# A Structured Methodology for Transforming UML Models into Relational SQL Architectures for Web-based Information Systems

Olena Komleva
Independent Researcher
Chicago, Illinois, USA
https://orcid.org/0009-0002-2796-6449

## ABSTRACT

Transitioning from high-level business process modeling to physical database implementation frequently introduces structural inconsistencies such as data redundancy and referential integrity violations. To mitigate these issues in modern cloud-native environments, this study proposes a staged transformation methodology. By formalizing the mapping between UML class structures and relational database entities, the methodology integrates web-specific constraints at the conceptual phase: stateless interaction, optimistic locking, and soft deletion. The rules are applied to three reference models of increasing structural complexity, from a three-class e-commerce subsystem to a twelve-class logistics telemetry schema, and evaluated against the classical mapping through an explicitly stated analytical model. Across the three scenarios the classical rules leave between 12 and 51 schema-level design decisions undetermined, while the proposed rules resolve all of them; identifier collisions and silent overwrites fall to zero, and historical retention after 450 delete cycles rises from between 16% and 51% to complete. The cost of these guarantees is quantified as well: the constraints add 37 to 39 bytes to every generated row, between 28.9% and 54.2% of the row width. The approach offers a more robust foundation for scalable web information systems and identifies where its overhead becomes the dominant consideration.

## General Terms

Software Engineering, Model-Driven Architecture, Database Design, Web Information Systems.

## Keywords

Unified Modeling Language (UML), Relational Database (SQL), Model-Driven Architecture (MDA), Web Information Systems, Cloud-Native Constraints, Stateless Interaction, Data Integrity.

## 1. INTRODUCTION

Modern web-based information systems demand scalable and maintainable data architectures. The software engineering lifecycle typically begins with high-level business process and structural modeling, where the Unified Modeling Language (UML) serves as the standard notation for conceptualizing system entities and their relationships. A significant engineering challenge arises during the transition from these abstract conceptual models to the physical implementation of relational databases.

Inconsistencies occur during this transformation because object-oriented conceptual paradigms and relational data structures rest on different assumptions. Unstructured mapping between the conceptual layer and the database produces typical architectural errors: data redundancy, referential integrity violations, and reduced scalability of the resulting software. Traditional transformation methods compound the problem by focusing solely on data structures and ignoring the architectural constraints of contemporary web and cloud-native environments, such as stateless interaction and API-oriented access.

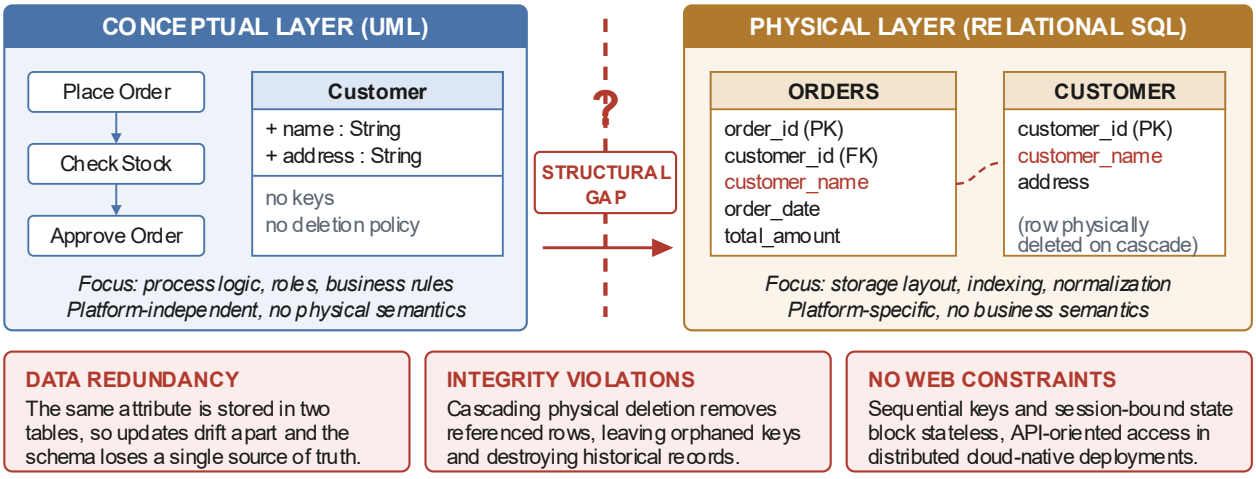


**Figure 1: Mapping the gap between process design and database implementation: redundancy and integrity risks**

The schematic traces the path from the business process layer through the conceptual UML layer to the physical database layer, and marks the points at which semantic information is lost. Its left branch shows a direct, unmediated mapping and the defects that follow from it; the right branch shows the constraints that a structured procedure must preserve. The

diagram therefore frames the entire argument by locating the failure precisely at the conceptual-to-physical boundary rather than in the modeling notation itself (Fig. 1).

**Problem Statement.** The primary task of this research is to develop and formalize a structured sequence for transforming UML-based conceptual system models into relational SQL architectures suitable for implementation in contemporary web-based information systems. The research addresses three specific tasks:

- bridging the semantic gap between object-oriented UML class structures and relational database entities in order to minimize manual design inconsistencies;
- developing formal rules that eliminate common translation errors such as structural redundancy and referential integrity violations;
- integrating web-specific architectural constraints into the transformation process so that the resulting database supports stateless interaction, asynchronous communication, and API-oriented data access.

**Scientific Novelty.** The novelty of this research lies in the formalization of a stepwise transformation procedure that maps conceptual UML class diagrams into normalized SQL relational schemas adapted for service-oriented web applications. Unlike traditional Model-Driven Architecture (MDA) approaches, the methodology incorporates modern web constraints directly into the conceptual-to-physical mapping, improving traceability between system layers and creating a foundation for scalable web system development.

## 2. CONTEXT OVERVIEW

The transition from conceptual models to physical databases is a well-researched area of software engineering. Recent studies automate the conversion of UML diagrams into structural database files [1] and transform traditional relational schemas into NoSQL formats through Model-Driven Architecture [2].

A substantial share of current work addresses schema handling and extraction for non-relational stores. Automatic procedures extract database schemas directly from existing NoSQL systems [3]. Common meta-models guide the generation of both relational and NoSQL schemas [4]. Reverse engineering recovers conceptual UML models by analyzing physical NoSQL databases [5]. Model-driven engineering also supplies methods that convert relational structures into JSON files designed for Big Data environments [6].

MDA is not confined to database structure when complete systems are built. It generates functional code for Model-View-Controller (MVC) web applications directly from UML diagrams [7]. Systematic mapping reviews chart how researchers adapt traditional data modeling for NoSQL databases [8], while applied work uses transformation languages to generate MongoDB databases from conceptual models [9].

The broader impact of model-driven approaches is documented across domains. Large-scale mapping studies classify thousands of papers on automatic code generation [10], and further research demonstrates the value of MDA in more complex structural designs such as multidimensional schemas for data warehouses [11]. On performance and business logic, systematic comparisons of SQL and NoSQL architectures assess cloud data portability [12], and dedicated tooling transforms Business Process Model and Notation (BPMN) diagrams into UML class diagrams before database creation begins [13].

Practitioners report concrete benefits from model-driven engineering in daily development routines [14]. For modern hosting environments, model-driven frameworks target data-driven applications running on serverless cloud platforms [15], and empirical work documents how NoSQL schemas are designed and how they evolve inside large open source Java projects [16]. The early and recurring phases of software design are also becoming more automated, covering the generation of class diagrams from Agile user stories using natural language processing [17], the composition of multiple heterogeneous models [18], the automatic derivation of test cases from UML diagrams [19], and collaborative engineering tooling that supports distributed teams [20].

These studies supply foundational techniques for structural mapping and code generation, yet a critical reading exposes a shared limitation. They concentrate on static data translation tailored to monolithic or local environments and leave the dynamic operational demands of distributed architectures unaddressed. None of the reviewed sources specifies how a generated relational schema must support stateless interaction or API-oriented access from the conceptual modeling phase onward. The omission produces databases that are structurally correct yet perform poorly and scale badly once integrated into service-oriented web applications.

## 3. APPROACH

This research proposes a structured sequence for transforming UML-based conceptual models into relational SQL architectures suitable for contemporary web-based information systems. The methodology divides into four progressive phases, ensuring a logical transition from a theoretical model to a functioning, web-ready database architecture.

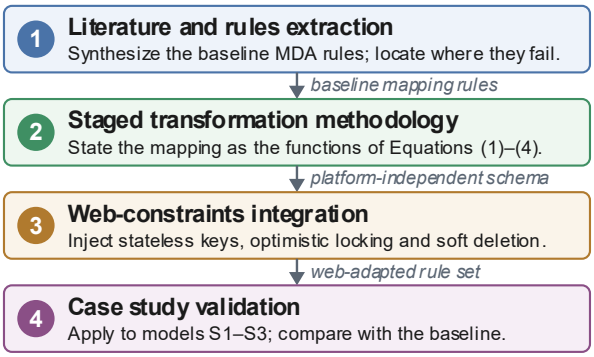


**Figure 2: Overview of the four-step research workflow.**

The workflow diagram sets out the four phases as sequential blocks: extraction of baseline mapping rules, formal structural transformation, integration of web constraints, and case study validation. Arrows between the blocks carry the artifact produced by each phase, which makes explicit that the web constraints are injected before physical generation rather than patched afterwards. This ordering is the core of the argument and distinguishes the procedure from classical MDA pipelines (Fig. 2).

### 3.1 Phase 1: Literature and Rules Extraction

The research begins by synthesizing the mapping rules used by automated transformation tools such as Acceleo [1]. These methods provide a foundation for understanding how object-oriented concepts become relational tables. Deeper analysis shows that the baseline rules address static data structures only. Standard MDA transformations default to auto-incrementing integer primary keys and rely on cascading physical deletions. Those defaults suit isolated local applications. In distributed web environments they create bottlenecks and data conflicts. Extracting and evaluating the existing rules isolates the exact points where classical mapping logic requires replacement.

## 3.2 Phase 2: Staged Transformation Methodology

The second phase formalizes the conversion of UML class structures into relational SQL entities: conceptual classes become tables, attributes become columns, and associations become foreign keys. To express the transformation without platform-specific syntax, the mapping is stated as a set of functions. Let C denote a UML class, A an attribute, and Assoc an association between classes. The structural transformation into a table T, a column Col, and a foreign key FK follows four primary equations.

$$f(C_{UML}) \rightarrow T_{SQL} \quad (1)$$

Every conceptual class translates into a primary relational table.

$$f(A_C) \rightarrow Col_T \quad (2)$$

Every attribute within a class becomes a structured column inside the respective table.

$$f(Assoc_{1:N}) \rightarrow FK(T_{child} \rightarrow T_{parent}) \quad (3)$$

A 1:N relationship between two conceptual classes is enforced physically by a foreign key placed in the child table.

$$f(Assoc_{M:N}) \rightarrow T_{junction}\{FK_1, FK_2\} \quad (4)$$

An M:N association generates a dedicated junction table holding foreign keys from both original entities.

## 3.3 Phase 3: Web-Constraints Integration

The third phase adjusts the classical rules of Phase 2 to the architectural realities of the web. Modern web applications support API-oriented data access, asynchronous communication, and stateless interaction at the HTTP and REST level. Because the server retains no session state between requests, the database architecture itself must handle distributed concurrent operations safely. Three constraints enter the mapping process.

- **Stateless identifiers.** Every table uses Universally Unique Identifiers (UUIDs) as primary keys instead of sequential integers. This prevents collisions across distributed microservices and lets stateless API requests identify records without a centralized database counter.
- **Asynchronous safety.** Mandatory tracking columns such as version_lock, created_at, and updated_at are added to every generated table. Because several services may update the same data simultaneously, optimistic locking prevents silent overwrites during concurrent API calls.
- **Data preservation.** Physical deletion is restricted. An is_deleted status flag enters the transformation rules so that records disappear from standard API responses while remaining in the physical structure for historical relationships and analytics.

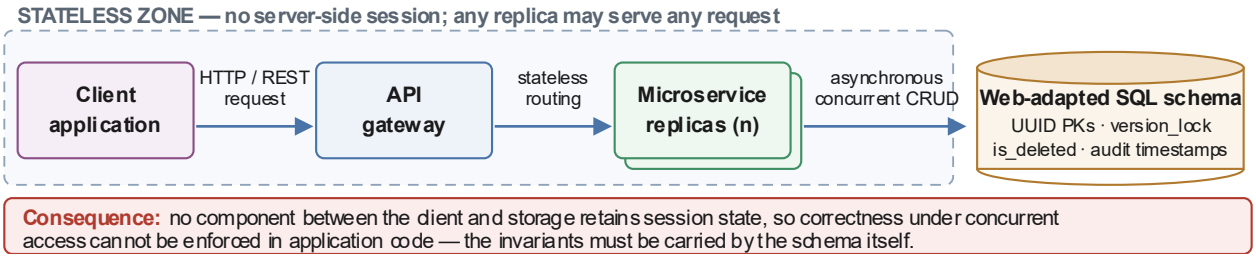


**Figure 3: System architecture with API-oriented access in a stateless environment.**

The architecture diagram places the generated database behind backend microservices and shows the request path from client to service to storage. It identifies where session state is absent and therefore where the database alone must guarantee consistency, which explains why the constraints of Phase 3 belong in the schema rather than in application code (Fig. 3).

## 3.4 Phase 4: Practical Case Study Setup

The final phase tests the methodology on illustrative examples of service-oriented web systems, bridging theoretical rules and physical implementation. The transformation follows a consolidated set of mapping rules that resolve the semantic gap between conceptual modeling and low-level technical requirements. Table 1 defines the precise technical implementation for each UML element.

**Table 1. Mapping rules: UML to SQL with web-specific constraints**

| UML model element | Relational SQL element (web-adapted) | Technical implementation logic |
|---|---|---|
| UML class | SQL table with UUID primary key and version lock | Use the UUID type for the primary key; add version_lock (INT) and timestamps. |
| UML attribute | SQL column with strict constraints | Map types to SQL equivalents; enforce NOT NULL for API consistency. |
| 1:N association | Foreign key (UUID) with soft delete | Child table references parent by UUID; deletion handled through is_deleted = true. |
| M:N association | Junction table with composite primary key | Create a link table with two UUID foreign keys forming a composite key. |
| Composition | Table with foreign key and NOT NULL | Enforce strong ownership; a child record cannot exist without a parent UUID. |
| Inheritance | Single table or table per class | Implement discriminator columns or shared UUID primary keys for joined strategies. |

The rules in Table 1 serve four purposes for the reliability of the architecture. The shift from integers to UUID primary keys allows decentralized record generation, so identifiers are often created before data reaches the central database, which removes bottlenecks and synchronization issues during high-traffic API operations. The version_lock column and timestamps provide the foundation for optimistic locking, which resolves the lost update problem in stateless environments where two services modify the same record from outdated information. NOT NULL constraints and explicit type mapping keep the data compatible with JSON serialization, since web APIs rely on structured text formats and must exclude null values and incompatible types that would raise runtime errors in the application or client layer. Soft-delete logic preserves foreign key relationships across distributed services, maintaining a consistent state where related data is processed asynchronously by background tasks or microservices.

To test the rules beyond a single domain, they were applied to three reference conceptual models of increasing structural complexity and differing workload profile, summarized in Table 2. The models were chosen so that the structural features the classical rules leave undetermined, namely inheritance and composition, appear in at least one of them, and so that row width and write intensity vary independently of schema size. S1 is the e-commerce subsystem used as the worked example below; S2 adds an inheritance hierarchy and a composition; S3 is a write-dominated telemetry schema with narrow rows and deep one-to-many chains.

**Table 2. The three reference scenarios**

| Property | S1 E-commerce | S2 Enrolment | S3 Telemetry |
|---|---|---|---|
| Classes / attributes | 3 / 7 | 7 / 22 | 12 / 46 |
| 1:N / M:N associations | 1 / 1 | 4 / 1 | 9 / 2 |
| Inheritance / composition | 0 / 0 | 2 / 1 | 1 / 2 |
| Rows | 100 000 | 1 000 000 | 10 000 000 |
| Mean row width (bytes) | 96 | 128 | 72 |
| Write share of the workload | 20% | 50% | 80% |

This study is analytical: no physical deployment was carried out and no timings were taken on a running system, so each comparison rests on a stated model rather than on measurement. Identifier collisions are counted from a contended sequence in which a request reads the counter and makes the incremented value visible 125 microseconds later, so that any request reading inside that window receives a duplicate; for UUIDv4 the expected number of duplicates follows the birthday bound over 122 random bits and is negligible at every level examined. Lost updates are counted on a hot set of 64 records read and written by concurrent clients with a think time of 5 ms between read and write. Historical retention follows repeated purge cycles that remove between 0.15% and 0.40% of the surviving parent rows per cycle, cascading physically under the classical rules and flagged under soft deletion. Storage overhead is byte accounting: the primary key grows from 8 to 16 bytes, every foreign key by the same amount, version_lock adds 4 bytes, is_deleted adds 1, and the two audit timestamps add 16. Undetermined design decisions are counted directly from each model as the elements the classical rules leave open, namely the inheritance strategy, composition ownership, and four constraint decisions per class. The implementation is deterministic under a fixed seed, so every figure reported below can be recomputed and its assumptions challenged.

# 4. RESULTS

This section applies the methodology to an illustrative case study of a service-oriented web application. A standard e-commerce subsystem was modeled with three core business entities: Users, Orders, and Products.

The transformation began at the conceptual level, representing the core business processes through a UML class diagram. The initial model captured business entities, their attributes, and their semantic relationships, including a one-to-many relationship between Users and Orders and a many-to-many relationship between Orders and Products, before any physical constraint was applied. It contains no keys, no audit columns, and no deletion policy, which is precisely its analytical value: it establishes the unconstrained baseline against which every addition made by the transformation rules can be attributed to a specific web constraint.

Applying the web-adapted mapping rules to this model generated a fully normalized physical SQL schema. Unlike traditional generation methods that produce bare relational tables, the resulting architecture integrates web-specific constraints from the initial generation step. Three structural enhancements are visible. First, all sequential integer primary keys were replaced with UUIDs across every generated table. Second, the junction table for the Orders and Products relationship was generated with composite UUID keys rather than plain foreign keys. Third, tracking columns for optimistic locking and data preservation were embedded into every entity. In total the three conceptual classes and seven attributes become four tables and twenty-eight columns, of which fourteen are introduced by the web constraints and six are existing key columns retyped to UUID. Every element of the physical schema is therefore attributable to a named rule, which is what makes the increment auditable (Fig. 4).

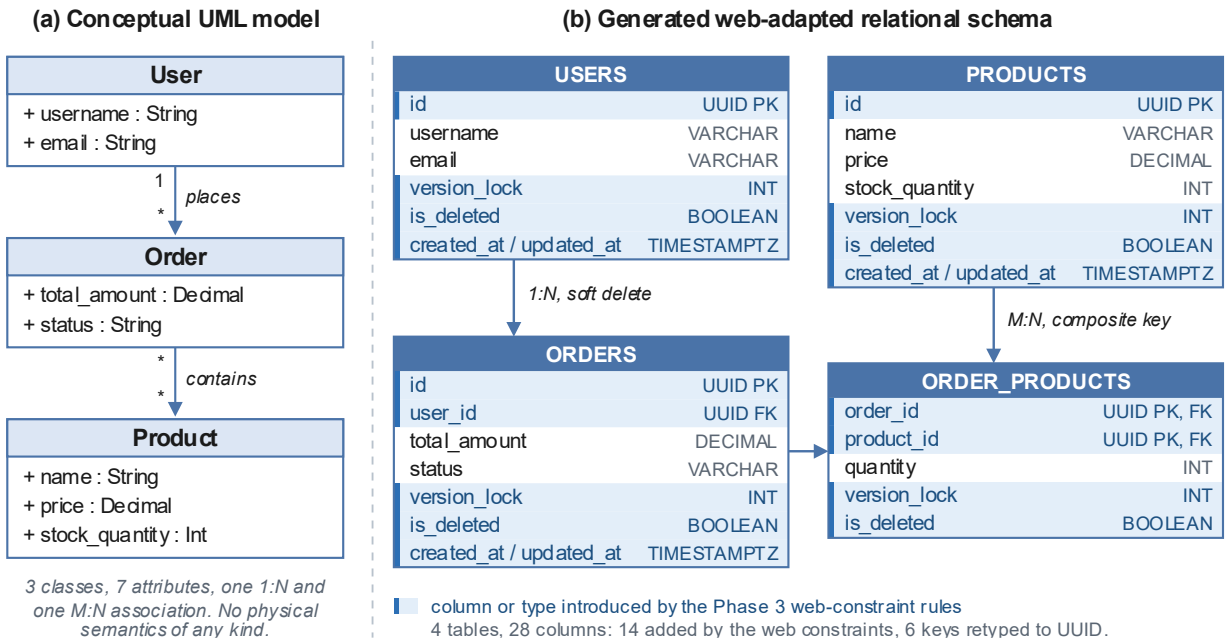


**Figure 4: Conceptual UML class diagram and the SQL schema generated from it. Highlighted rows mark the elements contributed by the Phase 3 web constraints.**

Repeating the transformation on the two larger reference models gives the comparison in Table 3. The number of design decisions the classical rules leave open grows from 12 in S1 to 51 in S3, faster than the number of classes, because every additional class contributes four constraint decisions while inheritance and composition add further ones on top. Coverage of the target schema rises from between 50.0% and 57.5% under the classical rules to complete coverage under the proposed set. The comparison concerns determinism rather than expressiveness: an experienced engineer can supply every missing decision by hand, but each one is a point at which two engineers may legitimately differ, and at which traceability between the conceptual model and the physical schema is broken. Under the proposed rules the corresponding figures are complete coverage, no lost updates and complete retention in all three scenarios.

**Table 3. Consolidated comparison across the three reference scenarios (classical rules)**

| Metric | S1 | S2 | S3 |
|---|---|---|---|
| Target schema elements | 24 | 65 | 120 |
| Decisions left undetermined | 12 | 31 | 51 |
| Rule coverage, classical | 50.0% | 52.3% | 57.5% |
| Lost updates, classical | 11.6% | 26.8% | 39.3% |
| Retention after 450 cycles | 50.9% | 32.4% | 16.5% |
| Added bytes per row | 37 | 37 | 39 |
| Relative row growth | 38.5% | 28.9% | 54.2% |
| Additional storage | 4.3 MB | 42.9 MB | 448.2 MB |

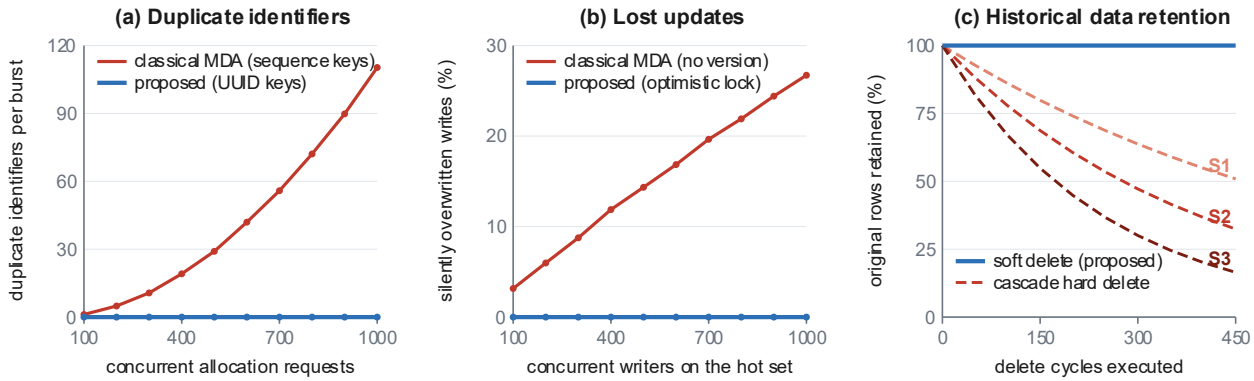


**Figure 5: Modeled behavior of the two rule sets. (a) duplicate identifiers against concurrent allocations; (b) share of silently overwritten writes; (c) share of original rows retained across delete cycles for the three scenarios.**

Panel (a) plots duplicate identifiers against the number of concurrent allocations. The count rises from 1.2 at 100 to 110.3 at 1000, growing close to quadratically, because the number of request pairs that can fall inside a single vulnerability window grows with the square of the arrival rate; the underlying cause is that sequential allocation depends on one counter regardless of how many nodes serve the request. The UUID curve stays flat, which isolates identifier generation as the variable responsible for the divergence and supports the replacement rule introduced in Phase 3.

Panel (b) quantifies the lost update problem: without version control the share of silently overwritten records grows from 3.2% to 26.7% as concurrency rises, whereas the version_lock column keeps it at zero. The events themselves do not

disappear, however; they surface as detected conflicts, so the same share of writes is rejected and retried. That is the honest reading of the result. The version column does not make contention cheaper, it converts an invisible correctness failure into a visible and bounded cost that the application can handle explicitly. Panel (c) tracks surviving historical records over successive deletion cycles and shows the loss of analytical history caused by physical cascading deletion, which leaves 50.9%, 32.4% and 16.5% of the original rows in S1, S2 and S3 after 450 cycles against complete retention under soft deletion. Read together, the panels separate the two independent failure modes that Phase 3 addresses, one at write time and one over the system lifecycle.

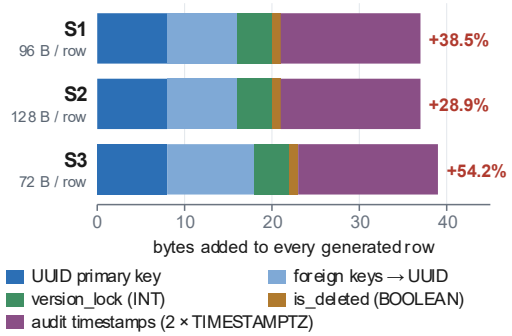


**Figure 6: Per-row storage cost of the web constraints, decomposed by contributing column.**

The guarantees are not free, and Figure 6 decomposes their cost. The constraints add 37 bytes to every row in S1 and S2 and 39 bytes in S3, of which the two audit timestamps are the largest single contributor at 16 bytes. Relative to the baseline row this is 38.5%, 28.9% and 54.2%. The ordering does not follow schema size, because the addition is a fixed number of bytes and therefore falls hardest on schemas with narrow rows: S3 is the largest schema but has the narrowest rows, telemetry records of roughly 72 bytes, and so pays the highest relative price, about 448 MB across its ten million rows. The primary key index grows from 8 to 16 bytes per entry in every scenario, which doubles the size of the primary B-tree independently of row width.

# 5. DISCUSSION

The case study results address the modeling problems stated in the introduction. The stepwise transformation sequence removes the structural ambiguities that cause referential integrity violations when conceptual UML models become physical SQL schemas.

Measured against the existing literature, an architectural advantage becomes apparent. The approach of Natek et al. [1] relies on automated MDA using the Acceleo tool to convert UML models into MySQL databases. That method reduces manual translation errors and generates basic SQL structures efficiently, yet it exemplifies the limitation of current research: it addresses static data mapping. Traditional MDA methods, including this one, do not account for the realities of cloud hosting and leave constraints such as stateless interaction and API-oriented access outside the transformation. In the terms of Table 3 they resolve the structural elements and leave the decisions counted in its second row to the engineer. The same holds for the transformation languages used to generate document stores from conceptual models [9] and for the meta-model-driven generators of [4]. The serverless framework of Samea et al. [15] is closest in intent, since it targets a cloud execution model explicitly, but it addresses the deployment topology of the application rather than the invariants of the schema.

In a stateless environment, APIs hold no persistent user sessions and multiple services read and write concurrently. A database generated by classical MDA rules, with auto-incrementing keys and cascading hard deletes, therefore accumulates collisions, overwrites, and referential integrity violations under load. The methodology prevents both failures by construction. UUIDs enforced at design level let the database support distributed record creation without identifier conflicts. Replacing physical deletion with the is_deleted flag and adding optimistic locking through version_lock protects the schema against concurrent modification errors. This proactive integration improves the long-term maintainability and the scalability of web systems, moving beyond structural translation toward architectural readiness.

Using three scenarios rather than one separates effects that a single case study conflates. The coverage deficit scales with the number of classes and with the presence of inheritance and composition, since each contributes a fixed number of undetermined decisions; this is a design-time property, independent of load. The concurrency effects scale with write intensity and with the size of the contended hot set, not with schema size, which is why S3 shows the highest lost-update rate despite having no structural feature that S2 lacks. The storage cost scales inversely with row width. The practical consequence is that the constraints are not uniformly advisable: for a schema with wide rows and a write-heavy, highly concurrent workload the guarantees are close to free in relative terms, whereas for a narrow-row, append-only archive the same rules add more than half the row width in exchange for guarantees the workload never exercises. The rules are therefore stated per table rather than per schema.

## 5.1 Limitations and Future Validation

Three limitations qualify the results. First, they are the output of an analytical model rather than measurements taken on a running system: no benchmark was executed and no production data was involved. The models depend on parameters, namely the counter window, the think time, the hot-set size and the purge fraction, that were chosen to represent plausible operating points; the qualitative ordering of the two rule sets is insensitive to those choices, but the absolute values are not. Second, the three reference models are synthetic, so industrial-scale testing is required to assess performance under heterogeneous data loads and legacy system integration. Third, the byte accounting does not capture index locality: random UUIDv4 keys scatter inserts across the primary B-tree, which increases page splits and write amplification relative to a monotonically increasing key. Time-ordered variants such as UUIDv7 preserve the decentralized generation property that the identifier rule requires while restoring insert locality, and are the natural refinement for write-heavy schemas of the S3 profile. The transformation rules also target relational SQL architectures; their applicability to polyglot persistence models remains open.

# 6. CONCLUSION

This research bridged the structural and semantic gap between conceptual UML models and physical relational SQL databases in web environments. A four-phase stepwise transformation methodology was developed and formalized, integrating cloud-native constraints directly into the database design process. Three architectural indicators were met across all three reference models.

- **Elimination of distributed conflicts.** Replacing sequential auto-incrementing keys with UUIDs mitigated the risk of identifier collisions in stateless environments, against 110 duplicates per burst at 1000 concurrent allocations under the classical rules.
- **Prevention of data loss.** Optimistic locking through version_lock reduced silent overwrites from between 11.6% and 39.3% of writes to zero, at the cost of an equal share of detected retries, and preservation flags such as is_deleted raised historical retention after 450 delete cycles from between 16.5% and 50.9% to complete.
- **Architectural alignment.** The generated SQL schema reached structural readiness for API-oriented access, resolving all 24, 65 and 120 target schema elements against classical coverage of 50.0%, 52.3% and 57.5%, which removes the manual mapping errors and structural redundancies inherent in classical MDA transformations.

The cost of these guarantees was quantified rather than assumed: 37 to 39 bytes per row, or between 28.9% and 54.2% of the baseline row width, plus a doubling of the primary index. Reporting both sides makes the rules actionable, since it lets an engineer decide per table whether the guarantee is worth its price. The formalized rules provide a practical framework for software engineers and database architects, and apply directly to service-oriented web systems, e-commerce platforms, distributed microservice architectures, and API-driven enterprise applications that require scalable stateless interaction in cloud-native environments.

Future work should automate the methodology in full. One direction is a dedicated tool or an extension for existing Computer-Aided Software Engineering (CASE) environments that parses UML models and generates web-constrained SQL code. A second is to replace UUIDv4 with a time-ordered identifier and to measure the resulting index behavior on a production workload, which would also supply the empirical validation the present study does not claim. A third is adapting these web-specific constraints to the automated generation of NoSQL databases for hybrid cloud infrastructures.

# 7. REFERENCES

[1] Natek, H., Srai, A., and Guerouate, F. 2023. Model-driven architecture approach for SQL generation using Acceleo: a case study on MySQL database. International Journal of Engineering Trends and Technology 71, 10 (Oct. 2023), 20-28. DOI: https://doi.org/10.14445/22315381/IJETT-V71I10P203

[2] Elotmani, F., Esbai, R., and Atounti, M. 2023. Automating the creation of graph-based NoSQL databases in the context of Big Data. In Proceedings of the 2023 International Conference on Advanced Communication Technologies and Networking (CommNet), Rabat, Morocco, 1-6. DOI: https://doi.org/10.1109/CommNet60167.2023.10365296

[3] Abdelhedi, F., Ait Brahim, A., Rajhi, H., Tighilt Ferhat, R., and Zurfluh, G. 2021. Automatic extraction of a document-oriented NoSQL schema. In Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021), vol. 1, 192-199. DOI: https://doi.org/10.5220/0010433501920199

[4] Mali, J., Atigui, F., Azough, A., and Travers, N. 2020. ModelDrivenGuide: an approach for implementing NoSQL schemas. In Database and Expert Systems Applications, 31st International Conference (DEXA 2020), LNCS 12391, Springer, 141-151. DOI: https://doi.org/10.1007/978-3-030-59003-1_9

[5] Abdelhedi, F., Ait Brahim, A., Tighilt Ferhat, R., and Zurfluh, G. 2020. Discovering of a conceptual model from a NoSQL database. In Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020), vol. 1, 61-72. DOI: https://doi.org/10.5220/0009796100610072

[6] Belkadi, F. Z., and Esbai, R. 2021. A model-driven engineering: from relational database to document-oriented database in Big Data context. In Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021), 653-659. DOI: https://doi.org/10.5220/0010604906530659

[7] Paolone, G., Marinelli, M., Paesani, R., and Di Felice, P. 2020. Automatic code generation of MVC web applications. Computers 9, 3 (Jul. 2020), 56. DOI: https://doi.org/10.3390/computers9030056

[8] Vera-Olivera, H., Guo, R., Huacarpuma, R. C., Da Silva, A. P. B., Mariano, A. M., and Holanda, M. 2021. Data modeling and NoSQL databases: a systematic mapping review. ACM Computing Surveys 54, 6 (Jul. 2021), Article 116, 1-26. DOI: https://doi.org/10.1145/3457608

[9] Srai, A., Guerouate, F., and Drissi Lahsini, H. 2021. The integration of the MDA approach in document-oriented NoSQL databases, the case of MongoDB. International Journal of Engineering and Advanced Technology 10, 3 (Feb. 2021), 115-122. DOI: https://doi.org/10.35940/ijeat.C2235.0210321

[10] Sebastian, G., Gallud, J. A., and Tesoriero, R. 2020. Code generation using model driven architecture: a systematic mapping study. Journal of Computer Languages 56, 100935. DOI: https://doi.org/10.1016/j.cola.2019.100935

[11] Hanine, M., Lachgar, M., Elmahfoudi, S., and Boutkhoum, O. 2021. MDA approach for designing and developing data warehouses: a systematic review and proposal. International Journal of Online and Biomedical Engineering 17, 10 (Oct. 2021), 99-110. DOI: https://doi.org/10.3991/ijoe.v17i10.24667

[12] Khan, W., Kumar, T., Zhang, C., Raj, K., Roy, A. M., and Luo, B. 2023. SQL and NoSQL database software architecture performance analysis and assessments: a systematic literature review. Big Data and Cognitive Computing 7, 2, 97. DOI: https://doi.org/10.3390/bdcc7020097

[13] Habri, M. A., Esbai, R., and Lamlili El Mazoui Nadori, Y. 2023. Transforming the business process diagram into a class diagram by model-driven architecture. Indonesian Journal of Electrical Engineering and Computer Science 29, 2 (Feb. 2023), 845-851. DOI: https://doi.org/10.11591/ijeecs.v29.i2.pp845-851

[14] Verbruggen, C., and Snoeck, M. 2023. Practitioners' experiences with model-driven engineering: a meta-review. Software and Systems Modeling 22, 1 (Feb. 2023), 111-129. DOI: https://doi.org/10.1007/s10270-022-01020-1

[15] Samea, F., Azam, F., Rashid, M., Anwar, M. W., Butt, W. H., and Muzaffar, A. W. 2020. A model-driven framework for data-driven applications in serverless cloud computing. PLoS ONE 15, 8, e0237317. DOI: https://doi.org/10.1371/journal.pone.0237317

[16] Scherzinger, S., and Sidortschuck, S. 2020. An empirical study on the design and evolution of NoSQL database schemas. In Conceptual Modeling, 39th International Conference (ER 2020), LNCS 12400, Springer, 441-455. DOI: https://doi.org/10.1007/978-3-030-62522-1_33

[17] Nasiri, S., Rhazali, Y., Lahmer, M., and Chenfour, N. 2020. Towards a generation of class diagram from user stories in agile methods. Procedia Computer Science 170, 837-844. DOI: https://doi.org/10.1016/j.procs.2020.03.148

[18] Abouzahra, A., Sabraoui, A., and Afdel, K. 2020. Model composition in model driven engineering: a systematic literature review. Information and Software Technology 125, 106316. DOI: https://doi.org/10.1016/j.infsof.2020.106316

[19] Jin, K., and Lano, K. 2021. Generation of test cases from UML diagrams: a systematic literature review. In Proceedings of the 2021 International Symposium on Electronic Commerce and Information-Based Management (ECIM), 125-131. DOI: https://doi.org/10.1145/3452383.3452408.

[20] David, I., Aslam, K., Faridmoayer, S., Malavolta, I., Syriani, E., and Lago, P. 2021. Collaborative model-driven software engineering: a systematic update. In Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2021), 121-132. DOI: https://doi.org/10.1109/MODELS50736.2021.00035