Software and Systems Modeling
https://doi.org/10.1007/s10270-026-01402-9

THEME SECTION PAPER

Leveraging LLMs to support co-evolution between definitions and instances of textual DSLs: a systematic evaluation

Weixing Zhang[1] · Bowen Jiang[1] · Yuhong Fu[2] · Anne Koziolek[1] · Regina Hebig[3] · Daniel Strüber[4,5]

Received: 12 February 2026 / Revised: 25 May 2026 / Accepted: 12 June 2026

Abstract

Software languages evolve over time for various reasons, such as the addition of new features. When the language's grammar definition evolves, textual instances that originally conformed to the grammar become outdated. For DSLs in a model-driven engineering context, there exists a plethora of techniques to co-evolve models with the evolving metamodel. However, these techniques are not geared to support DSLs with a textual grammar—applying them to textual language definitions and instances may lead to the loss of information from the original instances, such as layout information and comments, which are valuable for software comprehension and maintenance. This study systematically evaluates the potential of Large Language Model (LLM)-based solutions in achieving grammar and instance co-evolution for textual DSLs. By applying two advanced language models, Claude Sonnet 4.5 and GPT-5.2, and conducting ten experimental runs per case across ten case languages, we evaluate both the correctness of co-evolved instances and the preservation of human-oriented information such as comments and layout. Our results indicate high performance on small-scale cases ($\geq$94% precision and recall for instances with fewer than 20 lines requiring modification), but performance degraded with scale: Claude Sonnet 4.5 maintained 85% recall at 40 lines while GPT-5.2 showed greater sensitivity, failing entirely on the two largest instances. Instance scale also substantially impacts processing efficiency, with Claude's response time increasing nearly 18-fold for the largest case. In addition, we observe that grammar evolution complexity and deletion granularity impact performance more than change type alone, and prompt transferability across LLMs is limited. These findings identify the conditions under which LLM-based co-evolution succeeds and its scalability limitations, providing practitioners with insights into applicability and researchers with directions for addressing current limitations.

Keywords Co-evolution · Textual DSLs · Language definition · Instance · LLM

1 Introduction

Domain-specific languages (DSLs) are specialized languages designed to to describe and solve problems in a specific application domain. As domain knowledge evolves

Communicated by Riccardo Rubei, Antonio Cicchetti, and José Antonio López.

✉ Weixing Zhang
weixing.zhang@kit.edu

✉ Daniel Strüber
danstru@chalmers.se

Bowen Jiang
bowen.jiang@kit.edu

Yuhong Fu
yuhong.fu@adelaide.edu.au

Anne Koziolek
koziolek@kit.edu

Regina Hebig
regina.hebig@uni-rostock.de

[1] MCSE Group, Karlsruhe Institute of Technology, Karlsruhe, Germany

[2] Engineering and Information Technology, Adelaide University, Adelaide, Australia

[3] Institute of Computer Science, University of Rostock, Rostock, Germany

[4] Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

[5] Department of Software Science, Radboud University, Nijmegen, The Netherlands

Published online: 10 July 2026

and requirements change, DSLs often need to evolve accordingly [1]. For example, features may be added, and existing functionality may be adjusted, leading to a need to update the definition of the DSL, typically specified using grammars and/or meta-models, to introduce new language constructs and modify existing ones.

When the definition of a DSL evolves, existing instances face multiple challenges. First, instances may contain constructs that no longer conform to the new definition, requiring appropriate modification or replacement. Second, the new definition may introduce required language elements, necessitating corresponding additions to existing instances [2, 3]. These issues lead to a need for dedicated co-evolution approaches [4].

The model-driven engineering community has brought forward a plethora of available approaches for metamodel-instance co-evolution [4], but these works are generally focused on metamodel-based language definitions, usually in the context of graphical DSLs [4]. Textual DSLs are widely adopted in practice [5, 6] which can emulate the *look and feel* of familiar general-purpose languages and are easy to integrate with standard developer tools for versioning, differencing, and merging. Textual DSLs are developed in dedicated frameworks like *Xtext* [7], *langium* [8], and *textX* [9]. On a technical level, textual DSLs are defined through grammars and instantiated by textual instances.

Dedicated approaches to co-evolve textual instances are scarce. One possible way to address the co-evolution of textual instances is by using the available metamodel-based approaches. To that end, the original instance needs to be parsed into the form of a model and transformed back into textual form after the model is co-evolved. However, this approach leads to information loss: during the transformation process between textual instance and model, human-oriented information such as comments and formatting styles in the original instances cannot be retained [10]. While this information does not affect the formal semantics of a DSL instance, it serves a critical purpose during tasks such as code maintenance, debugging, and understanding design intent [11]. Hence, there is arguably a need to preserve such information during the co-evolution of instances.

In recent years, Large Language Models (LLMs) have demonstrated exceptional capabilities in code understanding, transformation, and generation [12]. These models do not only perform well at tasks that require an understanding of code structure, but also capture contextual information like comments. As such, they seem particularly well-suited to addressing the co-evolution problem for textual languages.

In this paper, we systematically evaluate the use of LLMs to support the co-evolution of grammar definitions and instances for textual DSLs. Given an original grammar, a textual instance conforming to it, and an evolved grammar, we leverage LLMs to automatically analyze the grammar differences and migrate the instance to conform to the evolved grammar while preserving human-oriented information. We focus on grammar definitions and instances developed using Xtext [7], a framework that is rooted in the Eclipse ecosystem and is particularly widely used in the MDE community. We harness our recently published dataset on Xtext-based language evolution cases retrieved from GitHub [13, 14], which allowed us to identify a collection of real-life cases in which automated support for co-evolution would be desirable. We employ two state-of-the-art Large Language Models—Claude Sonnet 4.5 and GPT-5.2—and conduct ten experimental runs per case language to account for non-determinism in LLM outputs. We address and answer the following research questions:

RQ1: How do instance size and grammar change extent affect the capability of LLMs to produce correct solutions when co-evolving textual instances? Answering this question allows us to identify the practical boundaries within which LLM-based co-evolution can be applied, and to determine when alternative approaches may be needed.

RQ2: How does the type of grammar evolution affect LLMs' capability in correctly co-evolving textual instances? Answering this question allows us to identify which categories of grammar changes LLMs handle well and which pose difficulties, informing practitioners about when to trust LLM-generated solutions.

RQ3: How do these factors affect LLMs' capability in preserving human-oriented information during DSL co-evolution? Answering this question allows us to understand under what conditions LLMs can preserve comments and formatting, which is the advantage of LLM-based approaches over traditional model-based co-evolution.

RQ4: To what extent do prompts transfer across different LLM generations? Answering this question allows us to understand whether a single prompt can work across different LLMs or whether LLM-specific adjustments are necessary.

Our results indicate that LLM-based co-evolution is promising, but its applicability depends on instance scale and grammar change complexity. Claude Sonnet 4.5 produced correct or near-correct evolved instances for most case languages, achieving high average precision and recall, whereas GPT-5.2 showed larger performance variation and failed on the largest or structurally more demanding cases. We also found that both models can preserve human-oriented information such as comments and formatting in smaller cases, but this capability becomes less reliable as instance size increases. Overall, the results suggest that LLM-based co-evolution can serve as a lightweight support mechanism for textual DSL evolution, but should be complemented by systematic validation when applied in practice.

The contribution of this paper is a systematic evaluation of LLM-based co-evolution for textual DSLs across

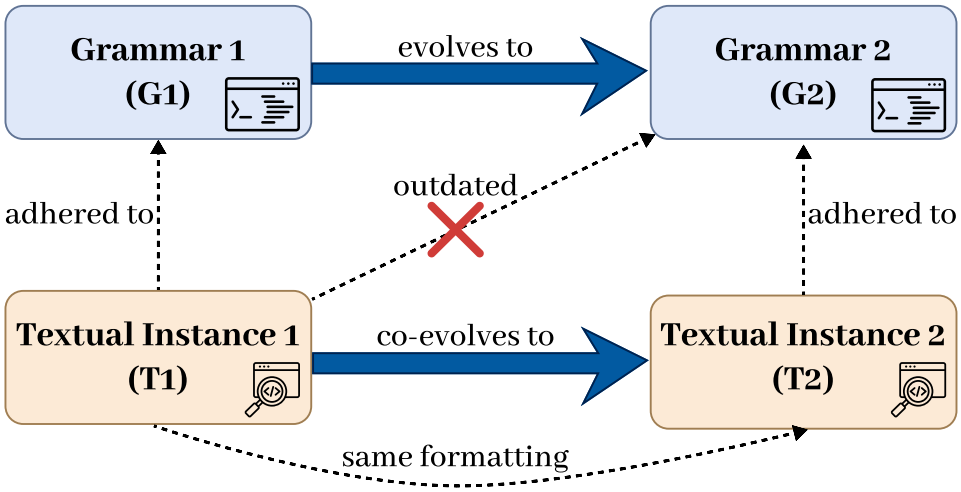

Fig. 1 When the grammar evolves, textual instances that originally adhered to it may no longer conform to it, and so need to be co-evolved to conform to it

ten real-world case languages. We quantified the correctness of co-evolved instances using precision and recall metrics, characterized grammar evolution types and their impact on co-evolution performance, measured the preservation of human-oriented information such as comments and formatting, and examined whether prompt designs transfer across different LLMs. Our findings identify the conditions under which LLM-based co-evolution succeeds and where it faces scalability limitations, providing practitioners with insights on the applicability of this approach and researchers with directions for addressing current limitations.

This paper is a significantly extended version of our previous workshop paper [15], in which we first presented our LLM-based approach and addressed a subset of the research questions stated above. For the present manuscript, we considerably broaden our evaluation scope and introduce new research questions to provide a more in-depth analysis of factors affecting LLM performance in grammar-instance co-evolution. This entailed extensive work on classifying grammar evolution types and analyzing their impact on co-evolution correctness (leading to the new RQ2), investigating whether different LLMs require different prompt designs (RQ4, new), expanding the evaluation metrics to include precision, recall, and normalized percentages, and extending the number of case languages from 7 to 10 to strengthen the generalizability of our findings. Since the LLMs used in our workshop paper (Claude-3.5 and GPT-4o) are now two generations behind the current state-of-the-art, and Claude-3.5 Sonnet has been retired from Anthropic's API, we updated our experiments to use Claude Sonnet 4.5 and GPT-5.2. We retained the same prompting text from the workshop paper to examine whether prompts designed for earlier models remain applicable to their successors, which also provides data for addressing RQ4. Based on the new findings, we also extended our discussion to the threats to the validity in Sect. 6.5 and the implications for practitioners seeking to apply LLMs in DSL evolution tasks (new Sect. 6.1).

In subsequent sections, we describe the problem we solve in this paper in Sect. 2 and introduce the background knowledge involved in this paper in Sect. 3. In Sect. 4, we present our research methodology and evaluation metrics. Then, Sect. 5 presents and analyzes experimental results, with Sect. 6 discussing the research findings and connections to existing research. Next, we review related research work on DSL evolution and LLM applications in Sect. 7. Finally, Sect. 8 concludes this research.

2 Problem description and motivation

Xtext is an Eclipse-based framework for developing textual programming languages and domain-specific languages [16]. As core artifacts for defining a language, it involves metamodels, specifying the abstract syntax, and grammars, specifying the concrete syntax of a language. As the language evolves, metamodels and grammars need to be systematically co-evolved. In our previous work [17], we have proposed an automated approach that can avoid information loss in grammars when the meta-model is changed and corresponding grammar updates have to be derived. However, after the grammar evolves, the instances that originally conformed to may no longer conform to it (see Fig. 1). It would be possible to make use of existing techniques for co-evolving models with evolving metamodels. For that, the textual instance could be parsed using the original grammar to gain the model representation (i.e., in XMI format) that conforms to the original metamodel. Then existing model migration technologies (such as EMFMigrate [18]) can be used to transform this model into a model that conforms to the new metamodel. Finally, we can transform the migrated model back into a textual instance that conforms to the new grammar.

However, in this process, the human-oriented information such as comments and code formatting in the original text instance is discarded during the first transformation (i.e., when transforming textual instances to model representations). As an example, there is a "15 min tutorial" [19] on the official Xtext website which provides an example called `Domainmodel`, which includes two versions of grammar and their corresponding instances, where the second version adds five grammar rules. Listing 1 shows the original instance (T1 in Fig. 1) before grammar evolution, i.e., a slightly modified version of the tutorial example to which we have added human-oriented information at various locations. Line 10 is empty, which, in the case of entities with many more attributes, is a useful way of grouping them, The definition of instance `HasAuthor` has been compressed from originally four lines to a single line (line 14) by removing whitespace, making the overall instance more compact and easier to overview. Line 19 uses comments as a way to discard a part from the instance that might potentially be included again at a later time (outcommenting). Line 24 contains an additional comment in a style commonly used to add rationale and context to individual statements. More subtlely, lines 9 and 11, use a different type of indentation than the rest of the instance, based on tabs, which could be the result of an ongoing manual review and refactoring.

Listing 1 The instance of Domainmodel that conforms to the grammar before the evolution.

```
1  /**
2   * This is the example before the
3       evolution.
4   * This is the header.
5   * */
6  datatype String
7
8  /* this is the first comment, added by
9      me */
10 entity Blog {
11          title: String
12
13          many posts: Post
14 }
15
16 entity HasAuthor { author: String }
17
18 entity Post extends HasAuthor {
19     title: String
20     content: String
21     //many comment: Comment
22     many comments: Comment
23 }
24
25 entity Comment extends HasAuthor {
26     content: String // this is the
27     second comment, added 2025-01-01
28 }
```

Listing 2 The instance of `DomainModel` that conforms to the evolved grammar and that are generated using traditional model transformation techniques.

```
1  datatype String
2
3  entity Blog {
4      title: String
5      many posts: Post
6  }
7
8  entity HasAuthor {
9      author: String
10 }
11
12 entity Post extends HasAuthor {
13     title: String
14     content: String
15     many comments: Comment
16 }
17
18 entity Comment extends HasAuthor {
19     content: String
20 }
```

Traditional co-evolution approaches in the MDE sphere focus on co-evolution on the abstract syntax level, that is, the impact of new and changed meta-model classes and relationships to model elements. Concrete syntax information without an abstract syntax counterpart – that is, human-oriented information such as comments and whitespace – is not covered and hence, lost in the process. This is illustrated by Listing 2 showing the outcome of applying such an approach to the exam instance, which leads to the loss of all comments and formatting information.

3 Background

3.1 Language evolution

The evolution of programming languages is primarily driven by the need to improve readability, maintainability, and execution efficiency while supporting increasingly complex software systems. Horowitz [20] noted that evolution typically involves grammatical improvements, the introduction of new features, paradigm shifts such as object orientation, and adapting to changing computational requirements. In domain-specific languages (DSLs), language evolution often encompasses changes in syntax, semantics, and implementation methods. According to Cleenewerck et al. [21], DSL evolution may include grammatical extensions (such as adding new keywords), semantic adjustments (like optimizing the execution of specific DSL statements), implementation improvements (such as enhancing compilers), and compatibility maintenance (like providing automatic migration tools). DSL evolution is similar to the one of general-purpose programming languages, but due to its

domain-specific nature, evolution is typically constrained by domain requirements, necessitating a balance between extensibility and stability.

3.2 Xtext-based DSLs

Xtext is a framework for developing programming languages and domain-specific languages [16]. Language engineers can define a language using Xtext's powerful grammar language to obtain a complete infrastructure including parsers, connectors, etc. Xtext-based DSL development includes two key artifacts, i.e., grammar and metamodel. Grammar represents the concrete syntax of the language, while metamodel represents the abstract syntax of the language [22]. Xtext supports two scenarios for developing languages, namely metamodel-driven scenarios and grammar-driven scenarios [17]. In the metamodel-driven scenario, language engineers first create a metamodel to represent the domain concepts and their relationships, generate grammar through the metamodel, and then generate the Xtext artifacts with the grammar. Those Xtext artifacts are for generating a textual editor. In the grammar-driven scenario, language engineers directly define the grammar of the language and then generate the infrastructure from the grammar. In this generation, the metamodel will be generated.

3.3 Large language models

LLMs are advanced AI models based on the transformer architecture [23], which employs self-attention mechanisms to capture relationships between different parts of the input text. Building on this foundation, models like BERT [24] introduced bidirectional pre-training techniques that enable deep contextual understanding by jointly conditioning on both left and right context in all layers. LLMs such as ChatGPT and Claude have been widely explored in code-related applications, with several studies reporting promising results in code completion, generation, and transformation tasks [25, 26] These capabilities can be further enhanced through carefully designed prompts (known as prompt engineering), which guide the models to perform specific text processing and generation tasks [27].

While prompt engineering offers a lightweight way to adapt LLM behavior, an alternative approach is fine-tuning, which adapts a pre-trained model by further training it on task-specific data. Fine-tuning can yield strong in-domain performance but requires substantial computational resources and curated training datasets [28]. For the co-evolution task studied in this paper, labeled training data pairing grammar evolution steps with correctly co-evolved instances is limited, making fine-tuning impractical. Prompt engineering is, therefore, a natural choice, as it allows us to leverage the general code understanding capabilities of state-of-the-art LLMs without the overhead of model training.

4 Methodology

This study was conducted in two phases, each following the same three-step methodology framework, as depicted in Fig. 2. In the first phase, presented in our workshop paper [15], we selected seven case languages from an available dataset [14] as evaluation objects (Step 1). We then designed an LLM-based method to co-evolve grammar and instances, including developing Python scripts to automate interactions with LLMs, and iteratively optimizing the prompt text using Claude-3.5 and GPT-4o based on the official Xtext Domainmodel tutorial example [19] (Step 2). Finally, we applied the two solutions implemented with the above LLMs to the seven case languages, evaluating the correctness of co-evolution and the preservation of human-oriented information (Step 3).

The three additional case languages were selected following the same criteria as the original seven, i.e., repositories containing both Xtext grammar files and conforming instance files with identifiable evolution steps in their commit history. In the second phase, for the present journal extension, we made the following extensions on top of the above. First, we selected three additional case languages (eclipse-typescript-xtext, JSFLibraryGenerator, and majordomo) from the same dataset, expanding the evaluation scope from seven to ten case languages. Second, since Claude-3.5 used in the first phase has been retired from Anthropic's API (as discussed in Sect. 1), we updated the LLMs to their current successors, i.e., Claude Sonnet 4.5 and GPT-5.2. Third, we introduced two new research questions, concerning the impact of grammar evolution types on LLM performance (RQ2) and the transferability of prompts across different LLMs (RQ4).

One methodological decision requires explanation: we retained the prompt text developed using Claude-3.5 and GPT-4o in the first phase, without re-optimization for the new LLMs or the expanded set of case languages in the second phase. This was a deliberate choice for two reasons. First, it enables us to investigate prompt transferability across LLM generations (RQ4), which would not be possible if the prompt had been re-optimized for the new models. Second, the prompt was designed to address general challenges in LLM-based co-evolution (over-generation of optional elements, omission of mandatory elements, and loss of human-oriented information) and was calibrated on the Domainmodel tutorial example rather than on any of the seven case languages, reducing the risk that it is overfit to the original set of cases. We acknowledge that optimizing the prompt with all ten case languages or with the newer LLMs might yield a different prompt; we return to this point in the discussion of threats to

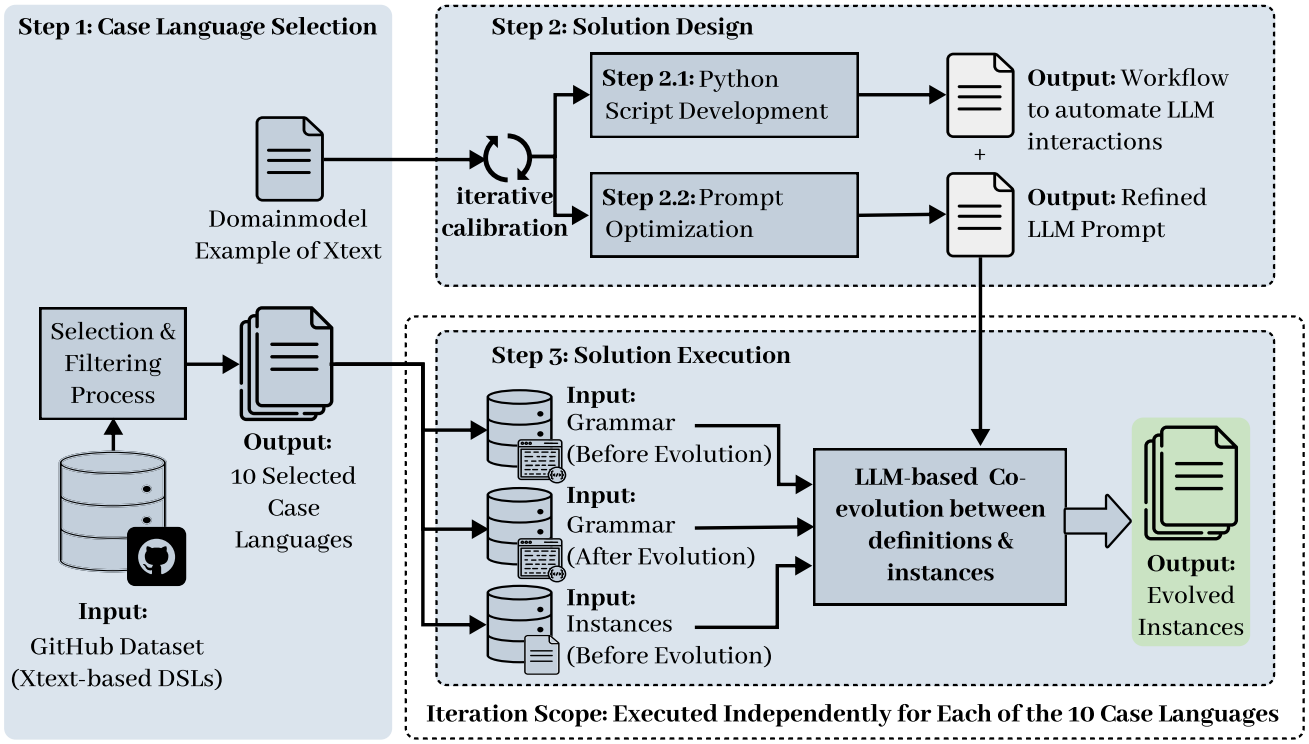

Fig. 2 Three-step research methodology

validity (Sect. 6.5). The following subsections describe each step of the three-step methodology in detail.

4.1 Case language selection

Compared with general-purpose programming languages, DSLs are designed for specific and evolving domains, where changes in domain concepts, stakeholder requirements, or implementation platforms can directly affect the language definition. Consequently, DSL metamodels and grammars are often more volatile than those of traditional programming languages. This volatility increases the practical need for approaches that support the co-evolution of DSL definitions and their existing instances.

We searched for case languages from an available dataset [13], since this dataset is specifically dedicated to Xtext-based DSLs. We limited our selection to repositories that contain both Xtext files and instance files. From the commit records of the repository, we can see that the Xtext files and instance files contained may contain many commits. For example, in the language *elite-se.xtext.languages.plantuml*, the grammar file "PlantUML.xtext" has 63 commits. For each language, we decided to pick a grammar from the commit that is closest to the present and ensure that there must be an instance that complies with the grammar, otherwise we will look for the grammar in earlier commits. The found grammar is the evolved grammar. Then, we continue to look for a grammar that differs from this version in earlier commits.

Once the grammar versions before and after the evolution step are determined, for each of them, we look for an instance that complies with them.

We identified ten case languages from the dataset. Their basic information is shown in Table 1, and the grammars before/after evolution and the instances that comply with this grammar found in their repositories are shown in Table 2 & 3. The selected case languages vary in instance size and grammar change complexity. Notably, the studied instances are relatively small in scale, and the distribution of grammar change types is uneven across cases, as we discuss in Sect. 6.5.

In Table 2, *Grammar 1* is a grammar with an earlier commit time, which is regarded as the grammar before the evolution, while *Grammar 2* is a grammar with a later commit time, which is referred to as the evolved grammar. Similarly, in Table 3 *Instance 1* is an instance with an earlier commit time, which is the object of LLM evolution operation (we call it the *original instance*), while *Instance 2* is an instance with a later commit time, which conforms to the evolved grammar, but may not be an evolved version of *Instance 1*, because the author of the instance may add or delete content irrelevant to the evolution. We will discuss this situation in the discussion section.

Table 1 Case languages and their basic information

Name	Domain	Category
xtext-orm [29]	Database	Data management and databases
xtext-dnn [30]	Deep learning	Artificial intelligence and machine learning
smart-dsl [31]	Blockchain	Security and networking
mongoBeans [32]	Database access	Data management and databases
elite-se.xtext.languages.plantuml [33]	GPL	programming languages
isis-script [34]	Java ISIS applications	Software development and engineering
CheckerDSL [35]	Test cases	Testing and verification
eclipse-typescript-xtext [36]	GPL	Programming languages
JSFLibraryGenerator [37]	Web UI / User interface	web and mobile development
majordomo [38]	Home automation	IoT, Embedded systems, and hardware

Table 2 Selected grammar and instance versions of the case languages. "1" denotes the grammar and instance version before the evolution and "2" denotes grammar and instance version after the evolution

Name	Grammar 1			Grammar 2		
	Date[1]	ID[2]	Rules[3]	Date[1]	ID[2]	Rules[3]
xtext-orm	2018-06-21	f84e2b3	15	2018-06-23	7cb74b2	22
xtext-dnn	2016-12-13	a4912d2	31	2016-12-17	15ccbc0	30
smart-dsl	2023-07-22	64259ba	8	2023-07-31	23424ac	9
mongoBeans	2012-06-04	279ecc8	9	2012-06-06	9f8360b	9
elite-se.xtext.languages.plantuml	2020-07-12	98f445d	52	2020-07-13	a41d99f	55
isis-script	2015-07-18	2546850	15	2015-09-05	f0380e8	21
CheckerDSL	2015-05-03	55911bf	42	2015-07-27	3fa6e6d	43
eclipse-typescript-xtext	2013-11-16	505cc89	23	2013-11-16	58e6e39	32
JSFLibraryGenerator	2016-03-27	44a1b28	6	2016-09-25	20efe92	7
majordomo	2013-02-06	cc01ae0	47	2013-02-08	a9967e0	47

[1] "Date" = the commit date of the grammar file
[2] "ID" = the commit ID of the grammar file's commit
[3] "Rules" = the total number of grammar rules in the grammar

4.2 Characterization of grammar changes

To answer RQ2, we need to understand what types of changes occurred between *Grammar 1* and *Grammar 2* in each case language. We performed a manual comparison of the two grammar versions for each of the ten cases.

For each case, we compared the grammar files line by line and identified the differences between the two versions. We recorded each change and categorized it along two dimensions.

The first dimension distinguishes between breaking and non-breaking changes [4]. A *breaking change* requires modifications to existing instances for them to conform to the new grammar. For example, if a keyword is renamed from `'Modifier'` to `'validator'`, all instances using the old keyword must be updated. A *non-breaking change* – for example, adding an optional attribute or a new rule that existing instances do not use – does not require modifications to existing instances.

The second dimension describes the operation type of each change. We use four descriptive categories: *Add* (a new rule, attribute, or keyword is introduced), *Delete* (an existing rule, attribute, or keyword is removed), *Rename* (an element keeps its structure but changes its name), and *Modify* (the structure or syntax of an existing element is changed). These categories are not intended as a formal taxonomy but as descriptive labels to characterize the changes we observed.

We applied this characterization process to all ten case languages. The results of this analysis are presented in Sect. 5.3, where we examine how these grammar change characteristics relate to LLM performance.

4.3 Solution design

In step 2, we designed a solution that leverages LLM to enable the co-evolution of grammar and instances in DSL and preserve human-oriented information in the instances during evolution. We will present the design of the solution in this section.

To implement and evaluate this approach, we selected two mainstream LLMs: Claude and ChatGPT, given their demonstrated capabilities in code understanding and generation tasks. The approach takes three inputs: the original grammar, an instance conforming to it (i.e., the instance to be evolved), and the evolved grammar, and contains a prompt we designed. With these inputs, LLMs are expected to analyze the grammar differences and perform instance evolution accordingly. This section first presents the detailed workflow of our approach, followed by the design of the prompting strategy.

4.3.1 Python script development

To achieve automated interaction with LLMs, we developed separate Python scripts for Claude Sonnet 4.5 and GPT-5.2 to replace manual operations on LLM product websites such as ChatGPT. These scripts require the configuration of corresponding API keys to access LLM services: Claude Sonnet 4.5 requires Anthropic's API key, while GPT-5.2 requires OpenAI's API key. We applied for API keys from Claude and OpenAI, respectively, and configured them in the Python scripts. We configure the maximum number of tokens that the model can generate in the current request (i.e., the `max_tokens` field) to 64000, which is a large number for the texts of the files from the ten case languages in this paper (Our consideration is that if we find that it is not large enough, we will increase it). Although the two scripts call different APIs, they follow the same workflow.

The workflow is as follows: First, the script reads three input files from specified local paths: the original grammar, the evolved grammar, and the instances that conform to the original grammar.

Considering that our solution does not require explicit training for a specific task, but allows LLM to analyze directly based on the prompt text we provide, the prompting technique we adopt is Zero-shot Chain-of-thoughts prompting [39].

We chose zero-shot over few-shot prompting [40] for two reasons: (1) grammar evolution varies considerably across cases in change types and complexity, making it difficult to select representative examples that generalize rather than bias the LLM;

(2) few-shot examples risk causing the LLM to overfit to the demonstrated evolution style rather than reasoning about the actual grammar differences. We consider a comparison with few-shot prompting as future work.

In our implementation, we send the text of one file at a time to the LLM, along with an additional sentence of hint text, such as *"Here is the initial version of the grammar (i.e., Grammar 1). Please remember this for future reference"*. The benefit is that this approach can be applied to larger languages in the future, reducing the likelihood that the total amount of text exceeds the LLM context window limit. When sending each text file, we do not require the LLM to analyze it, but only remember it. Only after all the text has been sent to the LLM do we start grammar analysis and instance evolution.

Finally, the script saves the evolved instance generated by the LLM to the local file system.

The Python scripts we created for this paper, grammars of the ten case languages, instances cloned from GitHub repositories, and instances generated by LLMs are available in the supporting materials [41].

4.3.2 Prompt optimization

To obtain a prompt that can effectively guide the LLM to co-evolve instance, we started with an initial prompt that we iteratively refined in the two LLM solutions based on the example *Domainmodel* in the official "15 min tutorial" of Xtext [19]. In this example, we made some changes to the instance before evolution and the grammar after evolution. The changes made to the instance before evolution have been introduced in Sect. 2, and we will introduce the changes to the grammar after evolution below. In each iteration, we used the prompt to drive the LLM to evolve the instance and then observe whether there are problems with the output instance, e.g., incorrectly modified elements. If there were problems with the output instance, we adjusted and optimized the prompt according to the problem, and entered the next iteration.

LLMs are generally affected by non-determinism, which we need to account for when evaluating the capability of the resulting approach. To this end, we repeated the co-evolution. When the instance was correctly co-evolved, we performed nine more co-evolution runs with the same prompt. Considering the uncertainty of LLM outputs, we decided that when at least six of the ten runs output good instances, we would use this version of the prompt as the final version. The out instance is good if it follows the evolved grammar and retains human-oriented information. The grammar before evolution is shown in Listing 3, which contains five grammar rules.

Table 3 Selected grammar and instance versions of the case languages. "1" denotes the grammar and instance version before the evolution and "2" denotes grammar and instance version after the evolution

Name	Instance 1			Instance 2		
	Date[1]	ID[2]	Lines[3]	Date[1]	ID[2]	Lines[3]
xtext-orm	2018-06-21	181dcd7	72	2018-06-23	7cb74b2	85
xtext-dnn	2016-12-13	a4912d2	33	2016-12-17	15ccbc0	42
smart-dsl	2023-07-22	64259ba	30	2023-07-31	23424ac	30
mongoBeans	2012-06-04	279ecc8	52	2012-06-06	9f8360b	29
elite-se.xtext.languages.plantuml	2020-07-12	98f445d	60	2020-07-13	a41d99f	60
isis-script	2015-07-22	1c88416	98	2015-09-05	f0380e8	68
CheckerDSL	2015-05-03	55911bf	173	2015-07-27	3fa6e6d	181
eclipse-typescript-xtext	2013-11-16	505cc89	8	2013-11-16	58e6e39	11
JSFLibraryGenerator	2016-03-27	44a1b28	1822	2016-09-25	20efe92	2176
majordomo	2013-02-06	cc01ae0	85	2013-02-08	a9967e0	86

[1] "Date" = the commit date of the grammar file
[2] "ID" = the commit ID of the grammar file's commit
[3] "Lines" = the total count of lines in the instance, empty lines included

The same tutorial provides an evolved grammar. This evolution is adding five new grammar rules, but does not involve any changes about the symbols (e.g., ';') in the grammar. To make the evolution changes also reflected in the symbol changes, we make two modifications to the evolved version, i.e., 1) in the `Entity` rule, we add commas (',') to the attribute features to separate them, instead of just spaces; 2) add a semicolon (';') as a terminator at the end of the `DataType` rule. In addition, we also deliberately added an optional attribute called `default` in the grammar rule `Feature` which means that LLMs need to identify more changes in the grammar. The final evolved grammar is shown in Listing 4.

Listing 3 The grammar of Domainmodel before the evolution.

```
...

Domainmodel:
    (elements+=Type)*;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends'
        superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':'
        type=[Type];
```

Listing 4 The grammar of Domainmodel after the evolution.

```
...
Domainmodel:
    (elements+=AbstractElement)*;

PackageDeclaration:
    'package' name=QualifiedName '{'
        (elements+=AbstractElement)*
    '}';

AbstractElement:
    PackageDeclaration | Type | Import;

QualifiedName:
    ID ('.' ID)*;

Import:
    'import' importedNamespace=
            QualifiedNameWithWildcard;

QualifiedNameWithWildcard:
    QualifiedName '.*'?;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID ';';

Entity:
    'entity' name=ID ('extends'
          superType=[Entity|
          QualifiedName])? '{'
        //(features+=Feature)*
        (features+=Feature  (','
         features+=Feature)*)?'}';
Feature:
    (many?='many')? name=ID ':'
         type=[Type|QualifiedName]
         ('(' default=ID ')')?;
```

The tutorial also provides an instance that conforms to the grammar before the evolution. But as we mentioned in Sect. 2, we added comments and format information to the instance before evolution (as shown in Listing 1) to evaluate whether the solution can preserve human-oriented information during co-evolution. We added four comments, one of which is changed from a normal instance line. In addition, we added an empty line and two tabs at different locations and put a multi-line code block into one line. Under the guidance of the prompting text, LLMs are expected to correctly identify this formatting information and replay it in the evolved instance.

We began with an initial prompting text that simply outlined the co-evolution task:

> **Initial prompt**:
> grammar_1 is the initial grammar of the DSL. We evolved it to get grammar_2. instance_1 was originally a text instance that followed grammar_1. Now I want you to analyze the differences between the two versions of the grammar and, based on this difference, modify instance_1 and get instance_2, which will follow grammar_2.

4.4 Evaluation metrics

In Step 3, we conducted solution execution and result evaluation. During the execution, we applied two Python scripts containing the final version of prompting text (one based on GPT-5.2 and the other on Claude Sonnet 4.5) to ten case languages. For each case language, we repeat the co-evolution ten times to account for non-determinism in LLM outputs, resulting in ten evolved instances per LLM per case language. We then established a set of metrics to measure different aspects of the evolved instances. These metrics are organized into three categories: correctness metrics, human-oriented information preservation metrics, and efficiency metrics. All metrics except response time were calculated through manual inspection of the evolved instances, as described below.

4.4.1 Correctness metrics

These metrics assess how correctly the LLM performs the co-evolution task.

Basic counts:

- *#LineReq*: Count of lines in instance 1 that require modification to conform to the evolved grammar.
- *#LineErr*: Count of lines with grammar errors in the evolved instance.
- *#LineEvl*: Count of lines in instance 1 that are evolved.
- *#LineEvlWrg*: Count of lines of instance 1 that are missing (or incorrectly evolved) in the evolved instance.

Derived ratios: To enable comparison across instances of different sizes, we compute the following normalized metrics:

- **Evolution precision**: The proportion of LLM-modified lines that are correct.

$$\text{Precision} = \frac{\#LineEvl - \#LineEvlWrg}{\#LineEvl}$$

- **Evolution recall**: The proportion of lines requiring evolution that are correctly evolved.

$$\text{Recall} = \frac{\#LineEvl - \#LineEvlWrg}{\#LineReq}$$

- **Error rate**: The proportion of lines in the evolved instance that contain grammar errors.

$$\text{ErrorRate} = \frac{\#LineErr}{\text{Total lines in evolved instance}}$$

4.4.2 Human-oriented information preservation metrics

These metrics assess the LLM's capability to preserve comments and formatting information during co-evolution.

Basic counts:

- *#LineCmtLost*: Count of lines of instance 1 with comments that are lost.
- *#LineCmtSave*: Count of lines of instance 1 with comments that are maintained.
- *#LineFmtLost*: Count of lines of instance 1 with format information that is lost.
- *#LineFmtSave*: Count of lines of instance 1 with format information that is maintained.

Derived ratios:

- **Comment retention rate**: The proportion of comment lines that are preserved.

$$\#\text{CmtRet} = \frac{\#LineCmtSave}{\#LineCmtSave + \#LineCmtLost}$$

- **Format retention rate**: The proportion of formatting lines that are preserved.

$$\#\text{FmtRet} = \frac{\#LineFmtSave}{\#LineFmtSave + \#LineFmtLost}$$

These rates are only computed for instances that contain comments or formatting information. For instances without comments, the comment retention rate is not applicable (N/A).

4.4.3 Efficiency metrics

To provide practical guidance for practitioners, we also measure the time required for LLM-based co-evolution and estimate the token consumption of each run.

- **Response time:** The wall-clock time (in seconds) from sending the prompt to receiving the complete evolved instance from the LLM API.
- **Token consumption:** The number of input and output tokens per run, estimated using tiktoken as an approximation.

We report the average response time across the ten runs for each case language, along with the estimated input token count. While we do not conduct a formal comparison with manual co-evolution time (which would require a controlled user study), we discuss the practical implications of the observed response times and token consumption in Sect. 6.

5 Results

In this section, we present two aspects of our results: (1) the final version of the prompting text we obtained; (2) the results obtained after applying our LLM-based solutions to the ten case languages, i.e., the instances evolved by the LLMs according to the evolution of the grammar, and our evaluation.

5.1 Finalization of prompting text

Following the method described in Step 2, we refined the prompt through two iterations using the *Domainmodel* example. The initial prompt produced syntactically valid but suboptimal results, revealing three key challenges in LLM-based co-evolution: (1) LLMs may under-generate by omitting mandatory syntactic elements such as delimiters, (2) LLMs tend to over-generate by instantiating newly added optional elements, and (3) LLMs do not inherently prioritize preserving human-oriented information. Based on these observations, we formulated three corresponding instructions that address these general challenges rather than case-specific errors. We verified the resulting prompt through ten runs on the *Domainmodel* example with both LLMs—when at least six of the ten runs produced correct results, we adopted it as the final version. This prompting text was developed in our workshop paper [15] using Claude-3.5 and GPT-4o. For this study, we applied the same prompt to Claude Sonnet 4.5 and GPT-5.2. This design choice allows us to examine whether prompts designed for earlier model versions remain applicable to their successors, addressing RQ4 on prompt transferability across LLM generations.

Through this prompt, we obtained instances from the LLMs (i.e., GPT-5.2 and Claude Sonnet 4.5) that complied with the evolved grammar. We verified this version of the prompt through repeated runs with both LLMs, and most instances obtained complied with the evolved grammar. Therefore, we decided to adopt it as the final version of the prompting text, as follows:

> **Final prompt**:
> grammar_1 is the initial grammar of the DSL. We evolved it to get grammar_2. instance_1 was originally a text instance that followed grammar_1. Now I want you to analyze the differences between the two versions of the grammar and, based on this difference, modify instance_1 and get instance_2, which will follow grammar_2. Please address the following things:
> 1. When evolving the instance, please do not omit any mandatory elements, such as characters enclosed by single quotes.
> 2. If grammar_2 adds a new grammar rule or a new attribute that is optional or in an "OR" relationship (i.e., |), then please do not instantiate it.
> 3. Do not miss or add any auxiliary information in the instance, e.g., comments, formats (white space, indents, tabs, empty lines, etc.).

Compared to the first version, the final version of the prompting text explicitly adds three items that the LLM needs to do. This is based on the problems we encountered in the process of optimizing the prompting text. The first is added because the LLM ignored the symbol ',', which is a mandatory element. The second is added because the LLM would actively instantiate the grammar rule "PackageDeclaration" which is a newly added optional rule. The third item is added because the LLM partially ignored comments.

5.2 Correctness evaluation (RQ1)

For each case language, we executed the co-evolution task repeatedly with each LLM and computed the average values of all metrics. The "Average" rows of Tables 4 show the average across all ten case languages.

The upper portion of Table 4 presents the correctness metrics for Claude Sonnet 4.5. Seven of the ten case languages (xtext-orm, smart-dsl, mongoBeans, plantuml, CheckerDSL, and majordomo) achieved 100% precision and 100% recall with zero grammar errors across all runs. xtext-dnn showed minor deviations with 94.67% precision and recall, caused

Table 4 Correctness metrics for Claude Sonnet 4.5 and GPT-5.2. Each row represents the average of ten runs

LLM	DSL	#LineReq	#LineErr	#LineEvl	#LineEvlWrg	Precision (%)	Recall (%)	ErrorRate (%)
Claude	xtext-orm	19	0	19	0	100	100	0
	xtext-dnn	15	0	15	0.8	94.67	94.67	0
	smart-dsl	3	0	3	0	100	100	0
	mongoBeans	4	0	4	0	100	100	0
	plantuml[1]	2	0	5	0	100	100	0
	isis-script	40	5.6	36.8	2.8	92.39	85	4.86
	CheckerDSL	8	0	8	0	100	100	0
	typescript[3]	0	0	0	0	N/A	N/A	0
	JSFLibraryGenerator	0	0	2	0	100	N/A	0
	majordomo	7	0	7	0	100	100	0
	Average[4]	9.8	0.56	9.98	0.36	98.56	97.5	0.49
GPT	xtext-orm	19	1.2	15.8	3	78	67.37	4.20
	xtext-dnn	15	0	15	0	100	100	0
	smart-dsl	3	0	3	0	100	100	0
	mongoBeans	4	0	4	0	100	100	0
	plantuml[1]	2	0	2.9	0	100	100	0
	isis-script	40	7.4	43.4	15.4	64.52	64.3	7.60
	CheckerDSL[2]	8	173	0	0	0	0	0
	typescript[3]	0	0	0	0	N/A	N/A	0
	JSFLibraryGenerator	0	571.7	1033.2	1033.2	0	N/A	4.20
	majordomo	7	0	7	0	100	100	0
	Average[4]	9.8	75.33	112.43	105.16	71.44	78.96	11.60

[1] Full name: elite-se.xtext.languages.plantuml
[2] GPT-5.2 failed to generate valid evolved instances for CheckerDSL in all ten runs
[3] Full name: eclipse-typescript-xtext
[4] N/A values are excluded from the average calculation; averages are computed over the applicable cases only

by an average of 0.8 incorrectly evolved lines per run. isis-script exhibited the lowest performance among cases with moderate size, with 92.39% precision, 85% recall, and a 4.86% error rate. This case language has 40 lines requiring modification and involves 21 grammar changes (as shown in Table 5). JSFLibraryGenerator, with a substantially larger Instance 1 (1822 lines), represents an edge case where the instance requires no modifications to conform to the evolved grammar (#LineReq = 0). Claude Sonnet 4.5 correctly identified this, making only minor adjustments (average two lines per run) with 100% precision. Similarly, eclipse-typescript-xtext requires no modifications to conform to the evolved grammar (#LineReq = 0), and both LLMs correctly produced no changes, resulting in N/A precision and recall. The overall average across all case languages shows 98.56% precision, 97.5% recall, and 0.49% error rate.

The lower portion of Table 4 presents the correctness metrics for GPT-5.2. Five case languages (xtext-dnn, smart-dsl, mongoBeans, plantuml, and majordomo) achieved 100% precision and recall with zero errors. However, GPT-5.2 failed to generate valid evolved instances for both CheckerDSL and JSFLibraryGenerator in all ten runs, resulting in 0% precision and 0% recall for these two cases. isis-script exhibited relatively lower performance among successful cases, with 64.52% precision and 64.3% recall. The overall average shows 71.44% precision, 78.8% recall, and 11.60% error rate, though the failures of CheckerDSL and JSFLibraryGenerator contribute substantially to the elevated error rate.

Comparing the two LLMs, Claude Sonnet 4.5 outperformed GPT-5.2 across all aggregate metrics. The precision difference is approximately 27 percentage points (98.56% vs. 71.44%), and the recall difference is approximately 18.5 percentage points (97.5% vs. 78.96%). Both LLMs performed well on the five smaller case languages (xtext-dnn, smart-dsl, mongoBeans, plantuml, and majordomo), which have fewer than 20 lines requiring modification. Performance degradation occurred primarily in cases with larger instances or more extensive grammar changes. isis-script (40 lines requiring modification, 21 grammar changes) posed difficulties for both LLMs. CheckerDSL (173 lines) caused complete failure for GPT-5.2 but was handled successfully by Claude Son-

Table 5 Grammar change characteristics of the ten case languages

Case language	Total changes	Breaking	Non-breaking	Primary operation types
CheckerDSL	5	4	1	*Modify*
plantuml	20	4	16	*Modify*, *Add*
xtext-orm	13	2	11	*Add*
mongoBeans	2	1	1	*Delete*, *Add*
smart-dsl	6	2	4	Rename
xtext-dnn	15	14	1	Rename
isis-script	21	6	15	*Add*, *Modify*, *Delete*
typescript	16	0	16	*Add*, *Modify*
JSFLibraryGenerator	11	1	10	*Add*
majordomo	11	3	8	*Modify*, Rename

Notes: [1] Full name: elite-se.xtext.languages.plantuml
[2] Full name: eclipse-typescript-xtext

net 4.5. JSFLibraryGenerator represents an edge case where no modifications were required; Claude Sonnet 4.5 correctly identified this, while GPT-5.2 did not.

Answer to RQ1: The capability of LLMs to produce correct co-evolution solutions varies with problem scale. For smaller cases in our dataset, both Claude Sonnet 4.5 and GPT-5.2 achieved high correctness, with precision and recall at or near 100%. For larger or more complex cases, performance differences emerged: Claude Sonnet 4.5 maintained 85% recall for isis-script (40 lines requiring modification) and achieved 100% precision for JSFLibraryGenerator. GPT-5.2 showed greater sensitivity to scale and complexity, dropping to 64.3% recall for isis-script and failing entirely for both CheckerDSL and JSFLibraryGenerator.

5.3 Impact of grammar evolution types (RQ2)

To answer RQ2, we first characterized the grammar changes in our case languages, then examined how these characteristics relate to LLM performance in co-evolving instances.

5.3.1 Grammar change characteristics

We analyzed the grammar changes between Grammar 1 and Grammar 2 for each of the ten case languages. Table 5 summarizes our findings.

The ten cases exhibit different evolution characteristics. The number of changes ranges from 2 (mongoBeans) to 21 (isis-script). The proportion of breaking changes varies from 15% (xtext-orm) to 93% (xtext-dnn). mongoBeans has the fewest changes: the keyword `'mongobean'` was removed, and an optional inline type definition was added. In contrast, isis-script has the most changes, involving the removal of the `repository` concept, restructuring of `action` and `parameter` syntax, and addition of a new `collection` concept. xtext-dnn stands out with the highest proportion of breaking changes. Its evolution consists of a systematic renaming of keywords from `kebab-case` to `camelCase` (e.g., `image-size` to `imageSize`), affecting 14 locations in the grammar. smart-dsl represents a concept renaming case, where `Modifier` was renamed to `validator` throughout the grammar, including the keyword, rule name, and references. xtext-orm and isis-script are characterized by additions. xtext-orm introduced a mandatory `Configuration` block and a directionality marker for entity fields. isis-script added collection support and restructured how actions and parameters are defined. CheckerDSL and PlantUML both have modifications as their primary operation type, but differ in breaking change proportion. CheckerDSL has four out of five changes as breaking, while plantuml has only four out of 20.

5.3.2 Relationship between grammar changes and LLM performance

We compared the grammar change characteristics in Table 5 with the correctness data in Tables 4, and analyzed the observed phenomena through specific evolution cases.

Statistical observations The number of changes does not show a linear relationship with LLM performance. xtext-dnn has 15 changes, and both LLMs performed well (100% precision and recall). isis-script has 21 changes, with degraded performance (Claude-Sonnet-4.5 86.67% recall, GPT-5.2 63% recall). mongoBeans (two changes) and smart-dsl (six changes) both achieved 100% correctness, though this may reflect their smaller overall scale. The proportion of breaking changes also cannot directly predict performance. xtext-dnn has the highest breaking change proportion (93%), yet both

LLMs handled it well. isis-script has a lower proportion (29%), yet performance was worse.

Context dependency of delete operations In isis-script, the IsisRepository rule was deleted in grammar 2. The correct evolution should remove the corresponding repository fragment from the instance (Listing 5 shows the original `repository` block, which should be removed). Both LLMs failed to execute this deletion correctly in multiple runs, and instead attempted to "evolve" its internal content (Listing 6 shows an incorrect example made by Claude).

Listing 5 The original `repository` block in *instance 1* of "isis-script".

```
...
@DomainServiceLayout(menuOrder="10")
    repository {

    @Action(semantics = SemanticsOf
       .SAFE)
    @ActionLayout(bookmarking =
       BookmarkPolicy.AS_ROOT)
      @MemberOrder(sequence="1")
            action listAll() {
                    container.
               allInstances
             (SimpleObject)
            }
            ...
    }
...
```

Listing 6 The *instance 2* output by Claude shows its error evolution on the `repository` block of "isis-script".

```
...
@DomainServiceLayout(menuOrder="10")
    service repository {

    @Action(semantics = SemanticsOf
          .SAFE)
    @ActionLayout(bookmarking =
         BookmarkPolicy.AS_ROOT)
            @MemberOrder(sequence
                  = "1")
            action void listAll {
                    body {

                        container.
              allInstances
             (SimpleObject)
                    }
            }
    ...
    }
...
```

In xtext-dnn, the `type` attribute in `BranchBody` was deleted (see Listing 7 and Listing 8). Both LLMs correctly identified and removed this attribute (see Listing 9 and Listing 10).

Listing 7 Grammar rule `BranchBody` in *grammar 1* of "xtext-dnn".

```
BranchBody:
    'type' REFERENCE type=BranchType&
    'operation' REFERENCE
     operation=OperationType
;
```

Listing 8 Grammar rule `BranchBody` in *grammar 2* of "xtext-dnn".

```
BranchBody:
    'eltwiseOperation' REFERENCE
      operation=OperationType
;
```

Listing 9 Instance fragment `branch` in *instance 1* of "xtext-dnn".

```
branch (name: "b1" in:"c3" out:32)
{
                operation -> PROD
                type -> residual
    ...
    }
```

Listing 10 Instance fragment `branch` in Claude-volved instance of "xtext-dnn".

```
branch (name: "b1" in:"c3" out:32)
{
                eltwiseOperation ->
                                PROD
    ...
    }
```

The comparison reveals: xtext-dnn's deletion is a single attribute, while isis-script's deletion is a complete rule containing multiple nested elements. This indicates that LLMs are more reliable when handling attribute-level deletions, but prone to errors when handling rule-level deletions (especially those involving nested structures).

Variations within the same language Beyond cross-language comparisons, isis-script also illustrates how different change types within the same language can lead to different outcomes. The attribute `returnType` of `IsisAction` changed from optional to mandatory, and both LLMs correctly added the return type in almost all runs. However, this observation should be interpreted with caution: the contrast between *Delete* and *Modify* outcomes here reflects the specific granularity of the *Delete* operation (rule-level deletion with nested structures) rather than change type per se, as *Delete* operations succeeded in xtext-dnn where deletion was at the attribute level. Within the same language, different change types led to different outcomes, further suggesting that factors beyond change type, such as deletion granularity and overall evolution complexity, play a role in determining LLM performance.

Impact of overall complexity Comparing isis-script and xtext-dnn: both contain *Delete* operations but with differ-

ent results. xtext-dnn's 15 changes are mainly systematic keyword renaming (kebab-case to camelCase), with limited structural changes. isis-script's 21 changes include removing the `repository` concept, restructuring `action` and `parameter` syntax, and adding `collection` concepts, with mixed types and big structural adjustments. When grammar evolution involves a mix of multiple change types and a lot of structural adjustments, LLMs are more prone to errors on individual changes—even when these change types can be handled correctly in simpler contexts.

Answer to RQ2: Grammar evolution type alone cannot predict LLM performance, e.g., *Delete*-type changes succeeded in xtext-dnn but failed in isis-script. Our case analysis reveals that LLM performance relates to the overall complexity of grammar evolution—when involving a mix of multiple change types and substantial structural adjustments, LLMs are more prone to errors. Deletion granularity is a factor: attribute-level deletions are easier to handle than rule-level deletions (especially those involving nested structures). Future research should focus on the overall complexity of grammar evolution rather than solely on change type classification.

5.4 Human-oriented information preservation (RQ3)

As described in Sect. 4.4, we executed the co-evolution task multiple times for each case language with the LLM-based approaches. Our data on human-oriented information retention were also averaged from these ten runs, as shown in Table 6. Note that five case languages (xtext-dnn, smart-dsl, plantuml, isis-script, and typescript) contain no comments in their *Instance 1*, so comment retention rates are not applicable for these cases.

The upper portion of Table 6 presents the human-oriented information preservation metrics for Claude Sonnet 4.5. For comment preservation, Claude Sonnet 4.5 retained all comments across all runs for the five case languages that contain comments (xtext-orm, mongoBeans, CheckerDSL, eclipse-typescript-xtext, and majordomo), achieving 100% comment retention rate with zero comments lost. For format preservation, seven of the ten case languages achieved 100% format retention rate. Both xtext-dnn and eclipse-typescript-xtext showed minor format loss with 97.58% and 97.50% retention, respectively. isis-script exhibited lower format retention at 71.12%, with an average of 28.3 formatting lines lost per run. JSFLibraryGenerator maintained 100% format retention, correctly preserving all 1822 lines of formatting information. The overall average across all case languages shows 100% comment retention and 96.60% format retention.

The lower portion of Table 6 presents the human-oriented information preservation metrics for GPT-5.2. For comment preservation, GPT-5.2 achieved 100% retention for xtext-orm, mongoBeans, eclipse-typescript-xtext, and majordomo. However, CheckerDSL shows 0% comment retention because GPT-5.2 failed to generate valid evolved instances for this case language in all ten runs, resulting in the complete loss of all 26 comment lines. For format preservation, five case languages (xtext-orm, xtext-dnn, smart-dsl, mongoBeans, and majordomo) achieved 100% format retention. plantuml showed minor format loss with 98.83% retention, and eclipse-typescript-xtext showed 90% retention. isis-script achieved 58.57% format retention with an average of 40.6 formatting lines lost per run. CheckerDSL shows 0% format retention due to complete generation failure. JSFLibraryGenerator shows 44.09% format retention, with an average of 1018.6 formatting lines lost per run. The overall average shows 80.00% comment retention and 79.10% format retention, though the CheckerDSL and JSFLibraryGenerator failures substantially affect these aggregate values.

Comparing the two LLMs, Claude Sonnet 4.5 outperformed GPT-5.2 in preserving human-oriented information. The comment retention difference is 20 percentage points (100% vs. 80.00%), and the format retention difference is approximately 17.5 percentage points (96.60% vs. 79.10%). When excluding the two cases where GPT-5.2 failed entirely (CheckerDSL and JSFLibraryGenerator), both LLMs achieved 100% comment retention for the remaining cases with comments. For format preservation excluding these two failure cases, Claude Sonnet 4.5 averaged 98.29% (calculated from the eight successful cases) while GPT-5.2 averaged 92.98%.

Answer to RQ3: Instance size affects LLMs' capability to preserve human-oriented information. For instances under 100 lines, both LLMs achieved high retention (100% for comments, >90% for formatting in most cases). Performance degraded for larger instances: isis-script (98 lines) showed 71.12% and 58.57% format retention for Claude and GPT, respectively; CheckerDSL (173 lines) caused GPT-5.2 to fail completely; JSFLibraryGenerator (1822 lines) was handled successfully by Claude (100% format retention) but caused GPT-5.2 to fail (44.09% format retention).

5.5 Prompt transferability across LLMs (RQ4)

To answer RQ4, we examined prompt transferability from two perspectives: first, we evaluated how the same prompt performs across different LLM generations; second, we tested whether prompt modifications affect different LLMs

Table 6 Human-oriented information preservation metrics for Claude Sonnet 4.5 and GPT-5.2. Each row represents the average of ten runs

LLM	DSL	#LineCmtLost	#LineCmtSave	CmtRet (%)	#LineFmtLost	#LineFmtSave	FmtRet (%)
Claude	xtext-orm	0	1	100	0	72	100
	xtext-dnn	N/A	N/A	N/A[1]	0.8	32.2	97.58
	smart-dsl	N/A	N/A	N/A[1]	0	30	100
	mongoBeans	0	6	100	0	52	100
	plantuml[2]	N/A	N/A	N/A[1]	0	60	100
	isis-script	N/A	N/A	N/A[1]	28.3	69.7	71.12
	CheckerDSL	0	26	100	0	173	100
	typescript[4]	0	1	100	0.2	7.8	97.50
	JSFLibraryGenerator	N/A	N/A	N/A[1]	0	1822	100
	majordomo	0	24	100	0	85	100
	Average[5]	0	6.44	100	2.93	240.37	96.60
GPT	xtext-orm	0	1	100	0	72	100
	xtext-dnn	N/A	N/A	N/A[1]	0	33	100
	smart-dsl	N/A	N/A	N/A[1]	0	30	100
	mongoBeans	0	6	100	0	52	100
	plantuml[2]	N/A	N/A	N/A[1]	0.7	59.3	98.83
	isis-script	N/A	N/A	N/A[1]	40.6	57.4	58.57
	CheckerDSL[3]	26	0	0	173	0	0
	typescript[4]	0	1	100	0.8	7.2	90
	JSFLibraryGenerator	N/A	N/A	N/A[1]	1018.6	803.4	44.09
	majordomo	0	24	100	0	85	100
	Average[5]	2. 89	3.56	80.00	123.37	119.93	7910

[1] Instance 1 contains no comments; CmtRet is not applicable
[2] Full name: elite-se.xtext.languages.plantuml
[3] GPT-5.2 generated empty files in all ten runs for CheckerDSL, resulting in the complete loss of human-oriented information
[4] Full name: eclipse-typescript-xtext
[5] N/A values are excluded from the average calculation; averages are computed over the applicable cases only

differently, which would indicate the need for LLM-specific optimization.

5.5.1 Performance of the same prompt across different LLMs

We applied the prompt developed in our workshop paper using Claude-3.5 and GPT-4o to Claude Sonnet 4.5 and GPT-5.2 without modification. Comparing Tables 4, the two LLMs performed similarly on smaller cases but diverged on larger ones.

For xtext-dnn, GPT-5.2 achieved 100% Precision and Recall while Claude Sonnet 4.5 achieved 94.67%. For smart-dsl, mongoBeans, plantuml, eclipse-typescript-xtext, and majordomo, both LLMs achieved 100% Precision and Recall. For xtext-orm, Claude Sonnet 4.5 achieved 100% while GPT-5.2 dropped to 78%. For isis-script, Claude Sonnet 4.5 achieved 86.67% Recall while GPT-5.2 dropped to 63%. For CheckerDSL, Claude Sonnet 4.5 successfully generated instances (though truncated) while GPT-5.2 output empty files in all ten runs. For JSFLibraryGenerator (1822 lines, #LineReq = 0), Claude Sonnet 4.5 correctly identified that no modifications were required and achieved 100% precision with minimal changes (average two lines per run). GPT-5.2 generated substantially altered outputs with 0% precision and 571.7 grammar errors on average.

5.5.2 Prompt adjustment experiment

To examine whether different LLMs benefit from different prompt designs, we tested an adjusted prompt with two additional instructions: (1) explicit guidance on handling deleted grammar rules, and (2) a requirement to always produce output. For (1), we added a sentence: "When evolving the instance, pay special attention to grammar rules that have been deleted in the evolved grammar. If a grammar rule no longer exists in Grammar 2, you must remove all instances of that rule from the evolved instance." For (2), we added a sentence: "Always output your best attempt at the evolved instance, even if incomplete. Never return an empty result."

Table 7 Data of evolution correctness on isis-script and CheckerDSL by both LLMs with the adjusted prompt

LLM	DSL	#LineReq	#LineErr	#LineEvl	#LineEvlWrg	Precision	Recall	ErrorRate
Claude-Sonnet-4.5	isis-script	40	4.6	40.7	4.6	88.89%	90.25%	11.11%
Claude-Sonnet-4.5	CheckerDSL	8	0	8	0	100%	100%	0%
GPT-5.2	isis-script	40	3.8	39.1	3.8	90.23%	88.25%	9.77%
GPT-5.2	CheckerDSL	8	3.5	5.2	0.6	62.80%	57.50%	42.20%

Table 8 Results of saving the human-oriented information of the isis-script and CheckerDSL instances by both LLMs with the adjusted prompt

LLM	DSL	#LineCmtLost	#LineCmtSave	CmtRet	#LineFmtLost	#LineFmtSave	FmtRet
Claude-Sonnet-4.5	isis-script	N/A	N/A	N/A	38.7	59.3	60.51%
Claude-Sonnet-4.5	CheckerDSL	0	26	100%	0.1	172.9	99.94%
GPT-5.2	isis-script	N/A	N/A	N/A	36.6	61.4	62.65%
GPT-5.2	CheckerDSL	0	26	100%	0.4	172.6	99.80%

In previous experiment, we observed that GPT-5.2 returned empty files on CheckerDSL, and both LLMs struggled with correctly removing instances of deleted grammar rules in the case language isis-script.

We evaluated this adjusted prompt on isis-script (40 lines requiring modification) and CheckerDSL (eight lines requiring modification). Tables 7 and 8 present the correctness and preservation metrics, while Table 9 shows response times.

The adjusted prompt affected the two LLMs differently. On isis-script, Claude Sonnet 4.5 showed decreased precision (92.39% to 88.89%) but improved recall (85% to 90.25%). GPT-5.2 showed big improvement in both precision (64.52% to 90.23%) and recall (64.3% to 88.25%). On CheckerDSL, Claude maintained perfect performance (100% precision and recall) while GPT-5.2 improved from complete failure (0%) to 62.80% precision and 57.50% recall.

For human-oriented information preservation (Table 8), both LLMs achieved 100% comment retention on CheckerDSL. Format retention showed slight degradation for Claude on both cases (CheckerDSL: from 100% to 99.94%, isis-script: from 71.12% to 60.51%), while GPT-5.2 maintained similar levels on CheckerDSL (99.80%) but also improved on isis-script (from 58.57% to 62.65%). Response times (Table 9) increased slightly for Claude on CheckerDSL (from 34.74 to 40.42 s) while remaining comparable on isis-script (from 28.93 to 29.12 s)(see Table 10).

Table 9 The execution duration of LLMs in isis-script and CheckerDSL evolution (using the adjusted prompt)

LLM	DSL	Duration (s)
Claude-Sonnet-4.5	isis-script	29.12
Claude-Sonnet-4.5	CheckerDSL	40.42
GPT-5.2	isis-script	20.07
GPT-5.2	CheckerDSL	27.51

Answer to RQ4: When applying prompts designed for Claude-3.5 and GPT-4o to their successors (Claude Sonnet 4.5 and GPT-5.2), both models performed similarly on smaller cases but diverged on larger ones, with GPT-5.2 failing completely on CheckerDSL (173 lines) while Claude succeeded. Prompt adjustment experiments revealed LLM-specific responses: the adjusted prompt improved GPT-5.2's isis-script precision from 64.52% to 90.23% while Claude's decreased from 92.39% to 88.89%, indicating that optimal prompts differ across LLM families.

5.6 Efficiency evaluation

Table 10 shows the average response time (in seconds) across ten runs for both LLMs. Response times range from 17.55 s (mongoBeans with Claude) to 558.8 s (JSFLibraryGenerator with Claude). For most case languages, response time is under 35 s. The average response time is 80.71 s for Claude Sonnet 4.5 and 36.57 s for GPT-5.2. The large difference in averages is primarily due to JSFLibraryGenerator, where Claude took nearly five times longer than GPT (558.8s vs 108.47s). Excluding this outlier, both LLMs show comparable response times.

Table 10 Average response time (in seconds) for each case language

DSL	Claude Sonnet 4.5	GPT-5.2
xtext-orm	23.89	25.75
xtext-dnn	23.88	22.17
smart-dsl	17.69	23.79
mongoBeans	17.55	24.1
plantuml[1]	26.8	26.3
isis-script	28.93	35.15
CheckerDSL	34.74	42.18
typescript[2]	23.12	24.58
JSFLibraryGenerator	558.8	108.47
majordomo	51.65	33.21
Average	80.71	36.57

[1] Full name: elite-se.xtext.languages.plantuml
[2] Full name: eclipse-typescript-xtext

Table 11 presents the estimated input and output token counts for each case language. Input tokens ranged from 951 (mongoBeans) to 36,625 (JSFLibraryGenerator), with an average of 5,971. Average output tokens were 788 for Claude Sonnet 4.5 and 476 for GPT-5.2. As expected, input token count scales with instance size, consistent with the response time pattern observed in Table 10: JSFLibraryGenerator, which has by far the largest input (36,625 tokens), also incurred the longest response time for both LLMs.

6 Discussion

Our evaluation demonstrates both the potential and limitations of LLM-based co-evolution for textual DSLs. In the following, we discuss these findings, open issues, and threats to validity.

6.1 Practical considerations

The efficiency results in Section 5.6 show that LLM-based co-evolution typically completes in under 35 s for most case languages. However, response time alone does not capture the full picture of practical feasibility. The time saved is valuable only if the output is correct. For cases where LLMs achieve 100% correctness (e.g., xtext-orm, smart-dsl, mongoBeans), the sub-30-second response time can replace manual work that would take longer. For cases with lower correctness (e.g., isis-script with 85% recall for Claude) or complete failure (e.g., CheckerDSL for GPT-5.2), the fast response requires manual correction or complete rework. Token consumption estimates, reported in Sect. 4.4.3 and Table 11, further confirm that input scale is the primary driver of both token usage and response time. We also estimate the total API cost for completing all ten case languages across ten runs to be approximately $2.97 for Claude Sonnet 4.5 and $1.71 for GPT-5.2 (see Table 11).

Our empirical study of Xtext-based DSL evolution on GitHub [14] found that grammar changes predominantly consist of additions, with rule increases occurring in 52.05% of repositories and rule deletions in only 6.39%. If LLM-based co-evolution can reliably support non-additive changes, a capability that may improve as LLM technology advances beyond the models evaluated in this study, it may gradually encourage practitioners to apply a broader range of grammar change types beyond additions, though verifying this would require dedicated longitudinal studies.

Compared to traditional model-based co-evolution appro aches, LLM-based co-evolution eliminates the need for writing explicit migration rules, reducing the initial setup effort. However, the unpredictability of LLM outputs means that practitioners must still review the results. We recommend LLM-based co-evolution for scenarios involving small instances (<100 lines) with limited grammar changes, where manual review effort is low. For larger-scale evolution tasks, a hybrid approach—using LLMs to generate initial migrations followed by manual refinement—may be more practical.

6.2 Preliminary observation on open-source models

To provide an initial reference point for open-source models, we conducted a preliminary experiment using *Qwen/Qwen3-Coder-Next* [42] across all ten case languages under the same experimental setup. The generated instances are included in the supporting materials [41]. A cursory inspection suggests that *Qwen/Qwen3-Coder-Next* underperforms both proprietary models in terms of co-evolution correctness: even on smaller cases such as xtext-orm and xtext-dnn, where Claude Sonnet 4.5 and GPT-5.2 achieved near-perfect correctness, noticeable errors were observed. On the largest case (JSFLibraryGenerator), it reproduced the original instance without performing any adaptation. Interestingly, response times were shorter than those of both proprietary models, which may reflect either faster inference or less thorough processing. Whether this speed advantage translates into any practical benefit remains unclear given the observed correctness issues. A systematic quantitative evaluation of *Qwen/Qwen3-Coder-Next* and other open-source models remains future work.

6.3 LLM-based vs. rule-based co-evolution

A natural question is why, or under which conditions, our LLM-based solution would be preferable over a conventional rule-based co-evolution tool, if such a tool existed for the case of grammars and textual instances. Building such a tool is challenging in the first place: beyond encoding how grammar

Table 11 Estimated token consumption and API cost for each case language (10 runs)

DSL	Input	Output (Claude)	Output (GPT)	Cost Claude ($)[5]	Cost GPT ($)[5]
xtext-orm	1,468	611	591	0.14	0.11
xtext-dnn	2,505	522	372	0.15	0.10
smart-dsl	1,228	327	426	0.09	0.08
mongoBeans	951	396	543	0.09	0.09
plantuml[1]	5,004	613	573	0.24	0.17
isis-script	2,511	835	1,026	0.20	0.19
CheckerDSL	4,769	1,470	0[2]	0.36	0.08
typescript[3]	1,451	329	139	0.09	0.04
JSFLibraryGenerator	36,625	1,633	49[4]	1.34	0.65
majordomo	3,195	1,147	1,042	0.27	0.20
Total	–	–	–	2.97	1.71

[1] Full name: elite-se.xtext.languages.plantuml
[2] GPT-5.2 generated empty files for CheckerDSL in all ten runs, resulting in zero output tokens
[3] Full name: eclipse-typescript-xtext
[4] GPT-5.2 failed to correctly process JSFLibraryGenerator, producing minimal output
[5] Cost estimated based on API pricing at the time of the study: $3.00/$15.00 per million input/output tokens for Claude Sonnet 4.5 and $1.75/$14.00 for GPT-5.2

edits map to instance edits, it must operate on a representation that preserves all concrete-syntax details and provide principled rules for how whitespace, line breaks, indentation, and comment placement are retained or repositioned as productions change. Without this lossless infrastructure, rule-based pipelines would need to rely on *parse–transform–prettyprint*, which tends to regenerate larger parts of the file and thus introduces widespread surface changes even when the required migration is small (see the example in Sect. 2). With a mature lossless rewriting infrastructure, rule-based solutions could offer determinism and reproducibility and would be preferable when grammar changes follow recurring patterns and must be applied at scale. However, for small- to medium-sized DSLs and instance corpora, where grammar evolutions are less frequent and applied to a limited number of artifacts, the effort required to implement and continuously maintain such infrastructure may not be justified. In these cases, our results indicate that LLMs provide a practical alternative: by operating directly on the original text, they can introduce only the syntactically required edits in situ, leaving surrounding formatting and comments unchanged, provided that the resulting instances are validated against the evolved grammar and corrected where necessary.

6.4 Future work

To address the limitations of grammar evolution complexity on LLM performance identified in this study, we plan to explore the following solutions in future work. First, for complex grammar evolution involving multiple change types and substantial structural adjustments, the evolution can be decomposed into multiple smaller incremental steps, with each step containing only a single or few change types, thereby reducing the complexity of each prompt. Second, a direction worth exploring is using LLMs to generate migration programs rather than having LLMs directly perform co-evolution. The advantages of this approach are that migration programs can be reviewed, tested, and reused, and can encode complex evolution logic into verifiable transformation rules. Furthermore, with the rapid development of LLM technology, more powerful models in the future may possess better long-context understanding and complex reasoning capabilities, enabling them to handle complex grammar evolution scenarios more effectively. Additionally, a systematic comparison between zero-shot and few-shot prompting strategies could reveal whether providing co-evolution examples improves LLM performance.

Beyond grammar-instance co-evolution, LLMs may also support other DSL engineering tasks. For example, in metamodel-grammar co-evolution, LLMs could help identify semantically equivalent grammar expressions that text-based comparison cannot detect [43]. LLMs may also assist in customizing DSL concrete syntax, such as transforming Xtext-generated grammars into Python-style grammars [44], and in providing more specific error messages when user input does not conform to the grammar [45]. We plan to apply LLM to these areas in our future work. Additionally, our preliminary experiment with *Qwen/Qwen3-Coder-Next* (Sect. 6.2) suggests that open-source models currently may lag behind proprietary ones on this task, and a systematic evaluation of open-source models across varying instance sizes and grammar change types would help clarify their

practical applicability and identify the conditions under which they become viable alternatives.

6.5 Threats to validity

Internal Validity When choosing the case language, some of the commit pairs we selected were too close together in the time perspective, which may have resulted in small differences between the two versions of the grammar. This may affect how representative the observed co-evolution scenarios are and thus pose a threat to internal validity. It is also important to study larger changes; therefore, in future work, we plan to study two versions of the language with larger changes, for example, to cover the changes between different UML versions. Additionally, the prompt used in our experiments was developed in the first phase of this study [15] using Claude-3.5 and GPT-4o, calibrated on the Domainmodel tutorial example. We retained this prompt without re-optimization for the second phase, which uses different LLMs and three additional case languages. While this decision was deliberate—enabling us to study prompt transferability (RQ4)—it introduces a potential threat: a prompt optimized with all ten case languages or with the newer LLMs might yield better performance. However, since the prompt addresses general co-evolution challenges rather than case-specific patterns, and was not calibrated on any of the seven original case languages, we consider this risk to be limited.

Construct Validity Similarly, while accessible, instances that can be found in the languages repository are likely there for demonstration and test purposes. Thus, they are likely smaller than productive instances would be. Moreover, in our evaluation cases, we identified human-created changes that were not strictly required by grammar evolution (e.g., added, removed, or refactored content for functional reasons). While this concern does not affect correctness, it implies that LLMs may not capture how humans co-evolve grammar instances.

External Validity Our case selection presents another threat regarding the characterization of LLM capability boundaries. While our dataset includes cases ranging from eight to 1822 lines, the distribution of instance size and evolution complexity is uneven. The largest case (JSFLibraryGenerator, 1822 lines) was found in the experiment to require no modifications (#LineReq = 0). The case with the most required modifications (isis-script) is relatively small at only 30 lines. We lack cases that combine both large instance size (>100 lines) AND extensive required modifications (>50 lines), which is necessary to fully determine LLM performance boundaries in complex evolution scenarios. While the nearly 18-fold increase in response time for JSFLibraryGenerator (Table 10) suggests that Claude may be approaching its capability boundary for processing large-scale instances, we cannot precisely define the exact location of this boundary in the absence of large-scale complex evolution cases. Furthermore, we did not systematically analyze the distribution of nesting structures across the selected case languages, which may influence the generalizability of our findings regarding deletion granularity. Future work should include more systematic experimental designs that vary instance size and evolution complexity, and incorporate a larger and more diverse set of case languages eyond the ten studied here, and extend the evaluation to DSL frameworks other than Xtext, to fully explore LLM capability boundaries.

7 Related work

7.1 DSL co-evolution

In an early systematic mapping study, Thanhofer-Pilisch et al. [46] explored the research landscape of DSL evolution, highlighting sparse cross-references between publications and limited focus on DSL characteristics. Hebig et al. [4] present a survey of approaches to co-evolve models with evolving metamodels, summarizing a multitude of approaches ranging from languages specialized to the automated generation of model transformations for dealing with non-breaking changes [47], to approaches that allow the definition of migration strategies, such as EMFMigrate[18] and Epsilon Flock [48], via approaches with predefined resolution strategies, like COPE [49] or the work of Gruschko et al. [50], to approaches utilizing constraint-based model search, such as CARE [51].

Previous work has investigated the role of concrete syntax during co-evolution for the case of graphical DSLs. Tolvanen et al. [52] proposed a framework for evaluating tool support for DSL co-evolution, together with an application to three specific tools for graphical DSLs: MetaEdit+ [53], EMF/Sirius [54], and Jjodel [55]. The framework assesses tool capabilities across four dimensions: location of change, nature of change, impact scope, and severity level. Considering changes to graphical concrete syntax, they find that all three tools provide adequate support: their comparison reports perfect scores for notation changes, meaning no assessed artifact depended adversely on the notation change itself. However, our focus is on the complementary case of textual DSLs, where layout and comments are represented as, e.g., whitespace-based grouping, implicit formatting rules, or concrete-syntax-tree information rather than as explicit model elements. Preserving these elements during grammar/model evolution raises different technical challenges. Furthermore, we explore a novel technical direction: leveraging Large Language Models to support co-evolution between grammar definitions and instances, offering an innovative perspective on solving textual DSL evolution problems.

7.2 Application of LLM in MDE

Zhang et al. [56] provide a systematic mapping of the LLM4MDE research landscape and show that current work is still concentrated on model generation, with LLMs having been explored for tasks such as model completion [57, 58], generation [59, 60], and model management operations [61, 62], whereas tasks such as model migration, DSL engineering, and metamodeling remain comparatively underexplored. Our work contributes to addressing this gap by systematically evaluating LLM-based co-evolution between grammar definitions and textual instances of DSLs.

A line of work closer to ours investigates LLMs for DSL engineering and textual DSL artifact generation. Joel et al. [63] surveyed LLM-based code generation for low-resource and domain-specific programming languages, highlighting that DSLs pose challenges beyond mainstream programming languages due to limited training data, specialized syntax and semantics, and the lack of standard benchmarks. Lamas et al. [64] proposed DSL-Xpert, a generic LLM-driven tool for translating natural-language instructions into code written in a user-specified textual DSL. Jiang et al. [65] studied LLM-based code generation for model transformation languages to specify model transformations, and evaluated across various model transformation languages. Brandolini et al. [66] developed langium-llm, enabling LLM-assisted DSL development through conversational interfaces based on either concrete syntax or abstract syntax representations. Costa et al. [67] proposed ModelMate, a recommender system for textual modeling languages that fine-tunes pre-trained language models to support identifier suggestion, line completion, and fragment completion for Xtext-based and other textual DSLs. These works demonstrate the potential of LLMs for supporting DSL engineering and DSL code generation. However, they mainly address the generation of new DSL artifacts or assistance during DSL development. Our work focuses on the co-evolution between definitions and instances of textual DSLs.

As for the co-evolution aspect, Kebaili et al. [68] explored the use of Large Language Models to address the co-evolution of code impacted by metamodel evolution. They proposed a prompt engineering-based approach that guides LLMs through structured prompts containing metamodel abstraction gaps, change information, and erroneous code. In their empirical study across seven Eclipse projects, their approach achieved an accuracy of 88.7%, reaching 95.2% for complex change scenarios. Unlike their focus on co-evolution between metamodels and generated code, our study addresses co-evolution between grammar definitions and textual instances.

dFinally, studies on LLM-based model transformation and rule generation report similar reliability and scalability challenges. For example, Kazai et al. [69] investigated ChatGPT-4 for UML-to-Java transformations, achieving 94% success for simple models but only 17% for complex ones—a scalability limitation consistent with our findings. Research on retrieval-augmented generation for OCL rule generation [70] revealed that retrieval method selection significantly impacts LLM performance, with semantic approaches outperforming lexical methods. These studies complement the DSL-oriented works above by showing similar reliability and scalability challenges in other MDE tasks.

8 Conclusion

This paper explored the potential of using LLMs (specifically Claude Sonnet 4.5 and GPT-5.2) to support co-evolution between DSL grammar definitions and instances. Our experiments across ten case languages demonstrated that LLMs can effectively handle co-evolution tasks while preserving human-oriented information like comments and formatting, particularly for smaller instances with limited grammar changes. Performance degraded with increasing instance size and grammar evolution complexity, with the two LLMs exhibiting different failure patterns.

We foresee the following directions for future work: First, to address the challenges posed by complex grammar evolution involving multiple change types and structural adjustments, future work should investigate techniques such as decomposing evolution into incremental steps or having LLMs generate reusable migration code rather than directly performing co-evolution. Second, an empirical investigation for comparing solutions generated by LLM-based approaches with those created by human developers could shed further light on the practical usefulness of generated solutions. Third, for complex cases in which several different solution strategies or tactics are feasible, investigating a way to create customizable prompts, or to create customizable migration code using an LLM, could further enhance the usefulness of LLM-based co-evolution.

Acknowledgements This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 501798263 and supported by the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF). It was partially funded by Vetenskapsrådet, grant 2021-04881_VR.

References

1. Lämmel, R.: Softw. Lang. Springer, Koblenz, Germany (2018)
2. Martin, T., Azvine, B.: Evolution of fuzzy grammars to aid instance matching. In: Int Symp Evolving Fuzzy Syst, 163–168. IEEE (2006)
3. Martin, T.P., Shen, Y., Azvine, B.: Incremental evolution of fuzzy grammar fragments to enhance instance matching and text mining. IEEE Trans. Fuzzy Syst. **16**(6), 1425–1438 (2008). https://doi.org/10.1109/TFUZZ.2008.925920
4. Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: A survey. IEEE Trans. Softw. Eng. **43**(5), 396–414 (2017). https://doi.org/10.1109/TSE.2016.2610424
5. Bock, F., Sippl, C., Heinz, A., Lauer, C., German, R.: Advantageous usage of textual domain-specific languages for scenario-driven development of automated driving functions. In: SysCon 2019, 1–8. IEEE, USA (2019). https://doi.org/10.1109/SYSCON.2019.8836912
6. Karu, M.: A textual domain specific language for user interface modelling. In: Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering. Lecture Notes in Electrical Engineering, 151, 85–996. Springer, New York (2012)
7. Bettini, L.: Implementing Domain-specific Languages with Xtext and Xtend. Packt Publishing Ltd, Birmingham B3 2PB, UK (2016)
8. Eclipse Foundation: langium. https://langium.org/, last accessed January 2026 (2024)
9. Dejanovic, I., Vaderna, R., Milosavljevic, G., Vukovic, Z.: Textx: A Python tool for domain-specific languages implementation. Knowl. Based Syst **115**, 1–4 (2017). https://doi.org/10.1016/J.KNOSYS.2016.10.023
10. Latifaj, M., Ciccozzi, F., Mohlin, M., Posse, E.: Towards automated support for blended modelling of UML-RT embedded software architectures. In: ECSA 2021 Companion. CEUR Workshop Proceedings. CEUR-WS.org, , Online (2021)
11. Bai, Y., Zhang, L., Zhao, F.: A survey on research of code comment. In: Proc. of ICMSS 2019, 45–51. ACM, China (2019). https://doi.org/10.1145/3312662.3312710
12. Rocco, J.D., Ruscio, D.D., Sipio, C.D., Nguyen, P.T., Rubei, R.: On the use of large language models in model-driven engineering. Softw. Syst. Model. **24**(3), 923–948 (2025). https://doi.org/10.1007/S10270-025-01263-8
13. Zhang, W., Strüber, D.: Tales from 1002 repositories: Development and evolution of Xtext-based DSLs on GitHub. In: SEAA 2024, 72–179. IEEE, France (2024). https://doi.org/10.1109/SEAA64295.2024.00034
14. Zhang, W., Strüber, D., Hebig, R.: Development and evolution of Xtext-based DSLs on GitHub: an empirical investigation. Empir. Softw. Eng. **31**(3), 48 (2026). https://doi.org/10.1007/S10664-025-10775-2
15. Zhang, W., Hebig, R., Strüber, D.: Leveraging LLMs to support co-evolution between definitions and instances of textual DSLs. In: STAF 2025 Workshops (Proc. of LLM4SE 2025, , Germany (2025) . (CEUR Workshop Proceedings. CEUR-WS.org)
16. Eclipse Foundation: Xtext Documentation. https://eclipse.dev/Xtext/documentation/102_domainmodelwalkthrough.html, last accessed January 2025 (2025)
17. Zhang, W., Holtmann, J., Strüber, D., Hebig, R., Steghöfer, J.: Supporting meta-model-based language evolution and rapid prototyping with automated grammar transformation. J. Syst. Softw. **214**, 112069 (2024). https://doi.org/10.1016/J.JSS.2024.112069
18. Wagelaar, D., Iovino, L., Ruscio, D.D., Pierantonio, A.: Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: Proc. of ICMT@TOOLS 2012. LNCS, 192–207. Springer, Czech Republic (2012). https://doi.org/10.1007/978-3-642-30476-7_13
19. Eclipse Foundation: 15 Minutes Tutorial. https://eclipse.dev/Xtext/documentation/102_domainmodelwalkthrough.html, last accessed January 2025 (2025)
20. Horowitz, E., Horowitz, E.: The evolution of programming languages. Fundamentals of Programming Languages, 1–31 (1984)
21. Cleenewerck, T., Czarnecki, K., Striegnitz, J., Völter, M.: Evolution and reuse of language specifications for DSLs (ERLS). In: ECOOP 2004. LNCS, 187–201. Springer, Norway (2004). https://doi.org/10.1007/978-3-540-30554-5_18
22. Wąsowski, A., Berger, T.: Domain-specific languages. In: Domain-Specific Languages: Effective Modeling. Automation, and Reuse, 87–142. Springer, Cham, Switzerland (2023)
23. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Adv. neural inform. process. syst. **30** (2017)
24. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Proc. of NAACL-HLT 2019, 4171–4186. Association for Computational Linguistics, USA (2019). https://doi.org/10.18653/V1/N19-1423
25. Husein, R.A., Aburajouh, H., Catal, C.: Large language models for code completion: A systematic literature review. Comput. Stand. Interfaces **92**, 103917 (2025). https://doi.org/10.1016/J.CSI.2024.103917
26. Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In: NeurIPS 2023 (2023)
27. Marvin, G., Hellen, N., Jjingo, D., Nakatumba-Nabende, J.: Prompt engineering in large language models. In: International Conference on Data Intelligence and Cognitive Informatics, 387–402. Springer (2023)
28. Shin, J., Tang, C., Mohati, T., Nayebi, M., Wang, S., Hemmati, H.: Prompt engineering or fine-tuning: An empirical assessment of llms for code. In: 2025 IEEE/ACM 22nd International Conference on Mining Softw. Rep. SR), 490–502. IEEE (2025)
29. blue995: xtext-orm. https://github.com/blue995/xtext-orm, last accessed January 2025 (2018)
30. Xpitfire: xtext-dnn. https://github.com/Xpitfire/xtext-dnn, last accessed January 2025 (2017)
31. bujosa: smart-dsl. https://github.com/bujosa/smart-dsl, last accessed January 2025 (2023)
32. szarnekow: mongoBeans. https://github.com/szarnekow/mongoBeans, last accessed January 2025 (2012)
33. elite-se: elite-se.xtext.languages.plantuml. https://github.com/elite-se/elite-se.xtext.languages.plantuml, last accessed June 2026 (2020)
34. vaulttec: isis-script. https://github.com/vaulttec/isis-script, last accessed January 2025 (2017)
35. ryanignatius: CheckerDSL. https://github.com/ryanignatius/CheckerDSL, last accessed January 2025 (2015)
36. vorburger and ibovet and bitdeli-chef: eclipse-typescript-xtext. https://github.com/vorburger/eclipse-typescript-xtext, last accessed February 2026 (2013)
37. stephanrauh and asterd: JSFLibraryGenerator. https://github.com/stephanrauh/JSFLibraryGenerator, last accessed February 2026 (2015)
38. jgraichen and cmur2: majordomo. https://github.com/majordomo/majordomo, last accessed February 2026 (2013)
39. Zhao, X., Li, M., Lu, W., Weber, C., Lee, J.H., Chu, K., Wermter, S.: Enhancing zero-shot chain-of-thought reasoning in large language

models through logic. In: Proc. of LREC/COLING 2024, 6144–6166. ELRA and ICCL, Italy (2024)
40. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: NeurIPS 2020 (2020)
41. Zhang, W., Hebig, R., Strüber, D.: 'Leveraging LLMs to support co-evolution between definitions and instances of textual DSLs' (2025). https://osf.io/dhjw2/
42. Qwen Team: Qwen3-coder-next technical report. Technical report. Accessed: 2026-02-03. https://github.com/QwenLM/Qwen3-Coder/blob/main/qwen3_coder_next_tech_report.pdf
43. Zhang, W., Hebig, R., Strüber, D., Steghöfer, J.: Automated extraction of grammar optimization rule configurations for metamodel-grammar co-evolution. In: Proc. of SLE 2023, 84–96. ACM, Portugal (2023). https://doi.org/10.1145/3623476.3623525
44. Zhang, W., Hebig, R., Steghöfer, J., Holtmann, J.: Creating Python-style domain specific languages: A semi-automated approach and intermediate results. In: Proc. of MODELSWARD 2023, 210–217. SCITEPRESS, Portugal (2023). https://doi.org/10.5220/0011744900003402
45. Zhang, W., Holtmann, J.: Technical report: Unresolved challenges and potential features in EATXT. CoRR **abs/2312.10250** (2023) https://doi.org/10.48550/ARXIV.2312.10250arXiv: 2312.10250
46. Thanhofer-Pilisch, J., Lang, A., Vierhauser, M., Rabiser, R.: A systematic mapping study on DSL evolution. In: SEAA 2017, 149–156. IEEE Comput. Soc., Austria (2017). https://doi.org/10.1109/SEAA.2017.25
47. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: ECOC 2008, 222–231. IEEE Comput. Soc., Germany (2008). https://doi.org/10.1109/EDOC.2008.44
48. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C., Poulding, S.M.: Epsilon Flock: a model migration language. Softw. Syst. Model. **13**(2), 735–755 (2014). https://doi.org/10.1007/S10270-012-0296-2
49. Herrmannsdoerfer, M.: COPE - A workbench for the coupled evolution of metamodels and models. In: SLE 2010. LNCS, 286–295. Springer, The Netherlands (2010). https://doi.org/10.1007/978-3-642-19440-5_18
50. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: Proc. of MoDSE, 3 (2007). Amsterdam, Netherlands
51. Schoenboeck, J., Kusel, A., Etzlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M.: CARE - A constraint-based approach for re-establishing conformance-relationships. In: APCCM 2014. CRPIT, 19–28. Australian Comput. Soc., New Zealand (2014)
52. Tolvanen, J., Kelly, S., Rocco, J.D., Pierantonio, A., Tinella, G.: A framework for evaluating tool support for co-evolution of modeling languages, tools and models. Softw. Syst. Model. **24**(2), 311–338 (2025). https://doi.org/10.1007/S10270-024-01218-5
53. MetaCase: MetaEdit+. https://www.metacase.com/products.html, last accessed January 2026 (2026)
54. Eclipse Foundation: Eclipse Sirius. https://eclipse.dev/sirius/, last accessed January 2026 (2025)
55. Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Università degli Studi dell'Aquila, Italy: Jjodel. https://www.jjodel.io/, last accessed January 2026 (2025)
56. Zhang, W., Jiang, B., Fu, Y., Cheng, H., Hummel, M., Scotti, V., Hagel, N., Li, J., Grossmann, G., Stumptner, M., Hebig, R., Strüber, D., Koziolek, A.: Large Language Models in Model-Driven Engineering: A Systematic Mapping Study (2026). https://doi.org/10.5281/zenodo.20064398.
57. Chaaben, M.B., Burgueño, L., Sahraoui, H.A.: Towards using few-shot prompt learning for automating model completion. In: NIER@ICSE 2023, 7–12. IEEE, Australia (2023). https://doi.org/10.1109/ICSE-NIER58687.2023.00008
58. Weyssow, M., Sahraoui, H.A., Syriani, E.: Recommending metamodel concepts during modeling activities with pre-trained language models. Softw. Syst. Model. **21**(3), 1071–1089 (2022). https://doi.org/10.1007/S10270-022-00975-5
59. Arulmohan, S., Meurs, M., Mosser, S.: Extracting domain models from textual requirements in the era of large language models. In: MODELS Companion 2023, 580–587. IEEE, Sweden (2023). https://doi.org/10.1109/MODELS-C59198.2023.00096
60. Chen, K., Yang, Y., Chen, B., López, J.A.H., Mussbacher, G., Varró, D.: Automated domain modeling with large language models: A comparative study. In: Proc. of MODELS 2023, 162–172. IEEE, Sweden (2023). https://doi.org/10.1109/MODELS58315.2023.00037
61. Abukhalaf, S., Hamdaqa, M., Khomh, F.: On Codex prompt engineering for OCL generation: An empirical study. In: MSR 2023, 148–157. IEEE, Australia (2023). https://doi.org/10.1109/MSR59073.2023.00033
62. Cámara, J., Troya, J., Burgueño, L., Vallecillo, A.: On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. Softw. Syst. Model. **22**(3), 781–793 (2023). https://doi.org/10.1007/S10270-023-01105-5
63. Joel, S., Wu, J., Fard, F.: A survey on llm-based code generation for low-resource and domain-specific programming languages. ACM Trans. Softw. Eng. Method. (2025). https://doi.org/10.1145/3770084
64. Lamas, V., R. Luaces, M., Garcia-Gonzalez, D.: Dsl-xpert: Llm-driven generic dsl code generation. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 16–20 (2024). https://doi.org/10.1145/3652620.3687782
65. Jiang, B., Hagel, N.J., Cheng, H., Jutz, B., Lange, A., Zhang, W., Sharma, R., Reussner, R., Koziolek, A.: Llm4mtls: Automated generation and empirical evaluation of model transformation languages. J. Object Technol. **25**(3), 1–14 (2026). https://doi.org/10.5445/IR/1000194182
66. Brandolini, L., Tisi, M., Sottet, J.: A language workbench extension to generate conversational interfaces for domain-specific languages. In: STAF 2025 Workshops (Proc. of LLM4SE 2025, , Germany (2025) . (CEUR Workshop Proceedings. CEUR-WS.org)
67. Durá, C., López, J.A.H., Cuadrado, J.S.: ModelMate: A recommender for textual modeling languages based on pre-trained language models. In: Proc. of MODELS 2024, 183–194. ACM, Austria (2024). https://doi.org/10.1145/3640310.3674089
68. Kebaili, Z.K., Khelladi, D.E., Acher, M., Barais, O.: An empirical study on leveraging LLMs for metamodels and code co-evolution. J. Object Technol. **23**(3), 1–14 (2024). https://doi.org/10.5381/JOT.2024.23.3.A6
69. Kazai, G., Osei, R.A., Bucaioni, A., Cicchetti, A.: Model transformations using LLMs out-of-the-box: Can accidental complexity be reduced? In: STAF 2025 Workshops (Proc. of LLM4SE 2025, , Germany (2025) . (CEUR Workshop Proceedings. CEUR-WS.org)
70. Li, K.C., Zolfaghari, V., Petrovic, N., Pan, F., Knoll, A.: Optimizing retrieval augmented generation for object constraint language. In: STAF 2025 Workshops (Proc. of LLM4SE 2025, , Germany (2025) . (CEUR Workshop Proceedings. CEUR-WS.org)

Weixing Zhang is a postdoctoral researcher in the MCSE group at the Karlsruhe Institute of Technology, Germany. He received his Ph.D. degree from the University of Gothenburg in October of 2025. Before that, he worked in industry for 7 years as a software engineer and he has extensive software development skills and experience. His research interests include empirical software engineering, MDE and DSL, and AI4SE.

Bowen Jiang received the M.Sc. in Computer Science (Fundamental Science and Engineering) from Waseda University (Tokyo, Japan) in 2024. Since October 2024, she is a Ph.D. researcher with the Institute of Information Security and Dependability (KASTEL) at Karlsruhe Institute of Technology (Karlsruhe, Germany). Her research interests include model-driven engineering, software testing, software maintenance, and AI4SE.

Yuhong Fu is a PhD candidate in the Industrial AI Research Centre at Adelaide University, expected to complete his Ph.D. degree in September 2026. He received his master's degree from the University of South Australia in 2018. His research fields include Model-Driven Engineering MDE, multi-level modelling, multi-view modelling, digital twins, interoperability, and AI4SE.

Anne Koziolek is a professor at Karlsruhe Institute of Technology. She is the head of the research group Modelling for Continuous Software Engineering (MCSE) at KASTEL – Institute of Information Security and Dependability. She is interested in conciliating model-based software engineering with development processes that have fast and agile feedback cycles and thus combine the benefits of both approaches. In particular, she is interested in tool support for systematic, yet low-cost model-based design space exploration to support making good design decisions.

Regina Hebig is Professor for Software Engineering at the University of Rostock, Germany. Her research interests include software evolution, AI4SE, software modelling, and software processes. She did her PhD at the University of Potsdam, Germany, in 2014 and her docent degree at the University of Gothenburg in 2019, where she worked as an Associate Professor until her change to Rostock in 2023.

Daniel Strüber is an associate professor at Chalmers and University at Gothenburg, Sweden, and an assistant professor at Radboud University in Nijmegen, the Netherlands. His research interests are in model-driven engineering, AI engineering, and variant-rich systems. He was awarded his doctoral degree from Philipps University Marburg, Germany, and worked as a post-doctoral researcher at University of Koblenz and Landau, Germany, and Gothenburg University, Sweden. He is a co-author of over 100 papers with six Best Paper Awards. He has been a Program Committee member of several leading conferences, including ICSE, ASE, MODELS, and a Program Chair of premier community venues, such as SPLC and ICGT.