Change-Aware Round-Trip Benchmarking of LLMs for Reliable and Efficient Artifact Co-Evolution★

Duy Dao[a], Alessio Bucaioni[a] and Antonio Cicchetti[a]

[a]*Mälardalen University, Västerås, Sweden*

ARTICLE INFO

Keywords:
Software Artifact Co-evolution
Round-trip Consistency
Large Language Models for Software Evolution

ABSTRACT

Large language models are increasingly embedded in software development, yet most evaluations still treat them as one-shot generators for isolated tasks such as code completion or refactoring. In real workflows, however, artifacts such as application programming interfaces, data models, and database schemas co-evolve, and changes must propagate across representations without breaking consistency. When propagation fails, developers incur extra validation, retries, and manual repair, which increases latency and infrastructure cost and undermines sustainable operation.

In this study, we ask whether large language models can preserve cross-artifact consistency under change in a round-trip workflow. We apply a controlled edit to one artifact, translate it to its coupled counterpart, and translate it back, then check whether the intended edit persists without drift (i.e., unintended semantic changes or syntactic invalidity). We instantiate this question by synchronizing class-oriented data models with relational database schemas. Using a curated dataset of paired models and schemas and a suite of controlled edit operations, we evaluate four large language model, GPT-5, Qwen3-Next-80B-A3B, DeepSeek V3, and Gemini 2.5, under a unified, reproducible protocol that measures (i) edit persistence, (ii) structural validity (parsability/loadability), and (iii) run-to-run consistency over repeated executions.

Our results show that the models handle small, routine edits reliably, but they struggle when edits require structural reorganization. Gemini 2.5 is the most consistent across runs; DeepSeek V3 often preserves the intended semantics but occasionally produces unparsable outputs; Qwen3-Next-80B-A3B exhibits high variance; and GPT-5 often recognizes the change but fails to propagate it coherently through the coupled representation.

We contribute a reproducible benchmark and evaluation framework for assessing LLM reliability under artifact co-evolution, together with empirical evidence of current limitations. Overall, the findings reveal a gap between detecting a change and propagating it coherently, underscoring the need for structural validation and human oversight to achieve dependable and cost-efficient LLM-assisted software evolution.

1. Introduction

Modern software systems are increasingly co-developed with large language models (LLMs), yet most evaluations still assess LLMs as one-shot generators for isolated tasks such as code completion or refactoring Jiang, Wang, Shen, Kim and Kim (2024). In realistic workflows, however, artifacts such as application programming interfaces, domain models, configuration files, and database schemas evolve continuously and must remain consistent as changes propagate across representations Qiu, Li and Su (2013). When this consistency breaks, teams face reliability risks (e.g., regressions and invalid migrations) and expensive rework, and they often fall back to repeated validation and regeneration cycles Stevens (2020). These cycles increase latency and infrastructure cost and amplify energy consumption, which directly undermines trust in LLM-assisted, iterative development and the sustainable use of AI tooling Ozkaya (2023); Laskar, Alqahtani, Bari, Rahman, Khan, Khan, Jahan, Bhuiyan, Tan, Parvez, Hoque, Joty and Huang (2024).

Empirical studies of software evolution show that application code and database schemas are tightly coupled and often evolve together as requirements change, architectures are refactored, and domain understanding improves Qiu et al. (2013). Common evolution patterns include renaming entities, splitting or merging concepts, introducing inheritance hierarchies to factor out shared attributes, normalizing free-form values into enumerated types, and adjusting multiplicities to enforce new business constraints Wimmer, Pérez, Jouault and Cabot (2012). These changes are pervasive in long-lived systems, including microservice-based architectures with relational persistence, where developers must propagate edits across artifacts while preserving semantic alignment and operational correctness.

In this work, we investigate whether LLMs can support artifact co-evolution by preserving consistency across coupled artifacts under change. We evaluate LLMs with a change-aware round-trip process: we apply a controlled change to one artifact, use the LLM to update its paired counterpart, and translate the result back to test whether the intended change persists without drift and whether both artifacts remain usable. We treat an outcome as consistent when (i) the intended modification is preserved after the

★This work is supported by: (a) the Swedish Agency for Innovation Systems through the projects "Secure: Developing Predictable and Secure IoT for Autonomous Systems" (2023-01899), (b) the Key Digital Technologies Joint Undertaking through the project "MATISSE: Model-based engineering of digital twins for early verification and validation of industrial systems" (101140216), and (c) the Clean Energy Transition Partnership through the project "FLEXI: Human-centered AI and digital twin powered energy system integration for flexibility markets" (101069750).

khanh.duy.dao@mdu.se (D. Dao); alessio.bucaioni@mdu.se (A. Bucaioni); antonio.cicchetti@mdu.se (A. Cicchetti)
ORCID(s):

round trip and (ii) both artifacts remain valid and aligned, meaning they can be parsed/loaded by standard tooling and still correspond to each other. Because reliability in practice also depends on predictability, we repeat each scenario across runs to measure run-to-run consistency under identical prompts and inputs.

We adopt the notion of round-trip from Model-Driven Engineering (MDE) An, Hu and Song (2008), and we take inspiration from checks used to validate model synchronizations Stevens (2007). In MDE, developers typically implement synchronizations through hand-written transformation programs; in contrast, we entrust all artifact manipulations to LLMs, without any predefined transformation code. We use the round-trip principle strictly as a testing mechanism, and we scope our setting to a single, well-defined instance: change-aware consistency preservation between class-oriented data models and relational database schemas under controlled edits. This scope lets us study LLM behavior precisely while still reflecting a common and well-documented co-evolution scenario. Our central research question is: can LLMs propagate and preserve changes across coupled artifacts while maintaining validity and alignment?

To ensure a meaningful evaluation and a reproducible ground truth, we rely on a well-established benchmark scenario and dataset from the community[1]. From this dataset, we selected ten pairs of coupled UML class diagrams and relational database schemas. For each pair, we applied ten incremental changes with varying complexity, ranging from simple edits such as renaming a class or attribute to structural modifications such as introducing new relationships or splitting entities. For each change, we executed the round-trip scenario and checked whether the modification persisted while both artifacts remained parseable and mutually consistent. Using this setup, we evaluate four state-of-the-art LLMs: GPT-5, Qwen3-Next-80B-A3B, DeepSeek V3, and Gemini 2.5, under a unified, reproducible protocol with identical prompts and evaluation criteria, and we characterize which edit types the evaluated LLM-based transformation systems handle reliably and where they fail. In total, the experiment includes ten base pairs, ten Δ configurations, four LLMs, and two transformation steps, resulting in 800 generated artifacts ($10 \times 10 \times 4 \times 2$).

Our results show that LLMs handle small, routine edits reliably but struggle with larger, structure-altering modifications. Among the evaluated LLMs, Gemini 2.5 demonstrates the most consistent and dependable performance across scenarios. DeepSeek V3 often produces semantically correct edits yet occasionally generates outputs that cannot be parsed or loaded by standard tools. Qwen3-Next-80B-A3B exhibits high variance across runs, alternating between near-perfect and incomplete transformations. GPT-5 frequently identifies the required change correctly but does not always propagate it coherently to the paired artifact.

[1] https://github.com/eMoflon/benchmarx/tree/main/examples/ecoretosql/BenchmarxEcoreToSQL/resources

In summary, this paper makes the following contributions:

1. A change-aware round-trip protocol for evaluating coupled UML class diagrams and relational database schemas under edits of varying complexity;
2. A comparative benchmark of four LLMs (GPT-5, Qwen3-Next-80B-A3B, DeepSeek V3, and Gemini 2.5) on a curated set of change scenarios, using standardized prompts and transparent, reproducible checks;
3. Empirical findings showing that LLMs perform reliably on small syntactic edits but struggle with structure-altering changes, exposing a gap between detecting and coherently propagating modifications and indicating where human oversight or auxiliary validation is required; and
4. An open research artifact set (prompts, inputs/outputs, and scoring scripts) to support replication and further studies in LLM-assisted software analysis and evolution.

The remainder of the paper is organized as follows. Section 2 reviews the background and related work. Section 3 details our methodology, including delta design, prompting, and evaluation. Section 4 presents our empirical results. Section 5 discusses implications for model-based and LLM-assisted software evolution. Finally, Section 6 concludes with avenues for future research.

2. Background and Related Work

This section introduces the foundational concepts needed to understand the remainder of the paper and reviews relevant related work.

2.1. Round-trip

Software artifacts such as schemas, configuration files, and models rarely evolve in isolation. When one artifact changes, corresponding updates are often required in others to preserve overall system consistency An et al. (2008); Stevens (2020). Engineers, therefore, need mechanisms to detect changes, map linked elements, propagate edits across artifacts, and verify that consistency still holds. We adopt a round-trip consistency test in which we introduce a change in one artifact, propagate it to its coupled counterpart, and then translate it back to the original representation. We then compare the outcomes to check that the intended edit persists and that no unintended deviations occur (drift), including semantic changes or syntactic invalidity. This idea aligns with classical bidirectional maintenance principles (e.g., check-then-enforce and round-trip laws), where a forward-then-back or back-then-forward cycle should return an equivalent state Stevens (2007). In practice, we view round-trip consistency as a five-step process: (i) observe or receive a change; (ii) identify corresponding elements between the two artifacts; (iii) propose a minimal, semantics-preserving propagation; (iv) validate syntactic and semantic constraints (e.g., loadability and invariants); and (v) surface ambiguities to human

reviewers when information is incomplete or mappings are non-bijective. Human-in-the-loop verification is common in software analysis and evolution, where measurements and design models must remain synchronized Buchmann and Westfechtel (2013); Greiner, Buchmann and Westfechtel (2016). Typical challenges include information loss across representations, conflicts from concurrent edits, tool and data interoperability issues, and scalability in heterogeneous environments Rahm, Graube and Urbas (2017).

2.2. Round-Trip Consistency in Practical Co-Evolution Scenarios

To illustrate why round-trip consistency matters in practice, consider a microservice that manages customer and order data with a relational persistence layer. As requirements evolve, a previously monolithic entity may be decomposed to separate billing and shipping concerns, a free-text status field may be replaced with a controlled enumeration, or shared properties may be lifted into a superclass to reduce duplication. Each change requires coordinated updates across application-level models and database schemas, including constraints, relationships, and mappings. In these scenarios, developers must ensure that edits applied on one side propagate consistently to the other and that subsequent changes do not introduce drift or regressions. An LLM-based assistant that can perform this propagation reliably, and that can signal uncertainty when information is missing, could reduce manual effort while improving dependability. Our round-trip protocol operationalizes this requirement by testing whether controlled changes persist across coupled artifacts when we propagate them forward and backward without predefined transformation logic. We conduct our experiments in a controlled benchmark setting rather than on mined industrial evolution histories. This design provides precise ground truth, systematic variation of change types, and fair comparison across models, while abstracting away from project-specific noise and irregular evolution sequences. We discuss the implications of this abstraction and the need for complementary studies on real evolution traces in Section 5.

2.3. UML-SQL as a Round-Trip Benchmark

Although transforming UML class diagrams to relational schemas is sometimes perceived as straightforward, the model transformation community uses this setting precisely because it exposes non-trivial challenges France, Kim, Ghosh and Song (2004); Ceri and Fraternali (1997). Object-relational mappings must address structural and semantic mismatches between object-oriented and relational representations, such as inheritance flattening strategies, association reification, multiplicity encoding, and constraint preservation. Flattening UML class hierarchies into tables, for example, requires navigating inheritance relationships (often transitively), while the reverse transformation from SQL to UML is inherently lossy because relational schemas do not explicitly encode hierarchy information Stevens (2007); Vassiliadis, Shehaj, Kalampokis and Zarras (2023). As a result, reconstructing the original UML structure from SQL cannot, in general, be performed with full precision. These characteristics make UML–SQL transformations challenging in round-trip and co-evolution settings: forward mappings must resolve structural mismatches, while backward mappings must cope with information loss and non-bijective correspondences Vassiliadis et al. (2023). In this sense, the core difficulty is not syntactic translation, but preserving semantic intent under evolution while maintaining cross-representation consistency.

2.4. Large Language Models

LLMs can lower the entry barrier in software engineering workflows by generating code and structured artifacts from natural-language or structured prompts. At the same time, they raise concerns about correctness, explainability, bias, and data governance. Ozkaya et al. Ozkaya (2023) position LLMs as partners in an AI-augmented software lifecycle and emphasize responsible use with respect to quality, privacy, and trust. Complementary surveys argue that capability claims should be grounded in systematic benchmarks, especially for reasoning, robustness, and consistency, properties that are central for transformation and synchronization pipelines Laskar et al. (2024); Chang, Wang, Wang, Wu, Yang, Zhu, Chen, Yi, Wang, Wang, Ye, Zhang, Chang, Yu, Yang and Xie (2024). To assess round-trip consistency, which requires forward derivation, backward synchronization, and preservation of the intended delta while keeping artifacts loadable, we evaluate a set of LLMs that reflect practical trade-offs in reasoning reliability, long-context handling, and computational efficiency. These trade-offs matter for real deployments because repeated propagation and validation steps can amplify both cost and error rates. GPT-5 is OpenAI's 2025 LLM designed to balance rapid response generation with extended reasoning depth[2]. DeepSeek V3 uses an efficiency-oriented Mixture-of-Experts (MoE) architecture with sparse activation to reduce computational cost while maintaining competitive reasoning quality[3]. Qwen3-Next-80B-A3B is a sparse-activation LLM optimized for long-context processing and reasoning throughput, with reported competitive results at lower cost[4]. Gemini 2.5, developed by Google DeepMind, exposes configurable reasoning budgets to trade off accuracy and latency dynamically[5].

2.5. Related Work

In LLM evaluation, Allamanis et al. Allamanis, Panthaplackel and Yin (2024) introduce round-trip correctness (RTC) as an unsupervised method that applies an operation and its inverse and checks for equivalence, and they report strong correlation with curated benchmarks. Ruiz et al. Ruiz, Grishina, Hort and Moonen (2024) use round-trip translation (RTT) for automated program repair, showing that translating code through an intermediate representation and back

[2] https://openai.com/chatgpt
[3] https://github.com/deepseek-ai/DeepSeek-V3
[4] https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list
[5] https://deepmind.google/models/gemini/pro/

can improve correctness without task-specific fine-tuning; surveys further consolidate this landscape and emphasize retrieval- and analysis-augmented prompting Yang, Cai, Liu, Le, Zhang, Bissyandé, Liu and Tian (2025). Compared to these works, we extend RTC-style validation beyond same-language checks by testing whether a change can traverse a model-to-schema boundary and back while still satisfying metamodel and schema validity.

Round-trip ideas also appear in learning and tooling. In unsupervised code translation, Lachaux et al. Lachaux, Roziere, Chanussot and Lample (2020) use back-translation (a forward-and-back cycle) to reduce reliance on direct supervision. In software engineering tools, round-trip engineering aims to keep models and code synchronized across edits on either side, and empirical comparisons of UML tooling emphasize maintaining consistency across such edits Rosca and Domingues (2021). Our study complements these lines by measuring per-change preservation and forward/backward asymmetries across a model-to-schema boundary.

Empirical studies show that database schemas and application code co-evolve tightly, and maintaining alignment between them remains a persistent challenge Qiu et al. (2013). Subsequent research addresses schema evolution and migration in developer workflows Vassiliadis et al. (2023); Campos and Affonso (2025). Our setting instantiates co-evolution in a controlled form, allowing us to test whether LLMs can propagate changes across artifacts and back while preserving intended edits and satisfying conformance constraints.

Recent work has begun to explore LLMs for model transformation tasks directly within MDE pipelines. Kazai et al. Kazai, Osei, Bucaioni and Cicchetti (2025) investigate whether LLMs can perform UML-to-Java transformations out-of-the-box, reporting high cumulative success rates for simple models but significant degradation as structural complexity increases, which motivates the need for stricter validation and structured feedback loops rather than one-shot generation.

Our work is also connected to refactoring and artifact co-evolution. Prior studies report recurring evolution patterns, including renames, entity decomposition, hierarchy extraction, and type normalization, often driven by changing domain understanding or non-functional requirements Wimmer et al. (2012). Mapping these changes across artifact boundaries remains difficult, which motivates automated support for change propagation and consistency checking Kretschmer, Khelladi, Lopez-Herrejon and Egyed (2021). By operationalizing these patterns as explicit Δ-operators and evaluating their preservation under round-trip transformations, we characterize how LLMs behave in co-evolution tasks rather than in one-shot generation tasks.

Complementary evidence is provided by Moezkarimi et al. Moezkarimi, Eriksson, Johansson, Bucaioni and Sirjani, who study LLM-based transformations from UML state diagrams to Rebeca models for formal verification, showing that LLMs can reduce transformation effort but require additional metadata and iterative correction to satisfy target-language constraints. Their findings further highlight the limitations of single-pass generation and the importance of explicit validation when crossing modeling formalisms.

Dao et al. Dao, Bucaioni and Cicchetti (2025) benchmark LLMs for Model Transformation by Example (MTBE) across multiple domains, including RDBMS-to-UML, UML-to-Java, and SysML-to-AAS. Their empirical results show that several LLMs already achieve high correctness with a single example pair in structurally regular scenarios, while performance does not consistently improve, and may even degrade, when additional example pairs are introduced. In particular, semantically rich transformations exhibit instability and sensitivity to structural complexity, despite increased supervision. These findings suggest that minimal-example configurations can already induce transformation capability, but robustness remains limited. Motivated by this observation, we adopt a single example pair setting to stress-test whether change-aware round-trip validation can enhance robustness under constrained supervision, rather than relying on scaling the number of examples.

Taken together, these strands show growing momentum toward using LLMs for transformation and co-evolution tasks in software engineering, particularly in example-driven and round-trip settings. At the same time, prior work highlights limitations that matter for dependable and cost-effective deployment: brittleness under semantic complexity, sensitivity to structural heterogeneity and long-context artifacts, dependence on precise prompting and schema grounding, and the need for safeguards such as metamodel conformance, traceability, human oversight, and round-trip verification to ensure correctness at scale.

3. Research Method

The objective of this study is to evaluate whether LLMs can maintain consistency across co-evolving artifacts by measuring how well they preserve an applied change throughout a round-trip transformation. As depicted in Fig. 1, we test whether an LLM preserves an applied delta (Δ) across a round trip between a source artifact S and a target artifact T, without any manually written transformation code. The LLM is expected to infer transformation behavior from a single example pair and to propagate Δ consistently across the coupled artifacts.

To this end, we employ a change-aware round-trip methodology using ten S–T pairs (Fig. 1). We first manually apply Δ on both sides (dotted arrows), producing $\Delta(S)$ and $\Delta(T)$ as ground truth; we then start the LLM-driven round trip from $\Delta(T)$ and evaluate whether the change survives the forward-and-back cycle without drift. We start the round trip from $\Delta(T)$ rather than from $\Delta(S)$ for two reasons. First, in the selected benchmark, the target SQL/XMI artifacts make many structural consequences of a change explicit through tables, columns, primary keys, foreign keys, annotations, and constraints. Starting from $\Delta(T)$, it tests whether the LLM can reconstruct the corresponding source-level abstraction S' from a more implementation-oriented

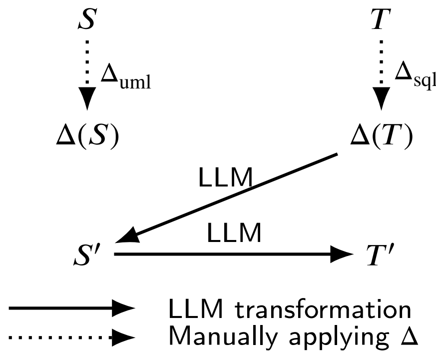

Figure 1: Round-trip transformation process.

representation and then regenerate a consistent target artifact T'. Second, this direction reflects a common co-evolution situation in which an implementation-side or schema-level artifact has changed, and the corresponding design/model artifact must be recovered or synchronized. The manually constructed $\Delta(S)$ is still used as the oracle for evaluating the intermediate S', while $\Delta(T)$ is used as the oracle for evaluating the final T'.

For each pair, we apply ten Δ configurations that span a range of structural and semantic edits. The workflow has four phases:

- prepare transformation prompts that include a single example pair to define correspondences between elements;
- apply Δ on both sides (yielding $\Delta(S)$ and $\Delta(T)$) and query the LLM with $\Delta(T)$ to generate its counterpart S';
- query the LLM again with S' to regenerate the corresponding target artifact T'; and
- validate S' and T' against ground-truth artifacts using a weighted evaluation framework that combines syntactic validity (loadability), structural correspondence, and Δ preservation.

This setup allows us to test whether each applied change is preserved across the round trip and to quantify the extent to which LLMs maintain consistency under controlled co-evolution scenarios.

3.1. Change Configurations

In this work, the source side is a UML class model implemented with EMF Ecore serialization, while the target side is relational SQL schemas serialized in XMI. We instantiate the round-trip evaluation in a bidirectional setting with two coupled artifact types: UML class diagrams on the source side (S) and relational SQL schemas on the target side (T). The experimental pairs are derived from the 10th Transformation Tool Contest (TTC) benchmark corpus[6].

[6] https://github.com/eMoflon/benchmarx/tree/main/examples/ecoretosql/BenchmarxEcoreToSQL/resources

From this corpus, we selected ten representative UML-SQL pairs that are structurally diverse yet conform to a consistent UML-to-SQL mapping.

We selected these pairs to provide controlled diversity in size and structural complexity while preserving a uniform mapping style. Our selection criteria include (i) the number of classes and tables, (ii) the presence of associations and foreign-key relationships, (iii) the use of inheritance and containment, and (iv) the diversity of attribute types and constraints. Concretely, the selected pairs range from small models with fewer than five classes/tables and mostly flat structures to larger models with more than ten classes/tables, multiple associations, inheritance hierarchies, and non-trivial constraint patterns. This stratified selection lets us assess LLM behavior across increasing structural complexity without conflating results with heterogeneous mapping conventions. Because all pairs originate from the same benchmark repository and follow the same correspondence principles, observed differences in performance can be attributed primarily to model size, structural richness, and the applied Δ operators rather than incompatible transformation styles.

For each base pair, we apply one operator Δ selected from a catalog of ten change configurations. Each Δ has a side-specific variant that conforms to the corresponding metamodel:

- On the source side (S), we apply the UML-conformant variant (Δ_{uml}), yielding $\Delta(S)$.
- On the target side (T), we apply the SQL-conformant variant (Δ_{sql}), yielding $\Delta(T)$.

This setup defines an experimental instance (S, T, Δ) with ground-truth artifacts $\Delta(S)$ and $\Delta(T)$.

The ten configurations cover modifications that challenge consistency across both syntax and semantics:

- *Rename Class*: rename a classifier and update all references and mappings; no structural or semantic change.
- *Move Attribute Structural*: move a property from one class to another, updating ownership and relocating the corresponding column and constraints.
- *Delete Attribute*: remove a property or column and all dependent constraints while preserving referential integrity.
- *Add Superclass*: introduce or extend an inheritance hierarchy, lifting shared attributes and reflecting inheritance in the schema mapping.
- *Change Multiplicity*: adjust cardinality and reflect it through nullability, uniqueness, or association tables to enforce new bounds.
- *Convert Containment to Reference*: change an owning association into a non-owning reference by separating the embedded part into its own entity and linking via a foreign key.

(a) Ecore original version with `DataNode`

```
<eClassifiers xsi:type="ecore:EClass" name="Node" abstract="true">
  <eOperations name="addLast" eType="#//Node">
    <eParameters name="newNode" eType="#//DataNode"/>
  </eOperations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="startOf"
      eType="#//List" eOpposite="#//List/start"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="number"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="DataNode" eSuperTypes="#//Node">
  <eOperations name="addLast" eType="#//Node">
    <eParameters name="newNode" eType="#//DataNode"/>
  </eOperations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="follower"
      eType="#//Node"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Leaf" eSuperTypes="#//Node">
  <eOperations name="addLast" eType="#//Node">
    <eParameters name="newNode" eType="#//DataNode"/>
  </eOperations>
</eClassifiers>
```

(b) Ecore renamed version with `ElementNode`

```
<eClassifiers xsi:type="ecore:EClass" name="Node" abstract="true">
  <eOperations name="addLast" eType="#//Node">
    <eParameters name="newNode" eType="#//ElementNode"/>
  </eOperations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="startOf"
      eType="#//List" eOpposite="#//List/start"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="number"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="ElementNode" eSuperTypes="#//Node">
  <eOperations name="addLast" eType="#//Node">
    <eParameters name="newNode" eType="#//ElementNode"/>
  </eOperations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="follower"
      eType="#//Node"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Leaf" eSuperTypes="#//Node">
  <eOperations name="addLast" eType="#//Node">
    <eParameters name="newNode" eType="#//ElementNode"/>
  </eOperations>
</eClassifiers>
```

Figure 2: Example of Rename Class Δ on the Ecore model. Yellow highlights the original identifier; green highlights the renamed identifier after applying Δ.

- *Rename Attribute*: rename a property or column and update corresponding getters, setters, and constraints; no structural change.
- *Split Class*: divide one class into two, redistribute attributes, and create linking associations or tables to preserve semantics.
- *Merge Classes*: combine two classes into one, unify attributes and constraints, and resolve name or type collisions.
- *Introduce Enum Type*: replace free-form values with enumerations, enforced through lookup tables or check constraints.

(a) XMI original version with `DataNode`

```
<ownedColumns
    name="DataNode"
    type="int"
    keys="//@ownedTables.0/@ownedForeignKeys.2">
  <properties>Unique</properties>
</ownedColumns>
...
<ownedTables
    name="DataNode"
    referencingForeignKeys="//@ownedTables.0/@ownedForeignKeys.2">
  <ownedAnnotations annotation="class"/>
  <ownedAnnotations annotation="concrete"/>
  <ownedColumns
      name="follower"
      type="int"
      keys="//@ownedTables.1/@ownedForeignKeys.0">
    <ownedAnnotations annotation="cross"/>
    <ownedAnnotations annotation="unidirectional"/>
    <ownedAnnotations annotation="single"/>
  </ownedColumns>
  <ownedColumns
      name="id"
      type="int"
      keys="//@ownedTables.1/@ownedPrimaryKey //@ownedTables.1/
@ownedForeignKeys.1">
    <properties>NotNull</properties>
  </ownedColumns>
</ownedTables>
```

(b) XMI renamed version with `ElementNode`

```
<ownedColumns
    name="ElementNode"
    type="int"
    keys="//@ownedTables.0/@ownedForeignKeys.2">
  <properties>Unique</properties>
</ownedColumns>
...
<ownedTables
    name="ElementNode"
    referencingForeignKeys="//@ownedTables.0/@ownedForeignKeys.2">
  <ownedAnnotations annotation="class"/>
  <ownedAnnotations annotation="concrete"/>
  <ownedColumns
      name="follower"
      type="int"
      keys="//@ownedTables.1/@ownedForeignKeys.0">
    <ownedAnnotations annotation="cross"/>
    <ownedAnnotations annotation="unidirectional"/>
    <ownedAnnotations annotation="single"/>
  </ownedColumns>
  <ownedColumns
      name="id"
      type="int"
      keys="//@ownedTables.1/@ownedPrimaryKey //@ownedTables.1/
@ownedForeignKeys.1">
    <properties>NotNull</properties>
  </ownedColumns>
</ownedTables>
```

Figure 3: Example of Rename Class Δ on the XMI model. Yellow highlights the original table/column name; green highlights the renamed table/column name after applying Δ.

Fig 2 and 3 show an example of how a Δ operation (in this case, Rename Class) is applied onto the oracle S and T model to become Δ(S) and Δ(T).

The ten Δ operators are manually defined but not arbitrary: they align with well-established atomic evolution patterns repeatedly reported in studies of model, schema, and code evolution, and they are commonly used to study

co-evolution and synchronization An et al. (2008); Wimmer et al. (2012); Kretschmer et al. (2021). Such atomic changes (e.g., renames, splits, merges, type normalization, and multiplicity updates) serve as building blocks from which more complex real-world evolutions are composed. Applying these Δ operators in isolation helps us control confounding factors and attribute successes and failures to specific evolution classes. This controlled design follows common practice in model-driven engineering, where researchers first study automated techniques under clearly defined evolution scenarios before considering entangled change histories. We apply each Δ configuration to all base pairs (dashed arrows in Fig. 1). These configurations stress both syntax-level fidelity (e.g., naming and containment) and schema-level semantics (e.g., inheritance and enumerations). For each applied Δ, we execute an LLM-based round trip in two stages:

- Target-to-Source: the LLM receives one fixed example pair (CompositeListDataEcore.ecore and CompositeListDataSQL.xmi) and concise mapping guidance, then transforms $\Delta(T)$ into an inferred source model S'.
- Source-to-Target: using the same example pair and guidance, the LLM transforms S' back into a target schema T'.

We repeat this procedure for four frontier LLMs available through hosted APIs at the time of the experiment: GPT-5, Qwen-3-Next-80B-A3B, DeepSeek V3, and Gemini 2.5. These models were selected because they represent widely used, high-performing general-purpose LLM families with strong structured-generation capabilities, broad context windows, and practical availability through stable API interfaces. This selection allows us to evaluate whether current state-of-the-art LLM services can preserve changes across round-trip co-evolution tasks without manually written transformation code. We did not include locally hosted open-weight models in this initial study because the goal was to benchmark frontier API-based systems under comparable prompting and output constraints, rather than to study deployment trade-offs such as hardware availability, quantization, inference configuration, or local serving infrastructure. In total, the experiment includes ten base pairs, ten Δ configurations, four LLMs, and two transformation steps, resulting in 800 generated artifacts ($10 \times 10 \times 4 \times 2$).

3.2. Model Generation with LLMs

Our prompting strategy follows an example-centric, conformance-first design tailored to round-trip evaluation Dao et al. (2025). We do not aim to elicit creative or approximate transformations; instead, we test whether LLMs can infer and apply consistent transformation behavior from a minimal, fixed example while respecting metamodel and serialization constraints. We formalize the prompt structure as:

$$\text{Prompt} = \underbrace{\text{Example Pair}}_{E} + \underbrace{\text{Input Model}}_{IM} + \underbrace{\text{Objective}}_{O} + \underbrace{\text{Constraints}}_{C} \tag{1}$$

where:

- *Example Pair* E: is a fixed, complete source–target example pair that illustrates the correspondence between UML class diagrams (Ecore/XMI) and relational SQL schemas (XMI); this pair is shared across all runs and is the only grounding for inferring mappings (e.g., classes–tables, attributes–columns, associations–foreign keys).
- *Input Model* IM: is the artifact to be transformed (either $\Delta(T)$ or S'), serialized in its native format, representing a concrete evolution step produced by one Δ configuration.
- *Transformation Objective* O: is a concise instruction that specifies the transformation direction (SQL-to-UML or UML-to-SQL) and requires preserving the applied change.
- *Hard Constraints* C: are the explicit constraints that enforce (i) output-only serialization (no explanatory text), (ii) strict metamodel conformance, and (iii) validity-first behavior when mappings are incomplete.

All prompts enforce the following principles:

- *Example-centric inference*: we instruct LLMs to rely exclusively on the provided example pair to infer correspondences; we do not encourage external knowledge, schema guessing, or stylistic generalization.
- *Conformance-first generation*: outputs must be syntactically loadable and metamodel-conformant; when the example pair does not provide sufficient evidence to determine the mapping, we instruct the LLM to prefer a valid but incomplete artifact over speculative structures.
- *Output-only serialization*: the prompt forbids prose explanations; the LLM must return only the target XMI serialization.
- *Delta awareness*: the LLM must preserve the applied Δ while keeping unrelated elements unchanged, so we can isolate propagation failures.
- *Handling unmapped elements*: if the input contains constructs not covered by the example mapping, we instruct the LLM to emit a valid artifact and flag uncertainty using XML comments rather than invent unsupported structures. If an output is not parsable or violates metamodel constraints, we resend the same prompt once to recover from serialization errors; we log this single retry explicitly and retain all raw prompt–response pairs.

For each run, we perform single-shot generation and record the raw serialization returned as S' (target-to-source) or T' (source-to-target). The prompt structure is shown in Listing 1, while Listings 2 and 3 illustrate how we instantiate the template for each direction.

```
You will be given two example models:

(1) a source model S (CompositeListDataEcore.ecore)
(2) its mapped target model T (CompositeListDataSQL.xmi)

Then you will receive a new input model, denoted Δ(T) or S'. Using only the
structure and correspondences you infer from the mapping between S and T,
transform the given input model into its equivalent counterpart.

Objective:
- If the input is Δ(T), generate the corresponding UML model S'.
- If the input is S', generate the corresponding SQL schema T'.

Constraints:
- Output only the target XMI serialization.
- Do not include any explanatory text.
- Ensure metamodel conformance and loadability.
- Preserve the applied change while keeping unrelated elements unchanged.
- If an element has no clear mapping in the example, emit a valid artifact and
flag the uncertainty using XML comments.
```

Listing 1: Prompt template used for all runs. The underlined text represents the variable portions of the template.

```
[Example Pair]
S: CompositeListDataEcore.ecore
T: CompositeListDataSQL.xmi

[Input Model]
Δ(T): SQL schema after applying Split Class to table Order

[Objective]
Generate the corresponding UML model S'.

[Constraints]
- Return only UML XMI.
- Do not include any explanation text.
- Ensure metamodel conformance and loadability.
```

Listing 2: Instantiated prompt for Δ_8 (Split Class), SQL→UML

```
[Example Pair]
S: CompositeListDataEcore.ecore
T: CompositeListDataSQL.xmi

[Input Model]
S': UML model where attribute \texttt{status} is converted to an enumeration

[Objective]
Generate the corresponding SQL schema T'.

[Constraints]
- Return only SQL XMI.
- Do not include any explanation text.
- Enforce metamodel conformance and loadability.
```

Listing 3: Instantiated prompt for Δ_{10} (Introduce Enum Type), UML→SQL

3.3. Evaluation Methodology

Our evaluation is a layered process that separates syntactic validity, structural correspondence, and semantic change preservation. We separate these dimensions because textual similarity alone does not capture semantic correctness, while fully automated semantic equivalence checking across heterogeneous UML and SQL artifacts is infeasible Rosca and Domingues (2021); Qiu et al. (2013). We therefore combine automated checks with targeted assessments to obtain a reproducible yet meaningful evaluation.

3.3.1. *Execution and Reproducibility*

Each LLM produces one output per prompt. If an output is not a valid serialization (e.g., not parsable), we resend the same prompt once to recover from serialization errors. This single retry is logged, and we archive all prompts, retries, and generated artifacts in the replication package. We do not adapt prompts or perform interactive correction.

3.3.2. *Syntactic Validity Checks (Automated)*

We first assess whether each generated artifact is syntactically valid and loadable with respect to its target metamodel. For both UML (S') and SQL (T') outputs, we verify that the artifact can be parsed and conforms to the corresponding metamodel using the Papyrus environment[7].

- *Yes*: the model parses successfully and conforms to the metamodel; we treat remaining differences as semantic mismatches relative to the intended Δ.
- *No*: the model fails to load due to parsing or conformance errors, indicating unresolved syntactic or structural issues.

Artifacts that fail this check are classified as syntactically invalid and are not considered correct regardless of subsequent comparisons.

3.3.3. *Structural Element-Level Comparison (Quantitative)*

For syntactically valid artifacts, we manually compare the generated models S' (UML) and T' (SQL) against their respective ground-truth counterparts $\Delta(S)$ and $\Delta(T)$ (Fig. 1). This step is not fully automated. To make the comparison systematic and easier to audit, we first normalize all artifacts by enforcing stable element ordering and normalized whitespace in XML/XMI. We then inspect the normalized serializations line by line using a visual diff tool, Beyond Compare, as support for identifying matches, missing elements, additions, and mismatches.

After normalization, each serialized line corresponds to a small number of concrete elements, such as XML/XMI tags or attributes. This makes line-level inspection a practical and repeatable proxy for structural preservation across many runs. The diff tool is used only to support visual comparison; the final classification of each line is performed manually according to the evaluation criteria below.

We classify each line as follows:

- *Lines of Code (LOC)*: total number of lines in the generated model, used as the normalization base.
- *Correct Lines (Cr)*: lines that exactly match the ground truth (+1 each).

[7] https://eclipse.dev/papyrus/index.html

- *Incorrect Lines (InCr)*: lines that are syntactically or structurally incorrect relative to the ground truth (-1 each).
- *Missing Elements (Miss)*: lines expected from the ground truth but absent in the generated model (-1 each).
- *Additional Lines (Ad)*: lines not present in the ground truth; we further classify additions as neutral (score 0) or harmful (score -1).

This comparison provides a scalable, quantitative proxy for structural preservation. We do not claim that line-level matching fully captures semantic equivalence; instead, we use it as a repeatable basis for comparing LLM behavior across many runs. To mitigate superficial mismatches (e.g., formatting variations, ordering differences, or benign serialization artifacts), we normalize outputs prior to comparison and interpret discrepancies in light of structural intent. Residual limitations of textual comparison are discussed further in Section 5

3.3.4. Semantic Δ Preservation

We assess semantic correctness via Δ preservation. For each run, we check whether the intended evolution step (e.g., rename, split, merge, enum introduction) appears in the generated artifact and is applied consistently. We distinguish three outcomes:

- *Correct Δ detection*: the intended change is present and consistently applied.
- *Incorrect Δ detection*: the change is partially applied or inconsistently propagated.
- *Undetected* Δ: the intended change is absent.

We treat undetected Δ cases as hard failures and apply a fixed penalty because failing to apply the intended evolution invalidates the transformation regardless of other similarities.

3.3.5. Aggregate Metrics

Based on the above classifications, we compute two aggregate metrics.

Percentage of Correctness (%Cr) captures the proportion of correct lines relative to expected content:

$$\%Cr = \frac{Cr}{LOC + Miss - Ad} \tag{2}$$

Weighted Success Rate (%WS) extends this measure by penalizing incorrect and missing elements, harmful additions, and undetected Δ instances:

$$\%WS = \frac{Cr - InCr - Miss - Ad[\text{harmful}] - 2[\Delta\text{undetected}]}{LOC} \tag{3}$$

The weighting scheme encodes a conservative preference for semantic soundness over verbosity. Because these weights reflect design choices rather than universal semantics, we report raw category counts alongside aggregate scores.

3.3.6. Round-Trip Considerations

Because T' is produced at the end of the round trip, its quality depends on the intermediate output S'. Errors introduced in S' can therefore propagate to T'. For T' evaluations, if the Δ is not detected at the end of the round trip, we do not compute %Cr and %WS for that run because the end-to-end objective is not met. We nevertheless compute summary statistics (mean and population standard deviation) using a fixed denominator of $N{=}10$, treating non-computed trials as zero contributions; this choice keeps comparisons fair across LLMs and Δ operators and applies a conservative penalty to end-to-end failures.

Tables 1 and 2 report the mean (μ) and population standard deviation (σ) computed over the ten base source-target pairs ($N{=}10$) for each Δ configuration and each LLM. The complete per-run values are provided in the supplementary data repository in section 7.

For a fixed Δ and LLM, let Cr_i and WS_i denote the line-level correctness and weighted success rate obtained on base pair i, where $i \in \{1, \dots, 10\}$. The mean correctness is computed as:

$$\mu_{\mathrm{Cr}} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{Cr}_i,$$

and the population standard deviation as:

$$\sigma_{\mathrm{Cr}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \left(\mathrm{Cr}_i - \mu_{\mathrm{Cr}}\right)^2}.$$

The same computation is applied independently to weighted success:

$$\mu_{\mathrm{WS}} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{WS}_i, \qquad \sigma_{\mathrm{WS}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \left(\mathrm{WS}_i - \mu_{\mathrm{WS}}\right)^2}.$$

Importantly, following the above evaluation protocol, runs in which the intended Δ is not detected at the end of the round trip contribute a value of 0 to both Cr_i and WS_i, while still being included in the fixed denominator $N{=}10$. This ensures that the reported μ and σ reflect end-to-end robustness rather than performance conditioned on successful detections only.

As a running example, we show how μ and σ are computed for Δ_1 (Rename Class) and GPT-5 in the $\Delta(T) \rightarrow S'$ stage (evaluated against $\Delta(S)$). From the full dataset (Table.docx), the ten per-run values are:

$$\{\mathrm{Cr}_i\}_{i=1}^{10} = \{100.00, 60.61, 48.72, 100.00, 0.00, 93.10, 55.56, 100.00, 68.00, 55.56\},$$

$$\{\mathrm{WS}_i\}_{i=1}^{10} = \{100.00, 2.63, 0.00, 100.00, 0.00, 82.76, 20.00, 61.90, 40.00, 0.00\}.$$

With $N=10$, the mean correctness is:

$$\begin{aligned}
\mu_{\mathrm{Cr}} &= \frac{1}{10}\sum_{i=1}^{10} \mathrm{Cr}_i \\
&= \frac{100.00 + 60.61 + 48.72 + 100.00 + 0.00}{10} \\
&\quad + \frac{93.10 + 55.56 + 100.00 + 68.00 + 55.56}{10} \\
&= \frac{681.55}{10} = 68.155.
\end{aligned}$$

The population standard deviation of correctness is:

$$\begin{aligned}
\sigma_{\mathrm{Cr}} &= \sqrt{\frac{1}{10}\sum_{i=1}^{10} \left(\mathrm{Cr}_i - \mu_{\mathrm{Cr}}\right)^2} \\
&= \sqrt{\frac{1}{10}\left[(100.00 - 68.155)^2 + \cdots + (55.56 - 68.155)^2\right]} \\
&\approx 30.1025.
\end{aligned}$$

Analogously, the mean weighted success is:

$$\begin{aligned}
\mu_{\mathrm{WS}} &= \frac{1}{10}\sum_{i=1}^{10} \mathrm{WS}_i \\
&= \frac{100.00 + 2.63 + 0.00 + 100.00 + 0.00}{10} \\
&\quad + \frac{82.76 + 20.00 + 61.90 + 40.00 + 0.00}{10} \\
&= \frac{407.29}{10} = 40.729.
\end{aligned}$$

and the population standard deviation is:

$$\begin{aligned}
\sigma_{\mathrm{WS}} &= \sqrt{\frac{1}{10}\sum_{i=1}^{10} \left(\mathrm{WS}_i - \mu_{\mathrm{WS}}\right)^2} \\
&\approx 40.1238.
\end{aligned}$$

Overall, this layered evaluation combines automated validity checks, quantitative structural comparison, and targeted Δ assessment to characterize LLM performance in preserving round-trip consistency under controlled co-evolution.

4. Result Evaluation

This section presents the empirical results from evaluating GPT-5, DeepSeek V3, Qwen3-Next-80B-A3B, and Gemini 2.5 across ten base source–target pairs and ten delta configurations. For every combination of base pair and delta, each LLM performs a full round-trip transformation, generating S' from $\Delta(T)$ and then generating T' from S', as described in Section 3. We validate the resulting artifacts against the corresponding ground-truth models $\Delta(S)$ and $\Delta(T)$ using the weighted evaluation framework introduced earlier. Tables 1 and 2 summarize outcomes for all LLMs and Δ configurations: Table 1 reports the first stage of the round trip ($\Delta(T) \rightarrow S'$, evaluated against $\Delta(S)$), while Table 2 reports the end-to-end outcome at the end of the round trip ($S' \rightarrow T'$, evaluated against $\Delta(T)$). For each LLM and Δ operator, the tables report:

1. the number of outputs loadable in Papyrus (i.e., syntactically valid and metamodel-conformant);
2. the frequency with which the intended delta change is correctly detected and propagated; and
3. the mean (μ) and population standard deviation (σ) of both correctness and weighted success rate, computed by the line-level weighted evaluation framework.

4.1. $\Delta(T) \rightarrow S'$ (vs. $\Delta(S)$)

From $\Delta(T)$ to S', Gemini 2.5 achieves the strongest overall performance, producing 95/100 loadable outputs and correctly detecting the intended delta in 88 cases. DeepSeek V3 and GPT-5 perform comparably in loadability (88/100), with slightly lower delta detection rates of 75 and 74, respectively. Qwen3-Next-80B-A3B trails behind with 71 loadable outputs and 55 successful delta detections, indicating higher variability in consistency preservation. Across all LLMs, simpler syntactic modifications such as Δ_1 (Rename Class) are consistently recognized and propagated. In contrast, more complex structural operations, notably Δ_8 (Split Class), are often detected but yield lower line-level fidelity, which indicates that models frequently identify the intended structural shift yet struggle to realize it precisely while preserving metamodel conformance.

Key observations

For the first round-trip stage, $\Delta(T) \rightarrow S'$, the main observations are:

- Local changes (renaming, deleting, moving an attribute) are easier because they affect only a small part of the model.
- Detecting the correct Δ is not enough. In several cases, the LLM identifies the intended change but only applies it partially in S'.
- Structural changes (splitting, merging classes) are harder because several related model elements must be updated together.
- The gap between correctness and weighted success shows that many outputs contain some correct elements but miss important parts of the expected model.
- LLMs can support reverse propagation, but they still need help with preserving dependencies between model elements.

4.1.1. GPT-5

In the $\Delta(T) \rightarrow S'$ direction, GPT-5 performs best on nominal and localized edits but declines on restructuring

Table 1
Performance Evaluation of LLMs in Round-trip Model Transformation between S' and $\Delta(S)$

Δ	**LLM**	**Loadability**	**Δ detected**	μ_{Cr}	σ_{Cr}	μ_{WS}	σ_{WS}
1 Rename Class	GPT-5	8	**10**	68.155	30.102	40.729	40.1238
	DeepSeek V3	7	**10**	**89.304**	**16.346**	**71.923**	**27.7419**
	Qwen3-Next-80B-A3B	**10**	6	72.963	20.290	42.82	35.5067
	Gemini 2.5	9	**10**	87.819	24.106	70.535	36.5791
2 Move Attribute	GPT-5	**10**	7	59.925	27.885	29.722	35.2372
	DeepSeek V3	**10**	9	**88.741**	**10.316**	62.585	**25.7895**
	Qwen3-Next-80B-A3B	9	6	64.523	32.539	39.716	36.7011
	Gemini 2.5	9	**10**	88.275	11.525	**70.38**	27.3321
3 Delete Attribute	GPT-5	**9**	**10**	56.968	23.419	20.206	30.0161
	DeepSeek V3	**9**	9	90.374	7.0232	63.754	**24.6588**
	Qwen3-Next-80B-A3B	7	9	68.638	20.0067	29.773	37.9727
	Gemini 2.5	**9**	**10**	**90.738**	**6.3815**	**74.558**	25.2623
4 Add Superclass	GPT-5	8	7	53.18	23.367	20.614	**28.6173**
	DeepSeek V3	9	8	80.32	14.162	52.219	30.7466
	Qwen3-Next-80B-A3B	7	5	74.66	15.772	36.949	29.8711
	Gemini 2.5	**10**	**9**	**83.74**	**12.739**	**60.166**	31.3722
5 Change Multiplicity	GPT-5	**9**	5	65.255	25.947	33.609	37.0212
	DeepSeek V3	8	4	85.693	9.5003	56.628	27.4797
	Qwen3-Next-80B-A3B	5	2	66.163	23.0181	26.235	**25.2205**
	Gemini 2.5	**9**	**8**	**87.074**	**7.5111**	**63.93**	26.1267
6 Containment to Ref	GPT-5	7	5	65.411	25.8106	27.559	35.1099
	DeepSeek V3	**10**	7	84.906	7.8061	58.203	24.1567
	Qwen3-Next-80B-A3B	6	7	66.319	27.7819	33.137	34.0538
	Gemini 2.5	9	**9**	**89.017**	**6.9987**	**69.194**	**24.1984**
7 Rename Attribute	GPT-5	8	5	55.338	32.5694	24.951	39.5274
	DeepSeek V3	**10**	**7**	**87.128**	**9.1307**	58.602	**26.1323**
	Qwen3-Next-80B-A3B	8	4	67.701	21.9873	26.347	32.9842
	Gemini 2.5	**10**	**7**	83.826	23.5163	**60.107**	34.0109
8 Split Class	GPT-5	**10**	**10**	50.07	16.2703	11.334	25.0717
	DeepSeek V3	9	**10**	66.227	11.4286	**25.49**	23.0351
	Qwen3-Next-80B-A3B	5	6	57.189	19.142	18.313	21.9935
	Gemini 2.5	**10**	9	**67.508**	**8.3479**	25.149	**18.3737**
9 Merge Classes	GPT-5	**10**	**9**	50.628	**23.8391**	18.254	**27.6201**
	DeepSeek V3	9	7	72.037	28.8962	45.463	38.2288
	Qwen3-Next-80B-A3B	8	5	61.677	29.2096	26.737	29.3943
	Gemini 2.5	**10**	**9**	**74.365**	27.681	**53.802**	38.817
10 Introduce Enum	GPT-5	9	6	55.26	25.1443	18.47	22.1305
	DeepSeek V3	7	4	73.514	11.8964	35.632	19.9948
	Qwen3-Next-80B-A3B	6	5	57.886	16.1992	11.586	**14.6509**
	Gemini 2.5	**10**	**7**	**74.085**	**10.2897**	**39.575**	19.2828

and typing-related changes. GPT-5 performs strongest on Δ_1 (Rename Class), where it attains the highest weighted success rate among all configurations. However, the high variance indicates substantial sensitivity to individual instances. It maintains moderate performance on dispersed, localized edits such as Δ_5 (Change Multiplicity) and Δ_6 (Convert Containment to Reference). GPT-5 struggles with

structure-altering or typing-intensive deltas such as Δ_8 (Split Class), where weighted success is relatively low and consistently weak. Similar challenges arise for Δ_9 (Merge Classes) and Δ_{10} (Introduce Enum Type), where correctness remains moderate, but low-weighted success indicates incomplete propagation of the applied changes. Overall, GPT-5 recognizes most Δ operators but often fails to propagate them coherently, which leads to missing or incorrect fragments that reduce weighted success. The high variability across configurations also suggests unstable behavior when transformations require global structural consistency.

4.1.2. *DeepSeek V3*

In this direction, DeepSeek V3 shows high correctness and strong stability, particularly for nominal and locally structural configurations. It performs consistently well on Δ_1 (Rename Class) and Δ_2 (Move Attribute). Peak correctness appears on Δ_3 (Delete Attribute) with low variance, indicating reliable handling of simple, localized deletions. DeepSeek remains competitive on moderately structural transformations such as Δ_4 (Add Superclass), Δ_5 (Change Multiplicity), and Δ_6 (Convert Containment to Reference). Performance declines on more disruptive transformations, notably Δ_8 (Split Class). The Δ_9 (Merge Classes) configuration yields moderate accuracy but high variance, suggesting inconsistent behavior across instances. Typing-related changes such as Δ_{10} (Introduce Enum Type) yield the weakest results in this group. Overall, DeepSeek V3 achieves stable correctness with moderate variance in weighted success. When performance degrades, it typically does so through missing rather than spurious elements, indicating that the model tends to under-generate instead of introducing harmful additions.

4.1.3. *Qwen3-Next-80B-A3B*

In the $\Delta(T) \rightarrow S'$ direction, Qwen3-Next-80B-A3B shows uneven performance: it achieves reasonable correctness on simple edits but exhibits notable instability on structural and typing changes. It performs best on Δ_1 (Rename Class) and Δ_2 (Move Attribute). While Δ_4 (Add Superclass) yields the highest mean correctness, its weighted success is only moderate, suggesting that Qwen often recognizes hierarchical modifications but struggles to propagate them consistently across dependent elements. Performance drops sharply for more complex transformations such as Δ_8 (Split Class) and Δ_{10} (Introduce Enum Type), while Δ_9 (Merge Classes) shows intermediate outcomes. Variance is moderate to high across several configurations, reflecting substantial run-to-run variability. Together with lower loadability and delta detection on demanding configurations, these results indicate fragility in Qwen's handling of global rewrites. While Qwen can achieve acceptable correctness on localized edits, weighted success degrades as the number of dependent updates increases, implying difficulties in maintaining consistent structural alignment across co-evolving artifacts.

4.1.4. *Gemini 2.5*

From $\Delta(T)$ to S', Gemini 2.5 performs consistently well across nearly all configurations, combining high correctness with high weighted success. Gemini's best results appear on Δ_3 (Delete Attribute) with low variance, indicating stable handling of localized deletions. It also performs strongly on nominal and locally structural transformations, including Δ_1 (Rename Class), Δ_2 (Move Attribute), Δ_5 (Change Multiplicity), and Δ_6 (Convert Containment to Reference). As with other LLMs, performance declines on the most complex structural transformations. Δ_8 (Split Class) yields the weakest outcomes, and Δ_9 (Merge Classes) shows the highest variance despite moderate means. Typing-related changes such as Δ_{10} (Introduce Enum Type) remain challenging, although Gemini performs noticeably better than Qwen and GPT-5. Overall, Gemini 2.5 achieves high correctness and balanced weighted success with modest variance across most configurations, suggesting a stronger capability to propagate edits coherently while preserving metamodel conformance.

4.2. $S' \rightarrow T'$ (vs. $\Delta(T)$)

At the end of the round trip, T' is downstream of S', so failures in producing a valid and Δ-preserving S' can propagate. Following our evaluation procedure, any run that does not detect Δ at the end of the round trip contributes 0 to the aggregates for %Cr and %WS (fixed denominator N=10). Gemini 2.5 again shows the most dependable performance, producing 77/100 loadable outputs and 82 successful delta detections. Qwen3-Next-80B-A3B follows with 70/100 loadable outputs but only 46/100 detections, indicating difficulty maintaining alignment during regeneration. DeepSeek V3 records the lowest loadability (55/100) yet achieves relatively strong detection (70/100), suggesting that it often identifies the intended delta even when structural validity fails. GPT-5 attains 65/100 loadable outputs and 68 detections, placing it in the mid-range on both measures.

Across models, Δ_1 (Rename Class) is handled most reliably: Gemini and DeepSeek achieve high correctness and weighted success, GPT-5 is moderate, and Qwen lags. Complex restructurings such as Δ_8 (Split Class) and Δ_9 (Merge Classes) are often detected but yield lower averages and higher variance, while type-level changes such as Δ_{10} (Introduce Enum Type) remain difficult for non-Gemini models.

Key observations

For the end-to-end round trip, $S' \rightarrow T'$, the main observations are:

- Performance is generally lower at the end of the round trip than after the first stage, showing that errors accumulate across transformations.
- A model can be loadable but still fail to preserve the intended Δ correctly.
- Local changes remain more stable because they require fewer dependent updates.
- Complex changes (split, merge, multiplicity, containment, enum changes) are more fragile because missing one related update can affect the final result.
- Round-trip evaluation should consider not only metamodel conformance, but also whether the same change is preserved across both directions.

4.2.1. GPT-5

GPT-5 shows uneven end-to-end performance: it maintains modest correctness on localized changes but weakens on type- and multiplicity-related transformations, while peaking on a structural refactoring. GPT-5 achieves its best weighted success on Δ_8 (Split Class), followed by Δ_1 (Rename Class) and Δ_3 (Delete Attribute). This peak reflects that, when GPT-5 detects and completes the broader restructuring, the resulting structure can match the ground truth well, but the same configuration also has many end-to-end failures that increase variance and lower aggregate robustness. The weakest outcomes occur on Δ_{10} (Introduce Enum Type) and Δ_7 (Rename Attribute). Variance is lower in some underperforming cases, such as Δ_5 (Change Multiplicity), indicating consistent underperformance rather than sporadic failures. Overall, GPT-5 detects many deltas (e.g., 10/10 detection for Δ_1 and Δ_3) but often fails to propagate them coherently across artifacts. Correctness tends to exceed weighted success, highlighting difficulties in preserving structural alignment and completeness through the round trip.

4.2.2. DeepSeek V3

DeepSeek V3 achieves high correctness and strong weighted success whenever it generates loadable artifacts and detects the intended delta, although its overall loadability is the lowest among models. Its best performance occurs on Δ_2 (Move Attribute) and Δ_8 (Split Class), with substantial variance. It also performs strongly on Δ_3 (Delete Attribute) and Δ_6 (Convert Containment to Reference). Performance declines on Δ_4 (Add Superclass) and drops more sharply on Δ_5 (Change Multiplicity) and Δ_{10} (Introduce Enum Type). On Δ_1 (Rename Class), DeepSeek achieves excellent detection (10/10) and high correctness, trailing only Gemini and outperforming GPT-5 and Qwen. Despite low aggregate loadability, DeepSeek produces accurate and competitive outputs when artifacts are syntactically valid and the delta is detected, including on structurally involved changes such as Δ_8. Variability concentrates in broader reorganizations and type-level edits, suggesting strength in local consistency but challenges with global structural dependencies.

4.2.3. Qwen3-Next-80B-A3B

Qwen3 performs best on localized attribute edits rather than rename transformations, with stronger correctness and weighted success when the scope of change is limited. The highest mean values occur on Δ_3 (Delete Attribute), with comparable performance on Δ_2 (Move Attribute). Performance declines on Δ_4 (Add Superclass), Δ_8 (Split Class), and Δ_{10} (Introduce Enum Type), and variance increases on broader or type-related transformations such as Δ_5. The Δ_8 configuration combines low means with low weighted success, reflecting consistently weak outcomes. Notably, delta detection is often weak even when correctness and weighted success are otherwise decent (e.g., Δ_7 with 2/10 detected and Δ_1 with 5/10), yielding a bimodal pattern: when Qwen recognizes the intended change, both metrics increase sharply; when it does not, performance collapses despite otherwise solid loadability. Overall, Qwen3-Next-80B-A3B achieves reasonable loadability across configurations but is constrained by inconsistent change recognition, which limits round-trip consistency in more complex scenarios.

4.2.4. Gemini 2.5

Gemini 2.5 achieves the strongest overall end-to-end performance, combining high correctness with consistently modest variance across several transformations. Gemini performs best on Δ_1 (Rename Class), and it maintains strong scores on Δ_5 (Change Multiplicity), Δ_3 (Delete Attribute), Δ_2 (Move Attribute), and Δ_9 (Merge Classes). It also performs strongly on Δ_4 (Add Superclass). The weakest results occur on Δ_8 (Split Class), with a secondary dip on Δ_{10} (Introduce Enum Type), reflecting the difficulty of coherent multi-entity and type-level restructuring. With the highest detection (82/100) and loadability (77/100) in this setting, Gemini 2.5 demonstrates superior round-trip consistency. Its weighted success closely tracks correctness across scenarios, indicating that Gemini not only detects and applies changes but also preserves structural alignment and metamodel conformance throughout the round-trip process.

5. Discussion

This section discusses the implications of our empirical results for using LLMs in round-trip software engineering tasks. We first interpret performance trends across Δ operators and models, then analyze recurrent failure patterns and their likely causes, and finally outline concrete directions for improving robustness in LLM-assisted co-evolution workflows.

Table 2
Performance Evaluation of LLMs in Round-trip Model Transformation between T' and $\Delta(T)$

Δ	LLM	Loadability	Δ detected	μ_{Cr}	σ_{Cr}	μ_{WS}	σ_{WS}
1 Rename Class	GPT-5	6	**10**	67.182	38.384	28.136	43.039
	DeepSeek V3	5	**10**	93.674	8.593	61.461	41.209
	Qwen3-Next-80B-A3B	9	5	45.859	47.062	24.925	37.584
	Gemini 2.5	**10**	**10**	**99.957**	**0.129**	**98.929**	**3.0723**
2 Move Attribute	GPT-5	4	6	40.483	42.893	24.416	**36.979**
	DeepSeek V3	5	**8**	**76.275**	**38.476**	63.641	38.019
	Qwen3-Next-80B-A3B	**8**	5	49.376	49.3805	38.227	42.545
	Gemini 2.5	7	**8**	69.97	43.95	**68.46**	44.90
3 Delete Attribute	GPT-5	**8**	**10**	44.447	35.388	27.716	36.447
	DeepSeek V3	7	9	79.491	35.978	53.71	40.651
	Qwen3-Next-80B-A3B	7	8	63.43	40.93	45.56	42.53
	Gemini 2.5	**8**	**10**	**90.416**	**22.937**	**83.084**	**28.408**
4 Add Superclass	GPT-5	4	6	26.35	**30.73**	9.12	27.35
	DeepSeek V3	7	7	60.86	41.62	46.90	39.95
	Qwen3-Next-80B-A3B	6	4	24.63	32.40	12.09	**21.22**
	Gemini 2.5	**8**	**8**	**75.08**	39.49	**69.88**	42.63
5 Change Multiplicity	GPT-5	4	5	22.35	**29.98**	7.49	**19.67**
	DeepSeek V3	6	3	19.813	39.628	19.809	39.621
	Qwen3-Next-80B-A3B	4	2	18.40	36.80	9.26	27.77
	Gemini 2.5	**9**	**7**	**69.885**	45.751	**69.661**	45.609
6 Containment to Ref	GPT-5	**7**	5	23.79	**33.19**	12.23	**27.99**
	DeepSeek V3	5	7	64.87	43.72	54.84	44.61
	Qwen3-Next-80B-A3B	6	7	48.293	42.207	23.688	38.660
	Gemini 2.5	**7**	**8**	**75.317**	39.462	**68.211**	44.767
7 Rename Attribute	GPT-5	7	5	19.042	**31.241**	3.852	**7.955**
	DeepSeek V3	4	6	59.001	48.195	46.907	45.082
	Qwen3-Next-80B-A3B	**9**	2	19.320	38.671	17.049	34.310
	Gemini 2.5	7	**7**	**69.258**	45.348	**68.871**	45.120
8 Split Class	GPT-5	**10**	8	51.786	32.255	28.266	36.236
	DeepSeek V3	7	**10**	**87.991**	**16.672**	59.477	32.942
	Qwen3-Next-80B-A3B	6	6	43.980	36.879	12.636	**25.493**
	Gemini 2.5	6	9	70.116	37.949	**61.629**	40.285
9 Merge Classes	GPT-5	**8**	**9**	36.385	**28.060**	13.495	**21.555**
	DeepSeek V3	5	6	55.292	45.293	47.541	39.767
	Qwen3-Next-80B-A3B	**8**	3	26.462	40.633	22.434	35.018
	Gemini 2.5	7	8	**75.194**	39.104	**70.827**	39.451
10 Introduce Enum	GPT-5	7	4	17.861	**30.906**	2.577	**7.731**
	DeepSeek V3	4	4	34.014	42.142	23.260	32.057
	Qwen3-Next-80B-A3B	7	4	24.25	37.25	10.79	21.00
	Gemini 2.5	**8**	**7**	**65.15**	43.23	**59.75**	41.81

5.1. Interpretation of Empirical Results

In this work, we benchmark four LLMs on a change-aware round-trip process across ten Δ configurations and ten base S-T pairs. Beyond reporting aggregate scores, we focus on where models fail or nearly succeed and what these outcomes imply for practical co-evolution. Overall, the results show that LLMs handle localized, low-impact changes reliably, such as renames and small, scope-limited

adjustments. For these Δ operators, most models achieve high success in a single iteration, which indicates that they detect the intended modification and apply it consistently when propagation touches a few dependent elements. In contrast, structural and semantic changes, including class splits, merges, inheritance introduction, and enum normalization, produce a sharp drop in success rates across all evaluated models. These edits require coordinated updates across multiple dependent elements (e.g., associations, foreign keys, constraints, and multiplicities). While LLMs often recognize that a structural change is required, they frequently fail to propagate it completely, which yields artifacts that remain loadable but drift semantically from the intended evolution. Across models, we observe a consistent asymmetry between the two round-trip steps. The first step ($\Delta(T) \rightarrow S'$) achieves higher loadability and higher Δ detection than the second step ($S' \rightarrow T'$), because defects introduced in S' can propagate into T'. In borderline cases, two behaviors recur:

- **Self-healing**, where the second step partially repairs errors introduced in S' and produces a T' that better matches $\Delta(T)$.
- **Error masking**, where T' is internally consistent and loadable but faithfully preserves a flawed S', so the output deviates from the intended edit even though it appears valid.

These behaviors matter directly for reliability and sustainable performance engineering. In practice, borderline-valid outputs can trigger extra validation, regeneration, and manual inspection, which increases latency, API cost, and energy consumption across iterative workflows. A key takeaway is therefore not only whether an LLM can generate an artifact, but whether it can do so with predictable end-to-end reliability that avoids repeated rounds of compute.

5.2. Error Patterns by Δ Operators

A qualitative inspection of incorrect outputs indicates that failures are systematic and strongly correlated with specific Δ operators rather than random noise. Renaming changes (e.g., Δ_1, Δ_7) are detected with high correctness and aligned weighted success, but failures often manifest as stale references. For example, a class or attribute may be renamed at its declaration site, while legacy identifiers persist in association ends, foreign keys, or constraints. Figures 4 and 5 illustrate a case where a class renaming is not correctly propagated from $\Delta(T)$ to T' by Qwen. In this example, the class `DataNode` is renamed to `ElementNode` in both $\Delta(S)$ and $\Delta(T)$. However, the corresponding models S' and T' generated by Qwen still retain the original name `DataNode`. This partial propagation results in artifacts that remain syntactically valid and loadable, yet are semantically inconsistent with the intended change. Such cases help explain the observed discrepancy between Δ-level change detection and the weighted success metric.

Structural reorganizations (splitting/merging entities) are the most error-prone across all LLMs. Typical failures include incomplete redistribution of attributes, missing or duplicated associations, orphaned foreign keys, and asymmetric updates where one side of a relationship is modified but its counterpart is not. Figures 6 and 7 illustrate a case where Gemini partially understands a class-splitting operation but fails to preserve its intended semantics. In $\Delta(S)$ and $\Delta(T)$, the original class `Node` is split into two classes, `NodeMeta` and `NodeData`. However, in the generated models S' and T', Gemini incorrectly assigns `NodeMeta` as the supertype of `NodeData`. This misinterpretation results in an unintended inheritance relationship and indicates difficulties in coordinating global structural changes that require consistent and synchronized updates across multiple dependent elements.

Type-discipline edits (e.g., introducing enumerations) frequently yield inconsistent enforcement of value domains. Some outputs introduce an enum or lookup table but fail to update all dependent attributes or constraints; others replace free-form values without enforcing the restricted domain. This leads to drift despite correct change detection and reflects ambiguity in how LLMs choose among alternative valid encodings (e.g., lookup tables versus check constraints). Figures 8 and 9 illustrate a case where DeepSeek fails to detect and correctly propagate the introduction of a new enumeration. In $\Delta(S)$, a new enum named `KeyType` is added with multiple `Literal` values. However, when this change is propagated to S', `KeyType` is incorrectly transformed into a regular class, thereby losing its enumeration semantics. A similar issue is observed on the target side. When comparing $\Delta(T)$ and T', the relational table `KeyType`, which encodes the enum in the database schema, is replaced by an unrelated table named `List_end` with entirely different

(a) Ecore $\Delta(S)$ with $\Delta 1$ Operator

```
</eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="Node" abstract="true">
   <eOperations name="addLast" eType="#//Node">
     <eParameters name="newNode" eType="#//ElementNode"/>
   </eOperations>
   <eStructuralFeatures xsi:type="ecore:EReference" name="startOf" eType="#//
List"
       eOpposite="#//List/start"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="number" eType="ecore
:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
 </eClassifiers>
```

(b) Ecore S' by Qwen with $\Delta 1$ Operator

```
</eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="Node" abstract="true">
   <eOperations name="addLast" eType="#//Node">
     <eParameters name="newNode" eType="#//DataNode"/>
   </eOperations>
   <eStructuralFeatures xsi:type="ecore:EReference" name="startOf" eType="#//
List"
       eOpposite="#//List/start"/>
 </eClassifiers>
```

Figure 4: Example of error where Rename Class $\Delta 1$ is not propagated properly to S' by Qwen. Green highlights the intended renamed element; red highlights the stale retained element.

(a) XMI Δ(T) with Δ1 Operator

```
<ownedColumns
        name=" ElementNode "
        type="int"
        keys="//@ownedTables.0/@ownedForeignKeys.2">
      <properties>Unique</properties>
    </ownedColumns>
...
<ownedTables
      name=" ElementNode "
      referencingForeignKeys="//@ownedTables.0/@ownedForeignKeys.2">
...
```

(b) XMI T' by Qwen with Δ1 Operator

```
<ownedColumns
        name=" DataNode "
        type="int"
        keys="//@ownedTables.0/@ownedForeignKeys.2">
      <properties>Unique</properties>
    </ownedColumns>
...
<ownedTables
      name=" DataNode "
      referencingForeignKeys="//@ownedTables.0/@ownedForeignKeys.2">
...
```

Figure 5: Example of error where Rename Class Δ1 is not generated properly in T' by Qwen. Green highlights the intended renamed element; red highlights the stale retained element.

(a) Ecore Δ(S) with Δ8 Operator

```
  <eClassifiers xsi:type="ecore:EClass" name=" NodeMeta " abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="number"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
  </eClassifiers>

  <eClassifiers xsi:type="ecore:EClass" name=" NodeData " abstract="true">
    <eOperations name="addLast" eType="#//NodeData">
      <eParameters name="newNode" eType="#//DataNode"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="startOf" eType="#//
List"
        eOpposite="#//List/start"/>
  </eClassifiers>
```

(b) Ecore S' by Gemini with Δ8 Operator

```
  <eClassifiers xsi:type="ecore:EClass" name="NodeMeta" abstract="true">
    <eOperations name="addLast" eType="#//NodeMeta">
      <eParameters name="newNode" eType="#//DataNode"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="startOf" eType="#//
List"
        eOpposite="#//List/start"/>
  </eClassifiers>

  <eClassifiers xsi:type="ecore:EClass" name="NodeData"
 eSuperTypes="#//NodeMeta" >
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="number" eType="ecore
:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
  </eClassifiers>
```

Figure 6: Example of error where Split Class Δ8 is not propagated properly to S' by Gemini. Green highlights the intended split classes; red highlights the unintended inheritance relation.

attributes. As a result, the generated target model fails to preserve both the structural and semantic properties of the newly introduced enumeration.

Multiplicity and containment edits expose weaknesses in enforcing cross-element constraints. Models may adjust cardinalities or references locally but neglect corresponding updates to nullability, uniqueness, or referential integrity, producing artifacts that are syntactically valid yet violate intended semantics. Figures 10 and 11 illustrate a case where GPT fails to correctly propagate a change in multiplicity. In $\Delta(S)$, the upper bound of the reference `Pair.values` is changed from -1 to 1, indicating that each `Pair` may contain at most one `Value`. However, when this change is propagated to S', the upper bound is incorrectly reverted to -1, restoring a multi-valued relationship. A similar issue occurs on the target side. In $\Delta(T)$, the relationship is encoded using a `ForeignKey` in the `Value` table, meaning that each `Value` references exactly one `Pair` via `Value.values_inverse`. When GPT generates T', it represents the association between `Pair` and `Value` as a generic composition without enforcing a uniqueness constraint. As a result, the generated model fails to preserve the intended restriction that a `Pair` can be associated with at most one `Value`.

Across all Δ operators, three underlying causes recur:

- LLMs perform well on localized edits but struggle with changes that require maintaining invariants across multiple, non-local elements, which aligns with known difficulties in managing long-range dependencies during structured generation.
- Even when an LLM identifies the intended Δ, it does not consistently enumerate and update all dependent elements needed to preserve consistency, such as secondary constraints or derived references.
- LLMs tend to prioritize producing a loadable artifact over a fully correct one. As a result, round-trip execution can preserve errors rather than expose them; self-healing occurs but is rarer and hard to distinguish automatically from error masking.

A recurrent theme is the gap between change detection and end-to-end performance. Most penalties arise from missing lines (e.g., omitted constraints or absent mapping stubs) and incorrect lines (e.g., wrong column types or mismatched references) rather than harmful additions. We observe near-miss patterns where an entity is renamed but legacy identifiers persist in dependent elements, or where an enum is introduced, but value checks remain absent. Gemini 2.5 tends to maintain end-to-end coherence (fewer stale references) and sometimes improves in the second step; DeepSeek V3 often achieves high local correctness but can lose loadability after the round trip when unresolved references accumulate, although it sometimes benefits from second-step self-healing; GPT-5 varies more across instances, suggesting sensitivity to local conventions (e.g., identifier styles and key patterns) that can swing outcomes

(a) XMI Δ(T) with Δ8 Operator

```
<ownedTables
    name="NodeMeta"
    referencingForeignKeys="//@ownedTables.1/@ownedForeignKeys.0 ...">
  ...
</ownedTables>

<ownedTables
    name="NodeData"
  ...
  <ownedForeignKeys
      column="//@ownedTables.12/@ownedColumns.0"
      referencedTable="//@ownedTables.11">
    <ownedAnnotations
        annotation="composition"/>
    <ownedEvents/>
  </ownedForeignKeys>
</ownedTables>
```

(b) XMI T' by Gemini with Δ8 Operator

```
<ownedTables
    name="NodeMeta"
    referencingForeignKeys="//@ownedTables.1/@ownedForeignKeys.0 ...">
  ...
  <ownedForeignKeys
      column="//@ownedTables.2/@ownedColumns.0"
      referencedTable="//@ownedTables.0">
    <ownedAnnotations
        annotation="root"/>
    <ownedEvents/>
  </ownedForeignKeys>
</ownedTables>

<ownedTables
    name="NodeData"
    referencingForeignKeys="//@ownedTables.0/@ownedForeignKeys.9">
  ...
  <ownedForeignKeys
      column="//@ownedTables.5/@ownedColumns.0"
      referencedTable="//@ownedTables.2">
    <ownedAnnotations
        annotation="superType"/>
    <ownedEvents/>
  </ownedForeignKeys>
</ownedTables>
```

Figure 7: Example of error where Split Class Δ8 is not generated properly in T' by Gemini. Green highlights the intended split entities; red highlights the unintended supertype encoding.

between instance-specific mappings and over-general templates.

More broadly, the results support the example-centric hypothesis: with a single example pair, LLMs can infer transformation correspondences and apply them to new inputs that follow the same mapping conventions. Because our example pair comes from the same repository as the evaluated base pairs, models achieve strong performance on structure-preserving edits. However, increasing the number of example pairs does not lead to systematic improvements for more complex deltas. In particular, structural reorganizations such as class splitting and merging, as well as type-level changes such as enum introduction, remain error-prone even when additional examples are provided. This suggests that the primary bottleneck is not the absence of pattern exposure, but the difficulty of maintaining global structural and typing consistency across dependent elements during regeneration. The main limiting factors are therefore semantic reorganization (keeping redistributed attributes and links consistent end-to-end) and type normalization (enforcing constrained value domains). These borderline cases point to practical improvements, such as prompts that make dependency updates explicit, lightweight post-checks for referential integrity and enum coverage, and round-trip guards that separate genuine self-healing from error masking.

(a) Ecore Δ(S) with Δ10 Operator

```
<eClassifiers xsi:type="ecore:EEnum" name="KeyType">
  <eLiterals name="PRIMARY"/>
  <eLiterals name="SECONDARY"/>
  <eLiterals name="TERTIARY"/>
</eClassifiers>
```

(b) Ecore S' by DeepSeek with Δ10 Operator

```
<eClassifiers xsi:type="ecore:EClass" name="KeyType">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:
EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
```

Figure 8: Example of error where Introduce Enum Δ10 is not propagated properly to S' by DeepSeek. Green highlights the intended enum encoding; red highlights the incorrect class encoding.

(a) XMI Δ(T) with Δ10 Operator

```
<ownedTables
    name="KeyType">
  <ownedAnnotations
      annotation="enum"/>
  <ownedColumns
      name="id"
      type="int"
      keys="//@ownedTables.11/@ownedPrimaryKey">
    <properties>NotNull</properties>
  </ownedColumns>
  ...
</ownedTables>
```

(b) XMI T' by DeepSeek with Δ10 Operator

```
<ownedTables name="List_end">
  <ownedAnnotations annotation="cross"/>
  <ownedAnnotations annotation="unidirectional"/>
  <ownedAnnotations annotation="single"/>
  <ownedColumns name="source" type="int" keys="//@ownedTables.13/
@ownedForeignKeys.0">
    <properties>NotNull</properties>
  </ownedColumns>
  ...
</ownedTables>
```

Figure 9: Example of error where Introduce Enum Δ10 is not generated properly in T' by DeepSeek. Green highlights the intended enum table; red highlights the unrelated generated table.

(a) Ecore Δ(S) with Δ5 Operator

```
<eClassifiers xsi:type="ecore:EClass" name="Pair">
  <eStructuralFeatures xsi:type="ecore:EReference" name="values" upperBound="
1"
```

(b) Ecore S' by GPT with Δ5 Operator

```
<eClassifiers xsi:type="ecore:EClass" name="Pair" eSuperTypes="#//DataElement
">
  <eStructuralFeatures xsi:type="ecore:EReference" name="values" upperBound="
-1" eType="#//Value" containment="true" eOpposite="#//Value/pair" />
```

Figure 10: Example of error where Change Multiplicity Δ5 is not propagated properly to S' by GPT. Green highlights the intended upper bound; red highlights the incorrectly restored multi-valued bound.

(a) XMI Δ(T) with Δ5 Operator

```
<ownedTables name="Value" ...>
  ...
  <ownedColumns
      name="values_inverse"
      type="int"
      keys="//@ownedTables.8/@ownedForeignKeys.0">
    <ownedAnnotations annotation="unidirectional"/>
    <ownedAnnotations annotation="containment"/>
  </ownedColumns>

  <ownedForeignKeys
      column="//@ownedTables.8/@ownedColumns.1"
      referencedTable="//@ownedTables.6">
    ...
  </ownedForeignKeys>
</ownedTables>
```

(b) XMI T' by GPT with Δ5 Operator

```
<ownedTables name="Pair_values_inverse_Value_pair">
  <ownedColumns name="Pair" type="int">
    <properties>NotNull</properties>
  </ownedColumns>
  <ownedColumns name="Value" type="int">
    <properties>NotNull</properties>
  </ownedColumns>
  ...
</ownedTables>
```

Figure 11: Example of error where Change Multiplicity Δ5 is not generated properly in T' by GPT. Green highlights the intended foreign-key-based encoding; red highlights the incorrect join-table-based encoding.

5.3. Implications and Potential Solutions

The observed failure modes suggest that improving LLM reliability for round-trip software engineering requires architectural support beyond prompt refinement alone. This perspective also aligns with AI-for-performance engineering: when transformations are part of an automated pipeline, reliability failures translate into measurable overhead in latency and compute, and avoiding wasted retries directly supports cost and energy reduction.

- *Lightweight static analyses as guards*: simple post-generation checks, such as detecting stale identifiers, verifying referential integrity, and ensuring complete enum coverage, can catch many recurring errors with minimal overhead. These analyses complement LLM generation rather than replacing it, and they reduce wasted computation by failing fast before downstream steps amplify errors.
- *Delta-specific prompting*: instead of a uniform prompt template, we can tailor prompts to each Δ operator by enumerating the affected dependent element classes (e.g., associations, foreign keys, constraints). This can help the LLM internalize propagation scope for complex edits and reduce regeneration cycles caused by omitted dependent updates.
- *Round-trip validation as a first-class signal*: discrepancies between the intended Δ and the end-to-end round-trip result can be surfaced as explicit feedback to developers or to an automated regeneration loop. This feedback is also key to distinguishing genuine self-healing from silent error masking, and it provides an interpretable signal that can guide when to retry versus when to escalate to humans.
- *Hybrid transformation pipelines*: combining LLM-based generation with lightweight deterministic checks or transformation components can improve robustness. In such pipelines, LLMs propose candidate updates in response to evolution steps, while analyzers or rules validate, constrain, and refine these candidates to enforce structural and semantic guarantees. For example, graph-based checks can detect missing foreign keys or stale references after a split or merge, and rule-based repairs can resolve violations before completing the round trip. This division of labor balances LLM flexibility with deterministic precision, which reduces silent inconsistencies and can lower end-to-end execution cost by avoiding repeated trial-and-error generation.

Overall, our findings indicate that current LLMs are effective at detecting intended changes, but less reliable at coherently propagating them under the structural and semantic complexity inherent to realistic transformation scenarios Ferko, Berardinelli, Bucaioni, Behnam and Wimmer (2024). Closing this gap requires not only more capable models but also transformation architectures that encode dependency awareness and validation into the round-trip process, so that reliability improves without disproportionate increases in compute, latency, or energy use.

5.4. Threats to Validity

We acknowledge several threats to validity using the standard framework of construct, internal, external, and conclusion validity Cruzes and Ben Othmane (2017).

5.4.1. *Internal validity*

Effects attributed to the applied changes may be confounded by LLM non-determinism and vendor updates. We

mitigate this risk by fixing decoding settings and archiving prompts and generated artifacts, but we cannot eliminate it entirely. Using a single example pair focuses the evaluation on example-based induction, yet may favor mappings that are well illustrated by that example. Loadability checks depend on the toolchains and their configurations; parser strictness can cause failures unrelated to semantic correctness. Our Δ detection relies on minimal structural signatures, so incidental structures may lead to false positives or false negatives. Reusing the same base pairs across edits can introduce pair-specific bias; aggregating results over ten distinct base pairs per Δ reduces, but does not fully remove, this effect.

5.4.2. Construct validity

We measure transformation quality using line-based categories and a weighted metric over normalized XML/XMI representations (stable ordering, normalized whitespace). Line counts imperfectly reflect semantic correctness: small but critical changes may be underweighted, while harmless formatting differences may be overweighted. The weights assigned to correct, incorrect, missing, and additional lines encode design choices that may not align with all downstream utility functions. Classifying additional lines as harmless versus harmful and determining Δ detection can introduce assessor bias. We mitigate this risk through a written codebook, normalization, and simple, auditable signatures, although some subjectivity remains.

5.4.3. External validity

The study focuses on the UML-to-SQL setting, using ten source–target base pairs and ten predefined Δ operators. Generalization to other metamodels, serializations, domains, larger schemas, or transformation families is therefore uncertain. The use of a single fixed example pair controls prompt breadth and improves reproducibility, but may limit applicability when real systems follow different mapping conventions. LLM versions evolve, and future releases may behave differently. Possible exposure of benchmark artifacts in training data could influence performance; we did not attempt to measure this effect. Results may also depend on specific parser versions, toolchain configurations, and LLM API behavior.

A related reproducibility threat is our reliance on hosted API-based LLMs. Even with fixed prompts, decoding parameters, and archived outputs, exact replication may be affected by model updates, access changes, provider-side serving modifications, or obsolescence of specific model versions. This limits reproducibility and makes the results best interpreted as a snapshot of the evaluated model services at the time of the study. To mitigate this threat, we archive all prompts, generated artifacts, evaluation sheets, and model/version identifiers where available. Future work should extend the benchmark to locally hosted open-weight models to separate model capability from provider-specific API behavior and to improve long-term reproducibility.

Another limitation concerns transformation direction. Our protocol evaluates the round trip $\Delta(T) \rightarrow S' \rightarrow T'$, but not the reverse direction $\Delta(S) \rightarrow T' \rightarrow S'$. The reverse direction may expose different failure modes because the UML/Ecore side abstracts away some implementation-level schema details, while the SQL/XMI side makes keys, foreign keys, constraints, and association encodings explicit. As a result, errors may arise at different points in the cycle depending on which side is used as the starting point. We therefore interpret our findings as direction-specific evidence for target-started round-trip co-evolution, rather than as a complete characterization of bidirectional behavior Eramo and Bucaioni (2013). Evaluating the reverse direction is an important extension for future work.

A further limitation is that each evolution step applies a single atomic Δ operator, whereas real-world version histories often combine multiple changes within a single commit or editing session. However, atomic deltas are a well-established abstraction in studies of model, schema, and code evolution, serving as canonical building blocks for co-evolution and synchronization research. Our design follows this tradition to enable controlled, causal analysis of LLM behavior under clearly defined evolution scenarios. That said, interactions between multiple concurrent changes are not captured. An important direction for future work is to study composed or compound Δ configurations, to assess whether LLM performance on such cases composes from its behavior on atomic deltas or whether new failure modes emerge due to change interactions.

5.4.4. Conclusion validity

Our summaries are based on ten runs per Δ per LLM, yielding repeated measurements but relatively small per-cell sample sizes and potential dependence, as the same base pairs are reused across edits. We report means and standard deviations and make no claims of statistical significance. Assumptions such as independence and equal variance may be violated if shared artifacts induce correlated errors across edits or models. Multiple comparisons across LLMs and Δ operators increase the risk of spurious patterns. We mitigate this threat by fixing decoding settings, normalizing outputs to reduce formatting noise, and reporting dispersion to highlight instability rather than over-interpreting small differences.

6. Conclusion and Future Work

To the best of our knowledge, this work is the first change-aware, round-trip benchmark of large language models for example-centric model transformation in a UML–SQL co-evolution setting, bridging class-oriented data models and relational database schemas. Using ten base pairs and ten change configurations, we evaluated four models, GPT-5, DeepSeek V3, Qwen3-Next-80B-A3B, and Gemini 2.5, under a conformance-first, example-driven protocol with weighted line-based scoring, loadability checks, and Δ-detection checks.

Our evaluation targets core software evolution concerns, including co-evolution, change propagation, and regression risk. Once correspondences are established, nominal and locally structural edits (e.g., rename, move/delete attribute, multiplicity updates, and containment-to-reference) are handled reliably. In contrast, semantically intensive restructurings (e.g., splitting or merging classes) and type normalization (e.g., introducing enums) remain challenging because they require coordinated updates across many dependent elements. From a performance and sustainability perspective, these failures also matter because they trigger extra validation, regeneration, and manual review, which increases latency and computational cost in iterative workflows.

Future work should therefore improve robustness through hybrid strategies that pair LLMs with explicit validation and analysis. Structured pipelines and ensembles can convert high change detection into end-to-end consistency by adding graph- and constraint-based checks, Δ-specific signatures, and lightweight post-processing that detects and repairs inconsistencies before completing the round trip. We also expect cooperative prompting and retrieval-augmented generation, grounded in metamodel fragments, mapping patterns, and schema histories, to reduce hallucinations and improve semantic accuracy, especially for restructurings.

Looking ahead, we plan to extend the round-trip protocol beyond UML and SQL to additional modeling and implementation languages and to larger, heterogeneous systems, which will allow us to study scalability and generalizability in realistic software evolution settings. Ultimately, these directions move toward explainable, analysis-driven transformation agents that combine example-based inference with principled validation to deliver trustworthy co-evolution with predictable, efficient execution.

7. Data Availability

We have made a public replication package available at: https://doi.org/10.5281/zenodo.17352440

References

Allamanis, M., Panthaplackel, S., Yin, P., 2024. Unsupervised evaluation of code llms with round-trip correctness. URL: https://arxiv.org/abs/2402.08699, arXiv:2402.08699.

An, Y., Hu, X., Song, I.Y., 2008. Round-trip engineering for maintaining conceptual-relational mappings, in: Bellahsène, Z., Léonard, M. (Eds.), Advanced Information Systems Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 296–311.

Buchmann, T., Westfechtel, B., 2013. Towards incremental round-trip engineering using model transformations, in: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, pp. 130–133. doi:10.1109/SEAA.2013.19.

Campos, G., Affonso, F., 2025. Dynaschema: A library to support the relational data schema evolution for the self-adaptive software domain, pp. 722–733. doi:10.5220/0013349000003929.

Ceri, S., Fraternali, P., 1997. Designing database applications with objects and rules: The idea methodology. URL: https://api.semanticscholar.org/CorpusID:46553323.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P.S., Yang, Q., Xie, X., 2024. A survey on evaluation of large language models. ACM Trans. Intell. Syst. Technol. 15. URL: https://doi.org/10.1145/3641289, doi:10.1145/3641289.

Cruzes, D., Ben Othmane, L., 2017. Threats to Validity in Empirical Software Security Research. pp. 275–300. doi:10.1201/9781315154855-10.

Dao, D., Bucaioni, A., Cicchetti, A., 2025. Learning to transform: Evaluating llms on model transformation by example, in: MDE Intelligence 2025. URL: http://www.es.mdu.se/publications/7236-.

Eramo, R., Bucaioni, A., 2013. Understanding bidirectional transformations with tggs and jtl. Electronic Communications of the EASST 57.

Ferko, E., Berardinelli, L., Bucaioni, A., Behnam, M., Wimmer, M., 2024. Towards interoperable digital twins: Integrating sysml into aas with higher-order transformations, in: 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), IEEE. pp. 342–349.

France, R., Kim, D.K., Ghosh, S., Song, E., 2004. A uml-based pattern specification technique. IEEE Transactions on Software Engineering 30, 193–206. doi:10.1109/TSE.2004.1271174.

Greiner, S., Buchmann, T., Westfechtel, B., 2016. Bidirectional transformations with qvt-r: A case study in round-trip engineering uml class models and java source code, in: 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 15–27.

Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S., 2024. A survey on large language models for code generation. URL: https://arxiv.org/abs/2406.00515, arXiv:2406.00515.

Kazai, G., Osei, R.A., Bucaioni, A., Cicchetti, A., 2025. Model transformations using llms out-of-the-box: Can accidental complexity be reduced?, in: First Workshop on Large Language Models For Generative Software Engineering. URL: http://www.es.mdu.se/publications/7201-.

Kretschmer, R., Khelladi, D., Lopez-Herrejon, R., Egyed, A., 2021. Consistent change propagation within models. Software and Systems Modeling 20. doi:10.1007/s10270-020-00823-4.

Lachaux, M.A., Roziere, B., Chanussot, L., Lample, G., 2020. Unsupervised translation of programming languages. URL: https://arxiv.org/abs/2006.03511, arXiv:2006.03511.

Laskar, M.T.R., Alqahtani, S., Bari, M.S., Rahman, M., Khan, M.A.M., Khan, H., Jahan, I., Bhuiyan, A., Tan, C.W., Parvez, M.R., Hoque, E., Joty, S., Huang, J., 2024. A systematic survey and critical review on evaluating large language models: Challenges, limitations, and recommendations, in: Al-Onaizan, Y., Bansal, M., Chen, Y.N. (Eds.), Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Miami, Florida, USA. pp. 13785–13816. URL: https://aclanthology.org/2024.emnlp-main.764/, doi:10.18653/v1/2024.emnlp-main.764.

Moezkarimi, Z., Eriksson, K., Johansson, A.A., Bucaioni, A., Sirjani, M., . Harnessing chatgpt for model transformation in software architecture: From uml state diagrams to rebeca models for formal verification, in: 4th International Workshop of Model-Driven Engineering for Software Architecture. URL: http://www.ipr.mdu.se/publications/7130-.

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

Qiu, D., Li, B., Su, Z., 2013. An empirical analysis of the co-evolution of schema and code in database applications, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 125–135. URL: https://doi.org/10.1145/2491411.2491431, doi:10.1145/2491411.2491431.

Rahm, J., Graube, M., Urbas, L., 2017. A proposal for an interactive roundtrip engineering system, in: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–7. doi:10.1109/ETFA.2017.8247576.

Rosca, D., Domingues, L., 2021. A systematic comparison of roundtrip software engineering approaches applied to uml class diagram. Procedia Computer Science 181, 861–868. URL: https://www.sciencedirect.com/science/article/pii/S1877050921002830, doi:https://doi.org/10.1016/j.procs.2021.01.240. cENTERIS 2020 - International Conference on ENTERprise Information

Systems / ProjMAN 2020 - International Conference on Project MANagement / HCist 2020 - International Conference on Health and Social Care Information Systems and Technologies 2020, CENTERIS/ProjMAN/HCist 2020.

Ruiz, F.V., Grishina, A., Hort, M., Moonen, L., 2024. A novel approach for automatic program repair using round-trip translation with large language models. URL: https://arxiv.org/abs/2401.07994, arXiv:2401.07994.

Stevens, P., 2007. Bidirectional model transformations in qvt: Semantic issues and open questions, in: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (Eds.), Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 1–15.

Stevens, P., 2020. Maintaining consistency in networks of models: bidirectional transformations in the large. Software & Systems Modeling 19. doi:10.1007/s10270-019-00736-x.

Vassiliadis, P., Shehaj, F., Kalampokis, G., Zarras, A.V., 2023. Joint source and schema evolution: Insights from a study of 195 foss projects, in: Stoyanovich, J., Teubner, J., Mamoulis, N., Pitoura, E., Mühlig, J. (Eds.), Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023, OpenProceedings.org. pp. 27–39. URL: https://dx.doi.org/10.48786/edbt.2023.03, doi:10.48786/edbt.2023.03.

Wimmer, M., Pérez, S., Jouault, F., Cabot, J., 2012. A catalogue of refactorings for model-to-model transformations. The Journal of Object Technology 11. doi:10.5381/jot.2012.11.2.a2.

Yang, B., Cai, Z., Liu, F., Le, B., Zhang, L., Bissyandé, T.F., Liu, Y., Tian, H., 2025. A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications. URL: https://arxiv.org/abs/2506.23749, arXiv:2506.23749.