A Formal Diagnostic Framework for Graph Repair Systems

Journal Title
XX(X):1–??

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Tung-Wei Lin[1,3], Gabe Fierro[2], Han Li[3], Tianzhen Hong[3], Pierluigi Nuzzo[1], Alberto Samgiovanni-Vincentelli[1]

Abstract

Graph-based data representations excel at capturing relations among entities, e.g., for semantic web, artificial intelligence, or database applications, and their utility depends critically on their quality, usually enforced via *graph schemas*. These schemas are often violated, e.g., during data ingestion, calling for repair actions to achieve compliance. This paper presents a formal diagnostic framework for graph repair methods. Current methods often lack rigor and generality, since they rely on *ad hoc* test datasets. Our method starts, instead, from a compliant graph and uses an abstract rewriting system to systematically introduce schema violations while ensuring provable coverage of the constraints in the schema. We apply the framework to several repair methods, including those based on large language models, and assess their performance across multiple metrics. Results show that our framework effectively differentiates the performance of different methods across various constraints, highlighting the importance of fine-grained, systematic tools for graph repair.

Keywords

Graph Repair, Data Cleaning, Graph Schema Languages

1 Introduction

Graphs are structured representations of entities and their relationships, widely used to encode domain knowledge in applications such as smart building management[1], genome clustering[2], and large-scale knowledge bases such as Wikipedia[3]. They also support emerging use cases in automatic software configuration and remote data access[4,5].

Graphs are populated through manual curation[6], automated conversion from existing databases, and extraction from unstructured text[7]. Their correctness and completeness are critical, as downstream tasks such as query generation[8] and question answering[9] depend on accurate facts. Large resources like Wikidata[3] and YAGO[10] link real-world entities to attributes. Cyber-physical ontologies such as Brick[1] and RealEstateCore[11] enable digital twins and automated control. Reference ontologies like QUDT[12] document physical constants and engineering units. As structured, factual, and symbolic repositories, graphs require rigorous validation.

Schema languages such as SHACL[13], ShEx[14], and PG-Schema[15] have proved to be effective in specifying and validating constraints over graphs. When a violation is detected, the invalid graph must be repaired. This is often a non-trivial task, since multiple repair strategies may exist. For example, Fig. 1 shows an academic publication schedule. The schema enforces temporal ordering with the well-established time ontology[16]: if node u has a `before` edge to node v, then the `end` time of u must be earlier than the `start` time of v. `submission phase1` adheres to this constraint,

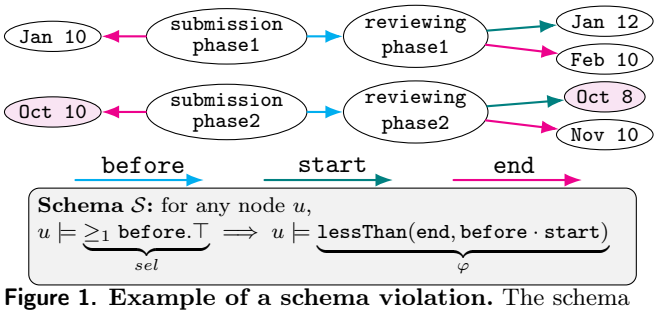


Figure 1. Example of a schema violation. The schema requires that a paper submission phase end before reviewing phase start. `submission phase2` violates this constraint. The schema syntax is explained in Sec. 4.

where `reviewing phase1` starts after `submission phase1` ends. `submission phase2`, however, violates the schema because it ends on `Oct 10` while `reviewing phase2` starts on `Oct 8`. There are many valid repairs, including the following two groups of options to satisfy the schema:

(Op1) Delay `reviewing phase2` to start any time after `Oct 10`.

(Op2) Shorten `submission phase2` to end before `Oct 8`.

[1]University of California, Berkeley, USA
[2]Colorado School of Mines, USA
[3]Lawrence Berkeley National Laboratory, Berkeley, USA

A variety of repair methods have been proposed, including symbolic reasoning[17], domain-specific heuristics with human input[4,18–20], learning from the edit history[21–23], and large language model (LLM) prompting[24,25]. However, evaluating and diagnosing such systems remains difficult. Existing test datasets tend to be limited in size (e.g., SHACL test suite[26]), scoped to just a few violation types[20,25], or tied to specific histories (e.g., Wikidata). This makes it difficult to compare repair systems, characterize the kind of repairs that are supported by different repair systems (e.g., some repair system can not reuse existing nodes in the graph), or analyze scalability.

In this paper, we introduce a formal diagnostic framework for Resource Description Framework (RDF) graph[27] and property graph[15] repair methods. Starting with a schema and a valid graph, our approach generates test datasets by injecting constraint violations into the valid graph, providing a coverage guarantee that every constraint in the schema is individually tested. Because the schema encapsulates domain logic, this exhaustive coverage ensures the practical relevance of test datasets. In addition, the dynamic injection of violations given any schema and graph allows the construction of novel test datasets with private schema and graphs, reducing the risk of data contamination in evaluating LLM methods[28]. Achieving coverage-guaranteed violation injection is non-trivial, as it requires a formalized mechanism to systematically negate arbitrary constraints within a schema to provide proof for coverage. Consequently, we contribute a diagnostic framework that enables the rigorous and fine-grained analysis of graph repair systems.

We validate the proposed framework on several repair systems, including a symbolic method, a supervised learning-based method trained on edit history, and an LLM-based system. Our contributions can be summarized as follows:

- A formalization of *violation expressions* for the systematic and controlled generation of schema violations.
- A dataset generation method enabling coverage-controlled test datasets for graph repair.
- A comparative evaluation of classical and LLM-based repair methods on three real-world graphs, illustrating the framework's diagnostic capabilities.

To the best of our knowledge, this is the first formal diagnostic framework enabling precise characterization of graph repair systems by the types of repairs they can perform and their accuracy.

The remainder of this paper is organized as follows. Sec. 2 reviews the related work. Sec. 3 outlines the evaluation framework while Sec. 4 introduces the preliminaries. Sec. 5 formalizes the violation expressions. Sec. 6 describes the evaluated repair systems. Sec. 7 defines the evaluation metrics, and Sec. 8 reports the results for all the repair systems. Finally, Sec. 9 concludes the paper.

2 Related Work

Graphs are essential for various applications, but costly to create and maintain. While automatically generating graphs from unstructured text is an active research area[7], a graph generated from text would still require post-processing and validation to ensure compliance with schemas[29]. An alternative approach to achieve compliance is knowledge graph completion, which aims to predict missing knowledge rather than repair violations[30–32]. This task is distinct from graph repair due to the absence of a schema.

Several languages have been proposed for expressing and validating constraints over graphs, including SHACL[13], ShEx[14], and PG-Schema[15]. SHACL and ShEx are designed for RDF graphs and PG-Schema is for property graphs. While syntactically different and designed for different data models, recent work[33] shows that these languages share a common abstract semantics. Our work leverages this foundation to define violation expressions in a language-agnostic way. For creating graph schemas, research has focused on mining methods from existing graphs[34–37].

Finally, to repair schema violations, we categorize repair systems into four paradigms. (i) **Human-in-the-loop:** Schimatos[18] proposes a graphical user interface to assist manual repair, BuildingMOTIF[19] provides repair templates for users, and Pachera et al.[20] assign independent violations to multiple users to parallelize repairs, Fan et al.[23] propagate user-confirmed repairs to deduce repairs. (ii) **Logic based:** Ahmetaj et al.[17] propose an automated solution by encoding the repair procedure into an answer set program. (iii) **Learning based:** Pellissier et al.[21] use rule mining on repair histories and, later, employ neural networks to predict repairs[22]. Both of these approaches require an edit history for training. (iv) **LLM based:** Arnaout et al.[24] and Terdalkar et al.[25] prompt LLMs for repairs. While these works propose repair methods, we focus on the evaluation of repair systems.

.

3 Diagnostic Framework for Graph Repair

Building on the abstraction of SHACL, ShEx, and PG-Schema identified in prior work[33], we develop a language-agnostic framework for generating schema violations.

In our framework, graphs are represented in a common model capturing both RDF and property graphs. A **schema** is a set of constraints of the form (sel, φ), where both sel and φ are unary logical expressions over nodes, called **shapes**. The **selector** sel identifies the set of nodes to which the constraint applies, and those nodes must conform to the shape φ. Formally, a graph $\mathcal{G}$ is **valid** with respect to a schema

Corresponding author:
Tung-Wei Lin, University of California, Berkeley, USA.
Email: twlin@berkeley.edu

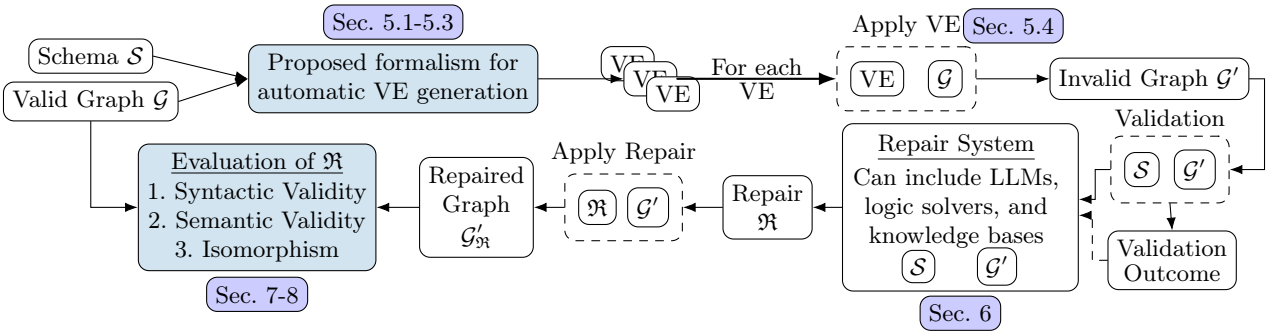


Figure 2. The proposed framework. The shaded blocks are our main contributions; tags specify the section where they are introduced.

$\mathcal{S}$ if, for every $(sel, \varphi) \in \mathcal{S}$ and node u:

$$\mathcal{S}, \mathcal{G}, u \models sel \implies \mathcal{S}, \mathcal{G}, u \models \varphi.$$

This abstract syntax supports predicates, quantifiers, logical connectives, and path expressions, enabling reasoning that is independent of any specific language (e.g., SHACL, ShEx, or PG-Schema).

The objective of this paper is the inverse of schema validation: given a valid graph $\mathcal{G}$ and a schema $\mathcal{S}$, we aim to systematically construct violations. For a constraint $(sel, \varphi) \in \mathcal{S}$, we seek a sequence of operations, such as adding or removing edges, that produces a graph $\mathcal{G}'$ such that, for at least a node u, we have $u \models sel$ but $u \not\models \varphi$. We express such modifications through **violation expressions** (VEs), defined in the abstract model so that they apply across different schema languages (formal definition in Sec. 5.1).

We use this formulation to design an automated test-case generation methodology. As shown in Fig. 2, the framework takes $\mathcal{S}$ and a valid $\mathcal{G}$, generates VEs, and applies each to a separate copy of $\mathcal{G}$, producing an invalid graph $\mathcal{G}'$. Each $(\mathcal{S}, \mathcal{G}')$ is a test case, enabling the attribution of a repair to a single violation expression. This paper is scoped to these single-violation test cases for diagnostic evaluations. Unlike multi-violation scenarios where repair failures may stem from interactions between violations, this isolation acts as a control, ensuring that performance drop is attributable to the specific constraint violation.

After generating $\mathcal{G}'$, $\mathcal{G}'$ is validated against $\mathcal{S}$ to obtain the validation outcome, which identifies the induced violations. The repair system then uses $\mathcal{G}'$, $\mathcal{S}$, and optionally, the validation outcome to produce a repair $\mathfrak{R}$, formalized as a set of graph operations. Applying $\mathfrak{R}$ to $\mathcal{G}'$ yields a repaired graph $\mathcal{G}'_{\mathfrak{R}}$. To assess the quality of $\mathfrak{R}$, we employ tiered metrics, ranging from basic syntactic correctness to the strictest criterion: whether $\mathfrak{R}$ restores the original graph $\mathcal{G}$. The definitions for these metrics are detailed in Sec. 7.

In the remainder of the paper we instantiate the abstract representation of the schema using the SHACL language, providing the relevant syntax and semantics for path expressions, node tests, and complex shape constructs, following the formalizations in [17,33,38,39]. All definitions carry over unchanged to other schema languages such as ShEx and PG-Schema by replacing the instantiation of shapes and path expressions.

4 Preliminaries

Definition 1. Common Graph. Let $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{K}$ be countable sets of nodes, predicates, and keys, respectively, and $\mathcal{V}$ be a countable set of values (e.g., the integer 42). A common graph is a pair $\mathcal{G} = (\mathcal{E}, \rho)$ where:

- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{P} \times \mathcal{N}$ is the finite set of edges (labeled with predicates, e.g., `before` in Fig. 1).
- $\rho : \mathcal{N} \times \mathcal{K} \rightharpoonup \mathcal{V}$ is a finite-domain partial function mapping node–key pairs to values.

Following the convention in [33], we adopt the unified view where *node* refers to elements in $\mathcal{N} \cup \mathcal{V}$ and *edge* refers to $\mathcal{P} \cup \mathcal{K}$. This is used for concise presentation of path expressions composed of elements in $\mathcal{P} \cup \mathcal{K}$ that apply to elements in $\mathcal{N} \cup \mathcal{V}$.

Definition 2. Atomic Graph Operations. Let $\mathcal{G} = (\mathcal{E}, \rho)$ be a common graph. The set of atomic operations is:

1. Edge Operations
   - $+e(u, p, v)$: Add the edge (u, p, v) to $\mathcal{E}$.
   - $-e(u, p, v)$: Remove the edge (u, p, v) from $\mathcal{E}$.
2. Attribute Operations
   - $+a(u, k, val)$: Set $\rho(u, k) = val$.
   - $-a(u, k)$: Remove the mapping for (u, k) from ρ.

More complex actions can be obtained by composing these atomic operations. A replace operation, for instance, simultaneously removes an existing element and adds a new one, indicated by the $\cdot$ symbol. We define a shorthand for replacing the target of an edge or the value of an attribute: $rep(u, q, v, v')$ denotes:

$$\begin{cases} -e(u, q, v) \cdot +e(u, q, v') & \text{if } q \text{ is a predicate } (q \in \mathcal{P}), \\ -a(u, q) \cdot +a(u, q, v') & \text{if } q \text{ is a key } (q \in \mathcal{K}). \end{cases}$$

Definition 3. Path Expressions. SHACL uses path expressions to navigate the graph. A path expression π is built recursively from predicates (p) and the following path operators, inverse (π^-), sequence $(\pi \cdot \pi')$, alternative $(\pi \cup \pi')$, zero-or-more repetitions (π^*), and zero-or-one $(\pi?)$. The grammar is as follows:

$$\pi ::= p \mid \pi^- \mid \pi \cdot \pi' \mid \pi \cup \pi' \mid \pi^* \mid \pi?,$$

where $p \in \mathcal{P} \cup \mathcal{K}$. The evaluation of π over a graph $\mathcal{G}$, denoted $[\![\pi]\!]^{\mathcal{G}}$, yields the set of all endpoint pairs (u, v) connected by paths matching the pattern. The formal semantics are given in Table 1. We omit the superscript $\mathcal{G}$ when it is clear from the context. For a given source node u, we define its set of π**-targets** as all the nodes connected from u via π: $[\![\pi]\!]^{\mathcal{G}}(u) = \{\, v \mid (u, v) \in [\![\pi]\!]^{\mathcal{G}} \,\}$. For example, in Fig. 1, $[\![$`before·start`$]\!]$(`submission phase1`)=$\{$`Jan 12`$\}$.

While a path expression is an abstract pattern, a concrete instance of a path expression in $\mathcal{G}$ is a trace τ, represented by a sequence of alternating nodes and edges:

$$\tau = \langle u_0, p_1, u_1, \ldots, p_k, u_k \rangle \quad (k \geq 0)$$

such that each $(u_i, p_{i+1}, u_{i+1}) \in \mathcal{G}$. Each triple (u_i, p_{i+1}, u_{i+1}) corresponds to a single edge traversal and is called a hop. For a non-zero-length trace ($k \geq 1$), the final hop is the triple (u_{k-1}, p_k, u_k). We denote the start and end nodes of a trace τ as $src(\tau) = u_0$ and $tgt(\tau) = u_k$. The set of all traces in $\mathcal{G}$ is denoted $\Gamma(\mathcal{G})$. A trace τ is said to **match** an expression π (written $\tau \models \pi$) if its endpoints $(src(\tau), tgt(\tau))$ are in $[\![\pi]\!]^{\mathcal{G}}$.

For example, consider the following trace in Fig. 1,

$$\tau_1 = \langle \texttt{submission phase2,before,} \\ \texttt{reviewing phase2,start,Oct 8} \rangle.$$

$\langle$`reviewing phase2,start,Oct 8`$\rangle$ is the final hop of τ_1. Since $(src(\tau_1), tgt(\tau_1)) \in [\![$`before·start`$]\!]$, τ_1 matches the path expression `before·start`.

Then, we define $\Gamma(u, \pi, v)$ as the set of all traces that match π and connect source u to target v:

$$\Gamma(u, \pi, v) = \{\tau \in \Gamma(\mathcal{G}) \mid \tau \models \pi \text{ and } src(\tau) = u \text{ and } tgt(\tau) = v\}.$$

Finally, to filter an arbitrary set of traces T for those ending at a specific node u, we define the target-restricted trace set:

$$T|_u = \{\tau \in T \mid tgt(\tau) = u\}.$$

Definition 4. Node Tests. SHACL supports constraints over nodes, e.g., for checking values or class membership. Following the literature[33], we abstract these constraints as a set Ω of **node tests**. For each SHACL keyword t and node or value y, $\omega_{t,y} \in \Omega$. A node test, test$(\omega_{t,y})$, is a unary predicate. i.e., test$(\omega_{t,y}) : (\mathcal{N} \cup \mathcal{V}) \to \{\top, \bot\}$*. Three cases we use throughout the paper are:

1. `sh:hasValue`, where test$(\omega_{\texttt{sh:hasValue},y})$ is true when the entity under evaluation equals y.
2. `sh:class`, where test$(\omega_{\texttt{sh:class},y})$ is true when the entity under evaluation is an instance of y or subclasses of y.
3. `sh:minLength`, where test$(\omega_{\texttt{sh:minLength},y})$ is true when the entity under evaluation is a string of length at least y.

Table 1. Evaluation of path expressions.

π	$[\![\pi]\!]^{\mathcal{G}} \subseteq (\mathcal{N} \cup \mathcal{V}) \times (\mathcal{N} \cup \mathcal{V})$
p	$\{(u, v) \mid (u, p, v) \in \mathcal{G}\}$
π^-	$\{(u, v) \mid (v, u) \in [\![\pi]\!]^{\mathcal{G}}\}$
$\pi \cdot \pi'$	$\{(u, v) \mid \exists w : (u, w) \in [\![\pi]\!]^{\mathcal{G}} \wedge (w, v) \in [\![\pi']\!]^{\mathcal{G}}\}$
$\pi \cup \pi'$	$[\![\pi]\!]^{\mathcal{G}} \cup [\![\pi']\!]^{\mathcal{G}}$
π^*	$\{(u, u) \mid u \in \mathcal{N}\} \cup [\![\pi]\!]^{\mathcal{G}} \cup [\![\pi \cdot \pi]\!]^{\mathcal{G}} \cup \ldots$
$\pi?$	$\{(u, u) \mid u \in \mathcal{N}\} \cup [\![\pi]\!]^{\mathcal{G}}$

We introduce shorthand: "$= y$" for test$(\omega_{\texttt{sh:hasValue},y})$, "$\in y$" for test$(\omega_{\texttt{sh:class},y})$, and "len $\geq y$" for test$(\omega_{\texttt{sh:minLength},y})$.

For example, using our shorthand for `sh:hasValue`, the test "$=$ `Oct 10`" checks for equality with the literal `Oct 10`. Thus, this test evaluates to $\top$ on the value `Oct 10` itself and $\bot$ on any different node in Fig. 1.

Definition 5. SHACL Shapes and Subshapes. Let Σ be a set of **shape names**, disjoint from $\mathcal{N}$, $\mathcal{V}$, $\mathcal{P}$, and $\mathcal{K}$. Each shape name $s \in \Sigma$ is associated with a shape formula $def(s, \mathcal{S})$ in the schema $\mathcal{S}$, following the notation in [39]. Shapes are defined inductively from two kinds of building blocks:

Atomic shapes α: the base cases with no proper subshapes:

$$\begin{aligned} \alpha ::= {} & \top \mid \text{test}(\omega) \mid \text{closed}(Q) \mid \text{eq}(\pi, q) \mid \text{disj}(\pi, q) \\ & \mid \text{lessThan}(\pi, q) \mid \text{lessThanEq}(\pi, q) \mid \text{uniqueLang}(\pi) \\ & \mid \neg\beta \quad (\beta \text{ is one of the above non-negated forms}) \end{aligned}$$

Compound shapes φ: formed from atomic shapes, shape names, and logical or cardinality operators:

$$\begin{aligned} \varphi ::= {} & \alpha \quad \text{(atomic shape)} \\ & \mid s \quad (s \in \Sigma) \\ & \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \neg\varphi \quad \text{(non-atomic negation)} \\ & \mid \forall\pi.\varphi \mid \geq_m \pi.\varphi \mid \leq_n \pi.\varphi, \end{aligned}$$

where $\omega \in \Omega$, $Q \subseteq \mathcal{P} \cup \mathcal{K}$, π, q are path expressions, and $m \in \mathbb{N}^+, n \in \mathbb{N}$. The semantics of $u \models \varphi$ are defined inductively over this syntax in Table 2. In particular, $\geq_n \pi.\varphi$ and $\leq_n \pi.\varphi$ are qualified cardinality constraints. To formalize their semantics, we define the set of **qualified** π**-targets** of a node u with respect to a shape φ as the subset of its π-targets that satisfy φ:

$$[\![\pi, \varphi]\!]^{\mathcal{G}}(u) \;=\; \{\, v \in [\![\pi]\!]^{\mathcal{G}}(u) \mid v \models \varphi \,\}.$$

The semantics of the cardinality constraints can then be expressed using this set as in Table 2.

Given a shape φ, its set of **subshapes**, written Sub(φ), is the set containing φ itself and all shapes

*The keywords covered are `sh:languageIn`, `sh:datatype`, `sh:nodeKind`, `sh:class`, `sh:minExclusive`, `sh:maxExclusive`, `sh:minInclusive`, `sh:maxInclusive`, `sh:minLength`, `sh:maxLength`, `sh:hasValue`, and `sh:pattern` (see Sec. 2 in [39]).

Table 2. Semantics of a SHACL shape φ. $<$ and $\leq$ are the standard orderings on comparable terms (e.g., numbers, strings, dates) and $v \not\sim w$ means v and w are literals with different language tags.

φ	$\mathcal{S}, \mathcal{G}, u \models \varphi$ if:
$\top$	trivially satisfied
s	$\mathcal{S}, \mathcal{G}, u \models def(s, \mathcal{S})$
$\text{test}(\omega_{t,y})$	$u \models \omega_{t,y}$
$\text{closed}(Q)$	$\forall p \in (\mathcal{P} \cup \mathcal{K}) \setminus Q:$ not $\mathcal{S}, \mathcal{G}, u \models_{\geq 1} p.\top$
$\text{eq}(\pi, q)$	$\{v \mid (u,v) \in [\![\pi]\!]^{\mathcal{G}}\} = \{v \mid (u,v) \in [\![q]\!]^{\mathcal{G}}\}$
$\text{disj}(\pi, q)$	$\{v \mid (u,v) \in [\![\pi]\!]^{\mathcal{G}}\} \cap \{v \mid (u,v) \in [\![q]\!]^{\mathcal{G}}\} = \emptyset$
$\varphi \wedge \varphi'$	$\mathcal{S}, \mathcal{G}, u \models \varphi$ and $\mathcal{S}, \mathcal{G}, u \models \varphi'$
$\varphi \vee \varphi'$	$\mathcal{S}, \mathcal{G}, u \models \varphi$ or $\mathcal{S}, \mathcal{G}, u \models \varphi'$
$\neg\varphi$	not $\mathcal{S}, \mathcal{G}, u \models \varphi$
$\forall\pi.\varphi$	$\forall v((u,v) \in [\![\pi]\!]^{\mathcal{G}} \rightarrow \mathcal{S}, \mathcal{G}, v \models \varphi)$
$\geq_n \pi.\varphi$	$\lvert[\![\pi, \varphi]\!]^{\mathcal{G}}(u)\rvert \geq n$
$\leq_n \pi.\varphi$	$\lvert[\![\pi, \varphi]\!]^{\mathcal{G}}(u)\rvert \leq n$
$lessThan(\pi, q)$	$\forall (u,v) \in [\![\pi]\!]^{\mathcal{G}}, (u,w) \in [\![q]\!]^{\mathcal{G}} : v < w$
$lessThanEq(\pi, q)$	$\forall (u,v) \in [\![\pi]\!]^{\mathcal{G}}, (u,w) \in [\![q]\!]^{\mathcal{G}} : v \leq w$
$uniqueLang(\pi)$	$\forall (u,v), (u,w) \in [\![\pi]\!]^{\mathcal{G}}$ and $v \neq w : v \not\sim w$

occurring inside it. Formally: $\text{Sub}(\varphi)$ is defined as

$$\begin{cases} \{\varphi\} & \varphi \text{ atomic}, \\ \text{Sub}(def(s, \mathcal{S})) & \varphi = s \in \Sigma, \\ \{\varphi\} \cup \text{Sub}(\varphi_1) \cup \text{Sub}(\varphi_2) & \varphi = \varphi_1 \wedge \varphi_2 \text{ or } \varphi_1 \vee \varphi_2, \\ \{\varphi\} \cup \text{Sub}(\psi) & \varphi = \neg\psi \text{ and } \varphi \text{ non-atomic}, \\ \{\varphi\} \cup \text{Sub}(\psi) & \varphi \in \{\forall\pi.\psi, \geq_n \pi.\psi, \leq_n \pi.\psi\}. \end{cases}$$

To illustrate the subshape definition, consider the selector $sel = \geq_1 \texttt{before}.\top$ from Fig. 1. Its set of subshapes is calculated as $\text{Sub}(sel) = \{\geq_1 \texttt{before}.\top\} \cup \text{Sub}(\top) = \{\geq_1 \texttt{before}.\top, \top\}$.

Definition 6. Non-recursive Schema. Let each shape name $s \in \Sigma$ be associated with a shape formula $def(s, \mathcal{S})$. We say that s *refers* to a shape name s' if there exists an integer $k \geq 1$ and a sequence of shape names $s_0, s_1, \ldots, s_k$ such that $s_0 = s$, $s_k = s'$, and s_i occurs in $def(s_{i-1}, \mathcal{S})$ for all $1 \leq i \leq k$. A schema $\mathcal{S}$ is *non-recursive* if no shape name refers to itself, i.e., there is no such sequence with $s_0 = s_k$. Following[33], we consider only non-recursive schemas.

Definition 7. Offending Subgraph. When a graph $\mathcal{G}$ is validated against a schema $\mathcal{S}$ and is invalid, there exists at least one **offended constraint** $(sel, \varphi) \in \mathcal{S}$ with the corresponding **offending focus** u such that $u \models sel$ and $u \not\models \varphi$. To analyze the cause of this violation, we define the **offending subgraph** $\textsf{OffSub}_{\mathcal{G}}(u, (sel, \varphi))$ as the portion of $\mathcal{G}$ **visited** during the evaluation of whether $u \models sel \implies u \models \varphi$ holds according to the SHACL specification[40]: $\textsf{OffSub}_{\mathcal{G}}(u, (sel, \varphi)) = (\mathcal{E}', \rho')$, where $\mathcal{E}' \subseteq \mathcal{E}$ and $\rho' \subseteq \rho$ are the edges and attribute assignments accessed during this evaluation.

For example, let the offending focus u be `submission phase2` in Fig. 1. Since u satisfies the selector $\geq_1 \texttt{before}.\top$, validating the shape $\texttt{lessThan}(\texttt{end}, \texttt{before} \cdot \texttt{start})$ involves visiting two traces from u: $\langle$`submission phase2`, `end`, `Oct 10`$\rangle$ and $\langle$`submission phase2`, `before`, `reviewing phase2`, `start`, `Oct 8`$\rangle$. The offending subgraph is composed of all nodes and edges from these traces, as shown in Fig. 3. Intuitively, every element in OffSub is relevant to repairing the violation. However, OffSub does not necessarily contain all the relevant information – additional parts of the graph (or external knowledge) may also be important.

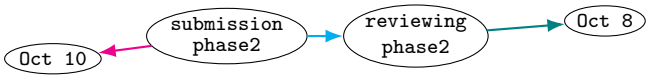


Figure 3. The offending subgraph of the violation in Fig. 1 with the offending focus `submission phase2`.

5 Abstract Rewriting System

To formalize constraint violation injection and derive guarantee that every constraint in the schema is covered, we introduce a formal system. As stated in Def. 5, a shape φ may be (i) atomic, (ii) a reference to another shape s, or (iii) a compound shape built from conjunction ($\varphi \wedge \varphi'$), disjunction ($\varphi \vee \varphi'$), negation ($\neg\varphi$), universal quantification ($\forall\pi.\varphi$), or qualified cardinality constraints ($\geq_n \pi.\varphi$ and $\leq_n \pi.\varphi$).

To unpack complex shapes into simpler components, we use an Abstract Rewriting System (ARS) $\mathcal{A} = (\texttt{Expr}, \rightarrow)$. The system starts with a high-level task—violating a complex shape—and rewrites it into simpler tasks using **violation expressions** `Expr` (Sec. 5.1) and **rewrite rules** $\rightarrow$ (Sec. 5.2). Rewriting starts with an initial violation expression derived from a constraint in the given schema and concludes when the expression is in **normal form**, i.e., no more rewrite rules can be applied. The repeated applications of rewrite rules induce a graph data structure for violation expressions. We define a traversal strategy to collect a complete set of expressions, ensuring coverage of all constraints in the schema (Sec. 5.3). Finally, the normal-form expressions are mapped to graph operations to create violations (Sec. 5.4). $\mathcal{A}$ is designed to be **strongly normalizing**, ensuring any expression reduces to normal form in finitely many steps.

Running Example. Fig. 4a illustrates our running example, a graph modeling academic publications. The graph contains individuals and papers that are `reviewedBy` these individuals. The schema $\mathcal{S}$ specifies that any `Paper` must be `reviewedBy` at least one but at most three `Reviewers`, where a `Reviewer` is defined as an individual who has a `workPhone` or `cellPhone` string at least of length 3. The graph currently satisfies the schema.

5.1 Violation Expressions

The repeated applications of rewrite rules induce a graph data structure for violation expressions, called expansion graph, which represents the space of possible strategies to violate a constraint. For our running example, the expansion graph violating (sel_1, φ_1) in Fig. 4a is shown in Fig. 4c. The nodes of the expansion graph are violation expressions. We now define the components of violation expressions.

A violation expression is constructed from a set of fundamental tasks combined with two task operators:

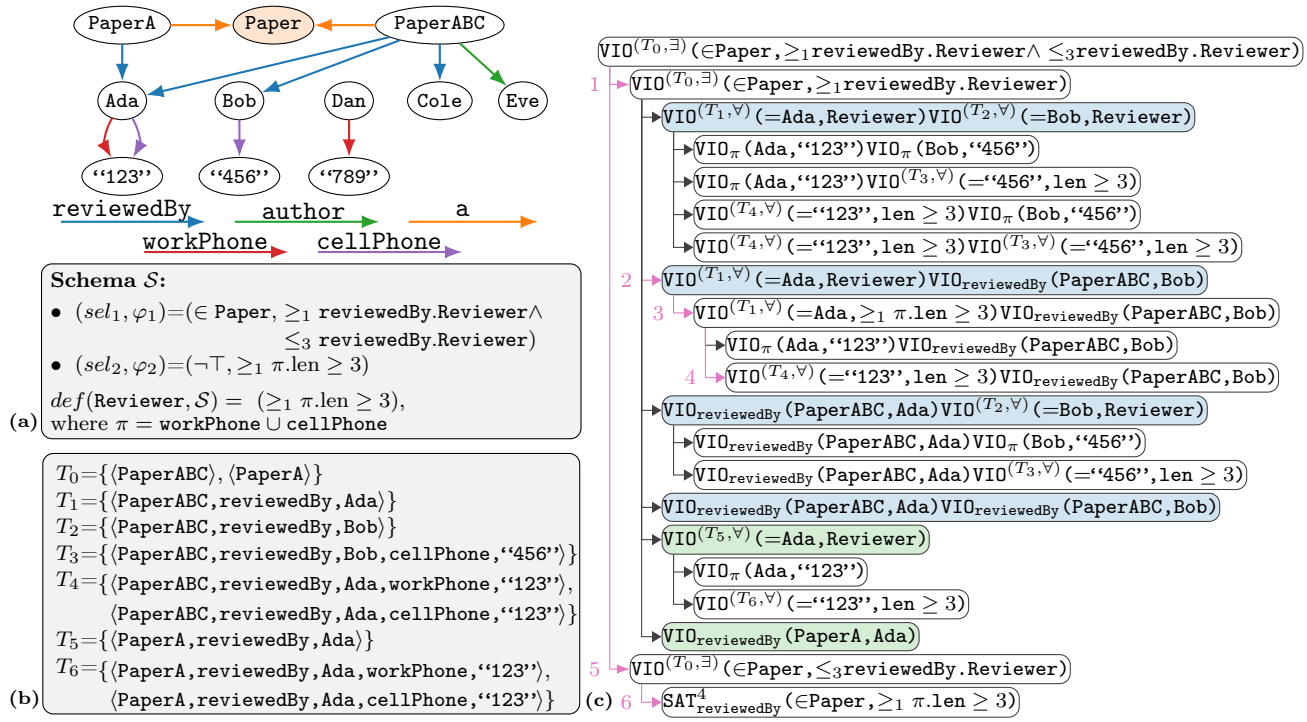


Figure 4. Running example. (a) A graph and schema modeling academic publications. (b) The trace sets used in (c). (c) Expansion graph and DFS from (sel_1, φ_1). In (c), the arrows denote the application of rewrite rules that transform a task into sub-tasks (e.g., the branching at the root represents a rewrite). Some intermediate steps such as shape unfolding Reviewer→ (≥1 π.len ≥ 3) are omitted.

- **Simultaneous** $(\cdot)$: A conjunction requiring that all tasks are achieved.
- **Alternative** $(+)$: A disjunction requiring that one of the tasks is achieved.

There are three fundamental tasks:

- **Shape violation task** VIO(sel, φ): Find a minimal set of graph operations that, when applied to the graph, cause a selected node $u \models sel$ to no longer satisfy φ.
- **Path violation task** VIO$_\pi(u, v)$: Find a minimal set of graph operations that, when applied to the graph, remove all π-matching traces between nodes u and v.
- **Qualified satisfaction task** SAT$^n_\pi(sel, \varphi)$: Find a minimal set of graph operations that, when applied to the graph, result in a selected node $u \models sel$ having n distinct π-targets, each satisfying the shape φ.

A challenge arises when rewriting shapes with path expressions. For example, consider violating (sel_1, φ_1) on the PaperA node in our running example. A subshape of φ_1 requires that PaperA have at least one valid Reviewer (≥1 reviewedBy.Reviewer). As we will see in Sec. 5.2, our ARS will follow the trace matching reviewedBy from PaperA to reach Ada and generate a sub-task: make Ada no longer a valid Reviewer. However, our ARS needs to remember that this sub-task on Ada is generated because of the trace from PaperA. This context is managed by provenance (T, Q), which augments the shape violation task to VIO$^{(T,Q)}(sel, \varphi)$, where:

- **Trace Set** $T \subseteq \Gamma(\mathcal{G})$ records the sequence of edges followed from the initial task's selected nodes to the current sub-task's selected nodes. In our example, our ARS navigates from PaperA to Ada, so the trace set is $\{\langle \text{PaperA}, \text{reviewedBy}, \text{Ada}\rangle\}$. Fig. 4b details the evolution of trace set T_0 into $T_1 \dots T_6$, which correspond to the trace sets used in the nodes in the expansion graph in Fig. 4c.
- **Mode** $Q \in \{\exists, \forall\}$ quantifies the traces that must be "modified" in T. The mechanism for modifying a trace is detailed in Sec. 5.4.1.
  - $Q = \forall$ (Modify All): Requires modifying *all* the traces in T. This is necessary when generating a violation to the "at least" $(\geq_n \pi.\varphi)$ constraints. For example, to make Ada an invalid Reviewer, we must violate the requirement that she has at least one phone string $(\geq_1 \pi.\text{len} \geq 3$, where $\pi = \text{workPhone} \cup \text{cellPhone})$. Since Ada is connected to "123" via both workPhone and cellPhone, our ARS needs to modify both ⟨Ada,workPhone,"123"⟩ and ⟨Ada,cellPhone,"123"⟩ to ensure Ada has no phone.
  - $Q = \exists$ (Modify One): Requires modifying just *one* trace in T. This occurs when generating a violation to the universal quantifier $\forall\pi.\varphi$ constraints, where one counterexample is sufficient.

Intuitively, provenance tells us which traces led to the current sub-task and the scope for modifying those traces. The syntax of the violation expressions in our

ARS is:

$$\texttt{Term} ::= \texttt{VIO}^{(T,Q)}(sel,\varphi) \mid \texttt{VIO}_{\pi}(u,v) \mid \texttt{SAT}^{n}_{\pi}(sel,\varphi) \mid \texttt{Term}\cdot\texttt{Term},$$
$$\texttt{Expr} ::= \texttt{Term} \mid \texttt{Expr} + \texttt{Term}.$$

Formally, the **expansion graph** for a constraint (sel, φ) is a directed acyclic graph defined as follows:

- **Nodes:** Each **node** corresponds to a `Term` in our grammar, a conjunctive set of tasks ($\texttt{T}_1 \cdot \texttt{T}_2 \cdots$) that must be achieved simultaneously. Nodes represent strategies for creating violations.
- **Root:** The **root** node of the graph is the initial violation task $\texttt{VIO}^{(T_0,Q_0)}(sel,\varphi)$. The initial provenance (T_0, Q_0) is set as:
  - $T_0 = \{\langle u\rangle \mid u \models sel\}$: The initial trace set contains a zero-length trace for each node u that matches sel. For our running example, since both `PaperA` and `PaperABC` satisfy the selector sel_1, the initial trace set is $T_0 = \{\langle\texttt{PaperABC}\rangle, \langle\texttt{PaperA}\rangle\}$.
  - $Q_0 = \exists$: The initial mode is $\exists$ by default until rewritten by subsequent rewrite rules.
- **Edges and Branching:** An edge represents the application of a rewrite rule. When the applied rewrite rule yields a disjunctive expression ($\texttt{T}_1 + \texttt{T}_2 + \cdots$), the parent node branches, generating one child for each term. In the running example, the task of violating (sel_1, φ_1) on the `PaperA` node (root in Fig. 4c) branches into two children: one for violating "at least one `Reviewer`" (node 1 in Fig. 4c) and one for violating "at most three `Reviewer`" (node 5 in Fig. 4c) because one can perform either action to make `PaperA` violate (sel_1, φ_1).

Consequently, the **internal nodes** of the graph are expressions that can be further rewritten, while the **leaf nodes** are expressions in normal form. Each unique path from the root to a leaf node represents a complete strategy for violating the constraint (sel, φ).

5.2 Rewrite Rules

We now present the rewrite rules that induce the expansion graph. The expansion graph will allow the exhaustive collection of violation expressions that will be explained in Sec. 5.3.

Rule 1: Shape Reference (s). When a shape reference s is in a task, it is replaced by the shape definition $def(s, \mathcal{S})$. This "unfolds" the reference so that rewriting proceeds on the actual shape formula.

$$\texttt{VIO}^{(T,Q)}(sel,s) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, def(s,\mathcal{S}))$$
$$\texttt{SAT}^{n}_{\pi}(sel,s) \longrightarrow \texttt{SAT}^{n}_{\pi}(sel, def(s,\mathcal{S}))$$

Rule 2: Conjunction ($\wedge$). A node satisfies $\varphi \wedge \varphi'$ if it satisfies both shapes. To violate the conjunction, it is enough to violate either φ or φ'. This choice is represented by the alternative task operator $+$.

$$\texttt{VIO}^{(T,Q)}(sel,\varphi\wedge\varphi') \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\varphi) + \texttt{VIO}^{(T,Q)}(sel,\varphi')$$

Rule 3: Disjunction ($\vee$). A node satisfies $\varphi \vee \varphi'$ if it satisfies at least one shape. To violate the disjunction, a node must fail *both* φ and φ'. The rewrite rule generates this conjunctive task for each node u that satisfies the selector sel. Formally,

$$\texttt{VIO}^{(T,Q)}(sel,\varphi\vee\varphi') \longrightarrow \sum_{u:u\models sel} \big(\textbf{if } u\models\varphi \textbf{ then } \texttt{VIO}^{(T|_u,Q)}(=u,\varphi)\big)\cdot \big(\textbf{if } u\models\varphi' \textbf{ then } \texttt{VIO}^{(T|_u,Q)}(=u,\varphi')\big)$$

The summation $\sum_{u:u\models sel}$ states that a violation can be performed on any node u satisfying the selector. For a chosen u, the provenance is updated using $T|_u$ (the subset of traces in T ending at u), and the $(\cdot)$ task operator creates the conjunctive task. The **if**...**then** condition allows omitting the sub-task for a shape u that is already violating a shape. If a shape (e.g., φ') is not satisfied by any node that matches the selector, then no violation task will be generated for φ' or any of its subshapes, making them *unreachable* in our ARS.

Rule 4: Negation ($\neg$). For a negated shape $\neg\varphi$, we push the negation inward using standard logical equivalences, as follows.

1. De Morgan's Law

$$\texttt{VIO}^{(T,Q)}(sel,\neg(\varphi\wedge\varphi')) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\neg\varphi\vee\neg\varphi')$$
$$\texttt{VIO}^{(T,Q)}(sel,\neg(\varphi\vee\varphi')) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\neg\varphi\wedge\neg\varphi')$$
$$\texttt{VIO}^{(T,Q)}(sel,\neg\neg\varphi) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\varphi)$$

2. Quantifier Dualities

$$\texttt{VIO}^{(T,Q)}(sel,\neg(\forall\pi.\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\geq_1\pi.\neg\varphi)$$
$$\texttt{VIO}^{(T,Q)}(sel,\leq_0\pi.\neg\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\forall\pi.\varphi)$$
$$\texttt{VIO}^{(T,Q)}(sel,\neg(\geq_n\pi.\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\leq_{n-1}\pi.\varphi)$$
$$\texttt{VIO}^{(T,Q)}(sel,\neg(\leq_n\pi.\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\geq_{n+1}\pi.\varphi)$$

3. Shape Unfolding

$$\texttt{VIO}^{(T,Q)}(sel,\neg s) \longrightarrow \texttt{VIO}^{(T,Q)}(sel,\neg def(s,\mathcal{S}))$$

This produces an equivalent form that can then be rewritten by other rules.

Rule 5: Qualified Cardinality ($\geq_n \pi.\varphi$). A node satisfies $\geq_n \pi.\varphi$ if it has at least n π-targets that satisfy φ. To violate this constraint, we must reduce the number of qualified π-targets to less than n. First, we determine the number of π-targets to invalidate. Let m be the current number of qualified π-targets. We must invalidate at least $k = m - n + 1$ of them. In our running example (Fig. 4a), `PaperABC` has $m = 2$ qualified reviewers (`Ada` and `Bob`). To violate $\geq_1$ `reviewedBy.Reviewer`, we must invalidate $k = 2 - 1 + 1 = 2$. For each of the k targets selected, we have two invalidation strategies:

1. **Invalidate the target:** Make the target node violate φ, e.g., make `Ada` no longer a valid `Reviewer`.

2. **Invalidate the trace:** Remove the π-matching trace connecting to the target node, e.g., remove the `reviewedBy` edge between `PaperABC` and `Ada`.

When a rewrite rule follows a trace from a node u to another node v to generate a sub-task, it updates the provenance by **trace extensions**. This operation appends new trace segments to existing traces. For instance, when following the `reviewedBy` edge from `PaperABC` to `Ada`, the initial trace $\langle \texttt{PaperABC} \rangle$ is extended to $\langle \texttt{PaperABC}, \texttt{reviewedBy}, \texttt{Ada} \rangle$. Formally,

$$T'_{u \to v} = \{\tau \circ \tau' \mid \tau \in T|_u \text{ and } \tau' \in \Gamma(u, \pi, v)\},$$

where $\circ$ denotes trace concatenation ($\langle w, p, u \rangle \circ \langle u, q, v \rangle = \langle w, p, u, q, v \rangle$). The full rewrite rule combines the two invalidation strategies:

$$\texttt{VIO}^{(T,Q)}(sel, \geq_n \pi.\varphi) \longrightarrow \sum_{u:u \models sel} \sum_{\substack{\Delta \subseteq [\![\pi,\varphi]\!](u) \\ |\Delta|=k}} \prod_{v \in \Delta} \left(\texttt{VIO}^{(T'_{u\to v}, \forall)}(= v, \varphi) + \texttt{VIO}_\pi(u, v) \right).$$

The outer summation selects a node u satisfying sel (e.g., `PaperABC`). The inner summation chooses a subset Δ of k nodes to invalidate (e.g., `Ada` and `Bob`). The product states that one of the invalidation strategies must be applied to each node in Δ. The $+$ states the choice between invalidating the target or invalidating the trace. The mode for the "invalidate target" sub-task is $Q = \forall$ since all traces from u to v must be modified. In our running example, since we must invalidate $k = 2$ targets and have two choices for each, we generate $2^2 = 4$ conjunctive sub-tasks, as shown by the blue nodes in Fig. 4c.

Rule 6: Qualified Cardinality ($\leq_n \pi.\varphi$). A node u satisfies $\leq_n \pi.\varphi$ if it has at most n π-targets that satisfy φ. To violate this constraint, we must increase the number of qualified π-targets to more than n. We directly apply the qualified satisfaction task $\texttt{SAT}_\pi^{n+1}$:

$$\texttt{VIO}^{(T,Q)}(sel, \leq_n \pi.\varphi) \longrightarrow \texttt{SAT}_\pi^{n+1}(sel, \varphi).$$

Violating a $\leq_n \pi.\varphi$ constraint always yields a terminal `SAT` task. Since the objective of the `SAT` task is to satisfy φ as an indivisible unit, our ARS does not decompose φ further, making the subshapes of φ *unreachable*.

Rule 7: Universal Quantifier ($\forall\pi.\varphi$). A node satisfies $\forall\pi.\varphi$ if all its π-targets satisfy φ. To violate this constraint, at least one π-target must fail φ. We have two invalidation strategies:

1. **Invalidate an existing target:** Select a π-target v and generate a sub-task to make v violate φ.
2. **Introduce a new violating target:** Add a new π-target that satisfies $\neg\varphi$.

The full rewrite rule combines the two invalidation strategies:

$$\texttt{VIO}^{(T,Q)}(sel, \forall\pi.\varphi) \longrightarrow \sum_{u:u \models sel} \sum_{v \in [\![\pi]\!](u)} \texttt{VIO}^{(T'_{u\to v}, \exists)}(= v, \varphi) + \texttt{SAT}_\pi^1(sel, \neg\varphi).$$

The rule consists of two parts connected by $+$, representing the two strategies. The first part invalidates an existing target. The summations select a node u satisfying sel and chooses a π-target v. The sub-task uses trace extension $T'_{u\to v}$ and mode $Q = \exists$, as violating a single path is sufficient. The second part introduces a new violating target by creating a new π-target that satisfies $\neg\varphi$. If no node satisfying the selector has π-targets, the first part of the rule vanishes. The only option is the `SAT` task, which is terminal. In this case, the subshapes of φ are *unreachable*.

In Rules 3, 6, and 7, we observed situations where subshapes can never appear in a `VIO` task. This occurs when a rewrite rule fails to produce `VIO` sub-tasks or yields a terminal `SAT` task. We formalize this with the notion of **shape reachability** in Def. 8, which captures when a subshape can appear in a `VIO` task during the rewriting process.

Definition 8. Shape Reachability. A subshape ψ of a shape φ is **reachable** from an initial task $\texttt{VIO}^{(T_0,Q_0)}(sel, \varphi)$ if $\psi = \varphi$ or if there is a sequence of rewrite steps

$$\texttt{VIO}^{(T_0,Q_0)}(sel, \varphi) \longrightarrow \cdots \longrightarrow \texttt{Expr},$$

where `Expr` contains a sub-task $\texttt{VIO}^{(T,Q)}(sel', \varphi')$ and ψ is reachable from $\texttt{VIO}(sel', \varphi')$.

A subshape ψ of a shape φ is **unreachable** if it is not reachable. This occurs if, for every sub-task $\texttt{VIO}^{(T,Q)}(sel', \varphi')$ generated during rewriting where ψ is a subshape of φ', one of the following blocking conditions is met:

- **Non-satisfied Disjunct:** $\varphi' = \varphi'' \vee \varphi'''$, ψ is a subshape of φ'', and no node satisfies both sel' and φ''.
- **At-most Cardinality:** $\varphi' = \leq_n \pi.\varphi''$ and ψ is a subshape of φ''.
- **Universal Quantifier with No Targets:** $\varphi' = \forall\pi.\varphi''$, ψ is a subshape of φ'', and no node satisfying sel' has a π-target.

For an unreachable shape ψ, our ARS will never generate a `VIO` task with ψ in it.

Having established shape reachability, we now turn to a fundamental property of our rewriting system: *termination*. The following theorem shows that, under mild assumptions, the rewriting process always terminates, i.e., $\mathcal{A}$ is strongly normalizing.

Theorem 1. Strong Normalization of $\mathcal{A}$. If $\mathcal{G}$ and $\mathcal{S}$ are finite and $\mathcal{S}$ is non-recursive, then the rewriting system $\mathcal{A}$ is strongly normalizing.

Proof. See App. A for the proof. □

5.3 Subshape Collection and De-duplication

To generate a coverage guaranteed test dataset, we must systematically navigate the expansion graph to exhaustively collect violation expressions. Furthermore, normalized violation expressions will be mapped to graph operations that create violations in Sec. 5.4, which

are applied to separate copies of the valid graph $\mathcal{G}$ to isolate the effect of violation expressions.

From Def. 8, the reachable subshapes are those that can appear in a shape violation task during rewriting. Our goal is to collect all the reachable subshapes of each constraint by traversing its expansion graph and recording the subshapes within violation tasks.

Since $\mathcal{S}$ is non-recursive, we process constraints in topological order, starting from shapes not referred to by others. For each (sel, φ), we build its expansion graph rooted at $\texttt{VIO}^{(T_0,Q_0)}(sel, \varphi)$ and run a depth-first search (DFS) to enumerate all the reachable subshapes $\psi \in \text{Sub}(\varphi)$. A subshape ψ is **covered** by a rewrite path if that path contains a $\texttt{VIO}^{(T,Q)}(sel', \psi)$ term.

During DFS, when a choice (+) is encountered, one branch is selected at random to form a path from root to leaf, yielding candidate violation expressions. Finally, we perform de-duplication: after all constraints are processed, rewrites within or across constraints may yield the same normal form expression. Since these represent identical graph operations, only one is retained.

Running Example. In Fig. 4a, since (sel_1, φ_1) is not referred to by other shapes and `Reviewer` is associated with the shape formula φ_2, DFS starts from $\texttt{VIO}^{(T_0,\exists)}(sel_1, \varphi_1)$. For the purpose of illustration, we first identify the set of five reachable subshapes $\text{Sub}(\varphi_1)$:

(a) $\geq_1$ `reviewedBy.Reviewer`$\wedge \leq_3$ `reviewedBy.Reviewer`

(b) $\geq_1$ `reviewedBy.Reviewer`

(c) $\leq_3$ `reviewedBy.Reviewer`

(d) $\geq_1 \pi$.`len` ≥ 3 (reachable via (b))

(e) `len` ≥ 3 (reachable via (b))

The DFS continues until all five reachable subshapes in $\text{Sub}(\varphi_1)$ are covered. To illustrate how the termination condition works, we examine two potential outcomes of DFS. In the following, the numbers 1-6 refer to the nodes in the expansion graph in Fig. 4c.

- $\mathcal{P}_1 = \{(1, 2, 3, 4)\}$: The DFS is incomplete because shape (c) $\leq_3$ `reviewedBy.Reviewer` is not covered.
- $\mathcal{P}_2 = \{(1, 2, 3, 4), (5, 6)\}$: The DFS terminates because the addition of path $(5, 6)$ covers the missing shape (c).

We observe that the traversal visits each subshape at most once. After traversal, de-duplication is a pass over the collected set. Therefore, the total run time scales linearly with the number of subshapes:

Proposition 1. Time Complexity. For a finite and non-recursive schema $\mathcal{S}$ with n_s subshapes, the subshape collection and de-duplication algorithm runs in $O(n_s)$.

Proof. See App. B for the proof. □

Table 3. Graph operations for $\texttt{VIO}^{(T,Q)}(sel, \varphi)$, given a node $u \models sel$ and the final hop (w, p, u) of a trace. u' is generated by editing u (e.g., adding, removing, transposing characters, or combinations thereof) to violate φ.

φ	Graph Operations
$= y$	$rep(w, p, u, u')$, where $u' \neq u$
$\neg = y$	$rep(w, p, u, y)$
len $\geq y$	$rep(w, p, u, u')$, where $len(u') < y$
$\neg$len $\geq y$	$rep(w, p, u, u')$, where $len(u') \geq y$

5.4 Apply Normalized Expressions

Once normalized, de-duplicated violation expressions have been collected, we describe how each of the three fundamental tasks is mapped to graph operations, which are applied to separate copies of the valid graph $\mathcal{G}$, and our framework's coverage guarantee.

5.4.1 Shape violation $\texttt{VIO}^{(T,Q)}(sel, \varphi)$ The goal of a normalized $\texttt{VIO}^{(T,Q)}(sel, \varphi)$ task is to make a selected node $u \models sel$ violate the atomic shape φ. The required graph operations depend on whether φ is negated:

- If φ is a **non-negated** atomic shape, the goal is to invalidate φ.
- If φ is a **negated** atomic shape of the form $\neg\beta$, the goal is to perform graph operations such that β is true.

Table 3 lists the graph operations for a subset of atomic shapes; the full list is in App. C.

Once the graph operations are determined, the provenance (T, Q) guides where to apply them. Since all the traces in T end at the selected node u by construction, we modify the final hop of these traces according to the mode Q. If $Q = \exists$, we modify only one trace in T; if $Q = \forall$, we modify all traces in T.

Running Example. Assume after DFS and de-duplication on the expansion graph in Fig. 4c we have collected the shape violation task $\texttt{VIO}^{(T_4,\forall)}$(=‘‘123’’, `len` ≥ 3), where

$T_4 = \{\langle$`PaperABC,reviewedBy,Ada,workPhone,‘‘123’’`$\rangle$, $\langle$`PaperABC,reviewedBy,Ada,cellPhone,‘‘123’’`$\rangle\}$.

The selected node u is ‘‘123’’ according to the selector =‘‘123’’. The shape `len` ≥ 3 is non-negated, so we must make it false. As defined in Table 3, we can then replace ‘‘123’’ with a new value that has a length less than three (e.g., ‘‘23’’). The traces in T_4 identify two final hops, the edge traversal corresponding to `workPhone` and `cellPhone`. Since the mode is $Q = \forall$, a replace operation is constructed for each hop, resulting in two operations:

- rep(`Ada,workPhone,‘‘123’’,‘‘23’’`)
- rep(`Ada,cellPhone,‘‘123’’,‘‘23’’`).

5.4.2 Path violation $\texttt{VIO}_\pi(u, v)$ The goal of a $\texttt{VIO}_\pi(u, v)$ task is to remove all π-matching traces between u and v. The process is a recursion on the structure of π. For a trace τ that matches π, the algorithm finds and removes

an essential edge in τ to break it. The recursion unfolds as follows:

- **Base case, predicate** p**:** a trace τ matches p if (u, p, v) is present. To violate this, we remove the edge: $-e(u, p, v)$ if $p \in \mathcal{E}$, or $-a(u, p)$ if $p \in \mathcal{K}$.
- **Alternative** $(\pi_1 \cup \pi_2)$**:** a trace τ matches $\pi_1 \cup \pi_2$ if it matches either π_1 or π_2. To ensure no match exists, we must eliminate both possibilities. Thus, the algorithm recursively calls itself on (u, π_1, v) and (u, π_2, v).

The remaining cases, (π_1^-), $(\pi_1 \cdot \pi_2)$, (π_1^*), and $(\pi_1?)$ follow the same recursive principle and are detailed in App. E.

5.4.3 Qualified satisfaction $\mathtt{SAT}_\pi^n(sel, \varphi)$ The goal of a $\mathtt{SAT}_\pi^n(sel, \varphi)$ task is to ensure that a selected node $u \models sel$ has exactly n distinct π-targets satisfying φ. Let m be the current number of such targets, the goal is to add $k = n - m$ new targets that satisfy φ. This is achieved in two steps:

- **Find qualified targets:** Search the graph $\mathcal{G}$ for nodes that satisfy φ but not already π-targets of u.
- **Construct** π**-matching trace:** Create a new π-matching trace to connect u to each found target.

Further details are in App. F.

After introducing how violation expressions are mapped to graph operations. We continue with our running example to illustrate the test case generation process.

Running Example. Recall $\mathcal{P}_2 = \{(1, 2, 3, 4), (5, 6)\}$ from our running example. Its leaf nodes (nodes 4 and 6) are expressions:

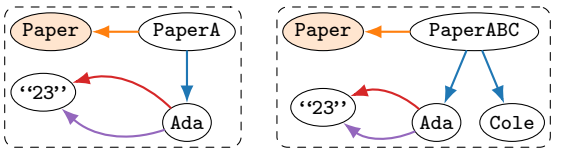
(E1) $\mathtt{VIO}^{(T_4,\forall)}(=\text{“123”}, \mathtt{len} \geq 3)\mathtt{VIO}_{\mathtt{reviewedBy}}(\mathtt{PaperABC}, \mathtt{Bob})$

(E2) $\mathtt{SAT}^4_{\mathtt{reviewedBy}}(\in \mathtt{Paper}, \geq_1 \pi.\mathtt{len} \geq 3)$

Take (E1) as an example. This expression applies two sets of operations to $\mathcal{G}$:

- $\mathtt{VIO}^{(T_4,\forall)}(=\text{“123”}, \mathtt{len} \geq 3)$ replaces the final hop of all traces in T_4 with a value failing the shape, e.g., “23”. This yields the following graph operations, disqualifying Ada as a Reviewer:
  - $rep(\mathtt{Ada}, \mathtt{workPhone}, \text{“123”}, \text{“23”})$
  - $rep(\mathtt{Ada}, \mathtt{cellPhone}, \text{“123”}, \text{“23”})$
- $\mathtt{VIO}_{\mathtt{reviewedBy}}(\mathtt{PaperABC}, \mathtt{Bob})$ is the base case of the path violation algorithm, disconnecting Bob from PaperABC, yielding $-e(\mathtt{PaperABC}, \mathtt{reviewedBy}, \mathtt{Bob})$.

Validating $\mathcal{G}'$ against $\mathcal{S}$ shows that both PaperA and PaperABC violate (sel_1, φ_1); their offending subgraphs are in Fig. 5. After applying E2 to a separate copy of $\mathcal{G}$, we obtain one additional invalid graph, yielding two test cases for the running example.



$\mathsf{OffSub}_{\mathcal{G}'}(\mathtt{PaperA}, (sel_1, \varphi_1))$ $\mathsf{OffSub}_{\mathcal{G}'}(\mathtt{PaperABC}, (sel_1, \varphi_1))$

Figure 5. Offending subgraphs of PaperA and PaperABC from applying E1 on the graph of the running example.

5.4.4 Eligibility Conditions and Coverage While we have outlined the procedure for each task, certain preconditions must be met for a task to be feasible. A task can be impossible for two reasons:

- The task is **logically invalid**. For example, the shape violation task $\mathtt{VIO}^{(T,Q)}(sel, \top)$ is impossible because it requires to violate a logical tautology ($\top$).
- The task is **operationally impossible**. For example, the shape violation task $\mathtt{VIO}^{(T,Q)}(sel, = u)$, T being a zero-length trace $\{\langle u \rangle\}$, is impossible, because our graph operations (Def. 2) are defined to modify edges and attributes, but a zero-length trace has no hop to modify.

To handle these cases, we introduce the concept of *eligibility* and use it as a filter: eligible tasks are mapped to graph operations, while ineligible tasks are discarded. A task is eligible only if it is both logically valid and operationally feasible. We now describe these two conditions.

Logical Precondition. A task must be logically valid.

- For a shape violation task $\mathtt{VIO}^{(T,Q)}(sel, \varphi)$: sel must not logically entail[†] φ, since we can not violate a constraint that is guaranteed to be true.
- For a qualified satisfaction task $\mathtt{SAT}_\pi^n(sel, \varphi)$: sel must select at least one node and φ must not be a contradiction, since we can not satisfy an unsatisfiable logical formula.

Operational Feasibility. A task is operationally infeasible when it must be induced via a zero-length trace $\langle u \rangle$, as this requires modifying the node u itself, which our edge-based graph operations can not do. To understand when this occurs for shape violation tasks, we first need to define intrinsic shapes.

Definition 9. Intrinsic Shapes. A shape φ is intrinsic if its truth value for a node u is independent of the graph's edges and attributes. For example, $= y$ is intrinsic, as it only depends on the node u itself.

We then detail when a task is operationally infeasible:

- For a shape violation task $\mathtt{VIO}^{(T,Q)}(sel, \varphi)$, the task is infeasible when φ is intrinsic and either of the following holds:

[†]Non-entailment includes cases such as: (i) selector matches at least one node ($\exists u\ (u \models sel)$), (ii) φ is not trivial ($\varphi \neq \top$ or any tautology), and (iii) φ is not implied by sel (e.g., if $sel = \varphi_1 \wedge \varphi_2$ and $\varphi = \varphi_1$, then φ is implied by sel and thus fails non-entailment).

- $Q = \forall$ and T contains at least one zero-length trace (as the "modify all" requirement can not be met), or
- $Q = \exists$ and all traces in T are zero-length traces (as there is no hop to modify).

- For a path violation task $\texttt{VIO}_\pi(u, v)$, the task is infeasible when the zero-length trace $\langle u \rangle$ matches π, which occurs when $u = v$ and π permits zero-length traces (e.g., via π_1^* or $\pi_1?$ path operators).

We now state the formal definition of eligibility.

Definition 10. Eligibility of Tasks. A task is *eligible* if and only if it meets the following conditions:

1. **Logical precondition**:
   - For $\texttt{VIO}^{(T,Q)}(sel, \varphi)$, sel does not logically entail φ.
   - For $\texttt{SAT}_\pi^n(sel, \varphi)$, sel matches at least one node and φ is not a contradiction.
2. **Operational feasibility:**
   - For $\texttt{VIO}^{(T,Q)}(sel, \varphi)$, the task is feasible unless φ is intrinsic and T contains only zero-length traces (or at least one if $Q = \forall$).
   - For $\texttt{VIO}_\pi(u, v)$, the task is feasible unless $\langle u \rangle$ matches π.

After ineligible tasks are discarded, the following theorem states that every remaining task yields a set of graph operations.

Theorem 2. Coverage Guarantee. For any schema $\mathcal{S}$ and valid graph $\mathcal{G}$, every reachable and eligible normalized shape violation task, every eligible path violation task, and every eligible qualified satisfaction task will generate a non-empty set of graph operations.

Proof. See App. D for the proof. □

6 Repair Systems

We introduce three representative automated graph repair systems: a symbolic method using answer set programming, a supervised-learning based method, and an LLM-based method. We focus on fully automated systems to ensure scalability. Consequently, we exclude Human-in-the-Loop methods (e.g., Pachera et al.[20]) and other interactive approaches.

Answer Set Programming-Based Repair. Ahmetaj et al.[17] repair violations using a logic-based framework called Answer Set Programming (ASP)[41]. ASP is a form of declarative programming where problems are modeled as logical programs to be solved by powerful and general-purpose solvers, e.g., Clingo[42].

Ahmetaj et al.[17] encode the schema, invalid graph, and graph operations into an answer set program. The program also includes an optimization objective to find repairs that minimize the number of graph operations, where each atomic operation (e.g., adding an edge) counts as one operation. This approach directly encodes the heuristic that the most desirable repair is one that alters the graph the least. The result is an *answer set*, which represents the complete collection of equally minimal repairs. From this set of potential solutions, the authors' implementation[43] selects the first answer returned by the solver to serve as the final repair.

A critical consideration for this approach is its computational cost. The solving of the encoded answer set program has a worst-case time complexity that is exponential in the combined size of the graph and the schema, which limits its scalability to large datasets.

Neural Knowledge Base Repair. Pellissier Tanon et al.[22] propose a supervised learning approach to learn repairs directly from historical graph edits. They train a neural network model with custom architecture to predict the correct repair for a given schema violation. The training data of the model consists of schema violations and the corresponding human-provided repairs. Once trained, it can predict repairs by identifying patterns in the graph and context surrounding the error. The primary limitation of this approach is its dependency on a substantial volume of training data, which may not be available for the specific graph one intends to repair. The authors' implementation[44] leveraged the extensive edit history of Wikidata[45], a crowd-sourced graph database, to train their model.

LLM-Based Repair. LLMs have recently emerged as a tool to interact with graphs[24,25]. Their primary limitation is their black-box nature, as performance is highly sensitive to the choice of model and the design of the prompt. This requires extensive empirical testing, a process known as prompt engineering[46]. Our framework supports this need for systematic evaluation; its ability to generate novel violations on private and unpublished graphs also mitigates the risk of data contamination from LLM training corpora[28]. For our evaluation, we selected four models and accessed them in January 2025: GPT4o (OpenAI), Claude 3.0 Opus (Anthropic), Gemini 1.5 Pro (Google), and Llama 3.1 405B (Meta, via AWS Bedrock).

To demonstrate how our framework provides systematic evaluation of the impact of different prompts, we construct a modular prompting template to provide necessary context, including Schema Context ($\mathcal{C}_m$) and Graph Context ($\mathcal{C}_g$). To interface with the LLM, all graph and schema information is serialized into the Turtle format[47], and the LLM is instructed to generate repairs in SPARQL syntax[48]. See App. H for the full prompt template. We define three configurations for both the schema and graph contexts, yielding nine variations, as detailed below.

Schema Context $\mathcal{C}_m$. The amount of schema information provided to the model is varied across three levels:

1. **Entire Schema (M)**: Provides the full schema $\mathcal{S}$. This offers maximum context but may exceed the LLM's context window for large schemas.
2. **Minimal Schema (S)**: Provides only the offended constraint (sel, φ). This improves scalability by focusing on the violated constraint.
3. **Annotated Schema (S_n)**: Extends the minimal schema (S) with natural language descriptions (e.g.,

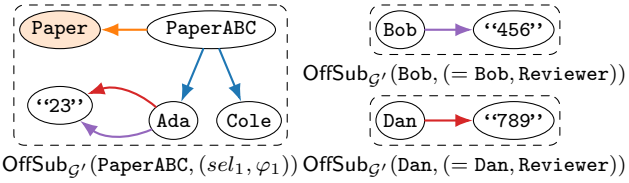


Figure 6. Focused subgraph context (F) of `PaperABC` violating (sel_1, φ_1) after E1 is applied in the running example.

`rdfs:label`). This lets us assess the effect of natural language descriptions.

Graph Context $\mathcal{C}_g$. Similarly, the amount of graph information is varied across three levels:

1. **Entire Graph (G)**: Provides the full graph $\mathcal{G}'$. Like the entire schema, this maximizes context at the cost of scalability.

2. **Focused Subgraph (F)**: Provides only the offending subgraph $\mathsf{OffSub}_{\mathcal{G}'}(u, (sel, \varphi))$ of offending focus u and offended constraint (sel, φ). This isolates the part of the graph where the error occurs. For cardinality constraints, $\varphi = \geq_n \pi.\varphi'$, we also include context around nodes that satisfy φ' to provide sufficient context. Specifically, F also includes the subgraph visited during the validation against φ'. For instance, in our running example where (E1) is applied so that `PaperABC` violates (sel_1, φ_1), the subgraphs for `Bob` and `Dan` are included because they satisfy the `Reviewer` constraint. The full context is visualized in Fig. 6.

3. **Focused Subgraph with Example (F^+)**: Augments the focused subgraph (F) with a positive example. In addition to the subgraph showing the error, it includes another subgraph where the same constraint is satisfied, acting as a one-shot example to guide the LLM.

7 Evaluation Metrics

We define three tiered metrics to evaluate the quality of a repair $\mathfrak{R}$. Each metric is assessed only if all lower-tier metrics are satisfied.

We formalize the metrics as binary indicator functions. Assume the i-th test case has repair $\mathfrak{R}$. Let $\mathcal{S}$ be the schema, $\mathcal{G} = (\mathcal{E}, \rho)$ be the original graph, $\mathcal{G}'$ be the invalid graph and $\mathcal{G}'_{\mathfrak{R}} = (\mathcal{E}'_{\mathfrak{R}}, \rho'_{\mathfrak{R}})$ be the result of applying repair $\mathfrak{R}$ to $\mathcal{G}'$. Let the *node identifiers* of $\mathcal{G}$ be N and the node identifiers of $\mathcal{G}'_{\mathfrak{R}}$ be N', where N is defined as the set of nodes appearing in $\mathcal{E}$ or the domain of ρ, and similarly, $N'_{\mathfrak{R}}$ is the set of nodes appearing in $\mathcal{E}'_{\mathfrak{R}}$ or the domain of $\rho'_{\mathfrak{R}}$.

1. **Syntactic Validity ($V_{syn}(i)$):** $V_{syn}(i) = 1$ if $\mathfrak{R}$ can be parsed by a standard library (e.g., RDFLib[49]) without error. Note: an empty repair is considered syntactically valid; else, $V_{syn}(i) = 0$.
2. **Semantic Validity ($V_{sem}(i)$):** $V_{sem}(i) = 1$ if $V_{syn}(i) = 1$ and eliminates all schema violations ($\mathcal{G}'_{\mathfrak{R}} \models \mathcal{S}$); else, $V_{sem}(i) = 0$.
3. **Isomorphism ($V_{iso}(i)$):** $V_{iso}(i) = 1$ if $V_{sem}(i) = 1$ and there exists a bijection $\Xi : N'_{\mathfrak{R}} \to N$ such that $(u, p, v) \in \mathcal{E}'_{\mathfrak{R}} \iff (\Xi(u), p, \Xi(v)) \in \mathcal{E}$ and $\rho'_{\mathfrak{R}}(u, k) = val \iff \rho(\Xi(u), k) = val$. Else, $V_{iso}(i) = 0$. (Note: Values $val \in \mathcal{V}$ are literals and map to themselves).

In Table 4, we report the success rate as the score for each metric, calculated as the average of these indicators over the number of test cases M ($\frac{1}{M}\sum_{i=1}^{M} V(i)$). For example, ASP generates 140 semantically valid repairs out of 144 test cases. Thus, its semantic validity score is $\frac{140}{144} \approx 97.22\%$.

Remark. Semantic validity and isomorphism formalize different notions of a "good" repair. Semantic validity captures correctness with respect to the schema; isomorphism checks whether the repair reverses the VE. As Fig. 1 illustrates, there can be multiple semantically valid repairs for a violation. To determine the correct repair, one often needs to inspect the surrounding context of the error[22] or consult external knowledge, such as a human expert[20]. We therefore report both metrics so that practitioners can prioritize schema conformance or isomorphism as appropriate. Using the original graph as a reference is a commonly adopted "silver standard" in knowledge-graph evaluation[20,50].

8 Evaluation Results

We evaluate the repair systems on three datasets of RDF graphs with corresponding SHACL schemas. **Brick Ontology**[1] standardizes building assets and their relationships. Using the 1.3 release, we construct eight graphs and schemas based on hardware specifications in the ASHRAE Guideline 36[51]. On average, the graphs contain 35 edges and the schemas contain 108 edges. In total, our framework yields 144 test cases. **LUBM Ontology**[52] is a synthetic dataset modeling the relationships between entities such as departments and professors in a university. We adapt the graph and schema from Figuera et al.[53,54]. The graph contains 229 edges and the schema contains 207 edges. Our framework yields 70 test cases. **QUDT Ontology**[12] provides a unified conceptual model for physical quantities and units. In the 2.1 release, the graph contains 81,293 edges and the schema contains 2,642 edges. Our framework yields 134 test cases. As this dataset is two to three orders of magnitude larger than the others, it serves to test the scalability of each repair method.

Our evaluation proceeds in two main stages. First, since LLMs are sensitive to prompt design and model choices, we leverage our framework to analyze the behavior of LLMs as repair systems. Sec. 8.1 characterizes the performance variance of the LLM, establishing its performance upper bound. Second, to demonstrate our framework's diagnostic ability across diverse approaches, we evaluate and compare the performance of all three repair methods in Sec. 8.2: answer set programming based-repair (ASP), neural knowledge base repair (Neural), and the performance bounds of LLM-based repair (LLM).

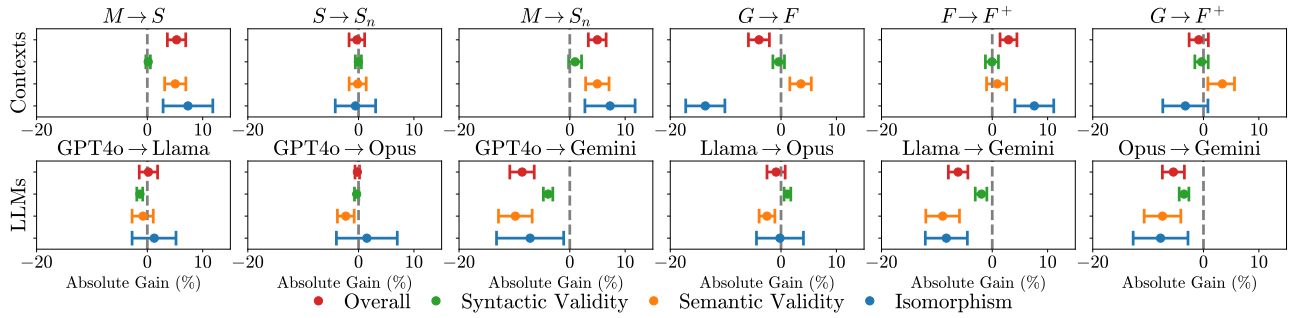


Figure 7. Estimated effects of context selection and LLM selection.

8.1 Impact of Context and Model Choice on LLM-Based Repair

We analyze the performance of four different LLMs across nine prompting strategies to measure the effects of both context and model choice on repair quality. Due to budget constraints, prompts that cost more than $0.50 are omitted. These exclusions only occur on the large-scale QUDT dataset and affect the following prompting strategies:

- The MG, SG, and S_nG prompts for all LLMs.
- The MF and MF^+ prompts for Claude 3.0 Opus, whose granular tokenization results in higher costs.

The fact that all the affected prompting strategies involve either the entire schema (M) or the entire graph (G) shows the poor scalability of M and S contexts. A detailed cost analysis for all models and prompts is available in App. H.

Schema Context. We investigate the following questions: (i) Does using M produce a different result from S? (ii) Does using S produce a different result from S_n? (iii) Does using M produce a different result from S_n? Fig. 7 reports the absolute gain in percentage points ($Score_{new} - Score_{old}$). For example, if the prompt strategy M achieves 80% semantic validity and the prompt strategy S achieves 85%, we report a semantic validity absolute gain of +5% for $M \to S$, where the 95% confidence interval is estimated using linear mixed-effects model (LMM) [55]. Further details on the statistical analysis are in App. I. Metrics whose intervals do not cross 0% are statistically significant ($p < 0.05$).

Switching from M to S improves all metrics except syntactic validity, suggesting that full-schema prompts may overload or distract the LLM, while concise prompts focusing on the violated constraint yield better repairs. S and S_n perform similarly, and both outperform M; however, S is more efficient as it requires fewer tokens.

Graph Context. In Fig. 7, we also analyze the impact of different graph contexts. Switching from G to F reduces the overall performance, especially for the isomorphism metric, but improves semantic validity. This indicates that while removing extraneous information helps the LLM focus on eliminating violations (thus improving semantic validity), it also deprives the model of the context needed to generate repairs that closely match the original graph. Unlike schema context, where reducing information ($M \to S$) is beneficial, reducing graph context harms the stricter metrics, suggesting that information not directly used in validation still aids the LLM in inferring suitable repairs. We also evaluate the F^+ context, which adds an example. The results show a significant improvement from F to F^+. Furthermore, F^+ achieves overall performance that is statistically insignificant from G, but without its prohibitive cost. Overall, F^+ offers a balance between performance and cost.

Summary of Context Impact. The most effective and scalable results are achieved with schema context S (minimal schema) and graph context F^+ (focused subgraph with example), which together provide essential information while controlling cost.

Model Choice. In Fig. 7, we compare different model choices. Replacing GPT4o, Llama, or Opus with Gemini all significantly degrade performance, while the other three perform similarly.

8.2 Comparison of Repair Systems

We now conduct a comparative evaluation of the three paradigms: answer set programming-based repair (ASP), neural knowledge base repair (Neural), and LLM-based repair (LLM).

8.2.1 Evaluation Setup To ensure a fair comparison, we tailor the evaluation setup for each repair system. This involves adapting the reference implementations and defining how to handle their output formats.

Answer Set Programming-Based Repair. The reference implementation [43] arbitrarily selects the first solution from its answer set, which may not represent the optimal performance of ASP. To provide a more thorough assessment, we evaluate the entire answer set of repairs the solver generates. We introduce two metrics for this purpose:

- Isomorphic Hit (binary): Equals 1 if the answer set contains the correct (isomorphic) repair, and 0 otherwise.
- Isomorphic Precision: Equals $\frac{\text{IsomorphicHit}}{|\text{AnswerSet}|}$. This measures how efficiently the solver returns the isomorphic repair.

Neural Knowledge Base Repair. The neural knowledge base repair presents two challenges. First,

Table 4. Average scores (%) of the repair systems. "Brick" averages over the Brick dataset; "All" averages over all datasets. Higher scores are better for all metrics: syntactic validity (Syn. Val.), semantic validity (Sem. Val.), isomorphism (Iso.), isomorphic hit (Iso. Hit), and precision (Iso. Prec.).

Scope	Method	Syn. Val.	Sem. Val.	Iso.	Iso. Hit	Iso. Prec.
Brick	ASP [17]	**100.0**	**97.22**	32.64	66.67	29.32
	Neural [22]	**100.0**	0	0	–	–
	Subset Avg	99.44	90.49	62.67	–	–
	Llama+SF^+	99.31	91.67	**68.06**	–	–
All	Neural [22]	**100.0**	10.37	8.36	–	–
	Subset Avg	96.96	87.95	52.41	–	–
	Llama+SF^+	96.84	**91.38**	**58.62**	–	–

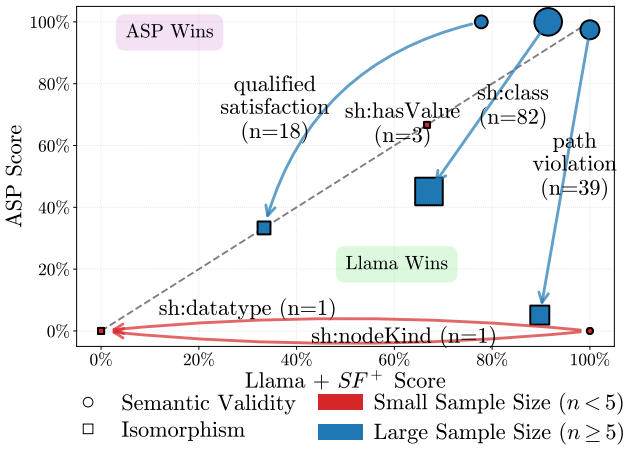


Figure 8. Per-constraint performance shift from semantic validity to isomorphism, comparing LLM and ASP on the Brick dataset. Point size $\propto$ sample count. `sh:nodeKind` requires a node be of a given node kind, e.g. a blank node. `path violation` includes SHACL keywords `sh:minCount` and `sh:qualifiedMinCount`. `qualified satisfaction` corresponds to SHACL keyword `sh:qualifiedMaxCount`.

it requires a training set of the graph being repaired, which is not available for our datasets. We address this by training the authors' official model using their provided Wikidata edit history data set [44]. Second, since the model is trained with Wikidata, it expects Wikidata-specific violation types, not SHACL. We therefore implement an adapter to translate SHACL violations into the format required by the model.

LLM-Based Repair. For the LLM-based approach, we exclude prompting strategies with excessive costs, leaving only four scalable strategies: SF, S_nF, SF^+, and S_nF^+. We report:

1. **Subset average:** The mean score across all four strategies and all four LLMs, representing the typical performance for the LLM-based approach.

2. **Best configuration:** The performance of Llama3.1 with SF^+, which was identified as one of the top-performers in Sec. 8.1. This represents the upper-bound for the LLM-based approach.

8.2.2 Analysis Preliminary experiments showed that ASP scaled poorly, failing to solve medium and large test cases (LUBM and QUDT) within several hours. We thus set a 30-minute time limit for all experiments. Under this constraint, ASP completed only the Brick ontology cases while other methods completed all test cases. Therefore, we only use Brick to assess the repair quality of ASP. On the Brick dataset (Table 4), both ASP and Neural achieve perfect syntactic validity. Neural's score is a consequence of it consistently producing empty repairs. Most syntactic errors from the LLM are formatting issues such as missing quotation marks.

Regarding semantic validity, Neural scores 0%, as its empty repairs do not fix violations. This poor performance highlights its sensitivity to domain shift. In contrast, ASP is highly effective, achieving 97.22% by repairing 140 out of 144 test cases. The four remaining failures are due to a non-algorithmic issue: the implementation's handling of *blank nodes*. In RDF graphs, entities are identified by identifiers and blank nodes are unnamed identifiers without a global name. The failures occurred because the reference implementation [43] struggled to correctly reference or create these anonymous entities in generating repairs. The discovery of this vulnerability showcases the utility of our framework in identifying practical robustness issues. Despite this implementation-related flaw, the high success rate of ASP demonstrates the power of logic-based solvers, which consistently outperform the LLMs.

The performance of ASP on isomorphism drops drastically. The isomorphic hit reveals that only 66.67% of the answer sets generated contained the correct repair. Our investigation reveals that the ASP encoding excludes some possible repairs: the ASP does not encode a mechanism to connect nodes disconnected from the offending focus. Take Fig. 6 for an example, to repair the violation of (sel_1, φ_1) for `PaperABC`, ASP's generated answer set includes adding new `workPhone` or `cellPhone` attributes (e.g., `"new_1475"` string) to `Ada` or `Cole`. However, connecting `PaperABC` to an existing node, `Bob` or `Dan` via `reviewedBy` is not included. Even though reusing `Bob` or `Dan` is also a minimal repair with only one graph operation.

The second block of Table 4 aggregates the results across all datasets (Brick, LUBM, QUDT), excluding ASP due to timeouts. LLMs outperform Neural on all metrics except syntactic validity because most repairs produced by Neural is empty, which is reflected in the decrease in Neural's semantic validity.

8.2.3 Per-constraint Diagnostic Analysis We then perform a per-constraint diagnostic analysis of the best LLM configuration (Llama3.1+SF^+) and ASP on the Brick dataset to characterize the kind of repairs that are supported by each repair method. Figure 8 plots, for each constraint, the LLM's performance (x-axis) and ASP's performance (y-axis), for semantic validity and isomorphism. The upper-left is when ASP wins and the

lower-right is when Llama+SF^+ wins. Point size indicates sample size. Since ASP completed only the Brick cases, the number of points is limited, and small samples exhibit high variance. As the metric becomes more stringent (semantic validity → isomorphism), nearly all points shift left, indicating lower LLM performance on the stricter metric.

A notable case is `path violation`, where both LLM and ASP are nearly perfect on semantic validity but LLM outperforms ASP by nearly 90% on isomorphism. As mentioned in Sec. 8.2.2, ASP can not perform repairs that reuse nodes disconnected from the offending focus, whereas for LLM, the graph context F^+ allows such reuse of nodes, as visualized in Fig. 6. This results in the drastic performance difference on different metrics for the same violations. This demonstrates the diagnostic power of our framework: *pinpoint the specific constraints that a repair method fails,* which is achieved by controlled and isolated injection of violations to ensure attributable performance difference to specific constraint violations without the confounder of interactions between violations.

9 Conclusion

Our framework systematically generates schema violations to produce test datasets with coverage guarantees, enabling per-constraint performance analysis. Built on an abstract rewriting system, it ensures multi-metric evaluation across diverse repair methods. Applying the framework to symbolic, supervised learning, and LLM-based methods reveals the strengths and weaknesses of each approach. The symbolic method achieves high semantic validity on small test cases; the supervised-learning based method is sensitive to domain shift; and for LLM-based methods, they are generally effective without dominating across all metrics, with their performance and scalability being highly dependent on the contextual information in the prompt.

A key contribution of our framework is the ability to characterize the specific types of repairs a system can and can not perform. For instance, our analysis revealed a limitation in the symbolic method: its inability to reuse disconnected nodes.

One limitation of our framework is its inability to handle recursive schemas. However, note that while recursion is supported by the ShEx standard, it remains undefined in SHACL. Future work will focus on applying multiple violation expressions to a single graph to investigate the interaction between violations. Another direction is to investigate the "ripple effect" (as seen in Fig. 5), where a single violation expression causes multiple nodes to violate a constraint.

10 Acknowledgment

This work was supported by the Laboratory Directed Research and Development (LDRD) Program of Lawrence Berkeley National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. Balaji B, Bhattacharya A, Fierro G et al. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. pp. 41–50.
2. Yu N, Nützmann HW, MacDonald JT et al. Delineation of metabolic gene clusters in plant genomes by chromatin signatures. *Nucleic Acids Research* 2016; 44(5): 2255–2265. DOI:10.1093/nar/gkw100.
3. Vrandečić D and Krötzsch M. Wikidata: A free collaborative knowledgebase. *Communications of the ACM* 2014; 57(10): 78–85. DOI:10.1145/2629489.
4. Fierro G, Pritoni M, Abdelbaky M et al. Mortar: An open testbed for portable building analytics. *ACM Trans Sen Netw* 2019; 16(1). DOI:10.1145/3366375. URL https://doi.org/10.1145/3366375.
5. Rodríguez-Muro M, Kontchakov R and Zakharyaschev M. Ontology-Based Data Access: Ontop of Databases. In Hutchison D, Kanade T, Kittler J et al. (eds.) *Advanced Information Systems Engineering*, volume 7908. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-38708-1 978-3-642-38709-8, 2013. pp. 558–573. DOI: 10.1007/978-3-642-41335-3_35.
6. Bollacker K, Evans C, Paritosh P et al. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. pp. 1247–1250.
7. Bi Z, Chen J, Jiang Y et al. Codekgc: Code language model for generative knowledge graph construction. *ACM Transactions on Asian and Low-Resource Language Information Processing* 2024; 23(3): 1–16.
8. Farzana S, Zhou Q and Ristoski P. Knowledge graph-enhanced neural query rewriting. In *Companion Proceedings of the ACM Web Conference 2023*. pp. 911–919.
9. Xu Z, Cruz MJ, Guevara M et al. Retrieval-augmented generation with knowledge graphs for customer service question answering. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. pp. 2905–2909.
10. Suchanek FM, Alam M, Bonald T et al. YAGO 4.5: A Large and Clean Knowledge Base with a Rich Taxonomy. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Washington DC USA: ACM. ISBN 9798400704314, pp. 131–140. DOI:10.1145/3626772.3657876.
11. Hammar K, Wallin EO, Karlberg P et al. The RealEstateCore Ontology. p. 16.
12. FAIRsharingorg. Qudt; quantities, units, dimensions and types, 2022. URL https://doi.org/10.25504/FAIRsharing.d3pqw7. Last Edited: Friday, May 6th 2022, 2:03, Last Accessed: Friday, December 13th 2024, 11:47, Last Reviewed: Monday, May 2nd 2022, 3:22.
13. Pareti P and Konstantinidis G. A review of shacl: from data validation to schema reasoning for rdf graphs. *Reasoning Web International Summer School* 2021; : 115–144.

14. Shex - shape expressions. https://shex.io/, 2025. Accessed: 2025-05-05.
15. Angles R, Bonifati A, Dumbrava S et al. Pg-schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 2023; 1(2): 1–25.
16. Hobbs JR and Pan F. An ontology of time for the semantic web. *ACM Transactions on Asian Language Information Processing (TALIP)* 2004; 3(1): 66–85.
17. Ahmetaj S, David R, Polleres A et al. Repairing shacl constraint violations using answer set programming. In *International Semantic Web Conference*. Springer, pp. 375–391.
18. Wright J, Rodríguez Méndez SJ, Haller A et al. Schimatos: a shacl-based web-form generator for knowledge graph editing. In *International Semantic Web Conference*. Springer, pp. 65–80.
19. Fierro G, Saha A, Shapinsky T et al. Application-driven creation of building metadata models with semantic sufficiency. In *Proceedings of the 9th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. pp. 228–237.
20. Pachera A, Bonifati A and Mauri A. User-centric property graph repairs. *Proceedings of the ACM on Management of Data* 2025; 3(1): 1–27.
21. Pellissier Tanon T, Bourgaux C and Suchanek F. Learning how to correct a knowledge base from the edit history. In *The World Wide Web Conference*. pp. 1465–1475.
22. Pellissier Tanon T and Suchanek F. Neural knowledge base repairs. In *The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings 18*. Springer, pp. 287–303.
23. Fan W, Lu P, Tian C et al. Deducing certain fixes to graphs. *Proceedings of the VLDB Endowment* 2019; 12(7): 752–765.
24. Arnaout H, Tran TK, Stepanova D et al. Utilizing language model probes for knowledge graph repair. In *Wiki Workshop 2022*.
25. Terdalkar H, Bonifati A and Mauri A. Graph repairs with large language models: An empirical study. In *Proceedings of the 8th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. pp. 1–10.
26. Gayo JEL, Knublauch H and Kontokostas D. Data Shapes Test Suite. https://w3c.github.io/data-shapes/data-shapes-test-suite/, 2025. Accessed: 2025-04-23.
27. Resource description framework (rdf). https://www.w3.org/RDF/, 2025. Accessed: 2025-05-05.
28. Kiela D, Bartolo M, Nie Y et al. Dynabench: Rethinking benchmarking in nlp. *arXiv preprint arXiv:210414337* 2021; .
29. Mihindukulasooriya N, Tiwari S, Enguix CF et al. Text2kgbench: A benchmark for ontology-driven knowledge graph generation from text. In *International Semantic Web Conference*. Springer, pp. 247–265.
30. Malaviya C, Bhagavatula C, Bosselut A et al. Commonsense knowledge base completion with structural and semantic context. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34. pp. 2925–2933.
31. Wang L, Zhao W, Wei Z et al. Simkgc: Simple contrastive knowledge graph completion with pre-trained language models. *arXiv preprint arXiv:220302167* 2022; .
32. Shen T, Zhang F and Cheng J. A comprehensive overview of knowledge graph completion. *Knowledge-Based Systems* 2022; 255: 109597.
33. Ahmetaj S, Boneva I, Hidders J et al. Common foundations for shacl, shex, and pg-schema. In *Proceedings of the ACM on Web Conference 2025*. pp. 8–21.
34. Duan X, Chaves-Fraga D, Derom O et al. Scoop all the constraints' flavours for your knowledge graph. In *European Semantic Web Conference*. Springer, pp. 217–234.
35. Cimmino A, Fernández-Izquierdo A and García-Castro R. Astrea: automatic generation of shacl shapes from ontologies. In *European Semantic Web Conference*. Springer, pp. 497–513.
36. Rabbani K, Lissandrini M and Hose K. Extraction of validating shapes from very large knowledge graphs. *Proceedings of the VLDB Endowment* 2023; 16(5): 1023–1032.
37. Barret N, Enache T, Manolescu I et al. Finding the pg schema of any (semi) structured dataset: a tale of graphs and abstraction. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, pp. 365–369.
38. Corman J, Reutter JL and Savković O. Semantics and validation of recursive shacl. In *The Semantic Web–ISWC 2018: 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, Proceedings, Part I 17*. Springer, pp. 318–336.
39. Jakubowski M. *Shapes Constraint Language: Formalization, Expressiveness, and Provenance*. PhD Thesis, Ph. D. Dissertation. Universiteit Hasselt and Vrije Universiteit Brussel, 2024.
40. Shapes constraint language (shacl). https://www.w3.org/TR/shacl/, 2025. Accessed: 2025-05-05.
41. Lifschitz V. *Answer set programming*, volume 3. Springer Cham, 2019.
42. Gebser M, Kaminski R, Kaufmann B et al. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming* 2019; 19(1): 27–82.
43. David R. Shacl-repairs. https://github.com/robert-david/shacl-repairs, 2025. Accessed: August 2025.
44. Tanon TP. Neural knowledge base repairs. https://github.com/Tpt/bass-materials, 2025. Accessed: August 2025.
45. Vrandečić D and Krötzsch M. Wikidata: a free collaborative knowledgebase. *Communications of the ACM* 2014; 57(10): 78–85.
46. Schulhoff S, Ilie M, Balepur N et al. The prompt report: a systematic survey of prompt engineering techniques. *arXiv preprint arXiv:240606608* 2024; .
47. Turtle syntax. https://www.w3.org/TR/turtle/, 2025. Accessed: 2025-05-05.
48. Simple protocol and rdf query language, (sparql). https://www.w3.org/TR/sparql11-query/, 2025. Accessed: 2025-05-05.

49. RDFLib. Rdflib. https://rdflib.readthedocs.io/en/stable/, 2025.
50. Paulheim H. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web* 2016; 8(3): 489–508.
51. ASHRAE G. 36: High performance sequences of operation for hvac systems. *American Society of Heating, Refrigerating and Air-Conditioning Engineers, Atlanta* 2018; .
52. Guo Y, Pan Z and Heflin J. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics* 2005; 3(2-3): 158–182.
53. Figuera M, Rohde PD and Vidal ME. Trav-shacl: Efficiently validating networks of shacl constraints. In *Proceedings of the Web Conference 2021*. pp. 3337–3348.
54. SDM-TIB. Tracedsparql, 2023. URL https://github.com/SDM-TIB/TracedSPARQL. Accessed: 2024-12-13.
55. Pinheiro JC and Bates DM. Linear mixed-effects models: basic concepts and examples. *Mixed-effects models in S and S-Plus* 2000; : 3–56.
56. Dershowitz N and Manna Z. Proving termination with multiset orderings. *Communications of the ACM* 1979; 22(8): 465–476.
57. rstr. https://github.com/leapfrogonline/rstr, 2025.
58. Zdaniuk B. Ordinary least-squares (ols) model. In *Encyclopedia of quality of life and well-being research*. Springer, 2024. pp. 4867–4869.

A Proof of Strong Normalization

We provide the proof for Theorem 1, restated below for convenience.

Theorem 1. Strong Normalization of $\mathcal{A}$. If $\mathcal{G}$ and $\mathcal{S}$ are finite and $\mathcal{S}$ is non-recursive, then the rewriting system $\mathcal{A}$ is strongly normalizing.

To prove that $\mathcal{A}$ is strongly normalizing, we show that every sequence of rewrites must be finite. We define a measure function μ that maps each violation expression `Expr` to an element of a well-founded set and prove that for any rewrite step $\mathtt{E}_1 \to \mathtt{E}_2$, the measure of the result is strictly smaller than the measure of the original, i.e., $\mu(\mathtt{E}_1) > \mu(\mathtt{E}_2)$.

A.1 Well-Founded Set

The well-founded set for our proof is the set of finite multisets of pairs of natural numbers, denoted $M(\mathbb{N} \times \mathbb{N})$. A multiset is a collection where elements may appear more than once. This set is ordered using the Dershowitz-Manna (DM) ordering[56], commonly used to prove program termination. The DM ordering is denoted as $\gg$, which is built upon a base order $>$ on the pairs $(d, c) \in \mathbb{N} \times \mathbb{N}$. The base order on pairs is defined as:

$$(d_1, c_1) > (d_2, c_2) \iff (d_1 > d_2) \text{ or } (d_1 = d_2 \wedge c_1 > c_2)$$

This ordering is well-founded, meaning an infinite descending chain of pairs $(d_1, c_1) > (d_2, c_2) > \cdots$ is impossible. This is because

- The d component is a natural number and can only decrease a finite number of times before it ends at 0.
- Once the d component stops decreasing, the c component must start strictly decreasing until it ends at 0.

The DM ordering extends this to multisets. Formally, given two finite multisets M_1 and M_2, we say $M_1 \gg M_2$ if M_2 can be obtained from M_1 by removing a single element $x \in M_1$ and replacing it with a *finite* (possibly empty) multiset of new elements Y, such that every new element $y \in Y$ is strictly smaller than x, i.e., $x > y$. For example, the multiset $\{(3,3),(3,3),(4,1)\}$ is greater than both the multisets $\{(3,3),(4,1)\}$ and $\{(2,1),(4,1)\}$.

The DM ordering is well-founded because the underlying order on the (d, c) pairs is itself well-founded. An infinite descending chain of multisets $M_1 \gg M_2 \gg M_3 \gg \cdots$ would imply the existence of an infinite descending chain of the (d, c) pairs, which is impossible.

A.2 Measure Function

We define the measure function μ that maps any violation expression to a multiset of (d, c) pairs. This measure is constructed from a primary measure d for quantifier depth and a secondary measure c for structural complexity.

A.2.1 Primary Measure: Quantifier Depth $d(\varphi)$ First, we define the function $d(\varphi)$ to be the maximum nesting depth of quantifiers in φ. $d(\varphi)$ is defined as

$$\begin{cases} 0 & \varphi \text{ atomic}, \\ d(def(s, \mathcal{S})) & \varphi = s \in \Sigma, \\ \max\{d(\varphi_1), d(\varphi_2)\} & \varphi = \varphi_1 \wedge \varphi_2 \text{ or } \varphi_1 \vee \varphi_2, \\ d(\psi) & \varphi = \neg\psi \text{ and } \varphi \text{ non-atomic}, \\ 1 + d(\psi) & \varphi \in \{\forall\pi.\psi,\ \geq_n \pi.\psi,\ \leq_n \pi.\psi\}. \end{cases}$$

The condition that the schema $\mathcal{S}$ is non-recursive is crucial here, as it guarantees that $d(s)$ is well-defined.

A.2.2 Secondary Measure: Structural Complexity $c(\varphi)$ Next, we define a function $c(\varphi)$ that maps φ to a natural number representing its complexity. $c(\varphi)$ is defined as

$$\begin{cases} 1 & \varphi \text{ atomic}, \\ 1 + c(def(s, \mathcal{S})) & \varphi = s \in \Sigma, \\ 1 + c(\varphi_1) + c(\varphi_2) & \varphi = \varphi_1 \wedge \varphi_2 \text{ or } \varphi_1 \vee \varphi_2, \\ 2 \cdot c(\psi) & \varphi = \neg\psi \text{ and } \varphi \text{ non-atomic}, \\ 1 + c(\psi) & \varphi \in \{\forall\pi.\psi,\ \geq_n \pi.\psi,\ \leq_n \pi.\psi\}. \end{cases}$$

A.2.3 The Full Measure $\mu(\mathtt{E})$ Finally, the measure $\mu(\mathtt{E})$ for any violation expression $\mathtt{E}$ is the multiset union of the measures of all the fundamental tasks it contains. For a single fundamental task $\mathtt{T}$, the measure is a singleton multiset:

$$\mu(\mathtt{T}) = \begin{cases} \{(d(\varphi), c(\varphi))\} & \mathtt{T} = \mathtt{VIO}^{(T,Q)}(sel, \varphi), \\ \{(0, c(\varphi))\} & \mathtt{T} = \mathtt{SAT}_\pi(sel, \varphi), \\ \{(0, 1)\} & \mathtt{T} = \mathtt{VIO}_\pi(u, v), \end{cases}$$

For a compound expression $\mathtt{E}$ composed of fundamental tasks $\mathtt{T}_1, \mathtt{T}_2, \cdots, \mathtt{T}_k$ (combined with $+$ or $\cdot$), the full measure is the multiset union of their individual measures.

$$\mu(\mathtt{E}) = \bigcup_{i=1}^{k} \mu(\mathtt{T}_i)$$

A.3 Strict Measure Decrease

We now show that for each rewrite rule $LHS \to RHS$, it holds that $\mu(LHS) \gg \mu(RHS)$.

Rule 1: Shape Reference (s)

$$\begin{aligned} &\mathtt{VIO}^{(T,Q)}(sel, s) \longrightarrow \mathtt{VIO}^{(T,Q)}(sel, def(s, \mathcal{S})) \\ &\mu(LHS) = \{(d(s), c(s))\} = \{(d(s), 1 + c(def(s, \mathcal{S})))\} \\ &\mu(RHS) = \{(d(s), c(def(s, \mathcal{S})))\} \\ &\quad \mathtt{SAT}^n_\pi(sel, s) \longrightarrow \mathtt{SAT}^n_\pi(sel, def(s, \mathcal{S})) \\ &\quad\quad \mu(LHS) = \{(0, 1 + c(def(s, \mathcal{S})))\} \\ &\quad\quad \mu(RHS) = \{(0, c(def(s, \mathcal{S})))\} \end{aligned}$$

Both rewrites replace one element with another. The d component is unchanged, while the c component strictly decreases. Thus, $\mu(LHS) \gg \mu(RHS)$.

Rule 2: Conjunction ($\wedge$)

$$\begin{aligned} \mathtt{VIO}^{(T,Q)}(sel, \varphi \wedge \varphi') \longrightarrow &\mathtt{VIO}^{(T,Q)}(sel, \varphi) + \\ &\mathtt{VIO}^{(T,Q)}(sel, \varphi') \\ \mu(LHS) = \{(&\max\{d(\varphi), d(\varphi')\}, 1 + c(\varphi) + c(\varphi'))\} \\ \mu(RHS) = \{&(d(\varphi), c(\varphi)), (d(\varphi'), c(\varphi'))\} \end{aligned}$$

The rewrite replaces one element with two new elements. Both pairs in $\mu(RHS)$ are strictly smaller than the single pair in $\mu(LHS)$. Thus, $\mu(LHS) \gg \mu(RHS)$.

Rule 3: Disjunction ($\vee$)

$$\texttt{VIO}^{(T,Q)}(sel, \varphi \vee \varphi') \longrightarrow \sum_{u:u\models sel} \big(\textbf{if } u \models \varphi \textbf{ then } \texttt{VIO}^{(T|_u,Q)}(= u, \varphi)\big) \cdot \big(\textbf{if } u \models \varphi' \textbf{ then } \texttt{VIO}^{(T|_u,Q)}(= u, \varphi')\big)$$

$$\mu(LHS) = \{(\max\{d(\varphi), d(\varphi')\}, 1 + c(\varphi) + c(\varphi'))\}$$
$$\mu(RHS) = \bigcup_{u:u\models sel} \{(d(\varphi), c(\varphi)), (d(\varphi'), c(\varphi'))\}$$

The RHS potentially omits pairs, depending on the **if** conditions.

The logic that $\mu(LHS) \gg \mu(RHS)$ is identical to Rule 2. One element is replaced by a finite multiset of new elements, all of which are strictly smaller. The finiteness of the graph $\mathcal{G}$ is thus an important condition here.

Rule 4: Negation ($\neg$)

1. De Morgan's Law

$$\texttt{VIO}^{(T,Q)}(sel, \neg(\varphi \wedge \varphi')) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \neg\varphi \vee \neg\varphi')$$
$$\mu(LHS) = \{(\max\{d(\varphi), d(\varphi')\}, 2(1 + c(\varphi) + c(\varphi')))\}$$
$$\mu(RHS) = \{(\max\{d(\varphi), d(\varphi')\}, 1 + 2c(\varphi) + 2c(\varphi'))\}$$
$$\texttt{VIO}^{(T,Q)}(sel, \neg(\varphi \vee \varphi')) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \neg\varphi \wedge \neg\varphi')$$
$$\mu(LHS) = \{(\max\{d(\varphi),d(\varphi')\}, 2(1 + c(\varphi) + c(\varphi')))\}$$
$$\mu(RHS) = \{(\max\{d(\varphi),d(\varphi')\}, 1 + 2c(\varphi) + 2c(\varphi'))\}$$
$$\texttt{VIO}^{(T,Q)}(sel, \neg\neg\varphi) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \varphi)$$
$$\mu(LHS) = \{(d(\varphi), 4c(\varphi))\}$$
$$\mu(LHS) = \{(d(\varphi), c(\varphi))\}$$

2. Quantifier Dualities

$$\texttt{VIO}^{(T,Q)}(sel, \neg(\forall\pi.\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \geq_1 \pi.\neg\varphi)$$
$$\mu(LHS) = \{(1 + d(\varphi), 2(1 + c(\varphi)))\}$$
$$\mu(RHS) = \{(1 + d(\varphi), 1 + 2c(\varphi))\}$$
$$\texttt{VIO}^{(T,Q)}(sel, \leq_0 \pi.\neg\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \forall\pi.\varphi)$$
$$\mu(LHS) = \{(1 + d(\varphi), 1 + 2c(\varphi))\}$$
$$\mu(RHS) = \{(1 + d(\varphi), 1 + c(\varphi))\}$$
$$\texttt{VIO}^{(T,Q)}(sel, \neg(\geq_n \pi.\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \leq_{n-1} \pi.\varphi)$$
$$\mu(LHS) = \{(1 + d(\varphi), 2(1 + c(\varphi)))\}$$
$$\mu(RHS) = \{(1 + d(\varphi), 1 + c(\varphi))\}$$
$$\texttt{VIO}^{(T,Q)}(sel, \neg(\leq_n \pi.\varphi)) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \geq_{n+1} \pi.\varphi)$$
$$\mu(LHS) = \{(1 + d(\varphi), 2(1 + c(\varphi)))\}$$
$$\mu(RHS) = \{(1 + d(\varphi), 1 + c(\varphi))\}$$

3. Shape Unfolding

$$\texttt{VIO}^{(T,Q)}(sel, \neg s) \longrightarrow \texttt{VIO}^{(T,Q)}(sel, \neg def(s, \mathcal{S}))$$
$$\mu(LHS) = \{(d(def(s, \mathcal{S})), 2(1 + c(def(s, \mathcal{S}))))\}$$
$$\mu(RHS) = \{(d(def(s, \mathcal{S})), 2c(def(s, \mathcal{S})))\}$$

In each case, the d component is unchanged, while the c component strictly decreases. Thus, $\mu(LHS) \gg \mu(RHS)$.

Rule 5: Universal Quantifier ($\forall\pi.\varphi$)

$$\texttt{VIO}^{(T,Q)}(sel, \forall\pi.\varphi) \longrightarrow \sum_{u:u\models sel} \sum_{v\in[\![\pi]\!](u)} \texttt{VIO}^{(T'_{u\to v},\exists)}(= v, \varphi) + \texttt{SAT}^1_\pi(sel, \neg\varphi)$$

$$\mu(LHS) = \{(1 + d(\varphi), 1 + c(\varphi))\}$$
$$\mu(RHS) = \bigcup_{\substack{u:u\models sel \\ v\in[\![\pi]\!](u)}} \{(d(\varphi), c(\varphi))\} \cup \{0, 2c(\varphi)\}$$

The d component of every task on the RHS is at most $d(\varphi)$, which is strictly smaller than the LHS d component of $1 + d(\varphi)$. Therefore, every element in $\mu(RHS)$ is strictly smaller than the element in $\mu(LHS)$. Thus, $\mu(LHS) \gg \mu(RHS)$.

Rule 6: Qualified Cardinality ($\geq_n \pi.\varphi$)

$$\texttt{VIO}^{(T,Q)}(sel, \geq_n \pi.\varphi) \longrightarrow \sum_{u:u\models sel} \sum_{\substack{\Delta\subseteq[\![\pi,\varphi]\!](u) \\ |\Delta|=k}} \prod_{v\in\Delta} \Big(\texttt{VIO}^{(T'_{u\to v},\forall)}(= v, \varphi) + \texttt{VIO}_\pi(u, v)\Big)$$

$$\mu(LHS) = \{(1 + d(\varphi), 1 + c(\varphi))\}$$
$$\mu(RHS) = \bigcup_{\substack{u:u\models sel \\ \Delta\subseteq[\![\pi,\varphi]\!](u) \\ v\in\Delta}} \{(d(\varphi), c(\varphi))\} \cup \{0, 1\}$$

The d component of every task on the RHS is at most $d(\varphi)$, which is strictly smaller than the LHS d component of $1 + d(\varphi)$. Therefore, every element in $\mu(RHS)$ is strictly smaller than the element in $\mu(LHS)$. Thus, $\mu(LHS) \gg \mu(RHS)$.

Rule 7: Qualified Cardinality ($\leq_n \pi.\varphi$) $\texttt{SAT}^{n+1}_\pi$.

$$\texttt{VIO}^{(T,Q)}(sel, \leq_n \pi.\varphi) \longrightarrow \texttt{SAT}^{n+1}_\pi(sel, \varphi)$$

$$\mu(LHS) = \{(1 + d(\varphi), c(\varphi))\}$$
$$\mu(RHS) = \{(0, c(\varphi))\}$$

The d component strictly decreases from $1 + d(\varphi)$ to 0. Thus, $\mu(LHS) \gg \mu(RHS)$.

Since every rewrite rule strictly decreases the measure under the well-founded DM ordering, no infinite rewrite sequence is possible. □

B Proof of Time Complexity

We show that the subshape collection and de-duplication algorithm runs in $O(n_s)$ time, where n_s is the number of subshapes in the finite, non-recursive schema $\mathcal{S}$.

DFS traversal. Since shape definitions can refer to other shapes in $\mathcal{S}$, the constraints in $\mathcal{S}$ form a dependency graph. As we assume $\mathcal{S}$ to be non-recursive, the dependency graph is a directed acyclic graph, allowing us to process the constraints in topological order.

For each constraint $(sel, \varphi) \in \mathcal{S}$ processed in topological order, we build an expansion graph rooted at $\texttt{VIO}^{(T_0,Q_0)}(sel, \varphi)$ and perform a depth-first search (DFS) on the expansion graph to collect all reachable subshapes. To prevent redundant work across the entire schema, a *global visited set* (e.g., a hash set) is maintained. This ensures that each reachable subshape is explored exactly once, even if it is referred to by multiple constraints.

Cost of traversal. Exploring a subshape consists of: (i) adding it to the global visited set, (ii) applying the corresponding rewrite rule(s), and (iii) pushing its immediate subshapes onto the DFS stack. The number of immediate subshapes of any shape is bounded by the grammar (at most two for binary connectives, one for unary operators), so the total number of edge traversals is $O(n_s)$. Thus, the DFS visits n_s nodes and $O(n_s)$ edges, giving $O(n_s)$ time.

De-duplication. After traversal, the set of normal-form violation expressions is de-duplicated using a hash set. Insertion into the hash set is $O(1)$ amortized per expression, and the number of expressions is $O(n_s)$ in the worst case, giving $O(n_s)$ time for this step.

Total time. Topological sorting of the C constraints takes $O(C + n_s)$ time, with $C \leq n_s$. Adding the traversal and de-duplication costs, the total running time is

$$O(C + n_s) + O(n_s) + O(n_s) = O(n_s).$$

Hence, the algorithm runs in time linear in the number of subshapes in $\mathcal{S}$. □

C More Graph Operations

This appendix provides the complete mapping from atomic shapes in normalized violation expressions to graph operations for inducing violations. The atomic shapes can be categorized into node test based (App. C.1) and path expression based (App. C.2).

C.1 Node Test Based Atomic Shapes

We expand on Table 3 in Sec. 5.4.1 by providing the complete mapping from *normalized violation expressions* to graph operations. As described in Sec. 5.4.1, a normalized violation expression $\texttt{VIO}^{(T,Q)}(sel, \varphi)$ targets *final hops* of traces in T for nodes u satisfying the selector sel, and applies modifications so that u no longer satisfies the atomic shape φ.

Table 5 lists all atomic shapes in SHACL supported by our framework, together with the graph operations to induce violations. Recall from Sec. 5.4.4 that if the targeted trace to modify is zero-length and φ is an *intrinsic* shape, the shape violation task is ineligible.

C.2 Path Expression-Based Atomic Shapes

Path expression-based atomic shapes require more elaborate violation strategies, as multiple sets of graph operations may induce a violation.

For each case, we recall the SHACL semantics, assume a normalized violation expression $\texttt{VIO}^{(T,Q)}(sel, \varphi)$ with a node $u \models sel$, and list possible strategies. Recall the notation $\texttt{VIO}_\pi(u, v)$ denotes the path violation task applied to the path expression π from source node u to target node v and the notation $[\![\pi]\!](u) = \{\, v \mid (u, v) \in [\![\pi]\!] \,\}$ denotes the π-targets of u, i.e., all the nodes connected from u via π.

C.2.1 $\varphi = \mathit{eq}(\pi, q)$ A node u satisfies $\varphi = \mathrm{eq}(\pi, q)$ iff $[\![\pi]\!](u) = [\![q]\!](u)$. Possible violation strategies:

- Construct a trace $\tau \models \pi$ such that $src(\tau) = u$ and $tgt(\tau) \notin [\![q]\!](u)$.
- Construct a trace $\tau \models q$ such that $src(\tau) = u$ and $tgt(\tau) \notin [\![\pi]\!](u)$.
- Select $v \in [\![\pi]\!](u)$ and apply $\texttt{VIO}_\pi(u, v)$.
- Select $v \in [\![q]\!](u)$ and apply $\texttt{VIO}_q(u, v)$.

C.2.2 $\varphi = \neg\mathit{eq}(\pi, q)$ A node u satisfies $\neg\mathrm{eq}(\pi, q)$ iff $[\![\pi]\!](u) \neq [\![q]\!](u)$. Possible violation strategies:

- For each $v \in [\![\pi]\!](u) \setminus [\![q]\!](u)$, construct $\tau \models q$ with $src(\tau) = u$ and $tgt(\tau) = v$; and for each $v \in [\![q]\!](u) \setminus [\![\pi]\!](u)$, construct $\tau \models \pi$ with $src(\tau) = u$ and $tgt(\tau) = v$.
- For each $v \in [\![\pi]\!](u) \setminus [\![q]\!](u)$, apply $\texttt{VIO}_\pi(u, v)$; and for each $v \in [\![q]\!](u) \setminus [\![\pi]\!](u)$, apply $\texttt{VIO}_q(u, v)$.

C.2.3 $\varphi = \mathit{disj}(\pi, q)$ A node u satisfies $\mathrm{disj}(\pi, q)$ iff $[\![\pi]\!](u) \cap [\![q]\!](u) = \emptyset$. Possible violation strategies for when $[\![\pi]\!](u) = [\![q]\!](u) = \emptyset$:

- Reuse any node v in the graph that is not u and construct traces τ_1, τ_2 such that $\tau_1 \models \pi$ and $\tau_2 \models q$ with $src(\tau_1) = src(\tau_2) = u$ and $tgt(\tau_1) = tgt(\tau_2) = v$.

Possible violation strategies for when $[\![\pi]\!](u)$ and $[\![q]\!](u)$ are not empty:

- Select $v \in [\![\pi]\!](u)$ and construct $\tau \models q$ with $src(\tau) = u$ and $tgt(\tau) = v$.
- Select $v \in [\![q]\!](u)$ and construct $\tau \models \pi$ with $src(\tau) = u$ and $tgt(\tau) = v$.

C.2.4 $\varphi = \neg\mathit{disj}(\pi, q)$ A node u satisfies $\neg\mathrm{disj}(\pi, q)$ iff $[\![\pi]\!](u) \cap [\![q]\!](u) \neq \emptyset$. Possible violation strategies:

- For each $v \in [\![\pi]\!](u) \cap [\![q]\!](u)$, apply $\texttt{VIO}_\pi(u, v)$ or apply $\texttt{VIO}_q(u, v)$.

Table 5. Graph operations for $\texttt{VIO}^{(T,Q)}(sel, \varphi)$, given a node $u \models sel$ and a zero-length trace $\langle u \rangle$ or the final hop (w, p, u) of a non-zero-length trace in T. "Ineligible" indicates that, by the coverage rules in Sec. 5.4.4, zero-length traces with intrinsic shapes are discarded.

φ	Graph Operations Given zero-length trace $\langle u \rangle$	Graph Operations Given final hop (w, p, u)
$\text{test}(\omega_{\texttt{sh:hasValue},y})$	ineligible	$rep(w, p, u, u')$, where $u' \neq u$
$\neg\text{test}(\omega_{\texttt{sh:hasValue},y})$	ineligible	$rep(w, p, u, y)$
$\text{test}(\omega_{\texttt{sh:class},y})$		$-e(u, \texttt{a}, c)$, for each subclass c of y
$\neg\text{test}(\omega_{\texttt{sh:class},y})$		$+e(u, \texttt{a}, c)$, for some subclass c of y
$\text{test}(\omega_{\texttt{sh:datatype},y})$	ineligible	$rep(w, p, u, z')$, where z' is not of datatype y
$\neg\text{test}(\omega_{\texttt{sh:datatype},y})$	ineligible	$rep(w, p, u, z')$, where z' is of datatype y
$\text{test}(\omega_{\texttt{sh:nodeKind},y})$	ineligible	$rep(w, p, u, z')$, where z' is not of node kind y
$\neg\text{test}(\omega_{\texttt{sh:nodeKind},y})$	ineligible	$rep(w, p, u, z')$, where z' is of node kind y
$\text{test}(\omega_{\texttt{sh:minExclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' \leq y$
$\neg\text{test}(\omega_{\texttt{sh:minExclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' > y$
$\text{test}(\omega_{\texttt{sh:maxExclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' \geq y$
$\neg\text{test}(\omega_{\texttt{sh:maxExclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' < y$
$\text{test}(\omega_{\texttt{sh:minInclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' < y$
$\neg\text{test}(\omega_{\texttt{sh:minInclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' \geq y$
$\text{test}(\omega_{\texttt{sh:maxInclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' > y$
$\neg\text{test}(\omega_{\texttt{sh:maxInclusive},y})$	ineligible	$rep(w, p, u, v')$, where $v' \leq y$
$\text{test}(\omega_{\texttt{sh:minLength},y})$	ineligible	$rep(w, p, u, u')$, where $len(u') < y$
$\neg\text{test}(\omega_{\texttt{sh:minLength},y})$	ineligible	$rep(w, p, u, u')$, where $len(u') \geq y$
$\text{test}(\omega_{\texttt{sh:maxLength},y})$	ineligible	$rep(w, p, u, u')$, where $len(u') > y$
$\neg\text{test}(\omega_{\texttt{sh:maxLength},y})$	ineligible	$rep(w, p, u, u')$, where $len(u') \leq y$
$\text{test}(\omega_{\texttt{sh:pattern},y})$	ineligible	$rep(w, p, u, u')$, where u' does not match y
$\neg\text{test}(\omega_{\texttt{sh:pattern},y})$	ineligible	$rep(w, p, u, x')$, where x' matches y
$\text{test}(\omega_{\texttt{sh:languageIn},y})$		$-e(u, @, y)$
$\neg\text{test}(\omega_{\texttt{sh:languageIn},y})$		$+e(u, @, y)$

u': value generated by editing u (adding/removing/transposing characters, etc.) to meet the violation condition.
v': numeric value randomly sampled to meet the violation condition.
z': value obtained by casting u to a different datatype or node kind.
x': string generated by a regex-based generator[57] to match a given pattern.
"@": special predicate for language tags.

C.2.5 $\varphi = \mathit{lessThan}(\pi, q)$ A node u satisfies lessThan(π, q) iff for all $v \in [\![\pi]\!](u)$ and $w \in [\![q]\!](u)$, $v < w$. Possible violation strategies:

- Select $w \in [\![q]\!](u)$ and construct $\tau \models \pi$ with $src(\tau) = u$ and $tgt(\tau) \geq w$.
- Select $v \in [\![\pi]\!](u)$ and construct $\tau \models q$ with $src(\tau) = u$ and $tgt(\tau) \leq v$.

C.2.6 $\varphi = \neg\mathit{lessThan}(\pi, q)$ A node u satisfies ¬lessThan(π, q) iff for all $v \in [\![\pi]\!](u)$ and $w \in [\![q]\!](u)$, $v \geq w$. Possible violation strategies for when $[\![\pi]\!](u) = [\![q]\!](u) = \emptyset$:

- Sample two nodes v, w such that $v < w$ and construct traces τ_1, τ_2 such that $\tau_1 \models \pi$ and $\tau_2 \models q$ with $src(\tau_1) = src(\tau_2) = u$ and $tgt(\tau_1) = v, tgt(\tau_2) = w$.

Possible violation strategies for when there exist $v \in [\![\pi]\!](u)$, $w \in [\![q]\!](u)$, and $v \geq w$.

- To all such v and w, apply $\texttt{VIO}_\pi(u, v)$ or apply $\texttt{VIO}_q(u, w)$.

C.2.7 $\varphi = \mathit{lessThanEq}(\pi, q)$ *and* $\neg\mathit{lessThanEq}(\pi, q)$ These are analogous to lessThan and ¬lessThan, except that equality is permitted in the constraint. Violation strategies are obtained by replacing the strict inequality with a non-strict one in the above cases.

C.2.8 $\varphi = \mathit{uniqueLang}(\pi)$ A node u satisfies uniqueLang(π) iff all distinct $v, w \in [\![\pi]\!](u)$ have different language tags. To violate this constraint (i.e., to introduce at least one pair of literals with the same language tag), possible strategies for when $[\![\pi]\!](u) = \emptyset$:

- Reuse nodes v, w with different language tags in the graph and construct a traces $\tau_1, \tau_2 \models \pi$ with $src(\tau_1) = src(\tau_2) = u$ and $tgt(\tau_1) = v, tgt(\tau_2) = w$.

Possible strategies for when $|[\![\pi]\!](u)| = 1$:

- Select $v \in [\![\pi]\!](u)$ and construct a trace $\tau \models \pi$ with $src(\tau) = u$ and $tgt(\tau) = v'$, where v' has the same language tag as v.

Possible strategies for when $|[\![\pi]\!](u)| > 1$:

- Select distinct $v, w \in [\![\pi]\!](u)$ and modify the language tag of v so that it matches the tag of w.

C.2.9 $\varphi = \neg$*uniqueLang*(π) A node u satisfies $\neg\text{uniqueLang}(\pi)$ iff there exist distinct $v, w \in [\![\pi]\!](u)$ with the same language tag. To violate this constraint (i.e., to make all language tags distinct), possible strategies include:

- For each pair (v, w) with the same language tag, apply $\texttt{VIO}_\pi(u)$ or $\texttt{VIO}_\pi(v)$.
- For each pair (v, w) with the same language tag, modify the language tag of u so that it differs from w.

App. G details how to create a π-matching trace connecting two nodes.

D Proof of Coverage Guarantee

In this appendix, we provide the proof of our coverage guarantee. Recall from Sec. 5 that:

- A shape φ is *reachable* if during the rewriting process of the ARS $\mathcal{A} = (\texttt{Expr}, \to)$, a shape violation task $\texttt{VIO}^{(T,Q)}(sel, \varphi)$ is generated.
- The ARS is *strongly normalizing*, meaning every rewriting sequence terminates in a finite number of steps in a normal form.
- A shape violation task $\texttt{VIO}^{(T,Q)}(sel, \varphi)$ is *eligible* if it satisfies both the *logical precondition* and *operational feasibility*.
- A path violation task $\texttt{VIO}_\pi(u, v)$ is *eligible* if it satisfies *operational feasibility*.
- A qualified satisfaction task $\texttt{SAT}^n_\pi(sel, \varphi)$ is *eligible* if it satisfies the *logical precondition*.

We now restate the theorem.

Theorem 2. Coverage Guarantee. For any schema $\mathcal{S}$ and valid graph $\mathcal{G}$, every reachable and eligible normalized shape violation task, every eligible path violation task, and every eligible qualified satisfaction task will generate a non-empty set of graph operations.

Proof. We prove the statement for (i) normalized shape violation tasks $\texttt{VIO}^{(T,Q)}(sel, \varphi)$, (ii) path violation tasks $\texttt{VIO}_\pi(u, v)$, and (iii) qualified satisfaction task $\texttt{SAT}^n_\pi(sel, \varphi)$.

Shape violation tasks. Let φ be a reachable shape in $\mathcal{S}$ with an eligible normalized violation task $\texttt{VIO}^{(T,Q)}(sel, \varphi)$. We proceed by showing reachability, termination, and non-emptiness of the operation set:

1. **Reachability.** By the construction of the ARS, every reachable shape appears as an argument of some violation task during the rewriting of the initial expression $\texttt{VIO}^{(T_0,Q_0)}(sel_0, \varphi_0)$ for the original constraint (sel_0, φ_0).
2. **Termination.** By the definition of normalized task, there is no applicable rewrite rule for $\texttt{VIO}^{(T,Q)}(sel, \varphi)$. The ARS terminates and φ is atomic.
3. **Non-emptiness of operations.** By the definition of eligibility (Def. 10), the mapping from $\texttt{VIO}^{(T,Q)}(sel, \varphi)$ to graph operations is non-empty.

Path violations tasks. Let $\texttt{VIO}_\pi(u, v)$ be an eligible path violation task. By the definition of eligibility (Def. 10), there is no zero-length trace between u and v matching π. The path-violation algorithm in Sec. 5.4.2 recursively decomposes π until reaching the base case p, where the match exists because (u, p, v) is present. Removing this hop via $-e(u, p, v)$ (for $p \in \mathcal{E}$) or $-a(u, p)$ (for $p \in \mathcal{K}$) invalidates the trace. Since eligibility rules out degenerate cases, at least one such essential hop exists for every trace in $\Gamma(u, \pi, v)$, and removing it yields a non-empty set of graph operations.

Qualified satisfaction tasks. Let $\texttt{SAT}^n_\pi(sel, \varphi)$ be an eligible qualified satisfaction task. By the definition of eligibility (Def. 10), there is at least one node $u \models sel$ and φ is not a contradiction. The path construction procedure outlined in App. G recursively constructs a trace τ matching π. Since eligibility rules out cases where there does not exist a node v that can satisfy φ. Connecting u with a node v that satisfies φ ($v \models \varphi$) yields a non-empty set of graph operations. □

E Details on Path Violation Tasks

This appendix complements Sec. 5.4.2 by detailing how to remove all π-matching traces for the remaining path expression forms: (π_1^-), $(\pi_1 \cdot \pi_2)$, (π_1^*), and $(\pi_1?)$. The procedure follows the same principle as outlined in Sec. 5.4.2: for each trace τ that matches π with $src(\tau) = u$ and $tgt(v)$, identify an essential hop matching π and remove it. Repeat until no π-matching trace remains.

As noted in Sec. 5.4.4, some cases are *operationally infeasible* (ineligible). This occurs when there exists a zero-length trace $\langle u \rangle$ matching π. In the following, we assume eligible tasks.

- **Inverse** (π_1^-): a trace τ matches π_1^- from nodes u to v if a trace matching π_1 exists from v to u. To violate this, the algorithm recursively calls itself on the inverted case (v, π_1, u).
- **Sequence** $(\pi_1 \cdot \pi_2)$: a trace τ matches $(\pi_1 \cdot \pi_2)$ from nodes u to v if there exists some node w such that a π_1-matching sub-trace exists from u to w and a π_2-matching sub-trace exists from w to v. To violates this, we must invalidate at least one of these two sub-traces. Therefore, the algorithm recursively calls itself on either (u, π_1, w) or (w, π_2, v).
- **Zero-or-more repetitions** (π_1^*): a non-zero-length trace τ matches π_1^* from nodes u to v if there exists nodes $w_1, w_2, \cdots$, such that $(u, \pi_1, w_1, \pi_1, \cdots, \pi_1, v)$. It is sufficient to invalidate the first segment (u, π_1, w_1) to invalidate the entire trace. Therefore, the algorithm recursively calls itself on (u, π_1, w_1).
- **Zero-or-one** $(\pi_1?)$: this is a special case of π_1^*. A non-zero-length trace τ matches $(\pi_1?)$ from nodes u to v if a trace matching π_1 exists from u to v. To violate this, the algorithm recursively calls itself on (u, π_1, v).

Termination. In all eligible cases, the process eventually reaches the *base case* p, where a trace τ matches

p because (u, p, v) is present. Removing this hop via $-e(u, p, v)$ (for $p \in \mathcal{E}$) or $-a(u, p)$ (for $p \in \mathcal{K}$) invalidates the trace.

F Details on Qualified Satisfaction Tasks

This appendix elaborates on the procedure for $\texttt{SAT}^n_\pi(sel, \varphi)$ tasks described in Sec. 5.4.3. The objective is to ensure that a node $u \models sel$ has exactly n distinct qualified π-targets that satisfy φ. Let m denote the current number of such targets. The goal is to add $k = n - m$ new targets that satisfy φ. This is achieved in two steps: First, find qualified targets. Second, construct π-matching traces. The details are as follows.

Candidate qualified targets are located using the *class-aware subgraph monomorphism search* proposed by Fierro et al.[19]. The shape φ is first converted into its graph representation $\mathcal{G}_\varphi$. A search is then performed over $\mathcal{G}$ to retrieve the largest subgraphs that are monomorphic to subgraphs of $\mathcal{G}_\varphi$. Each retrieved subgraph corresponds to a portion of $\mathcal{G}$ that satisfies either φ in full or a proper subshape of φ.

When a retrieved subgraph $\bar{\mathcal{G}}(\bar{\mathcal{E}}, \bar{\rho}) \subseteq \mathcal{G}(\mathcal{E}, \rho)$ is fully monomorphic to $\mathcal{G}_\varphi$, there exists a node $v \in \bar{\mathcal{G}}$ such that $v \models \varphi$. In this case, a trace τ matching π ($\tau \models \pi$) is constructed, with $src(\tau) = u$ and $tgt(\tau) = v$, and added to $\mathcal{G}$.

If the retrieved subgraph is monomorphic only to a proper subgraph of $\mathcal{G}_\varphi$, the corresponding node v satisfies a proper subshape of φ. To make v satisfy φ, the missing entities and relationships are generated using an LLM. This is because symbolic random string generators will create entities that are too dissimilar from the existing entities (e.g., `"new_0585"`). Once v satisfies φ, a π-matching trace from u to v is created as in the full-match case.

This process is repeated until k distinct π-targets satisfying φ have been obtained.

In this way, the subgraph monomorphism search maximizes reuse of existing graph elements, resorting to synthetic entity creation only when necessary, thereby satisfying the $\texttt{SAT}^n_\pi(sel, \varphi)$ requirement while minimizing changes to the original graph.

Running Example. Consider the task

$$\texttt{SAT}^4_{\texttt{reviewedBy}}(\in \texttt{Paper}, \geq_1 \pi.\texttt{len} \geq 3)$$

from our running example in Fig. 4c. Let $\varphi = (\geq_1 \pi.\texttt{len} \geq 3)$, a node of type `Paper` is required to have exactly four distinct `reviewedBy` targets that satisfy φ.

From the set {`PaperABC`, `PaperA`}, both of which satisfy $\in$ `Paper`, we randomly select `PaperABC` as the subject node u. The current set of qualified π-targets for u is {`Ada`, `Bob`}, giving $m = 2$. Since $n = 4$, we require $k = 2$ additional targets. Applying the subgraph monomorphism search to $\mathcal{G}$, we retrieve a node `Dan` that satisfies φ and is not already in the target set. After adding a `reviewedBy` edge from `PaperABC` to `Dan` is added, the remaining requirement is reduced to $k = 1$. The search then identifies `Cole`, which matches only a proper subshape of φ. To make `Cole` satisfy φ, we prompt an LLM as illustrated in Fig. 9 to generate the missing entities. Once `Cole` satisfies φ, a `reviewedBy` edge from `PaperABC` to `Cole` is added, completing the set of four qualified targets. App. G details how to create a π-matching trace connecting two nodes.

Generate an entity name to replace `urn:___param___#p1` in the following graph.
`ex:Cole workPhone urn:___param___#p1.`
Your answer must be semantically similar to the corresponding entity in the following example but not exactly identical.
`ex:Dan workPhone "789".`
Compare with the example, observe if `urn:___param___#p1` should be a URIRef. If so, make it a valid URIRef. Otherwise, make it a Literal. Return your answer in json without explanations. {"answer":your answer, "is URIRef":true/false}.

Figure 9. Example prompt for qualified satisfaction task.

G Path Construction Tasks

This appendix describes the procedure for constructing a π-matching trace between two nodes u and v. This procedure is used in path expression-based atomic shape violation tasks (App. C.2) and qualified satisfaction tasks (App. F).

The construction process is a recursion on the structure of π. Given two endpoints, nodes u and v, and a path expression π, the algorithm determines the structural requirements for a match and adds the necessary edges or attributes. The recursion terminates when a *base case* is reached:

- **Base case, predicate** p: A trace from node u to v matches p if (u, p, v) is present. If $p \in \mathcal{E}$, add the edge $+e(u, p, v)$; if $p \in \mathcal{K}$, add the attribute $+a(u, p)$.
- **Alternative** $(\pi_1 \cup \pi_2)$: A trace from node u to v matches $(\pi_1 \cup \pi_2)$ if it matches either π_1 or π_2. To construct a trace matching this path expression, we can choose either possibility. Thus, the algorithm recursively calls itself on (u, π_1, v) or (u, π_1, v).
- **Sequence** $(\pi_1 \cdot \pi_2)$: A trace from node u to v matches $(\pi_1 \cdot \pi_2)$ if there exists some node w such that a π_1-matching trace exists from u to w and a π_2-matching trace exists from w to v. To construct a trace matching this path expression, we must select an intermediate node w, then the algorithm recursively calls itself on (u, π_1, w) and (w, π_2, v).
- **Inverse** (π_1^-): A trace from node u to v matches (π_1^-) if the trace from v to u matches π. To construct a trace matching this path expression, the algorithm recursively calls itself on (v, π_1, u).
- **Zero-or-more repetitions** (π_1^*): A trace from node u to v matches (π_1^*) trivially via the zero-length trace $\langle u \rangle$ if $u = v$. If $u \neq v$, a trace from node u to v matches (π_1^*) if there exists nodes $w_1, w_2, \cdots$, such that $(u, \pi_1, w_1, \pi_1, \cdots, \pi_1, v)$. To construct a trace matching this path expression, we select a set of intermediate nodes $w_1, w_2, \cdots, w_m$, and the

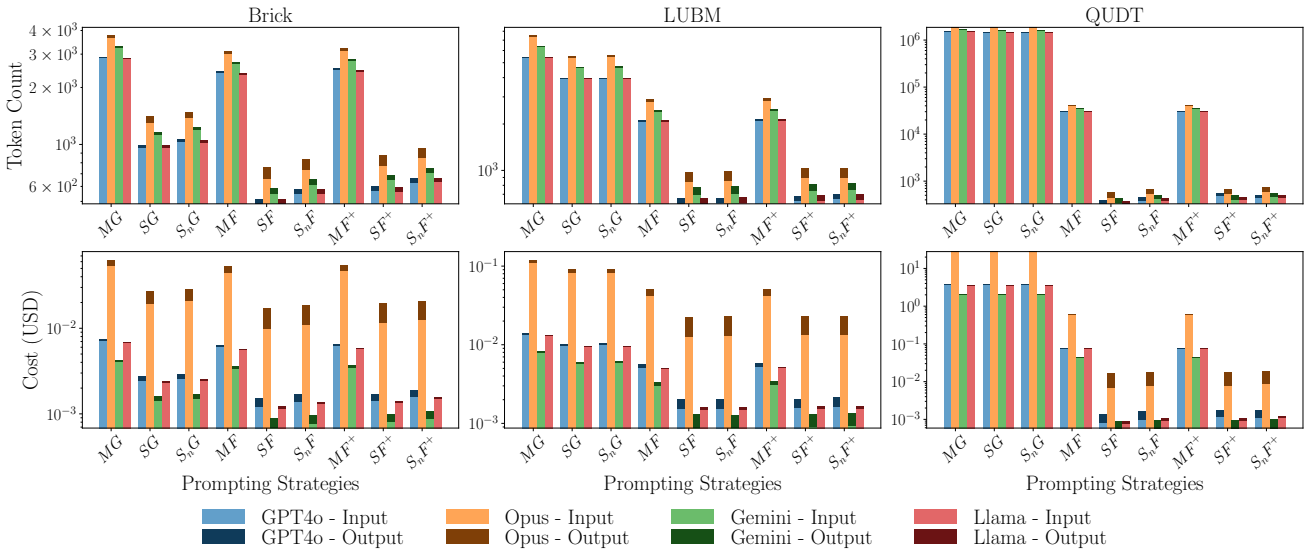


Figure 10. Average token count and cost for different prompting strategies and LLMs.

algorithm recursively calls itself on each consecutive pair $(u, \pi_1, w_1), (w_m, \pi_1, v)$, and $(w_i, \pi_1, w_{i+1}), \forall i \in \{1, \cdots, m-1\}$.

- **Zero-or-one** $(\pi_1?)$: This is a special case of π_1^*. A trace from node u to v matches $(\pi_1?)$ trivially via the zero-length trace $\langle u \rangle$ if $u = v$. If $u \neq v$, The algorithm recursively calls itself on (u, π_1, v).

Termination. The process eventually reaches the base case p, where the match can be realized by adding (u, p, v) via $+e(u, p, v)$ (for $p \in \mathcal{E}$) or $+a(u, p)$ (for $p \in \mathcal{K}$).

H LLM-based Repair System Details

This appendix provides more details on the implementation and results of LLM-based repair. The LLMs evaluated are GPT4o (OpenAI), Claude 3.0 Opus (Anthropic), Gemini 1.5 Pro (Google), and Llama 3.1 405B (Meta, via AWS Bedrock). All models were accessed in January 2025. The pricing structures are listed in Table 6.

As stated in Sec. 6, we use a modular prompt template (Fig. 11) to perform an ablation study. There are three configurations for both the schema context $\mathcal{C}_m$ and the graph context $\mathcal{C}_g$. They are listed here for convenience.
Schema Context $\mathcal{C}_m$.

1. **Entire Schema** (M): Provides the full schema $\mathcal{S}$.
2. **Minimal Schema** (S): Provides only the offended constraint (sel, φ).
3. **Annotated Schema** (S_n): Extends S with natural language descriptions (e.g., `rdfs:label`).

Graph Context $\mathcal{C}_g$.

1. **Entire Graph** (G): Provides the full graph $\mathcal{G}'$.
2. **Focused Subgraph** (F): Provides only the offending subgraph $\mathsf{OffSub}_{\mathcal{G}'}(u, (sel, \varphi))$ of offending focus u and offended constraint (sel, φ). For cardinality constraints, $\varphi = \geq_n \pi.\varphi'$, we also include context around nodes that satisfy φ'.
3. **Focused Subgraph with Example** (F^+): Augments F with a positive example.

Table 6. Pricing Model. Costs are in USD per 1 million tokens.

Model	Input Cost	Output Cost
GPT4o	\$2.5	\$10
Claude 3.0 Opus	\$15	\$75
Gemini 1.5 Pro	\$1.25	\$5
Llama 3.1 405B	\$2.4	\$2.4

Fig. 10 reports average input/output token counts and associated costs. Due to budget constraints, prompts that cost more than \$0.50 are omitted. These exclusions only occur on the large-scale QUDT dataset and affect the following prompting strategies:

- The MG, SG, and S_nG prompts for all LLMs.
- The MF and MF^+ prompts for Claude 3.0 Opus, whose granular tokenization results in higher costs.

Therefore, the output token counts and costs are omitted in such cases in Fig. 10. As expected, using the entire schema and graph (MG) yields the highest input token counts and costs across all datasets. Adding natural language descriptions $(S_n$ vs. $S)$ increases input size only slightly for LUBM, due to the fact that such annotations are sparse in the schema. With the exact same prompt, Claude 3.0 Opus shows the highest counts and costs among LLMs, reflecting its more granular tokenization.

Subsequently, in Fig. 12, we present the repair quality of each combination of prompting strategy and LLM, evaluated using the evaluation metrics outlined in Sec. 7. The gray entries are prompts that exceed our \$0.5 budget. Finally, we present the full results for all repair methods (answer set programming based, supervised learning based, and LLM based) in Table 7.

Primer
You are an expert in repairing RDF graphs that violate SHACL shapes. Output a SPARQL operation that fixes the violation.

Schema Context
Focus Node: `<urn:bldg/ahu>`
Violated source SHACL shape:
```
[] sh:path brick:feeds;
   sh:qualifiedMinCount 1;
   sh:qualifiedValueShape [ sh:node
<urn:co2_zone> ].
```
Reason: qualifiedMinCount is violated
```
<urn:co2_zone> a sh:NodeShape;
   sh:property [ sh:path brick:hasPoint; ... ]
```

Graph Context
RDF knowledge graph:
```
<urn:bldg/ahu> a brick:Air_Handling_Unit;
   brick:feeds <urn:bldg/zone1> ;
   brick:hasPoint <urn:bldg/outside_co2_sensor>.
...
```

Instructions
Use "INSERT DATA { }", "DELETE DATA { }", or "DELETE { } INSERT { } WHERE { }" to fix the violation with minimal change that is contextually appropriate. Invent placeholder names or remove existing instances only if necessary. Do not use nested curly brackets.
Respond only with valid json format using key "answer" without explanations. For example, {"answer":"INSERT DATA ..."} or {"answer":"DELETE DATA {...}"} or {"answer":"DELETE {...} INSERT {...} WHERE {...}"}.

Figure 11. Prompt template for our LLM-based repair systems

Table 7. Average scores (%) of all repair systems on all datasets.

Method	Syn. Val.	Sem. Val.	Iso.	Iso. Hit	Iso. Prec.
ASP[17]	**100.0**	40.23	13.51	27.59	12.13
Neural[22]	**100.0**	10.37	8.36	-	-
Subset Avg	96.96	87.95	52.41	-	-
Llama+SF^+	96.84	**91.38**	**58.62**	-	-

I Details on the Statistical Analysis

In the statistical analysis in Sec. 8.1, we seek to determine whether using different contexts or LLMs produces different results. In the following, we take the example of investigating whether using the schema context M differs from S to illustrate.

Our data are structured such that multiple measurements are taken across various datasets, LLMs, and evaluation metrics. This leads to repeated measures and correlated observations within each grouping factor (e.g., measurements from the same LLM are not independent). Standard linear models assume that all observations are independent[58], which is violated in this setting. Linear mixed-effects models (LMMs) are designed to address this issue: they allow us to model both the systematic effects of our main variables of interest (fixed effects) and the random variability due to repeated measurements within groups (random effects). This provides more accurate estimates and valid statistical inference in the presence of hierarchical data.

A linear mixed-effects model combines two types of effects:

1. Fixed effects capture the average influence of variables of primary interest that are consistent across all observations. In our example, the fixed effect is the schema context M vs. S, where M is the reference and S is the comparison.

2. Random effects account for random variation attributable to grouping factors, such as datasets, LLMs, and metrics. These effects capture the correlation and heterogeneity within groups.

Mathematically, for an outcome y_{ijkl} measured for schema context i, dataset j, LLM k, and metric l, the model can be written as:

$$y_{ijkl} = \beta_0 + \beta_1 \cdot \text{Context}_i + b_j^{(\text{dataset})} + b_k^{(\text{LLM})} + b_l^{(\text{metric})} + \epsilon_{ijkl},$$

where

- β_0 is the intercept (mean outcome for the reference. In our example, M is considered the reference that S will compare with),
- β_1 is the fixed-effect coefficient for the context (difference between M and S),
- $b_j^{(\text{dataset})} \sim \mathcal{N}(0, \sigma^2_{\text{dataset}})$ is the random intercept for dataset j,
- $b_k^{(\text{LLM})} \sim \mathcal{N}(0, \sigma^2_{\text{LLM}})$ is the random intercept for LLM k,
- $b_l^{(\text{metric})} \sim \mathcal{N}(0, \sigma^2_{\text{metric}})$ is the random intercept for metric l,
- $\epsilon_{ijkl} \sim \mathcal{N}(0, \sigma^2)$ is the residual error.

This formulation enables us to estimate the effect of changing the schema context from M to S while controlling for the repeated measures and correlation introduced by datasets, LLMs, and metrics.

The effect of changing the schema context from M to S is given by the estimated fixed-effect coefficient, $\hat{\beta}_1$. This value represents the average difference in the outcome variable when using S instead of M, after accounting for variability due to datasets, LLMs, and metrics. The estimation of the confidence interval for this effect quantifies the uncertainty of the estimate. Its calculation can be found in Pinheiro et al.[55].

A key assumption of linear mixed-effects models is that the residual errors (ϵ_{ijkl}) are normally distributed with mean zero and constant variance. This normality assumption underpins the validity of statistical inference (such as confidence intervals and p-values). To test for normality, we employ the Shapiro-Wilk (SW) test. The null hypothesis of the SW test assumes that the data follows a normal distribution. We set the significance level for the SW test at 0.05. Therefore, if the p-value is greater than 0.05, the residuals are considered to be normally distributed, satisfying the normality

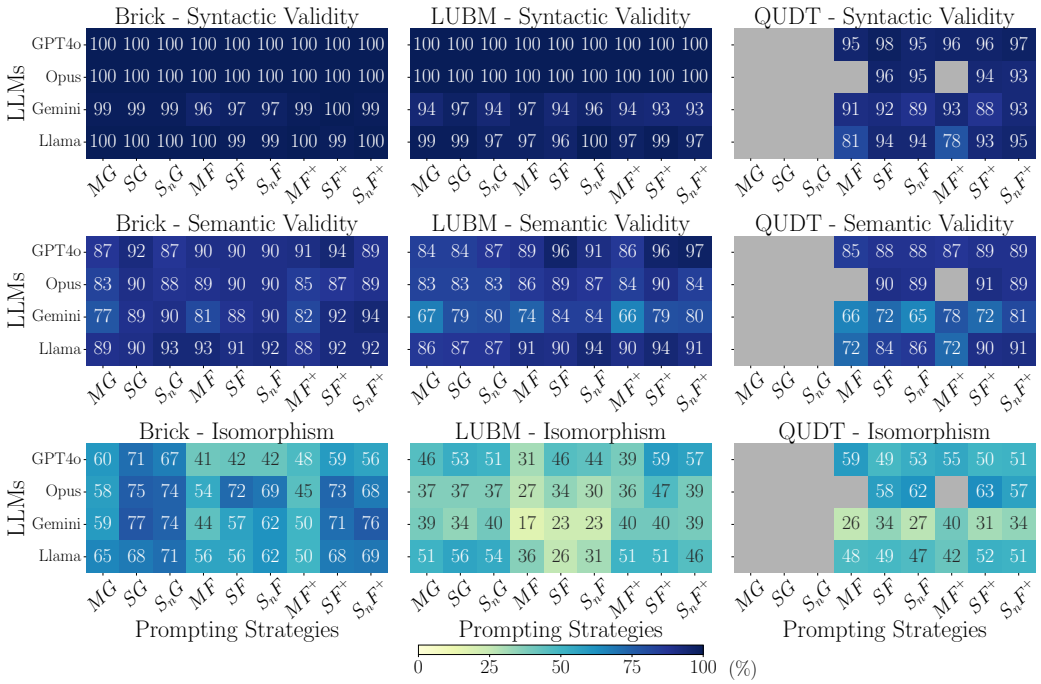


Figure 12. Performance of each evaluation metric for every combination of LLM, strategy, and dataset.

assumption of the LMM. If the residuals do not pass the normality test, we apply various transformations to the raw data and refit the LMM. The transformations considered are:

1. Arcsine Square Root Transformation: $\arcsin\sqrt{x}$.
2. Logit Transformation: $\ln\frac{x+\epsilon}{1-p+\epsilon}$, where $\epsilon = 10^{-6}$ is used to avoid numerical error.
3. Box-Cox Transformation: $\begin{cases}\frac{y^{\lambda}-1}{y}, \text{if } \lambda \neq 0\\ \ln y, \text{ else}\end{cases}$. Here, λ is optimized to best approximate a normal distribution.

If none of the transformations results in normal residuals after refitting the model, we proceed without any transformation and report the SW p-values directly. For any transformation applied, we back-transform the estimated effects to their original scale.

In Tables 8 and 9, we present the raw averages of the reference group and the comparison group. The critical value for both the estimated effect and the Shapiro-Wilk test is set to 0.05. If the p-value of the estimated effect is less than 0.05, it indicates a significant difference between the reference group and the comparison group; otherwise, the cell in the table is shaded with . Similarly, if the p-value of the SW test is greater than 0.05, indicating that the residuals are normally distributed, the cell is not shaded; otherwise, it is shaded with .

Table 8: **Context Selection.** indicates the averages of the reference and comparison group are not considered different. indicates the residuals are not considered normal.

	Treatment	Reference Group	Comparison Group	Reference Group Avg	Comparison Group Avg	Estimated Effect	p-value	Transf-ormation	SW test p-value
Overall	$M \to S$	MG, MF, MF^+	SG, SF, SF^+	68.31%	73.71%	5.28%	4.4×10^{-10}	-	1.8×10^{-1}
	$S \to S_n$	SG, SF, SF^+	S_nG, S_nF, S_nF^+	73.71%	73.41%	-0.30%	6.7×10^{-1}	-	3.8×10^{-7}
	$M \to S_n$	MG, MF, MF^+	S_nG, S_nF, S_nF^+	68.31%	73.41%	4.98%	4.8×10^{-9}	arcsine	5.0×10^{-2}
	$G \to F$	MG, SG, S_nG	MF, SF, S_nF	74.93%	69.37%	-3.97%	2.4×10^{-5}	arcsine	5.6×10^{-2}
	$F \to F^+$	MF, SF, S_nF	MF^+, SF^+, S_nF^+	69.37%	72.31%	2.94%	1.5×10^{-4}	-	8.4×10^{-2}
	$G \to F^+$	MG, SG, S_nG	MF^+, SF^+, S_nF^+	74.93%	72.31%	-0.86%	3.3×10^{-1}	-	4.4×10^{-6}
Syntactic Validity	$M \to S$	MG, MF, MF^+	SG, SF, SF^+	96.87%	97.59%	0.16%	4.5×10^{-1}	box-cox	6.3×10^{-2}
	$S \to S_n$	SG, SF, SF^+	S_nG, S_nF, S_nF^+	97.59%	97.56%	-0.03%	9.1×10^{-1}	-	1.5×10^{-2}
	$M \to S_n$	MG, MF, MF^+	S_nG, S_nF, S_nF^+	96.87%	97.56%	0.96%	1.1×10^{-1}	-	3.5×10^{-8}
	$G \to F$	MG, SG, S_nG	MF, SF, S_nF	99.04%	96.80%	-0.42%	4.3×10^{-1}	-	2.2×10^{-9}
	$F \to F^+$	MF, SF, S_nF	MF^+, SF^+, S_nF^+	96.80%	96.74%	-0.06%	9.2×10^{-1}	-	1.0×10^{-9}
	$G \to F^+$	MG, SG, S_nG	MF^+, SF^+, S_nF^+	99.04%	96.74%	-0.35%	5.7×10^{-1}	-	2.0×10^{-11}
Semantic Validity	$M \to S$	MG, MF, MF^+	SG, SF, SF^+	82.70%	87.87%	5.04%	2.4×10^{-7}	-	5.8×10^{-1}
	$S \to S_n$	SG, SF, SF^+	S_nG, S_nF, S_nF^+	87.87%	87.72%	-0.16%	8.4×10^{-1}	-	2.8×10^{-1}
	$M \to S_n$	MG, MF, MF^+	S_nG, S_nF, S_nF^+	82.70%	87.72%	4.97%	4.0×10^{-6}	-	3.6×10^{-1}
	$G \to F$	MG, SG, S_nG	MF, SF, S_nF	85.21%	86.11%	3.59%	5.8×10^{-4}	arcsine	9.8×10^{-2}
	$F \to F^+$	MF, SF, S_nF	MF^+, SF^+, S_nF^+	86.11%	86.89%	0.90%	3.4×10^{-1}	logit	5.7×10^{-1}
	$G \to F^+$	MG, SG, S_nG	MF^+, SF^+, S_nF^+	85.21%	86.89%	3.41%	1.3×10^{-2}	logit	6.2×10^{-1}
Isomorphism	$M \to S$	MG, MF, MF^+	SG, SF, SF^+	45.00%	52.53%	7.34%	1.4×10^{-3}	-	1.4×10^{-1}
	$S \to S_n$	SG, SF, SF^+	S_nG, S_nF, S_nF^+	52.53%	51.97%	-0.56%	7.6×10^{-1}	-	6.4×10^{-2}
	$M \to S_n$	MG, MF, MF^+	S_nG, S_nF, S_nF^+	45.00%	51.97%	7.28%	1.7×10^{-3}	logit	6.9×10^{-2}
	$G \to F$	MG, SG, S_nG	MF, SF, S_nF	56.42%	43.91%	-13.67%	3.4×10^{-14}	-	7.8×10^{-1}
	$F \to F^+$	MF, SF, S_nF	MF^+, SF^+, S_nF^+	43.91%	51.51%	7.60%	2.2×10^{-5}	-	1.7×10^{-1}
	$G \to F^+$	MG, SG, S_nG	MF^+, SF^+, S_nF^+	56.42%	51.51%	-3.27%	1.2×10^{-1}	-	1.3×10^{-1}

Table 9: **LLM Selection** indicates the averages of the reference and comparison group are not considered different. indicates the residuals are not considered normal.

	Treatment	Reference Group Avg	Comparison Group Avg	Estimated Effect	p-value	Transformation	SW test p-value
Overall	GPT4o →Llama 3.1 405B	73.32%	73.52%	0.20%	8.1×10^{-1}	-	2.9×10^{-6}
	GPT4o →Claude 3.0 Opus	73.32%	73.47%	-0.21%	2.9×10^{-1}	logit	5.7×10^{-2}
	GPT4o →Gemini 1.5 Pro	73.32%	67.35%	-8.60%	1.0×10^{-15}	arcsine	1.8×10^{-1}
	Llama 3.1 405B →Claude 3.0 Opus	73.52%	73.47%	-0.89%	2.9×10^{-1}	-	7.3×10^{-1}
	Llama 3.1 405B →Gemini 1.5 Pro	73.52%	67.35%	-6.17%	7.0×10^{-12}	-	5.5×10^{-1}
	Claude 3.0 Opus →Gemini 1.5 Pro	73.47%	67.35%	-5.43%	8.6×10^{-8}	-	5.7×10^{-1}
Syntactic Validity	GPT4o →Llama 3.1 405B	99.04%	96.38%	-1.33%	3.2×10^{-9}	box-cox	7.2×10^{-2}
	GPT4o →Claude 3.0 Opus	99.04%	99.00%	-0.35%	8.0×10^{-2}	-	9.2×10^{-8}
	GPT4o →Gemini 1.5 Pro	99.04%	95.12%	-3.92%	2.1×10^{-19}	-	8.9×10^{-2}
	Llama 3.1 405B →Claude 3.0 Opus	96.38%	99.00%	1.16%	1.5×10^{-4}	-	4.1×10^{-1}
	Llama 3.1 405B →Gemini 1.5 Pro	96.38%	95.12%	-1.95%	2.6×10^{-5}	box-cox	4.0×10^{-1}
	Claude 3.0 Opus →Gemini 1.5 Pro	99.00%	95.12%	-3.51%	3.2×10^{-15}	-	2.8×10^{-2}
Semantic Validity	GPT4o →Llama 3.1 405B	89.42%	88.54%	-0.77%	4.3×10^{-1}	arcsine	1.7×10^{-1}
	GPT4o →Claude 3.0 Opus	89.42%	87.23%	-2.30%	2.9×10^{-3}	-	1.4×10^{-1}
	GPT4o →Gemini 1.5 Pro	89.42%	79.58%	-9.83%	2.4×10^{-10}	-	7.5×10^{-1}
	Llama 3.1 405B →Claude 3.0 Opus	88.54%	87.23%	-2.53%	4.8×10^{-4}	-	9.6×10^{-1}
	Llama 3.1 405B →Gemini 1.5 Pro	88.54%	79.58%	-8.96%	7.6×10^{-9}	-	1.9×10^{-1}
	Claude 3.0 Opus →Gemini 1.5 Pro	87.23%	79.58%	-7.40%	1.2×10^{-5}	-	2.1×10^{-1}
Isomorphism	GPT4o →Llama 3.1 405B	51.21%	52.33%	1.26%	5.4×10^{-1}	box-cox	5.4×10^{-2}
	GPT4o →Claude 3.0 Opus	51.21%	52.36%	1.50%	5.9×10^{-1}	-	2.7×10^{-1}
	GPT4o →Gemini 1.5 Pro	51.21%	44.04%	-7.17%	2.1×10^{-2}	-	4.8×10^{-1}
	Llama 3.1 405B →Claude 3.0 Opus	52.33%	52.36%	-0.19%	9.3×10^{-1}	-	7.4×10^{-2}
	Llama 3.1 405B →Gemini 1.5 Pro	52.33%	44.04%	-8.29%	1.9×10^{-5}	-	4.2×10^{-1}
	Claude 3.0 Opus →Gemini 1.5 Pro	52.36%	44.04%	-7.75%	2.2×10^{-3}	-	6.7×10^{-1}