An Empirical Study on Leveraging LLMs for Metamodels and Code Co-evolution

Zohra Kaouter Kebaili[*], **Djamel Eddine Khelladi**[*], **Mathieu Acher**[†], **and Olivier Barais**[‡]
[*]CNRS, Univ. Rennes, IRISA, INRIA, France
[†]INSA Rennes, IUF, IRISA, INRIA, France
[‡]Univ. Rennes, IRISA, INRIA, France

ABSTRACT

Metamodels play an important role in MDE and in specifying a software language. They are cornerstone to generate other artifacts of lower abstraction level, such as code. Developers then enrich the generated code to build their language services and tooling, e.g., editors, and checkers. When a metamodel evolves, part of the code is regenerated and all the additional developers' code can be impacted. Thus, requiring erroneous code to be co-evolved accordingly.

In this paper, we explore a novel approach to mitigate the challenge of metamodel evolution impacts on the code using LLMs. In fact LLMs stand as promising tools for tackling increasingly complex problems and support developers in various tasks of writing, correcting and documenting source code, models, and other artifacts. However, while there is an extensive empirical assessment of the LLMs capabilities in generating models, code and tests, there is a lack of work on their ability to support their maintenance. In this paper, we focus on the particular problem of metamodels and code co-evolution. We first designed a prompt template structure that contains contextual information about metamodel changes, the abstraction gap between the metamodel and the code, and the erroneous code to co-evolve. To investigate the usefulness of this template, we generated three more variations of the prompts. The generated prompts are then given to the LLM to co-evolve the impacted code.

We evaluated our generated prompts and other three of their variations with ChatGPT version 3.5 on seven Eclipse projects from OCL and Modisco evolved metamodels. Results show that ChatGPT can co-evolve correctly 88.7 % of the errors due to metamodel evolution, varying from 75% to 100% of correctness rate. When varying the prompts, we observed increased correctness in two variants and decreased correctness in another variant. We also observed that varying the temperature hyperparameter yields better results with lower temperatures. Our results are observed on a total of 5320 generated prompts. Finally, when compared to the quick fixes of the IDE, the generated prompts co-evolutions completely outperform the quick fixes.

KEYWORDS Metemodel evolution, code evolution, LLM, chatgpt, coevolution, prompt engineering

1. Introduction

Large language models (LLMs) have emerged in the field of natural language processing, exhibiting high aptitude to transform and generate textual data. Taking advantage of LLMs is highly dependent on good prompts. The use of LLMs is based on the human capacity of crafting high quality prompts: precise and concise. AI community gave the term "Prompt engineering" to the process of designing and refining prompts (Clarisó & Cabot 2023). Since their appearance, LLMs have been applied in different domains of scientific research, such as Software Engineering and Model-Driven Engineering (MDE) (Ozkaya 2023; Abukhalaf et al. 2023; Liu et al. 2023; Hou et al. 2023;

JOT reference format:
Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, and Olivier Barais. *An Empirical Study on Leveraging LLMs for Metamodels and Code Co-evolution*. Journal of Object Technology. Vol. 23, No. 3, 2024. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0) http://dx.doi.org/10.5381/jot.2024.23.3.a6

Pearce et al. 2022; Sobania et al. 2022; Ziegler et al. 2022; Vaithilingam et al. 2022; Nguyen & Nadi 2022; Döderlein et al. 2022; Nathalia et al. 2023; Yetiştiren et al. 2023; Guo et al. 2023; Fu et al. 2023; Kabir et al. 2023; Chaaben et al. 2023; Cámara et al. 2023; Abukhalaf et al. 2023).

In MDE, metamodels are cornerstone. They define domain concepts and the relations between them (Cabot & Gogolla 2012). The metamodel is used to generate other artifacts, such as models, transformations, constraints, and code. The generated code can be used later as a code API to build editors, debuggers, and other language services and tooling. The evolution of the metamodel represents one of the challenges encountered in MDE. When the metamodel evolves, the code API is re-generated, and as a consequence, the additional code of the tools built on this code API are impacted and may be broken. However, few approaches have addressed the challenge of metamodels and code co-evolution. In particular, (Riedl-Ehrenleitner et al. 2014; Kanakis et al. 2019; Pham et al. 2017; Jongeling et al. 2020, 2022; Zaheri et al. 2021) focused on consistency checking between models and code, but not its co-evolution. Other works (Yu et al. 2012; Khelladi, Combemale, Acher, Barais, & Jézéquel 2020; Khelladi, Combemale, Acher, & Barais 2020) proposed to co-evolve the code semi-automatically. While LLMs have been so far empirically evaluated to generate qualitative code, refining it, repairing it if vulnerable or augment it (Ozkaya 2023; Abukhalaf et al. 2023; Liu et al. 2023; Hou et al. 2023; Pearce et al. 2022; Sobania et al. 2022; Ziegler et al. 2022; Vaithilingam et al. 2022; Nguyen & Nadi 2022; Döderlein et al. 2022; Nathalia et al. 2023; Yetiştiren et al. 2023; Guo et al. 2023; Fu et al. 2023; Kabir et al. 2023), only few works evaluated LLMs in the context of MDE activities, such as generation of models and constraints (Chaaben et al. 2023; Cámara et al. 2023; Abukhalaf et al. 2023). However, to the best of our knowledge, no existing study evaluated the LLMs capabilities to support developers in the problem of metamodels and code co-evolution.

In this paper, we fill this gap. We explore a novel approach to mitigate the challenge of metamodel evolution impacts on the code using LLMs. Our approach is based on prompt engineering, where we design and generate natural language prompts to best co-evolve the impacted code due to the metamodel evolution. We first designed a prompt template structure that contains contextual information about metamodel changes, the abstraction gap between the metamodel and the code, and the erroneous code to co-evolve. To investigate the usefulness of this template structure, we generated three more variations of these prompts. The generated prompts are then given to the LLM to co-evolve the impacted code errors.

We evaluated our generated prompts and other three of their variations with ChatGPT version 3.5 on seven Eclipse Modeling Framework (EMF) projects from OCL, Modisco, and papyrus with three evolved metamodels. Results show that ChatGPT can co-evolve correctly 88.7% of the errors due to metamodel evolution, varying from 75% to 100% of correctness rate. When varying the prompts, we observed increased correctness in two variants and decreased correctness in another variant. We also observed that varying the temperature hyperparameter yields better results with lower temperatures. Our results are observed on a total of 5320 generated prompts. Finally, when compared to the quick fixes of the IDE, the generated prompts co-evolutions completely outperform the quick fixes.

The paper is structured as follows. Section 2 gives a motivating example to illustrate the problem of metamodels and code co-evolution. Section 3 presents our approach for generating prompts. Section 4 details our followed methodology in this empirical study. Section 5 reports on the obtained results and discusses threats to validity. Section 6 discusses related work, and Section 7 concludes the paper.

2. Motivating example

To illustrate the adressed challenge, let us have an example. Figure 1 shows an excerpt of the version 0.9.0 of "Modisco Discovery Benchmark" metamodel. Modisco is an academic initiative project implemented in the Eclipse platform that has evolved numerous times in the past to support the development of model-driven tools, reverse engineering, verification, and transformation of existing software systems (Bruneliere et al. 2010, 2014). Figure 1 illustrates some of the domain concepts **Discovery**, **Project**, and **ProjectDiscovery** used for the discovery and reverse engineering of an existing software system. From these metaclasses, a first code API is generated, containing Java interfaces and their implementation classes, a factory, a package, etc. Listing 1 shows a snippet of the generated Java interfaces and classes from the metamodel in Figure 1.

The generated code API is further enriched by the developers with additional code functionalities in the "Modisco Discovery Benchmark" project and its dependent projects as well. For instance, by implementing the methods defined in metaclasses and advanced functionalities in new classes. Listing 2 shows the two classes `Report` and `JavaBenchmarkDiscoverer` of the additional code ($\boxed{\textit{Line } 4,8}$ in the same project "Modisco Discovery Benchmark" and in another dependent project, namely the "Modisco Java Discoverer Benchmark" project). In version 0.11.0, the "Modisco Discovery Benchmark" metamodel evolved with several significant changes, among which the following impacting changes: *1)* Renaming the property `DicoveryDate` of the class `JavaBenchmarkDiscoverer` to `DiscoveryDate`, and *2)* Moving the property *discoveryTimeInSeconds* from metaclass `Discovery` to `DiscoveryIteration`.

After applying these modifications, the code of Listing 1 is re-generated from the evolved version of the metamodel, which impacts the existing additional code depicted in Listings 2.

```
//Discovery Interface
  public interface Discovery extends EObject {
 double getTotalExecutionTimeInSeconds();
 void setTotalExecutionTimeInSeconds(double value);
       ...
       }
//Project Interface
  public interface ProjectDiscovery extends
    Discovery {...}
//DiscoveryImpl Class
  public class DiscoveryImpl extends EObjectImpl
    implements Discovery {
```

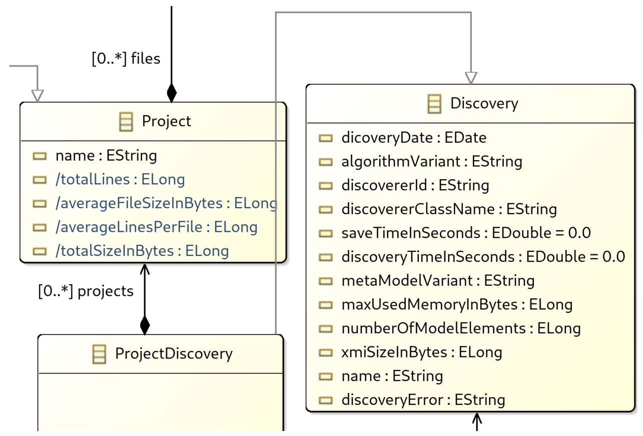

[0..*] files
Project
name : EString
/totalLines : ELong
/averageFileSizeInBytes : ELong
/averageLinesPerFile : ELong
/totalSizeInBytes : ELong
[0..*] projects
ProjectDiscovery
Discovery
dicoveryDate : EDate
algorithmVariant : EString
discovererId : EString
discovererClassName : EString
saveTimeInSeconds : EDouble = 0.0
discoveryTimeInSeconds : EDouble = 0.0
metaModelVariant : EString
maxUsedMemoryInBytes : ELong
numberOfModelElements : ELong
xmiSizeInBytes : ELong
name : EString
discoveryError : EString

Figure 1 Excerpt of Modisco Benchmark metamodel in version 0.9.0.

```
11   public double getTotalExecutionTimeInSeconds()
    {...}
12   public void setTotalExecutionTimeInSeconds(
    double totalExecTime) {...}
13       ...
14  }
```

Listing 1 Excerpt of the generated code in org.eclipse.modisco.infra.discovery.benchmark.

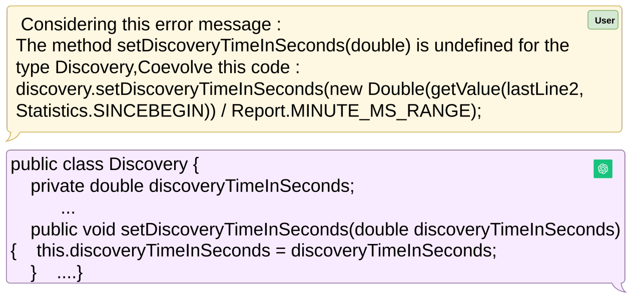
```
1
2   public class Report {
3    ...
4      discovery.setDiscoveryTimeInSeconds(...);
5   }
6
7   public class JavaBenchmarkDiscoverer extends
    AbstractModelDiscoverer<IFile> {
8    ...
9      discovery.setDicoveryDate(new Date());
10   ...
11  }
```

Listing 2 Excerpt of the additional code V1.

```
1
2   public class Report {
3    ...
4      discovery.getIterations().get(0).
5                     setDiscoveryTimeInSeconds(...);
6    ...
7   }
8
9   public class JavaBenchmarkDiscoverer extends
    AbstractModelDiscoverer<IFile> {
10   ...
11     discovery.setDiscoveryDate(new Date());
12   ...
13  }
```

Listing 3 Excerpt of the additional code V2.

The resulting errors in the original code in version 0.9.0 are underlined in red in Listing 2. Listing 3 presents the final result of the manual developer's co-evolution in version 0.11.0. The co-evolved code is underlined in green. The changes *rename* of the property *DicoveryDate* and the *move* of the property



Considering this error message :
The method setDiscoveryTimeInSeconds(double) is undefined for the type Discovery,Coevolve this code :
discovery.setDiscoveryTimeInSeconds(new Double(getValue(lastLine2, Statistics.SINCEBEGIN)) / Report.MINUTE_MS_RANGE);

```
public class Discovery {
   private double discoveryTimeInSeconds;
      ...
   public void setDiscoveryTimeInSeconds(double discoveryTimeInSeconds)
{    this.discoveryTimeInSeconds = discoveryTimeInSeconds;
   }    ....}
```

Figure 2 ChatGPT primitive answer to the naive prompt.

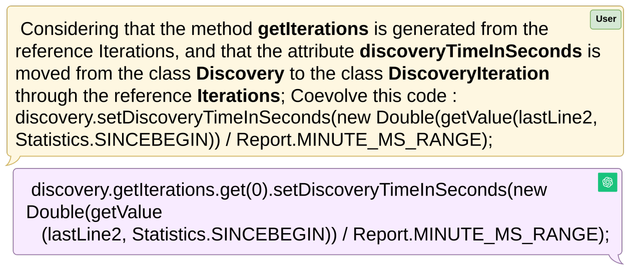
discoveryTimeInSeconds impact their usages ($\boxed{\textit{Line } 4, 8}$ in Listing 2). The impact of renaming *DicoveryDate* is co-evolved by replacing *setDicoveryDate* by *setDiscoveryDate*. The impact of moving the property *discoveryTimeInSeconds* is co-evolved by extending the call path of the method *setDiscoveryTimeInSeconds* through the reference *iterations* by calling the method *getIterations* and getting the first element of the returned list of DiscoveryIteration objects.

Developers unfortunately manually co-evolve the code, which is tedious, error-prone, and time-consuming. One help developers get is from the IDE and the provided quick fixes. For example, when using Eclipse quick fixes to co-evolve these errors, it suggests creating the method `setDiscoveryTimeInSeconds` in the class `Discovery`, which does not meet the required co-evolutions shown in Listing 3.

With the ever-growing popularity and promising results of LLMs, a developer can prompt an LLM to suggest a co-evolution. For example, we asked ChatGPT to co-evolve the error resulted from moving the property *discoveryTimeInSeconds* by giving the erroneous code ($\boxed{\textit{Line } 4}$ in Listing 2) with the message of the error taken from eclipse Problems window. This is the first intuition when using ChatGPT because the developer does not know necessarily the metamodel change causing the error and finding it due to the abstraction gap is a tedious and error-prone task. Figure 2 shows that ChatGPT proposes to create a method named *setDiscoveryTimeInSeconds* in the class *Discovery*, which is totally wrong because it does not fit the causing change.

Our Hypothesis is that the LLM fails because our problem is more complex than simply repairing a code error. It must understand the original impacting metamodel change traced to the code error, as well as the abstraction gap between the two artefacts of metamodels and code. After improving the prompt, ChatGPT succeeded to give the right resolution as show in Figure 3. Our vision is that this contextual rich information must be injected in the prompt. Thus, the quality of the prompt is a key for the LLM to solve this problem of metamodels and code co-evolution.

The next section presents our contribution for a contextualized information rich prompts-based co-evolution of metamodel and code using LLMs.



Considering that the method **getIterations** is generated from the reference Iterations, and that the attribute **discoveryTimeInSeconds** is moved from the class **Discovery** to the class **DiscoveryIteration** through the reference **Iterations**; Coevolve this code : discovery.setDiscoveryTimeInSeconds(new Double(getValue(lastLine2, Statistics.SINCEBEGIN)) / Report.MINUTE_MS_RANGE);

```
discovery.getIterations.get(0).setDiscoveryTimeInSeconds(new Double(getValue
  (lastLine2, Statistics.SINCEBEGIN)) / Report.MINUTE_MS_RANGE);
```

Figure 3 ChatGPT improved answer with the enriched prompt with contextual information.

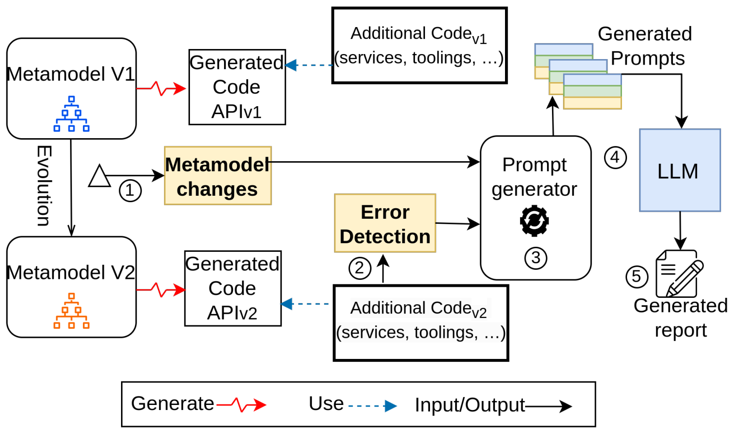


Figure 4 Overall Approach for Prompt-based co-evolution.

3. Prompt-Based Approach

This section introduces our approach to generate the prompts needed for the code co-evolution. It first gives an overview of the approach. Then details the structure of the generated prompts, before to detail each part of it and how it is generated. Finally, it describes our prototype implementation.

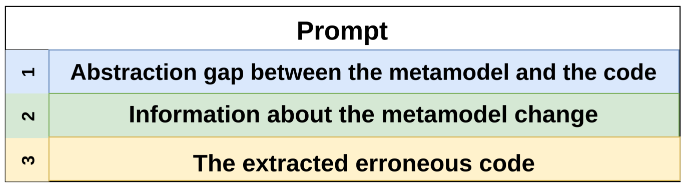
3.1. Overview

Figure 4 shows the overall steps of our approach. We start by retrieving the list of changes that describe the evolution between the old metamodel and the new metamodel ①.

When the metamodel evolves, the code API is regenerated, therefore the additional code is broken. The additional code is then parsed to collect the list of errors ②. The list of changes and the list of errors are the inputs of our prompt generator. The goal is to generate a prompt for each error, including sufficient information about the change and the error itself ③. Each generated prompt is used to request ChatGPT to give a correction for the concerned error ④. A global report is generated for the additional code to allow the developer to have a visual output about the generated prompts and the answers of the LLM ⑤. Algorithm 1 further depicts the overall method of co-evolution based on a given LLM. After we parse the project, we retrieve the list of errors per class (Lines 2-3). Then for each error, we generate the prompt (Lines 4-5) and call the LLM and record its co-evolution response for analysis (Lines 6-7).

Algorithm 1: ChatGPT Co-evolution

```
  Data: EcoreModelingProject, changesList
1 javaClasses ← Parse(EcoreModelingProject)
2 for ( jc ∈ javaClasses) do
3     errorsList ← getErrors(jc)
4     for (error : errorsList) do
5         prompt ← promptGenerator(error, changesList, jc)
6         coevolutionResponse ← callLLM(prompt)
7         addToReport(error, prompt, coevolutionResponse)
8     end
9 end
```



Prompt	
1	Abstraction gap between the metamodel and the code
2	Information about the metamodel change
3	The extracted erroneous code

Figure 5 Generated Prompt Structure.

3.2. Generated Prompt Structure

Figure 5 shows the overall structure of the envisioned prompts we will generate in order to co-evolve the code errors. In fact, the rationale behind the prompt structure is that our problem does not only concern the code errors to repair, but it is also related to the use of the code generated from the metamodels and their changes. Therefore, to contextualize our problem, we must explain : 1) what is the code generated from the metamodels, 2) what is the impacting metamodel change, and 3) what is the impacted code error to co-evolve. Concerning the prompt prefix we use "Co-evolve this code" to ask ChatGPT for one co-evolution. Next subsections detail the different parts of the prompt structure.

3.3. Abstraction Gap Between Metamodels and Code

One main distinction from simply repairing code errors is the interplay between the metamodels and the additional code through the generated code from the metamodels. This is due to the gap in abstraction in between metamodels and generated code. In fact, for each metamodel various code elements are generated for each metamodel element and with different patterns. Table 1 classifies and provides illustrative examples for the different generated code elements from each metamodel element, namely metaclass, attribute/reference, and method. For example, take the case of a metaclass[1]. EMF generates a corresponding interface and a class implementation, a *createClass()* method in the factory class, three literals (*i.e.,* constants) for the class and an accessor method in the package class, and a corresponding to create adapter method. For the attribute case, EMF generates the signature and implementation of a getter and a setter, an accessor, and a literal. This classification is essential to match the different errors with the right used generated code elements to explicit the abstraction gap. If a class is changed, all its generated elements are impacted and their usages will be er-

[1] For simplicity, we refer to a metaclass by simply a class in the remaining of the paper.

roneous. For example, if a class is renamed, every invocation of the method "get+className" will be erroneous and must be co-evolved. Thus, we consider the abstraction gap as the first contextual information we inject in the prompts for the LLM to co-evolve the code errors.

3.4. Metamodel Evolution Changes

A metamodel represents a high abstraction level of a domain. Like any software artefact, metamodels evolve to meet domain changing requirements (Mens 2008). Herrmannsdoerfer et al. (Herrmannsdoerfer et al. 2011) distinguish two types of metamodel changes: *atomic* and *complex*. Atomic changes are additions, removals, and updates of a metamodel element. Complex changes consist of a combination of atomic changes (Vermolen et al. 2011; Khelladi et al. 2015). For example, push property is a complex change where a property is pushed from a parent class to an inheriting child class. This is composed of two atomic changes: delete a property and add a property (Herrmannsdoerfer et al. 2011). Many approaches in the literature (Alter 2015; Williams et al. 2012; Cicchetti et al. 2009; Langer et al. 2013; Vermolen et al. 2011; Khelladi, Hebig, et al. 2016; Bettini et al. 2022) exist to detect metamodel changes between two versions. Particularly in our work, we use (Khelladi, Hebig, et al. 2016; Khelladi, Bendraou, & Gervais 2016) to extract the changes between two metamodel versions.

In practice, we focus on the impacting metamodel changes that will require co-evolution of the code and not on the non-impacting changes. For example, an add change of a class does not require co-evolution. However, a delete change or a change of type will impact the code that must be co-evolved. The list of impacting metamodel changes (Iovino et al. 2012; Cicchetti et al. 2009) we consider in the prompts is as follows: *1)* Delete property[2] p in a class `C`, *2)* Delete class `C`, *3)* Rename element e in a class `C`, *4)* Generalize property p multiplicity from a single value to multiple values in a class `C`, *5)* Move property p from class `Source` to `Target` through a reference *ref*, *6)* Extract class of properties $p_1, ..., p_n$ from `Source` to `Target` through a reference *ref*, *7)* Push property p from super class `Super` to sub classes $\texttt{Sub}_1$,...,$\texttt{Sub}_n$, *8)* Inline class `Source` to `Target` with properties $p_1, ..., p_n$, and *9)* Change property p type from `S` to `T` in a class `C`.

Thus, we consider these definitions of metamodel changes as the second contextual information we inject in the prompts for the LLM to co-evolve the code errors.

3.5. Extracted Code Errors

Now that we have two main ingredients needed for the generation of the prompts. We only require the erroneous code to be co-evolved.

To do so, we parse the code (i.e., *compilation units*) to access the Abstract Syntax Trees (ASTs) and retrieve the code errors. Each error contains the necessary information to locate the exact impacted AST node in the parsed global AST (*i.e.,* char start and end) and to process it (*i.e.,* message). After that, we simply extract the sub-AST corresponding to the code containing the error. We consider three possible situations, namely 1) if the error is in a method, we extract the whole method, 2) if the error is in the imports, we extract the list of imports, 3) if the error is in the class definition or the fields, we extract it without the class's methods. This constitutes the final part of the contextual information we inject in the prompts for the LLM to co-evolve the code errors. Note that we simply specify in the prompt before the code the order to *"Co-evolve this code: "*.

[2] Property refers to Attribute, Reference, and Method.

	Prompt
1	The method **setDiscoveryTimeInSeconds** is generated from The attribute **discoveryTimeInSeconds**
2	The attribute **discoveryTimeInSeconds** is moved from the class **Discovery** to the class **DiscoveryIteration** through the reference **Iterations**
3	Co-evolve this code : discovery.**setDiscoveryTimeInSeconds**(new Double(getValue(lastLine2, Statistics.SINCEBEGIN)) / Report.MINUTE_MS_RANGE);

Figure 6 Move attribute prompt example.

3.6. Prompt Generation

Algorithm 2 allows generating prompts following the specified structure in Figure 5. It first finds the ASTNode corresponding to the error in the code (Line 1). Then, it iterates over the list of metamodel changes to match the error node with the code usage to identify the impacted abstraction gap (Lines 6-8). After that, it summarizes the impacting metamodel change (Line 11). Finally, it extracts the erroneous code (Line 21) and puts together all three contextual information into one generated prompt (Lines 22-25). Figure 6 shows an example of the generated prompt for the error in Listing 2 (Line 4) due to the move of property *DiscoveryTimeInSeconds* in the metamodel.

3.7. Prototype Implementation

We implemented our solution as an eclipse Java plugin handling Ecore/EMF metamodels and their Java code. To retrieve the list of errors, we used JDT eclipse plugin [3]. To launch our tool, we added a command to the context menu when selecting a java project. Generated prompts are sent to ChatGPT (see next section 4.1), more specifically, its OpenAI API endpoint "https://api.openai.com/v1/chat/completions". We used Java JSON package to send our prompts and to receive ChatGPT responses. The error location, the corresponding generated prompt and ChatGPT responses are parsed in CSV file to have a visible results and to keep the history of proposed the co-evolutions. Moreover, quick fixes (our baseline) are called using *org.eclipse.jdt.ui.text.java.IQuickAssistProcessor* and *org.eclipse.jdt.ui.text.java.IJavaCompletionProposal*.

4. Methodology

This section describes our methodology for empirically assessing the capabilities of ChatGPT in addressing the problem of metamodel and code co-evolution. It first describes the selected LLM and the followed evaluation process. Then, it presents our research questions and the data set.

[3] Eclipse Java development tools (JDT): https://www.eclipse.org/jdt/core/

Table 1 Classification of the different patterns of the generated code element from the metamodel elements. [Examples illustrated for a metaclass *Rule*, property *Status*, and method *Execute()*]

Metamodel element type	**Generated code elements**	**Pattern of the generated code elements**	**Examples**
Metaclass	Interface	"MetaClassName"	*Rule*
	createClass() (in metamodelFactory class)	"create"+"MetaClassName"()	*createRule()*
	Literals of the class	"META_CLASS_NAME" "META_CLASS_NAME"+"_"+ "FEATURE_COUNT" "META_CLASS_NAME"+ "_"+"OPERATION_COUNT"	*RULE*, *RULE_FEATURE_COUNT*, *RULE_OPERATION_COUNT*
	Accessor of Meta objects (in metamodelPackage class)	"get"+"MetaClassName"()	*getRule()*
	Class implementation	"MetaClassNameImpl"	*RuleImpl*
	Adapter	"create"+"MetaClassName"+"Adapter"	*createRuleAdapter()*
Attribute (Same for a Reference)	Signature of getters and setters	"get"+"AttributeName"(), "set"+"AttributeName"()	*getStatus()*, *setStatus()*
	Accessor of Meta objects	"get"+"MetaClassName"+"_"+"AttributeName"()	*getRule_Status()*
	Literal	"META_CLASS_NAME"+"__"+"ATTRIBUTE_NAME"	*RULE__STATUS*
	Implementation of getters and setters	"get"+"AttributeName"(), "set"+"AttributeName"()	getStatus(), setStatus()
Method	Declaration of the method	"methodName"()	*Execute()*
	Accessor of meta objects	"get"+"MetaClass"+"__"+"MethodName"()	*getRule__Execute()*
	Literal	"META_CLASS_NAME"+"__"+"METHOD_NAME"	*RULE___EXECUTE*
	Implementation of the method	"methodName"()	*Execute()*

4.1. Selected LLM

We chose to use ChatGPT *GPT-3.5-turbo* in chat mode. It currently points to *gpt-3.5-turbo-0613* released in June 2023. We opted for this model because of four factors. The first one is that prompt content is only textual, we don't need to inject audio or image content. The second one is its capacity to generate good answers for requests about code and models generation (Nathalia et al. 2023; Yetiştiren et al. 2023; Guo et al. 2023; Fu et al. 2023; Kabir et al. 2023; Chaaben et al. 2023; Cámara et al. 2023). The third factor is that it was the latest API version that is accessible, *gpt-4* API still not accessible for us. The last one is related to the high popularity of ChatGPT as a tool. It has more than 100 million users, and its website saw more than 1.7 billion visitors in the last three months, with Software and Software Development as visitors' top category[4].

4.2. Evaluation Process

First, as we aim to query ChatGPT to co-evolve the erroneous code due to metamodel evolution, we need to provoke the errors in the code. To do so, we replace the original metamodel by the evolved metamodel. Then, we regenerate the code API with EMF. This will cause errors in the additional code that must co-evolve. After that, we must map the errors with the causing metamodel change before to generate a prompt with all appropriate information to be able to co-evolve the code errors. We then rely on the OpenAI API to query ChatGPT before to analyze its results. Finally, we measure the correctness of ChatGPT co-evolution by comparing its co-evolution with the manually co-evolved version by developers. Note that the comparison is processed manually by authors. This allows us to measure the *correctness* reached by ChatGPT . Correctness varies from 0 to 1, i.e., 0% to 100% and is defined as follows:

$$Correctness = \frac{LLMCoevolutions \cap ManualCoevolutions}{ManualCoevolutions}$$

We chose the structure shown in Figure 5 since it contains the contextual information needed for our problem of metamodels and code co-evolution. This structure was built after few manual naive attempts with ChatGPT , starting from a minimum context (as shown in the Motivating Example of Figure 2 that failed)

[4] https://www.similarweb.com/fr/website/chat.openai.com/#demographics

Algorithm 2: Prompt Generator Algorithm

```
Data: error, changesList, javaClass
errorNode ← findErrorAstNode(javaClass, error)
found ← false
while (change ∈ changesList & ¬found) do
    switch change do
        case RenameClass do
            if (errornode.name="get"+change.name ) then
                found ← true
                abstractionGap="The method "+
                  errornode.name+" is generated from the
                  metaclass "+ change.name
            else if ... then
                /*treat other abstraction gaps*/
            changeInfo= "The metaclass "+
              change.oldName+" is renamed to
              "+change.newName
        end
        case RenameProperty do
            ...
        end
        case DeleteProperty do
            ...
        end
    end
end
codeError ← extractCodeError(errornode)
prompt.add(abstractionGap)
prompt.add(changeInfo)
prompt.add(codeError)
return prompt
```

and by enriching the structure with more context information. However, we do not claim its completeness in terms of needed information or if it is the best structure. Other variants can lead to different results. To investigate more this choice, we generated three more variations of this structure to observe its effect on the results. Table 2 contains each variation and its corresponding explanation. Since our prompt contains three parts, we can change the order of these parts (Order Change operator) and the size of the erroneous code we put in the prompt (Minimal Code operator). Finally, rather than asking for a single co-evolution solution, we ask for alternative ones (Alternative Answers operator) in the prompt prefix. To stress out our evaluation, we conducted 5 runs separated by almost a day, the time we needed for one run to check manually the results of each project, which means that each generated prompt is proposed to ChatGPT five times. This aims to check whether ChatGPT gives a same or different answer, hence, assess its robustness. Finally, we compare ChatGPT to a baseline, namely the IDE quick fixes that are provided to repair the code errors. Note that we do not take as a baseline ChatGPT with prompts that only contain the code error, since as shown in Section 2 and Figure 2, it does not work.

4.3. Research Questions

To assess the capabilities of ChatGPT in the co-evolution of code, we set the following research questions.

RQ1 To what extent can ChatGPT co-evolve the code with evolved metamodels? This aims to assess the ability of ChatGPT to give correct resolutions to co-evolve the code according to the metamodel changes.

RQ2 How does varying the temperature hyperparameter affect the output of the co-evolution? The temperature hyperparameter controls the creativity of the language model. This question aims to assess the capability of ChatGPT to co-evolve the erroneous code when given less or more creativity in the generation of the solution.

RQ3 How does varying the prompt structure affect the output of the co-evolution? This aims to assess the quality improvement of the co-evolutions due to the prompts' variations.

RQ4 How does ChatGPT proposed co-evolution compare to the quick fix baseline? As quick fixes are provided by default in an IDE to repair the code errors, this question aims to assess which method outperforms the other in the task of code co-evolution with evolving metamodels.

4.4. Data set

This section presents the used data set in our empirical study, to be found in the attached supplementary material[5].

We chose Eclipse Modeling Framework (EMF) platform as technological space, which allows us to build modeling tools and applications based on Ecore metamodels (Steinberg et al. 2008). First, we aimed at selecting metamodels with meaningful real evolutions that do not consist in only deleting metamodel elements, but rather including complex evolution changes (see subsection 3.4).

This selection criterion resulted in seven Java projects from three case studies of three different language implementations in Eclipse, namely OCL (MDT 2015b), Papyrus (MDT 2015c), and Modisco (MDT 2015a) with their versions that was manually co-evolved by developers, that represent our ground of truth. OCL is a standard language defined by the Object Management Group (OMG) to specify First-order logic constraints. Papyrus is an industrial project led by CEA[6] to support model-based simulation, formal testing, safety analysis, etc. Modisco is an academic initiative to support development of model-driven tools, reverse engineering, verification, and transformation of existing software systems. Thus, the case studies cover standard, industrial, and academic languages that have evolved several times for more than 10 years of continuous development period.

Table 3 presents the details on the selected case studies, in particular about their metamodels and the occurred changes during evolution. The total of applied metamodel changes was 330 atomic changes, including 19 complex changes in the three metamodels. Table 4 presents the details on the size of the seven projects and code of the original versions that we co-evolve in addition to the number of errors after the metamodels evolution.

5. Results

This section now presents our results answering each RQ.

[5] https://figshare.com/s/bf35039892799c0e6f34
[6] http://www-list.cea.fr/en/

Table 2 Variation operators of our original prompt (OP).

Variation Operators	Explanation
Order Change (OC)	We change the order between the three structured parts of the prompt. We start by describing the metamodel change before the abstraction gap.
Minimal Code (MC)	Instead of giving the whole method that contains the code error to co-evolve, we only give the instruction of code error.
Alternative Answers (AA)	Instead of asking the LLM to give one solution of co-evolution, we specifically ask for alternative ways to co-evolve the code error.

Table 3 Details of the metamodels and their evolutions.

Case study	Evolved metamodels	Versions	Atomic changes in the metamodel	Complex changes in the metamodel
OCL	Pivot.ecore in project ocl.examples.pivot	3.2.2 to 3.4.4	Deletes: 2 classes, 16 properties, 6 super types Renames: 1 class, 5 properties Property changes: 4 types; 2 multiplicities Adds: 25 classes, 121 properties, 36 super types	1 pull property 2 push properties
Modisco	Benchmark.ecore in project modisco.infra.discovery.benchmark	0.9.0 to 0.13.0	Deletes: 6 classes, 19 properties, 5 super types Renames: 5 properties Adds: 7 classes, 24 properties, 4 super types	4 moves property 6 pull property 1 extract class 1 extract super class
Papyrus	ExtendedTypes.ecore in project papyrus.infra.extendedtypes	0.9.0 to 1.1.0	Deletes: 10 properties, 2 super types Renames: 3 classes, 2 properies Adds: 8 classes, 9 properties, 8 super types	2 pull property 1 push property 1 extract super class

5.1. RQ1: Assessing the ability of ChatGPT to give correct code co-evolutions

To assess the ability of ChatGPT to give correct code co-evolutions, we ran over the errors caused by the metamodel changes and generated 266 prompts that we submitted to ChatGPT . Note that we ran our original prompts asking for a single co-evolution of the code error. We further set a fixed temperature hyperparameter to 0.2. A value of 0 restricts the generation of a solution towards a more deterministic one and the more it is higher the more creative the model gets in generating a different solution. Thus, we chose 0.2 to allow only little creativity while restricting the model to not get a different answer for the same prompts in different executions, since we ask for a single solution. Among the 266 generated prompts, ChatGPT gave, on average, a correct code co-evolution in 88.7% of the time, varying from 75% to 100%. When taking only on complex changes into consideration, correctness improves on average from 88.7% to 95.2%, *i.e.,* ChatGPT performs better for complex changes. Results show a promising ability of ChatGPT to actually co-evolve code with evolving metamodels when given the right contextual information in the prompts, rather than simply the errors and their messages.

RQ_1 insights: Results confirm that ChatGPT is able at 88.7% to correctly co-evolve code due to metamodel evolution thanks to the information on the abstraction gap and the impacting metamodel change. It also performs better on complex changes.

5.2. RQ2: Studying the impact of temperature variation on the output co-evolutions

The temperature hyperparameter controls the creativity of the language model. The temperature hyperparameter in ChatGPT can be set between 0 and 2. Yet, it is suggested to only set it between 0 and 1 in the documentation[7]. Thus, to assess the effect of the temperature in co-evolving the code, we will vary it on 0, 0.2, 0.5, 0.8, and 1.0. Table 5 shows the obtained results.

Overall, we observe that as the temperature increases, the correctness of code co-evolutions suggested by ChatGPT decreases. Notably, the correctness improves when the temperature is increased from 0 to 0.2. However, it subsequently degrades with

[7] https://platform.openai.com/docs/api-reference/chat

Table 4 Details of the projects and their caused errors by the metamodels' evolution.

Evolved metamodels	Projects to co-evolve in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of LOC	N^o of Impacted classes	N^o of total errors
OCL Pivot.ecore	$[P1]$ ocl.examples.xtext.base	12	181	17599	10	29
Modisco Benchmark.ecore	$[P2]$ modisco.infra.discovery.benchmark	3	28	2333	1	6
	$[P3]$ gmt.modisco.java.discoverer.benchmark	8	21	1947	4	30
	$[P4]$ modisco.java.discoverer.benchmark	10	28	2794	9	56
	$[P5]$ modisco.java.discoverer.benchmark.javaBenchmark	3	16	1654	9	73
Papyrus ExtendedTypes.ecore	$[P6]$ papyrus.infra.extendedtypes	8	37	2057	8	44
	$[P7]$ papyrus.uml.tools.extendedtypes	7	15	725	7	28

Table 5 Measured correctness rate per temperature.

	Projects						
Temperature	P1	P2	P3	P4	P5	P6	P7
0.0	76%	62.5%	100%	80%	86%	93.7%	92.8%
0.2	**84%**	**75%**	**100%**	**82%**	**88%**	**95.8%**	**96.4%**
0.5	57%	50%	100%	58%	68%	87.5%	89.2%
0.8	50%	37%	0%	32%	46%	62.5%	85.7%
1.0	30%	29%	0%	26%	40%	68.7%	78.5%

Table 6 Measured correctness rate for different prompt variations. [↗ and ↘ are increase and decrease in correctness.]

	Projects						
Variations	P1	P2	P3	P4	P5	P6	P7
Original Prompt (OP)	**84%**	**75%**	**100%**	**82%**	**88%**	**95.8%**	**96.4%**
Order change (OC)	84%	66%↘	100%	74%↘	86%↘	95.8%	96.4%
Minimal Code (MC)	88.4%↗	87.5%↗	100%	86%↗	96%↗	97.9%↗	96.4%
Alternative Answers (AA)	84%	79%↗	100%	92%↗	92%↗	95.8%	96.4%

the further increase in temperature, up to 1. At temperatures 0.8 and 1, we observed the worst decrease in correctness. The best performance of ChatGPT is obtained at the temperature 0.2. Note that even with a temperature set to 0, which implies no creativity, ChatGPT yields results that are nearly as satisfactory as those obtained with a temperature of 0.2.

Moreover, when we repeated the same prompt for each temerature for 5 times, the five answers of ChatGPT were similar. ChatGPT gives almost the same proposition of co-evolution, sometimes it uses different terms to comment its answer. For example: *"// Remove the following line since DiscoveredProject is removed"* and *" // Code using DiscoveredProject should be removed"*. However, they are the same co-evolutions. Sometimes ChatGPT also uses intermediate variables to give the same co-evolution, for example: *"return aReport.generate()"* and *benchmarkModel=aReport.generate() return benchmarkModel"*. We believe that this result of obtaining the same co-evolutions over 5 runs shows the efficiency of the prompt template in Figure 5. By including the necessary information (i.e., abstraction gap, causing change information, and code error), it allows ChatGPT to narrow the scope of possibilities and to propose similar answer for each unique prompt in each run.

***RQ_2* insights:** Results show that lower temperature of 0 and 0.2 give better co-evolutions from ChatGPT with 0.2 being the best we observed. Interestingly, we obtained the same results over 5 different runs, which suggests the efficiency our prompt structure in narrowing the scope of possibilities of ChatGPT 's answers.

5.3. RQ3: Studying the impact of prompt structure variation on the output co-evolutions

In our approach, we propose a possible structure for the generated prompts (cf. Subsection 3.6). However, there is no assurance that this represents the best proposition. Therefore, we varied the structure that we proposed in three different ways, as shown in Table 2. Then we ran the three variations of the original prompts in order to assess the fluctuation and effect on the correctness of the proposed co-evolutions. Here we set the temperature to 0.2 based on results of RQ2.

Table 6 shows the obtained results. Overall, we observe slight increase and decrease compared to the original Prompts structure (**OP**). We observe that changing the order in the prompt by first describing the metamodel change (**OC**) decreases the correctness of the proposed ChatGPT co-evolutions. It decreased by -9% in P2, -8% in P4 and by -2% in P5. We observed that in particular when describing the abstraction gap with the generated elements for the metamodel deleted classes, ChatGPT assumes that the code classes still exist in the code and were not deleted. Thereby, altering sometimes the correctness of its proposed co-evolutions.

However, the two other prompt variations of giving a minimal code containing the error and asking for alternative solutions delivered better overall results. Surprisingly, on the one hand, giving only the code instruction containing the error (**MC**) gave the best results. It improved by +4.4% in P1, +12.5% in P2, +4% in P4, +8% in P5, and +2.1% in P6. Our observation is that this improvement is sometimes due to the inability of ChatGPT to find the impacted code element and co-evolve it within complex and long methods. On the other hand, when asking for alternatives, it improved only by +4% in P2, +10% in P4, and +4% in P5. We observed that when ChatGPT is unable to find the correct resolution with (**OP**) prompts, it is unlikely to find the right one when asking for alternatives (**AA**). Only in few cases it could find the correct co-evolutions. The generation of prompts and saving the results took on average from about 10 seconds per prompt for the Original Prompts to 84 seconds per prompt in the case of Alternative Answers (AA) variation prompts. Finally, when focusing only on complex changes, correctness improves on average from 88.7% to 97.6% for **(MC)** and **(AA)**. This implies that, overall, ChatGPT also performs better for complex changes when varying the prompts. This can be explained by the fact that complex changes provide much more context information that guide better ChatGPT to give better responses.

> ***RQ_3* insights:** Results show an improvement with two variants out of the three we explored. However, gains are not significant compared to our original structure of the prompt. Variants also perform better on complex changes.

5.4. RQ4: Comparison with quick fixes as baseline

To compare to a baseline the obtained results of proposed co-evolutions with our generated prompts, we checked what is the best quick fix an IDE proposes for each error. Algorithm 3 shows the followed steps to run the quick fixes on our projects automatically. It iterates over the java classes and for each error, we automatically apply the top quick fix suggested by the IDE. It stops when all the errors disappear or if the remaining errors have no possible quick fixes purposed by the IDE.

In Table 7, we present the percentage of errors that quick fixes eliminated for each project. While the quick fixes eliminated from 41% to 100% errors. We use the term eliminated instead of correct co-evolution because no quick fix was applied as expected to the manual developers' co-evolutions. In other words, the correctness rate of automatic quick fixes are equal to 0 and are not suited for the task of metamodels and code co-evolution. For example, concerning errors caused by class or property deletion from the metamodel, renaming a class or an attribute, moving, pushing or pulling attributes or methods from a class to another, the quick fixes proposed to create them back in their old containers. This is in contradiction of the applied metamodel changes and the code co-evolution.

For errors caused by changing a variable's type, the quick fixes always proposed to add a cast with a wrong type.

Algorithm 3: Quick fixes for coevolution

```
   Data: EcoreModelingProject
 1 javaClasses ← Parse(EcoreModelingProject)
 2 for ( jc ∈ javaClasses) do
 3     errorsList ← getErrors(jc)
 4     while (!errorsList.isEmpty() & hasQuickFix) do
 5         error ← errorsList.next()
 6         if error.hasQuickickFix() then
 7             useQuickFixes(error)
 8             refreshJavaClass(jc)
 9             refreshErrorsList(jc, errorsList)
10     end
11 end
```

Table 7 Number of applied Quick Fixes for each project and per evolved metamodel.

Evolved metamodels	Co-evolved projects	% of eliminated errors
OCL Pivot.ecore	[P1]	41%
Modisco Benchmark.ecore	[P2]	100%
	[P3]	100%
	[P4]	83%
	[P5]	67%
Papyrus ExtendedTypes.ecore	[P6]	69%
	[P7]	93%

Similarly, as naive prompts with only the code errors and their messages, quick fixes do not take in consideration the knowledge of the abstraction gap and the information contained in its causing metamodel changes. For example, the quick fix `create the missing method m()` is applied no matter the metamodel change (i.e., deletion, moving, pulling, or pushing changes) and no matter the metamodel element it was generated from. Our approach of generated prompts takes into account the context of the impacted code, the abstraction gap, and the causing metamodel change thanks to the prompt template that we designed (cf. Table 5).

> ***RQ_4* insights:** Results show that ChatGPT with our generated prompts completely outperforms the quick fixes in correctly co-evolving the code.

5.5. Threats to Validity

This section discusses threats to validity w.r.t. Wohlin et al. (Wohlin et al. 2012).

5.5.1. Internal Validity. A first internal threat is that we only varied the temperature hyperparameter over ChatGPT API. The documentation suggests not to modify top_p and temperature at the same time, so we chose to let the default value of top_p = 1. Moreover, to measure the correctness, we analyzed the developers' manual co-evolution. To mitigate the risk of misidentifying a manual co-evolution, for each impacted code error, we mapped it in the co-evolved class version. If we did not find it, we further searched in other classes in case the original impacted part of the code was moved into another class. Thus, our objective was to reduce the risk of missing any mappings between an error in the original code and its co-evolved version. Moreover, as our co-evolution relies on the quality of detected metamodel changes. We also validated each detected change by checking whether it occurred between the original and evolved metamodels. This alleviates the risk of relying on an incorrect metamodel change that would degrade the generated prompts and lead to wrong co-evolution from ChatGPT .

5.5.2. External Validity. We implemented and ran the empirical study for EMF and Ecore metamodels and Java code. Note that the choice of Java is imposed by EMF and no other languages are supported. Thus, we naturally do not generalize to other languages. We also relied only on ChatGPT and GPT-3.5-turbo released in june 2023. Therefore, we cannot generalize our approach to other LLMs and other future versions of ChatGPT . It is also unclear how our findings would transfer to other benchmarks other than EMF Ecore metamodels and java code. Further experiments are necessary in future to get more insights and- before any generalization claims.

5.5.3. Conclusion Validity. Our empirical study show promising results with ChatGPT being able to generate correct code co-evolution when given the necessary contextual information. It showed to be useful, with an average of 88.7% correctness (from 75% to 100%). The results also show the benefit over relying on quick fixes. Nonetheless, even though we evaluated it on real evolved projects, we must evaluate it on more case studies to have more insights and statistical evidence.

5.6. Discussion and Limitations

The rationale behind the prompt structure, which an important part in our empirical study, is that our problem concerns the use of the code generated from the metamodels and their changes and not only error repair. Moreover, our results show that ChatGPT can co-evolve the code correctly by setting lower temperature, especially in case of complex metamodel changes. Furthermore, repeating the experimentation five time has led to the same results shown in Table 5 and Table 6, which confirms the robustness of ChatGPT in the code co-evolution task and that his capability is not due to randomness. Finally, we handled a single metamodel with changes that are independent between them. Treating the case of multiple metamodel and the case of interdependent changes would need setting an order of priority between them, which we left for a future work.

6. Related Work

This section discusses close related work that focuses on empirically evaluating LLMs on code and MDE artifacts.

In literature, several studies delve into examining how the evolution of the metamodel influences the generated artifacts. In particular (Kessentini, Wimmer, & Sahraoui 2018; Kessentini et al. 2019; Cicchetti et al. 2008; Herrmannsdoerfer et al. 2009; Garcés et al. 2009; Wachsmuth 2007) have focused on the co-evolution of metamodel and models, (Batot et al. 2017; Khelladi et al. 2017; Correa & Werner 2007; Kusel, Etzlstorfer, Kapsammer, Retschitzegger, Schoenboeck, et al. 2015) studied the metamodel and constraints co-evolution, and other on the metamodel and transformations co-evolution (Kessentini, Sahraoui, & Wimmer 2018; Khelladi et al. 2018; Garcés et al. 2014; García et al. 2013; Kusel, Etzlstorfer, Kapsammer, Retschitzegger, Schwinger, & Schonbock 2015). Other approaches focused on the model consistency repair (e.g., (Kretschmer et al. 2017; Kretschmer, Khelladi, Lopez-Herrejon, & Egyed 2021; Kretschmer, Khelladi, & Egyed 2021; Macedo et al. 2013; Pinna Puissant et al. 2015)). However, only a few works addressed the challenge of metamodels and code co-evolution. In particular, (Riedl-Ehrenleitner et al. 2014; Kanakis et al. 2019; Pham et al. 2017; Jongeling et al. 2020, 2022; Zaheri et al. 2021) focused on consistency checking between models and code, but not on its co-evolution. Other works (Yu et al. 2012; Khelladi, Combemale, Acher, Barais, & Jézéquel 2020) proposed to co-evolve the code. However, the former handles only the generated code API, it does not handle additional code and aims to maintain bidirectional traceability between the model and the code API. The latter supports a semi-automatic co-evolution requiring developers' intervention.

Furthermore, several works evaluated the use of LLMs in software engineering tasks. Early studies on Copilot focus on the exploration of the security of the generated code (Pearce et al. 2022), comparison of the performances of Copilot with mutation-based code generation techniques (Sobania et al. 2022), and the impact on productivity and the usefulness of Copilot for developers (Ziegler et al. 2022; Vaithilingam et al. 2022). Nguyen et al. (Nguyen & Nadi 2022) performed an early empirical study on the performance and understandability of Copilot generated code on 34 problems from Leetcode. Doderlein et al. (Döderlein et al. 2022) extended the study of Nguyen et al. (Nguyen & Nadi 2022) and run an empirical study on the effect of varying temperature and prompts on the generated code with Copilot and Codex. They used a total of 446 questions to solve from Leetcode and Human Eval data set. Nathalia et al. (Nathalia et al. 2023) evaluated the Performance and Efficiency of ChatGPT compared to beginners and experts software engineers. Yeticstiren et al. (Yetiştiren et al. 2023) compared the code quality generated from Copilot, CodeWhisperer, and ChatGPT, showing an advantage for ChatGPT in generating correct solutions. Guo et al. (Guo et al. 2023) ran an empirical study on ChatGPT and its capabilities in refining code based on code reviews. Fu et al. (Fu et al. 2023) also evaluated ChatGPT and its ability to detect, classify, and repair vulnerable code. Finally, Kabir et al. (Kabir et al. 2023) evaluated ChatGPT ability to generate code and to maintain it by improving it based on a new feature description to add in the code. All the above studies focused on either evaluating the ability of LLMs to generate qualitative code, refining it, repairing it if vulnerable, or augmenting it. However, none of them specifically explored the task of code co-evolution.

Moreover, other studies focused on evaluating LLMs in MDE activities. Chen et al. (Chen et al. 2023) propose a comparative study between GPT-3.5 and GPT-4 in automatically generating domain models. This work shows that GPT-4 has better modeling results. Chaaben et al. (Chaaben et al. 2023) showed how using few-shot learning with GTP3 model can be effective in model completion and in other modeling activities. Camara et al. (Cámara et al. 2023) further assessed how good ChatGPT is in generated UML models. Finally, Abukhalaf (Abukhalaf et al. 2023) run an empirical study on the quality of generated OCL constraints with Codex. However, these studies also focused on the ability of LLMs to generate MDE artifacts, such as models and constraints, but not on their co-evolution. Only Fu et al. (Fu et al. 2023) looked at repairing vulnerable code with ChatGPT. Jiang et al. (Jiang et al. 2023) proposed self-augmented code generation framework based on LLMs called SelfEvolve. SelfEvolve allows generating code and keep correcting it iteratively with the LLM. Zhang et al. (Zhang et al. 2023) proposed Codeditor, an LLM based tool for code co-evolution between different programming languages. It learns code evolutions as edit sequences and then uses LLMs for multilingual translation.

To the best of our knowledge, no study investigated the ability of LLMs in the MDE problem of code co-evolution when metamodels evolve. We empirically evaluated how effective is ChatGPT in solving this co-evolution problem.

7. Conclusion

Co-evolving the impact of metamodel evolution on the code is costly and yet challenging. In this paper, we proposed a prompt-based approach for metamodel and code co-evolution. This approach relies on designing and generating a rich contextual information that we inject in the prompts, namely the abstraction gap knowledge, the metamodel changes information and the impacted code to be co-evolved. We evaluated our approach on seven EMF projects and three evolved metamodels from three different Eclipse EMF implementations of OCL, Modisco and Papyrus, with a total of 5320 generated prompts. Results show that on average ChatGPT has successfully proposed 88.7% of correct co-evolutions with our original generated prompts. We evaluated then the impact of the temperature variation on the proposed co-evolutions. We found that ChatGPT gives better responses with lower temperature values of 0 and 0.2. Moreover, when experimenting other variations of the structure of generated prompts, we observed that there was improvement in two variations. The first one is giving the minimum impacted code to co-evolve in the prompt, and the second one is requiring alternative answers for the co-evolution. However, varying the prompt by changing the order of the contextual information degraded a little the quality of the proposed co-evolutions. Finally, we compared our approach with the quick fixes of the IDE as a baseline. Results show that our approach significantly outperforms the quick fixes that do not take into account the context of the abstraction gap and the metamodel changes.

As future work, we intend to evaluate or approach in other technological spaces than EMF, such as OpenAPI and the challenge of the API evolution impact on clients' code. After that, we plan to transform our approach into a DSL-based approach with a graphical user interface for the output report instead of a CSV file. To facilitate prompt generation and enhance the option of prompt variation, a DSL would be a viable solution. We also plan to replicate our empirical study with other LLMs and other contexts of co-evolution (e.g., between code and test (Le Dilavrec et al. 2021)). Another actionable element is to investigate the mining of contextual information from Software Engineering tasks to enrich the prompts, and then improve baseline results Finally, since our contributions focus on empirically studying the use of ChatGPT in metamodel and code co-evolution, we plan to implement an alternative of the quick fix engine in the Eclipse IDE based on our generated prompt structure. Integrating our prompt-based metamodel and code co-evolution in the IDE will have a direct impact in helping MDE developers and language engineers.

Acknowledgments

The research leading to these results has received funding from the *ANR* agency under grant *ANR JCJC MC-EVO*^{2} *204687*.

References

Abukhalaf, S., Hamdaqa, M., & Khomh, F. (2023). On codex prompt engineering for OCL generation: An empirical study. In *2023 ieee/acm 20th international conference on mining software repositories (msr)* (p. 148-157).

Alter, S. (2015). Work system theory: A bridge between business and IT views of systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *9097*, 520–521. doi: 10.1007/978-3-319-19069-3

Batot, E., Kessentini, W., Sahraoui, H., & Famelis, M. (2017). Heuristic-based recommendation for metamodel—ocl coevolution. In *Acm/ieee 20th int. conference on model driven engineering languages and systems (models)* (pp. 210–220).

Bettini, L., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2022). An executable metamodel refactoring catalog. *Software and Systems Modeling*, *21*(5), 1689–1709.

Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information and Software Technology*, *56*(8), 1012–1032.

Bruneliere, H., Cabot, J., Jouault, F., & Madiot, F. (2010). Modisco: a generic and extensible framework for model driven reverse engineering. In *Ieee/acm international conference on automated software engineering* (pp. 173–174).

Cabot, J., & Gogolla, M. (2012). Object constraint language (ocl): a definitive guide. In *Formal methods for model-driven engineering* (pp. 58–90). Springer.

Cámara, J., Troya, J., Burgueño, L., & Vallecillo, A. (2023). On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml. *Software and Systems Modeling*, 1–13.

Chaaben, M. B., Burgueño, L., & Sahraoui, H. (2023). Towards using few-shot prompt learning for automating model completion. In *Ieee/acm 45th int. conf. on software engineering: New ideas and emerging results (icse-nier)* (pp. 7–12).

Chen, K., Yang, Y., Chen, B., López, J. A. H., Mussbacher, G., & Varró, D. (2023). Automated domain modeling with large language models: A comparative study. In *2023 acm/ieee 26th international conference on model driven engineering languages and systems (models)* (pp. 162–172).

Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In *2008 12th international ieee enterprise distributed object computing conference* (pp. 222–231).

Cicchetti, A., Di Ruscio, D., & Pierantonio, A. (2009). Managing dependent changes in coupled evolution. In *Int. conf. on theory and practice of model transformations* (pp. 35–51).

Clarisó, R., & Cabot, J. (2023). Model-driven prompt engineering. In *Acm/ieee 26th int. conference on model driven engineering languages and systems (models)* (pp. 47–54).

Correa, A., & Werner, C. (2007). Refactoring object constraint language specifications. *Software & Systems Modeling*, *6*(2), 113–138.

Döderlein, J.-B., Acher, M., Khelladi, D. E., & Combemale, B. (2022). Piloting copilot and codex: Hot temperature, cold prompts, or black magic? *arXiv preprint arXiv:2210.14699*.

Fu, M., Tantithamthavorn, C., Nguyen, V., & Le, T. (2023). Chatgpt for vulnerability detection, classification, and repair: How far are we? *arXiv preprint arXiv:2310.09810*.

Garcés, K., Jouault, F., Cointe, P., & Bézivin, J. (2009). Managing model adaptation by precise detection of metamodel changes. In *European conf. on model driven architecture-foundations and applications, ECMDA-FA* (pp. 34–49).

Garcés, K., Vara, J. M., Jouault, F., & Marcos, E. (2014). Adapting transformations to metamodel changes via external transformation composition. *Software & Systems Modeling*, *13*(2), 789–806.

García, J., Diaz, O., & Azanza, M. (2013). Model transformation co-evolution: A semi-automatic approach. In *Int. conf. software language engineering, SLE* (pp. 144–163).

Guo, Q., Cao, J., Xie, X., Liu, S., Li, X., Chen, B., & Peng, X. (2023). Exploring the potential of chatgpt in automated code refinement: An empirical study. *arXiv preprint arXiv:2309.08221*.

Herrmannsdoerfer, M., Benz, S., & Juergens, E. (2009). Cope-automating coupled evolution of metamodels and models. In *Ecoop* (Vol. 9, pp. 52–76).

Herrmannsdoerfer, M., Vermolen, S. D., & Wachsmuth, G. (2011). An extensive catalog of operators for the coupled evolution of metamodels and models. In *International conference software language engineering, SLE* (pp. 163–182).

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., ... Wang, H. (2023). Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*.

Iovino, L., Pierantonio, A., & Malavolta, I. (2012). On the impact significance of metamodel evolution in mde. *Journal of Object Technology*, *11*(3), 3–1.

Jiang, S., Wang, Y., & Wang, Y. (2023). Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*.

Jongeling, R., Fredriksson, J., Ciccozzi, F., Carlson, J., & Cicchetti, A. (2022). Structural consistency between a system model and its implementation: a design science study in industry. In *The european conference on modelling foundations and applications (ecmfa)*.

Jongeling, R., Fredriksson, J., Ciccozzi, F., Cicchetti, A., & Carlson, J. (2020). Towards consistency checking between a system model and its implementation. In *International conference on systems modelling and management* (pp. 30–39).

Kabir, M. M. A., Hassan, S. A., Wang, X., Wang, Y., Yu, H., & Meng, N. (2023). An empirical study of chatgpt-3.5 on question answering and code maintenance. *arXiv preprint arXiv:2310.02104*.

Kanakis, G., Khelladi, D. E., Fischer, S., Tröls, M., & Egyed, A. (2019). An empirical study on the impact of inconsistency feedback during model and code co-changing. *The Journal of Object Technology*, *18*(2), 10–1.

Kessentini, W., Sahraoui, H., & Wimmer, M. (2018). Automated co-evolution of metamodels and transformation rules: A search-based approach. In *International symposium on search based software engineering* (pp. 229–245).

Kessentini, W., Sahraoui, H., & Wimmer, M. (2019). Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology*, *106*, 49–67.

Kessentini, W., Wimmer, M., & Sahraoui, H. (2018). Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search. In *Proceedings of the 21th acm/ieee international conference on model driven engineering languages and systems* (pp. 101–111).

Khelladi, D. E., Bendraou, R., & Gervais, M.-P. (2016). Ad-room: a tool for automatic detection of refactorings in object-oriented models. In *The 38th international conference on software engineering companion* (pp. 617–620).

Khelladi, D. E., Bendraou, R., Hebig, R., & Gervais, M.-P. (2017). A semi-automatic maintenance and co-evolution of ocl constraints with (meta) model evolution. *Journal of Systems and Software*, *134*, 242–260.

Khelladi, D. E., Combemale, B., Acher, M., & Barais, O. (2020). On the power of abstraction: a model-driven co-evolution approach of software code. In *Proceedings of the acm/ieee 42nd international conference on software engineering: new ideas and emerging results* (pp. 85–88).

Khelladi, D. E., Combemale, B., Acher, M., Barais, O., & Jézéquel, J.-M. (2020). Co-evolving code with evolving metamodels. In *Proceedings of the acm/ieee 42nd international conference on software engineering* (pp. 1496–1508).

Khelladi, D. E., Hebig, R., Bendraou, R., Robin, J., & Gervais, M.-P. (2015). Detecting complex changes during metamodel evolution. In *Advanced information systems engineering (caise)* (pp. 263–278).

Khelladi, D. E., Hebig, R., Bendraou, R., Robin, J., & Gervais, M. P. (2016). Detecting complex changes and refactorings during (Meta)model evolution. *Information Systems*, *62*, 220–241.

Khelladi, D. E., Kretschmer, R., & Egyed, A. (2018). Change propagation-based and composition-based co-evolution of transformations with evolving metamodels. In *Proceedings of the 21th acm/ieee international conference on model driven engineering languages and systems* (pp. 404–414).

Kretschmer, R., Khelladi, D. E., Demuth, A., Lopez-Herrejon, R. E., & Egyed, A. (2017). From abstract to concrete repairs of model inconsistencies: An automated approach. In *Asia-pacific software engineering conf., APSEC* (pp. 456–465).

Kretschmer, R., Khelladi, D. E., & Egyed, A. (2021). Transforming abstract to concrete repairs with a generative approach of repair values. *Journal of Systems and Software*, *175*, 110889.

Kretschmer, R., Khelladi, D. E., Lopez-Herrejon, R. E., & Egyed, A. (2021). Consistent change propagation within models. *Software and Systems Modeling*, *20*, 539–555.

Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schoenboeck, J., Schwinger, W., & Wimmer, M. (2015). Systematic co-evolution of ocl expressions. In *11th APCCM 2015* (Vol. 27, p. 30).

Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., & Schonbock, J. (2015). Consistent co-evolution of models and transformations. In *Acm/ieee 18th models* (pp. 116–125).

Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., & Kappel, G. (2013). A posteriori operation detection in evolving software models. *Journal of Systems and Software*, *86*(2), 551–566.

Le Dilavrec, Q., Khelladi, D. E., Blouin, A., & Jézéquel, J.-M. (2021). Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions. In *Int. conf. on soft. maintenance and evolution, ICSME* (pp. 206–216).

Liu, C., Bao, X., Zhang, H., Zhang, N., Hu, H., Zhang, X., & Yan, M. (2023). Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*.

Macedo, N., Guimaraes, T., & Cunha, A. (2013). Model repair and transformation with echo. In *Ieee/acm int. conf. on automated software engineering (ase)* (pp. 694–697).

MDT. (2015a). *Model development tools. modisco.* http://www.eclipse.org/modeling/mdt/?project=modisco.

MDT. (2015b). *Model development tools. object constraints language (ocl).* http://www.eclipse.org/modeling/mdt/?project=ocl.

MDT. (2015c). *Model development tools. papyrus.* http://www.eclipse.org/modeling/mdt/?project=papyrus.

Mens, T. (2008). *Introduction and roadmap: History and challenges of software evolution*. Springer.

Nathalia, N., Paulo, A., & Donald, C. (2023). Artificial intelligence vs. software engineers: An empirical study on performance and efficiency using chatgpt. In *The 33rd annual international conference on computer science and software engineering* (pp. 24–33).

Nguyen, N., & Nadi, S. (2022). An empirical evaluation of github copilot's code suggestions. In *The 19th international conference on mining software repositories* (pp. 1–5).

Ozkaya, I. (2023). Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, *40*(3), 4-8. doi: 10.1109/MS.2023.3248401

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 ieee symposium on security and privacy (sp)* (pp. 754–768).

Pham, V. C., Radermacher, A., Gerard, S., & Li, S. (2017). Bidirectional mapping between architecture model and code for synchronization. In *Ieee international conference on software architecture, ICSA* (pp. 239–242).

Pinna Puissant, J., Van Der Straeten, R., & Mens, T. (2015). Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, *14*, 461–481.

Riedl-Ehrenleitner, M., Demuth, A., & Egyed, A. (2014). Towards model-and-code consistency checking. In *Annual computer software and applications conference* (pp. 85–90).

Sobania, D., Briesch, M., & Rothlauf, F. (2022). Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the genetic and evolutionary computation conference* (pp. 1019–1027).

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *Emf: eclipse modeling framework*. Pearson Education.

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems, extended abstracts* (pp. 1–7).

Vermolen, S. D., Wachsmuth, G., & Visser, E. (2011). Reconstructing complex metamodel evolution. In *International conference on software language engineering* (pp. 201–221).

Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *Ecoop* (Vol. 7, pp. 600–624).

Williams, J. R., Paige, R. F., & Polack, F. A. (2012). Searching for model migration strategies. In *Proceedings of the 6th international workshop on models and evolution* (pp. 39–44).

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*.

Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., & Montrieux, L. (2012). Maintaining invariant traceability through bidirectional transformations. In *International conference on software engineering (icse)* (pp. 540–550).

Zaheri, M., Famelis, M., & Syriani, E. (2021). Towards checking consistency-breaking updates between models and generated artifacts. In *Int. conf. on model driven engineering languages and systems companion (models-c)* (pp. 400–409).

Zhang, J., Nie, P., Li, J. J., & Gligoric, M. (2023). Multilingual code co-evolution using large language models. *arXiv preprint arXiv:2307.14991*.

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., . . . Aftandilian, E. (2022). Productivity assessment of neural code completion. In *Acm sigplan international symposium on machine programming* (pp. 21–29).