JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

LLM4MTLs: Automated Generation and Empirical Evaluation of Model Transformation Languages

Bowen Jiang*, **Nathan Hagel***, **Haowei Cheng**†, **Benedikt Jutz***, **Arne Lange***, **Weixing Zhang***, **Rahul Sharma***, **Ralf Reussner***, and **Anne Koziolek***

*Karlsruhe Institute of Technology, Germany
†Waseda University, Japan

ABSTRACT Model transformation languages (MTLs) are domain-specific languages used to transform models conforming to a given metamodel into other models, including textual models such as source code. Developing correct model transformations in these languages is challenging and requires both language-specific and domain knowledge, creating a need for automated assistance and thus motivating the use of large language models (LLMs) for MTL code generation. However, due to the limited availability of training data and executable examples, LLM-generated MTL code is often not syntactically valid or semantically usable out of the box. This paper presents *LLM4MTLs*, an automated workflow for constructing and comparing prompting strategies for LLM-generated MTL code, together with an evaluation suite and an empirical evaluation. The workflow systematically explores prompt constructions combining few-shot prompting, grammar prompting, and helper methods inclusion, and evaluates them using both syntactic and semantic metrics. We construct an evaluation suite spanning four MTLs (ATL, ETL, QVTo, and the Reactions language) with executable reference scripts and manually written test suites, and evaluate across three LLMs. We find that few-shot prompting consistently improves syntactic quality across all four MTLs while gains in semantic correctness are uneven and language-dependent. For ATL, Pass@1 remains unchanged across all strategies and models, indicating that few-shot prompting improves surface-level syntax more readily than deep transformation semantics. Grammar prompting stabilizes code generation when combined with few-shot examples, but in isolation, it can be ineffective or even counterproductive for certain model–language combinations. Furthermore, including helper methods in the prompt as a complementary amplifier can be beneficial. Finally, LLM Model choice influences syntactic correctness and similarity for certain MTLs, particularly ETL and QVTo, while its influence on semantic correctness remains limited across all MTLs.

KEYWORDS Model Transformation Languages, Large Language Models, Code Generation, Prompt Engineering, Grammar Prompting, Domain-specific Languages

1. Introduction

Large Language Models (LLMs) have now been firmly established in generating source code from natural language descriptions (Joel et al. 2025; Amalfitano et al. 2025). While code generation is mostly done for general-purpose languages such as Python or Java, LLMs are also increasingly being explored in specialized domains, including Model-Driven Software Development (MDSD). For example, LLMs are used to generate model instances (Garaccione et al. 2025) and to verify the consistency of models (Eisenberg et al. 2025).

Model Transformations are central to MDSD (Mens & Van Gorp 2006). In a model transformation, a set of source models is transformed into a set of target models for purposes of analysis, simplification, or generation of runnable General Purpose Language (GPL) code, like Java or C++. Transforma-

JOT reference format:
Bowen Jiang, Nathan Hagel, Haowei Cheng, Benedikt Jutz, Arne Lange, Weixing Zhang, Rahul Sharma, Ralf Reussner, and Anne Koziolek. *LLM4MTLs: Automated Generation and Empirical Evaluation of Model Transformation Languages*. Journal of Object Technology. Vol. 25, No. 3, 2026. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2026.25.3.a22

tions consist of a set of transformation rules written in an Model Transformation Language (MTL) which provides optimized syntax and language features for definition. Examples of MTL include the Atlas Transformation Language (ATL) (Jouault et al. 2008), Epsilon Tranmsformation Language (ETL) (Kolovos & Garcia-Dominguez 2022), and Acceleo (Brun & Pierantonio 2008). Nevertheless, creating transformations between models can be a timely and challenging task, requiring knowledge not only about the MTL, but also about both source and target models, or even about the domains where those models originate (Anastasakis et al. 2007).

Given the challenge of manually developing correct transformations, it is natural to investigate whether LLMs can assist in generating MTL code. However, applying LLMs to this task presents unique challenges. Firstly, MTLs are Domain Specific Languages (DSLs) that have limited representation in the training data of LLMs, which implies that LLMs typically lacks knowledge of MTL-specific syntax and idioms (Joel et al. 2024). Secondly, MTLs and model transformations depend on metamodel semantics concerning the source and the target metamodel, which are the rules and definitions of how concrete model instances can be defined. Although metamodels offer the necessary context, they also introduce specialized semantic constraints that require domain-specific reasoning, which remains challenging for LLM (Joel et al. 2024).

Out-of-the-box prompting frequently leads to the generation of transformation code with syntactic or semantic errors, even when metamodel information is provided. The state of the art shows LLM-generated transformations succeed only in simple scenarios, but fail in more complex tasks (Buchmann 2024). In practice, finding effective prompts becomes a manual trial-and-error process (Ye et al. 2023; Pryzant et al. 2023). This leaves two research gaps. Firstly, there is no systematic and reproducible approach for optimizing and evaluating LLM-based MTL code generation. Secondly, there is a lack of publicly available evaluation suites with reference transformations and executable test suites across multiple MTLs (Zhang, Jiang, Fu, Cheng, et al. 2026; Burgueño et al. 2025). This paper addresses both gaps by investigating the following **Research Question**: **How can we systematically construct and evaluate prompting strategies for LLM-based MTL code generation, and to what extent do such strategies improve syntactic and semantic quality across different MTLs?**

We introduce *LLM4MTLs*, an automated, metric-driven workflow that standardizes prompt construction, code generation, and evaluation for MTL code generation. The workflow targets measurable syntactic and semantic outcomes and provides a reproducible means of exploring and comparing different prompt configurations. We further construct an evaluation suite of transformation examples spanning four MTLs: ATL (Foundation 2018), ETL (Kolovos & Garcia-Dominguez 2022), QVT-Operational Language (QVTo) (OMG 2016), and the Reactions language of the Vitruvius framework (Klare et al. 2021). The evaluation suite includes reference transformation scripts, metamodels, and manually written test suites for both syntactic and semantic validation. Using this workflow and evaluation suite, we conduct an empirical evaluation of generated MTL code combining prompting strategies, including few-shot prompting, grammar prompting, and Xtext-based language-specific helper method inclusion across three LLMs: *GPT-5.1* (2025), *Gemini 2.5 Pro* (2025), and *Claude Sonnet 4.5* (2025). The workflow is realized using n8n workflow automation (*N8n.Io - AI Workflow Automation Tool* 2025); all components are containerized, and all artifacts are provided in a replication package (Jiang et al. 2026). In summary, this paper makes the following three contributions:

- An evaluation suite of 47 transformation examples across four MTLs, consisting of executable reference scripts, metamodels, and test suites for automated validation.
- An automated workflow for prompt generation, MTL code generation, and metric-driven evaluation.
- An empirical evaluation of our workflow across four MTLs and three LLMs, offering systematic evidence on how different prompting strategies and LLMs choices affect the syntactic quality and semantic correctness of generated MTLs code.

The rest of this paper is structured as follows. In section 2, we provide background on MTLs as our application domain. In section 3, we survey related work with regards to using LLMs for generating code for MTLs, and other low-resource languages, such as other DSLs. In section 4, we provide an overview of our optimization workflow and describe the implementation. We present our evaluation suite and the results of our evaluation afterwards in section 5. We discuss the findings, limitations, and threats to validity in section 6, and finally, we conclude the paper in section 7.

2. Background

2.1. Model-Driven Software Development

MDSD is the definition and use of software models, such as Unified Modeling Language (UML) class or sequence diagrams, to create runnable software systems. It advances the concept of Model-Based Engineering (MBE) further. There, the same models are used for design and communication, but developers do not generate runnable program code from these models (Brambilla et al. 2017).

In the context of MDSD, the term "model" usually does not refer to machine learning models, such as LLMs. Instead, models conform to structures similar to those in Object-Oriented Programming (OOP). To be more exact, a model must conform to a *metamodel*, which is another model that defines a set of valid models or *instances* (Brambilla et al. 2017). Metamodels consist of an abstract syntax that describes the structure of models, at least one concrete syntax to express models with (such as grammars for textual models), and the semantics for models (Brambilla et al. 2017). Commonly used standards for metamodelling include the Meta Object Facility (MOF) to define the abstract syntax (*Meta Object Facility* 2019), and the Object Constraint Language (OCL) to describe additional restrictions on models through their semantics (OMG 2014). One of the main purposes of MDSD is to automate software development. This can be done either by interpreting the models directly, or to generate code from them by applying model

transformations on them (Brambilla et al. 2017, Sect. 3.1).

2.2. Model Transformations

Model transformations automatically convert a set of *source models* into a set of *target models*. They require a *definition* that describes how this conversion occurs. Such a definition is a set of *transformation rules*. Each rule describes how to translate constructs of the source- into constructs of the target metamodel (Kleppe et al. 2003). Since MDSD treats code as models, model transformations can also take code as source and target models. In fact, generating running code from models is a main purpose of model transformations (Mens & Van Gorp 2006). Following Brambilla et al., we call model transformations with code as source model Text-to-Model (T2M) transformations, transformations with code as target model Model-to-Text (M2T) transformations, and transformations only with non-code models Model-to-Model (M2M) transformations.

Model transformations are usually written with their own DSLs, such as the Atlas Transformation Language (Jouault et al. 2008), or the Query/View/Transformation (QVT) family (OMG 2016) of transformation languages. Of course, transformations can also be written in a GPL such as Java. However, users of MTLs find that these specialized languages offer numerous features that increase their comprehensibility and productivity, among other quality attributes (Höppner et al. 2022).

2.3. View-based Modeling – The VSUM approach

View-Based Modeling (Atkinson et al. 2010) is an approach in which views are tailored to the needs of individual stakeholders, thereby reducing the perceived complexity of the overall system under investigation. At its core lie Virtual Single Underlying Models (VSUMs) (Klare et al. 2021), which consolidate all essential model information and expose views that contain only relevant aspects while omitting extraneous details. The Vitruvius framework implements the VSUM approach and provides the *Reactions Language* (Klare 2021), a domain-specific language for specifying consistency preservation rules. Each rule reacts to a designated change in one view and propagates the resulting modifications throughout the system, ensuring that all constituent models remain mutually consistent at all times.

3. Related Work

3.1. Applying LLMs for model transformations

LLMs is finding widespread application for model transformation tasks. They are used both to generate model transformation rules and to do the actual transformation themselves (**?**). Therefore, we present works from both research areas here.

Buchmann prompted M2M transformations in Java with ChatGPT for the Families-to-Persons example (Anjorin et al. 2017), a commonly used case study in the field of model transformations (Buchmann 2024). The generated model transformations worked well in the batch case, i.e., converting one source model to a target model. However, prompting for incremental transformations, which only transform the changed part of a model, failed with either compilation errors or failing tests. Pontes Miranda et al. evaluated the prompting of view definitions in the ViewPoint Definition Language (VPDL) of the EMF Views framework (Pontes Miranda et al. 2024). View definitions transform interrelated models with different metamodels into another model, or view, that combines the information of the underlying models. The authors include view descriptions and metamodels converted to PlantUML for the prompt, and use chain-of-thought prompting. They ensure the syntactical correctness of the output by parsing and reprompting the LLM. The performance of LLMs is limited at best, however, since the generated view definitions capture only a subset of the relations between models and their views (Precision $\in [0, 0.58]$).

In (Cibrián et al. 2025), the authors proposed an agent-based LLM pipeline that transforms Modelica models to the SysML v2 representation. Their approach delivered promising results with a mean precision of 89.05%. The usage of LLMs also caused some problems; it introduced inconsistencies in the required validation loops to correct the mistakes made by the LLM. Their evaluation also lacked the semantic verification and was based only on structure.

3.2. LLMs in other MDSD areas

In (Cámara et al. 2023), the authors investigated the generation of UML diagrams via prompts to ChatGPT. The diagrams created or assisted by this method were generally correct, but contained syntactical errors. Some modeling concepts that UML supports, such as multiple inheritance or integrity constraints, are not covered by ChatGPT. PlantUML seems to work best notation-wise than other modeling notations/languages to create UML diagrams, e.g., USE (the UML-based Specification Environment). Rather than avoiding the LLM assisted modeling, the authors recommend improving the quality and quantity of publicly available modeling artifacts so that LLMs have more data to train on.

Generating OCL constraints with the help of LLMs is the focus of (Abukhalaf et al. 2024) and works reasonably well. PathOCL, the name of the presented method, combines prompting techniques with simple path coverage that subsets the UML class diagram to only relevant portions of the model. The results are promising and need further investigation and optimization to be used reliably. This work also saw better and more publicly available MDSD artifacts as the limiting factor to better performance of LLMs in this task. Beyond model and constraint generation, LLMs have also been applied to instance generation in the context of language evolution. Zhang et al. (Zhang et al. 2025; Zhang, Jiang, Fu, Koziolek, et al. 2026) explored using LLMs to support grammar-instance co-evolution for Xtext-based textual DSLs, exploring the potential of LLMs to preserve auxiliary information such as comments and formatting. Hagel et al. (Hagel et al. 2024) analyzed how LLMs can be applied to generate models using a textual DSL in a model-based low-code tool. Results show that the LLM was capable of generating a proprietary, to the LLM unknown DSL, and a user study showed that task completion time could be reduced. While those studies apply LLMs to DSL-related code generation, their work focuses on instance evolution, whereas ours addresses MTL generation from natural language. In Eisenberg et al. (Eisenberg et al. 2025) the authors apply LLMs to detect conflicts in versioned models

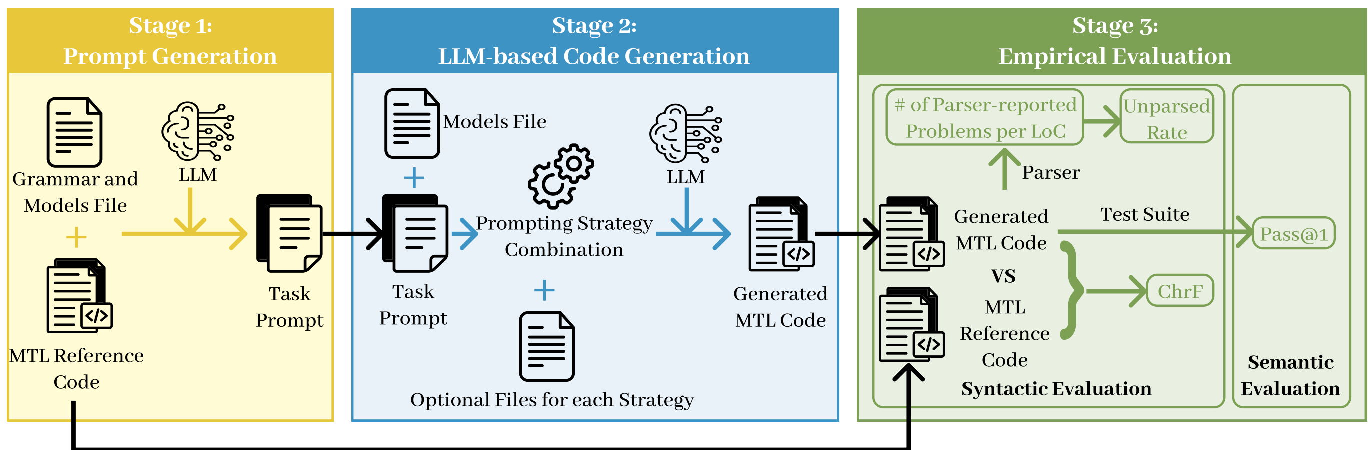

Figure 1 Overview of the LLM4MTLs Approach.

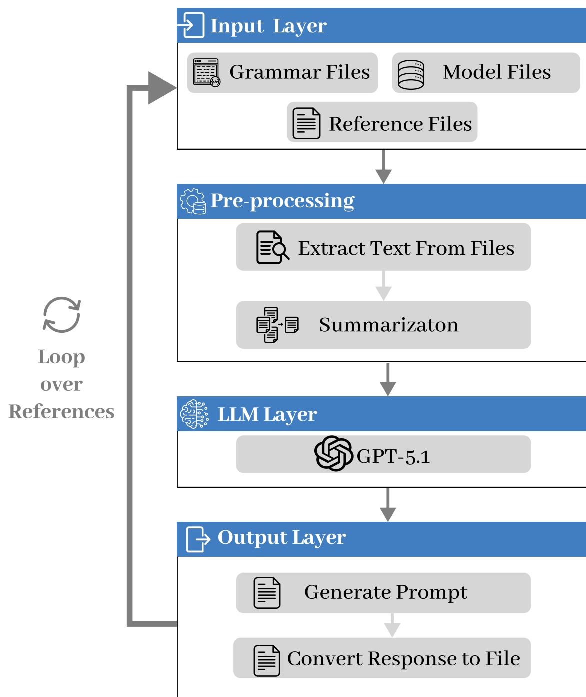

Figure 2 Detailed workflow of Prompt Generation.

and automatically resolve them using LLMs. In (Hagel et al. 2025) the authors discuss the usage of LLMs for explicit model consistency based on natural language or formal consistency rules. Our study addresses MTL generation, including the generation of the Reactions language for automated consistency maintenance, rather than applying LLMs to resolve inconsistencies directly. Besides, LLMs have also been explored for adjacent domain-specific and formal modelling tasks, including self-adaptive systems (Li et al. 2024b,a) and discrete controller synthesis (Ishimizu et al. 2025).

4. LLM4MTLs - An Automated Workflow

We present *LLM4MTLs*, an automated workflow for improving the reliability of LLM-generated MTL code through systematic prompt composition and metric-driven evaluation. The workflow targets measurable syntactic and semantic outcomes and provides a reproducible means of exploring and comparing different prompt configurations. An overview is depicted in Figure 1, where the three main stages are distinguished by color. In the **Prompt Generation** stage (yellow), *task prompts* are automatically derived from existing MTL reference implementations to address the lack of structured evaluation datasets (subsection 4.1). In the **Code Generation** stage (blue), a range of prompting strategies and their combinations are applied to generate MTL code from these task prompts (subsection 4.2). In the **Empirical Evaluation** stage (green), the generated code is assessed against the reference implementations using syntactic similarity, syntactic correctness, and semantic correctness metrics (section 5). Here, *Grammar Files* refer to extended Backus–Naur form (EBNF) grammars of the target MTL; *Model Files* refer to **metamodel artifacts** for both Prompt Generation and Code Generation. Test suite additionally loads concrete **model instances**.

Our workflow emphasizes automation and thus supports efficient repetition and fast feedback cycles when searching for an optimal prompting strategy configuration. Strategies and the associated resources can be adjusted to accommodate different target MTLs. To ensure reproducibility, all required components—including the workflow definition, model files (metamodel artifacts), MTL grammars, and code snippets—are bundled within containers. Our workflow is implemented using n8n (*N8n.Io - AI Workflow Automation Tool* 2025), a self-hosted low-code tool that enables LLM workflows to be visually constructed and edited. To support traceability and systematic comparison across prompt configurations, all prompts and LLM-generated artifacts are persisted. The complete implementation, including workflow definitions and evaluation artifacts, is made available as a replication package (Jiang et al. 2026).

4.1. Prompt Generation

We enable an automated dataset construction across MTLs by employing an LLM to reverse-engineer *task prompts* from existing reference code snippets, formulated in a developer-to-AI voice. The snippets provided enable the LLM to capture context and intent even when it is unfamiliar with a given MTL and is, consequently, incapable of generating high-quality code directly. Minor imperfections in the generated *task prompts* are acceptable, because human-authored prompts are rarely perfect too. Crucially, by encoding additional contextual details directly in the prompt, our automated generation approach facilitates adaptation to further languages and broader example sets. The generated *task prompt*, together with the associated model files (metamodel artifacts), serves, in the end, as input for baseline code generation.

Figure 2 illustrates the prompt generation workflow. Upon execution, the workflow reads the reference code snippets from a specified directory and iterates over each file. To enrich the contextual information available to the LLM, it optionally loads the corresponding model files (metamodel artifacts) and grammar definitions. The name of each reference file is preserved to maintain the association between the generated prompt and its source. To ensure stylistic consistency, every query is prefixed with the system prompt shown in Listing 1, which instructs the LLM to produce concise prompts that capture only the transformation intent and the relevant method names. Although this system prompt is formulated for the Reactions language, it can be straightforwardly adapted to other MTLs.

System Prompt

```
1  You are acting as a senior developer who knows the
   Model Transformation Language: **Reactions Language**
   inside out. Your task is to reconstruct the NATURAL-
   LANGUAGE request that could have produced the code
   shown in the next message, as if a developer were
   asking an AI assistant to generate that code.
2  Rules
3  1. You MUST explicitly state the reaction and routine
   names.
4  2. You MUST describe the intent on a high abstraction
   level.
5  3. DO NOT say 'model::component', 'component' is enough
    for the context.
6    DO NOT say 'in the update block...', rather describe
    generally what happens in the routine and reaction.
7    Do NOT mention 'snippet', 'reverse-engineer', or any
    analysis meta language.
8  4. When there are parts not included in the grammar,
   provide the method names.
9  5. Write in the *developer-to-AI* voice, e.g.  'Create
   a transformation [...] that...'.
10 6. Keep the request under **100 words**.
```

Listing 1 System Prompt for Prompt Generation (Reactions)

The workflow pairs the system prompt with a user prompt that supplies the additional context required to generate correct transformation definitions. Specifically, the user prompt contains the reference code, along with the extracted and summarized content of the associated model and grammar files. A representative example—the PERSONTOFAMILY transformation for the Reactions language—is presented in Listings 2 and 3, illustrating how the intent of the original code is clearly conveyed in the generated prompt. The prompt generation workflow thereby expands the dataset with consistent, semantically aligned prompt–code pairs. To ensure prompt quality, all generated prompts were manually reviewed for semantic correctness and fidelity to the intent of the original code. This verification step is critical, as all subsequent strategy evaluations are predicated on these prompts.

Reactions Language

```
1  [...]
2  reactions: familiesToPersons
3  in reaction to changes in families
4  execute actions in persons
5  reaction DeletedMember {
6    after element families::Member deleted
7    call deletePerson(affectedEObject)
8  }
9  routine deletePerson(families::Member member) {
10   match {
11     val person = retrieve persons::Person corresponding
        to member
12     val family = retrieve families::Family
        corresponding to person
13   }
14   update {
15     removeObject(person)
16     removeCorrespondenceBetween(member, person)
17     removeCorrespondenceBetween(family, person)
18   }
19 }
```

Listing 2 PersonToFamily.reactions Example

Prompt

```
1  Create a transformation in the **Reactions Language**
   that synchronizes the *families* and *persons* models.
2  Define a reaction named **DeletedMember** that triggers
    whenever a `Member` is deleted in the families model.
   This reaction should call a routine named **
   deletePerson**, which retrieves the corresponding `
   Person` and its `Family`, then removes the `Person`
   object and deletes all correspondences between the
   deleted `Member`, the `Person`, and the `Family`.
```

Listing 3 Generated Task Prompt

4.2. Code Generation

Compared to the prompt generation stage, the code generation workflow illustrated in Figure 3 incorporates additional steps for optimization and execution. The workflow first reads the task prompt files produced by the prompt generation stage, and extracts the task prompt text, the associated model files (metamodel artifacts), and the name of the corresponding reference code snippet. It then applies the prompting strategies selected by the user.

Three strategies are currently supported: (i) *few-shot prompting* (k=3), which augments the prompt with three representative reference examples of the target MTL; (ii) *grammar prompting*, which appends an EBNF grammar excerpt of the target MTL to impose explicit syntactic constraints and guide the LLM toward syntactically valid outputs (Wang et al. 2023); and (iii) *helper method inclusion*, which supplies predefined utility functions to the LLM's context. The third strategy can be used for MTLs that express key transformation constructs outside of their syntax. This is the case for the Reactions language, where managing consistency happens within Java-defined methods, that need to be called within reactions. The n8n workflow is designed to accommodate additional strategies through the straightforward

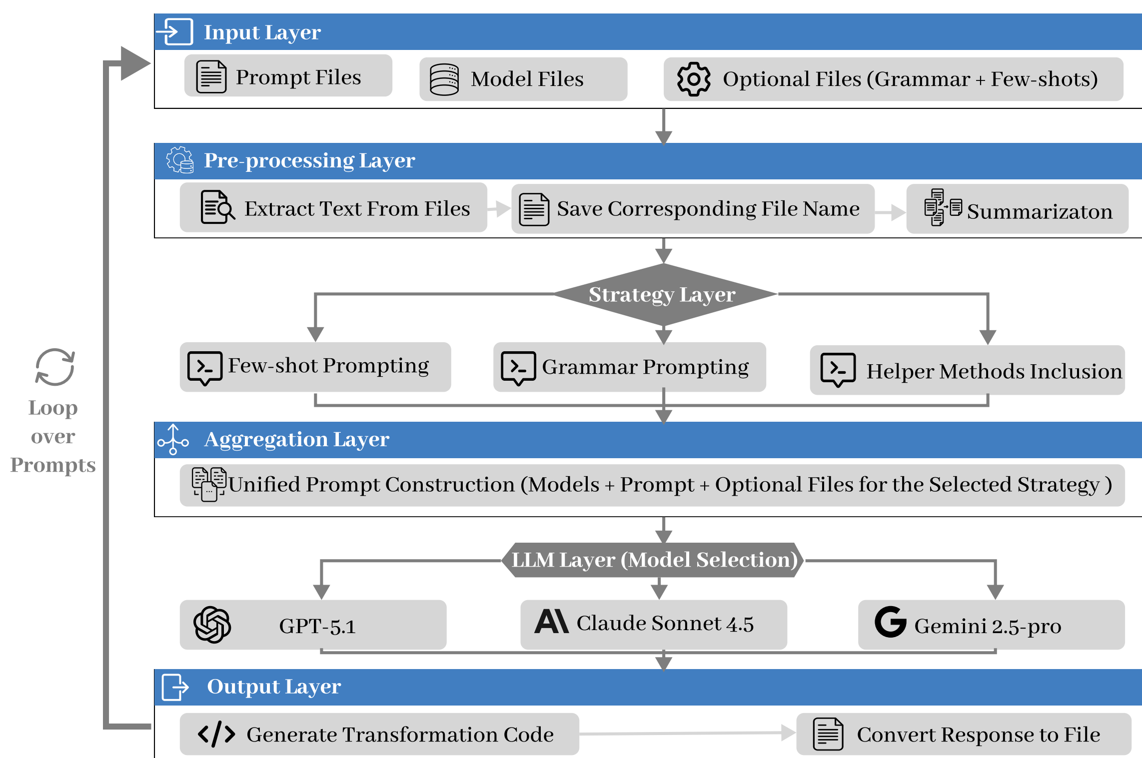

Figure 3 Detailed workflow of LLM-based MTL Code Generation

inclusion of further files or resources in the prompt.

System Prompt

```
1 You are an expert developer for the **Reactions
  language** (model transformation DSL). Your job is to
  translate the user's natural-language specification
  into a complete, syntactically valid .reactions file.
2 Rules
3 1. Follow the DSL grammar exactly (imports,
  transformation block, reactions, routines, guards,
  create/update sections, persistence paths,
  correspondence links, etc.).
4 2. Use the transformation / reaction / routine names
  provided by the user whenever they are specified.
5 3. If a name is missing, invent a concise, CamelCase
  name that matches the intent.
6 4. Use as much Reactions language as possible and use
  Xtend only when necessary
7 5. Do **not** wrap it in Markdown fences, and do **not
  ** add commentary, explanations, or blank lines beyond
  what the language requires.
```

Listing 4 System Prompt for Code Generation (Reactions)

Based on the selected strategy options, the content of the corresponding files is loaded from a specified directory and incorporated into the final user prompt. This aggregated prompt is then submitted to a user-selected LLM, which generates the corresponding code and writes the output to the designated location. This process is repeated for each queued task prompt. As with prompt generation, the prompt is divided into a system prompt and a user prompt.

Listing 4 shows the system prompt used for code generation in the Reactions language as a representative example. The instructions embedded in the system prompt enforce adherence to the target MTL grammar and are designed to minimize hallucination.

4.3. Adaptation to other Model Transformation Languages

Adapting this workflow to another MTL requires that the configuration of input and output files be set up appropriately. This includes the specification of the location of the new MTL's references, additional files required for e.g., few-shot prompting, and the desired save destination for the workflow's output. Whether the step of loading the model files (metamodel artifacts) needs to be updated depends on whether the generated code is to be applied to the same model files. These configurations can be performed directly within the respective workflow nodes. Regarding adaptation steps that go beyond reconfiguring the locations of the input and output paths, the system prompt of the workflow must be adjusted to reflect the characteristics and capabilities of the desired MTL. For instance, when applying the workflow to ATL, ETL, and QVTo, the system prompts from the Reactions language, see Listings 1 and 4, had to be updated. Furthermore, the improvement strategies, which are applicable for an MTL, can vary and not all may be applicable. The Reactions language, for instance, included an additional improvement strategy for Xtext-based MTL: including helper methods into prompt. Depending on the MTL's characteristics, further strategies could also be included. To ensure that the semantics of the generated MTL code are validated appropriately, tests should be included. When applying the workflow

to a new MTL, the manual effort required depends on whether pre-existing tests are used, must be written manually, or can be generated automatically. This also includes setting up the respective testing frameworks, which were different for the languages we investigated.

5. Results and Evaluation

We evaluate which prompting strategies improve LLM-generated MTL code, considering the raw output of the LLM using only a task prompt and model files (metamodel artifacts) without any additional context as the baseline. The goal is to systematically evaluate LLM-based code generation for MTLs across two dimensions: syntactic quality and semantic correctness. We frame the evaluation around three evaluation questions (EQs) which operationalize the overarching Research Question posed in section 1:

EQ1 To what extent do the prompting strategies improve the *syntactic* quality of LLM-generated MTL code in terms of syntactic similarity and correctness compared to the baseline?

EQ2 To what extent do the prompting strategies improve the *semantic* correctness of LLM-generated MTL code compared to the baseline?

EQ3 To what extent does the choice of LLM model affect the quality of the generated model transformation code across different MTLs?

5.1. Evaluation Metrics

Syntactic Similarity To assess how closely the generated transformation code resembles the reference implementation, we employ the **Character n-gram F-score (ChrF) metric** (Popović 2015), which measures character-level F-score between generated and reference code. Unlike token-level metrics such as BLEU (Papineni et al. 2001), ROUGE (Lin 2004), or METEOR (Banerjee & Lavie 2005), ChrF operates at the character level, making it more appropriate for programming languages where single characters such as semicolons or brackets carry syntactic significance. Although ChrF is not a reliable indicator of functional correctness, it provides a meaningful signal of how structurally divergent two code snippets are.

Syntactic Correctness LLMs may hallucinate invalid syntax or unknown tokens, yielding code that cannot be parsed. The **Unparsed Rate** (Bassamzadeh & Methani 2024) quantifies the proportion of generated snippets that fail parsing due to syntactic errors; lower values indicate that a larger share of the output is at least syntactically valid. To obtain a finer-grained view of syntactic quality, we additionally report the **average number of parser-reported problems per line of code** (PPL_s) for each generated snippet s. This metric complements the Unparsed Rate by distinguishing between snippets that fail to parse entirely and those that contain only minor, Localized errors. We compute PPL_s by dividing the total number of parser-reported problems for snippet s by its non-empty, non-comment Lines of Code (LoC). The parser infrastructure differs across MTLs to leverage the most accurate tooling available: for the Reactions language and QVTo, we use ANTLR-based parsers constructed from the respective grammars, for ATL, we use the Eclipse ATL parser and for ETL, we use the Eclipse Epsilon parser.

One limitation of PPL_s is that reported problem counts depend on the error recovery mechanisms of the parsers that we use. In particular, the ANTLR error-recovery mechanism may suppress subsequent errors once an earlier error triggers a recovery action. Accordingly, the values we provide for PPL_s are lower bounds.

Semantic Correctness To evaluate semantic correctness, we execute each generated snippet against concrete model instances and inspect the resulting output models. We measure semantic quality using the **Pass@1** metric (Paul et al. 2024), defined as the proportion of *parsable* snippets for which the first generated candidate passes all associated test cases. This metric is widely used in program synthesis and code-generation benchmarks (Paul et al. 2024).

For the Reactions language, test cases operate on a VSUM: some changes are applied to its constituent models, and the reactions file to preserve consistency. For ATL, ETL, and QVTo, separate test suites transform concrete source model instances. In any case, we assert that the resulting target models are equal to the reference All test suites were *manually written* to ensure that the semantic validation correctly reflects the intended transformation behavior.

5.2. Experiment Setup

We evaluate our approach on three LLMs: ***GPT-5.1*** (**2025**), ***Gemini 2.5 Pro*** (**2025**), and ***Claude Sonnet 4.5*** (**2025**). All models are treated as black-box generators, and no model parameters, weights, or training data are modified.

For each MTL in the evaluation suite, every combination of prompting strategy and LLM is executed on all available transformation scripts. The prompting strategies evaluated are: (i) **Baseline**: the task prompt together with the associated model files (metamodel artifacts) only; (ii) **Few-shots Prompting**: the baseline augmented with three representative few-shot examples; (iii) **Grammar Prompting**: the baseline augmented with the EBNF grammar of the target MTL; (iv) **Few-shots + Grammar Prompting**: the combination of few-shot and grammar prompting; and (v) **Few-shots + Grammar Prompting + Helper Methods Inclusion**—the combination further augmented the prompt with predefined helper methods in the prompt (applicable only to the Reactions language, where Xtext-based utility functions are required).

5.3. Evaluation Suite

To ensure a transparent and reproducible empirical evaluation across multiple MTLs, we construct an evaluation suite comprising executable transformation artifacts from four representative languages: Reactions, ATL, ETL, and QVTo. For each language, the evaluation suite includes: (i) executable transformation scripts serving as reference implementations, (ii) the corresponding input and output metamodels and model instances, (iii) manually written test suites that verify semantic correctness by executing the transformation and comparing the output against expected results, and (iv) parser-based syntactic validation that counts parser-reported problems per snippet.

Table 1 Evaluation Suite statistics across the evaluated MTLs.

Language	#	LoC			Complexity (avg)		
		Min	Max	Avg	#R/H	MC	#MM/M
Reactions	12	17	98	31.0	1.0 / 1.7	3.4	6/0
ATL	15	15	277	96.3	5.5 / 2.2	14.2	27/15
ETL	10	6	97	29.3	3.3 / 0.5	4.7	7/7
QVTo	10	13	25	18.5	3.7 / 0.0	3.8	1/0

[1] # = number of transformation scripts.
[2] LoC = non-empty, non-comment lines of code. Min./Max. = LoC range.
[3] #R/H = Avg. number of transformation rules / helpers per script
[4] MC = Avg. McCabe complexity per script.
[5] #MM/M = number of associated metamodel/model artifacts.

We draw artifacts from publicly available and official repositories (*Vitruv-CaseStudies* 2026; *ATL Zoo Benchmark* 2026; *Eclipse Epsilon* 2026; *org.eclipse.qvto* 2026), prioritizing diversity in transformation intent (e.g., creation, update, mapping, refactoring) and selecting only executable, end-to-end examples. Natural diversity in script complexity is preserved: ATL examples tend to involve larger scripts and a higher number of associated model artifacts, whereas QVTo and ETL examples are more compact. Furthermore, the Reactions language is defined for a specific transformation tasks, which is consistency preservation across multiple models (Klare et al. 2021). In contrast, ATL, ETL and QVTo are general-purpose transformation languages, and more commonly used. Thus, these languages are likely to be reflected in LLM training data, whereas Reactions are unlikely to occur in such training data.
Table 1 summarizes the evaluation suite scale. We report the number of transformation scripts and code-size statistics in LoC, measured as non-empty, non-comment lines. We further report three complexity indicators, averaged per script, to characterize the transformation complexity of each task inspired by Götz et al. (2021): the number of transformation rules, the number of helpers, and the McCabe cyclomatic complexity calculated over all rules and helpers. We additionally report the number of associated metamodels and model artifacts, which also influence execution complexity and evaluation cost.

5.4. Results and Findings

To assess the statistical significance of the observed improvements, we employ a two-level testing procedure. The ANOVA statistic is defined in Howell (1992).

Overall significance across LLMs To evaluate whether different LLMs exhibit significantly different performance within a given prompting strategy, we used non-parametric tests appropriate for the metric type. For continuous metrics, including ChrF similarity (see Table 2) and PPL_s (see Table 4), we applied the **Friedman test** across LLMs. For binary outcomes, including parsability (see Table 3) and the combined event *parsable and tests passed*, we used **Cochran's Q test** across LLMs. If one of the LLMs within a selected strategy (e.g., Few-Shot) produced significantly different results ($p < 0.05$), the triple is gray-shaded.

Pairwise comparisons against baseline Individual comparisons against the baseline are conducted per LLM, pairing each transformation script across strategies. For *binary* outcomes (parsability and test passed & unpassed), we apply **McNemar's exact test**. For *continuous* metrics (ChrF scores and PPL_s), we apply the **Wilcoxon signed-rank test**. Results significantly better than the baseline ($p < 0.05$) are marked with *.
For clarity, the abbreviations used in the tables are as follows: **FS** represents few-shot prompting; **GR** represents grammar prompting; **HM** represents inputting some defined helper methods as prompt.

5.4.1. Syntactic Evaluation (EQ1)

To answer EQ1, we evaluate syntactic quality using three complementary metrics: ChrF for character-level similarity, the Unparsed Rate for parsability, and PPL_s for fine-grained error density.

ChrF Table 2 reports the mean ChrF scores across all four MTLs. For the Reactions language, baseline values of GPT and Gemini are comparatively low, scoring 0.62 and 0.60, whereas Claude already achieves 0.78. One plausible explanation is that Claude's training data contains structurally similar code, although this cannot be verified directly. Across the remaining three MTLs, baseline scores vary: ATL and ETL exhibit moderate baselines (from 0.57 to 0.59 and from 0.61 to 0.64), while QVTo shows stronger initial values (from 0.69 to 0.79), suggesting differing degrees of prior model exposure to these languages.
Few-shot prompting consistently yields the largest ChrF gains across all four languages, with particularly pronounced effects for ETL and Reactions (significantly better than the baseline for three LLMs).) Grammar prompting alone provides smaller but consistent improvements, likely because it enforces correct syntactic constructs without conveying typical language idioms or usage patterns. The FS + GR combination stabilizes performance at high levels across all models for ETL, QVTo, and Reactions. For the Reactions language, adding helper methods (FS + GR + HM) results in only marginal changes in ChrF.
The Friedman test confirms statistically significant differences among three LLMs for QVTo ($p < 0.05$), whereas differences for ETL do not reach significance.

Finding 1

Few-shot prompting is the primary driver of syntactic similarity across all four MTLs. Grammar prompting alone provides moderate but consistent gains, and its combination with few-shot prompting (FS + GR) consistently achieves the highest ChrF scores.

Unparsed Rate Table 3 reports the average unparsed rate of generated snippets. **For the Reactions language, the baseline unparsed rate is 1.00 across all three models, meaning that no baseline output is syntactically valid.** For QVTo, the baseline is uniformly poor (from 0.90 to 1.00). These results underscore the difficulty that LLMs face with low- or no-resource MTLs. In contrast, ATL, as a widely used MTL with more public resources, exhibits a substantially lower baseline (from 0.20 to 0.27), consistent with its greater prevalence in publicly

	Reactions			ATL			ETL			QVTo		
Strategy	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude
Baseline	0.62	0.60	0.78	0.59	0.57	0.58	0.64	0.61	0.61	0.69	0.73	0.79
Few-shot (FS)	0.87*	0.85*	0.88*	0.64*	0.58	0.59	0.74*	0.71*	0.75*	0.80*	0.80*	0.81
Grammar (GR)	0.80*	0.78*	0.85*	0.59	0.57	0.58	0.63	0.63	0.62	0.64	0.70	0.80
FS + GR	0.86*	0.88*	0.87*	0.63*	0.57	0.59	0.75*	0.71*	0.74*	0.80*	0.80*	0.83
FS + GR + HM	0.86*	0.87*	0.87*	–	–	–	–	–	–	–	–	–

Table 2 Mean ChrF scores (Results significantly better than the baseline ($p < 0.05$) are marked with *. Higher is better). Gray-shaded triples within one strategy show significantly different results of the three LLMs.

	Reactions			ATL			ETL			QVTo		
Strategy	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude
Baseline	1.00	1.00	1.00	0.27	0.27	0.20	0.30	1.00	0.10	1.00	0.90	1.00
Few-shot (FS)	0.58	0.75	0.33*	0.20	0.27	0.20	0.10	0.60	0.10	0.20*	0.10*	0.00*
Grammar (GR)	0.92	0.17*	0.33*	0.60	0.27	0.13	0.30	0.90	0.20	0.40*	0.60	0.90
FS + GR	0.25*	0.42*	0.17*	0.13	0.13	0.13	0.20	0.80	0.20	0.00*	0.00*	0.20*
FS + GR + HM	0.17*	0.5*	0.08*	–	–	–	–	–	–	–	–	–

Table 3 Average unparsed rate (Results significantly better than the baseline ($p < 0.05$) are marked with *. Lower is better). Gray-shaded triples within one strategy show significantly different results of the three LLMs.

available code. ETL shows heterogeneous baseline behavior: Gemini produces no parsable output (1.00), whereas Claude achieves a rate of only 0.10.

Few-shot prompting substantially reduces the unparsed rate, particularly for QVTo, where it drops to 0.00–0.20 across all models. Grammar prompting alone has a limited effect for the Reactions language for GPT, likely because many critical constructs, such as Xtext XExpressions, are not explicitly captured by the grammar. The FS + GR combination achieves the strongest reductions overall, bringing QVTo to 0.00 for both GPT and Gemini. For the Reactions language, the addition of helper methods (FS + GR + HM) yields further improvements, reducing the unparsed rate to 0.08–0.50 depending on the model.

The Cochran's Q test confirms statistically significant differences among three LLMs for ETL ($p < 0.05$).

Finding 2

For low-resource MTLs such as the Reactions language and QVTo, few-shot prompting, grammar prompting, and helper method inclusion provide the most significant gains. For more widely known languages such as ATL and ETL, few-shot prompting alone is generally good to reduce the unparsed rate, even for the more complex transformations present in ATL.

Average Number of Parser-reported Problems per Line of Code (PPL_s) Table 4 reports the average number of parser-reported problems per line of code. For Reactions, the baseline error density ranges between 0.0396 and 0.0562. Few-shot prompting reduces the average number of errors for Gemini and Claude. Grammar prompting alone shows mixed effects: while Gemini benefits substantially (0.0083), GPT error density increases (0.0769). The combined FS + GR strategy stabilizes performance across models and significantly reduces errors for Claude (0.0159). Finally, the addition of helper methods (FS + GR + HM) yields the lowest error density for GPT (0.0145) and remains consistently low for Claude. For ATL, the baseline error density is already low (from 0.0092 to 0.0351), indicating that the models generate largely well-formed ATL code even without optimization. Few-shot prompting and the FS + GR combination further reduce this to near-zero levels. A notable outlier is grammar prompting alone for GPT on ATL (0.2757), which significantly *increases* the error rate compared to the baseline; this suggests that providing the grammar without accompanying examples may mislead certain models into producing syntactically aberrant constructs. For ETL, the baseline varies across models (0.0085 for Claude to 0.0667 for GPT). Few-shot prompting substantially reduces GPT's error rate to 0.0008, while the effect on Gemini and Claude is less pronounced. QVTo exhibits the highest baseline error density (from 0.1119 to 0.3057), consistent with the high unparsed rates observed in Table 3. Both few-shot prompting and the FS + GR combination yield significant reductions, with FS + GR bringing GPT and Gemini to 0.0000 errors per line.

The Friedman test confirms statistically significant differences among three LLMs for ETL ($p < 0.05$) generation with few-shot prompting and for Reactions and ATL with grammar prompting, and for Reactions with the combination of few-shot and grammar prompting with heper method inclusion, whereas the differences for QVTo do not reach significance.

Strategy	Reactions			ATL			ETL			QVTo		
	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude
Baseline	0.0396	0.0562	0.0427	0.0092	0.0351	0.0138	0.0667	0.0644	0.0085	0.3057	0.1674	0.1119
Few-shot (FS)	0.0652	0.0335	0.0289	0.0024	0.0050	0.0072	0.0008	0.0354	0.0085	0.0159*	0.0100*	0.0000*
Grammar (GR)	0.0769	0.0083*	0.0360	0.2757*	0.0204	0.0005	0.0412	0.0401	0.0122	0.1242	0.0784	0.0598
FS + GR	0.0342	0.0437	0.0159*	0.0019	0.0007	0.0007	0.0209	0.0324	0.0105	0.0000*	0.0000*	0.0099*
FS + GR + HM	0.0145*	0.0546	0.0160*	–	–	–	–	–	–	–	–	–

Table 4 Average number of syntax errors per line of code (LoC) (Results significantly better than the baseline ($p < 0.05$) are marked with *. Lower is better). Gray-shaded triples within one strategy show significantly different results of the three LLMs.

Finding 3

The combination of few-shot, grammar prompting, and helper method inclusion (for Reactions only) consistently achieves the lowest syntax error density across MTLs. Grammar prompting alone can be counterproductive for certain LLM models and language combinations.

5.4.2. Semantic Evaluation (EQ2) To answer EQ2, we now turn to semantic correctness, evaluating whether the parsable snippets produce functionally correct transformations.

Pass@1 Table 5 reports the Pass@1 rate, computed exclusively on syntactically parsable snippets. For Reactions, the baseline Pass@1 is 0.00 because the unparsed rate is 1. Few-shot prompting yields improvements, increasing Pass@1 to 0.25–0.50, with statistically significant improvements for Claude. Grammar prompting alone does not improve semantic correctness. The combined FS + GR strategy further improves performance for Gemini and GPT. The addition of helper methods (FS + GR + HM) provides the highest Pass@1 rate for GPT (0.58). However, this improvement is not consistent across all LLM models, as Gemini and Claude do not show additional gains and even decrease. The reason for the marginal contribution of helper method inclusion might be that the few-shot examples serve to show how the helper methods work without having to explicitly include them. However, the primary challenge lies in semantics and reasoning about the transformation rules rather than predefined helper methods.
For ATL, the Pass@1 rate remains at 0.67 across all strategies and models. This result stands in contrast to ATL's strong syntactic performance (cf. Table 2 and Table 4), suggesting that while LLMs are capable of producing syntactically well-formed ATL code, they struggle with the semantic complexity of the underlying transformation logic. As shown in Table 1, ATL scripts in the evaluation suite are considerably longer and more complex than those of other languages (average 96 LoC vs. 18–31 LoC, average 14.2 McCabe complexity vs. 3.4–3.8 McCabe complexity), involving more intricate rule mappings and inter-rule dependencies. The LLMs appears to handle the surface-level syntax of these longer scripts adequately, but fails to capture the deeper semantic intent. For ETL, the baseline Pass@1 is low (from 0.00 to 0.20), and few-shot prompting yields the most substantial improvement, raising Pass@1 to 0.30–0.60 depending on the model. The FS + GR combination achieves comparable results (from 0.10 to 0.50), with GPT and Claude benefiting most. QVTo exhibits the most significant improvement: from a near-zero baseline (0.00–0.10) to 0.60–0.70 with few-shot prompting and 0.50–0.70 with FS + GR. Grammar prompting alone yields negligible improvement, consistent with the pattern observed across all other metrics.
The Cochran's Q test does not reach statistically significant differences among three LLMs.

Finding 4

Few-shot prompting consistently provides the greatest improvements in Pass@1, while combining it with grammar prompting can further semantic correctness in some cases. **Notably, syntactic proficiency does not imply semantic correctness**: for ATL, the LLMs produce well-formed code but fail to capture the semantic complexity inherent in longer and more complex transformation scripts. For QVTo, few-shot prompting transforms the generated code from almost entirely non-functional to majority-correct.

5.4.3. Influence of LLM Choice (EQ3) This evaluation question explores whether the choice of LLM influences the quality of generated model transformation code across MTLs. The results show that the influence of LLM choice is highly variable across MTLs and evaluation metrics. For QVTo, the ChrF scores are significantly different between LLMs for all prompting strategies, reflecting constant differences in syntactic similarity (see gray shades in Table 2). Likewise, ETL exhibits a constant and statistically significant difference between LLMs from parsability and PPL_s across multiple strategies (see gray shades in Tables 3 and 4), reflecting that syntactic correctness is sensitive to the LLM used for generation. The Reactions Language shows statistically significant LLM differences mainly in parsability and PPL_s for several strategies, reflecting a moderate sensitivity. In contrast, ATL exhibits only isolated or no significant differences between LLMs across most strategies and metrics.
Despite these syntactic corrections and similarity differences, the Pass@1 rate does not show consistent statistically significant variation between LLMs across any MTL (see gray shades in Table 5), indicating that semantic correction is less dependent on the choice of LLMs.

Strategy	Reactions			ATL			ETL			QVTo		
	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude	GPT	Gemini	Claude
Baseline	0.00	0.00	0.00	0.67	0.67	0.67	0.20	0.00	0.10	0.00	0.10	0.00
Few-shot (FS)	0.42	0.25	0.50*	0.67	0.67	0.67	0.60	0.30	0.60	0.60*	0.60	0.70*
Grammar (GR)	0.00	0.00	0.00	0.67	0.67	0.67	0.10	0.00	0.00	0.00	0.20	0.10
FS + GR	0.50*	0.42	0.50*	0.67	0.67	0.67	0.50	0.10	0.50	0.50	0.70*	0.60*
FS + GR + HM	0.58*	0.42	0.42	–	–	–	–	–	–	–	–	–

Table 5 Pass@1 rate on syntactically parsable outputs (Results significantly better than the baseline ($p < 0.05$) are marked with *. Higher is better). Gray-shaded triples within one strategy show significantly different results of the three LLMs.

Finding 5

LLM Model choice influences syntactic correctness and similarity, particularly for ETL and QVTo respectively, while its influence on semantic correction remains limited.

6. Discussion

6.1. Limitations

This study is limited in language and model scope: four MTLs (Reactions language, ATL, ETL and QVTo) are evaluated across three LLMs (GPT-5.1, Gemini 2.5-Pro, Claude Sonnet 4.5), which constrains generalizability, in particular for other low-resource languages. The strategy portfolio focuses on few-shot, grammar prompting, and helper method inclusion; more advanced techniques, such as Retrieval-augmented generation (RAG) or fine-tuning were not explored due to time and data constraints. The metric set is also constrained: ChrF is surface-based and pass@1 captures only the first attempt. Additionally, nondeterminism remains an inherent risk despite fixed parameters. The workflow is best characterized as structured and extensible rather than fully automated end-to-end, i.e., adaptation to a new MTL still requires language-specific manual effort, including system prompt redesign, resource curation, and test suite development. The current evaluation setup based on reverse-engineered prompts is systematic and reproducible, but may not fully represent real user prompt distributions, and whether the workflow performs equally well on underspecified or messy/misleading natural-language requests remains an open question. A further limitation concerns potential training data contamination, as publicly available resources such as *ATL Zoo Benchmark* (2026) may have been included in the training corpora of the LLMs.

6.2. Threats to Validity

Internal Validity. LLM non-determinism can shift results, and technical configuration (prompts, metamodel setup) influences all metrics. However, variations in prompt formulation or model temperature settings could still affect reproducibility across different experimental runs.

External Validity. This study was evaluated on only four MTLs with limited dataset sizes, which restricts the generalizability of the results. The results may not represent the characteristics of all MTLs. Furthermore, we tested only three commercial LLMs, and the performance of open-source models or future models may differ. Future work needs to validate the effectiveness of the workflow on more MTLs and larger-scale datasets.

Construct Validity. The metrics we adopted (ChrF, Unparsed Rate, Pass@1, etc.) may not fully capture all dimensions of code quality. For example, ChrF focuses on character-level similarity and may assign lower scores to functionally equivalent code with different structures. Pass@1 only evaluates the correctness of the first generation, without considering the potential for iterative refinement. Additionally, the coverage of test cases may affect the assessment of semantic correctness. We mitigate this by combining syntactic and semantic metrics for comprehensive evaluation, but a more complete quality assessment workflow remains to be explored.

Conclusion Validity. Statistical significance testing (ANOVA) confirmed differences among strategies, but the extremely poor baseline performance of the Reactions language (Unparsed Rate of 1.00) inflates the relative improvement of the strategies, though the absolute performance remains moderate (e.g., Pass@1 up to 0.33). For ATL, improvements from certain strategies did not reach statistical significance or even showed no improvement (e.g., Pass@1), indicating that result stability varies across different MTLs. These findings suggest that the effectiveness of the workflow may depend on the characteristics of the target language. More broadly, the findings highlight a disconnect between syntactic and semantic improvement: Although prompt engineering can improve syntax correctness, especially for low-resource MTLs, it is inconsistent when it comes to the improvement of semantic correctness for more complex transformations. This is consistent with the bottleneck situations across different MTLs. For low-resource MTLs such as the Reactions language, the primary bottleneck lies in the LLM's insufficient syntax knowledge of the MTL, which can be largely mitigated through few-shot prompting; whereas for more widely used languages such as ATL, the bottleneck shifts to semantic and domain grounding, which current prompting strategies have yet to fully address.

6.3. Workflow Generalizability and Extensibility

The workflow's modular design facilitates adaptation to new MTLs. As demonstrated in subsection 4.3, extending to other MTLs primarily required adjusting system prompts and curat-

ing language-specific resources (grammar files, examples, test suites), while the core pipeline remained unchanged. This suggests that similar adaptations could apply to other MTLs, though the effectiveness may vary depending on language characteristics and available training data. Beyond the current prompting strategies, the workflow can accommodate more advanced techniques. RAG could augment prompts with dynamically retrieved domain knowledge, addressing the challenge of limited MTL training data. Fine-tuning, while resource-intensive, may become viable as more high-quality MTL datasets emerge. Additionally, automating parser creation and test execution would further reduce manual effort, making the workflow accessible to a broader range of MTL practitioners.

7. Conclusion and Future Work

This paper presented *LLM4MTLs*, an automated and reproducible workflow for improving the reliability of LLM-generated MTL code and systematically evaluating its quality, together with an evaluation suite spanning four MTLs and an empirical evaluation on that suite. Rather than modifying or fine-tuning LLMs, the workflow treats each LLM as a black-box code generator and focuses exclusively on prompt construction as the lever for improvement. It provides an automated pipeline that standardizes prompt generation, code generation, and evaluation across MTLs. Using this workflow, we conducted an empirical evaluation on an evaluation suite spanning the Reactions language, ATL, ETL, and QVTo, with different prompting strategy combinations across three LLMs, assessing both syntactic quality and semantic correctness. The few-shot prompting is the primary driver of syntactic improvement, though its ability to improve semantic correctness diminishes for complex transformations, such as ATL. Grammar prompting stabilizes generation when combined with few-shot examples, but can be counterproductive in isolation. Finally, LLM Model choice influences syntactic correctness and similarity, particularly for ETL and QVTo respectively, while its influence on semantic correctness remains limited across all MTLs.

Future Work Several directions emerge from this work. First, we plan to explore the use of *agentic AI* approaches that incorporate iterative refinement loops, where the LLM receives parser or test results feedback and refines its output autonomously. Second, *RAG* could dynamically select the most relevant examples and grammar fragments for a given task. Additionally, *grammar-constrained decoding* offers a promising direction for eliminating parse errors. Another direction avenue to pursue includes qualitative analysis of semantic errors. While the present approach evaluates the generated transformation to see if the output models match the correct semantics, there is no investigation as to why these output models fail, for instance, whether model elements are missing or structurally misplaced. By conducting such an analysis, more actionable feedback for targeted prompt refinement could be gained.

Acknowledgements

This paper is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263. Thanks to our Textician Dan Shea.

References

Abukhalaf, S., Hamdaqa, M., & Khomh, F. (2024, April 14th). PathOCL: Path-Based Prompt Augmentation for OCL Generation with GPT-4. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering* (pp. 108–118). ACM. doi: 10.1145/3650105.3652290

Amalfitano, D., Metzger, A., Autili, M., Fulcini, T., Hey, T., Keim, J., ... Vogelsang, A. (2025). A research roadmap for augmenting software engineering processes and software products with generative AI. *CoRR*, *abs/2510.26275*. Retrieved from https://doi.org/10.48550/arXiv.2510.26275 doi: 10.48550/ARXIV.2510.26275

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2007). Uml2alloy: A challenging model transformation. In *International conference on model driven engineering languages and systems* (pp. 436–450).

Anjorin, A., Buchmann, T., & Westfechtel, B. (2017). The families to persons case. In A. García-Domínguez, G. Hinkel, & F. Krikava (Eds.), *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), marburg, germany, july 21, 2017* (Vol. 2026, pp. 27–34). CEUR-WS.org.

Atkinson, C., Stoll, D., & Bostan, P. (2010). Orthographic Software Modeling: A Practical Approach to View-Based Development. In L. A. Maciaszek, C. González-Pérez, & S. Jablonski (Eds.), *Evaluation of Novel Approaches to Software Engineering* (pp. 206–219). Springer.

Atl zoo benchmark. (2026). Retrieved from https://github.com/atlanmod/mondo-atlzoo-benchmark

Banerjee, S., & Lavie, A. (2005, June). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In J. Goldstein, A. Lavie, C.-Y. Lin, & C. Voss (Eds.), *Proceedings of the ACL workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization* (pp. 65–72). Association for Computational Linguistics.

Bassamzadeh, N., & Methani, C. (2024, July 3rd). *A comparative study of DSL code generation: Fine-tuning vs. optimized retrieval augmentation* (No. arXiv:2407.02742). arXiv. doi: 10.48550/arXiv.2407.02742

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-Driven Software Engineering in Practice*. Springer International Publishing. doi: 10.1007/978-3-031-02549-5

Brun, C., & Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29–34.

Buchmann, T. (2024, October 31st). Prompting Bidirectional Model Transformations - The Good, The Bad and The Ugly. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems* (pp. 550–555). Association for Computing Machinery. doi: 10.1145/3652620.3687802

Burgueño, L., Di Ruscio, D., Sahraoui, H., & Wimmer, M. (2025, May). Automation in model-driven engineering: A look back, and ahead. *ACM Trans. Softw. Eng. Methodol.*, *34*(5). doi: 10.1145/3712008

Cibrián, E., Olivert Iserte, J. F., Casella, F., García Rodríguez, M., Alvarez-Rodríguez, J. M., & Llorens, J. (2025, October 4th). *Automating Model Transformation from Modelica to SysML v2 with LLM-based Agents* (SSRN Scholarly Paper No. 5564442). doi: 10.2139/ssrn.5564442

Claude Sonnet 4.5. (2025). Retrieved 2025-09-11, from https://www.anthropic.com/claude/sonnet

Cámara, J., Troya, J., Burgueño, L., & Vallecillo, A. (2023, June 1st). On the assessment of generative AI in modeling tasks: An experience report with ChatGPT and UML. *Software and Systems Modeling*, *22*(3), 781–793. doi: 10.1007/s10270-023-01105-5

Eclipse epsilon. (2026). Retrieved from https://github.com/eclipse-epsilon/epsilon (Accessed: 2026-02-17)

Eisenberg, M., Klikovits, S., Wimmer, M., & Wielan, K. (2025, October 8). Towards LLM-enhanced Conflict Detection and Resolution in Model Versioning. In *Proceedings of the ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems* (p. 267 - 273). IEEE. doi: 10.1109/MODELS67397.2025.00032

Foundation, E. (2018). *M2M/ATL/Syntax.* Retrieved from https://wiki.eclipse.org/M2M/ATL/Syntax (Accessed December, 2025)

Garaccione, G., Calabrese, D. M., Coppola, R., & Ardito, L. (2025, October). A Comparison of Different Large Language Models for the Generation of UML Class Diagrams. Association for Computing Machinery.

Gemini 2.5 Pro. (2025). Retrieved 2025-09-11, from https://deepmind.google/models/gemini/pro/

Götz, S., Tichy, M., & Kehrer, T. (2021). Dedicated model transformation languages vs. general-purpose languages: A historical perspective on atl vs. java. In *Modelsward* (pp. 122–135).

Gpt-5.1. (2025). Retrieved 2025-09-11, from https://platform.openai.com

Hagel, N., Hili, N., Bartel, A., & Koziolek, A. (2025). Towards llm-powered consistency in model-based low-code platforms. In *22nd IEEE international Conference on Software Architecture, ICSA - Companion, Odense, Denmark, March 31 - April 4, 2025* (pp. 364–369). IEEE. doi: 10.1109/ICSA-C65153.2025.00058

Hagel, N., Hili, N., & Schwab, D. (2024). Turning low-code development platforms into true no-code with llms. In M. Wimmer, A. Egyed, B. Combemale, & M. Chechik (Eds.), *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS companion 2024, Linz, Austria, september 22-27, 2024* (pp. 876–885). ACM. doi: 10.1145/3652620.3688334

Howell, D. C. (1992). *Statistical methods for psychology*. PWS-Kent Publishing Co.

Höppner, S., Haas, Y., Tichy, M., & Juhnke, K. (2022, August 17th). Advantages and Disadvantages of (Dedicated) Model Transformation Languages. *Empirical Software Engineering*, *27*(6), 159. doi: 10.1007/s10664-022-10194-7

Ishimizu, Y., Yamauchi, T., Chen, S., Cai, J., Li, J., & Tei, K. (2025). Automatic syntax error repair for discrete controller synthesis using large language model. *CoRR*, *abs/2512.07261*. Retrieved from https://doi.org/10.48550/arXiv.2512.07261 doi: 10.48550/ARXIV.2512.07261

Jiang, B., Hagel, N., & Cheng, H. (2026). *LLM-based Code Generation for Model Transformation Language* (Replication Package No. 19683666). Zenodo. Retrieved 2026-06-05, from https://doi.org/10.5281/zenodo.19683666 doi: 10.5281/zenodo.19683666

Joel, S., Wu, J., & Fard, F. (2024). A survey on llm-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*.

Joel, S., Wu, J., & Fard, F. (2025, October 7th). A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. *ACM Transactions on Software Engineering and Methodology*, 3770084. doi: 10.1145/3770084

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008, June 1st). ATL: A Model Transformation Tool. *Science of Computer Programming*, *72*(1), 31–39. doi: 10.1016/j.scico.2007.08.002

Klare, H. (2021). Building Transformation Networks for Consistent Evolution of Interrelated Models. doi: 10.5445/IR/1000133724

Klare, H., Kramer, M. E., Langhammer, M., Werle, D., Burger, E., & Reussner, R. (2021, January). Enabling Consistency in View-Based System Development — The Vitruvius Approach. *Journal of Systems and Software*, *171*, 35. doi: 10.1016/j.jss.2020.110815

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA explained - the Model Driven Architecture: practice and promise*. Addison-Wesley. Retrieved from http://www.informit.com/store/mda-explained-the-model-driven-architecture-practice-9780321194428

Kolovos, D., & Garcia-Dominguez, A. (2022). The Epsilon Playground. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (p. 131–137). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3550356.3556507

Li, J., Zhang, M., Li, N., Weyns, D., Jin, Z., & Tei, K. (2024a). Exploring the potential of large language models in self-adaptive systems. In L. Baresi, X. Ma, & L. Pasquale (Eds.), *Proceedings of the 19th international symposium on software engineering for adaptive and self-managing systems, SEAMS 2024, lisbon, portugal, april 15-16, 2024* (pp. 77–83). ACM. Retrieved from https://doi.org/10.1145/3643915.3644088 doi: 10.1145/3643915.3644088

Li, J., Zhang, M., Li, N., Weyns, D., Jin, Z., & Tei, K. (2024b). Generative AI for self-adaptive systems: State of the art and research roadmap. *ACM Trans. Auton. Adapt. Syst.*, *19*(3), 13:1–13:60. Retrieved from https://doi.org/10.1145/3686803 doi: 10.1145/3686803

Lin, C.-Y. (2004, July). ROUGE: A package for automatic

evaluation of summaries. In *Text summarization branches out* (pp. 74–81). Association for Computational Linguistics.
Mens, T., & Van Gorp, P. (2006, March). A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, *152*, 125–142. doi: 10.1016/j.entcs.2005.10.021
Meta Object Facility [Standard]. (2019, October 1st). Retrieved from https://www.omg.org/spec/MOF/2.5.1
N8n.io - AI workflow automation tool. (2025). Retrieved from https://n8n.io/
OMG. (2014, December). *Object Constraint Language.* Retrieved 2024-03-07, from https://www.omg.org/spec/OCL/2.4/PDF
OMG. (2016, June). *About the MOF Query/View/Transformation Specification Version 1.3.* Retrieved from https://www.omg.org/spec/QVT/1.3/About-QVT
org.eclipse.qvto. (2026). Retrieved from https://github.com/eclipse-qvto/org.eclipse.qvto (Accessed: 2026-02-17)
Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2001). BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02* (p. 311). Association for Computational Linguistics. doi: 10.3115/1073083.1073135
Paul, D. G., Zhu, H., & Bayley, I. (2024, June 18th). *Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review* (No. arXiv:2406.12655). arXiv. (version: 1) doi: 10.48550/arXiv.2406.12655
Pontes Miranda, J. W., Bruneliere, H., Tisi, M., & Sunyé, G. (2024, October 17th). Towards an In-Context LLM-Based Approach for Automating the Definition of Model Views. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering* (pp. 29–42). ACM. doi: 10.1145/3687997.3695650
Popović, M. (2015, September). chrF: character n-gram f-score for automatic MT evaluation. In O. Bojar et al. (Eds.), *Proceedings of the Tenth Workshop on Statistical Machine Translation* (pp. 392–395). Association for Computational Linguistics. doi: 10.18653/v1/W15-3049
Pryzant, R., Iter, D., Li, J., Lee, Y. T., Zhu, C., & Zeng, M. (2023, May). Automatic prompt optimization with "gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*. doi: 10.48550/arXiv.2305.03495
Vitruv-casestudies. (2026). Retrieved from https://github.com/vitruv-tools/Vitruv-CaseStudies (Accessed: 2026-02-17)
Wang, B., Wang, Z., Wang, X., Cao, Y., A Saurous, R., & Kim, Y. (2023). Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, *36*, 65030–65055.
Ye, Q., Axmed, M., Pryzant, R., & Khani, F. (2023). Prompt engineering a prompt engineer. *arXiv preprint arXiv:2311.05661*.
Zhang, W., Hebig, R., & Strüber, D. (2025). Leveraging llms to support co-evolution between definitions and instances of textual dsls. In *Proceedings of the first large language models for software engineering workshop (llm4se), co-located with staf 2025.* Koblenz, Germany.
Zhang, W., Jiang, B., Fu, Y., Cheng, H., Hummel, M., Scotti, V., … Koziolek, A. (2026). *Large language models in model-driven engineering: A systematic mapping study.* Karlsruhe Institute of Technology (KIT). Retrieved from https://publikationen.bibliothek.kit.edu/1000193410 doi: 10.5445/IR/1000193410
Zhang, W., Jiang, B., Fu, Y., Koziolek, A., Hebig, R., & Strüber, D. (2026). Leveraging llms to support co-evolution between definitions and instances of textual dsls: A systematic evaluation. *arXiv preprint arXiv:2602.11904*.

About the authors

Bowen Jiang is a PhD researcher at Karlsruhe Institute of Technology (KIT), Germany. She received her M.Sc. from WASEDA University, Japan. Her research interests include MDE, software testing, and AI4SE. You can contact the author at bowen.jiang@kit.edu or visit https://mcse.kastel.kit.edu/staff_bowen_jiang.php.

Nathan Hagel is a PhD researcher at KIT, Germany. His research interests include MDE, uncertainty management for cyber-physical systems, DSLs, and cloud systems. You can contact the author at nathan.hagel@kit.edu or visit https://mcse.kastel.kit.edu/staff_nathan_hagel.php.

Haowei Cheng is a PhD student at Waseda University, Japan. You can contact the author at haowei.cheng@fuji.waseda.jp or visit https://haowei614.github.io/.

Benedikt Jutz is a PhD researcher at KIT. His research interests include concurrent editing techniques for multiple models and DSLs. You can contact the author at benedikt.jutz@kit.edu or visit https://dsis.kastel.kit.edu/staff_benedikt_jutz.php.

Arne Lange is a Postdoctoral researcher at KIT. His research interests include multi-level modeling and model consistency preservation. You can contact the author at arne.lange@kit.edu or visit https://dsis.kastel.kit.edu/staff_arne_lange.php.

Weixing Zhang is a PostDoc at KIT. He received his PhD at the University of Gothenburg. His research interests include SE, Empirical SE, AI4SE.You can contact the author at weixing.zhang@kit.edu or visit https://wilson008.github.io/.

Rahul Sharma is a PostDoc at Karlsruhe Institute of Technology. You can contact the author at rahul.sharam@kit.edu or visit https://dsis.kastel.kit.edu/staff_rahul_sharma.php.

Ralf Reussner is a professor at KIT since 2006. His research interests include software architecture, predictable software quality, and view-based design methods for software-intensive technical systems. You can contact the author at reussner@kit.edu or visit https://dsis.kastel.kit.edu/staff_ralf_reussner.php.

Anne Koziolek is a professor at KIT, Germany. She received her PhD degree from KIT in 2011. She is interested in MDE and agile development processes. You can contact the author at koziolek@kit.edu or visit https://mcse.kastel.kit.edu/staff_Koziolek_Anne.php.