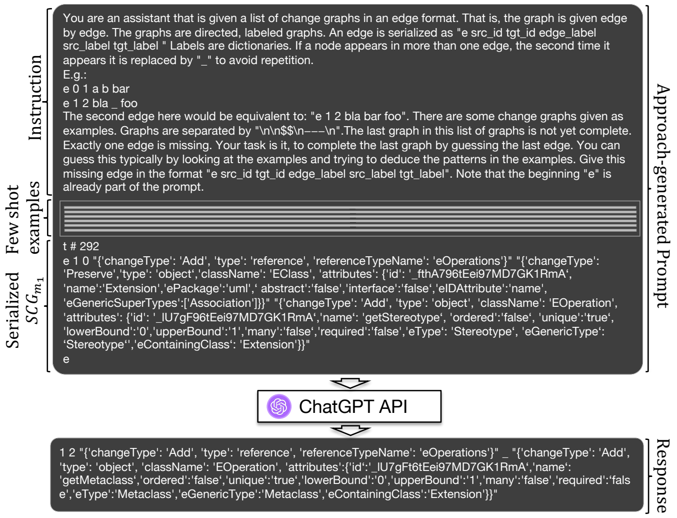
The probability distribution is typically derived from a *corpus* of documents, containing (some of) the tokens. With the success of transformer architecture [73], LLMs have become quite popular now and are used in plenty of domains including software engineering [61, 78, 77]. There are two tactics available 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 [55], 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 [55], 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).



```
You are an assistant that is given a list of change graphs in an edge format. That is, the graph is given edge
by edge. The graphs are directed, labeled graphs. An edge is serialized as "e src_id tgt_id edge_label
src_label tgt_label " Labels are dictionaries. If a node appears in more than one edge, the second time it
appears it is replaced by "_" to avoid repetition.
E.g.:
e 0 1 a b bar
e 1 2 bla _ foo
The second edge here would be equivalent to: "e 1 2 bla bar foo". There are some change graphs given as
examples. Graphs are separated by "\n\n$$\n–––\n".The last graph in this list of graphs is not yet complete.
Exactly one edge is missing. Your task is it, to complete the last graph by guessing the last edge. You can
guess this typically by looking at the examples and trying to deduce the patterns in the examples. Give this
missing edge in the format "e src_id tgt_id edge_label src_label tgt_label". Note that the beginning "e" is
already part of the prompt.
t # 292
e 1 0 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eOperations'}" "{'changeType':
'Preserve','type': 'object‘,'className': 'EClass', 'attributes': {'id': '_fthA796tEei97MD7GK1RmA‘,
'name':'Extension','ePackage':'uml',‘ abstract':'false','interface':'false‘,'eIDAttribute':'name',
'eGenericSuperTypes':['Association']}}" "{'changeType': 'Add', 'type': 'object', 'className': 'EOperation',
'attributes': {'id': '_lU7gF96tEei97MD7GK1RmA‘,'name‘: 'getStereotype‘, 'ordered':'false‘, 'unique':'true‘,
'lowerBound':'0','upperBound':'1','many':'false‘,'required':'false','eType‘: 'Stereotype‘, 'eGenericType':
‘Stereotype‘','eContainingClass‘: 'Extension'}}"
e
```

```
1 2 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eOperations'}" _ "{'changeType': 'Add',
'type': 'object', 'className': 'EOperation', 'attributes':{'id‘:'_lU7gFt6tEei97MD7GK1RmA‘,'name‘:
'getMetaclass‘,'ordered':'false‘,'unique‘:'true‘,'lowerBound':'0','upperBound':'1','many':'false','required':‘fals
e','eType':'Metaclass','eGenericType':'Metaclass','eContainingClass':'Extension'}}"
```

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.

B. Overview and Design Choices

Utilizing LLMs for software model completion gives rise to several challenges addressed by RAMC: how to provide context,

[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 [54], is itself a model according to its meta-metamodel, ECORE, and therefore covered by the present work.

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 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.

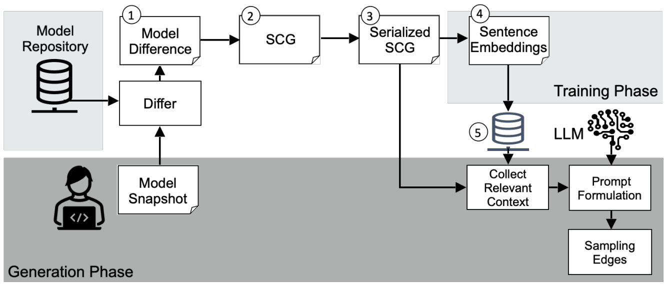


Figure 3: Overview of RaMc.

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 [58]. 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 [14], explained in detail on the supplementary website. As few-shot samples, we select 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

[6] software models can become huge compared to the limited number of tokens that can be given to a LLM.

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

(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 model completion candidates from the LLM by using a beam search, or, instructing the model to generate several new edges. The edge sampling algorithms are given in detail 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 [66], using SiDiff [62] 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

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

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

We evaluate the accuracy of our proposed approach, RaMc, by comparing it to the closely related work of Chaaben et al. [15], 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 2.5: *How does* RaMc *compare against the state of the art (Chaaben et al. [15])?*

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

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).

RQ 4: *What insights can be gained when comparing domain-specific fine-tuning to our retrieval-based approach* RaMc*?*

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.

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

[8] https://networkx.org
[9] https://python.langchain.com/
[10] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[11] https://www.trychroma.com
[12] MagicDraw is a modeling tool commonly used in industries for UML and SysML (system modeling).

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

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 [53, 52] 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. [70]: 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 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

[13] https://repairvision.github.io/evaluation/.

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

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 2.5 (RQ 2.5): To address RQ 2.5, 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. [15]. 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 3 (RQ 3): We answer RQ 3 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 4 (RQ 4): 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 RQ1, 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.

			Change	Type		Total
Dataset	Format	Structure	Structure	Structure	Semantic	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 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

[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 [16]. 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.

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 2.5 (RQ 2.5): 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 3 (RQ 3): 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 4 (RQ 4): 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 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

[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.

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 regarding the individual research questions, which includes outlining hypotheses for potential future research directions, followed by a discussion of threats to validity in Section V-F.

RQ 1: In the first experiment, we have seen a promising number of correct completions 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 for a real-world industrial setting*. This indicates that LLMs with retrieval-augmented generation are a promising technique for model completion. Note that token processing times fall within the millisecond range, and the 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 improves the correctness of the approach and that it significantly performs better if a similar example is available in the context. We conjecture that similarity-based retrieval is capable of retrieving relevant changes that follow a similar pattern and therefore boosts the completion capabilities. 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 2.5: We have observed that, in all instances where new elements with associations are recommended, RAMC consistently outperforms random retrieval and Chaaben et al. [15]. These results reinforce our findings from RQ 1, namely that leveraging LLMs with retrieval-augmented generation represents a viable approach for model completion.

RQ 3: 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 entangled changes (i.e., 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 which MAGICDRAW does not provide direct support. 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 4: 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 4, ranging from 29% for perfect prediction of entire code blocks to 69% for a few tokens in a single code statement [19]. 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.

With respect to external validity, we included two real-world datasets (REPAIRVISION and INDUSTRY), and we study real-world change scenarios, taken from the observed history in these real-world repositories. We have chosen our test samples small enough to be still able to conduct a manual semantic analysis but large enough for drawing 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 a acceptable degree of generalizability for the current state of research. Investigating merits of LLMs for model completion is an emerging topic, so many questions are open. Still, our results set a lower bound for the potential of LLMs in this area, with promising results, insights, and hypothesis for further research.

VI. CONCLUSION

We investigated the merits of using LLMs for software model completion. 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. We found that larger LLMs and more epochs contribute to better performance for fine-tuning and that more diverse repositories increase the difficulty for the software model completion task. 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. Anyway, 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) an the Python code of our approach on a supplementary website. We are not allowed to 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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

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

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

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

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

[59] 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.

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

[61] 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.

[62] 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.

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

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

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

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[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*, page 1–10. ACM, 2022.

[78] 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.

Appendix

A. Sampling of Experiment Samples

Every dataset consists of several projects (e.g., submodels in the case of the Industry dataset), and we ensure that they are represented with the same distribution in our samples. Furthermore, we ensure that every project is included, at least, once, even if it is very small. This leaves us with 210 samples for the Synthetic dataset, 221 samples for the RepairVision dataset, and 200 samples for the Industry dataset. For every project in the dataset we used a diversity sampling strategy to obtain a diverse range of samples. Note that, we used the same diversity sampling strategy (described in Section G) as during the retrieval of "few-shot samples" but applied to the full range of test set examples. For the Industry dataset, since we perform a manual analysis of semantic correctness there, we further reduced the dataset. We could have just randomly (or with the procedure above) down-sampled, for example, to 100 samples. Instead, since we generally observe a strong homogeneity in the real-world datasets (i.e., many repeating patterns in Industry and RepairVision datasets), we wanted to ensure that we do not decrease the heterogeneity too much. We therefore decided to further select samples from industry with a more controlled procedure:

We further examine the prompt and completion pairs to classify the changes into semantic clusters that we defined. We skimmed the dataset once and came up with a list of change patterns, for example, "interface added between components". We agreed on a final list of patterns. We then recorded whether the training samples contain a change that falls into the same class. We then only included samples that are unique according to the number of training examples in the prompt, the class of the change that we assigned, and whether there is a similar change in the training samples or not (which has also been decided with the help of the patterns we defined for the changes). This leaves 122 samples for the model completion task on the Industry dataset.

B. Formalization

In addition to the terminology from our formalization in Section III, we will define further operators and then formalize our approach RaMc on top: We define a serialization operator $s\colon \mathcal{G} \to \Sigma^*$, where Σ^* is the set of all strings. This operator s takes a (partial) simple change graph $g \in \mathcal{G}$ and returns a serialization for this graph (the detailed serialization is given in the supplementary website). Furthermore, we define $\mathrm{r}\colon \Sigma^* \to {\Sigma^*}^k$, which, given a (partial) serialized simple change graph $s(g) \in \Sigma^*$, retrieves k "similar" serialized simple change graphs from a (vector) database. We define the prompt operator $\mathrm{prompt}\colon \Sigma^* \times {\Sigma^*}^k \to \Sigma^*$, which, given a tuple $(s(g), \mathrm{r}(s(g))$ of the (partial) simple change graph and the retrieved similar serialized simple change graphs, constructs the final prompt (described in detail in the supplementary website). Given the prompt instruction $i := \mathrm{prompt}((\mathrm{s(g)}, \mathrm{r(s(g)}))$, we can generate serialized completed simple change graph candidates by sampling tokens with a LLM. That is, we sample tokens ω_{j+1} from $\mathbb{P}(\omega_{j+1} | i\, \omega_1 \ldots \omega_j)$, until the entropy becomes too large or a complete edge serialization has been sampled. A large entropy of the language model token probabilities can be seen as an indicator[18] for a high uncertainty of further tokens. We denote this candidate generation by

$$\begin{aligned} \mathrm{cg}_{\mathrm{LLM}}\colon \Sigma^* &\to \Sigma^* \\ \mathrm{cg}_{\mathrm{LLM}}(i) &\mapsto s(g)\, \omega_1 \ldots \omega_J, \end{aligned}$$

where J is the total number of sampled tokens. By $s(g)\, \omega_1$, we denote the string concatenation of $s(g)$ and ω_1, and likewise for the rest of this expression. Finally, we parse $s(g)\, \omega_1 \ldots \omega_J$ as a graph (if possible), and interpret it as a completed simple change graph, which represents a model transformation γ. It can happen (although it rarly happens in practise as can be seen in the evaluation section of this paper) that the completed string does not represent our simple change graph serialization format. In this case, we record this failure and consider the model completion as failed. Identifying the parsed graph with the corresponding edit operation, we denote this parsing operator by $s^{-1}\colon \Sigma^* \to \mathcal{E}$. With this notation, the entire model completion approach, RaMc, can be formalized by

$$\begin{aligned} \mathrm{C}_{\mathrm{RaMc}}\colon \mathcal{T} &\to \mathcal{T} \\ (m_1, m_2) &\mapsto \pi(m_1, s^{-1} \circ \mathrm{cg}_{\mathrm{LLM}} \circ \mathrm{prompt} \\ &\qquad \circ \mathrm{id} \times \mathrm{r} \circ s \circ \mathrm{SCG}(m_1, m_2)). \end{aligned}$$

C. Details for Baseline Comparison

1) Evaluated Datasets

A proper level where we can compare several approaches to model completion would be to move to a meta-meta level (following the MOF). This allows us to compare Chaaben et al.'s approach that works for a subset of UML class diagrams and a subset of activity diagrams against our approach that works directly on the abstract syntax graph of the models. By interpreting the abstract syntax of our models as class diagrams, we were able to compare (part of) their approach to ours. In our dataset, we had already classified the changes and the changes that are of interest for the recommendation of new concepts, corresponded to a class of changes we called "Add_node" in our samples. We selected these changes (which left us with 51 samples for the revision dataset)

2) Evaluation Method

In our comparison, we did focus on concept recommendation and association recommendation. Attribute recommendation in class diagrams is quite comparable to concept recommendation. Indeed, in ECore deciding between a reference to another concept or an attribute of an EClass is more like a design choice and both are considered to be a EStructuralFeature.

We mapped the examples we observed to corresponding concept recommendations. E.g., when an EClass with name User is added via a containment to an EClass with the name Software, we used the tuple [EClass.Software, EClass.User] in Chaaben et al.'s approach. Depeding on the concrete ECore

[18] assuming a well-calibrated LLM

concept, we replaced *name* by the corresponding identifier (e.g., key).

To get a clearer picture of the pros and cons of the approaches, we decided to report independently the accuracy of the correct concepts being recommended, the accuracy of correct association being recommended, and we further split in correct type (e.g., EClass in the example above) and correct name (e.g., Software, or User in the example above).

3) Replication of their approach

To use the approach introduced by Chaaben et al. [15], as the BASELINE some small adjustments needed to be done, to make a fair comparison possible.

We utilized the same few-shot examples, following the premise that these could originate from unrelated models. We could have decided to choose the few-shot samples from the dataset, similar to our approach. Anyway, since we consider this a crucial difference of the approach, we used the few-shot samples exactly as in the implementation of their approach. The few-shot samples were loaded from a file that we used in the re-implementation of their approach. We build on their serialization of concepts and enhance the queries by additionally incorporating the partial domain model in a similar manner. We select between one to four pairs of related concepts, enclosing the concept names in brackets. Then, they query GPT-3 (text-davinci-002) multiple times to suggest the most frequently occurring concepts. We use the same temperature setting of 0.7 and a maximum token length of 20 tokens, which is sufficient in our REVISION dataset to suggest at least one new pair of concepts. Excess tokens are removed. We also considered upgrading the model used for the BASELINE to GPT-4. This transition would necessitate additional modifications in their approach. When utilizing GPT-4 with the existing prompts, the model begins generating natural language text that is not directly related to the specific use case. This deviation occurs because the text-davinci-002 model is not designed for chat-like interactions, unlike ChatGPT and GPT-4. Consequently, changing the model would require a redesign of their prompts to align with the capabilities of these models.

The authors employed a sampling strategy coupled with a ranking method, so we similarly query GPT-3 multiple times using a variety of prompts, each consisting of the same few-shot examples but with different queries that incorporate a subsets of model elements from the partial model. In their implementation, the authors sampled (random or all) pairs of concepts from the model at hand. This method does not facilitate effective real-time responses, particularly for larger software models, thus making it impractical for scenarios like those encountered in our REVISION dataset (about 685 queries on average, see I). Since we had simple change graphs at hand, we decided to sample the edges/associations from these simple change graphs, ensuring that the number of elements in the partial model for each query ranges between one and four elements.

Furthermore, when suggesting new concepts, the paper considers both elements of each pair as new concepts. Unfortunately, one of these elements is usually already present in the partial model. This results in existing model elements being ranked at the top, rather than new concepts, leading to poorer outcomes. We have improved upon this by filtering out classes that already exist in the model.

It is noteworthy that several recommender systems make a list of k recommendations. We did intentionally decide to set k equal to one, since in a real world scenario (compare to GitHub Copilot), one would typically not come up with a list, but directly integrate one recommendation in the IDE. To ensure a fair comparison, we focus on the single most frequently occurring completion.

D. Simple Change Graphs and Their Labels

In Definition III.4, we defined simple change graphs as subgraphs of a difference graph, which is a labeled graph. In this section, we explain in more detail, how we derive the labels from the models and the change graph.

We assume a simplified metamodel, in which we have classes that carry a name, that is, the type of a model element. A class has attributes that have a attribute name and attribute value, and references that have a reference type.

For a given model, we then use this simplified metamodel to derive a labeled graph (cf. Definition III.2): we map objects (i.e., instances of a class) to a node of the labeled graph and instances of references to edges. By this, we ensure that our graph representation is structurally equivalent to an abstract syntax graph of the model (difference). Nodes and edges in the graph carry a label. For nodes, this label is a JSON representation of the object. It has a attribute type with its value equal to the name of the class the object is an instance of. It also contains all attributes with their values for the given object(assuming we can serialize the attribute). More concretely, the attributes are a contained as a nested JSON inside the node label JSON with JSON attribute names equal to the attribute name and JSON value given by the attribute value. Finally, for the edge labels, we use a JSON that has a JSON attribute type with value equal to the reference type.

Next, for the difference graph, we simply add to each node and each edge another JSON attribute changeType, with value equal to Add, Preserve, or Remove, depending on the change type in the difference graph. For modified attributes, we add another node attached to the necessarily preserved object with a JSON label indicating the attribute value *before* and *after* the change. Since a simple change graph is a subgraph of the difference graph, this construction also defines the labels of the simple change graph.

Note that in some cases (e.g., to check for type correctness), we can simply remove attribute information from our labels, thus obtaining a graph that has only information about the type structure. We use this graph, for example, to check for type correctness of model completions. Furthermore, this graph without attribute information can also be helpful for other use cases, where we are only interested in the type structure, for example, in change pattern mining use cases, or if we want to define a reusable template for edit operations.

Now that we know how to construct a labeled graph for a given model difference, we will next see how we serialize these labeled graphs.

E. Serialization Format

In this section, we explain our serialization format for graphs, called EdgeList, which will be part of the prompt being send to the LLM (cf. Figure 2).

The serialization of a graph starts with a header line (indicating an id of the graph).

```
t # <graph_id>
```

After the header, all edges of the graph are serialized edge-by-edge, where one edge will correspond to one line in the serialization format. An edge is represented by one line of the following format:

```
e <src_id> <tgt_id> <src_label> <tgt_label> <edge_label>
```

Here, <src_label>, <tgt_label>, and <edge_label> are the labels of the labeled graph corresponding to the simple change graph (cf. Section D), and <src_id> and <tgt_id> are identifiers for the source and target vertices of the edge, respectively.

An extract of an example simple change graph serialization is given in Listing 1.

When we designed this serialization format, we had already the application of LLMs for model completion in mind. More common graph serialization formats start with a list of nodes and then list edges between these nodes. Instead, we define nodes implicitly, while defining edges. Therefore, node labels of already defined nodes will be duplicated in our approach. In practise, we avoid this though, by replacing an already defined node label by an *empty JSON*.

Especially in the case of fine-tuning, we do want to avoid that the LLM has first to *guess* the right nodes of the graph before it continues with the edges. The EdgeList format allows for a continues generation of edges and avoids this break of listing nodes before edges.

```
t # 1
e 0 1 {..."add", "type":"port"} {..."add", "type":"
↪ component"} {..."add", "type":"port"}
e 0 2 {..."add", "type":"requirement"} {..."add", "type":"
↪ component"} {..."add", "type":"requirement"}
```

Listing 1: An example SCG in the EdgeList format.

In a textual representation, we have to linearize also the listing of the edges, that is, we need to decide on an ordering of the edges of the graph. In our case, the order of edges for this serialization is determined using a depth-first search, since it proved to perform best in a pilot study. Nevertheless, other serialization strategies are conceivable and could be investigated as part of future work.

F. Candidate Generation

We utilize two different tactics/algorithms to generate candidates for the software model completion. In the first tactic, we keep the control over the sampling procedure and use the language model to generate the completions token-wise. We therefore use this tactic only with a "completion-like" interface. This tactic is more expensive, since we have to process the entire context for every token. Especially for GPT-4, this tactic is not feasible (without major adaptions). For the second tactic, we let utilize the LLM's capabilities to directly generate entire candidate completions. In the present study, we are using this tactic for all completions generated with GPT-4.

1) Beam-like Sampling Algorithm

The candidate generation works as follows (see pseudo code in Listing 1): The algorithm takes a set of *incomplete* edit operation candidates (in the form of serialized simple change graphs) and uses the fine-tuned language model to sample new edge candidates and appends them to the incomplete edit operation candidate (Line 12). The sampling generates all possible extensions above a certain probability threshold. Since we cannot guarantee that the extensions lead to a correct EdgeList serialization, we check the syntactical correctness and reject incorrect extensions (Line 13). Furthermore, even syntactically valid extensions could be invalid according to the metamodel and have to be rejected likewise (Line 14). After that, the corresponding simple change graph represents a valid edit operation by definition. Based on a graph isomorphism test, we then filter out duplicates (Line 15). Although graph isomorphism is theoretically expensive from a computational perspective, in our setting, it is acceptable since we have only a few medium size graphs, and employ Weisfeiler-Lehman hashes [32] to speed up the comparison. We add complete candidates to the output list (Line 19) and repeat this process until all candidates are complete (Line 9). Whether a candidate is complete is checked using several conditions such as the total probability of the candidate, a drop in the probability of a generated edge, or a generated stop token.

2) ChatModel Instruction

An alternative to the token-wise beam search above is to let the LLM decide when to stop. If multiple candidates should be generated, one could sample with a certain temperature > 0.

For our completion generation, we use the following instruction prompts:

Listing 2: Single edge completion prompt.

```
You are an assistant that is given a list of change graphs
in an edge format. That is, the graph is given edge by edge
. The graphs are directed, labeled graphs. An edge is
serialized as
"e src_id tgt_id edge_label src_label tgt_label"

Labels are dictionaries. If a node appears in more than one
 edge, the second time it appears it is replaced by "_" to
avoid repetition.

E.g.:
e 0 1 a b bar
e 1 2 bla _ foo

The second edge here would be equivalent to:
"e 1 2 bla bar foo"

There are some change graphs given as examples. Graphs are
separated by "\n\n$$\n---\n".

The last graph in this list of graphs is not yet complete.
Exactly one edge is missing.
Your task is it, to complete the last graph by guessing the
 last edge. You can guess this typically by looking at the
```

Algorithm 1: Pseudocode for the candidate generation.

```
Function generateCandidates($\varepsilon$, $\mathcal{L}$, $\mathcal{TM}$):
begin
    Input: $\varepsilon$ – given context serialization
    $\mathcal{L}$- fine-tuned language model
    $\mathcal{TM}$- metamodel
    Output: $[\varepsilon_1, \ldots, \varepsilon_n]$ - list of candidates
    incomplete ← [$\varepsilon$]; ▷ set of incomplete edit operations
    complete ← [] ;  ▷ set of complete edit operations
    while size(incomplete) $> 0$ do
        ext ← [];  ▷ set of extended edit operations
        foreach op ∈ incomplete do
            ext += sampleEdges($\mathcal{L}$, op);
            ext ← checkCorrectSCG(ext);
            ext ← checkMetaModel($\mathcal{TM}$, ext);
            ext ← prune(ext, complete);
            incomplete ← [];
            foreach $\tilde{\varepsilon}$ ∈ ext do
                if complete($\tilde{\varepsilon}$) then
                    complete += $\tilde{\varepsilon}$;
                else
                    incomplete += $\tilde{\varepsilon}$;
    return complete
```

```
examples and trying to deduce the patterns in the examples.
 Give this missing edge in the format
”e src_id tgt_id edge_label src_label tgt_label”. Note that
 the beginning ”e” is already part of the prompt.
```

Listing 3: Multiple edge completion prompt.

```
You are an assistant that is given a list of change graphs
in an edge format. That is, the graph is given edge by edge
. The graphs are directed, labeled graphs. An edge is
serialized as
”e src_id tgt_id edge_label src_label tgt_label”

Labels are dictionaries or concatenations of change type
and node/edge type. If a node appears in more than one edge
, the second time it appears it can be replaced by ”_” to
avoid repetition.

E.g.:
e 0 1 a b bar
e 1 2 bla _ foo

The second edge here would be equivalent to:
”e 1 2 bla bar foo”

There are some change graphs given as examples. Graphs are
separated by ”\n\n$$\n---\n”.

The last graph in this list of graphs is not yet complete.
Some edges are missing.
Your task is it, to complete the last graph by guessing the
 missing edges. You can guess this typically by looking at
the examples and trying to deduce the patterns in the
examples. Give the missing edges in the format
”e src_id tgt_id edge_label src_label tgt_label”. Note that
 the beginning ”e” is already part of the prompt. After the
 last edge of the change graph, add two new lines.
```

G. *Diversity Based Few-Shot Sample Retrieval*

RAMC involves retrieving similar examples to the software model the user is currently working on. To ensure diversity, typical implementations of maximum marginal relevance retrieve elements, element by element, and ensures maximal distance to the already existing elements. This can lead to below optimal samples, because samples that have already been retrieved are later not removed. In essence, typical maximum marginal relevance implementation can get stuck in local optima.

In our sampling algorithm, the goal is the same as in maximum marginal relevance. That is, we want to select samples that are similar to a given input but the samples themselves are diverse. We extend on maximum marginal relevance by using the following retrieval procedure: First, for a given embedding, we retrieve a given number n of elements that are similar to this given embedding. We call this set S. Second, from S, we want to draw another sample of a given size k, that maximizes the distances between all elements. Initially, we draw k random elements from S. Let's call this set D Third, we choose one of these elements e and replace it by an element from $(S \setminus D) \cup \{e\}$ that has maximum distance to the $D \setminus \{e\}$. Finally, we iterate this procedure for a given number of iterations and try to choose at least one element of the initial set D once.

H. *Few-shot examples*

Due to space limitations, we omitted the few-shot examples in Figure 1 which are presented here. These examples demonstrate the retrieval of, in this specific case, four few-shot instances through our vector store. The similarity-based retrieval mechanism is further detailed in Section G.

```
t # 5175
e 2 1 ”{’changeType’: ’Add’, ’type’: ’reference’, ’
referenceTypeName’: ’eOperations’}” ”{’changeType’: ’
Preserve’, ’type’: ’object’, ’className’: ’EClass’, ’
attributes’: {’id’: ’_ftfz6d6tEei97MD7GK1RmA’, ’
eAnnotations’:[’org.eclipse.emf.ecore.impl.
EAnnotationImpl@1d8d14f1 (source: http://www.eclipse.org/
emf/2002/GenModel)’,’org.eclipse.emf.ecore.impl.
EAnnotationImpl@c8ca1dd (source: duplicates)’],’name’:’
Classifier’,’ePackage’:’uml’,’abstract’:’true’,’interface
’:’false’, ’eIDAttribute’:’name’,’eStructuralFeatures’:[’
isAbstract’,’generalization’,’powertypeExtent’,’feature’,’
inheritedMember’,’redefinedClassifier’,’general’,’
substitution’,’attribute’,’representation’,’
collaborationUse’,’ownedUseCase’,’useCase’],’
eGenericSuperTypes’:[’org.eclipse.emf.ecore.impl.
EGenericTypeImpl@239c2926 (expression: Namespace)’,’org.
eclipse.emf.ecore.impl.EGenericTypeImpl@526bc7ba (
expression: RedefinableElement)’,’org.eclipse.emf.ecore.
impl.EGenericTypeImpl@6999e7c8 (expression: Type)
’,’...’]}}” ”{’changeType’: ’Add’, ’type’: ’object’, ’
className’: ’EOperation’, ’attributes’: {’id’: ’
_mrycqN6tEei97MD7GK1RmA’, ’name’:’getAllUsedInterfaces’,’
ordered’:’false’,’unique’:’true’,’lowerBound’:’0’,’
upperBound’:’-1’,’many’:’true’,’required’:’false’,’eType’:’
Interface’,’eGenericType’:’org.eclipse.emf.ecore.impl.
EGenericTypeImpl@762545f6 (expression: Interface)’,’
eContainingClass’:’Classifier’}}”
e 2 0 ”{’changeType’: ’Add’, ’type’: ’reference’, ’
referenceTypeName’: ’eOperations’}” _ ”{’changeType’: ’Add
’, ’type’: ’object’, ’className’: ’EOperation’, ’attributes
’: {’id’: ’_mrycp96tEei97MD7GK1RmA’, ’name’:’
getUsedInterfaces’,’ordered’:’false’,’unique’:’true’,’
lowerBound’:’0’,’upperBound’:’-1’,’many’:’true’,’required
’:’false’,’eType’:’Interface’,’eGenericType’:’org.eclipse.
emf.ecore.impl.EGenericTypeImpl@3d23f56e (expression:
Interface)’,’eContainingClass’:’Classifier’}}”
```

```
$$
---
t # 1250
e 2 1 "{'changeType': 'Add', 'type': 'reference', '
referenceTypeName': 'eOperations'}" "{'changeType': '
Preserve', 'type': 'object', 'className': 'EClass', '
attributes': {'id': '_ftfz6d6tEei97MD7GK1RmA', '
eAnnotations':['org.eclipse.emf.ecore.impl.
EAnnotationImpl@50bd114f (source: http://www.eclipse.org/
emf/2002/GenModel)','org.eclipse.emf.ecore.impl.
EAnnotationImpl@11c9b440 (source: duplicates)'],'name':'
Classifier','ePackage':'uml','abstract':'true','interface
':'false','eIDAttribute':'name','eStructuralFeatures':['
isAbstract','generalization','powertypeExtent','feature','
inheritedMember','redefinedClassifier','general','
ownedUseCase','useCase','substitution','attribute','
representation','collaborationUse','ownedSignature'],'
eGenericSuperTypes':['org.eclipse.emf.ecore.impl.
EGenericTypeImpl@1504a6f7 (expression: Namespace)','org.
eclipse.emf.ecore.impl.EGenericTypeImpl@65db7f4d (
expression: RedefinableElement)','org.eclipse.emf.ecore.
impl.EGenericTypeImpl@225a383c (expression: Type)
','...']}}" "{'changeType': 'Add', 'type': 'object', '
className': 'EOperation', 'attributes': {'id': '
_inuJYt6tEei97MD7GK1RmA', 'name':'getOperation','ordered':'
false','unique':'true','lowerBound':'0','upperBound':'1','
many':'false','required':'false','eType':'Operation','
eGenericType':'org.eclipse.emf.ecore.impl.
EGenericTypeImpl@4e5c0171 (expression: Operation)','
eContainingClass':'Classifier','eParameters':['name']}}"
e 1 0 "{'changeType': 'Add', 'type': 'reference', '
referenceTypeName': 'eParameters'}" _ "{'changeType': 'Add
', 'type': 'object', 'className': 'EParameter', 'attributes
': {'id': '_inuJY96tEei97MD7GK1RmA', 'name':'name','ordered
':'false','unique':'true','lowerBound':'1','upperBound
':'1','many':'false','required':'true','eType':'String','
eGenericType':'org.eclipse.emf.ecore.impl.
EGenericTypeImpl@bbcf831 (expression: String)','eOperation
':'getOperation'}}"

$$
---
t # 2292
e 0 2 "{'changeType': 'Remove', 'type': 'reference', '
referenceTypeName': 'eAnnotations'}" "{'changeType': '
Preserve', 'type': 'object', 'className': 'EClass', '
attributes': {'id': '_fthA796tEei97MD7GK1RmA', '
eAnnotations':['org.eclipse.emf.ecore.impl.
EAnnotationImpl@2fa33653 (source: http://www.eclipse.org/
emf/2002/GenModel)','org.eclipse.emf.ecore.impl.
EAnnotationImpl@59d423ca (source: duplicates)'],'name':'
Extension','ePackage':'uml','abstract':'false','interface
':'false','eOperations':['non_owned_end','is_binary','
getStereotype','getStereotypeEnd','isRequired','
getMetaclass','metaclassEnd'],'eStructuralFeatures':['
isRequired','metaclass'],'eGenericSuperTypes':['org.eclipse
.emf.ecore.impl.EGenericTypeImpl@3ff99636 (expression:
Association)']}}" "{'changeType': 'Remove', 'type': 'object
', 'className': 'EAnnotation', 'attributes': {'id': '
_oBpkOd6tEei97MD7GK1RmA', 'source':'http://www.eclipse.org/
emf/2002/GenModel','details':['org.eclipse.emf.ecore.impl.
EStringToStringMapEntryImpl@95f12e0 (key: documentation,
value: An extension is used to indicate that the properties
 of a metaclass are extended through a stereotype, and
gives the ability to flexibly add (and later remove)
stereotypes to classes.)'],'eModelElement':'Extension'}}"
e 2 3 "{'changeType': 'Remove', 'type': 'reference', '
referenceTypeName': 'details'}" _ "{'changeType': 'Remove',
 'type': 'object', 'className': 'EStringToStringMapEntry',
'attributes': {'id': '_oBpkOt6tEei97MD7GK1RmA', 'key':'
documentation','value':'An extension is used to indicate
that the properties of a metaclass are extended through a
stereotype, and gives the ability to flexibly add (and
later remove) stereotypes to classes.'}}"
e 0 4 "{'changeType': 'Add', 'type': 'reference', '
referenceTypeName': 'eAnnotations'}" _ "{'changeType': 'Add
', 'type': 'object', 'className': 'EAnnotation', '
attributes': {'id': '_0oByC96tEei97MD7GK1RmA', 'source':'
http://www.eclipse.org/emf/2002/GenModel','details':['org.
eclipse.emf.ecore.impl.EStringToStringMapEntryImpl@5cd02377
 (key: documentation, value: An extension is used to
indicate that the properties of a metaclass are extended
through a stereotype, and gives the ability to flexibly add
 (and later remove) stereotypes to classes.\\n<p>Merged
from package UML (URI {@literal http://www.omg.org/spec/UML
/20110701}).</p>)'],'eModelElement':'Extension'}}"
e 4 1 "{'changeType': 'Add', 'type': 'reference', '
referenceTypeName': 'details'}" _ "{'changeType': 'Add', '
type': 'object', 'className': 'EStringToStringMapEntry', '
attributes': {'id': '_0oByDN6tEei97MD7GK1RmA', 'key':'
documentation','value':'An extension is used to indicate
that the properties of a metaclass are extended through a
stereotype, and gives the ability to flexibly add (and
later remove) stereotypes to classes.\\n<p>Merged from p
'}}"

$$
---
t # 88
e 0 2 "{'changeType': 'Add', 'type': 'reference', '
referenceTypeName': 'eAnnotations'}" "{'changeType': '
Preserve', 'type': 'object', 'className': 'EClass', '
attributes': {'id': '_fZD13N6tEei97MD7GK1RmA', '
eAnnotations':['org.eclipse.emf.ecore.impl.
EAnnotationImpl@68481491 (source: http://www.eclipse.org/
emf/2002/GenModel)','org.eclipse.emf.ecore.impl.
EAnnotationImpl@4faef368 (source: duplicates)'],'name':'
DataType','ePackage':'cmof','abstract':'false','interface
':'false','eIDAttribute':'name','eStructuralFeatures':['
ownedOperation','ownedAttribute'],'eGenericSuperTypes':['
org.eclipse.emf.ecore.impl.EGenericTypeImpl@7ebe7d9f (
expression: Classifier)']}}" "{'changeType': 'Add', 'type':
 'object', 'className': 'EAnnotation', 'attributes': {'id':
 '_ffDLSt6tEei97MD7GK1RmA', 'source':'http://www.eclipse.
org/emf/2002/GenModel','details':['org.eclipse.emf.ecore.
impl.EStringToStringMapEntryImpl@5e553e0a (key:
documentation, value: A data type is a type whose instances
 are identified only by their value. A data type may
contain attributes to support the modeling of structured
data types.)'],'eModelElement':'DataType'}}"
e 2 1 "{'changeType': 'Add', 'type': 'reference', '
referenceTypeName': 'details'}" _ "{'changeType': 'Add', '
type': 'object', 'className': 'EStringToStringMapEntry', '
attributes': {'id': '_ffDLS96tEei97MD7GK1RmA', 'key':'
documentation','value':'A data type is a type whose
instances are identified only by their value. A data type
may contain attributes to support the modeling of
structured data types.'}}"
e 3 4 "{'changeType': 'Remove', 'type': 'reference', '
referenceTypeName': 'details'}" "{'changeType': 'Remove', '
type': 'object', 'className': 'EAnnotation', 'attributes':
{'id': '_fZD13d6tEei97MD7GK1RmA', 'source':'http://www.
eclipse.org/emf/2002/GenModel','details':['org.eclipse.emf.
ecore.impl.EStringToStringMapEntryImpl@77d8e24f (key:
documentation, value: A data type is a type whose instances
 are identified only by their value. A DataType may contain
 attributes to support the modeling of structured data
types.\\n\\n\\n\\nA typical use of data types would be to
represent programming language primitive types or CORBA
basic types. For example, integer and string types are
often treated as data types.\\r\\nDataType is an abstract
class that acts as a common superclass for different kinds
of data types.)'],'eModelElement':'DataType'}}" "{'
changeType': 'Remove', 'type': 'object', 'className': '
EStringToStringMapEntry', 'attributes': {'id': '
_fZD13t6tEei97MD7GK1RmA', 'key':'documentation','value':'A
data type is a type whose instances are identified only by
their value. A DataType may contain attributes to support
the modeling of structured data types.\\n\\n\\n\\nA typical
 use of data types would be to'}}"
e 0 3 "{'changeType': 'Remove', 'type': 'reference', '
referenceTypeName': 'eAnnotations'}" _ _

$$
---
```

I. Further (pre-)processing and filtering steps

We perform some additional filtering steps during the (pre-)processing of simple change graphs and the sampling. For the sake of clarity, we omitted them in the description of the approach and experiment description. The applied filters are the following:

- Because we have a limitted context size available for the LLMs, very long attribute descriptions (for example in comments) are limited to a length of 200 characters. Everything longer than 200 characters has been cut and "..." are appended.
- When sampling few-shot samples, and the overall prompt size becomes too long, we remove few-shot samples until the prompt fits into the model.
- Serialized simple change graphs that are too large to fit in the context of the language model are filtered.
- To save tokens and therefore reduce language model usage costs, we do not repeat node labels, but instead replace them by a "_" token.
- We filtered duplicated simple change graphs.
- Models from the original REPAIRVISION dataset that could not be loaded or had empty history were removed. The description of the dataset parameters in Section V-B describes the state after this filtering.

J. Detailed Results of the Industry Dataset and Experiment 3

Table VI summarizes the results of our results from Experiment 3. We distinguished between four completion task characteristics in the rows of the table: noise present, project specific change, complex change, and reoccurring pattern. All four are binary relations. "Noise Present" indicates whether there are changes entangled in the task or in the few-shot examples. We considered the task to be a "Project Specific Change", if the change was not common for the modeling language (SysML), but rather some pattern we observed for this project only. "Complex Change" indicates a change that does consist of several interconnected atomic changes (e.g., adding an attribute or adding a class, i.e., implicitly we distinguish between atomic changes and complex changes, as common in the field [38]). Finally "Reoccurring Pattern" describes if we observe the pattern of the task at hand also in the few-shot samples. That is, aside from concrete attribute values, the change happens often in the project and can be retrieved via our semantic retrieval. Correctness is classified as follows in the columns of the table: We only consider semantically correct completions (evaluated via a manual analysis) as *correct*. The *incorrect* completions, we further classify in "semantically conceivable" (i.e., the change is not the one observed in the ground truth, but without further context it would also be meaningful), "Semantically Incorrect" (i.e., format correct, structurally correct, but the completion has a meaning different from the ground truth completion), "structurally incorrect" (i.e., a reference connects no the right nodes, or a new class is associated to another class where nothing should be added, etc.), and "format incorrect" (i.e., without error correction, the graph serialization could not be parsed, e.g., because an existing node id is reused by another node).

Remark. In Section V, we stated that there is no significant relationship between the correctness and the number of few-shot examples that are added to the prompt. For the INDUSTRY dataset, we additionally recorded, if (at least one) similar pattern is among the few-shot examples. Separating these two cases – i.e., there is a similar pattern among the few-shot examples or not – we see that in the first case there is no significant relationship between the number of few-shot examples, while in the second case there is a significant relationship.

This suggest that as long as a similar few-shot example is available, the amount of few-shot examples does not matter too much, while in the case that the examples are rather unrelated, the amount plays a role.

K. Detailed Results of the Fine-Tuning Experiments

This section delves into a detailed analysis of our last experiment highlighting the influence of various factors on the *average token accuracy*. We are especially interested in the model token accuracy of the fine-tuned language model in relationship with the properties of the dataset and the properties of the fine-tuning such as the number of fine-tuning epochs and the base language model used. We fine-tune one LLM per simulated repository. As base models we choose text-ada-001, text-curie-001, and text-davinci-003 from the GPT-3 family. Since fine-tuning the text-davinci-003 model is quite expensive (i.e., 3 Cents per thousand tokens at the time of this experiment), we fine-tuned this model only for the model repositories where the perturbation probability equals 100% (the ones which are typically the harder ones). This leaves us with a total of 112 fine-tuned models (24 simulated repositories for text-ada-001 and text-curie-001 and 8 for text-davinci-003, which is 24*2*2+ 8*2*1 = 112) and a total fine-tuning cost of 347$. Building on the insights previously touched upon, our analysis reveals a strong correlation between average token accuracy and the number of fine-tuning epochs. Furthermore, it becomes evident that larger models exhibit better performance in terms of average token accuracy. Regarding the repository properties, we only find significant negative correlations with the perturbation probability (VII). We therefore also analyze model completions from a graph matching perspective (like already mentioned in Experiment 4). Since generating all completion candidates for all test samples of all fined-tuned language models would be even more expensive, we select two fine-tuned language models, the less cost-intensive alternative, and perform the analysis of the model completions on them.

L. Related Work

1) Current problems in the research domain

Research in model-driven engineering faces several challenges that should receive increased attention in the future.

Furthermore, the scarcity of reusable datasets [21, 59, 46, 10] for many use cases in model-driven engineering hinders the comparison of different approaches, which is then often reduced to a qualitative analysis. The lack of proper datasets also poses a challenge for the development and evaluation of data-driven (e.g., machine learning) approaches in model-driven engineering. To circumvent this lack of datasets, many authors in model-driven engineering research report on experiences using their approaches in a concrete application context, that is, as part of a tool. Reporting on an evaluation in a concrete

Table VI: Comparison of different failure types along several characteristics of the completion task. This table summarizes the results of our manual analysis of the INDUSTRY dataset.

		Level of Correctness						
Task Characteristic		Correct	Incorrect				Total	Total (%)
		Semantically Correct	Semantically Conceivable	Semantically Incorrect	Structurally Incorrect	Format Incorrect		
Noise Present	TRUE	30.77%	23.08%	15.38%	15.38%	15.38%	13	11
	FALSE	66.06%	14.68%	8.26%	4.59%	6.42%	109	89
Project Specific Change	TRUE	74.55%	3.64%	7.27%	5.45%	9.09%	55	45
	FALSE	52.24%	25.37%	10.45%	5.97%	5.97%	67	55
Complex Change	TRUE	66.67%	23.81%	4.76%	0.00%	4.76%	21	17
	FALSE	61.39%	13.86%	9.90%	6.93%	7.92%	101	83
Reoccurring Pattern	TRUE	71.13%	17.53%	6.19%	2.06%	3.09%	97	80
	FALSE	28.00%	8.00%	20.00%	20.00%	24.00%	25	20
Total		62.30%	15.57%	9.02%	5.74%	7.38%	122	100

Table VII: Pearson correlations of the average token accuracy w.r.t. several properties. Repo$_D$ denotes the number of revisions, Repo$_E$ the number of applied edit operations, and Repo$_P$ the perturbation probability.

	Repo$_D$	Repo$_E$	Repo$_P$	Epochs	Token Count	Base Model
Average Token Accuracy	0.16	0.13	-0.22*	0.69**	0.08	0.43**
Token Accuracy (All)	0.16	0.13	-0.22*	0.69**	0.08	0.43**
Token Accuracy (Ada)	0.26	0.22	-0.43*	0.72**	0.14	–
Token Accuracy (Curie)	0.13	0.12	-0.35*	0.82**	0.02	–
Token Accuracy (Davinci)	0.02	-0.04	–	0.94**	-0.06	–

(**: $p < .001$, *: $p < 0.05$)

application setting, again, makes it difficult to compare against the approach, especially if the application context or the tool is not available to the public and/or user studies are performed.

There are only a few datasets available that can be used to evaluate model completion. In the concrete example of model completion, the evaluation is often performed on a dataset of model snapshots, from which elements are removed artificially. Instead, it would be more realistic to have pairs of to-be-completed models and their completed counterparts.

Finally, there are no commonly accepted evaluation metrics and often technologies or proposed approaches are evaluated in a manner that is only applicable for the specific use case at hand. Only a minority of the literature reports on metrics that are independent of their specific approach and only depending on the use case (i.e., model completion). For instance, a model completion methodology could suggest the top-10 names for meta-model classes for inclusion in a meta-model, with the evaluation of this method focusing solely on the accuracy of these ten recommendations. Consequently, this creates a challenge in directly comparing such an approach to others that might recommend a single name while also suggesting relationships between the newly added class and existing classes. Further it would require a ground truth of to-be-completed and completed models, which is not available for most datasets. But even here, its not easy to define what a correct completion is. For example, if a model element is missing in the incomplete model, but the model element is not required for the model to be valid, is it a correct completion or not? Likewise, if a recommended class name is a synonym of the correct class name, is it a correct completion or not? Note that for source code, there are commonly accepted datasets such as HumanEval [17] and evaluation metrics [17] to evaluate code completion approaches. For example, since there is a well-defined execution semantics, the evaluation of a code completion approach can be performed by checking the correctness of the code completion in a test suite. For many models (e.g., UML, SysML, Ecore, etc.), there is no well-defined execution semantics and therefore a test approach for evaluation would not be applicable to software models, in general.

2) *Comparison and differentiation from other approaches*

In Table VIII, we summarize the related work with a specific focus on the model completion task. For each approach, we included information about the specific task, the method used and the evaluation process, including the data used for evaluation and the prerequisites required for replicating the evaluation. Given the sometimes challenging nature of tracking the availability of artifacts, we acknowledge that some information might not be entirely accurate, and we apologize for any inadvertent inaccuracies. A main finding of our analysis of related work is that, currently, a direct comparison with other approaches is for most approaches infeasible, due to the field's novelty, the absence of commonly accepted metrics, or a focus of previous work on artificial datasets without real-world model evolution. In the following paragraph, we will delve into the reasons why a direct comparison to the approaches in Table VIII is not easily possible.

a) (Partial) model completion

The approach by Agt-Rickauer et al. [2, 3] focuses on suggesting related class names, potential sub- or super-class names, and different names for connections given a specific focus point in the model. However, their approach does not extend to suggesting attributes, operation names, or relationship types, making a comparison to our approach challenging.

Table VIII: Figures

Paper	Task	Method	Evaluation	Data	Prerequisites (for evaluation)	History
[1] [67]	Single-step operations and similar, related Simulink systems	Information retrieval (association rule mining, frequency based matching)	Metrics analysis (prediction, accuracy, error classification)	Simulink (available)	None	No
[31]	Library blocks	Information retrieval (association rules, collaborative filtering)	Metrics analysis (precision, recall, and F-measure)	Simulink (available)	None	No
[2] [3]	Related classes, possible sub- or super-classes, relationships between elements, names of elements	Knowledge graphs, semantic web technologies	Planned user study but not yet conducted (no metric)	UML (not available)	Conceptual knowledge bases (semantically related terms, built from natural language data) and semantic network (not available)	No
[22]	Activity nodes	Information retrieval (similarity-based, pattern mining, pattern as relationships between activity nodes)	Metrics analysis (HitRate, Precision, Recall, and F1 Score)	Business process modeling (not available)	Database constructed from existing processes (not available)	No
[23]	Entities in metamodels (classes, structural features), no support for types of the recommended attributes, relationships	Information retrieval (similarity-based, collaborative filtering strategy)	Metrics analysis (rather best case scenario – out of N items some of the (possibly huge) model are correct – (success rate (SR@N), precision, recall, F1 score))	Ecore metamodels (available)	Predefined categories/labels beneficial	No
[27]	Class names, attributes, operations	Clustering algorithm (on semantic relations)	User study (relevant and new recommendations (PN), non useful recommendations (NU) not recommended but included in individual design (NR), relevant rate (TP), accuracy rate of new suggestions (TN))	UML (not available)	Clustered UML diagrams (not available)	No
[40]	Edit rules	Association rule mining	Metrics analysis (Precision)	Eclipse GMF Project meta-models (available)	Catalog of change patterns (not available)	Yes
[42] [43] [49]	Model completion	Rule-based matching	User study (number of saved user actions, time against manual completion)	UML (not available)	Catalog of change patterns (not available)	No
[25]	Model completion	Information retrieval (similarity-based, graph kernels, TF-IDF)	Metrics analysis (but rather a best-case scenario, check whether one of top-N recommendation is correct –, evaluated on "token" level (Success rate, Precision, Recall, and F-measure (modified)), Structural correctness (e.g., one new class connected to 2 or more other classes) not evaluated (and not reflected in the approach))	ModelSet, Maven repository (reverse engineered class diagrams from Java code), JSON crawled from GitHub, Ecore metamodels (partially available)	None	No
[15]	Class names, attributes, association names	Machine Learning (GPT-3, few-shot learning)	Metrics analysis (30 models selected and evaluated manually, precision, recall)	ModelSet (available)	None	No
[11]	Model completion	Machine Learning (reuse pre-trained models, project-specific training, NLP-based system, word embedding similarity based on textual information)	Metric analysis (but rather best case scenario, out of N items some of the (possibly huge) model are correct –)(Precision, Recall))	Industrial data (notice management system for incidents in municipal water supply and sewage in Malaga)(not available)	Textual information required (of project and/or related business domain)(not available)	No
[24]	Type of elements (class, attribute, ...) but no details like name and values	Machine Learning (Encoder-Decoder LSTM neural network)	Metric analysis, limited possibilies due to ignoring names and values, out of N items some are correct, not clear how many items get recommended (Success rate, precision recall)	BPMN	None	Yes

Conversely, our approach goes far beyond suggesting the names of new elements. Furthermore, the evaluation of their method is deeply integrated within the tool, making a direct comparison impossible. This approach relies on conceptual knowledge bases (comprising semantically related terms built from natural language data) and a semantic network, neither of which are available for external validation. Consequently, it is challenging to verify when and how the suggestions are semantically and structurally correct.

The approach by Elkamel et al. [27] suggests entities in metamodels, such as classes and structural features, but does not support types for the recommended attributes or relationships. In an offline phase, they use a clustering algorithm to partition UML classes collected from various UML class diagrams based on the semantic relations between their characteristics. Subsequently, they recommend semantically similar whole classes, and individual methods and attributes of that class can be accepted or rejected. Their approach is not based on historical data. On the other hand, we cannot apply their approach to our data, as they only suggest entire classes while our approach focuses on a more general setting. Their approach directly relies on user feedback, as entire classes are suggested for the user including its attributes to accept or reject. As a result, it is rare for an entire class to be completely correct initially. While their focus is more on the user setting, we focus on the core effectiveness of the LLM technology. This makes a direct comparison between our methods unintuitive.

Similarly, Di Rocco et al. [23] propose an approach for suggesting new classes and structural features (attributes and references) in metamodels. Their method generates recommendations as a ranked list of classes if the active context is a package, or as a ranked list of structural features if the active context is a class. This approach involves identifying a subset of the most similar metamodels from given metamodel repositories and determining the most similar contexts within that subset. However, the method lacks support for recommending the types of attributes and relationships.

Di Rocco et al. [24] further present a recommender system that uses an Encoder-Decoder neural network to assist modelers with performing editing operations. The system learns from past modeling activities and is evaluated on a BPMN dataset. These past activities are modeled as a edit operation sequences. One limitation of this specific format is that the changes of an element in the edit operation sequences can be scattered throughout the complete sequence, with possibly hundreds of other edit operations between them. This also means that connected/related elements or elements that belong together can appear at completely different locations within such a sequence. These connected/related elements can give valuable context to the model completion task. If then, as in the work by Di Rocco et al., only the last 10 edit operations are considered, important information regarding the local (graph-like context) might get lost. This issue becomes more pronounced as models increase in size. One could instead not only focus on the last x edit operations, but instead put the entire history of a model into the LLM context. Anyway, especially for large models, providing the entire history of the model as context is infeasible and may not fit in the context of an LLM.

We cannot compare their approach to our model completion approach because it does not include the specific details and values of operations. For instance in the example in Listing 4.

```
set-att name BPMN2ActionContributor to #200
```

Listing 4: An example of the NEMO [24]

In the setAtt operation, the class name being created isn't suggested. Instead, each event is simplified to a tuple <setAtt, class, name>. While their approach focuses on proposing simplified completions, as highlighted in their work, our approach suggests more complex model elements with detailed, specific values (e.g. class names, concrete attribute values). An example of a concrete, linearized model completion suggestion of RAMC is given Listening 5.

```
1 2 "{'changeType': 'Add', 'type': 'reference',
'referenceTypeName':'eOperations'}" _ "{'changeType':'Add',
'type': 'object', 'className': 'EOperation', 'attributes'
{'id':'_lU7gFt6tEei97MD7GK1RmA','name':
'getMetaclass','ordered':'false','unique':'true',
'lowerBound':'0','upperBound':'1','many':'false',
'required':'false','eType':'Metaclass',
'eGenericType':'Metaclass','eContainingClass':
'Extension'}}"
```

Listing 5: A RAMC completion candidate

The competition candidate includes a specific change to the model, detailing how it is connected to other elements (as a quick reminder, 1 and 2 represent the source and target nodes, followed by the edge attributes in the first). It suggests specific values and the names of changed attribute elements and much more. All in all, comparing our work to the work by Di Rocco et al. would be an unfair comparison for both sides.

Di Rocco et al. [25] focus on model completion by suggesting new classes and structural features (attributes, references, methods, and fields). This is achieved by constructing a separate graph for each class in the models and use graph kernel similarity to identify the most similar items among the training set to the partial model that should be completed. However, their approach does not ensure structural correctness, such as where to add a class or how elements should be connected overall. Additionally, they report very low precision and recall values, indicating that their method would likely perform even worse on our real-world and industrial datasets.

Regarding the use of language models, Chaaben et al. [15] use LLMs and acknowledge that their results are preliminary, considering only a few UML examples (30 domain models, selected manually from the dataset ModelSet). Their evaluation focuses on suggesting class names, attributes of classes and association names. They do not consider historical data, therefore without major adaptation, we cannot perform our experiments on their data. We primarily focus on historical data because we aim to gather real-world examples. Instead of randomly excluding model components and using them as ground truth, we study actual real-world evolution, making the setting much more realistic. Additionally, their approach does not scale for larger models, so our real-world models are by far too large to fit within the context window of the GPT-3 model (text-davinci-002). This challenge is precisely why we decided to focus on historical data in combination with simple change graph slicing. Anyway, since the work by Chaaben et al. [15] is the most closely rated work, we re-implement their approach and tailor it to our dataset to enable a direct comparison between their method and ours.

b) Additional data required for model completion

There are also other approaches, such as those employing rule-based matching based on a predefined catalog of change patterns (edit operations), where additional data is actually required to compare their approach to ours [42, 43, 49, 40]. The work by Burgueno et al. [11] relies on knowledge extracted from textual documents to provide meaningful suggestions. Our approach is pure model completion and we do not include further information in the model completion, i.e., unlike Burgueno et al. [11] we are in a pure model completion setting.

c) Related but distinct tasks

While several other works focus on related but different tasks, we will highlight a few examples here. Mentioning all of them would exceed the scope of this paper, but we hope these examples will clarify this category of related appraoches. The work by Ohrndorf et al. [53], which specifically proposes a model repair approach rather than model completion. Their method generates repair proposals for inconsistencies introduced by incomplete editing processes. Their approach focuses on examples in the revision history, in which constraints were at some point violated and subsequently at a later point fixed. While their method ensures constraints are preserved, it does not handle adding or changing functionality within the system, leaving the modeler to perform the actual modeling work. Like we can not compare fixing syntactic errors in source to suggesting new source code, we can also not compare model repair to model completion. Gomes et al. [30] focus on creating and evolving a system domain model based on interactions in natural language from non-technical users. They utilize Natural Language Processing (NLP) to interpret the users' intents expressed in natural language and transfer these intents to commands the system can understand. This represents an entirely different task. They do not suggest functional changes to the model, but translate the user intentions to a machine readable commands. Kögel et al. [40] present an approach that recommends edit rules and therefore also requires a catalog of change patterns.

d) Domain-specific applications

There is a category of approaches focusing on specific domain languages [31, 1, 67]. Their approaches are limited to Simulink models, which is why we cannot apply them to our data.

Deng et al. [22] propose an approach focused on business process models, specifically BPMN. Their method is not transferable to other domains. They mine relationships among activity nodes from existing processes, store these relations as patterns in a database, and then compare new processes with these patterns. This comparison recommends suitable activity nodes from the most matching patterns to assist in building a new process.

Our analysis above underscores the current challenges faced by the research community. This lack of baselines and benchmarking infrastructure is a critical point. The field is currently in a state of developing the necessary infrastructure, and we are contributing to this effort.