

Systematic Evaluation of Knowledge Graph Repair with Large Language Models

Tung-Wei Lin^{1,2}, Gabe Fierro^{3,4}, Han Li², Tianzhen Hong², Pierluigi Nuzzo¹,
and Alberto Sangiovanni-Vincentelli¹

¹ UC Berkeley, USA

{twlin, nuzzo, alberto}@eecs.berkeley.edu,

² Lawrence Berkeley National Laboratory, USA

{hanli, THong}@lbl.gov

³ National Renewable Energy Laboratory, USA

⁴ Colorado School of Mines, USA

gtfierro@mines.edu

Abstract. We present a systematic approach for evaluating the quality of knowledge graph repairs with respect to constraint violations defined in shapes constraint language (SHACL). Current evaluation methods rely on *ad hoc* datasets, which limits the rigorous analysis of repair systems in more general settings. Our method addresses this gap by systematically generating violations using a novel mechanism, termed violation-inducing operations (VIOs). We use the proposed evaluation framework to assess a range of repair systems which we build using large language models. We analyze the performance of these systems across different prompting strategies. Results indicate that concise prompts containing both the relevant violated SHACL constraints and key contextual information from the knowledge graph yield the best performance.

Keywords: Shapes Constraint Language · Knowledge Graph Repair · Large Language Models · Brick Models

1 Introduction

Knowledge graphs (KGs) are structured representations of information that encode directed relationships between labeled entities. They excel at representing domain-specific knowledge and serve as machine-readable reference databases for diverse applications, including smart building management [9], genome clustering [47], and general knowledge bases such as Wikipedia [42]. More recently, KGs have also been used for automatic software configuration and remote data access [19,36].

KGs are populated using a variety of techniques, including manual curation [11], automated translation from existing databases, and extraction from unstructured text [10]. Users rely on the correctness and completeness of KGs to support downstream tasks. Machine learning models, including language models, extract facts from KGs for tasks such as query generation [17] and question

answering [45]. Large KGs like Wikidata [42] and YAGO [41] facilitate search by linking real-world entities to their attributes. Cyber-physical KGs, built with ontologies such as Brick [9] and RealEstateCore [24], enable digital twins and automatic configuration of fault detection and control. Reference ontologies like QUDT [15] document physical constants and engineering units. As structured, factual, and symbolic repositories, KGs require rigorous validation to ensure their reliability.

The Shapes Constraint Language (SHACL) [32], recently standardized by W3C [4], enables the specification and validation of Resource Description Framework (RDF) KGs [1] against sets of constraints called *shapes*. The *validation report* obtained after validating a KG against SHACL shapes details which constraints were violated on which KG nodes but provides limited feedback on how to repair these violations. Existing repair methods often combine manual intervention with domain-specific heuristics [44,20,19,5], or depend on access to KG editing history [33,34,16]. The absence of comprehensive benchmarks further complicates the evaluation and advancement of automatic KG repair techniques.

This paper introduces a systematic framework to evaluate KG repair systems using SHACL validation reports. Existing evaluation methods are limited, since they either focus on datasets with editing history, such as Wikidata [42], or small, manually curated benchmarks like the SHACL test suite [22]. The *ad hoc* nature of these datasets limits fine-grained analysis of repair system performance. Instead, we look for accurate characterizations of KG repair systems that can also help understand how these systems behave on new and unseen graphs and constraints. We propose a novel evaluation method that systematically generates SHACL violations through *violation-inducing operations* (VIO), for which correct fixes are known. We can then evaluate repair methods across many different types of violations and repairs on arbitrary KGs.

Using this framework, we investigate the use of *large language models* (LLMs) for KG repair. We argue that LLMs offer three key advantages. First, LLMs embed domain knowledge, potentially enabling better heuristics for repair suggestions. Techniques like retrieval-augmented generation [49] can also incorporate external facts to support repairs. Second, LLMs have demonstrated success in pattern matching [31], suggesting they can synthesize repairs from complex SHACL constraints without the extensive programming required by existing techniques. Third, the multi-step problem-solving capabilities of agentic LLM systems [46] may enable more sophisticated repairs than prior approaches. Our contributions can be summarized as follows:

- We introduce and formalize *violation-inducing operations*, which systematically enumerate all violations on a KG with respect to a set of SHACL shapes.
- We propose a VIO-based method for generating datasets of SHACL violations. The generation method provides a high degree of control over the size and shape, or constraint, coverage for evaluating repair systems.
- We present a generic LLM-based KG repair method and evaluate it on three real-world KGs and four commercial and open-source LLMs.

The proposed framework enables the systematic evaluation of KG repair systems by generating datasets with high degree of control over which violations occur and which repairs are necessary to fix the violations. We can then characterize KG repair systems based on the kinds of repairs they can make and how accurate they are. We can also vary the size of generated datasets to measure the scaling performance of a repair system for metrics like cost and computation time. We demonstrate the use of the framework to analyze the performance of LLM-based KG repair systems by prompting strategy and choice of model.

2 Background and Related Work

KGs are essential for a wide range of applications, but they are expensive to create and maintain. They can be developed through expert-driven manual design [47,9], data mining and representation learning techniques [29,10], automated translation from existing sources [18,37], or a combination of these methods. Regardless of how they are constructed, KGs require post-processing and verification to ensure their correctness and completeness [30]. These qualities are typically enforced through compliance checking against domain-specific constraints, which define permissible KG statements. A *validation process* detects violations of these constraints. In this paper, we focus on methods that repair such violations to restore compliance. In contrast, KG completion aims to predict missing knowledge rather than address constraint violations [28,43,39].

2.1 Shapes Constraint Language (SHACL)

SHACL [32] is a formal specification language to ensure data quality in KGs. A SHACL validation engine such as pySHACL [13] checks whether a KG conforms to a set of specifications, called *manifest*. After validation, the engine produces a report that identifies constraint violations and can be used to guide subsequent repairs. However, these reports typically provide only minimal information, leaving users to determine how best to address the violations. This presents a major challenge, as there are many ways to repair a violation.

Similar to database repair [40], one approach is to simply remove inconsistent data, but this risks discarding valuable information. Other methods [5,6] compute repairs with minimal cardinality to preserve as much information as possible. However, minimal cardinality edits, or other symbolic heuristic methods like BuildingMOTIF [20], are not always sufficient to determine the best repair. A significant reason is that these methods cannot consider information beyond what is representable as symbols in the graph; lexical information and domain information are not available to guide the repair process.

2.2 KG Repair Systems

A KG repair process must enumerate a set of possible repairs given a set of violated constraints. These constraints can be considered individually [44,7] or

in groups [5,20,16]. Methods for generating KG repairs vary in the degree of human or expert knowledge they incorporate, in addition to learning from past repairs or incorporating external reference knowledge. Schimatos [44] implements a user interface for manual repair of the KG, and Pellissier et al. [33] and Fan et al. [16] learn repairs from logs of past repairs, which include both automated and manual repairs. BuildingMOTIF [20] uses expert-derived heuristics to suggest repairs based on domain knowledge in the smart building domain, while Ahmetaj et al. [5] generate sets of possible repairs using answer set programming.

After generating a set of possible repairs, a process must choose which repairs to apply to the KG. Methods based on direct or historical manual input [44,7] have the benefit of human knowledge or user confirmation, which are deemed to lead to high-quality repairs. Automated techniques [5,19,7,34] often rely on heuristics or quantifiable metrics to choose the “best” repair, e.g., by choosing the minimal cardinality fix [5] or custom heuristics to rank fixes based on the amount of information they add to the KG [7,20].

KG repair methods also differ in the kinds of repairs they can generate, especially with respect to whether a repair can include information that is already present in the KG but not in the validation report. Ahmetaj et al. [5] generate fresh values to repair violations of existential constraints that complain about missing values. Others [7,34,44,20] can all suggest repairs that incorporate information that is already in the KG.

Our investigation of LLMs for SHACL-based KG repair addresses the prevalent role of expert-curated heuristics in producing and prioritizing graph repairs. Purely symbolic methods [5] are limited in their ability to assess the appropriateness of heuristics such as minimal cardinality, particularly when lacking access to semantic cues such as naming patterns in the KG. Manual methods [44] cannot scale to large numbers of fixes. The remaining heuristic methods discussed above cannot be proven to generalize to the possible repairs required by a KG, nor can they easily differentiate between symbolically similar but semantically different scenarios. LLMs, by contrast, can memorize domain facts and capture relationships between them, enabling strong performance on different language-based tasks [27,12]. They may act as substitutes for the combinations of heuristics and constraint solvers for KG repair. This paper proposes initial benchmarks and a methodology for evaluating the performance of LLMs on repair tasks.

3 Evaluation Framework for LLM-Based KG Repair

We posit that LLMs can enhance KG repair with natural language processing and domain-specific understanding. LLM-based repair systems could automate SHACL repairs, making SHACL easier for practical use. They could integrate symbolic reasoning components, such as a logic programming solver, with a semantic component that considers domain-specific information. However, research on such KG repair systems hinges on accurate evaluation frameworks with rigorous metrics, a gap we aim to fill.

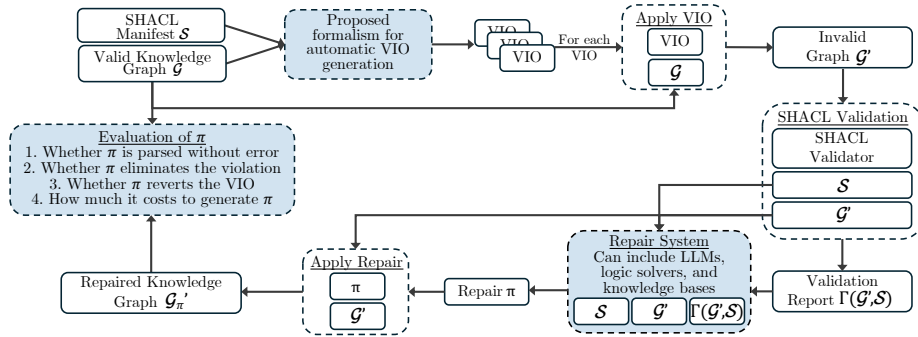


Fig. 1: **The proposed evaluation framework. The shaded blocks are our main contributions.**

In this section, we outline our automated test-case generation methodology for KG repair systems and the corresponding evaluation metrics. As illustrated in Fig. 1, the evaluation framework takes a SHACL manifest \mathcal{S} and a valid KG \mathcal{G} (free of violations) and generates a set of VIOs, represented as SPARQL operations [2], such as addition (`add(\cdot)`) and removal (`remove(\cdot)`) operations. Each VIO is applied on a copy of \mathcal{G} , resulting in an invalid KG \mathcal{G}' . Each pair of \mathcal{S} and \mathcal{G}' constitutes a test case for the repair system. After validating \mathcal{G}' against \mathcal{S} , we obtain a validation report. The report along with \mathcal{G}' and \mathcal{S} is used by the repair system to generate a repair π , represented as `add(\cdot)` and `remove(\cdot)` SPARQL operations. Applying π to \mathcal{G}' yields a repaired KG \mathcal{G}'_{π} . We assess the repair using several metrics: First, π should be parsed without error. Second, π should eliminate the violation, i.e., \mathcal{G}'_{π} should be free of violations when validated against \mathcal{S} . Third, π should revert the VIO and recover the original KG \mathcal{G} . Finally, π should be cost-efficient to generate.

The evaluation framework treats the repair system as a black box, requiring no internal knowledge to generate validation datasets. We showcase the proposed evaluation framework on a family of prototype LLM-based repair systems that we build since existing repair methods either involve human intervention [44,20,19], require editing history [33,34,16] or output sets of repairs [5]. We evaluate several LLM prompt templates to measure the impact of different pieces of information on the performance of the repair system. Our results demonstrate the utility of our evaluation framework for designing repair systems and provide insights into effective prompting strategies for LLM-based SHACL repair.

4 Generation of Violation-Inducing Operations

We begin by introducing key concepts for the VIO generation algorithm, using the running example in Fig. 2. Fig. 2a specifies that papers must be reviewed by at least one and at most three qualified reviewers, defined as professors who are also committee members. Fig. 2b presents a knowledge graph that satisfies these

```

:PaperShape a sh:NodeShape;
sh:targetClass ex:Paper;
sh:property :ReviewedByShape.
:ReviewedByShape a sh:PropertyShape;
sh:path ex:reviewedBy;
sh:qualifiedValueShape :ReviewerShape;
sh:qualifiedMaxCount 3;
sh:qualifiedMinCount 1.
:ReviewerShape a sh:NodeShape;
sh:targetNode ex:Dan;
sh:class ex:Professor,
ex:CommitteeMember.

```

Fig. 2 (a) SHACL manifest \mathcal{S}

```

ex:PaperABC a ex:Paper;
ex:reviewedBy ex:Alice, ex:Bob,
ex:Clark;
ex:author ex:Ethan.
ex:PaperA a ex:Paper;
ex:reviewedBy ex:Alice.
ex:Alice a ex:Professor,
ex:CommitteeMember.
ex:Bob a ex:Professor,
ex:CommitteeMember.
ex:Clark a ex:Student.
ex:Dan a ex:Professor,
ex:CommitteeMember.

```

Fig. 2 (b) Knowledge graph \mathcal{G}

constraints, modeling the relationships between two papers—`ex:PaperABC` and `ex:PaperA`—and their respective reviewers and authors.

4.1 Preliminaries

Definition (Knowledge Graph). A knowledge graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ consists of nodes \mathcal{V} and edges \mathcal{E} . A triple $(u, e, v) \in \mathcal{G}$ iff $u, v \in \mathcal{V}$ and $e \in \mathcal{E}$.

Definition (SHACL Manifest). The SHACL manifest \mathcal{S} consists of two disjoint sets of shapes, namely, node shapes and property shapes. A shape is a tuple, $\Sigma_s = \langle s, \tau_s, \mu_s, \kappa_s \rangle \in \mathcal{S}$, where s is the unique shape name of Σ_s , $\tau_s : \mathcal{G} \rightarrow 2^{\mathcal{V}^5}$ is the targeting function that produces the set of *focuses* to be validated against s . For example, in Fig. 2, $\tau_{\text{PaperShape}} = \{\text{ex:PaperABC}, \text{ex:PaperA}\}$, as declared by the `sh:targetClass` predicate. $\mu_s : \mathcal{V} \rightarrow 2^{\mathcal{V}}$ is the value mapping function, which, for a given node v , returns the set of *values* that are evaluated during the validation of v . For a node shape, $\mu_s(v)$ produces a singleton set with v being the only member. For a property shape, $\mu_s(v)$ produces the set of nodes in \mathcal{G} that can be reached from v with the path mapping p_s of s (i.e., the parameter value of `sh:path`). For example, $\mu_{\text{ReviewedByShape}}(\text{ex:PaperA}) = \{\text{ex:Alice}\}$. $\kappa_s = \{\kappa_s^i = (\xi_s^i, \omega_s^i) \mid i \in I_s\}$ is the set of pairs of SHACL constraints and their parameter values. The permissible data type of ω_s^i depends on the corresponding constraint ξ_s^i . For example, if $\xi_s^i = \text{sh:class}$, then ω_s^i is a class name. $I_s = \{1, \dots, n_s\}$ is the index set for the constraints of s and n_s is the number of constraints in s . For example, $n_{\text{PaperShape}} = 1$ and $\kappa_{\text{PaperShape}} = \{\kappa_{\text{PaperShape}}^1 = (\text{sh:property}, \text{:ReviewedByShape})\}$. On the other hand, ω_s^i is the name of another shape if and only if κ_s^i and ξ_s^i are shape-based. In this case, there exists a shape $\Sigma_{\omega_s^i} = \langle \omega_s^i, \tau_{\omega_s^i}, \mu_{\omega_s^i}, \kappa_{\omega_s^i} \rangle$. If there exists j such that $\kappa_{\omega_s^i}^j$ is shape-based, then $\Sigma_{\omega_s^i}$ and $\Sigma_{\omega_{\omega_s^i}^j}$ are both part of the dependency of Σ_s . For example, $\Sigma_{\text{ReviewedByShape}}$ and $\Sigma_{\text{ReviewerShape}}$ are in the dependency of $\Sigma_{\text{PaperShape}}$.

Definition (Validation Semantics). The SHACL standard defines a validation function $\Gamma(v, \kappa_s^i)$ that takes a node $v \in \mathcal{V}$ and the i -th constraint κ_s^i of s and maps them to a validation result $\rho_{s,v}^i \in \mathcal{R}$, where \mathcal{R} is the set of all possible validation results, \mathcal{R}' , augmented with the empty result, i.e., $\mathcal{R} = \mathcal{R}' \cup \{\emptyset\}$. Γ uses the value mapping function μ_s to find the set of value nodes $\mu_s(v)$ and produce

⁵ $2^{\mathcal{V}}$ denotes the power set of \mathcal{V} , the set of all possible subsets of \mathcal{V}

$\rho_{s,v}^i \neq \emptyset$ if v violates κ_s^i . In that case, s is called the source shape, κ_s^i is called the source constraint, and v is called the focus of the result $\rho_{s,v}^i$. Otherwise, if $\rho_{s,v}^i = \emptyset$, we say that v **satisfies** κ_s^i , denoted by $v \models \kappa_s^i$. Furthermore, we say

$$v \models \Sigma_s \text{ if } \forall i \in I_s, v \models \kappa_s^i \quad \text{and} \quad \mathcal{G} \models \mathcal{S} \text{ if } \forall \Sigma_s \in \mathcal{S}, \forall v \in \tau_s(\mathcal{G}), v \models \Sigma_s.$$

We abuse the notation $\Gamma(\mathcal{G}, \mathcal{S})$ to express the set of validation results of \mathcal{G} against a manifest \mathcal{S} , which is also called the validation report.

Given a KG \mathcal{G} that satisfies a manifest \mathcal{S} , we design an algorithm that generates a set of VIOs. A $\text{VIO}(\kappa_s^i, \mathcal{F})$ takes the i -th constraint of s and a set of focus nodes \mathcal{F} and maps them to a set of SPARQL operations. $\text{VIO}(\kappa_s^i, \mathcal{F})$ is defined to be shape-based if and only if κ_s^i is shape-based.

Running Example. Fig. 2 shows a manifest \mathcal{S} and a KG \mathcal{G} that satisfies \mathcal{S} . VIOs of non-shape-based constraints (such as `sh:class`) correspond to straightforward edits. Handling shape-based constraints such as `sh:property` requires recursively resolving and violating the nested constraints. In the following section, we introduce an abstract rewriting system [26] that expands shape-based VIOs into non shape-based ones.

4.2 Abstract Rewriting System

We define the rewriting objects of the rewriting system $\mathcal{A} = (\text{Expr}, \rightarrow)$.

$$\text{Term} ::= \text{VIO}(\kappa_s^i, \mathcal{F}) \mid \text{Term} \cdot \text{Term}, \quad \text{Expr} ::= \text{Term} \mid \text{Expr} + \text{Term},$$

where \cdot represents simultaneous application (analogous to logical **and**), and $+$ represents alternative application (analogous to logical **or**). The arrow (\rightarrow) denotes a set of rewrite rules that transforms the left-hand object into the right-hand object. An object is said to be in **normal form** (or is normalized) if no rewrite rules can be applied. The system is **strongly normalizing** if every object can be normalized in finitely many steps; that is, every rewriting sequence terminates. Repeated application of the rewrite rules to objects in \mathcal{A} produces an **expansion tree**: each expression corresponds to a node, with leaf nodes representing normalized expressions, and internal nodes representing expressions with shape-based VIOs that can be further rewritten by the rules below. Given $\text{VIO}(\kappa_s^i, \mathcal{F})$, where $\kappa_s^i = (\xi_s^i, \omega_s^i)$, recall that if κ_s^i is shape-based, ω_s^i is a shape name. The specific rewrite rules are provided for different values of ξ_s^i as follows.

Rule 1: $\xi_s^i = \text{sh:node}$ or $\xi_s^i = \text{sh:property}$. A violation is induced if any value node of any focus $f \in \mathcal{F}$ violates any of the $\{1, \dots, n_{\omega_s^i}\}$ constraints of ω_s^i . The first rule is thus defined as follows:

$$\text{Rule 1: } \text{VIO}(\kappa_s^i, \mathcal{F}) \rightarrow \text{VIO}(\kappa_{\omega_s^i}^1, \mathcal{F}') + \dots + \text{VIO}(\kappa_{\omega_s^i}^{n_{\omega_s^i}}, \mathcal{F}'),$$

where $\mathcal{F}' = \{v \mid \forall f \in \mathcal{F}, \forall v \in \mu_s(f)\}$. The plus signs separate the children of $\text{VIO}(\kappa_s^i, \mathcal{F})$ in the expansion tree and each child VIO represents a choice to induce violation on κ_s^i .

Rule 2: $\xi_s^i = \text{sh:qualifiedValueShape}$. In addition to ω_s^i as the parameter value, $\text{sh:qualifiedMaxCount}$ and $\text{sh:qualifiedMinCount}$ are two additional parameters. To violate $\text{sh:qualifiedMaxCount}$, all constraints in the dependency of $\Sigma_{\omega_s^i}$ must be satisfied. Therefore, no rewriting rule is required. We define the corresponding SPARQL operation in App. A. We focus on the case when there exists $h \in I_s$ such that $\xi_s^h = \text{sh:qualifiedMinCount}$ as follows.

We define $\psi_{\omega_s^i}(f) = \{v \in \mu_s(f) \mid v \models \Sigma_{\omega_s^i}\}$, a value mapping function that is stricter than μ_s , in the sense that $\psi_{\omega_s^i}(f)$ only admits nodes $v \in \mu_s(f)$ that satisfy $\Sigma_{\omega_s^i}$. For a focus node $f \in \mathcal{F}$, we must make edits such that $|\psi_{\omega_s^i}(f)|$ becomes strictly smaller than ω_s^h , the parameter value of $\text{sh:qualifiedMinCount}$ that is a positive integer. For a node $v \in \psi_{\omega_s^i}(f)$, there are two strategies to remove v from $\psi_{\omega_s^i}(f)$: (a) Make v violate ω_s^i , or (b) Make v unreachable from f via the path mapping p_s . Therefore, for any subset $\epsilon_f^m \subseteq \psi_{\omega_s^i}(f)$, such that $|\epsilon_f^m| = |\psi_{\omega_s^i}(f)| - \omega_s^h + 1$, we apply a combination of the above two strategies for all nodes $v \in \epsilon_f^m$. We create an artificial node shape Σ_{v_node} and property shape $\Sigma_{f_v_prop}$ to translate the operations into VIOs, where

- $\tau_{v_node}(\mathcal{G}) = \{v\}$, $\mu_{v_node}(v) = \{v\}$, $\kappa_{v_node} = \{(\text{sh:node}, \omega_s^i)\}$ and
- $\tau_{f_v_prop}(\mathcal{G}) = \{f\}$, $\mu_{f_v_prop}(f) = \{v\}$, $\kappa_{f_v_prop} = \{(\text{sh:minCount}, 1)\}$.

In addition, since $\Sigma_{f_v_prop}$ is a property shape, we let the path mapping $p_{f_v_prop} = p_s$. We define $\Phi(\epsilon_f^m) = \left\{g(v) : \epsilon_f^m \rightarrow \{v_node, f_v_prop\}\right\}$, which contains all possible ways to choose “node” and “prop” for each $v \in \epsilon_f^m$. The second rule is defined as follows

$$\text{Rule 2: } \text{VIO}(\kappa_s^i, \mathcal{F}) \rightarrow \sum_{f \in \mathcal{F}} \sum_{\substack{\epsilon_f^m \subseteq \psi_{\omega_s^i}(f) \\ |\epsilon_f^m| = |\psi_{\omega_s^i}(f)| - \omega_s^h + 1}} \sum_{g \in \Phi(\epsilon_f^m)} \prod_{v \in \epsilon_f^m} \text{VIO}(\kappa_{g(v)}^1, \tau_{g(v)}(\mathcal{G})).$$

We implement logical constraints sh:or and sh:and in a similar manner for the evaluation in Sec. 7. We leave sh:xone and sh:not to future work. The following theorem states that, under mild assumptions, \mathcal{A} is strongly normalizing and thus is guaranteed to terminate.

Theorem 1 (Strong Normalization of \mathcal{A}). *If \mathcal{G} and \mathcal{S} are finite and there are no recursive shapes [14], then the rewriting system \mathcal{A} is strongly normalizing.*

Proof. See App. C for the proof.

4.3 Constraint Collection and De-duplication

To systematically evaluate the repair system, we create test cases where each constraint in \mathcal{S} is violated. Assuming \mathcal{S} has no recursive shapes, we first topologically sort its shapes. We build expansion trees and perform a depth-first search (DFS) from each root shape. Each child node in an expansion tree represents an option to violate its parents. Thus, if multiple children exist, one is chosen randomly. The DFS collects violations until all constraints in \mathcal{S} are encountered. Finally, we remove duplicates—VIOs with the same constraint and

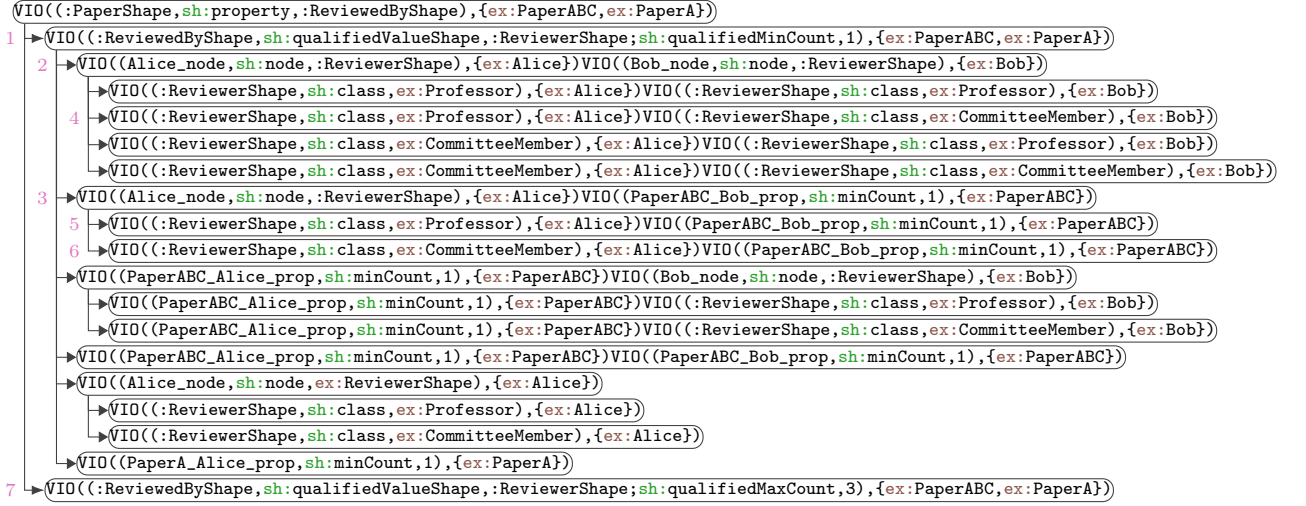


Fig. 3: Expansion tree and DFS on the running example.

focus that lead to isomorphic invalid graphs—to avoid bias when different shapes share identical constraints and focuses.

Running Example. Fig. 3 shows the expansion tree of the running example. The shapes in topological order are `:PaperShape`, `:ReviewedByShape`, and `:ReviewerShape`. Performing DFS from `:PaperShape`, the set of paths $\mathcal{P}_1 = \{(1, 2, 4), (7)\}$ collects all the constraints. Conversely, the set of paths $\mathcal{P}_2 = \{(1, 3, 5), (7)\}$ does not collect all constraints, whereas $\mathcal{P}_3 = \{(1, 3, 5), (1, 3, 6), (7)\}$ does. Consequently, DFS terminates for \mathcal{P}_1 and \mathcal{P}_3 , but not for \mathcal{P}_2 .

4.4 Apply Normalized Expressions on Copies of KG

For each leaf node in the set of paths collected from the DFS, we make a copy of \mathcal{G} and apply a predefined set of SPARQL operations to obtain \mathcal{G}' . Given $\text{VIO}(\kappa_s^i, \mathcal{F})$, where $\kappa_s^i = (\xi_s^i, \omega_s^i)$, we apply the SPARQL operations based on the value of ξ_s^i as follows.

$\xi_s^i = \text{sh:class}$. For any f in \mathcal{F} , f satisfies κ_s^i only if $(v, a, \omega_s^i) \in \mathcal{G}$ for all $v \in \mu_s(f)$, where ω_s^i is a class name. Therefore, we randomly choose one $f \in \mathcal{F}$ and one $v \in \mu_s(f)$ and perform the SPARQL operation `remove((v, a, ω_s^i))` to remove the triple (v, a, ω_s^i) from \mathcal{G} . When s is a property shape and $\mu_s(f) = \emptyset$ (i.e., f has no value node), we perform the SPARQL operation `add((f, p_s , ℓ))` to add the triple (f, p_s, ℓ) to \mathcal{G} , where ℓ is a randomly selected literal from \mathcal{G} .

$\xi_s^i = \text{sh:minCount}$. For any f in \mathcal{F} , f satisfies κ_s^i only if $|\mu_s(f)| \geq \omega_s^i$, where ω_s^i is a positive integer. Therefore, we randomly choose a subset $\mathcal{M} \subseteq \mu_s(f)$ of size $|\mu_s(f)| - \omega_s^i + 1$ and perform `remove((f, p_s , v))` for all $v \in \mathcal{M}$.

Other non shape-based constraints can be implemented similarly. We implement `sh:datatype`, `sh:nodeKind`, `sh:maxCount`, `sh:hasValue`, and `sh:in` for the evaluation in Sec. 7, leaving additional constraints for future work.

Table 1: Summary of contexts.

	Label	Description
SHACL Manifest Context	M	The entire SHACL manifest excluding natural language descriptions of classes
	S	The source constraint and dependency
	S_n	S unioned with natural language descriptions of classes
Knowledge Graph Context	G	The entire knowledge graph
	F	Triples used to validate the focus node and relevant value nodes
	F^+	F unioned with a positive example

Table 2: Dataset statistics. The \mathcal{G} and \mathcal{S} sizes of Brick are averaged across eight different KGs and manifests.

	\mathcal{G} size (#triples)	\mathcal{S} size (#triples)	test cases	α mean	max
Brick	35	108	144	3.23	13
LUBM	229	207	70	2.51	14
QUDT	81,293	2,642	134	1.5	26

Validation Report
Conforms: False
Results (2):
Constraint Violation in QualifiedMinCountConstraintComponent:
Source Shape: :ReviewedByShape
Focus Node: ex:PaperABC
Result Path: ex:reviewedBy
Constraint Violation in QualifiedMinCountConstraintComponent:
Source Shape: :ReviewedByShape
Focus Node: ex:PaperA
Result Path: ex:reviewedBy

Fig. 4: Validation report $\Gamma(\mathcal{G}', \mathcal{S})$, where \mathcal{G}' is derived from applying E1 to \mathcal{G} . \mathcal{G} and \mathcal{S} are defined in Fig. 2.

Running Example. Consider \mathcal{P}_3 from our running example, with leaf nodes

- (E1) $\text{VIO}(\text{:ReviewerShape}, \text{sh:class}, \text{ex:CommitteeMember}, \{\text{ex:Alice}\}) \cdot \text{VIO}(\text{(PaperABC_Bob_prop}, \text{sh:minCount}, 1), \{\text{ex:PaperABC}\})$
- (E2) $\text{VIO}(\text{:ReviewerShape}, \text{sh:class}, \text{ex:Professor}, \{\text{ex:Alice}\}) \cdot \text{VIO}(\text{(PaperABC_Bob_prop}, \text{sh:minCount}, 1), \{\text{ex:PaperABC}\})$
- (E3) $\text{VIO}(\text{:ReviewedByShape}, \text{sh:qualifiedValueShape}, \text{:ReviewerShape}; \text{sh:qualifiedMaxCount}, 3), \{\text{ex:PaperABC}, \text{ex:PaperA}\})$

By taking E1, for example, we apply the SPARQL operations: `remove((ex:Alice, a, ex:CommitteeMember))`, which disqualifies `ex:Alice` as a `:Reviewer` for `ex:PaperABC`, and `remove((ex:PaperABC, ex:reviewedBy, ex:Bob))`, which removes `ex:Bob` as a `:Reviewer` of `ex:PaperABC`. These operations are performed on the same copy of \mathcal{G} to produce \mathcal{G}' . We validate \mathcal{G}' against \mathcal{S} and obtain the validation report in Fig. 4. E1 produces two validation results, though it was to violate one constraint, $(\text{:PaperShape}, \text{sh:property}, \text{:ReviewedByShape})$, with one focus `ex:PaperABC`. The VIO also affects `ex:PaperA`. We define the **amplification factor** α of an expression E as the number of non-empty validation results in $\Gamma(\mathcal{G}', \mathcal{S})$. In this case, we have $\alpha(\text{E1}) = 2$. For a KG with high dependencies, α can be high. We apply E2 and E3 on copies of \mathcal{G} to generate two more violating instances of the KG, yielding three test cases for the running example.

5 Prototype LLM-Based Repair Systems

To demonstrate our evaluation framework, we develop a family of LLM-based KG repair systems, each utilizing a different large language model and prompting

Primer You are an expert in repairing RDF graphs that violate SHACL shapes. Output a SPARQL operation that fixes the violation.
Violation Context Focus Node: <urn:bldg/ahu> Violated source SHACL shape: <pre> [] sh:path brick:feeds; sh:qualifiedMinCount 1; sh:qualifiedValueShape [sh:node <urn:my_site_constraints/co2_zone>]. </pre> Reason: qualifiedMinCount is violated
SHACL Manifest Context SHACL shapes graph: <pre> <urn:my_site_constraints/co2_zone> a sh:NodeShape; sh:property [sh:path brick:hasPoint; ... <omit> ... </pre>
Knowledge Graph Context RDF knowledge graph: <pre> <urn:bldg/ahu> a brick:Air_Handling_Unit; brick:hasPoint <urn:bldg/outside_co2_sensor>. ... <omit> ... </pre>
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 {...}"}.

Fig. 5: Prompt template for our LLM-based KG repair systems

strategy. Our framework supports a fine-grained analysis of these systems, as described in Sec. 7.

Each system takes as input the invalid graph \mathcal{G}' , the manifest \mathcal{S} , and the validation report $\Gamma(\mathcal{G}', \mathcal{S})$. Given a validation result in $\Gamma(\mathcal{G}', \mathcal{S})$, the LLM is prompted to generate a repair. The prompting template for the repair systems has five sections (Fig. 5): Primer, Violation Context, SHACL Manifest Context (\mathcal{C}_m), KG Context (\mathcal{C}_g), and Instructions. We define three configurations for both \mathcal{C}_m and \mathcal{C}_g , yielding nine repair system variations summarized in Table 1.

For the Manifest Context \mathcal{C}_m , we define three variations with respect to a validation result $\rho_{s,v}^i$ with source shape Σ_s and focus v . (i) M : The entire Turtle-serialized [3] manifest \mathcal{S} , providing the most detailed context but may exceed the LLM context window. (ii) S : A subset of \mathcal{S} containing only the source shape and its dependent shapes to improve scalability by reducing context size while retaining essential validation information. (iii) S_n : Extends S by adding natural language descriptions (e.g., `dcterms:description`, `rdfs:label`) for classes to assess the value of natural language context.

For the KG context \mathcal{C}_g , we define three variations with respect to a validation result $\rho_{s,v}^i$ with source shape Σ_s , focus v , and the chosen manifest context \mathcal{C}_m . (i) G : The entire Turtle-serialized KG \mathcal{G} , providing the most detailed context but may exceed the LLM context window. (ii) $F(\mathcal{C}_m)$: A subset of \mathcal{G} , including only triples involved in validating the focus against the source shape to produce $\rho_{s,v}^i$. If the source constraint is `sh:qualifiedMinCount` and $\Sigma_{s'}$ is the object of the corresponding `sh:qualifiedValueShape`, we include triples from validating nodes that satisfy $\Sigma_{s'}$. Essentially, $F(\mathcal{C}_m)$ removes information in G that is not used in the validation to improve scalability. (iii) $F(\mathcal{C}_m)^+$: Extends $F(\mathcal{C}_m)$ with

```

:ReviewedByShape a sh:PropertyShape;
sh:path ex:reviewedBy;
sh:qualifiedValueShape :ReviewerShape;
sh:qualifiedMinCount 1.
:ReviewerShape a sh:NodeShape;
sh:class ex:Professor,
        ex:CommitteeMember.

```

Fig. 6 (a) S for the running example

```

ex:PaperABC a ex:Paper;
ex:reviewedBy ex:Alice, ex:Clark.
ex:Alice a ex:Professor.
ex:Bob a ex:Professor,
        ex:CommitteeMember.
ex:Clark a ex:Student.
ex:Dan a ex:Professor,
        ex:CommitteeMember.

```

Fig. 6 (b) $F(S)$ for the running example

a positive example by including triples from validating a node that satisfies the source shape. This serves as additional context to show how Σ_s is satisfied.

Running Example. Assume we obtain \mathcal{G}' by applying E1, resulting in the validation result $\rho^1_{\text{ReviewedByShape}, \text{ex:PaperABC}}$. We show S and $F(S)$ in Fig. 6. Note that in Fig. 6b, $(\text{ex:PaperABC}, \text{ex:author}, \text{ex:Ethan})$ is not included because the triple was not used in validating ex:PaperABC against $\Sigma_{\text{ReviewedByShape}}$, as defined in Fig. 6a. On the other hand, ex:Bob and ex:Dan satisfy :ReviewerShape , so the relevant triples are included.

6 Evaluation Metrics

We define four tiered metrics to evaluate the quality of the generated repair. Each tier is assessed only if all the lower-tier metrics are satisfied. Additionally, we describe how the cost of generating a repair is measured.

The metrics are listed below in order of increasing stringency. We start with **syntactic validity**: A repair π should be parsed without error. Therefore, π is syntactically valid if it adheres to correct SPARQL syntax. As a second metric, we consider **semantic validity**: After a syntactically correct π is applied to the invalid graph \mathcal{G}' to obtain \mathcal{G}'_{π} , the violation should be eliminated. Thus, π is semantically valid if $\mathcal{G}'_{\pi} \models \mathcal{S}$. We then evaluate a property which we term **relaxed isomorphism**: A semantically valid π should ideally revert the VIO to recover the original KG \mathcal{G} . However, inferring the exact string from the provided context can be challenging without access to an external knowledge base. Therefore, we relax the traditional definition of isomorphism for RDF graphs [25] to allow non-exact string matches for the literals. If \mathcal{G}'_{π} and \mathcal{G} are isomorphic when all literals are replaced with a placeholder string **placeholder**, then π is relaxed-isomorphic. Finally, a relaxed-isomorphic π is further tested for exact **isomorphism**. If \mathcal{G}'_{π} and \mathcal{G} are exactly isomorphic, then π is isomorphic.

Recall that different variations of manifest and KG contexts include varying levels of details and consequently, different token counts. Moreover, commercial LLMs have distinct pricing models (see App. D for details). We choose the following LLMs: (i) GPT4o (OpenAI), (ii) Claude 3.0 Opus (Anthropic), (iii) Gemini 1.5 Pro (Google), and (iv) Llama 3.1 405B (Meta, via AWS Bedrock). For each combination of manifest and KG context and LLM, we evaluate the token counts and cost of generating π . All LLMs were accessed in January 2025.

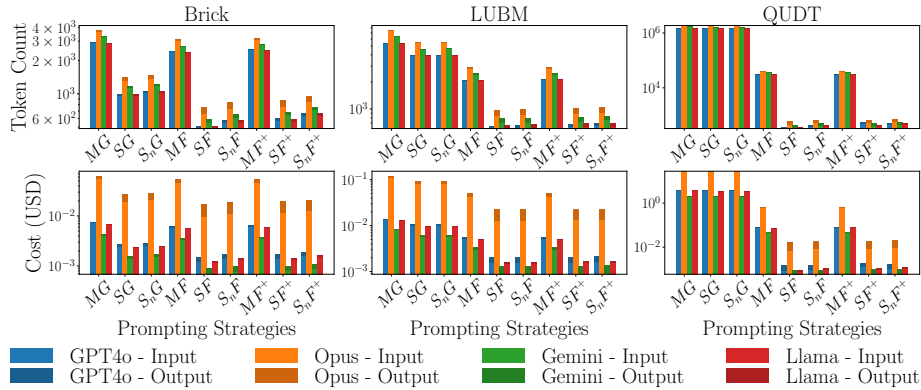


Fig. 7: Average token count and cost for different prompting strategies and LLMs.

7 Evaluation Results

We test our evaluation framework and repair systems on three datasets. Each consists of KGs with corresponding SHACL manifests. **Brick Ontology** [9] standardizes semantic descriptions of the physical, logical, and virtual assets in buildings and their relationships. Using the 1.3 release, we construct eight manifests based on HVAC hardware specifications from ASHRAE Guideline 36 [8] and manually create one KG for each system. **LUBM Ontology** [23] is a synthetic graph modeling a university. We adapt the KG and manifest from Figuera et al. [21,38] by adding triples to create a valid graph. **QUDT Ontology** [15] is a unified conceptual representation of quantities, quantity kinds, units, and related concepts. Using the 2.1 release, this dataset is two to three orders of magnitude larger than the others, and serves to test the scalability of the repair process. Descriptive statistics for each data set are in Table 2.

7.1 Evaluation of Prototype Repair System

Given a validation report of an invalid graph, $\Gamma(\mathcal{G}', \mathcal{S})$, we randomly choose one validation result to construct different prompting strategies as described in Sec. 5. If multiple VIO expressions are applied to the same graph, a more sophisticated mechanism would be required to resolve interdependencies between validation results; we leave this for future work.

With three variations each for the manifest and KG contexts, we obtain nine distinct prompting strategies. For simplicity, we omit the explicit dependency of the KG context on the manifest context in our notation (e.g., MF denotes $MF(M)$). We apply these nine strategies across the four LLMs described in Sec. 6 and three datasets, evaluating both cost and repair quality.

Cost of generating the repair. Fig. 7 presents the average input and output token counts and associated costs. Due to budget constraints, we do not query

Table 3: Average scores for all LLMs and prompting strategies.

	Syntactic Validity	Semantic Validity	Relaxed Isomorphism	Isomorphism
Average	97.35%	86.17%	54.07%	49.94%

the LLM for a response if the average input cost exceeds \$0.5; thus, output token counts and costs for these cases are excluded from the figure.

Fig. 7 shows that using the entire manifest and KG as context (MG) results in the highest input token count and the highest expense across all datasets. The inclusion of natural language descriptions for classes (S_n versus S) increases input tokens only minimally for LUBM, as its ontology contains few such definitions. We also observe that Claude 3.0 Opus has the most granular tokenization, leading to the highest token counts and costs.

Quality of the repair. We evaluate the four LLMs on three datasets using the nine prompting strategies. Table 3 reports the percentage of test cases passing each evaluation metric, averaged across datasets (see App. E for detailed results). **Syntactic validity** is nearly 100%, with errors mainly due to missing or extra closing brackets or quotations, indicating that LLMs handle SPARQL syntax well. However, **isomorphism** scores the lowest; we hypothesize that giving the repair system access to external knowledge bases could improve this, which we leave for future work.

We next assess the effects of different manifest contexts, KG contexts, and LLM choices on repair quality.

Manifest 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 ? To answer these, we collect relevant results and fit a linear mixed-effects model (LMM) [35] to estimate the effect of using different manifest contexts. We use LMM because our measurements of nine prompting strategies on three datasets using four LLMs violate the independence assumptions required for ordinary least squares linear regression [48]; for example, measurements from the same LLM are correlated. Further details on LMM can be found in App. F.

Fig. 8 shows the average effect (percentage change, $\Delta\%$) across all metrics and individually, with 95% confidence intervals. Metrics not crossing the 0% line are statistically significant (p -value < 0.05). The results indicate that switching from M to S improves all metrics except **syntactic validity**, which is already near 100%. This suggests that providing the entire manifest (M) overloads and distracts the LLM, while focusing on essential validation information (S) leads to better repairs. Switching from S to S_n has no significant effect. Changing from M to S_n yields similar results as changing from M to S . Overall, concise prompts perform best; S is preferred since it requires fewer tokens than S_n and yields reliable results.

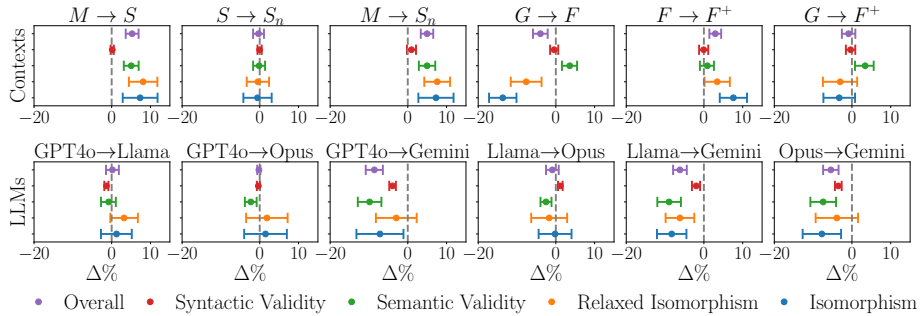


Fig. 8: **Estimated effects of context selection and LLM selection.**

KG Context. We also analyze the impact of different KG contexts. As shown in Fig. 8, switching from G to F negatively affects overall metrics, especially **relaxed isomorphism** and **isomorphism**, 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 valuable context needed to generate repairs that closely match the original KG. In contrast to the manifest context, where reducing information ($M \rightarrow S$) is beneficial, reducing KG context ($G \rightarrow F$) harms the stricter metrics. This suggests that information not directly used in validation still provides useful context for the LLM to infer the most appropriate repair.

Adding a positive example ($F \rightarrow F^+$) improves overall performance, particularly for **isomorphism**. Recall that F^+ augments F with a positive example from \mathcal{G} that is not involved in validating the focus node. Changing from G to F^+ only improves **semantic validity**, with little effect on other metrics. Notably, compared to $G \rightarrow F$, the negative impact on **relaxed isomorphism** and **isomorphism** is much smaller for $G \rightarrow F^+$. We hypothesize that including more context about the focus node could further improve performance beyond G . Overall, F^+ offers a good balance between cost, scalability, and performance.

LLM Selection. We also assess which LLM yields the highest quality repairs. Fig. 8 shows that switching from GPT-4o, Llama 3.1 405B, or Claude 3.0 Opus to Gemini 1.5 Pro significantly reduces performance. However, there is no significant difference among GPT-4o, Llama 3.1 405B, and Claude 3.0 Opus. Considering cost, GPT-4o and Llama 3.1 405B are preferable, as Claude 3.0 Opus incurs higher costs due to its granular tokenization.

In summary, our analysis shows that the best repair results are achieved by including only the manifest and KG information used to validate the focus node v against the source shape Σ_s , plus a positive example for additional context. This fine-grained insight is uniquely enabled by our evaluation framework.

8 Conclusion

This paper presents a systematic evaluation framework and metrics for repairing knowledge graphs using SHACL manifests. We assess the semantic and contextual capabilities of four commercial LLMs across nine prompting strategies, yielding three key insights: (i) For manifest context, including the source shape and its dependencies is essential for effective repairs; (ii) For KG context, restricting context to only the focus node and its dependencies reduces repair quality, but adding a positive example preserves performance; and (iii) Among the LLMs tested, three yield similar performances, whereas GPT4o and Llama 3.1 405B offer advantages in cost. Our framework enables fine-grained analysis of repair systems, supporting the automated generation and maintenance of high-quality knowledge graphs for downstream applications.

References

1. Resource description framework (rdf). <https://www.w3.org/RDF/>, accessed: 2025-05-05
2. Simple protocol and rdf query language, (sparql). <https://www.w3.org/TR/sparql11-query/>, accessed: 2025-05-05
3. Turtle syntax. <https://www.w3.org/TR/turtle/>, accessed: 2025-05-05
4. World wide web consortium (w3c). <https://www.w3.org/>, accessed: 2025-05-05
5. Ahmetaj, S., David, R., Polleres, A., Šimkus, M.: Repairing shacl constraint violations using answer set programming. In: International Semantic Web Conference. pp. 375–391. Springer (2022)
6. Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 68–79 (1999)
7. Arnaout, H., Tran, T.K., Stepanova, D., Gad-Elrab, M.H., Razniewski, S., Weikum, G.: Utilizing language model probes for knowledge graph repair. In: Wiki Workshop 2022 (2022)
8. ASHRAE, G.: 36: High performance sequences of operation for hvac systems. American Society of Heating, Refrigerating and Air-Conditioning Engineers, Atlanta (2018)
9. Balaji, B., Bhattacharya, A., Fierro, G., Gao, J., Gluck, J., Hong, D., Johansen, A., Koh, J., Ploennigs, J., Agarwal, Y., 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 (2016)
10. Bi, Z., Chen, J., Jiang, Y., Xiong, F., Guo, W., Chen, H., Zhang, N.: Codekgc: Code language model for generative knowledge graph construction. *ACM Transactions on Asian and Low-Resource Language Information Processing* **23**(3), 1–16 (2024)
11. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: 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 (2008)
12. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter,

- C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language Models are Few-Shot Learners (Jul 2020). <https://doi.org/10.48550/arXiv.2005.14165>
13. Car, N.: pyshacl. <https://github.com/RDFLib/pySHACL> (2024). <https://doi.org/10.5281/zenodo.4750840>, for all versions/latest version
 14. Corman, J., Reutter, J.L., 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. pp. 318–336. Springer (2018)
 15. FAIRsharing.org: Qudt; quantities, units, dimensions and types (2022), <https://doi.org/10.25504/FAIRsharing.d3ppqw7>, last Edited: Friday, May 6th 2022, 2:03, Last Accessed: Friday, December 13th 2024, 11:47, Last Reviewed: Monday, May 2nd 2022, 3:22
 16. Fan, W., Lu, P., Tian, C., Zhou, J.: Deducing certain fixes to graphs. Proceedings of the VLDB Endowment **12**(7), 752–765 (2019)
 17. Farzana, S., Zhou, Q., Ristoski, P.: Knowledge graph-enhanced neural query rewriting. In: Companion Proceedings of the ACM Web Conference 2023. pp. 911–919 (2023)
 18. Fierro, G., Prakash, A.K., Mosiman, C., Pritoni, M., Raftery, P., Wetter, M., Culler, D.E.: Shepherding Metadata Through the Building Lifecycle. In: Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation. pp. 70–79. ACM, Virtual Event Japan (Nov 2020). <https://doi.org/10.1145/3408308.3427627>
 19. Fierro, G., Pritoni, M., Abdelbaky, M., Lengyel, D., Leyden, J., Prakash, A., Gupta, P., Raftery, P., Peffer, T., Thomson, G., Culler, D.E.: Mortar: An open testbed for portable building analytics. ACM Trans. Sen. Netw. **16**(1) (Dec 2019). <https://doi.org/10.1145/3366375>, <https://doi.org/10.1145/3366375>
 20. Fierro, G., Saha, A., Shapinsky, T., Steen, M., Eslinger, H.: 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 (2022)
 21. Figuera, M., Rohde, P.D., Vidal, M.E.: Trav-shacl: Efficiently validating networks of shacl constraints. In: Proceedings of the Web Conference 2021. pp. 3337–3348 (2021)
 22. Gayo, J.E.L., Knublauch, H., Kontokostas, D.: Data Shapes Test Suite. <https://w3c.github.io/data-shapes/data-shapes-test-suite/> (2025), accessed: 2025-04-23
 23. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Journal of Web Semantics **3**(2-3), 158–182 (2005)
 24. Hammar, K., Wallin, E.O., Karlberg, P., Halleberg, D.: The RealEstateCore Ontology. p. 16
 25. Hogan, A.: Skolemising blank nodes while preserving isomorphism. In: Proceedings of the 24th International Conference on World Wide Web. pp. 430–440 (2015)
 26. Klop, J.W., Klop, J.: Term rewriting systems. Centrum voor Wiskunde en Informatica (1990)
 27. Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., Newman, B., Yuan, B., Yan, B., Zhang, C., Cosgrove, C., Manning, C.D., Ré, C., Acosta-Navas, D., Hudson, D.A., Zelikman, E., Durmus, E., Ladhak, F., Rong, F., Ren, H., Yao, H., Wang, J., Santhanam, K., Orr, L., Zheng, L., Yuksekgonul, M., Suzgun, M., Kim, N., Guha, N., Chatterji, N.,

- Khattab, O., Henderson, P., Huang, Q., Chi, R., Xie, S.M., Santurkar, S., Ganguli, S., Hashimoto, T., Icard, T., Zhang, T., Chaudhary, V., Wang, W., Li, X., Mai, Y., Zhang, Y., Koreeda, Y.: Holistic Evaluation of Language Models (Oct 2023). <https://doi.org/10.48550/arXiv.2211.09110>
28. Malaviya, C., Bhagavatula, C., Bosselut, A., Choi, Y.: Commonsense knowledge base completion with structural and semantic context. In: Proceedings of the AAAI conference on artificial intelligence. vol. 34, pp. 2925–2933 (2020)
 29. Melnyk, I., Dognin, P., Das, P.: Knowledge graph generation from text. arXiv preprint arXiv:2211.10511 (2022)
 30. Mihindukulasooriya, N., Tiwari, S., Enguix, C.F., Lata, K.: Text2kgbench: A benchmark for ontology-driven knowledge graph generation from text. In: International Semantic Web Conference. pp. 247–265. Springer (2023)
 31. Mirchandani, S., Xia, F., Florence, P., Ichter, B., Driess, D., Arenas, M.G., Rao, K., Sadigh, D., Zeng, A.: Large language models as general pattern machines. arXiv preprint arXiv:2307.04721 (2023)
 32. Pareti, P., Konstantinidis, G.: A review of shacl: from data validation to schema reasoning for rdf graphs. Reasoning Web International Summer School pp. 115–144 (2021)
 33. Pellissier Tanon, T., Bourgaux, C., Suchanek, F.: Learning how to correct a knowledge base from the edit history. In: The World Wide Web Conference. pp. 1465–1475 (2019)
 34. Pellissier Tanon, T., Suchanek, F.: Neural knowledge base repairs. In: The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings 18. pp. 287–303. Springer (2021)
 35. Pinheiro, J.C., Bates, D.M.: Linear mixed-effects models: basic concepts and examples. Mixed-effects models in S and S-Plus pp. 3–56 (2000)
 36. Rodríguez-Muro, M., Kontchakov, R., Zakharyashev, M.: Ontology-Based Data Access: Ontop of Databases. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) Advanced Information Systems Engineering, vol. 7908, pp. 558–573. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41335-3_35
 37. Ryen, V., Soylu, A., Roman, D.: Building Semantic Knowledge Graphs from (Semi-)Structured Data: A Review. Future Internet **14**(5), 129 (Apr 2022). <https://doi.org/10.3390/fi14050129>
 38. SDM-TIB: Tracedsparql (2023), <https://github.com/SDM-TIB/TracedSPARQL>, accessed: 2024-12-13
 39. Shen, T., Zhang, F., Cheng, J.: A comprehensive overview of knowledge graph completion. Knowledge-Based Systems **255**, 109597 (2022)
 40. Staworko, S., Chomicki, J., Marcinkowski, J.: Prioritized repairing and consistent query answering in relational databases. Annals of Mathematics and Artificial Intelligence **64**(2), 209–246 (2012)
 41. Suchanek, F.M., Alam, M., Bonald, T., Chen, L., Paris, P.H., Soria, J.: 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. pp. 131–140. ACM, Washington DC USA (Jul 2024). <https://doi.org/10.1145/3626772.3657876>
 42. Vrandečić, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Communications of the ACM **57**(10), 78–85 (Sep 2014). <https://doi.org/10.1145/2629489>

43. Wang, L., Zhao, W., Wei, Z., Liu, J.: Simkgc: Simple contrastive knowledge graph completion with pre-trained language models. arXiv preprint arXiv:2203.02167 (2022)
44. Wright, J., Rodríguez Méndez, S.J., Haller, A., Taylor, K., Omran, P.G.: Schimatos: a shacl-based web-form generator for knowledge graph editing. In: International Semantic Web Conference. pp. 65–80. Springer (2020)
45. Xu, Z., Cruz, M.J., Guevara, M., Wang, T., Deshpande, M., Wang, X., Li, Z.: 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 (2024)
46. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: Synergizing reasoning and acting in language models. In: International Conference on Learning Representations (ICLR) (2023)
47. Yu, N., Nützmann, H.W., MacDonald, J.T., Moore, B., Field, B., Berriri, S., Trick, M., Rosser, S.J., Kumar, S.V., Freemont, P.S., Osbourn, A.: Delineation of metabolic gene clusters in plant genomes by chromatin signatures. *Nucleic Acids Research* **44**(5), 2255–2265 (Mar 2016). <https://doi.org/10.1093/nar/gkw100>
48. Zdaniuk, B.: Ordinary least-squares (ols) model. In: Encyclopedia of quality of life and well-being research, pp. 4867–4869. Springer (2024)
49. Zhao, S., Yang, Y., Wang, Z., He, Z., Qiu, L.K., Qiu, L.: Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely (Sep 2024). <https://doi.org/10.48550/arXiv.2409.14924>

A VIO Definition for `sh:qualifiedMaxCount`

Following the discussion in Sec. 4.2 on the rewrite rule for $\text{VIO}(\kappa_s^i, \mathcal{F})$, where \mathcal{F} is a set of focus nodes and $\kappa_s^i = (\xi_s^i, \omega_s^i)$ denotes the i -th constraint-parameter pair for shape Σ_s , with ω_s^i being a shape name and I_s the index set of constraints for Σ_s). Consider the case where $\xi_s^i = \text{sh:qualifiedValueShape}$. If there exists $h \in I_s$ such that $\xi_s^h = \text{sh:qualifiedMaxCount}$, then a violation of `sh:qualifiedMaxCount` requires that all constraints in the dependency of $\Sigma_{\omega_s^i}$ be satisfied. Thus, no rewrite rule is needed. Instead, we directly define the SPARQL operation for $\text{VIO}(\kappa_s^i, \mathcal{F})$ when $\xi_s^i = \text{sh:qualifiedValueShape}$ and `sh:qualifiedMaxCount` constraint exists.

For a focus node $f \in \mathcal{F}$, we define $\psi_{\omega_s^i}(f) = \{v \in \mu_s(f) \mid v \models \Sigma_{\omega_s^i}\}$, a value mapping function that is stricter than the ordinary value mapping function defined in Sec. 4.1 μ_s , in the sense that $\psi_{\omega_s^i}(f)$ only admits nodes $v \in \mu_s(f)$ that satisfy $\Sigma_{\omega_s^i}$. We must make edits such that $|\psi_{\omega_s^i}(f)|$ becomes strictly larger than ω_s^h , the positive integer parameter value of `sh:qualifiedMaxCount`. We need to construct a set $\epsilon_f^M \subseteq \mathcal{V}$, where $\forall v \in \epsilon_f^M, v \models \Sigma_{\omega_s^i}$ and $v \notin \psi_{\omega_s^i}(f)$. Furthermore, $|\epsilon_f^M| = \omega_s^h - |\psi_{\omega_s^i}(f)| + 1$. We perform class-aware subgraph monomorphism search in \mathcal{G} following Fierro et al. [20] for nodes that satisfy $\Sigma_{\omega_s^i}$. If only partial matches of $\Sigma_{\omega_s^i}$ are identified, we prompt an LLM to mint new entity names (see App. B for details). We then define the SPARQL operation: `add((f, p_s, v))` for all $v \in \epsilon_f^M$.

B Subgraph Monomorphism Search for $\Sigma_{\omega_s^i}$

The subgraph monomorphism search proposed by Fierro et al. [20] first converts $\Sigma_{\omega_s^i}$ to a graph $\mathcal{G}_{\omega_s^i}$ then retrieves the largest subgraphs in \mathcal{G} that are monomorphic to subgraphs of $\mathcal{G}_{\omega_s^i}$. Each retrieved subgraph corresponds to a part of \mathcal{G} that satisfies a maximum number of constraints in $\Sigma_{\omega_s^i}$. For a retrieved subgraph $\bar{\mathcal{G}}(\bar{\mathcal{V}}, \bar{\mathcal{E}}) \subseteq \mathcal{G}(\mathcal{V}, \mathcal{E})$ that is fully monomorphic to $\mathcal{G}_{\omega_s^i}$, we identify the node $\bar{v} \in \bar{\mathcal{V}}$ such that $\bar{v} \models \Sigma_{\omega_s^i}$ and add it to ϵ_f^M if it is not already in $\psi_{\omega_s^i}(f)$. However, if the retrieved subgraph is only monomorphic to a subgraph of $\mathcal{G}_{\omega_s^i}$, we prompt an LLM to mint new entities so that the identified $\bar{v} \in \bar{\mathcal{V}}$ satisfies $\Sigma_{\omega_s^i}$.

Running Example. Consider `VIO((:ReviewedByShape, sh:qualifiedValueShape, :ReviewerShape; sh:qualifiedMaxCount, 3), {ex:PaperABC})`, where we have $\psi_{\text{ReviewerShape}}(\text{ex:PaperABC}) = \{\text{ex:Alice}, \text{ex:Bob}\}$. We must construct a set $\epsilon_{\text{ex:PaperABC}}^M$ of size $3 - 2 + 1 = 2$. The subgraph monomorphism search retrieves a node `ex:Dan` $\in \mathcal{V}$ that satisfies `:ReviewerShape` and is not already in $\{\text{ex:Alice}, \text{ex:Bob}\}$. We still need one more node that satisfies `:ReviewerShape` but the only existing subgraphs in \mathcal{G} are

1. `(ex:Alice, a, ex:Professor, ex:CommitteeMember)`,
2. `(ex:Bob, a, ex:Professor, ex:CommitteeMember)`,
3. `(ex:Dan, a, ex:Professor, ex:CommitteeMember)`

```

Generate an entity name to replace urn:___param___#name in the following graph.
urn:___param___#name a ex:Professor, ex:ComitteeMember.
Your answer must be semantically similar to the corresponding entity in the following example but
not exactly identical.
ex:Alice a ex:Professor, ex:ComitteeMember.
Compare with the example, observe if urn:___param___#name 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}.

```

Fig. 9: Example Prompt for `sh:qualifiedMaxCount`

that is monomorphic to $\mathcal{G}_{\text{ReviewerShape}}$. Therefore, we prompt an LLM as in Fig. 9 to mint a fresh entity name.

C Proof of Strong Normalization

This section proves Theorem 1: if the SHACL manifest \mathcal{S} and knowledge graph \mathcal{G} are finite, and there are no recursive shapes [14], then the rewriting system defined in Sec. 4.2 is strongly normalizing; that is, every rewriting sequence terminates. Following Corman et al. [14] that a shape is recursive if it refers to itself; specifically, a shape Σ_s is recursive if and only if its dependency contains Σ_s .

We begin with an informal intuitive argument, followed by a more formal proof.

Since \mathcal{S} is finite and contains no recursive shapes, the shape dependencies in \mathcal{S} form a finite directed acyclic graph (DAG). When rewrite rules are applied, they correspond to traversals within this DAG: rule 1 moves to a child shape, while rule 2 remains at the current shape. Because both \mathcal{G} and \mathcal{S} are finite, only finitely many applications of rule 2 can occur before the next time rule 1 is applied. As every path in a finite DAG is of finite length, every traversal sequence must terminate. Thus, the rewriting process is strongly normalizing.

To formalize the above argument, we define a shape sequence starting with a shape Σ_{s_1} as:

$$\sigma(\Sigma_{s_1}) ::= \Sigma_{s_1}, \Sigma_{s_2}, \dots, \Sigma_{s_k}$$

where $\Sigma_{s_{i+1}}$ follows Σ_{s_i} iff there exists $j \in I_{s_i}$ such that $\omega_{s_i}^j = s_{i+1}$ ⁶. If no such j exists, then Σ_{s_i} is the last element of the sequence. We denote the length of $\sigma(\Sigma_s)$ as $\text{len}(\sigma(\Sigma_s))$.

Assuming \mathcal{S} is finite and contains no recursive shapes, the shape dependency structure forms a directed acyclic graph (DAG). A shape corresponds to a node in the DAG, and there is a directed edge from shape Σ_{s_i} to shape $\Sigma_{s_{i+1}}$ whenever there is a constraint $\kappa_{s_i}^j = (\xi_{s_i}^j, \omega_{s_i}^j)$ such that $\omega_{s_i}^j = s_{i+1}$. Furthermore, there are no recursive shapes; therefore, the directed graph is acyclic. By the properties of DAGs:

⁶ Recall that I_{s_i} denotes the index set of constraints of shape Σ_{s_i} and that the j -th constraint $\kappa_{s_i}^j = (\xi_{s_i}^j, \omega_{s_i}^j)$ refers to another shape $\Sigma_{s_{i+1}}$ if $\omega_{s_i}^j = s_{i+1}$.

1. All paths are of finite length.
2. There are finitely many distinct finite paths.

Let $\Xi(\Sigma_s)$ denote the set of all shape sequences starting with Σ_s . By the DAG properties, $\Xi(\Sigma_s)$ is finite and contains only finite sequences. To prove the strong normalization property of \mathcal{A} , we define a decreasing complexity measure χ for $i \in I_s$ to show the progress after each rewrite :

$$\chi(\text{VIO}(\kappa_s^i, \mathcal{F})) = \max_{\sigma(\Sigma_s) \in \Xi(\Sigma_s)} \text{len}(\sigma(\Sigma_s)), \quad (1)$$

which measures the length of the longest shape sequence starting with Σ_s . For compound terms, χ is defined as:

$$\chi(\text{VIO}(\kappa_{s_1}^i, \mathcal{F}_1) + \text{VIO}(\kappa_{s_2}^j, \mathcal{F}_2)) = \max \{ \chi(\text{VIO}(\kappa_{s_1}^i, \mathcal{F}_1)), \chi(\text{VIO}(\kappa_{s_2}^j, \mathcal{F}_2)) \}$$

$$\chi(\text{VIO}(\kappa_{s_1}^i, \mathcal{F}_1) \cdot \text{VIO}(\kappa_{s_2}^j, \mathcal{F}_2)) = \max \{ \chi(\text{VIO}(\kappa_{s_1}^i, \mathcal{F}_1)), \chi(\text{VIO}(\kappa_{s_2}^j, \mathcal{F}_2)) \}$$

The rewrite terminates when $\chi(\cdot) = 0$. This corresponds to the intuition that a path traversal in the DAG reaches the end and terminates. Then, we prove that χ is decreasing with the rewrite rules. Given $\text{VIO}(\kappa_s^i, \mathcal{F})$, where $\kappa_s^i = (\xi_s^i, \omega_s^i)$ is the i -th constraint-parameter pair of shape Σ_s . First, **Rule 1: $\xi_s^i = \text{sh:node or } \xi_s^i = \text{sh:property}$** :

$$\text{VIO}(\kappa_s^i, \mathcal{F}) \rightarrow \text{VIO}(\kappa_{\omega_s^i}^1, \mathcal{F}') + \dots + \text{VIO}(\kappa_{\omega_s^i}^{n_{\omega_s^i}}, \mathcal{F}').$$

See Sec. 4.2 for the detailed definition of rewrite rule 1. Since \mathcal{S} is finite by assumption, the sum on the right-hand side is finite. Additionally, elements in $\Xi(\Sigma_{\omega_s^i})$ are shape sequences with prefix $\Sigma_{\omega_s^i}, \dots$, which are subsequences of elements in $\Xi(\Sigma_s)$ with prefix $\Sigma_s, \Sigma_{\omega_s^i}, \dots$. Therefore, according to eq. (1), $\chi(\cdot)$ is decreased by 1 for every expression on the right hand side of rule 1. This corresponds to the intuition that rule 1 moves to the child shape in the path traversal of DAG. Therefore,

$$\chi(\text{VIO}(\kappa_s^i, \mathcal{F})) - 1 = \chi \left(\sum_{j \in I_{\omega_s^i}} \text{VIO}(\kappa_{\omega_s^i}^j, \mathcal{F}) \right)$$

This implies that χ strictly decreases by 1 after applying rule 1. Next, we analyze **Rule 2: $\xi_s^i = \text{sh:qualifiedValueShape}$** :

$$\text{VIO}(\kappa_s^i, \mathcal{F}) \rightarrow \sum_{f \in \mathcal{F}} \sum_{\substack{\epsilon_f^m \subseteq \psi_{\omega_s^i}(f) \\ |\epsilon_f^m| = |\psi_{\omega_s^i}(f)| - \omega_s^h + 1}} \sum_{g \in \Phi(\epsilon_f^m)} \prod_{v \in \epsilon_f^m} \text{VIO}(\kappa_{g(v)}^1, \{f\}).$$

See Sec. 4.2 for the detailed definition of rewrite rule 2. Recall that we introduce artificial shapes, Σ_{v_node} and $\Sigma_{f_v_prop}$ for $f \in \mathcal{F}$ and $v \in \epsilon_f^m$. See Sec. 4.2 for the definition of ϵ_f^m . The targeting functions, value mapping functions, and the constraints of the artificial shapes are defined as follows.

- $\tau_{v_node}(\mathcal{G}) = \{v\}$, $\mu_{v_node}(v) = \{v\}$, $\kappa_{v_node} = \{(\mathbf{sh}:\mathbf{node}, \omega_s^i)\}$ and
 - $\tau_{f_v_prop}(\mathcal{G}) = \{f\}$, $\mu_{f_v_prop}(f) = \{v\}$, $\kappa_{f_v_prop} = \{(\mathbf{sh}:\mathbf{minCount}, 1)\}$.
 Note that only κ_{v_node} is shape-based and $\kappa_{f_v_prop}$ is not. Moreover, $\kappa_{v_node} = \{(\mathbf{sh}:\mathbf{node}, \omega_s^i)\}$ refers to the same shape, $\Sigma_{\omega_s^i}$, as the constraint $\kappa_s^i = (\xi_s^i, \omega_s^i)$ before rewriting. This corresponds to the intuition that rule 2 stays at the current shape in the path traversal of DAG. Lastly, $\kappa_{v_node} = \{(\mathbf{sh}:\mathbf{node}, \omega_s^i)\}$ involves the predicate $\mathbf{sh}:\mathbf{node}$, which implies rule 1 will be applied following the application of rule 2.

Since \mathcal{G} and \mathcal{S} are finite and the number of artificial shapes introduced are bounded, rule 2 produces a finite sum of products. According to eq. (1), $\chi(\cdot)$ remains constant after the application of rule 2. Therefore,

$$\chi(\text{VIO}(\kappa_s^i, \mathcal{F})) = \chi \left(\sum_{f \in \mathcal{F}} \sum_{\substack{\epsilon_f^m \subseteq \psi_{\omega_s^i}(f) \\ |\epsilon_f^m| = |\psi_{\omega_s^i}(f)| - \omega_s^h + 1}} \sum_{g \in \Phi(\epsilon_f^m)} \prod_{v \in \epsilon_f^m} \text{VIO}(\kappa_{g(v)}^1, \{f\}) \right).$$

Although applying rule 2 does not decrease χ , the finiteness of \mathcal{S} guarantees that only finitely many such steps occur. In addition, application of rule 1 follows the application of rule 2. In conclusion, given that rewrite rules either strictly decrease χ or keep it constant only for finitely many steps, the rewriting system is strongly normalizing. Every rewrite sequence terminates after finitely many steps. \square

D LLM Pricing

We accessed all the LLMs in January, 2025. The costs are listed in Table 4.

Table 4: **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

E Detailed Results

We evaluated four LLMs on three datasets using nine prompting strategies. Fig. 10 reports the percentage of test cases passing each evaluation metric.

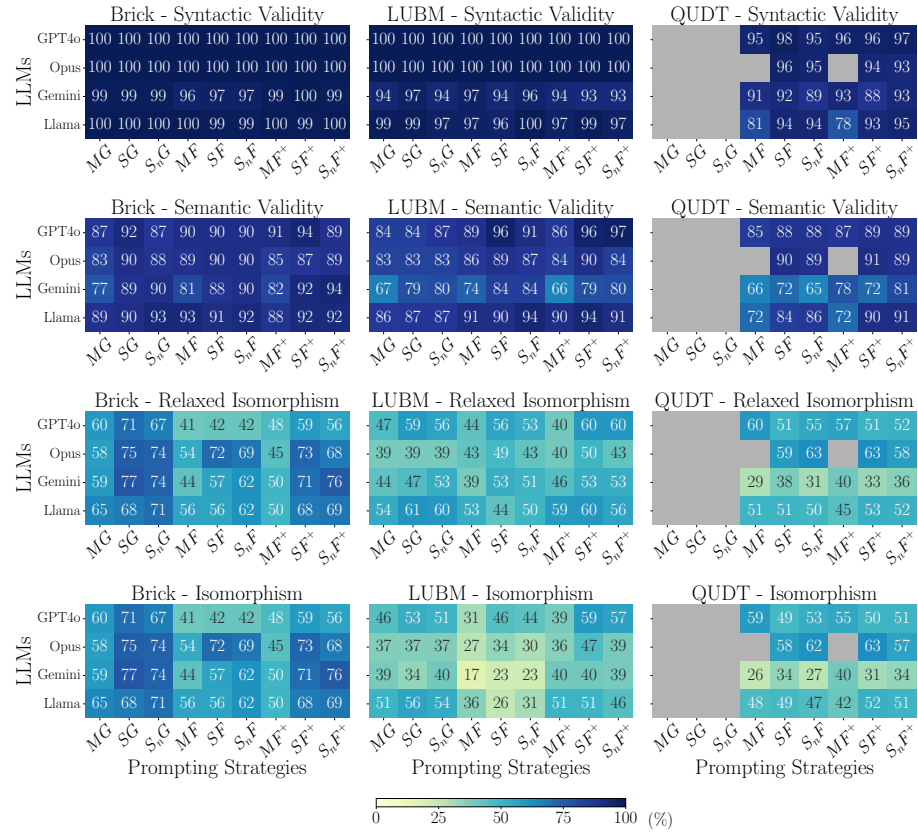


Fig. 10: Performance of each evaluation metric for every combination of LLM, strategy, and dataset.

F Details on Statistical Analysis

In our statistical analysis in Sec. 7, 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 manifest 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 [48], 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 manifest 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 manifest 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),
- $b_j^{(\text{dataset})} \sim \mathcal{N}(0, \sigma_{\text{dataset}}^2)$ is the random intercept for dataset j ,
- $b_k^{(\text{LLM})} \sim \mathcal{N}(0, \sigma_{\text{LLM}}^2)$ is the random intercept for LLM k ,
- $b_l^{(\text{metric})} \sim \mathcal{N}(0, \sigma_{\text{metric}}^2)$ 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 manifest context from M to S while controlling for the repeated measures and correlation introduced by datasets, LLMs, and metrics.

The effect of changing the manifest 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. [35].

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 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 5 and 6, 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 5: **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	$M \rightarrow S$	MG, MF, MF^+	68.31%	73.71%	5.28%	4.4×10^{-10}	-	1.8×10^{-1}
	$S \rightarrow S_n$	SG, SF, SF^+	73.71%	73.41%	-0.30%	6.7×10^{-1}	-	3.8×10^{-7}
	$M \rightarrow S_n$	S_nG, S_nF, S_nF^+	73.41%	73.41%	4.98%	4.8×10^{-9}	arcsine	5.0×10^{-2}
	$G \rightarrow F$	MG, SG, S_nG	74.93%	69.37%	-3.97%	2.4×10^{-5}	arcsine	5.6×10^{-2}
	$F \rightarrow F^+$	MF, SF, S_nF	69.37%	72.31%	2.94%	1.5×10^{-4}	-	8.4×10^{-2}
	$G \rightarrow F^+$	MG, SG, S_nG	74.93%	72.31%	-0.86%	3.3×10^{-1}	-	4.4×10^{-6}
Syntactic Validity	$M \rightarrow S$	MG, MF, MF^+	96.87%	97.59%	0.16%	4.5×10^{-1}	box-cox	6.3×10^{-2}
	$S \rightarrow S_n$	SG, SF, SF^+	97.59%	97.56%	-0.03%	9.1×10^{-1}	-	1.5×10^{-2}
	$M \rightarrow S_n$	S_nG, S_nF, S_nF^+	96.87%	97.56%	0.96%	1.1×10^{-1}	-	3.5×10^{-8}
	$G \rightarrow F$	MG, SG, S_nG	99.04%	96.80%	-0.42%	4.3×10^{-1}	-	2.2×10^{-9}
	$F \rightarrow F^+$	MF, SF, S_nF	96.80%	96.74%	-0.06%	9.2×10^{-1}	-	1.0×10^{-9}
	$G \rightarrow F^+$	MG, SG, S_nG	99.04%	96.74%	-0.35%	5.7×10^{-1}	-	2.0×10^{-11}
Semantic Validity	$M \rightarrow S$	MG, MF, MF^+	82.70%	87.87%	5.04%	2.4×10^{-7}	-	5.8×10^{-1}
	$S \rightarrow S_n$	SG, SF, SF^+	87.87%	87.72%	-0.16%	8.4×10^{-1}	-	2.8×10^{-1}
	$M \rightarrow S_n$	S_nG, S_nF, S_nF^+	82.70%	87.72%	4.97%	4.0×10^{-6}	-	3.6×10^{-1}
	$G \rightarrow F$	MG, SG, S_nG	85.21%	86.11%	3.59%	5.8×10^{-4}	arcsine	9.8×10^{-2}
	$F \rightarrow F^+$	MF, SF, S_nF	86.11%	86.89%	0.90%	3.4×10^{-1}	logit	5.7×10^{-1}
	$G \rightarrow F^+$	MG, SG, S_nG	85.21%	86.89%	3.41%	1.3×10^{-2}	logit	6.2×10^{-1}
Relaxed Isomorphism	$M \rightarrow S$	MG, MF, MF^+	48.67%	56.84%	8.11%	1.5×10^{-5}	-	8.3×10^{-1}
	$S \rightarrow S_n$	SG, SF, SF^+	56.84%	56.38%	-0.47%	7.5×10^{-1}	-	1.3×10^{-2}
	$M \rightarrow S_n$	S_nG, S_nF, S_nF^+	48.67%	56.38%	7.58%	8.0×10^{-6}	-	9.0×10^{-1}
	$G \rightarrow F$	MG, SG, S_nG	59.04%	50.66%	-7.64%	1.6×10^{-4}	-	2.1×10^{-1}
	$F \rightarrow F^+$	MF, SF, S_nF	50.66%	54.09%	3.43%	3.9×10^{-2}	-	6.1×10^{-1}
	$G \rightarrow F^+$	MG, SG, S_nG	59.04%	54.09%	-3.03%	1.8×10^{-1}	logit	7.5×10^{-2}
Isomorphism	$M \rightarrow S$	MG, MF, MF^+	45.00%	52.53%	7.34%	1.4×10^{-3}	-	1.4×10^{-1}
	$S \rightarrow S_n$	SG, SF, SF^+	52.53%	51.97%	-0.56%	7.6×10^{-1}	-	6.4×10^{-2}
	$M \rightarrow S_n$	S_nG, S_nF, S_nF^+	45.00%	51.97%	7.28%	1.7×10^{-3}	logit	6.9×10^{-2}
	$G \rightarrow F$	MG, SG, S_nG	56.42%	43.91%	-13.67%	3.4×10^{-14}	-	7.8×10^{-1}
	$F \rightarrow F^+$	MF, SF, S_nF	43.91%	51.51%	7.60%	2.2×10^{-5}	-	1.7×10^{-1}
	$G \rightarrow F^+$	MG, SG, S_nG	56.42%	51.51%	-3.27%	1.2×10^{-1}	-	1.3×10^{-1}

Table 6: **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	Transf-ormation	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}
Syntactic Validity	Claude 3.0 Opus → Gemini 1.5 Pro	73.47%	67.35%	-5.43%	8.6×10^{-8}	-	5.7×10^{-1}
	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}
Semantic Validity	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}
	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}
Relaxed Isomorphism	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}
	GPT4o → Llama 3.1 405B	53.63%	56.83%	3.21%	7.7×10^{-2}	-	6.5×10^{-2}
	GPT4o → Claude 3.0 Opus	53.63%	55.27%	1.84%	5.0×10^{-1}	-	1.1×10^{-1}
Isomorphism	GPT4o → Gemini 1.5 Pro	53.63%	50.67%	-2.96%	2.7×10^{-1}	-	5.2×10^{-1}
	Llama 3.1 405B → Claude 3.0 Opus	56.83%	55.27%	-1.71%	4.7×10^{-1}	-	5.1×10^{-2}
	Llama 3.1 405B → Gemini 1.5 Pro	56.83%	50.67%	-6.17%	1.1×10^{-3}	-	6.3×10^{-1}
	Claude 3.0 Opus → Gemini 1.5 Pro	55.27%	50.67%	-3.89%	1.6×10^{-1}	-	1.8×10^{-1}
	GPT4o → Llama 3.1 405B	51.21%	52.33%	1.26%	5.4×10^{-1}	box-cox	5.4×10^{-2}
Isomorphism	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}