LLM-Based Formalization of Engineering Requirements into Ontology-Constrained Knowledge Graphs

Alessandro Stefanone
Department of Mechanical Engineering,
Politecnico di Milano,
Via La Masa, 1
Milan, Italy
email: alessandro.stefanone@polimi.it

Marco Rossoni[1]
Politecnico di Milano,
Via La Masa, 1
Milan, Italy
email: marco.rossoni@polimi.it

Giorgio Colombo
Politecnico di Milano,
Via La Masa, 1
Milan, Italy
email: giorgio.colombo@polimi.it

Requirements engineering plays a central role in mechanical design, yet technical requirements remain predominantly expressed in natural language, limiting traceability, validation, and computational reasoning. This work presents an ontology-constrained pipeline for transforming natural-language engineering requirements into Industrial Ontologies Foundry (IOF)-grounded knowledge graphs enriched with QUDT-based quantitative semantics. The pipeline decomposes text blocks into individual prescriptive clauses, extracts structural slots and constraint atoms through a typed intermediate representation, normalizes quantitative expressions via QUDT unit and quantity-kind grounding, and instantiates IOF-compliant OWL ABox graphs. The transformation is implemented as a hybrid neuro-symbolic workflow that combines Large Language Models (LLMs) with typed intermediate representations, rule-based post-processing, and description-logic reasoning. Evaluation on a Formula SAE rules corpus, intentionally selected to stress quantitative constraint handling, shows good structural reliability in the evaluated setting, but also highlights important end-to-end limitations. Decomposition accuracy was 63.74%, indicating that clause segmentation remains a major source of propagated error. Conditional on correctly decomposed requirements, slot-level extraction achieved a macro accuracy of 94.50%; however, the stricter row-level criterion requiring all structure and quantity slots to be correct was 69.38%. Quantitative constraint identification reached 97.64% precision and 96.88% recall. Normalization coverage was 98.78%, with residual errors primarily attributable to quantity-kind disambiguation. At the graph level, 93.33% of grounded artifacts passed all ontology-conformance checks. These results indicate that ontology-constrained LLM pipelines can support requirements formalization, while still requiring human oversight for decomposition-sensitive, logically complex, or compliance-critical use cases. The code and data used in this study are openly available in the ontology-req-pipeline GitHub repository.

Keywords: Requirements Engineering, Knowledge Graphs, Ontology, Large Language Models, Engineering Design

1 Introduction

Requirements Engineering (RE) plays a fundamental role in product design by defining the goals, constraints, and performance conditions that delimit the solution space [1]. As formalized in ISO 29148:2018 [2], requirements translate stakeholder needs into prescriptive technical statements that a system must satisfy. Within engineering design theory, requirements encode functional intent and performance constraints that guide the transformation from functions to behaviors and structures [3]. Consequently, requirements constitute a primary source of design knowledge, shaping decision-making throughout the product lifecycle [4].

Despite their central role, technical requirements are predominantly expressed in natural language [5] and still need human interpretation to be managed in industrial practice [6]. This process is labor-intensive, error-prone, and difficult to scale, particularly in early design phases where decisions are made under uncertainty and downstream implications are difficult to anticipate [7]. The absence of explicit, machine-interpretable representations limits the ability to systematically trace constraints, validate consistency, and support computational design reasoning across lifecycle stages.

To address this challenge, researchers have explored Natural Language Processing (NLP) and machine learning techniques for automated requirement analysis, including classification, entity extraction, and ambiguity detection [8,9]. While this class of approaches improves syntactic and lexical processing, it often lacks the semantic grounding required to formally represent requirements as prescriptive design knowledge [10]. More recently, LLMs have demonstrated enhanced contextual understanding; however, they remain prone to inconsistency, limited domain generalization, and hallucinated outputs [11]. In engineering contexts, these limitations are particularly critical, as errors may propagate into downstream system models and compromise design integrity [6].

Bridging this gap requires a semantic layer capable of explicitly representing entities, relationships, and prescriptive constraints in a form suitable for reasoning and validation [12]. Knowledge Graphs (KGs) provide structured representations of domain concepts and their relations, but their effectiveness depends on robust ontological grounding. In engineering design, task-specific schemas often lack the semantic expressiveness needed to represent concepts unambiguously, which can result in redundant or inconsistent data representations. Using an ontology to constrain admissible node types and relations provides a guardrail for KG construction: it governs schema evolution, enforces a shared vocabulary, and limits the proliferation of overlapping classes and properties. Because top-level ontologies are typically too abstract to capture design-relevant distinctions, mid-level industrial ontologies, such as the Industrial Ontologies Foundry (IOF) Core ontology [13], provide

[1]Corresponding Author.
June 25, 2026

a more suitable foundation by formally modeling information content entities, material artifacts, processes, and prescriptive specifications. However, engineering requirements frequently contain quantitative constraints whose semantics depend on explicit modeling of units, dimensions, and quantity kinds. For this reason, integrating a complementary ontology for quantitative semantics, such as QUDT (Quantities, Units, Dimensions and Types) [14], is essential to ensure consistent representation of measurable constraints. Grounding requirements within such an integrated ontological framework enables consistent interpretation, logical reasoning, quantitative normalization, and lifecycle traceability. As a matter of fact, recent research ontology-based product lifecycle management further highlights the importance of formally structured knowledge frameworks for managing design information across lifecycle stages [15,16]. These approaches demonstrate the value of ontological modeling for lifecycle integration and traceability; however, they typically assume that structured ontology-aligned artifacts already exist. The systematic transformation of natural-language engineering requirements into such ontology-grounded representations remains largely unaddressed.

Building on this perspective, this work proposes a formally defined, ontology-constrained methodology for transforming natural-language technical requirements into IOF-grounded KGs. The pipeline integrates four stages: (i) semantic decomposition of requirement clauses, (ii) structured extraction of prescriptive constraints through a typed intermediate representation, (iii) quantitative normalization via QUDT-based unit and quantity-kind grounding, and (iv) ontology-aligned instantiation with automated reasoning and structural conformance validation. The proposed approach combines the flexibility of LLMs in interpreting natural-language requirements with ontology-based semantic grounding and validation, thereby transforming technical requirements into a computable knowledge graph that can be used operationally for consistent downstream analysis. Empirical results indicate that such ontology-constrained LLM pipelines can achieve high structural reliability in engineering-grade requirement formalization. This evaluation uses a multi-level protocol on a stratified requirements corpus, measuring slot-level extraction accuracy, quantitative normalization correctness, semantic quality of the generated graphs, and ontology-conformance of the resulting KGs. The resulting representation largely preserves prescriptive structure in the evaluated cases, supports consistency checking, and provides a semantic backbone for reasoning, traceability, and downstream design-support applications. At the same time, the evaluation shows that reliability is not uniform across pipeline stages: decomposition and strict row-level correctness remain substantially weaker than slot-level extraction metrics, so the reported results should be interpreted as evidence of promising formalization support rather than fully autonomous requirement understanding.

In methodological terms, the contribution is not a new standalone extraction model or a new ontology, but a neuro-symbolic formalization framework that connects probabilistic parsing to deterministic ontology-enforced graph construction. The key advance is the introduction of an explicit semantic contract between these stages through the typed intermediate representation and the subsequent grounding-and-repair procedure. The engineering-requirements setting serves as the motivating application domain in which this framework is defined and evaluated. The empirical evaluation therefore consists of a main single-domain benchmark (FSAE) and a small cross-domain ablation intended as an initial probe rather than as evidence of broad generalization.

The remainder of this paper is organized as follows. Section 2 reviews related work in requirements engineering, natural language processing, and ontology-based knowledge representation, positioning the proposed contribution within neuro-symbolic approaches for engineering design. Section 3 formalizes the research gap and outlines the main contributions of the study. Section 4 defines the task and presents the end-to-end methodology, including decomposition, structured extraction through a typed intermediate representation, QUDT-based quantitative normalization, and IOF-constrained KG grounding with reasoning and validation. Section 5 reports the experimental evaluation on a Formula SAE rules document, including extraction performance, normalization results, graph inference outcomes, ontology-conformance analysis, and runtime profiling. Section 6 discusses the implications of the findings for mechanical design applications, analyzes representative success and failure cases, and outlines deployment considerations and limitations. Finally, Section 7 concludes the paper and identifies directions for future research in ontology-grounded AI for engineering design. To support transparency and reproducibility, the code and evaluation datasets used in this study are publicly available on GitHub[2].

2 Related Work

This section reviews prior work relevant to the formalization of prescriptive design knowledge from natural-language requirements. We first summarize foundational perspectives that position requirements as carriers of design intent, constraints, and rationale in engineering design. We then review automated requirements analysis approaches, progressing from traditional NLP and machine learning methods to LLM-based techniques, and discuss their limitations with respect to semantic consistency and downstream reasoning. NLP and LLM approaches are reviewed before ontology and KG approaches because the former address extraction from text, whereas the latter provide the semantic grounding layer needed to formalize the extracted content. Finally, we examine ontology and KG-based representations, as well as recent neuro-symbolic approaches, as semantic grounding mechanisms that enable reasoning and validation.

2.1 Requirements as Prescriptive Design Knowledge. Engineering design theory consistently positions requirements as carriers of functional intent and constraints that guide solution development. Axiomatic Design [17] defines functional requirements as the minimal independent set of conditions a design must satisfy, while Gero's Function-Behaviour-Structure framework [3] conceptualizes design as the transformation of functional expectations into structural realizations. Within these perspectives, requirements encode prescriptive goals and performance constraints independently of specific implementations. Related work in engineering design has similarly emphasized structured representations of design intent through functional modelling; for example, the Functional Basis tradition formalizes product functions using standardized verb–object descriptions over flows of material, energy, and signal [18].

Despite their central role, requirements are typically represented informally and dispersed across heterogeneous documents, limiting traceability and computational integration in early design stages [19]. While prior work has emphasized knowledge modeling across process and concurrent engineering views [20], the formalization of requirements as machine-interpretable prescriptive entities remains limited.

In other words, requirements have long been treated as structured design knowledge within engineering theory, yet are predominantly processed as textual artifacts within AI and NLP pipelines (see Section 2.2). Accordingly, although engineering design theory already treats requirements as structured carriers of intent and constraints, most AI and NLP workflows still process them primarily as text. This gap motivates the need for ontologically grounded representations that can make requirement semantics explicit and support reasoning, validation, and lifecycle traceability.

2.2 NLP and Machine Learning for Requirements Engineering. Despite sustained progress, most RE approaches have been primarily aimed at supporting manual document review rather than enabling the formalization of requirements into prescriptive

[2] https://github.com/alessandrostefanone-polimi/ontology-req-pipeline

design knowledge. Up to recent years, NLP-based approaches have been dominated by reliance on traditional techniques such as tokenization, part-of-speech tagging, parsing, and rule-based feature extraction [5].

Requirements templates, such as EARS [21] and Rupp's templates [22], were introduced to reduce ambiguity and improve consistency in requirements specification. While template-based approaches improve readability and standardization, they rely on predefined syntactic patterns and shallow semantic interpretations, limiting their applicability to complex industrial requirements and preventing the explicit representation of design constraints in a form suitable for reasoning. Tiwari et al. [23] developed a rule-based tool to translate requirements into EARS and Rupp's template, demonstrating how this methodology enhances both readability and disambiguation. However, their approach proved inefficient for the complex sentences typical of industrial scenarios, as their rule-based method relies on shallow semantics. Unlike structured documentation templates, the representation proposed in this study makes entity types, semantic relations, and quantitative-value structures explicit in a machine-interpretable graph form, thereby enabling graph querying beyond document organization alone.

While NLP and machine learning approaches have significantly improved the automated analysis of requirements text, they primarily operate at the syntactic or statistical semantic level and do not address the formalization of requirements as prescriptive design knowledge suitable for reasoning and validation. Recent advances in machine learning, including word embedding models [24] and neural network architectures such as long short-term memory, recurrent, and convolutional networks, have enabled more effective pattern recognition in requirements text [25]. In particular, domain-adapted embeddings improved performance in classification tasks, as demonstrated by successful applications in the automotive domain [26].

To automate downstream analysis, researchers and practitioners increasingly shifted from document review to information extraction, converting requirements into structured representations; however, these outputs typically remain not ontologically grounded, which limits formal reasoning and consistency checking. The introduction of transformer architectures and attention mechanisms [27] enabled context-aware representations of requirements text, paving the way for LLMs that operate at a higher semantic abstraction level.

2.3 Large Language Models for Requirements and Design Tasks. Building on advances in transformer architectures, LLMs have recently been explored for requirements engineering and design-related tasks due to their ability to capture richer semantic patterns from natural-language text. Hou et al. [28] surveyed the application of LLMs in requirements engineering, identifying their use across a range of tasks including ambiguity resolution, requirements classification, term identification, coreference detection, and traceability automation. Compared to traditional NLP techniques, transformer-based pretrained language models (e.g., BERT [29]) and subsequent generative LLMs demonstrate superior performance in extracting contextual information by leveraging bidirectional attention and large-scale pretraining.

Several studies have investigated the use of LLMs to improve task-level performance in requirements engineering workflows. Lim [11] evaluated fine-tuned LLMs for systems engineering tasks such as functional versus non-functional requirements classification, non-functional requirements sub-type categorization, and named entity recognition for structured data mapping. While the results confirmed the effectiveness of LLMs when adapted to domain-specific data, the study also highlighted persistent challenges related to the limited availability of labeled datasets and reduced generalization across domains. Similarly, Gräßler et al. [30] employed generative language models to produce augmented datasets for requirements classification, showing that data augmentation improves performance within a given domain but degrades significantly when applied cross-domain, underscoring the contextual sensitivity of LLM-based approaches. Data augmentation and adaptation strategies are particularly relevant for LLM-based requirements engineering tasks. In this respect, Schiller et al. [31] showed that fine-tuned models can achieve improved performance in the generation of requirements documents, further highlighting the importance of task-specific data and adaptation when applying LLMs in requirements-related settings.

Beyond classification and labeling tasks, recent research has focused on exploiting LLMs to increase the machine readability of technical requirements by translating natural-language Statements into structured or semi-formal representations. Holter and Ell [32] proposed a semantic decomposition of requirements into scope, condition, and demand components, demonstrating that LLMs can identify meaningful semantic elements within engineering requirements. Their work suggests that linking extracted entities to a formal knowledge representation could enable the identification of common conceptual patterns across requirements. Perko and Wotawa [33] investigated the translation of natural-language requirements into Answer Set Programming (ASP) representations using LLMs, showing promising results but also reporting issues in syntactic correctness and semantic consistency. Norheim and Rebentisch [34] explored the use of generative LLMs for rephrasing requirements into generic templates and semi-formal logic, reducing reliance on handcrafted rules but requiring substantial expert intervention to ensure reliable and consistent outputs.

Prompt engineering and few-shot prompting techniques have further demonstrated that providing structured contextual information can significantly improve LLM performance across requirements elicitation, specification, analysis, and validation tasks [35]. In the design domain, multimodal LLMs have also been evaluated for document understanding and information retrieval tasks. Doris et al. [36] introduced the DesignQA benchmark to assess generative models on technical documentation, highlighting how contextual grounding strongly influences performance. However, Regenwetter et al. [37] showed that state-of-the-art LLMs still struggle with constrained engineering design problems and structured or tabular data, revealing limitations in their ability to reason over formally constrained domains. These limitations are not confined to requirements engineering. Recent work has also evaluated LLMs in conceptual design decision tasks such as material selection [38], comparing model recommendations against expert preferences across multiple scenarios. The study reports low variance of recommendations across distinct design contexts and a tendency to overestimate material appropriateness, underscoring persistent challenges in aligning generative LLM outputs with expert engineering judgment in downstream design decisions.

Results obtained by existing studies demonstrate that LLMs significantly advance the extraction of implicit semantics from requirements text and enable the generation of structured representations beyond the capabilities of traditional NLP and machine learning approaches. Nevertheless, these representations are typically task-specific, ad hoc, or weakly formalized, and do not provide explicit, ontologically grounded models of prescriptive design knowledge suitable for constraint propagation, lifecycle traceability, and formal design validation. The limitations observed in state-of-the-art applications of LLMs to requirements engineering and design tasks therefore motivate the need to ground the reasoning and generation capabilities of language models in an explicit semantic layer.

2.4 Knowledge Graphs and Ontologies for Design Knowledge Representation. Ontologies and KGs have long been adopted in industrial engineering as a means of organizing complex and heterogeneous data in a semantically consistent manner. By establishing shared vocabularies and formal relationships across organizational functions, ontology-based approaches support interoperability, traceability, and information integration throughout the product lifecycle [39]. In the manufacturing domain, early efforts such as the Manufacturing Reference Ontology (MRO) proposed

by Usman et al. [40] demonstrated how formal semantic models can mitigate ambiguity and improve interpretability across diverse industrial contexts.

Within the engineering design domain, substantial research has focused on developing formal representations of design knowledge to systematize the product development process. Ontology-based approaches have been proposed to structure requirements, functions, and design artifacts coherently, supporting traceability and consistency across design phases. For instance, Lin et al. [41] introduced an ontology to formalize technical requirements and their dependencies, highlighting the central role of requirements as carriers of goals and constraints in systems engineering. Similarly, Colombo et al. [42] emphasized that design within functional constraints relies on the interplay between ontological knowledge, which captures static concepts and relationships, and procedural knowledge, which governs the dynamic execution of design activities.

Notwithstanding extensive prior work on ontology-based representations for design, most engineering ontologies remain manually authored, domain-specific, and weakly connected to large volumes of unstructured requirements artifacts produced in industrial practice. This limitation is increasingly problematic in data-driven and AI-assisted design contexts, where scalable and machine-interpretable representations of prescriptive knowledge are required. The Industrial Ontologies Foundry (IOF) addresses this gap by providing explicit modeling constructs for requirement specifications, design specifications, and satisfaction relations [13]. Built upon the Basic Formal Ontology (BFO) [43], IOF ensures domain-independent semantic commitments and logical coherence. In this work, IOF is adopted as the structural grounding backbone for requirement formalization.

Model-Based Systems Engineering (MBSE) provides a complementary route toward machine-interpretable requirements by shifting system descriptions from documents to model-based artifacts, most commonly expressed in SysML [44]. Although SysML standardizes constructs for capturing requirements and tracing them to functional and architectural elements, SysML models remain largely semi-formal: their semantic interpretation is frequently tool and profile-dependent, and automated reasoning is typically limited to tool-specific validation rules rather than logic-based inference over shared semantics [44]. In this context, a constrained IOF-grounded knowledge graph can act as an explicit semantic substrate that complements MBSE by making ontological commitments explicit and enabling logic-based validation and constraint reasoning over prescriptive design knowledge [13].

In parallel, recent research has explored neuro-symbolic approaches that combine symbolic knowledge representations with neural models to improve reasoning, robustness, and interpretability [12]. Hybrid architectures have emerged as a promising strategy to combine scalable language processing with formally structured design knowledge. Approaches such as GraphRAG [45] exemplify the use of KGs as semantic indices to provide structured context for language model generation. In contrast to taxonomies or task-specific schemas, ontologically grounded KGs provide explicit semantic commitments that support validation and reasoning over prescriptive constraints. Recent work on context graphs and context-engineering workflows similarly highlights the value of structured semantic context for constraining and improving language-model outputs [46].

In the context of engineering design, several recent studies have shown that incorporating structured knowledge representations can significantly improve the coherence and consistency of AI-assisted design processes. Massoudi and Fuge [47] introduced the Design State Graph (DSG) to encode procedural design knowledge acquired by agentic language models, demonstrating improved logical consistency across design phases. Similarly, Pan et al. [48] leveraged a KG derived from user needs to constrain LLM-based conceptual design generation, achieving superior results compared to ungrounded generative models. These studies highlight the effectiveness of semantic backbones in reducing redundancy and enforcing design consistency, but they often rely on manually curated or domain-specific knowledge representations.

Recent work has also shown that knowledge graphs can support the organization and retrieval of experiential design knowledge in engineering design contexts, for example by structuring teardown-derived observations and designer-generated relations into interactive graph representations [49]. However, such approaches primarily focus on organizing experiential design knowledge rather than formalizing prescriptive natural-language requirements into representations structured for ontology-based validation and reasoning.

Existing research confirms the importance of ontologies and KGs for representing design knowledge and grounding AI-based reasoning. Nevertheless, a key gap remains in the automated extraction and formalization of prescriptive design knowledge from natural-language requirements into ontologically grounded representations. Addressing this gap is essential to enable scalable, trustworthy, and reasoning-capable AI systems for engineering design.

3 Research Gap and Main Contribution

Existing research in requirements engineering and AI-assisted design can be broadly grouped into two streams. First, NLP and machine-learning approaches improve syntactic analysis and semantic extraction from requirements text, but typically stop short of producing formally grounded, reasoning-ready representations. Second, ontology- and knowledge-graph–based approaches provide the semantic rigor required for validation and multi-hop reasoning, yet are often manually curated, domain-specific, and weakly connected to large volumes of unstructured industrial requirements artifacts. A methodological gap therefore persists: there is no systematically defined, ontology-constrained transformation pipeline that converts natural-language engineering requirements into logically coherent, mid-level industrial KGs with explicit quantitative semantics and structural validation.

The primary contribution of this work is the overall transformation framework rather than any single component in isolation. Specifically, the novelty lies in coupling LLM-based requirement parsing with a typed intermediate representation, deterministic quantitative normalization, ontology-constrained ABox grounding, and logic-based conformance enforcement within a single auditable pipeline. Among these elements, the most distinctive methodological contribution is the grounding-and-validation workflow that converts extracted requirement semantics into ontology-committed OWL instance graphs while constraining probabilistic generation through explicit semantic contracts.

This work addresses that gap by introducing a formally specified, staged transformation pipeline that integrates language-model–based semantic parsing with deterministic ontological grounding under IOF and QUDT commitments. Unlike prior LLM-based structuring approaches, the proposed method does not merely reformat requirements into templates or semi-formal logic. Instead, it produces OWL ABox graphs that (i) reuse an established industrial ontology, (ii) enforce modeling constraints through validation and reasoning, and (iii) preserve quantitative semantics through explicit unit and quantity-kind normalization.

The core contributions of this study, taken together as a single ontology-constrained requirement-formalization framework, are:

- A formally defined end-to-end transformation $\Phi = \gamma \circ \nu \circ \varepsilon \circ \delta$ that maps natural-language requirement clauses into ontology-grounded KG instances without introducing new schema elements.

- A typed intermediate representation that bounds LLM generation, supports traceability through explicit evidence spans, and serves as a semantic contract between probabilistic extraction and deterministic grounding.

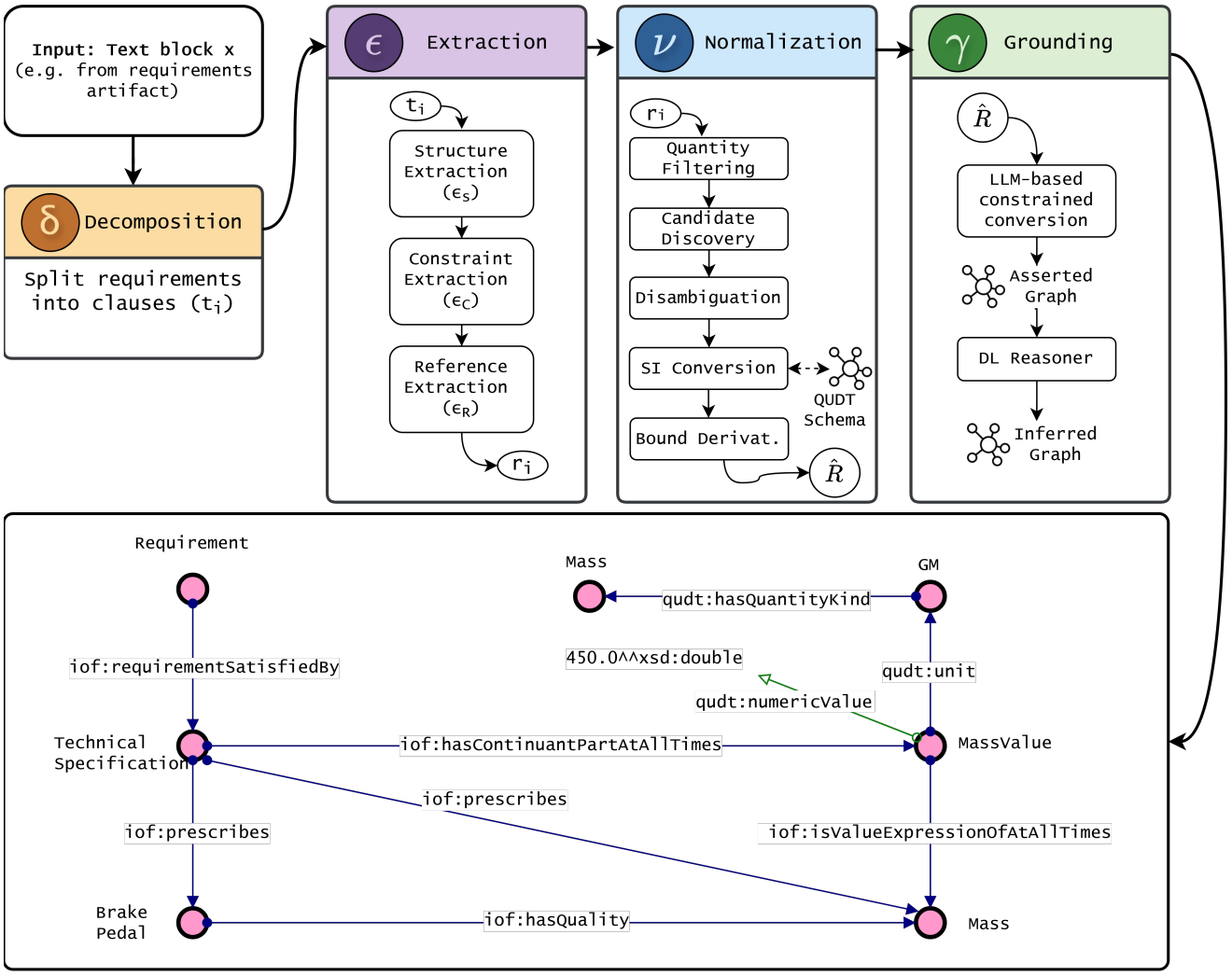

Fig. 1 Overview of the pipeline for transforming textual requirements into an ontology-based graph.

- A QUDT-aligned quantitative normalization mechanism that resolves surface-form units and quantity expressions into dimensionally consistent, ontology-compatible representations.
- A hybrid grounding workflow combining LLM graph proposal, deterministic sanitization, and description-logic reasoning to enforce IOF structural commitments and detect modeling inconsistencies.
- An empirical evaluation on a quantitatively dense engineering corpus showing strong slot-level and graph-conformance performance, while also identifying decomposition and row-level correctness as the main bottlenecks for end-to-end reliability.

From a mechanical design perspective, the contribution is not merely linguistic automation, but the establishment of a semantically grounded bridge between textual requirements artifacts and formally structured design knowledge. This bridge enables logical validation, constraint traceability, and reasoning-ready representations suitable for downstream design-support workflows. The empirical study is structured around the following research questions:

- **RQ1**. Is it possible to transform natural-language engineering requirements into IOF-grounded OWL ABox graphs through a formally staged workflow?
- **RQ2**. How reliably does the pipeline preserve requirement structure and quantitative semantics across decomposition, extraction, and normalization phases?
- **RQ3**. How structurally valid are the generated graphs with respect to the ontology-conformance constraints adopted in this work?

4 Methodology

This section formalizes the task of transforming natural-language technical requirements into IOF-grounded KGs and details the end-to-end pipeline that implements this transformation. It is important to note that the pipeline is not deterministic end-to-end: decomposition, extraction, selected normalization decisions, and the initial graph proposal depend on LLM outputs, whereas schema validation, quantitative conversion, sanitization, conformance checks, and fallback logic are applied through deterministic post-processing once candidate outputs have been produced. Figure 1 shows the general idea.

4.1 Document preprocessing and corpus preparation. The input to the proposed pipeline is not a raw PDF artifact itself, but a text-based intermediate representation derived from the source document. In the present implementation, source documents were first converted through an OCR-based document conversion step

that produces page-level textual transcriptions together with preserved page boundaries. The resulting text was then segmented into sentence-like spans and stored with provenance metadata, including document identifier, page number, and sentence index. Candidate requirement statements were subsequently selected through modal-expression filtering and retained as JSONL records for downstream processing. This preprocessing was intentionally designed to preserve realistic document noise rather than produce a manually normalized gold transcription. As a result, the benchmark reflects the kind of imperfect textual input commonly encountered in industrial document-processing pipelines, where extracted spans may still contain layout artifacts, section numbering, list markers, or local formatting residue. The semantic formalization pipeline introduced in the following sections therefore operates on PDF-derived text spans rather than on manually curated requirement clauses. Because this conversion stage is OCR- and layout-driven rather than semantics-aware, it introduces several limitations. Complex formatting structures such as nested bullet lists, tables, equations, superscripts or subscripts, multi-line rules, and cross-references were imperfectly linearized.

4.2 Task Definition and Goals. The first methodological assumption is that the requirements are expressed in English natural language. While alternative semantic decompositions of requirements have been proposed (e.g., [32]), the task is hereby defined according to ISO 29148:2018 [2], which frames a requirement as a prescriptive Statement that constrains a system or system element under specified conditions and within a defined scope. In this work, ISO/IEC/IEEE 29148:2018 is adopted to define the notion of a requirement at the natural-language level, whereas IOF Core is adopted only at the ontological grounding stage to represent requirement-related informational artifacts as OWL individuals. The proposed pipeline therefore acts as a transformation layer between ISO-style requirement statements and IOF-grounded specification structures.

Let Σ^* denote the set of finite strings over an alphabet Σ. An input text block $x \in \Sigma^*$ denotes a contiguous span of text extracted from a requirements artifact (e.g., a product requirements document). In the present study, such text blocks are obtained from PDF documents transcriptions rather than from manually curated requirement statements. Accordingly, x should be interpreted as a realistic extracted text span whose boundaries and surface form may still reflect upstream document-conversion artifacts. In industrial practice, a single text block may contain multiple prescriptive clauses due to formatting conventions, company-specific writing rules, or embedded lists. Therefore, the first goal of the methodology is to decompose x into a finite sequence of individual requirement clauses.

Each individual requirement clause is represented by the tuple in Eq. 1:

$$r = \langle s, m, \kappa, a, o, C, R \rangle \tag{1}$$

where s is the subject of the requirement (the system, component, or entity being specified), m is the modality marker (e.g., *shall*, *must*), κ denotes the condition or scope under which the prescription applies, a is the prescriptive action or predicate asserted of the subject, o is the object argument of a (i.e., the entity that the prescribed action is directed toward or specified for, when explicitly stated), C is the list of constraints extracted from the clause, and R is the set of external references (e.g., references to tables, figures, other documents, or requirement identifiers).

To make Eq. 1 concrete, consider the requirement: "During braking, the pedal shall maintain a minimum clearance of 10 mm from the chassis, as specified in Table 3." In this case, the subject is the pedal (s), the modality is shall (m), the condition/scope is during braking (κ), the prescribed action is maintain (a), and the object is clearance (o). The constraint set C includes the quantitative minimum-bound constraint on the clearance value (minimum 10 mm) and its relational target (from the chassis), while R includes the external reference "Table 3".

In accordance with ISO 29148:2018, constraints in C may attach to different clause elements (subject, object, action, condition/scope, or modality); consequently, constraint targets are not fixed at the tuple level; rather, they are explicitly captured in the extracted constraint structures produced by the extraction stage.

At this point, the overall task can be formalized as semantic parsing into an ontologically constrained ABox: mapping natural-language requirements into an instance-level KG that reuses the vocabulary and modeling commitments of a mid-level industrial ontology, enabling logical reasoning and validation.

Let $\mathcal{G}$ denote the set of admissible KGs. The mapping function $\Phi : \Sigma^* \to \mathcal{G}$ is modeled as a composition of stage-wise functions:

$$\Phi(x) = (\gamma \circ \nu \circ \varepsilon \circ \delta)(x) \tag{2}$$

where:

- δ (*decomposition*) splits a text block into individual requirement clauses;
- ε (*extraction*) produces structured requirement frames, constraint atoms, and external references, each with evidence spans in the source text;
- ν (*normalization*) normalizes quantity strings to a representation that uses QUDT Units and Quantity Kinds vocabularies [14];
- γ (*grounding*) instantiates IOF-grounded graph patterns from the extracted frames and normalized constraint structures.

To account for multi-clause text blocks, δ is defined to return a finite sequence of clauses:

$$\delta : \Sigma^* \to \mathcal{R}^* \tag{3}$$

where $\mathcal{R}$ is the set of individual requirement clauses. The resulting KG is obtained by grounding each clause and merging the produced instance assertions:

$$\Phi(x) = \bigcup_{r_i \in \delta(x)} (\gamma \circ \nu \circ \varepsilon)(r_i). \tag{4}$$

From a practical perspective, decomposing the task into modular functions was aimed at reducing error propagation and supporting maintainability: probabilistic components (LLM-based decomposition, extraction, and selected normalization/grounding decisions) are constrained to typed intermediate representations, while deterministic post-processing and logical reasoning enforce structural consistency.

The output graph $G \in \mathcal{G}$ is an OWL instance graph (ABox) that reuses classes and properties from the IOF Core ontology [13] and a subset of the classes and relations provided by QUDT [14], following the guidelines provided by [50] and detailed in Section 4.6. As such, the result of the pipeline does not define new ontology classes; rather, the task is to create instances (individuals) of already existing ontology classes. Formally, a graph is defined as:

$$G = (V, E), \tag{5}$$

where V are the nodes while E are the edges. In the context of this work, the former is a finite set of Resource Description Framework (RDF) resources (identified by IRIs), the latter a finite set of RDF triples. As each edge should be either between two resources (e.g., `:Req123 iof:hasSubject :Motor_1`) or between a resource and a literal (e.g., `:Quantity_1 qudt:numericValue "10.0"^^xsd:double`), only two kinds of assertions are allowed:

$$E \subseteq (V \times P \times V) \cup (V \times D \times L), \tag{6}$$

where P is a set of object properties, D is a set of datatype properties, and L is the set of literals. Let C_{IOF} and R_{IOF} denote

the sets of classes and object properties defined in IOF Core. Let $C^*_{QUDT} \subseteq C_{QUDT}$ and $R^*_{QUDT} \subseteq R_{QUDT}$ denote the task-specific subsets of QUDT classes and properties admitted by the pipeline (units, quantity kinds, and value patterns). Every individual must be explicitly typed using the predicate `rdf:type`; hence for each individual $v \in V$ there exists at least one class $c \in C_{IOF} \cup C^*_{QUDT}$ such that $(v, \texttt{rdf:type}, c) \in E$. Likewise, all object-property assertions use relations drawn exclusively from $R_{IOF} \cup R^*_{QUDT}$.

4.3 Decomposition (δ): Splitting into Individual Requirements. The decomposition function δ partitions an input text block $x \in \Sigma^*$ into a finite sequence $(t_1, \ldots, t_n)$, where each t_i corresponds to a textual clause expressing exactly one prescriptive obligation. Each resulting clause is intended to contain a single modality marker and its associated predicate, in accordance with the definition of a requirement in ISO 29148:2018.

The decomposition process is implemented using an LLM guided by a structured prompt and few-shot examples to identify appropriate splitting boundaries. Specifically, the language model is instructed to split a text block into individual requirements when any of the following conditions hold:

- multiple distinct prescriptive modalities (e.g., *shall, must, should, will, is required to*) govern different predicates;
- coordinated predicates express independent obligations;
- bullet or numbered list items each encode a standalone obligation.

Conversely, the prompt instructs the model not to split clauses when qualifiers constrain a single obligation. Such qualifiers include temporal or conditional scopes (e.g., "when X", "during Y"), tolerance specifications (e.g., "±10%"), and parenthetical refinements or unit specifications.

The decomposition output is constrained to a typed JavaScript Object Notation (JSON) schema containing requirement indices and clause texts. The overall procedure implementing δ is summarized in Algorithm 1.

Algorithm 1 Decomposition of text blocks into individual requirements (δ)

Require: Input text block $x \in \Sigma^*$
Ensure: List of indexed individual requirement clauses $\{t_i\}_{i=1}^n$
1: Initialize prompt template with few-shot examples
2: Construct prompt P_δ with input text x
3: Query LLM with P_δ and structured output schema `IndividualRequirementsResponse`
4: Parse LLM response into list `individual_requirements` of `(req_idx, text)` chunks
5: **return** $\{t_i\}_{i=1}^n$

4.4 Extraction (ε): Structured Information Extraction. The extraction stage transforms each individual requirement clause t_i into a structured representation $r = \langle s, m, \kappa, a, o, C, R \rangle$. This process is decomposed into three sub-tasks, each handled by a specialized LLM call with task-specific prompts and constrained decoding to a specified JSON schema. To make explicit the data dependencies between sub-tasks, intermediate outputs are defined as described in Eqs. 7–10.

$$\varepsilon^i_s(x) := \varepsilon_s(t_i, x), \tag{7}$$

$$\varepsilon^i_C(x) := \varepsilon_C\big(t_i, x, \varepsilon^i_s(x)\big), \tag{8}$$

$$\varepsilon^i_R(x) := \varepsilon_R\big(t_i, x, \varepsilon^i_s(x), \varepsilon^i_C(x)\big), \tag{9}$$

$$\varepsilon(t_i, x) := \big\langle \varepsilon^i_s(x),\ \varepsilon^i_C(x),\ \varepsilon^i_R(x) \big\rangle, \tag{10}$$

where ε_s is the structure extraction function, ε_C is the constraint extraction function conditioned on structure, and ε_R is the reference extraction function conditioned on structure and constraints. The final requirement instance r is obtained by assembling the fields yielded by $\varepsilon^i_s(x)$, $\varepsilon^i_C(x)$, and $\varepsilon^i_R(x)$.

4.4.1 Extraction Data Model and Controlled Vocabularies. Now that ε has been formalized as a staged functional transformation, the correctness and interoperability of its outputs depend on a strictly defined intermediate representation (IR). Before detailing the three sub-tasks of ε, we therefore introduce the data model that constrains its outputs. This IR serves as the formal interface between extraction, normalization, and grounding. The data model shown in Figure 2 is a task-specific intermediate representation introduced in this work. It is not taken directly from IOF Core or QUDT, nor is it intended as a standalone ontology of engineering requirements. Rather, it acts as a typed contract between extraction, normalization, and grounding: it constrains LLM outputs, preserves evidence spans for traceability, and ensures compatibility with the ontology-constrained instantiation stage. Its design is informed by the clause structure of ISO 29148 requirements and by the practical need to bridge probabilistic extraction with rule-based validation and downstream IOF/QUDT grounding.

At record level, the extractor returns

$$R = \langle idx,\ source,\ x,\ \mathcal{R} \rangle, \tag{11}$$

where x is the original input text block and $\mathcal{R}$ is an ordered list of individual requirements. The structural frame S_i comprises the ISO-aligned slots `subject`, `modality`, `condition`, `action`, and `object`. Span-bearing fields are represented as character-offset tuples over x (0-indexed, end-exclusive), ensuring that every extracted element remains explicitly traceable to its source evidence. The `condition` object additionally includes an EARS-style pattern ([21]) label used to classify applicability contexts (`ubiquitous`, `event-driven`, `state-driven`, `unwanted_behaviors`, `optional_features`).

The intermediate representation of each constraint c_j supports clause-internal logic (`group` with `AND/OR`), explicit attachment semantics (`target`), provenance (`evidence`), and hierarchical modifier chains (`depends_on`). Qualifiers encode polarity and scope (`negated`, `preferred`, `scope_all`).

The value field is typed as a discriminated union to preserve semantic compatibility between operators and operands: quantitative values (`ValueQuantity`), enumerations (`ValueEnum`), booleans, and entity/event references (`ValueRef`), with a `none` fallback for constraints that are structurally valid but do not expose an explicit value surface form. References are represented independently through `Reference` objects with evidence spans, expected IOF-compatible type, and optional resolution field.

The IR uses closed vocabularies to reduce lexical variability and support rule-based validation after probabilistic extraction. The main vocabularies (see Appendix A) are: `ModalityVocab`, `OperatorVocab`, `AttributeKindVocab`, `ValueKindVocab`, target-kind vocabulary (`subject`, `object`, `action`, `condition`, `modality`), and reference expected-type vocabulary (`MaterialEntity`, `Process`, `Quality`, `InformationContentEntity`, `Unknown`). This typed contract is the primary mechanism used to maintain extraction robustness while preserving direct compatibility with the QUDT normalization stage (Section 4.5) and subsequent IOF grounding.

4.4.2 Structure Extraction (ε_s). The first sub-task extracts the core structural elements of an individual requirement clause t_i, i.e. subject (s), modality (m), condition/scope (κ), action (predicate) (a), and, when explicitly stated, object (o). These slots reflect the semantic components defined by ISO 29148 and correspond

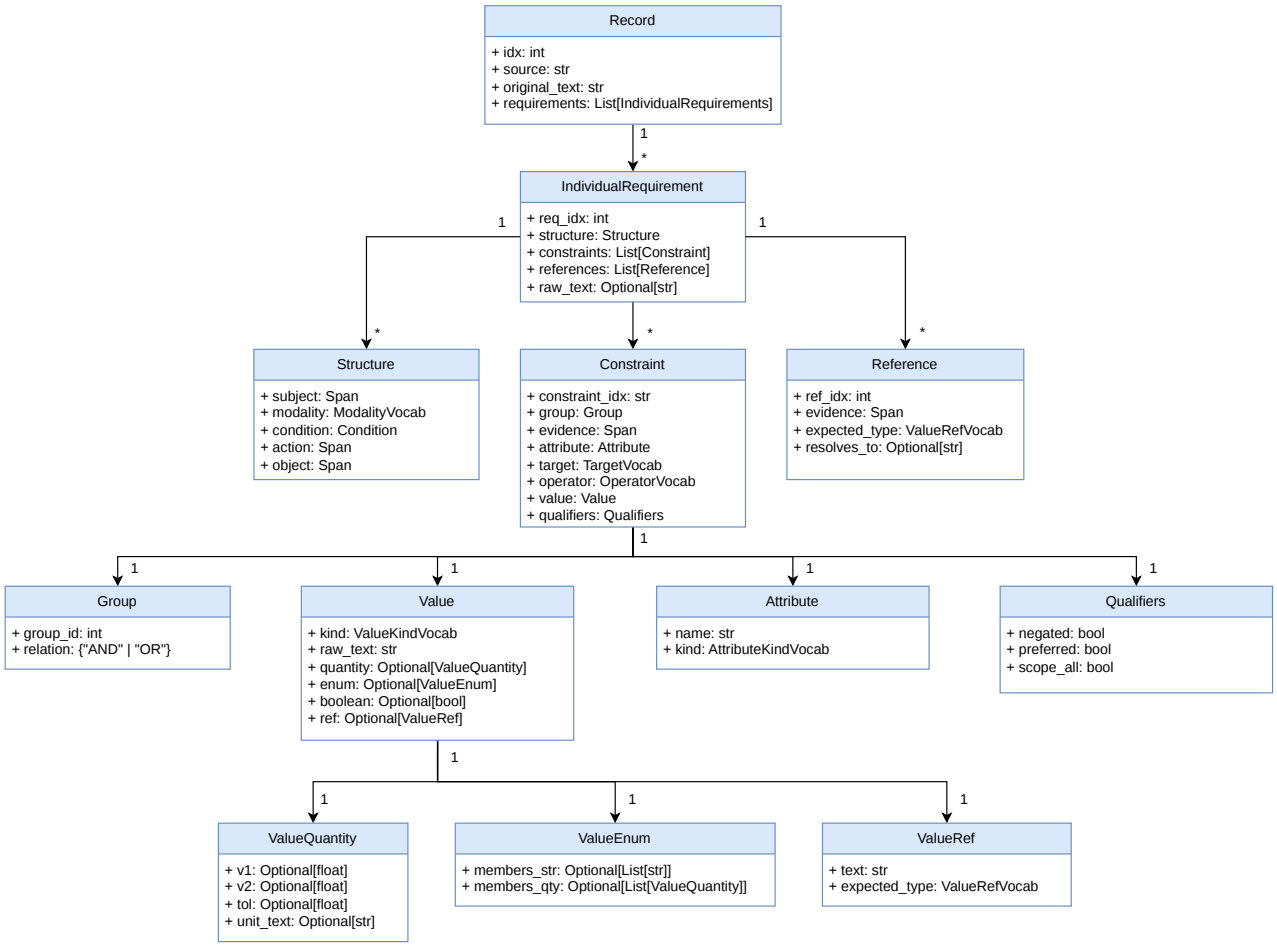

Fig. 2 Data model used for checking extraction outputs against a structured intermediate representation

to the syntactic structures operationalized in widely used requirements templates [21,22], thereby providing a standardized foundation for downstream constraint extraction, normalization, and ontology grounding. Each extracted element is linked to an explicit evidence span, reported as character offsets relative to the original text block x, ensuring full traceability.

Algorithm 2 Structure extraction for an individual requirement (ε_s)

Require: Individual requirement text t_i, original text x
Ensure: Structure tuple $\mathcal{Z} = \langle s, m, \kappa, a, o \rangle$ with text spans
1: Construct prompt P_{ε_s} with t_i and x
2: Query LLM with P_{ε_s} and schema `StructureResponse`
3: $\langle s, m, \kappa, a, o \rangle \leftarrow$ Parse response field `structure` into `Structure` schema
4: **return** $\mathcal{Z} = \langle s, m, \kappa, a, o \rangle$

4.4.3 Constraint Extraction (ε_C). The second sub-task identifies the prescriptive constraints expressed in an individual requirement clause. In the scope of this study, a constraint is an information content entity that restricts the design solution or implementation of the target product or system [2]. Each extracted constraint is modeled as a constraint atom, i.e., a minimal, self-contained statement that constrains an attribute, relation, or event with an explicit operator and value, and that can be grounded and validated independently downstream. Constraint atoms are extracted by analyzing the spans returned by ε_s (for subject, condition/scope, action, object, and modality) and by linking each atom to an evidence span in the original text block x. In accordance with ISO 29148, a constraint may apply to different clause elements; therefore, each atom also records its attachment target (subject-, object-, action-, condition-, or modality-scoped), as determined from the local syntactic and semantic context.

Formally, each atom $c \in C$ is represented as

$$c = \langle attr,\ op,\ val,\ q,\ tgt,\ ev,\ g,\ dep \rangle, \tag{12}$$

where $attr$ denotes the constrained attribute or attribute-bearing relation, op is a normalized operator drawn from a closed vocabulary (`OperatorVocab`), val is a typed value expression, q is a set of qualifiers that modulate the prescription (i.e., negation, preference, or universal scope), tgt is the attachment target, ev is the evidence span, g is the logical group assignment (AND/OR), and dep is an optional dependency pointer used for nested modifier chains. The closed vocabulary `OperatorVocab` is used to reduce surface-form variability and enable rule-based validation and downstream grounding. External references are extracted in the subsequent stage ε_R using structure and constraints as context.

Algorithm 3 summarizes the procedure. The LLM is prompted to emit only schema-conformant constraint atoms using the defined closed vocabularies. All referenced types and vocabularies are specified in the data model described in Section 4.4.1, while Table 1 summarizes the semantic role of each field together with its extraction and validation strategy.

Table 1 Constraint atom intermediate representation: fields, semantics, and validation.

Field	Type / Vocabulary	Semantics	Extraction and validation
attr.kind	`AttributeKindVocab`	Coarse type of the constrained attribute or relation mentioned in the clause.	Predicted by the LLM from the local phrase denoting the constrained aspect; validated by compatibility with *val.kind*.
attr.name	string	Surface-form name of the constrained attribute/relation (e.g., "flow rate", "material").	Copied from evidence span; validated by non-empty span within x.
op	`OperatorVocab`	Canonical operator expressing the constraint relation between *attr* and *val*.	LLM maps lexical cues (e.g., "at least", "between", "during") to an operator; validated by membership in the enum.
val.kind	`ValueKindVocab`	Type of the value expression that satisfies the operator semantics.	LLM assigns type; validated against *attr.kind* and presence of required subfields.
val.quantity	`ValueQuantity`	Numeric magnitude(s) as written, including unit string prior to QUDT normalization.	LLM extracts numbers and unit surface form; validated by parsability and presence of unit for dimensional quantities.
q	`Qualifiers`	Modifiers of deontic force or scope (e.g., "shall not", "should").	Derived by an LLM from modality, negation markers, and quantifiers. Validated by compatibility with `Qualifiers` class

4.4.4 Reference Extraction (ε_R). The third sub-task identifies external dependencies R that are required to interpret a requirement clause but are not locally defined within t_i. Such references capture information necessary to complete the semantic interpretation of the requirement.

Algorithm 3 Constraint extraction from individual requirement

Require: Individual requirement t_i, structure $\langle s, m, \kappa, a, o \rangle$, original text x
Ensure: Ordered list of constraint atoms $C = [c_1, \ldots, c_k]$
Load closed vocabularies (`OperatorVocab`, `AttributeKindVocab`, `ValueKindVocab`)
Construct prompt P_{ε_C} with t_i, s, m, κ, a, o, and vocabularies
Query LLM with P_{ε_C} with constrained schema `ConstraintsResponse`
$C \leftarrow$ Parse response into list of `Constraint` instances
for each constraint c_j **do**
  if c_j.value.kind = "quantity" **then**
    Extract numeric values v_1, v_2 (if range), tolerance
    Extract unit_text as written in source
  else if c_j.value.kind = "enum" **then**
    Extract enumeration members
  else if c_j.value.kind = "boolean" **then**
    Extract boolean value
  else if c_j.value.kind $\in$ "entity_ref", "event_ref" **then**
    Extract reference text and expected IOF class
  end if
end for
return C

Typical referenced entities in engineering requirements include standards, classifications, methods, events, components, and document artifacts (e.g., tables, figures, formulas). In addition, requirements frequently contain anaphoric references, where a clause t_i implicitly refers to an entity introduced in a previous sentence or paragraph using pronouns (e.g., “it”, “they”) or definite descriptions. These references introduce semantic dependencies that must be resolved or at least explicitly recorded to support consistent grounding and validation.

Accordingly, the reference extraction function ε_R identifies and structures such dependencies, conditioned on the structural frame and constraint atoms previously extracted. Each reference is linked to an evidence span in the original text block x and assigned an expected IOF-compatible type (see Section 4.4.1). The goal of ε_R is not to resolve all references definitively, but to explicitly surface the entities and artifacts required for a complete semantic representation of the requirement.

Algorithm 4 summarizes the procedure applied to extract references from an individual requirement clause t_i.

4.4.5 Overall Extraction Process. Algorithm 5 combines the three sub-tasks into a complete extraction pipeline that processes all individual requirements.

4.5 Normalization (ν): QUDT Quantity and Unit Mapping. The normalization stage ν enriches quantitative constraint atoms by mapping surface-form unit strings to canonical QUDT quantity kinds and units, converting numeric values to SI representations when applicable, and deriving operator-consistent numeric bounds. Formally, ν operates on the set of constraint atoms C produced by ε_C and transforms each quantitative atom into a dimensionally grounded representation suitable for ontology instantiation.

The workflow comprises five main steps:

(1) **Quantity filtering**: process only constraints with `value.kind = quantity`, a primary numeric value, and a non-empty `unit_text`;
(2) **Candidate discovery**: query G_{QUDT} for quantity kinds applicable to the extracted unit; if unavailable, use dataframe-based unit-code fallback and, when enabled, semantic search over I_{QUDT};
(3) **Disambiguation**: use LLM-based selection for the best quantity kind and best unit among candidates;
(4) **SI conversion**: convert primary/secondary values to SI using QUDT conversion multipliers (and offsets when defined);
(5) **Bound derivation**: compute lower/upper bounds and inclusion flags from the normalized operator and tolerance.

The normalization process can be formulated as a constrained search problem over the QUDT KG. The QUDT RDF graph G_{QUDT} is queried programmatically to retrieve candidate quantity kinds and associated units compatible with the extracted unit surface form.

To support semantic similarity matching when lexical alignment is insufficient, a searchable embedding index I_{QUDT} is constructed.

Algorithm 4 Reference extraction from individual requirement (ε_R)

Require: Individual requirement t_i, structure $\langle s, m, \kappa, a, o \rangle$, constraints C, original text x
Ensure: Ordered list of references $R = [r_1, \ldots, r_p]$
1: Construct prompt P_{ε_R} with t_i, $\langle s, m, \kappa, a, o \rangle$, and C
2: Query LLM with P_{ε_R} with constrained schema `References-Response`
3: $R \leftarrow$ Parse response into list of `Reference` instances
4: **return** R

Algorithm 5 Complete extraction pipeline ($\varepsilon(t_i, x)$)

Require: Indexed requirement clauses $\{(req\_idx_i, t_i)\}_{i=1}^{n}$, original text x
Ensure: Record with list of individual requirements
1: Initialize requirements list $\mathcal{R} \leftarrow \emptyset$
2: **for** each indexed individual clause (req_idx_i, t_i) **do**
3: $\langle s, m, \kappa, a, o \rangle \leftarrow \varepsilon_s(t_i, x)$ ▷ Algorithm 2
4: $C \leftarrow \varepsilon_C(t_i, \langle s, m, \kappa, a, o \rangle, x)$ ▷ Algorithm 3
5: $R \leftarrow \varepsilon_R(t_i, \langle s, m, \kappa, a, o \rangle, C, x)$ ▷ Algorithm 4
6: $r_i \leftarrow \langle req\_idx_i, s, m, \kappa, a, o, C, R, t_i \rangle$
7: Append r_i to $\mathcal{R}$
8: **end for**
9: Construct `Record` $Rec \leftarrow \langle idx,$ source metadata, $x, \mathcal{R} \rangle$
10: **return** Rec

The index is built from textual descriptions of QUDT entities (including quantity kind name, symbol, unit, URI, description, and label) and embedded using the all-MiniLM-L6-v2 model[3]. The embeddings are stored using ChromaDB[4].

Additionally, a structured dataframe D_{QUDT} stores metadata extracted from G_{QUDT} to enable deterministic filtering and compatibility checks during candidate selection.

The normalization stage maps quantitative constraints to QUDT units and quantity kinds to enable consistent dimensional representation. This is essential to capture the semantics of values, quantity kinds, and measurement units in requirement sentences t_i. A simplified overview of the normalization workflow is shown in Figure 3, where deterministic operations and LLM-mediated decisions are distinguished explicitly.

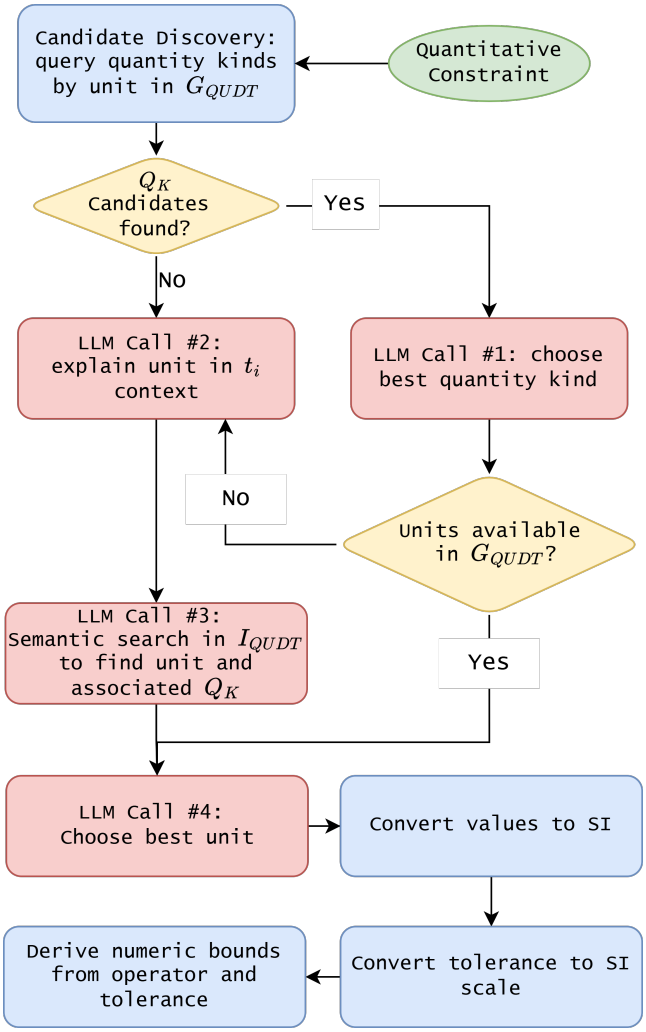

Fig. 3 Simplified workflow of QUDT normalization ν. Red blocks identify LLM calls, while blue blocks identify deterministic steps.

4.6 Grounding (γ): IOF KG Instantiation. The grounding stage maps the normalized extraction output to an OWL ABox aligned with IOF Core and a constrained subset of QUDT. Formally, given the normalized record $\hat{R} = \langle idx, x, \hat{\mathcal{R}} \rangle$ produced by ν, grounding is defined as:

$$\gamma : (\hat{R}, \mathcal{O}_{IOF}) \rightarrow G_A, \tag{13}$$

where $\mathcal{O}_{IOF}$ denotes the IOF Core TBox context used during instantiation, and G_A is the asserted graph.

At the requirement level, G_A is obtained by grounding each normalized requirement instance $\hat{r}_i \in \hat{\mathcal{R}}$ and merging the resulting assertion sets. In the evaluation pipeline, an optional reasoning step is applied to G_A to materialize additional logically entailed ABox assertions under $\mathcal{O}_{IOF}$. The resulting inferred graph is denoted by G_I, with $G_I \supseteq G_A$. If reasoning fails or produces inconsistencies, the pipeline preserves G_A as the fallback output.

4.6.1 Ontological Pattern. In the IOF Core ontology, the characterization of a requirement is grounded in the class `RequirementSpecification`, which is formally defined through two first-order axioms, expressed here in academic logic style.

$$\begin{aligned} &\textsc{RequirementSpecification}(x) \rightarrow \\ &\textsc{InformationContentEntity}(x) \wedge \\ &\exists y\colon \big(\textsc{ObjectiveSpecification}(y) \wedge \textsc{isAbout}(x, y)\big) \end{aligned} \tag{14}$$

The first axiom states that every requirement specification is an `InformationContentEntity` that must be about at least one `ObjectiveSpecification`. In IOF Core, an `ObjectiveSpecification` is defined as an information content entity that prescribes what the outcome of some process should be. This means that each requirement specification necessarily refers to one or more intended system outcomes, functioning as a normative description of the goals or results that a process should achieve. Accordingly, the grounding stage should not be read as redefining the ISO notion of a requirement, but as providing an ontological representation pattern for the artifacts derived from such requirement statements.

$$\begin{aligned} &\textsc{InformationContentEntity}(x) \wedge \\ &\exists y\colon \big(\textsc{Entity}(y) \wedge \textsc{satisfiesRequirement}(y, x)\big) \rightarrow \\ &\textsc{RequirementSpecification}(x) \end{aligned} \tag{15}$$

[3] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2#usage-sentence-transformers, Last Accessed: June 25, 2026.
[4] https://www.trychroma.com/, Last Accessed June 25, 2026.

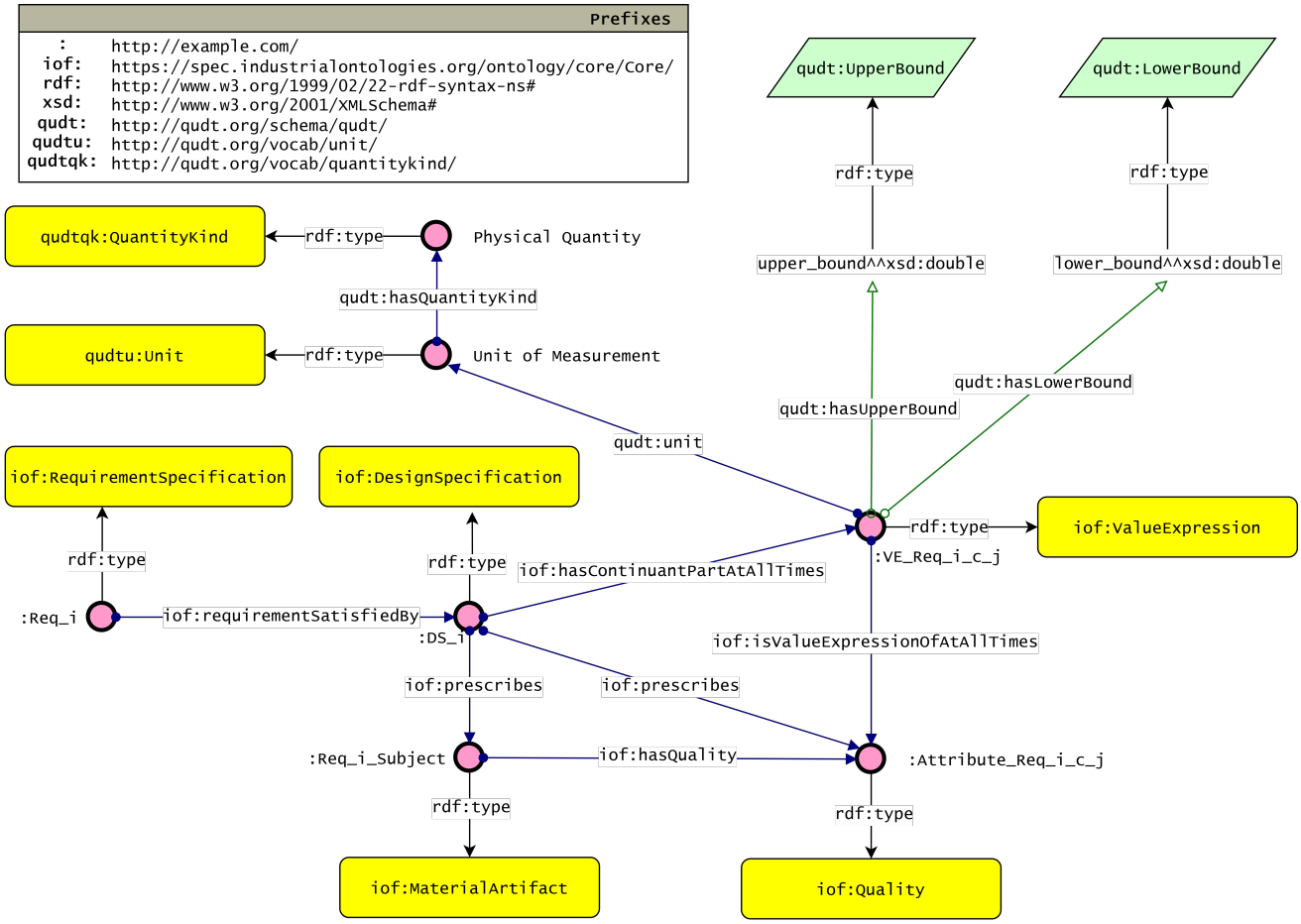

Fig. 4 Ontological grounding pattern used for instantiating requirement specifications in IOF Core.

The second axiom provides the other direction: any information content entity that is satisfied by some entity in the domain must itself be a requirement specification. Taken together, the two axioms articulate the dual nature of requirements in IOF Core: they are informational artifacts prescribing objectives and, at the same time, satisfiable conditions that systems or components may fulfill in practice.

IOF Core distinguishes requirement and design specifications. The former are about objective specifications (desired outcomes of processes) and are satisfiable by entities. The latter are information content entities that prescribe only continuants (in the BFO sense). More specifically, a design specification is an information content entity that prescribes some continuant and that satisfies some requirement specification.

This distinction marks an ontological boundary: requirements describe what outcome should be achieved (via objective specifications), whereas design specifications describe the continuant-level structure or configuration (via prescriptions over continuants).

Based on these axioms, the grounding operator γ instantiates requirement specifications according to the ontological category of the constrained bearer. In particular, requirement specifications that constrain processes or process characteristics are grounded through a plan-oriented pattern.

For each requirement instance, the grounded `RequirementSpecification` is linked via `iof:requirementSatisfiedBy` to a specification individual typed according to the ontological nature of the constrained bearer. When the constrained entity is processual (e.g., `PlannedProcess`, `Process`, `ProcessCharacteristic`, or `Event`), the associated specification is typed as `PlanSpecification`. The process-level intent is then represented through `iof:prescribes` assertions from the `PlanSpecification` to the corresponding processual individuals.

Conversely, constraints over continuants are grounded via `DesignSpecification`. To preserve this ontological separation, a deterministic post-processing safeguard verifies the prescribed targets and retypes any specification initially emitted as `DesignSpecification` to `PlanSpecification` whenever its prescribed bearer is identified as an occurrent entity.

To operationalize these ontological commitments, the grounding prompt P_γ enforces three modeling assumptions. First, a vocabulary-closure assumption: the model must reuse only classes and properties from IOF Core and the authorized QUDT subset, without introducing new schema terms. Second, a specification-typing assumption: each requirement-level prescription is instantiated through a `RequirementSpecification` and an associated specification individual typed according to the ontological category of the constrained bearer (continuant-oriented prescriptions mapped to `DesignSpecification`, occurrent/process-oriented prescriptions mapped to `PlanSpecification`). Third, a quantitative-eligibility assumption: `iof:ValueExpression` and `qudt:QuantityValue` nodes are created only when a corresponding normalized quantitative value has been produced by ν, following a one-to-one naming policy derived from requirement and constraint indices.

QUDT integration follows the IOF authors' guidelines for joint use of IOF and QUDT [50]. Quantity values are represented as `qudt:QuantityValue` individuals connected to units via `qudt:unit`, while quantity kinds are associated at the unit level through `qudt:hasQuantityKind`. Numeric magnitudes are represented using `qudt:numericValue` or appropriate bound predicates (e.g., inclusive/exclusive minimum and maximum). Non-quantitative constraints are explicitly prevented from generating

quantitative value artifacts. The base pattern (continuant-side) used as a reference for building the grounding prompt P_γ is depicted using Graffoo notation [51] in Figure 4. This pattern should likewise be understood as a task-specific grounding pattern rather than as a complete ontological model of all engineering requirements. It employs a constrained subset of IOF Core commitments, complemented by QUDT-based quantitative value modeling, in order to instantiate requirement-level ABox graphs from the extracted intermediate representation. This pattern is therefore best interpreted as an application-specific ontology-aligned instantiation template for prescriptive technical requirements, especially those containing explicit measurable constraints.

4.6.2 Reasoning and Inference. Grounding is implemented as a hybrid workflow comprising two tightly coupled phases, illustrated in Figure 5. The design deliberately separates probabilistic graph proposal from deterministic semantic enforcement.

The first phase consists of an agentic IOF instantiation loop conditioned on the full structured payload produced by ν. An LLM generates a candidate Turtle graph under strict ontology constraints: reuse IOF vocabulary only, avoid schema invention, maintain stable individual naming, and represent quantitative constraints through value-expression nodes aligned with normalization outputs. The generated graph is then iteratively repaired (up to a bounded number of iterations) using ontology-aware validation and description-logic (DL) reasoner feedback. Before each reasoning attempt, deterministic sanitization rules enforce key IOF modeling commitments. For example, misuse of `iof:hasSpecifiedOutput` on specification-like subjects is corrected, and specification individuals are retyped when prescriptions over occurrent entities are detected. This loop produces the asserted graph G_A.

If no quantitative entries are present, any accidental quantitative artifacts emitted during the LLM phase are deterministically removed to ensure consistency between extraction semantics and graph content.

After enrichment, the graph is validated against an internal IOF–QUDT pattern checker enforcing structural constraints adopted in this pipeline (e.g., quantity kinds attached at unit level, absence of forbidden bound predicates, and consistency between normalized bounds and asserted literals). When violations persist and quantitative entries exist, an LLM-guided repair step is triggered, followed again by deterministic enrichment and sanitization.

Finally, DL reasoning may be applied to materialize additional entailed assertions, producing the inferred graph G_I with $G_I \supseteq G_A$. If reasoning detects inconsistencies that cannot be resolved, the pipeline preserves G_A as fallback output.

The second phase injects quantitative semantics deterministically from the `normalized_quantities` produced by ν. For each normalized quantitative entry, the implementation:

- instantiates a canonical value node with identifier pattern `VE_req{req_idx}_c{constraint_idx}`,
- links it to the selected unit via `qudt:unit`,
- associates the unit with its quantity kind via `qudt:hasQuantityKind`,
- materializes numeric magnitudes or interval bounds using QUDT data properties.

This hybrid design localizes probabilistic behavior to graph proposal and high-level repair, while maintaining deterministic control over quantitative grounding semantics and strict adherence to the IOF+QUDT integration assumptions encoded in the prompt.

4.7 Evaluation Metrics. The pipeline is evaluated along four complementary dimensions: decomposition quality, extraction quality, normalization quality, and ontology conformance. Let $\mathcal{D} = \{x^{(1)}, \ldots, x^{(N)}\}$ be the evaluated corpus of input text blocks, and let the corresponding pipeline outputs be $\delta(x^{(k)})$, $\varepsilon(\cdot)$, $\nu(\cdot)$,

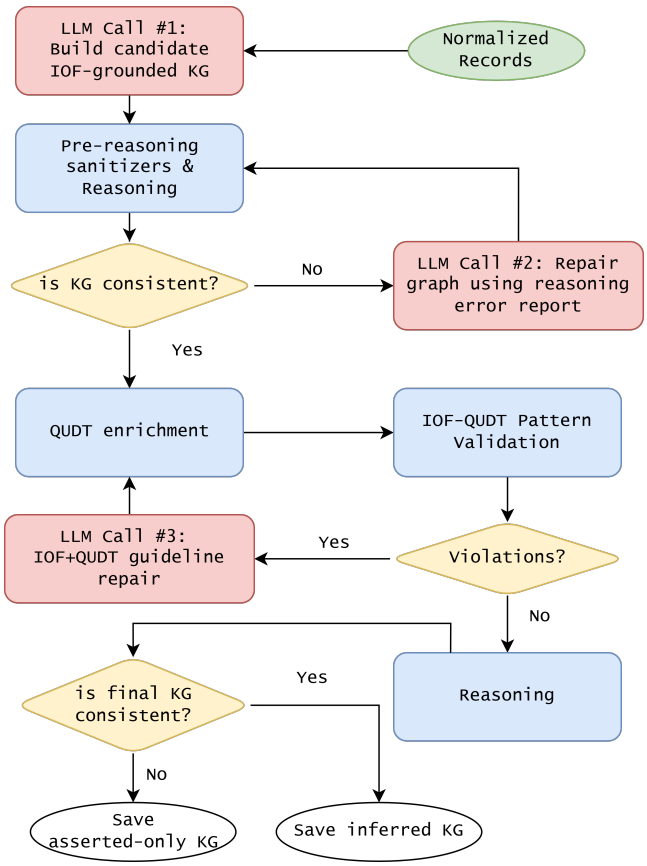

Fig. 5 Grounding workflow γ. Red blocks represent LLM calls while blue blocks identify deterministic steps.

and $\gamma(\cdot)$ as defined in the previous subsections. Manual judgments are based on semantic equivalence rather than exact lexical matching; therefore, paraphrastic variations are considered correct provided that scope and constraint semantics are preserved.

4.7.1 Decomposition metrics. For the decomposition stage, let $\hat{R}_{jud}$ denote the set of predicted individual requirement clauses produced by δ for which a manual judgment is available. Each predicted clause $\hat{r}_i \in \hat{R}_{jud}$ is assigned a binary judgment indicating whether it constitutes an acceptable individual requirement split. Decomposition accuracy is then defined as

$$\mathrm{Acc}_\delta = \frac{\mathrm{Correct}_\delta}{\mathrm{Judged}_\delta},$$

where $\mathrm{Correct}_\delta$ is the number of judged predicted clauses accepted as valid decompositions, and $\mathrm{Judged}_\delta = |\hat{R}_{jud}|$ is the total number of judged predicted clauses. This metric evaluates whether the decomposition step produces requirement units that can be meaningfully treated as standalone prescriptive clauses in the downstream stages.

4.7.2 Extraction metrics. For structural slots $\mathcal{Z} = \{s, m, \kappa, a, o\}$ (subject, modality, condition, action, object), slot-level accuracy is defined as

$$\mathrm{Acc}(z) = \frac{\mathrm{Correct}(z)}{\mathrm{Judged}(z)}, \quad z \in \mathcal{Z},$$

and macro structural accuracy as

$$\mathrm{MacroAcc}_{struct} = \frac{1}{|\mathcal{Z}|} \sum_{z \in \mathcal{Z}} \mathrm{Acc}(z).$$

Table 2 Conformance check set used in evaluation. All six checks are implemented as SPARQL violation queries and treated as error-severity checks.

ID	Family	Violation condition
C01	$\mathcal{K}_{backbone}$	Subject of `iof:requirementSatisfiedBy` is not typed as `iof:RequirementSpecification`
C02	$\mathcal{K}_{backbone}$	A `iof:RequirementSpecification` has no `iof:requirementSatisfiedBy` target
C03	$\mathcal{K}_{backbone}$	The target of `iof:requirementSatisfiedBy` is not typed as `iof:DesignSpecification` or `iof:PlanSpecification`
C04	$\mathcal{K}_{qty}$	A `qudt:QuantityValue` has no `qudt:unit`
C05	$\mathcal{K}_{qty}$	A unit used by `qudt:QuantityValue` lacks `qudt:hasQuantityKind`
C06	$\mathcal{K}_{qty}$	A `qudt:QuantityValue` is not attached via `iof:isValueExpressionOfAtSomeTime` or `iof:isValueExpressionOfAtAllTimes`

The macro formulation avoids bias toward more frequently occurring slots.

For quantitative-constraint identification, let $\widehat{C}^{jud}_{qty}$ denote the set of extracted quantitative constraints with completed manual judgments, $C^{tp}_{qty} \subseteq \widehat{C}^{jud}_{qty}$ the subset judged as true quantitative constraints, and C^{miss}_{qty} the set of manually added missing quantitative constraints. Define:

$$C^{gold}_{qty} = C^{tp}_{qty} \cup C^{miss}_{qty}.$$

Precision and recall are:

$$\mathrm{Prec}_{qty} = \frac{|C^{tp}_{qty}|}{|\widehat{C}^{jud}_{qty}|}, \qquad \mathrm{Rec}_{qty} = \frac{|C^{tp}_{qty}|}{|C^{gold}_{qty}|}.$$

In other words, $\mathrm{Prec}_{\mathrm{qty}}$ measures how often an extracted quantitative constraint is valid when the system proposes one, whereas $\mathrm{Rec}_{\mathrm{qty}}$ measures how completely the system recovers the full set of quantitative constraints present in the source. On true quantitative constraints, field-level accuracies are reported for operator, value, and unit:

$$\mathrm{Acc}_{op} = \frac{\mathrm{Correct}_{op}}{\mathrm{Judged}_{op}},$$

$$\mathrm{Acc}_{val} = \frac{\mathrm{Correct}_{val}}{\mathrm{Judged}_{val}},$$

$$\mathrm{Acc}_{unit} = \frac{\mathrm{Correct}_{unit}}{\mathrm{Judged}_{unit}}.$$

4.7.3 Normalization metrics. Normalization quality is evaluated through coverage and semantic correctness. Let $\widehat{Q}$ denote the set of quantitative outputs produced by ν. Coverage measures the proportion of extracted quantitative constraints that were successfully normalized:

$$\mathrm{Cov}_{norm} = \frac{|\widehat{Q}|}{|\widehat{C}^{jud}_{qty}|}.$$

On true quantitative constraints, semantic correctness is assessed through:

$$\mathrm{Acc}_{qk} = \frac{\mathrm{Correct}_{qk}}{\mathrm{Judged}_{qk}},$$

$$\mathrm{Acc}^{norm}_{unit} = \frac{\mathrm{Correct}^{norm}_{unit}}{\mathrm{Judged}^{norm}_{unit}},$$

$$\mathrm{Acc}_{eq} = \frac{\mathrm{Correct}_{eq}}{\mathrm{Judged}_{eq}}.$$

where Acc_{eq} measures whether normalized quantity kind, unit, and value have been correctly represented according to the SI measurement system and QUDT ontology.

4.7.4 Conformance metrics. A series of 6 conformance metrics were implemented as SPARQL queries to provide additional guardrails to the KG generated by the LLM. Let $\mathcal{K}$ be the set of ontology conformance checks, and $v_k(G)$ the number of violations of check $k \in \mathcal{K}$ on graph G. Conceptually, $\mathcal{K}$ is organized into two main families:

$$\mathcal{K} = \mathcal{K}_{backbone} \cup \mathcal{K}_{qty},$$

where $\mathcal{K}_{backbone}$ verifies the minimum requirement-specification backbone (requirement typing, existence of satisfaction links, and valid typing of linked specifications), and $\mathcal{K}_{qty}$ verifies quantitative/value modeling well-formedness (presence of units for quantity values, availability of quantity-kind links at unit level, and correct attachment of value expressions to constrained bearers). In the present evaluation, $\mathcal{K}$ is intentionally kept compact to focus on final-KG structural soundness: three backbone checks and three quantitative checks. This reduced set targets high-impact structural failures that directly affect interpretability and queryability of the generated KG. The conformance checks used to evaluate the graphs generated by the pipeline are reported in Table 2 and implemented through SPARQL queries (see Appendix C)

Over the evaluated graph set $\mathcal{G}_{eval}$, we report:

$$\mathrm{PassAll} = \frac{|\{G \in \mathcal{G}_{eval} : \sum_{k\in\mathcal{K}} v_k(G) = 0\}|}{|\mathcal{G}_{eval}|},$$

$$\mathrm{PassErr} = \frac{|\{G \in \mathcal{G}_{eval} : \sum_{k\in\mathcal{K}_{err}} v_k(G) = 0\}|}{|\mathcal{G}_{eval}|},$$

where $\mathcal{K}_{err} \subseteq \mathcal{K}$ denotes error-severity checks. In addition, total violations $\sum_{G\in\mathcal{G}_{eval}} \sum_{k\in\mathcal{K}} v_k(G)$ and per-check prevalence are reported.

Per-check prevalence is defined as:

$$\mathrm{Prev}(k) = \frac{|\{G \in \mathcal{G}_{eval} : v_k(G) > 0\}|}{|\mathcal{G}_{eval}|}, \qquad k \in \mathcal{K}.$$

These metrics provide two complementary views of conformance: global graph-level pass rates and localized error distributions associated with specific ontology constraints.

5 Results

This section reports the empirical evaluation of the proposed requirement-to-KG mapping function on two datasets, with metrics defined in Section 4.7. The first dataset is an evaluation set composed of 120 input text blocks x extracted from a Formula SAE rulebook [36]; it is used for the main quantitative assessment of extraction, normalization, grounding, and annotation reliability. The second dataset is a small comparative ablation set comprising 20 requirements (r_i) selected from the CubeSat documentation [52] and two robotics domain sources [53,54]; it is used to examine the contribution of the pre-processing stages (ε and ν) and of the agentic graph-repairing loop relative to zero-shot grounding (γ).

5.1 Datasets and Evaluation Protocol. The main quantitative evaluation dataset was constructed from the Formula SAE (FSAE) rules corpus used in [36]. The source engineering document was first converted from PDF into text through Mistral OCR[5], producing page-level transcriptions that were subsequently segmented into sentence-like textual spans. Each span was stored together with provenance metadata. Only minimal cleanup was applied, so that the benchmark would retain document-conversion artifacts that are likely to occur in practical engineering-document workflows rather than relying on a manually normalized transcription. This preprocessing stage produced 988 extracted spans. Then, a regular expression filter was applied to retain only rows containing explicit number-unit expressions, including both standard forms (e.g., "50 mm") and LaTeX-like forms (e.g., `\mathrm{m/s}`). This yielded 189 eligible rows.

This filtering was adopted because quantitative constraints are the most demanding case for the proposed normalization and IOF+QUDT grounding workflow; therefore, the FSAE benchmark was designed as a stress test of measurable requirement formalization rather than as a coverage benchmark for all requirement types.

Each row was then assigned to a stratum defined by unit family (mechanical, electrical, percentages, etc.) and text-length bucket to balance quantitative domains and syntactic complexity. Using a fixed random seed, a development subset ($n = 20$) and a disjoint test subset ($n = 120$) were sampled to ensure that each available unit family was represented at least once in each split whenever possible. The development subset was used only for prompt refinement, schema checking, and deterministic rules tuning. In practice, this refinement phase focused on reducing decomposition errors in list-structured spans, improving schema-conformant extraction outputs, and eliminating recurrent grounding/post-processing failures observed during pilot runs. On the other hand, all reported quantitative and structural results on the FSAE benchmark are computed on the held-out test subset. To complement the main FSAE evaluation, a second dataset of 20 requirements was constructed from the CubeSat Design Specification [52] and two requirements documents produced as public deliverables in EU-funded robotics projects [53,54]. This second dataset was intentionally small and was not used as a second benchmark of comparable statistical scope. Instead, it was introduced as a comparative ablation set to probe two methodological questions: (i) whether pre-processing steps (decomposition, extraction, and normalization) improve the semantic quality of the grounded representation relative to grounding from raw requirement text, and (ii) whether the KG repair loop of the grounding procedure described in Section 4.6 provides benefits over a zero-shot LLM grounding configuration. The selected requirements were chosen to cover varied constraint patterns across different domains, including domain-specific quantity kinds and units. All the tests executed in the context of this study were done using GPT-5.1 (`gpt-5_1-2025-11-13`) as LLM.

5.2 Annotation Procedure and Inter-Annotator Agreement. The evaluation of the decomposition, extraction, and normalization steps on the FSAE test set was based on manual item-level judgments over pipeline outputs, which were subsequently aggregated into the metrics defined in Section 4.7. The pipeline was first executed to produce candidate individual requirements r_i, structured requirement fields $\mathcal{Z} = \{s, m, \kappa, a, o\}$, and normalized quantitative constraints C. For the decomposition step (δ), each predicted individual requirement r_i was assigned a binary label `decomposition_error`. A value of `false` indicates that the split produced an acceptable individual requirement, whereas `true` indicates an incorrect split, for example because the predicted item was incomplete or because list items required to preserve the requirement meaning had been detached incorrectly. For the extraction step (ε), each decomposed requirement was evaluated independently at the level of the fields subject s, modality m, condition κ, action a, and object o. Each field was judged using a binary correctness label based on semantic equivalence rather than exact string matching. A prediction was marked correct whenever it preserved the intended meaning and scope of the source requirement, even if the wording differed slightly. Incomplete requirements r_i which had fields not explicitly expressed in the input text were treated as decomposition errors and excluded from the denominator of the corresponding slot-level accuracy. For the normalization step(ν), only predicted quantitative constraints were considered to assess the performance of the workflow. Each predicted quantity was first judged for whether it corresponded to a true measurable constraint in the source text (`is_true_quantitative_constraint`). For all the quantitative constraints extracted, the evaluation concerned the correctness of the extracted operator, value, and unit, together with the correctness of the normalized quantity kind and the numerical and SI-unit equivalence of the normalization result. To assess annotation consistency, inter-annotator agreement was computed across five annotation sets: the main annotation set, curated by the authors, and four additional annotators who independently labeled partially overlapping subsets of the FSAE evaluation set. The secondary annotators were two Ph.D.-level engineers and two Ph.D. students with experience in engineering design. All annotators were provided with annotation guidelines defining acceptable decomposition boundaries, semantic-equivalence criteria for slot judgments, and the distinction between identification and normalization errors for quantitative constraints. Each secondary annotator reviewed 30 requirements, but the effective denominator varies across labels because agreement was computed only on non-missing judgments for the specific item under consideration. In particular, quantity-specific sublabels were available only for predicted quantitative constraints, and some follow-up judgments were not applicable when a candidate was not judged as truly quantitative. Because the overlap design was incomplete and not all items were labeled by the same number of annotators, we report observed pairwise agreement as a measure of annotation consistency. Aggregate rows in Table 3 are reported as macro averages across labels within each group rather than as pooled reliability coefficients. Inter-annotator agreement was computed on the original independent annotations, prior to any adjudication. Disagreements were resolved by the authors during adjudication to produce the final reference labels used for performance computation.

The annotation protocol has some limitations. The final reference labels were adjudicated by the authors, and the secondary annotations covered only partially overlapping subsets of the evaluation set. Consequently, the reported agreement values should be

Table 3 Inter-annotator agreement for the manual evaluation labels.

Label	Ratings	Agreement
Requirement-level labels		
Aggregate	1746	80.4%
`decomposition_error`	371	67.2%
`subject_correct`	275	79.4%
`modality_correct`	275	97.1%
`condition_correct`	275	76.5%
`action_correct`	275	85.3%
`object_correct`	275	77.2%
Quantity-constraint labels		
Aggregate	1378	94.6%
`is_true_quantitative_constraint`	233	97.2%
`operator_correct`	229	94.2%
`quantity_value_correct`	229	98.6%
`unit_correct`	229	98.6%
`quantity_kind_correct`	229	91.4%
`equivalence_correct`	229	87.8%

[5]https://mistral.ai/it/news/mistral-ocr, Last Accessed: June 25, 2026

Table 4 Decomposition (δ) and Structure Extraction (ε_s) results on the held-out FSAE test set.

Metric	Correct	Judged	Value
Acc_δ	160	251	63.74%
$\text{Acc}(s)$	158	160	98.75%
$\text{Acc}(m)$	160	160	100.00%
$\text{Acc}(\kappa)$	156	160	97.50%
$\text{Acc}(a)$	140	160	87.50%
$\text{Acc}(o)$	142	160	88.75%
$\text{MacroAcc}_{\text{struct}}$	–	–	94.50%
All structure slots correct (row level)	132	160	82.50%
All structure and quantity slots correct (row level)	111	160	69.38%

Table 5 Quantitative Constraint Extraction (ε_C) and normalization (ν) results on the held-out FSAE test set.

Metric	Correct	Judged	Value
Prec_{qty}	124	127	97.64%
Rec_{qty}	124	128	96.88%
Acc_{op}	120	124	96.77%
Acc_{val}	122	124	98.39%
Acc_{unit}	114	124	91.94%
Cov_{norm}	162	164	98.78%
Acc_{qk}	110	124	88.71%
Acc_{eq}	111	124	89.52%

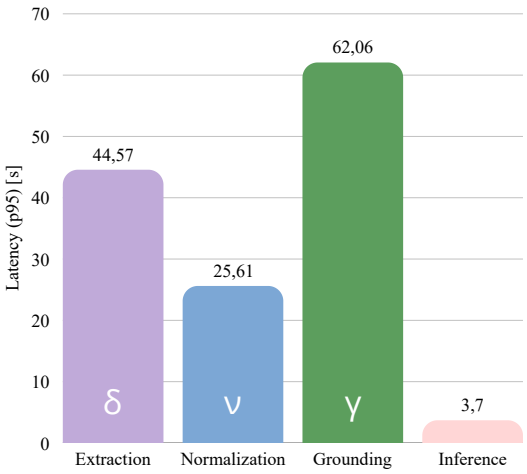

Fig. 6 Pipeline stage p95 latency on the FSAE test set.

interpreted as evidence of annotation consistency, not as a full reliability study over the complete dataset. In addition, binary semantic-equivalence judgments inevitably simplify borderline cases, especially for underspecified objects, implicit subjects, scoped conditions, and quantity-kind disambiguation. These limitations should be considered when interpreting the reported extraction and normalization accuracies.

5.3 Performance and Reliability of the Requirement-to-KG Pipeline. Table 4 summarizes decomposition (δ) and Structure Extraction (ε_s) performance. Structure-level metrics were computed only on correctly decomposed rows. Fields not explicitly present in the source text (such as missing subjects or objects) were excluded from the denominator of the corresponding slot-level accuracy. The most common decomposition errors occurred in list-structured passages, where introductory clauses were treated as standalone requirements or bullet items were split into incomplete fragments. Additional failures arose when headings, section identifiers, or cross-references bled into the extracted span. Table 5 summarizes quantitative constraint extraction and normalization performance. Precision (Prec_{qty}) and recall (Rec_{qty}) evaluate identification of true quantitative constraints, whereas operator, value, unit, quantity kind, and equivalence accuracies are reported only for extracted constraints judged to be truly quantitative. Normalization coverage (Cov_{norm}) uses all extracted quantities as the denominator.

For what concerns graph-level results for the grounding stage (γ), it is relevant to assess whether the structured outputs produced by the upstream stages can be converted into IOF-grounded KGs that remain structurally valid after reasoning and post-processing. In the experimental run on the FSAE dataset described in Section 5.1, grounding succeeded for all 120 input records, yielding an end-to-end graph-generation success rate of 100.00% with respect to the evaluation inputs. Inference was attempted on all grounded graphs. Description-logic reasoning completed successfully for 109/120 graphs (90.83%), while 11/120 graphs (9.17%) failed during inference. Table 6 reports the per-check violation profile on the conformance checks described in Section 4.7. Violations were concentrated in two checks only. C02 (missing `iof:requirementSatisfiedBy` linkage) accounted for 3 violations across 3 graphs (2.50%), while C03 (invalid typing of the satisfaction target as design/plan specification) accounted for 7 violations across 5 graphs (4.17%). No violations were observed for C01, C04, C05, or C06.

For each pipeline stage, all valid non-negative latencies observed across the full run were collected, and the 95th percentile $p95$ was then computed to characterize the completion time of an individual processing step. The stage-level $p95$ latency profile identifies grounding as the primary computational bottleneck at 62.06 s, followed by extraction at 44.57 s, normalization at 25.61 s, and inference at 3.70 s (Figure 6). This distribution is consistent with the ontology-constrained grounding and repair loop described in Section 4.2, which involves iterative LLM generation, logical reasoning, and deterministic post-processing. Extraction and normalization exhibit moderate latency due to multiple structured LLM calls and semantic search over QUDT resources. In contrast, the final description-logic inference contributes comparatively little to overall runtime, indicating that the principal computational cost arises from probabilistic graph instantiation rather than symbolic reasoning. These measurements reflect the current prototype implementation and were obtained without batching or parallelization. They therefore represent an upper-bound estimate of per-record latency under sequential execution.

Figure 7 reports a representative successful grounding example from the FSAE test set. The example illustrates the type of artifact produced by the pipeline when decomposition, structured extraction, quantitative normalization, and ontology-aligned grounding succeed jointly. This example is intended to illustrate the structure and information content of a successful output artifact, complementing the aggregate metrics reported in Tables 4, 5 and 6.

5.4 Comparative Ablation Across Pipeline Configurations. To complement the evaluation on the FSAE test set, a small com-

Table 6 Ontology-conformance results on the final inferred graph set.

Check ($k \in \mathcal{K}$)	**Meaning**	**Violations** ($v_k(G)$)	**Graphs affected**	Prev(k)
C01	Requirement typed as `iof:RequirementSpecification`	0	0	0.00%
C02	Requirement has `iof:requirementSatisfiedBy` link	3	3	2.50%
C03	Satisfaction target typed as Design/Plan specification	7	5	4.17%
C04	`qudt:QuantityValue` has `qudt:unit`	0	0	0.00%
C05	Unit declares `qudt:hasQuantityKind`	0	0	0.00%
C06	QuantityValue linked via `isValueExpressionOf`	0	0	0.00%
All checks	Graph-level pass rate (PassAll)	–	112/120	93.33%
Error checks	Graph-level error-pass rate (PassErr)	–	112/120	93.33%

parative ablation study was conducted on a cross-domain set of carefully selected requirements ($N = 20$). The aim of this experiment is not to establish a second benchmark of comparable statistical scope, but to probe the effect of two methodological choices under domain shift: (i) structured preprocessing through decomposition, extraction, and normalization prior to grounding; and (ii) the use of a DL-reasoning-enabled repair loop during grounding, hereafter referred to as the "agentic loop". Accordingly, four pipeline configurations were evaluated by combining the presence or absence of these two components:

(1) Structured Pre-processing with Agentic Grounding;
(2) Structured Pre-processing with Zero-shot Grounding;
(3) Raw Requirement Text with Agentic Grounding;
(4) Raw Requirement Text with Zero-shot Grounding;

Table 7 reports the results in terms of correctly inferred graphs and throughput (records/min). In this context, a correctly inferred graph denotes a graph for which grounding and reasoning completed successfully without inconsistency and produced a final graph artifact eligible for downstream structural evaluation. Across the 20 cross-domain requirements, configurations including the agentic loop produced a higher number of correctly inferred graphs than their zero-shot counterparts. This pattern was observed both when structured preprocessing was enabled and when grounding was performed directly from raw requirement text. The result suggests that iterative ontology-aware repair contributes additional graph-level robustness beyond single-pass generation alone. A second contrast concerns computational cost. The complete pipeline, combining structured preprocessing with the agentic loop, was the slowest configuration at 1.48 records/min, followed by the structured zero-shot variant at 1.82 records/min. The raw-input variants were faster overall, with raw agentic grounding reaching 2.60 records/min and raw zero-shot grounding achieving the highest throughput at 6.51 records/min. These measurements indicate a clear trade-off between structural robustness and execution speed in the tested setting. The ablation also highlighted differences that are not fully captured by the graph-level success counts alone. Although raw-input grounding could still produce logically admissible graphs, qualitative inspection showed that the absence of structured pre-processing more frequently led to broader and less targeted graph constructions, particularly when requirements contained explicit quantitative constraints or multiple semantically distinct elements. Figure 8 provides a representative semantic comparison between structured and raw-input grounding outputs. Taken together, these results do not support broad statistical claims about cross-domain generalization; however, they do provide initial evidence that the two main methodological choices play different roles in the pipeline.

Table 7 Ablation study on the 20-requirement cross-domain subset.

		Agentic loop	
		Absent	**Present**
Pre-processing	**Absent**	11/20 (55%) 6.51 rec/min	19/20 (95%) 2.60 rec/min
	Present	12/20 (60%) 1.82 rec/min	19/20 (95%) 1.48 rec/min

6 Discussion

6.1 General Considerations. The quantitative results indicate that the pipeline reliably extracts prescriptive requirement structure and preserves quantitative logic through normalization. Slot-level extraction accuracy exceeds 85% across all ISO-aligned fields (Table 4), while quantitative-constraint detection achieves both high precision and recall (Table 5). This suggests that most explicit technical constraints expressed in the FSAE requirements are captured consistently.

At graph level, 120 final artifacts were produced from 120 grounded records, with reasoner-complete inference achieved for 109/120 cases and asserted-only fallback artifacts for the remaining inconsistent graphs. Conformance checks show that 112/120 graphs pass all structural checks (93.33%; Table 6). Residual violations are limited to requirement-to-specification linkage (C02) and specification typing (C03), while no violations were observed for requirement typing (C01), quantity-unit attachment (C04), unit-to-quantity-kind linkage (C05), or value-expression attachment (C06). This distribution indicates that the remaining issues are concentrated in the requirement-specification grounding pattern, rather than in the quantitative-value modeling layer or in systematic misuse of the IOF backbone. This consideration closely relates to the inherent probabilistic nature arising from generating knowledge graphs using LLMs. Positioning these results against prior work requires caution: direct quantitative comparison is difficult because existing LLM-based requirements-engineering evaluations use different task formulations, output schemas, and metrics. For example, Lim et al. [11] evaluate sequence-labeling performance over requirement-related token classes, while Holter and Ell [32] assess decomposition into scope, condition, and demand using text-similarity metrics. In contrast, the present work evaluates a staged formalization pipeline that includes structured extraction, quantitative normalization, and ontology-grounded graph instantiation. For this reason, the cited studies are relevant as related task-level references, but they are not directly comparable baselines for the end-to-end ontology-constrained formalization task considered here.

From a mechanical design engineering perspective, these results are encouraging for requirements formalization and traceability. Prescriptive intent is largely preserved, quantitative constraints are mostly propagated to the grounded graph, and final artifacts are frequently structurally valid for downstream querying and consistency analysis. The remaining risks are concentrated in semantically critical grounding decisions, particularly the correct typing of the specification that satisfies the requirement and the preservation of requirement-specification linkage, which directly affect engineering interpretation and reasoning reliability.

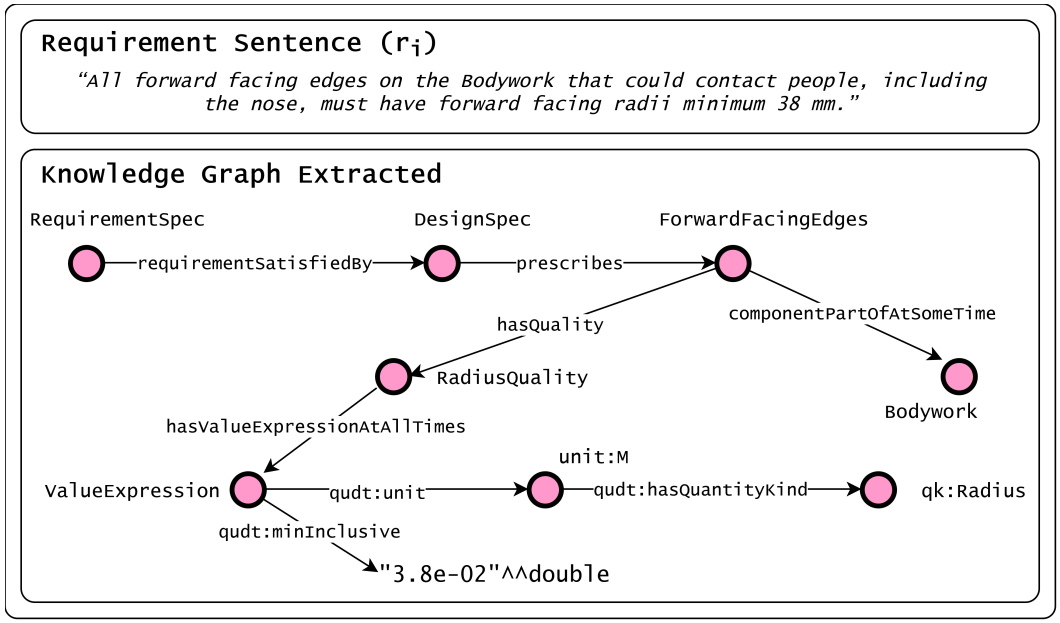

Fig. 7 Representative successful grounding example: a requirement sentence from the FSAE dataset is transformed into an IOF/QUDT-grounded graph fragment.

The ablation results show that configurations including the agentic loop yielded a higher number of successfully inferred graphs than the corresponding zero-shot variants, indicating a positive contribution of iterative ontology-aware repair to graph-level structural robustness. Differences associated with structured preprocessing are less pronounced in the aggregate success counts, but are visible in the organization of the generated outputs. As shown in Figure 8, the graph produced from pre-processed input follows the extracted requirement structure more closely and represents the quantitative prescription through a more targeted grounding pattern, whereas the raw-input variant produces a broader graph that is still interpretable but less tightly aligned with the clause-level semantic decomposition. This qualitative contrast complements the aggregate ablation results by showing that the two factors affect different aspects of performance: the agentic loop primarily influences successful inference, while pre-processing more strongly affects representational focus and semantic organization. From a practical perspective, the grounded graph representation can support downstream operations that are difficult to implement reliably over unstructured requirement text alone. Examples include querying all requirements associated with a given subsystem, constrained bearer, or quality; filtering requirements by quantity kind or unit; checking whether quantitative constraints have been normalized consistently; and tracing requirement–specification relations for validation or reasoning-oriented analysis. Unlike structured documentation templates, the present representation does not only improve organization or readability, but also makes entity typing, semantic relations, and value structures explicit in a machine-interpretable form (e.g. via SPARQL queries).

6.2 Limitations and Implications for Deployment. Despite the encouraging conformance and extraction results, the evaluation also highlights important limitations that constrain how the generated KGs should be interpreted and deployed in practice. In particular, the residual failures are not uniformly distributed across all stages of the pipeline, but concentrate in specific classes of semantically demanding requirements. The most recurrent failure patterns are associated with: (i) higher-order normative logic, such as alternative compliance pathways and nested conditional structures; (ii) ontological characterization errors in the requirement-specification grounding pattern; and (iii) domain-sensitive semantic disambiguation when the intended engineering meaning is only partially recoverable from local lexical cues. The pipeline is effective at identifying local entities, quantitative constraints, and participation relations, but it remains less robust when the source statement encodes alternatives, internal conjunctions, or scoped dependencies between admissible compliance branches. Figure 9 illustrates a representative example. In the source requirement, spherical rod ends and spherical bearings must satisfy one of two admissible mounting conditions. The extracted graph preserves the main requirement backbone, the relevant processual context, and the associated participants, but it does not encode the mutually exclusive *one-of* relation between the two branches. Instead, the two admissible configurations are flattened into parallel constraint-linked structures attached to the same mounting process. As a result, the output remains informative for traceability and entity-level inspection, yet it weakens the normative semantics of the original requirement by failing to represent that the branches are alternatives rather than jointly cumulative conditions. The limitations highlighted above have direct implications for deployment in engineering practice, thus the present results do not yet justify fully autonomous use in high-stakes settings where compliance depends on the exact preservation of alternative pathways, conditional validity, or ontologically precise grounding. Safety-critical review, certification checking, and downstream automated reasoning over complex compliance logic would still require targeted human oversight.

6.3 Methodological Limitations. The proposed pipeline is designed for industrial-style English requirements characterized by explicit deontic modalities and measurable constraints. Its design choices, namely structured extraction, ontology-closed grounding, and QUDT-aligned normalization, implicitly define the operational scope within which performance can be expected to remain stable.

Several limitations follow from these assumptions. First, domain shift may occur when processing requirements from domains substantially different from those represented in the prompting examples or evaluation corpus. Second, cross-sentence coreference resolution remains limited; while reference extraction surfaces dependencies, full discourse-level resolution is not guaranteed. Third, grounding may remain incomplete when referenced entities are opaque, underspecified, or ambiguous, leading to null or partially typed resolution targets. Fourth, LLM variability may introduce minor extraction differences across runs, although schema vali-

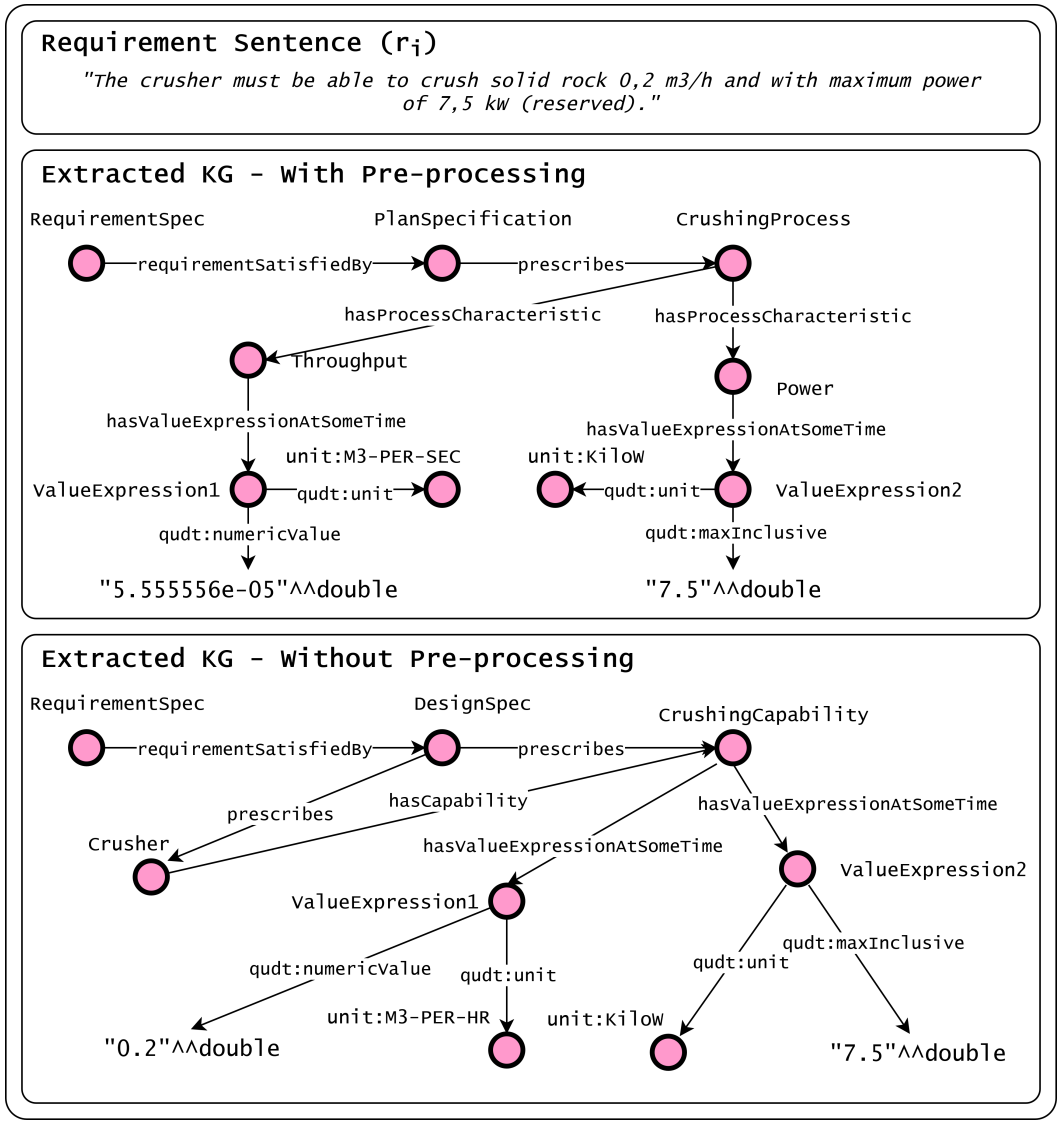

Fig. 8 Representative comparison of grounded KGs for the same requirement under two pipeline configurations. Top: output generated with structured preprocessing (decomposition, extraction, and normalization) followed by grounding. Bottom: output generated by grounding the raw requirement text without preprocessing. The pre-processed configuration yields a graph centered on an explicit `CrushingProcess` with separate value expressions for throughput and power, whereas the raw-input configuration produces a more compact graph centered on a generic `CrushingCapability`.

dation and deterministic post-processing mitigate structural divergence. A further limitation concerns the dependence of the pipeline on the selected ontology and controlled vocabularies. This dependence is a strength insofar as it constrains admissible entity and relation types, reduces schema invention, and enables compatibility with formal validation and reasoning. At the same time, transfer to a new application domain is not cost-free. Adaptation may be required at three levels: (i) revision of prompts and examples to better reflect target-domain phrasing, (ii) extension or tailoring of controlled vocabularies and quantitative normalization mappings, and (iii) verification that the relevant domain concepts can be represented adequately under the selected ontology commitments. This effort is lower when the target domain remains close to the industrial engineering scope already covered by IOF and QUDT, and higher when domain-specific semantics or specialized quantitative concepts become central. An additional limitation concerns document preprocessing. The current pipeline assumes that requirement statements can be recovered as text from source PDFs with sufficiently preserved reading order and local structure. In practice, this assumption may fail for scanned documents, complex layouts, nested lists, tables, page breaks, equations, or visually encoded references. Under such conditions, OCR and layout linearization may merge adjacent rules, split a single requirement across multiple spans, or corrupt quantitative surface forms before semantic decomposition begins. These failures are external to the ontology-grounding logic itself, but they can still affect end-to-end performance by distorting clause boundaries, suppressing contextual references, or introducing malformed quantity expressions. The reported results should therefore be interpreted as evaluating the requirement-formalization pipeline under realistic PDF-derived input conditions rather than as isolating ontology-grounding performance from upstream document-conversion effects. Moreover, the use of closed vocabularies in the intermediate representation represents another limitation of the presented approach. While these vocabularies reduce lexical variability and simplify validation and grounding, they also restrict expressive flexibility: uncommon domain phrasing, borderline semantic cases, or requirement formulations that do not align well with the predefined categories may be forced into approximate labels or left only partially represented.

Human oversight remains essential in high-stakes deployments,

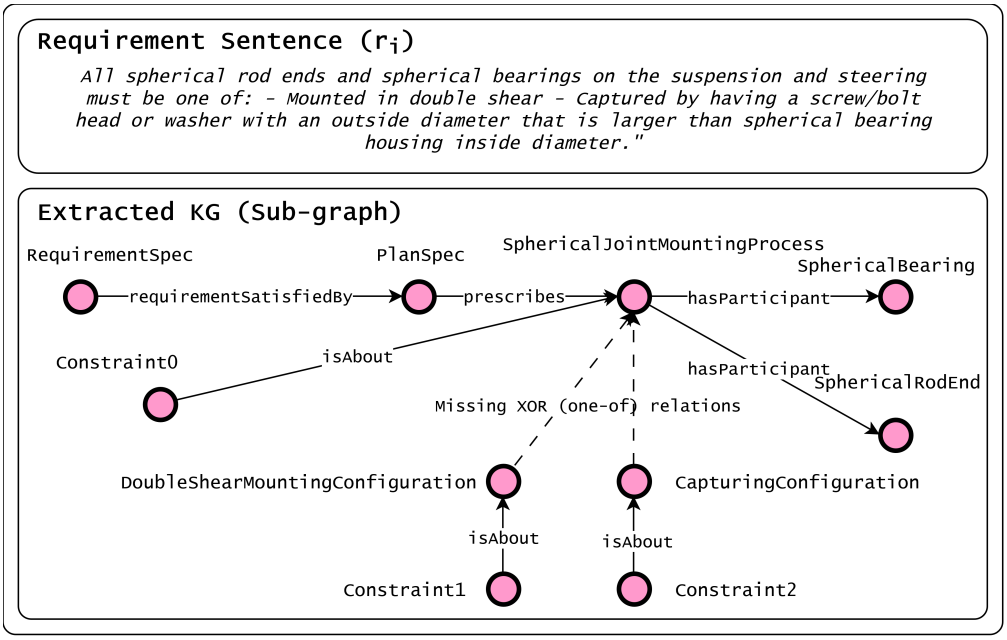

Fig. 9 Representative failure case involving higher-order normative logic. The requirement specifies that spherical rod ends and spherical bearings must satisfy one of two alternative mounting conditions. The extracted sub-graph preserves the main requirement backbone and identifies the relevant process and participating entities, but the mutually exclusive one-of relation between the two admissible compliance branches is not explicitly represented.

particularly when resolving ambiguous references or validating ontology mappings beyond coarse typing. The pipeline preserves traceability through explicit evidence spans and typed intermediate representations, thereby enabling systematic human review of uncertain extractions.

7 Conclusion

This work presented a staged neuro-symbolic pipeline for transforming natural-language technical requirements into IOF-grounded knowledge graphs through decomposition, structured extraction, quantitative normalization, and ontology-constrained grounding. The approach combines probabilistic language modeling with typed intermediate representations, deterministic post-processing, and description-logic reasoning to preserve semantic intent while enforcing ontological commitments.

The empirical evaluation shows that, in the evaluated setting, natural-language engineering requirements can be transformed into ontology-grounded graph representations with good structural reliability. The results indicate that the pipeline can preserve requirement structure and explicit quantitative semantics sufficiently well to support reviewable validation-, traceability-, and reasoning-oriented downstream analysis in the evaluated setting, although decomposition errors and strict row-level failures limit fully autonomous use. The comparative ablation provides preliminary evidence that the agentic grounding loop can improve graph-level robustness in the tested cases, but the small cross-domain sample should be interpreted as an exploratory stress test rather than as evidence of broad generalization. At the same time, the study also shows that robust ontology-grounded semantic formalization remains more difficult than local extraction alone. The most important remaining failures arise in semantically critical grounding decisions, especially when requirements involve higher-order logical structure, alternative compliance pathways, or domain-sensitive grounding ambiguities. These limitations make clear that the current method should not yet be interpreted as a fully autonomous engineering decision-support capability.

A further practical boundary concerns the dependence of end-to-end performance on upstream requirement extraction from real documents. The present pipeline operates on text spans derived from document conversion and segmentation steps, and errors introduced during OCR, layout linearization, span selection, or requirement extraction can propagate downstream into decomposition, normalization, and grounding. In this sense, the reported results should be interpreted not only as a measure of the formalization workflow itself, but also as evidence of how strongly real-document preprocessing affects ontology-grounded requirement modeling in practice. In practical terms, the present contribution is best understood as a formalization layer for requirements traceability, consistency checking, and knowledge integration, rather than as a complete end-to-end design-support system. The value of the method lies in establishing a semantically explicit bridge between textual requirements artifacts and ontology-grounded representations that can be queried, validated, and connected to broader engineering knowledge infrastructures.

Future work should therefore proceed in four directions. First, the extraction of requirements from real engineering documents should be strengthened, particularly in the presence of complex layouts, lists, tables, equations, and cross-references. Second, explicit modeling of higher-order normative logic (AND/OR/XOR structures, scoped conditions, and compliance pathways) should be incorporated into the intermediate representation to prevent semantic flattening during grounding. Second, substantially broader cross-domain evaluation is necessary before making claims about domain-general robustness. to assess robustness beyond automotive competition rules and the scope of the small cross-domain ablation study. Third, more comprehensive coverage-based evaluation methods are needed to quantify semantic completeness at scale, complementing high-precision sampled validation.

Author Contributions

Alessandro Stefanone: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - original draft, Writing - review & editing. **Marco Rossoni:** Conceptualization, Supervision, Methodology, Validation, Writing - original draft, Writing - review & editing. **Giorgio Colombo:** Conceptualization, Supervision, Resources, Writing - review & editing.

Data Availability Statement

The data and code that support the findings of this study are openly available on GitHub at https://github.com/alessandrostefanone-polimi/ontology-req-pipeline(ontology-req-pipeline).

Appendix A: Controlled Vocabularies for Structured Extraction

This appendix reports the controlled vocabularies used by the extraction and normalization stages to constrain model outputs and reduce lexical variability. Each vocabulary defines an enumerated set of admissible symbols for a specific field in the intermediate representation (IR), supporting deterministic post-processing and consistent grounding into the IOF/QUDT-aligned ABox (Tables 8-13).

Table 8 OperatorVocab: normalized operators for constraint expressions.

Operator	Meaning
`eq`	Equal.
`neq`	Not equal.
`lt`	Lower than.
`le`	Lower than or equal to.
`gt`	Greater than.
`ge`	Greater than or equal to.
`between`	Range constraint (typically lower and upper bounds).
`one_of`	Exactly one element in a list of candidates is admissible.
`all_of`	All elements in a list of candidates must hold simultaneously.
`has_feature`	Presence/availability of a non-numeric feature.
`type_is`	Typing constraint prescribing that an entity is of a specific type.
`uses_method`	Constrains *how* an action/process is carried out (method constraint).
`before`	Temporal ordering: applies before a referenced event/time.
`after`	Temporal ordering: applies after a referenced event/time.
`during`	Temporal scoping: applies during a referenced event/state.
`until`	Temporal bound: applies up to a referenced event/time.

Table 9 ModalityVocab: normalized deontic markers for prescriptive clauses.

Token	Meaning
`shall`	Obligation (typical normative requirement form).
`must`	Obligation (often equivalent in strength to `shall` in specifications).
`should`	Recommendation (non-mandatory, preferred).
`may`	Permission / optionality.
`will`	Statement of intent or future behavior (may be descriptive depending on context).
`is`	Copular form used in some specifications (often descriptive unless used prescriptively by context).

Table 10 ValueKindVocab: normalized types of values used in constraint atoms.

Value kind	Meaning / grounding implication
`quantity`	Quantitative value (numeric value + unit; may include tolerance/range); grounded using QUDT value patterns.
`enum`	Closed set of admissible values (list/set membership).
`boolean`	Truth-valued constraint (`true`/`false`).
`entity_ref`	Reference to an entity not explicitly defined/typed in the clause; grounding uses an expected type (see ValueRefVocab).
`event_ref`	Reference to an event/state not explicitly defined/typed in the clause; grounding uses an expected type (see ValueRefVocab).
`none`	No value specified; used to avoid hallucinated values and to support partial constraints.

Table 11 AttributeKindVocab: normalized kinds of constrained attributes.

Attribute kind	Meaning / grounding implication
`quantity`	Constrained attribute is a measurable quantity/quantity kind (e.g., pressure, mass flow rate).
`enum`	Constrained attribute is categorical with an admissible set of values.
`boolean`	Constrained attribute is truth-valued (e.g., enabled/disabled, present/absent).
`event`	Constrained attribute is an event/state used for scoping or triggering.
`relation`	Constrained attribute is a non-quantitative relation/predicate between entities.

Table 12 ValueRefVocab: expected ontological type for referenced values.

Expected type	Interpretation
`MaterialEntity`	Physical object or system element.
`Process`	Event, activity, operation, or state-like occurrence.
`Quality`	Quality inhering in an entity (e.g., mass, temperature, hardness).
`InformationContent Entity`	Information artifact (e.g., document, identifier, requirement reference).
`Unknown`	Type could not be determined reliably from the clause context.

Table 13 TargetVocab: attachment points for constraints within a requirement clause.

Target	Meaning
`subject`	Constraint applies to the subject entity (bearer of obligation).
`condition`	Constraint applies to the condition/scope κ under which the requirement holds.
`action`	Constraint applies to the prescribed action/predicate (how/when/with what performance it occurs).
`object`	Constraint applies to the object argument of the action (if explicitly present).

Appendix B: LLM Prompts

This appendix reports the exact prompts used for all LLM-invoked stages.

B.1 Prompt P_δ: Decomposition.

```
<task>
Decompose the input text into atomic prescriptive
     requirements.
An atomic requirement expresses exactly ONE obligation:
one modality + one predicate applied to one subject,
possibly constrained by qualifiers (conditions, time,
     ranges).
</task>

<definition>
A requirement MUST be split if and only if:
- Two or more distinct prescriptive modalities (e.g., shall
     , must, should, will, is required to)
  govern different predicates, OR
- Coordinated predicates express independent obligations
  (e.g., "shall X and shall Y"), OR
- Bullet or numbered list items each express a standalone
     obligation.

A requirement MUST NOT be split when:
- The text contains tolerances, ranges, or uncertainty
     modifiers
  (e.g., +-10%, between X and Y, at least, no more than),
- Conditional or temporal clauses constrain a single
     predicate
  (e.g., when X, during Y, under condition Z),
- Parenthetical phrases, units, or references only refine
     an obligation.
</definition>

<instructions>
- Preserve original wording as much as possible.
- If a later clause omits the subject or modality, inherit
     it implicitly
  (e.g., "shall X and Y" -> two outputs, second may start
     with "shall Y").
- Do NOT paraphrase, normalize, or correct grammar.
- Output JSON ONLY, with the exact schema below.
- Do NOT include spans, offsets, or any fields other than
     those specified.

Output schema:
{{
  "individual_requirements": [
    {{
      "req_idx": int,
      "text": str
    }}
  ]
}}
</instructions>

<examples>
Example 1 - Single obligation:
original_text: "System shall operate at 5 V."
output:
{{
  "individual_requirements": [
    {{ "req_idx": 0, "text": "System shall operate at 5 V."
       }}
  ]
}}

Example 2 - Coordinated obligations:
original_text: "The vehicle shall start within 2 seconds
     and shall stop within 5 meters."
output:
{{
  "individual_requirements": [
    {{ "req_idx": 0, "text": "The vehicle shall start
     within 2 seconds" }},
    {{ "req_idx": 1, "text": "shall stop within 5 meters"
     }}
  ]
}}

Example 3 - Single obligation with qualifiers:
original_text: "The pump shall operate at 10 bar during
     startup."
output:
{{
  "individual_requirements": [
    {{ "req_idx": 0, "text": "The pump shall operate at 10
     bar during startup." }}
  ]
}}

Example 4 - Bullet list:
original_text:
"- Battery shall provide 20 kWh.
 - Charger shall support 400 V."
output:
{{
  "individual_requirements": [
    {{ "req_idx": 0, "text": "Battery shall provide 20 kWh
     ." }},
    {{ "req_idx": 1, "text": "Charger shall support 400 V."
       }}
  ]
}}
</examples>

<input>
original_text: {original_text}
</input>

Return JSON only.
```

B.2 Prompt P_{ε_s}: Structure Extraction.

```
<task>
Given an individual requirement clause, extract ISO/IEC/
     IEEE 29148 structural slots:
subject, modality, condition (with EARS pattern), action,
     and object.
</task>

<slot-definitions>
- subject (Span):
  The entity that bears the obligation (system/component/
     actor/artifact).
  Extract the minimal noun phrase naming the obligated
     entity.

- modality:
  Prescriptive keyword. Allowed ONLY: ["shall","must","
     should","may","will","is"].

- condition (Condition):
  A clause that scopes applicability (WHEN / UNDER WHICH
     STATE / IN RESPONSE TO WHICH EVENT).
  If absent, condition.present=false and the span fields
     are empty.

- action (Span):
  The main verb phrase describing what the subject must do.
  Exclude condition text. Include auxiliaries (e.g., "be
     able to", "be set to") and adverbials.

- object (Span):
  The explicit target/complement of the action if present (
     direct object or PP/NP complement that
  provides the target of the action or the constrained
     phrase).
  If there is no explicit object/complement, return an
     empty span.
</slot-definitions>
```

```
<ears-patterns>
Set condition.EARS_pattern using ONE label:
- ubiquitous: no explicit condition text
- event-driven: triggered by an event (when/if/upon/after/
     in the event that)
- state-driven: applies in a state (during/while/as long as
     /when in state)
- unwanted_behaviors: fault/abnormal response (upon fault/
     in case of error/failure)
- optional_features: optional/configuration dependent (if
     enabled/configured/optional)

Rules:
- If NO condition text exists → condition.present=false
     and EARS_pattern="ubiquitous".
- Otherwise condition.present=true and choose the most
     specific pattern supported by text.
- Do NOT infer implicit conditions.
</ears-patterns>

<span-rules>
- All spans MUST be exact substrings of original_text.
- Offsets are character offsets in original_text.
- Do NOT paraphrase or normalize.
- Do NOT include req_idx inside spans.
- Do NOT merge condition text into action.

IMPORTANT: Span has NO default values.
Therefore you MUST always output subject, action, and
     object as Span objects.
If a slot is not explicitly present, output the EMPTY SPAN
     sentinel:
  {{ "text": "", "start": 0, "end": 0 }}

For condition:
- If condition.present=false, set:
    condition.text="", condition.start=0, condition.end=0,
     condition.EARS_pattern="ubiquitous"
</span-rules>

<output-schema>
{{
  "req_idx": {ar.get('req_idx', 0)},
  "structure": {{
    "subject": {{ "text": str, "start": int, "end": int }},
    "modality": "shall"|"must"|"should"|"may"|"will"|"is",
    "condition": {{
      "present": bool,
      "EARS_pattern": "ubiquitous"|"event-driven"|"
     unwanted_behaviors"|"state-driven"|"optional_features
     ",
      "text": str,
      "start": int,
      "end": int
    }},
    "action": {{ "text": str, "start": int, "end": int }},
    "object": {{ "text": str, "start": int, "end": int }}
  }}
}}
</output-schema>

<examples>
Example 1 - Ubiquitous, explicit object:
original_text: "The system shall operate at 5 V."
input requirement: "The system shall operate at 5 V."
output:
{{
  "req_idx": 0,
  "structure": {{
    "subject": {{ "text": "The system", "start": 0, "end":
     10 }},
    "modality": "shall",
    "condition": {{ "present": false, "EARS_pattern": "
     ubiquitous", "text": "", "start": 0, "end": 0 }},
    "action": {{ "text": "operate", "start": 17, "end": 24
     }},
    "object": {{ "text": "at 5 V", "start": 25, "end": 31
     }}
  }}
}}

Example 2 - Event-driven:
original_text: "When powered on, the controller shall log
     faults."
input requirement: "When powered on, the controller shall
     log faults."
output:
{{
  "req_idx": 0,
  "structure": {{
    "subject": {{ "text": "the controller", "start": 17, "
     end": 31 }},
    "modality": "shall",
    "condition": {{ "present": true, "EARS_pattern": "event
     -driven", "text": "When powered on", "start": 0, "end
     ": 15 }},
    "action": {{ "text": "log", "start": 38, "end": 41 }},
    "object": {{ "text": "faults", "start": 42, "end": 48
     }}
  }}
}}

Example 3 - No explicit object (EMPTY SPAN sentinel):
original_text: "The pump shall start."
input requirement: "The pump shall start."
output:
{{
  "req_idx": 0,
  "structure": {{
    "subject": {{ "text": "The pump", "start": 0, "end": 8
     }},
    "modality": "shall",
    "condition": {{ "present": false, "EARS_pattern": "
     ubiquitous", "text": "", "start": 0, "end": 0 }},
    "action": {{ "text": "start", "start": 15, "end": 20
     }},
    "object": {{ "text": "", "start": 0, "end": 0 }}
  }}
}}
</examples>

<input>
original_text: {original_text}
input requirement: {ar.get('text','')}
</input>

Return JSON only.
```

B.3 Prompt P_{ε_C}: Constraint Extraction.

```
<task>
Extract ALL prescriptive constraints expressed in the given
      individual requirement clause.
Each output element MUST be a constraint atom: a minimal,
     self-contained statement that
constrains an attribute/relation/event via an explicit
     operator and typed value, grounded
by an evidence span in original_text.
</task>

<inputs>
- original_text: full source text (offset reference; all
     spans must refer to this)
- atomic_requirement: the clause to analyze
- structure: extracted spans (subject/modality/condition/
     action/object) to use as anchors
</inputs>

<closed-vocabularies>
Attribute.kind: "quantity" | "enum" | "boolean" | "event" |
      "relation"
Value.kind: "quantity" | "enum" | "boolean" | "entity_ref"
     | "event_ref" | "none"
```

```
Operator (use ONLY one):
  "eq","neq","lt","le","gt","ge","between",
  "one_of","all_of",
  "has_feature","type_is","uses_method",
  "before","after","during","until"
Group.relation (use ONLY one): "AND" | "OR"
Target.kind (use ONLY one): "subject"|"object"|"action"|"
     condition"|"modality"
</closed-vocabularies>

<span-rules>
- evidence.text MUST be an exact substring of original_text
     .
- evidence.start/evidence.end are character offsets in
     original_text.
- Do NOT paraphrase evidence.
- All extracted strings inside value.raw_text and ref.text
     MUST be substrings of original_text.
</span-rules>

<operator-mapping-guidelines>
Map surface forms to canonical operators:
- "at least", "no less than", "minimum" -> ge
- "at most", "no more than", "maximum"  -> le
- "less than" -> lt
- "greater than" -> gt
- "between X and Y", "from X to Y" -> between (v1=X, v2=Y)
- "±", "+/-", "plus or minus" -> quantity.tol (operator
     stays eq unless it is a range)
- temporal cues: "before"->before, "after"->after, "during
     "->during, "until"->until
- existence/parts/features:
    "shall have/include/with" -> has_feature (relation
     constraint)
    "shall be made of/type of" -> one_of/type_is depending
     on form
</operator-mapping-guidelines>

<state-vs-feature-disambiguation>
The phrase "with ..." can express either:
(A) a required part/feature (component possession), or
(B) an operating condition / configuration state.

Use (A) has_feature + entity_ref/MaterialEntity when "with
     ..." introduces a noun naming a component/part:
  e.g., "with shock absorbers", "with a filter", "with a
     protective coating".

Use (B) during + event_ref/Process when "with ..."
     introduces a state/configuration of an agent or system
     :
  e.g., "with a driver seated", "with the cover closed", "
     with the system powered on",
        "with the valve closed", "with the engine off", "
     with brakes engaged".
For (B):
  - target.kind MUST be "condition"
  - attribute.kind MUST be "event"
  - value.kind MUST be "event_ref"
  - ref.expected_type MUST be "Process"
</state-vs-feature-disambiguation>


<target-selection-guidelines>
For each constraint, you MUST output target.kind and may
     output target.ref.

1) Determine target.kind by where the constrained phrase
     semantically attaches:
   - If the constraint constrains the system/component
     named in structure.subject -> target.kind="subject"
   - If it constrains an entity introduced in structure.
     object -> target.kind="object"
   - If it constrains the action/process in structure.
     action (timing, rate of the action, ordering) ->
     target.kind="action"
   - If it constrains the applicability scope (WHEN/DURING/
     IN STATE) -> target.kind="condition"
   - If it constrains modality strength explicitly ("must"
     vs "should" etc.) -> target.kind="modality" (rare)

2) target.ref (optional):
   Use a stable pointer when possible. Use ONLY one of:
   - "SUBJECT" to refer to structure.subject
   - "OBJECT" to refer to structure.object
   - "ACTION" to refer to structure.action
   - "CONDITION" to refer to structure.condition
   - Or "C{{n}}" to refer to another constraint_idx (
     meaning: this constraint attaches to the entity
     introduced by constraint n)
   If unsure, omit ref.
</target-selection-guidelines>

<grouping-guidelines>
group is REQUIRED for every constraint.

- Default: all constraints in the clause belong to group {{
      "group_id": "g0", "relation": "AND" }}.
  Rationale: multiple constraints in one requirement clause
      are conjunctive unless the text states otherwise.

- If the clause contains explicit OR / alternatives ("or",
     "either ... or ...", "A or B"):
    Put the alternative constraints in a shared OR group, e
     .g. {{ "group_id": "g1", "relation": "OR" }}.

- If you have both AND and OR in the same clause:
    Use multiple groups:
      * One AND group for the always-required constraints (
     usually g0)
      * One OR group for alternatives (g1)
    A constraint belongs to exactly ONE group (no nesting).
</grouping-guidelines>

<depends_on-guidelines>
depends_on captures hierarchical attachment among
     constraints (modifier chains).

Use depends_on when a constraint refines another constraint
     's introduced entity, typically in patterns like:
- "X with Y" where Y is a feature/part/property of X
- "suspension system with shock absorbers ... with wheel
     travel of 50 mm"
  => constraints about shock absorbers and wheel travel
     depend_on the constraint that introduced "suspension
     system"

Rules:
- depends_on is the constraint_idx of the parent constraint
     .
- If the constraint is independent (directly about subject/
     action/condition/object slot), set depends_on=null.
- Use depends_on especially for repeated "with ..." chains
     and nested noun-phrase modifiers.
</depends_on-guidelines>

<qualifiers-guidelines>
qualifiers are booleans and MUST be present.

- qualifiers.negated=true ONLY if explicit negation exists
     in the clause
  (e.g., "shall not", "must not", "not permitted", "
     prohibited").
  Do NOT infer negation.

- qualifiers.preferred=true if:
    (a) modality is "should", OR
    (b) explicit preference markers occur ("preferably", "
     ideally", "recommended").
  Else false.

- qualifiers.scope_all=true ONLY if explicit universal
     quantifiers scope the constraint
  ("all", "each", "every"). Else false.
```

```
</qualifiers-guidelines>

<value-extraction-guidelines>
Value must match the operator and attribute kind.

- Quantity constraints:
    value.kind="quantity"
    value.quantity.v1 is required when a numeric value
     appears.
    value.quantity.v2 required only for "between" ranges.
    value.quantity.tol used for ± tolerances when present.
    value.quantity.unit_text as written (do NOT normalize
     units beyond special cases below).
    evidence should include the whole quantitative phrase (
     e.g., "minimum wheel travel of 50 mm").

- Enum constraints:
    value.kind="enum"
    enum.members_str: list of strings exactly as in text (
     trim whitespace).
    Use operator "one_of" for "A or B" alternatives (single
      atom).
    Use operator "all_of" if the text requires all listed
     members simultaneously.

- Boolean constraints:
    value.kind="boolean"
    Only set boolean if explicitly stated true/false in the
      text.

- References:
    value.kind in {"entity_ref","event_ref"}
    ref.text is the referenced phrase as written.
    expected_type guess ONLY one of:
      "MaterialEntity"|"Process"|"Quality"|"
     InformationContentEntity"|"Unknown"

- If you cannot identify a value, use value.kind="none" and
      set other value fields to null,
  but ONLY if the constraint is still meaningful (e.g.,
     existence/feature requirement).
</value-extraction-guidelines>

<output-schema>
Return JSON ONLY:

{{
  "req_idx": {ar.get('req_idx', 0)},
  "constraints": [
    {{
      "constraint_idx": int,
      "group": {{ "group_id": str, "relation": "AND"|"OR"
     }},
      "target": {{ "kind": "subject"|"object"|"action"|"
     condition"|"modality", "ref": str|null }},
      "evidence": {{ "text": str, "start": int, "end": int
     }},
      "attribute": {{ "name": str, "kind": "quantity"|"enum
     "|"boolean"|"event"|"relation" }},
      "operator": "eq"|"neq"|"lt"|"le"|"gt"|"ge"|"between
     "|"one_of"|"all_of"|"has_feature"|"type_is"|"
     uses_method"|"before"|"after"|"during"|"until",
      "value": {{
        "kind": "quantity"|"enum"|"boolean"|"entity_ref"|"
     event_ref"|"none",
        "raw_text": str,
        "quantity": {{ "v1": float|null, "v2": float|null,
     "tol": float|null, "unit_text": str|null }} | null,
        "enum": {{ "members_str": [str], "members_qty":
     [{{"v1":float|null,"v2":float|null,"tol":float|null,"
     unit_text":str|null}}] }} | null,
        "boolean": bool | null,
        "ref": {{ "text": str, "expected_type": "
     MaterialEntity"|"Process"|"Quality"|"
     InformationContentEntity"|"Unknown" }} | null
      }},
      "depends_on": int | null,
      "qualifiers": {{ "negated": bool, "preferred": bool,
     "scope_all": bool }}
    }}
  ]
}}
</output-schema>

<few-shot-examples>
Example 1 — Multiple conjunctive constraints (default AND
     group g0), with hierarchy via depends_on:
original_text: "The vehicle must have a suspension system
     with shock absorbers and wheel travel of at least 50
     mm."
atomic_requirement: "The vehicle must have a suspension
     system with shock absorbers and wheel travel of at
     least 50 mm."
structure: subject="The vehicle", action="have", object="a
     suspension system with shock absorbers and wheel
     travel of at least 50 mm"
output:
{{
  "req_idx": 0,
  "constraints": [
    {{
      "constraint_idx": 0,
      "group": {{ "group_id": "g0", "relation": "AND" }},
      "target": {{ "kind": "object", "ref": "OBJECT" }},
      "evidence": {{ "text": "suspension system", "start":
     25, "end": 41 }},
      "attribute": {{ "name": "suspension system", "kind":
     "relation" }},
      "operator": "has_feature",
      "value": {{
        "kind": "entity_ref",
        "raw_text": "suspension system",
        "quantity": null,
        "enum": null,
        "boolean": null,
        "ref": {{ "text": "suspension system", "
     expected_type": "MaterialEntity" }}
      }},
      "depends_on": null,
      "qualifiers": {{ "negated": false, "preferred": false
     , "scope_all": false }}
    }},
    {{
      "constraint_idx": 1,
      "group": {{ "group_id": "g0", "relation": "AND" }},
      "target": {{ "kind": "object", "ref": "C0" }},
      "evidence": {{ "text": "shock absorbers", "start":
     47, "end": 61 }},
      "attribute": {{ "name": "shock absorbers", "kind": "
     relation" }},
      "operator": "has_feature",
      "value": {{
        "kind": "entity_ref",
        "raw_text": "shock absorbers",
        "quantity": null,
        "enum": null,
        "boolean": null,
        "ref": {{ "text": "shock absorbers", "expected_type
     ": "MaterialEntity" }}
      }},
      "depends_on": 0,
      "qualifiers": {{ "negated": false, "preferred": false
     , "scope_all": false }}
    }},
    {{
      "constraint_idx": 2,
      "group": {{ "group_id": "g0", "relation": "AND" }},
      "target": {{ "kind": "object", "ref": "C0" }},
      "evidence": {{ "text": "wheel travel of at least 50
     mm", "start": 66, "end": 96 }},
      "attribute": {{ "name": "wheel travel", "kind": "
     quantity" }},
      "operator": "ge",
      "value": {{
```

```
        "kind": "quantity",
        "raw_text": "50 mm",
        "quantity": {{ "v1": 50.0, "v2": null, "tol": null,
      "unit_text": "mm" }},
        "enum": null,
        "boolean": null,
        "ref": null
      }},
      "depends_on": 0,
      "qualifiers": {{ "negated": false, "preferred": false
     , "scope_all": false }}
    }}
  ]
}}

Example 2 — Explicit OR alternatives (OR group g1), plus an
      AND constraint (g0):
original_text: "The housing shall be made of steel or
     aluminum and have a protective coating."
atomic_requirement: "The housing shall be made of steel or
     aluminum and have a protective coating."
output:
{{
  "req_idx": 0,
  "constraints": [
    {{
      "constraint_idx": 0,
      "group": {{ "group_id": "g1", "relation": "OR" }},
      "target": {{ "kind": "object", "ref": "OBJECT" }},
      "evidence": {{ "text": "steel or aluminum", "start":
     30, "end": 46 }},
      "attribute": {{ "name": "material", "kind": "enum"
     }},
      "operator": "one_of",
      "value": {{
        "kind": "enum",
        "raw_text": "steel or aluminum",
        "quantity": null,
        "enum": {{ "members_str": ["steel","aluminum"], "
     members_qty": [] }},
        "boolean": null,
        "ref": null
      }},
      "depends_on": null,
      "qualifiers": {{ "negated": false, "preferred": false
     , "scope_all": false }}
    }},
    {{
      "constraint_idx": 1,
      "group": {{ "group_id": "g0", "relation": "AND" }},
      "target": {{ "kind": "object", "ref": "OBJECT" }},
      "evidence": {{ "text": "protective coating", "start":
      62, "end": 79 }},
      "attribute": {{ "name": "protective coating", "kind":
      "relation" }},
      "operator": "has_feature",
      "value": {{
        "kind": "entity_ref",
        "raw_text": "protective coating",
        "quantity": null,
        "enum": null,
        "boolean": null,
        "ref": {{ "text": "protective coating", "
     expected_type": "MaterialEntity" }}
      }},
      "depends_on": null,
      "qualifiers": {{ "negated": false, "preferred": false
     , "scope_all": false }}
    }}
  ]
}}

Example — Condition-as-state introduced by "with ..." (
     event_ref):
original_text: "The device shall maintain accuracy of 1 mm
     with the cover closed."
atomic_requirement: "The device shall maintain accuracy of
     1 mm with the cover closed."
output includes:
- accuracy constraint (quantity)
- condition constraint:
  attribute.kind="event"
  operator="during"
  value.kind="event_ref"
  ref.expected_type="Process"
  evidence="with the cover closed"
</few-shot-examples>

<input>
original_text: {original_text}
atomic_requirement: {ar.get('text','')}
structure: {structure.model_dump_json()}
</input>

[SPECIAL CASES: QUANTITY NORMALIZATION]
1) Angles: unit_text="DEG" for degrees, "rad" for radians.
2) Percentages: unit_text="PERCENT".
3) Dimensionless: unit_text="DimensionlessUnit".
4) Fractional inches (e.g., 5/16"): convert to decimal v1
     and unit_text="IN".
5) Mixed units same system (e.g., 5 ft 3 in): convert to a
     single unit (prefer inches).
6) Mixed units different systems (e.g., 8mm (5/16")):
     choose ONE system (prefer SI if present) and convert.

Return JSON only.
```

B.4 Prompt P_{ε_R}: Reference Extraction.

```
<task>
Extract external references needed to interpret the atomic
     requirement.
A "reference" is any mentioned entity or information
     artifact that is not defined locally
in the atomic requirement but is required to understand,
     validate, or ground it downstream
(e.g., standards, methods, classifications, named
     procedures/events, component identifiers,
anaphoric mentions that point to prior context).

Do NOT hallucinate. If none exist, return an empty list.
</task>

<inputs>
- original_text: full source text (offset reference)
- atomic_requirement: the clause to analyze
- structure: spans for agent/modality/condition/predicate/
     object
- constraints: extracted constraint atoms (may include
     entity_ref/event_ref; use these as signals)
</inputs>

<what-counts-as-a-reference>
Create a Reference when the text includes one of the
     following:
1) Standards / regulations / specifications (e.g., "ISO
     29148", "ASTM D1234", "MIL-STD-810")
   -> expected_type = "InformationContentEntity"
2) Methods / procedures / protocols / test names (e.g., "
     per the XYZ method", "using ABC procedure")
   -> expected_type = "Process" (if it denotes an activity)
      OR "InformationContentEntity"
      (if it denotes a document/protocol). Choose the best
     fit from text.
3) Classifications / named taxonomies / compliance classes
     (e.g., "Class A", "SIL 2", "IP67")
   -> expected_type = "InformationContentEntity" (default)
     unless clearly a Quality label.
4) Named components / part numbers / drawing IDs / model
     identifiers referenced but not defined
   (e.g., "Valve V-101", "P/N 123-456", "Drawing DWG-001")
   -> expected_type = "MaterialEntity"
5) Named events or operating modes referenced but not
```

```
     defined locally
   (e.g., "shutdown event", "startup sequence", "emergency
     stop")
   -> expected_type = "Process" (event/occurrence)
6) Anaphora / deixis referring to prior context (e.g., "it
     ", "they", "this valve", "the above")
   -> expected_type = inferred best guess (often "
     MaterialEntity" or "InformationContentEntity"),
      but do NOT resolve it; set resolves_to = null.
</what-counts-as-a-reference>

<what-does-not-count>
Do NOT output references for:
- Generic nouns fully defined by local context ("the system
     ", "the valve") unless anaphoric
  ("it", "they", "this component") or explicitly pointing
     to earlier content ("the above").
- Units, numbers, tolerances, or pure quantitative values (
     those are constraints, not references).
- Common verbs or adjectives ("operate", "robust") with no
     external dependency.
</what-does-not-count>

<span-and-output-rules>
- evidence.text MUST be an exact substring of original_text
     .
- evidence.start/end are character offsets in original_text
     .
- resolves_to MUST be null (do not attempt entity
     resolution here).
- Do NOT include req_idx inside spans.
- Return JSON only matching ReferencesResponse schema.
</span-and-output-rules>

<output-schema>
{{
  "req_idx": {ar.get('req_idx', 0)},
  "references": [
    {{
      "ref_idx": int,
      "evidence": {{ "text": str, "start": int, "end": int
     }},
      "expected_type": "MaterialEntity" | "Process" | "
     Quality" | "InformationContentEntity" | "Unknown",
      "resolves_to": null
    }}
  ]
}}
</output-schema>

<few-shot-examples>
Example 1 — Standard reference:
original_text: "The system shall comply with ISO 29148."
atomic_requirement: "The system shall comply with ISO
     29148."
output:
{{
  "req_idx": 0,
  "references": [
    {{
      "ref_idx": 0,
      "evidence": {{ "text": "ISO 29148", "start": 28, "end
     ": 36 }},
      "expected_type": "InformationContentEntity",
      "resolves_to": null
    }}
  ]
}}

Example 2 — Method / procedure reference:
original_text: "The coating shall be tested according to
     ASTM D3359."
atomic_requirement: "The coating shall be tested according
     to ASTM D3359."
output:
{{
  "req_idx": 0,
  "references": [
    {{
      "ref_idx": 0,
      "evidence": {{ "text": "ASTM D3359", "start": 40, "
     end": 50 }},
      "expected_type": "InformationContentEntity",
      "resolves_to": null
    }}
  ]
}}

Example 3 — Component identifier reference:
original_text: "Valve V-101 shall remain closed during
     startup."
atomic_requirement: "Valve V-101 shall remain closed during
      startup."
output:
{{
  "req_idx": 0,
  "references": [
    {{
      "ref_idx": 0,
      "evidence": {{ "text": "V-101", "start": 5, "end": 10
      }},
      "expected_type": "MaterialEntity",
      "resolves_to": null
    }},
    {{
      "ref_idx": 1,
      "evidence": {{ "text": "startup", "start": 36, "end":
      43 }},
      "expected_type": "Process",
      "resolves_to": null
    }}
  ]
}}

Example 4 — Anaphora:
original_text: "The valve shall be installed upstream. It
     shall withstand 2 bar."
atomic_requirement: "It shall withstand 2 bar."
output:
{{
  "req_idx": 0,
  "references": [
    {{
      "ref_idx": 0,
      "evidence": {{ "text": "It", "start": 34, "end": 36
     }},
      "expected_type": "MaterialEntity",
      "resolves_to": null
    }}
  ]
}}
</few-shot-examples>

<input>
original_text: {original_text}
atomic_requirement: {ar.get('text','')}
structure: {structure.model_dump_json()}
constraints: {constraints_json}
</input>

Return JSON only.
```

B.5 Prompt P_{v_c}: Unit context normalization.

```
You are an expert in physical quantities, units of
     measurement, and engineering requirements.
Given an input sentence and an extracted unit token (which
     may be informal or partially specified),
write a short explanation of what the unit represents in
     that sentence, including the likely
physical quantity, whether it is absolute or relative (e.g
     ., gauge vs absolute pressure),
and what object or phenomenon it describes.
```

```
Constraints:
- Output 2-3 concise sentences.
- Keep the explanation focused on how the unit is being
     used in the provided sentence.
- Do not invent details not implied by the sentence.

Few-shot examples:

Input sentence:
"The vessel shall be designed for 600 psig during normal
     operation."
Unit: "psig"
Explanation:
"The unit psig is pounds per square inch gauge, indicating
     pressure relative to atmospheric pressure. Here it
     describes the vessel's design operating pressure."

Input sentence:
"The signal must settle within 20 ms after activation."
Unit: "ms"
Explanation:
"ms is milliseconds, a unit of time duration. It refers to
     how quickly the signal must stabilize after activation
     ."

Input sentence:
"Flow shall be at least 5 lpm through the cooling loop."
Unit: "lpm"
Explanation:
"lpm stands for liters per minute, a volumetric flow rate.
     It specifies the required flow through the cooling
     loop."

Now analyze the new sentence.

Input sentence:
"{input_sentence}"
Unit: "{unit}"
Explanation:
```

B.6 Prompt P_{v_q}: Natural Language Query generation for semantic search in I_{QUDT}.

```
You are an expert in physical quantities, units of
     measurement, and the QUDT ontology.
Your task is to formulate a single concise natural-language
      query that will be used
for semantic search over a ChromaDB collection of QUDT
     quantity kinds and their units.

Goal:
Given a QUDT unit code, write a natural-language query that
     :
1. Expresses what physical quantity the unit measures.
2. Includes your best guess of the QUDT quantity kind name
     in CamelCase
   (for example: Velocity, Mass, Pressure, Torque,
     VolumetricFlowRate).
3. Mentions the unit code itself.

The semantic index you are querying contains documents with
     :
- the quantity kind URI,
- a QUDT-style quantity kind name (e.g. "AbsoluteHumidity",
      "Work", "Torque"),
- the unit code.

Constraints:
- Output exactly ONE short, grammatically correct sentence.
- Do NOT explain your reasoning.
- Do NOT output any quotation marks around the sentence.
- Explicitly include the guessed QUDT quantity kind name in
      CamelCase in the sentence.
- Use the provided unit code verbatim.
- Tolerance phrases (e.g., "±10") simply describe allowed
     variation around a value; they do not change the
     requirement or operator and should not spawn extra
     requirements.
- If you are unsure of the exact QUDT class name, choose
     the most reasonable CamelCase name you can
  (e.g., "LinearVelocity", "AngularVelocity", "Volume", "
     Power", "Energy").

Additional context about how the unit appears in the input
     sentence (use this to guide the guessed quantity kind)
     :
{unit_context if unit_context is not None else "No
     additional context provided."}

Few-shot examples:

Unit: "M-PER-SEC"
Natural-language query:
Find the QUDT quantity kind Velocity that is measured in
     meters per second (M-PER-SEC).

Unit: "KiloGM"
Natural-language query:
Retrieve the QUDT quantity kind Mass whose unit is kilogram
      (KiloGM).

Unit: "PA"
Natural-language query:
Find the QUDT quantity kind Pressure that is measured in
     pascals (PA).

Unit: "N-M"
Natural-language query:
Retrieve the QUDT quantity kind Torque whose unit of
     measure is newton metre (N-M).

Unit: "L-PER-MIN"
Natural-language query:
Find the QUDT quantity kind VolumetricFlowRate that uses
     litre per minute (L-PER-MIN) as a unit.

Now generate the natural-language query for the new unit.

Unit: {unit}
Natural-language query:
```

B.7 Prompt P_{v_u}: Best unit selection prompt.

```
You are an expert in physical quantities, engineering
     requirements, and the QUDT ontology.
Your task is to select the most appropriate SI unit from a
     given list of QUDT unit URIs,
based on the context provided by an input sentence and the
     associated quantity kind.

Inputs:
1. An input sentence describing a requirement, constraint,
     or parameter.
2. A list of candidate SI unit URIs from QUDT. Each URI has
      the form:
   http://qudt.org/vocab/unit/<UNIT_CODE>
3. The URI of the quantity kind in QUDT.

Your job:
- Choose exactly ONE unit URI from the given list that best
      fits the meaning and scale
  implied by the input sentence for the given quantity kind
     .
- Always choose a unit that appears in the provided list;
     never invent a new URI.
- If multiple units are plausible, choose the one that fits
      best with respect to the input sentence provided.
- Tolerances (e.g., "±10" or "plus or minus") simply
     describe allowed variation around the stated value and
      do not change the underlying requirement or operator.
- Return ONLY the chosen unit URI, with no explanation or
```

```
     extra text.

Few-shot examples:

Example 1
---------
Input sentence:
"The beam length shall be specified with a tolerance of ±
     0.5."

Quantity kind:
http://qudt.org/vocab/quantitykind/Length

Candidate units:
[
  "http://qudt.org/vocab/unit/M",
  "http://qudt.org/vocab/unit/CentiM",
  "http://qudt.org/vocab/unit/MilliM"
]

Best unit URI:
"http://qudt.org/vocab/unit/MilliM"

(Reasoning, not to be output: Length tolerances in
     mechanical design are often expressed in millimetres.)

Example 2
---------
Input sentence:
"The design pressure of the vessel shall not exceed 10^6 Pa
     ."

Quantity kind:
http://qudt.org/vocab/quantitykind/Pressure

Candidate units:
[
  "http://qudt.org/vocab/unit/PA",
  "http://qudt.org/vocab/unit/KiloPA",
  "http://qudt.org/vocab/unit/BAR"
]

Best unit URI:
"http://qudt.org/vocab/unit/PA"

(Reasoning, not to be output: PA is the unit code in QUDT
     for pascals (Pa, which is cited in the input sentence)
     .

Example 3
---------
Input sentence:
"The motor shall provide a torque of at least 50 Nm at the
     shaft."

Quantity kind:
http://qudt.org/vocab/quantitykind/Torque

Candidate units:
[
  "http://qudt.org/vocab/unit/N-M",
  "http://qudt.org/vocab/unit/N",
  "http://qudt.org/vocab/unit/J"
]

Best unit URI:
"http://qudt.org/vocab/unit/N-M"

(Reasoning, not to be output: Torque in the input sentence
     is measured in newton metres, not in newtons or joules
     .)

Now solve the following instance.

Input sentence:
"{input_sentence}"

Quantity kind:
{quantity_kind}

Candidate units:
[{units_str}]

Remember:
- Select exactly ONE URI from the list.
- Answer with the URI only, no explanation and no
     additional text.
```

B.8 Prompt $P_{v_{qk}}$: Best quantity kind selection.

```
You are an expert in physical quantities, units of
     measurement, and the QUDT ontology.

Your task is to select the most appropriate QUDT quantity
     kind from a given list of QUDT quantity kind URIs,
     based on the context provided by an input sentence.

Inputs:
1. An input sentence describing a requirement, constraint,
     or parameter.
2. A list of candidate QUDT quantity kind URIs.
Your job:
- Choose exactly ONE quantity kind URI from the given list
     that best fits the meaning implied by the input
     sentence.
- Always choose a quantity kind that appears in the
     provided list; never invent a new URI.
- If multiple quantity kinds are plausible, choose the one
     that fits best with respect to the input sentence
     provided.
- Return ONLY the chosen quantity kind URI, with no
     explanation or extra text.
Few-shot examples:
Example 1
---------
Input sentence:
"The beam length shall be specified with a tolerance of ±
     0.5."
Quantity kind candidates:
[
  "http://qudt.org/vocab/quantitykind/Length",
  "http://qudt.org/vocab/quantitykind/Mass",
  "http://qudt.org/vocab/quantitykind/Time"
]
Best quantity kind URI:
"http://qudt.org/vocab/quantitykind/Length"

Example 2
---------
Input sentence:
"The design pressure of the vessel shall not exceed 10^6 Pa
     ."
Quantity kind candidates:
[
  "http://qudt.org/vocab/quantitykind/Pressure",
  "http://qudt.org/vocab/quantitykind/Temperature",
  "http://qudt.org/vocab/quantitykind/Volume"
]
Best quantity kind URI:
"http://qudt.org/vocab/quantitykind/Pressure"
Example 3
---------
Input sentence:
"The motor shall provide a torque of at least 50 Nm at the
     shaft."
Quantity kind candidates:
[
  "http://qudt.org/vocab/quantitykind/Torque",
  "http://qudt.org/vocab/quantitykind/Energy",
  "http://qudt.org/vocab/quantitykind/Force"
]
Best quantity kind URI:
"http://qudt.org/vocab/quantitykind/Torque"
```

```
Now solve the following instance.
Input sentence:
"{input_sentence}"
Quantity kind candidates:
[{quantity_kind_candidates}]
Best quantity kind URI:
```

B.9 Prompt P_{γ}: Grounding.

```
You are an ontological engineer. Given a structured
     extraction of semantic entities from a requirement
     sentence, produce an OWL graph in Turtle (.ttl) format
     .

Inputs you must use directly:
- Structured extraction + alignment payload (JSON below),
     including `normalized_quantities` entries from the
     NormalizedRecord.
- Full IOF Core ontology (Core.rdf, RDF/XML) as
     authoritative vocabulary and axioms.

Goals (in order):
1) Map the whole requirement into IOF classes and
     properties. Do not define new classes or properties of
      any kind.
2) Avoid blank nodes; mint readable IRIs under the base.
3) Only for true quantitative constraints, create a value
     individual typed iof:ValueExpression.
4) Reuse individuals rather than duplicating (e.g., the
     same quality/agent that the value qualifies).
5) Keep the graph OWL DL compatible and minimal.

Graph pattern constraints (mandatory):

- Create exactly one iof:RequirementSpecification
     individual :Req_0 and attach a comment to it with the
     original_text of the input requirement.
- The requirement specification individual must be
     connected to a specification instance via iof:
     requirementSatisfiedBy.
- Use iof:DesignSpecification when the requirement
     constrains continuants (artifacts/material entities/
     qualities of continuants).
- Use iof:PlanSpecification when the requirement constrains
      processes, events, procedures, or process
     characteristics.
- Do not use QUDT classes or properties. They will be
     handled in a later enrichment step. Once you
     instantiate a iof:ValueExpression, that must be a leaf
      node with no further QUDT properties.
- Use iof:ValueExpression class for value individuals. Do
     not use its subclasses (such as iof:
     MeasuredValueExpression).
- Connect each quantitative ValueExpression to its
     constrained bearer using iof:
     isValueExpressionOfAtSomeTime / iof:
     isValueExpressionOfAtAllTimes (or the inverse
     hasValueExpression* relation from the bearer).
- iof:hasSpecifiedOutput has domain iof:PlannedProcess.
     Never use iof:hasSpecifiedOutput on any specification
     individual (DesignSpecification / PlanSpecification /
     RequirementSpecification / ObjectiveSpecification /
     InformationContentEntity). Use iof:prescribes for
     specification semantics.
- A iof:DesignSpecification MUST NOT prescribe occurrents.
     In particular, never assert:
  - DesignSpecification iof:prescribes Process
  - DesignSpecification iof:prescribes PlannedProcess
  - DesignSpecification iof:prescribes
     ProcessCharacteristic
  - DesignSpecification iof:prescribes Event
- For every quantitative constraint, create exactly one
     value node with stable naming `:VE_req<req_idx>_c<
     constraint_idx>`. Do not create additional
     quantitative ValueExpression nodes for the same
     constraint.
- Never create iof:ValueExpression for relation, enum,
     boolean, event_ref, entity_ref, or textual-conformance
      constraints.
- Only create iof:ValueExpression when the payload has
     matching evidence:
  - `constraints[*].value.kind == "quantity"` and
  - a corresponding `normalized_quantities` entry for the
     same `(req_idx, constraint_idx)`.
- If there are zero quantitative constraints / zero `
     normalized_quantities` rows, output MUST contain zero
     `:VE_req*` individuals and zero `iof:ValueExpression`
      instances.
- If a specification prescribes any process-like entity,
     type it as iof:PlanSpecification (not iof:
     DesignSpecification).
- If a value-expression node has QUDT numeric/bound data
     properties, do not assert iof:hasSimpleExpressionValue
      on that same node.

Mandatory prefixes (always include):
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix iof: <https://spec.industrialontologies.org/
     ontology/core/Core/> .
@prefix bfo: <http://purl.obolibrary.org/obo/> .
@prefix qudt: <http://qudt.org/schema/qudt/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://example.org/req/{record_idx}#> .

Authoritative ontology (full Core.rdf content):
{ontology_context}

Return ONLY OWL/Turtle (TTL syntax), no commentary, no RDF/
     XML, base IRI http://example.org/req/{record_idx}#
Before final output, verify each req_idx and each
     constraint_idx is represented in triples.
```

B.10 Prompt $P_{\gamma r}$: KG repair by feedback loop.

```
The previous owl graph was inconsistent or incomplete. Fix
     it and output corrected OWL/Turtle only.

Rules (must follow):
- Keep base IRI and identifiers.
- Do not invent new classes/properties; only IOF/BFO IRIs
     provided in the context.
- Represent the full requirement (subject, modality,
     condition, action, object, constraints, references) in
      IOF/BFO.
- Prefer tightening types/domains to restore consistency
     over dropping constraints.
- Do not use iof:hasSpecifiedOutput on specification
     individuals; iof:hasSpecifiedOutput is only for iof:
     PlannedProcess subjects.
- If a value node has QUDT numeric/bound data properties,
     do not keep iof:hasSimpleExpressionValue on that same
     node.

Ontology context (Core.rdf):
{self.ontology_context}

Inputs:
- Structured payload (authoritative):
{json.dumps(self.payload, indent=2)}
- Previous graph:
{inconsistent_owl}
- Reasoner report:
{report}

Output constraints:
- Return Turtle syntax only.
- Do not return RDF/XML.
- Do not include XML declarations (for example: <?xml
     ...?>).
```

B.11 Prompt $P_{\gamma_{QUDT}}$: KG enrichment prompt for adding units, quantitykinds, and values.

```
Enrich this IOF graph with QUDT according to the IOF-QUDT
     guideline.

Mandatory pattern:
- Quantitative value nodes must be iof:ValueExpression and
     qudt:QuantityValue.
- Connect value node to unit with qudt:unit.
- Connect unit node to quantity kind with qudt:
     hasQuantityKind.
- Quantity kind nodes must be explicitly declared as
     individuals with `a qudt:QuantityKind`.
- Prefer `qudtu:` for unit individuals and `qudtqk:` for
     quantity-kind individuals in Turtle serialization.
- Do not use qudt:hasQuantityKind on value nodes.
- Do not use iof:*MeasuredValue* properties.
- For bounded constraints, use:
  - qudt:minInclusive / qudt:maxInclusive when inclusive
  - qudt:minExclusive / qudt:maxExclusive when exclusive
  - qudt:lowerBound / qudt:upperBound when inclusivity is
     unknown
- Never use qudt:lowerBoundInclusive or qudt:
     upperBoundInclusive.
- If bounds are present, do not keep qudt:numericValue.
- If qudt:numericValue or any QUDT bounds are present on a
     node, do not keep iof:hasSimpleExpressionValue on that
      node.
- Guardrail: apply QUDT ONLY to constraints that are
     quantitative in NormalizedRecord.
- If NormalizedRecord has zero quantitative rows, return
     the graph unchanged except for removing accidental
     quantitative artifacts (`iof:ValueExpression`, `qudt
     :*` nodes/triples) that are not backed by
     NormalizedRecord.
- Never create `iof:ValueExpression` or `qudt:
     QuantityValue` from textual conformance/reference
     constraints.

NormalizedRecord quantitative row count:
{quant_row_count}

NormalizedRecord quantitative summary:
{normalized_brief}

Graph:
{owl_text}
```

B.12 Prompt $P_{\gamma_{QUDT,r}}$: QUDT repair prompt.

```
Repair the graph so that it follows this IOF+QUDT pattern
     exactly:
- Quantitative value node: rdf:type iof:ValueExpression and
      qudt:QuantityValue.
- Value node uses qudt:unit -> Unit node.
- Quantity kind must be on Unit: Unit qudt:hasQuantityKind
     QuantityKind.
- Each quantity kind resource must be explicitly typed: `a
     qudt:QuantityKind`.
- Do NOT place qudt:hasQuantityKind on ValueExpression
     nodes.
- Bounds use qudt:minInclusive/maxInclusive or qudt:
     minExclusive/maxExclusive, or qudt:lowerBound/qudt:
     upperBound when inclusivity is unknown.
- Never use qudt:lowerBoundInclusive or qudt:
     upperBoundInclusive.
- If bounds exist, remove qudt:numericValue.
- If qudt:numericValue or any QUDT bounds are present on a
     node, remove iof:hasSimpleExpressionValue from that
     node.
- Use iof:hasValueExpressionAtSomeTime/AtAllTimes and
     inverses; do not use measured-value predicates.
- Keep existing IRIs and requirement meaning.
- Prefer `qudtu:` for unit individuals and `qudtqk:` for
     quantity-kind individuals in Turtle serialization.
- Guardrail: only quantitative constraints from
     NormalizedRecord may use iof:ValueExpression / qudt:
     QuantityValue.
- If NormalizedRecord has zero quantitative rows, remove
     any accidental iof:ValueExpression/qudt:* quantitative
      nodes and return a graph with no quantitative
     modeling.

NormalizedRecord quantitative row count:
{quant_row_count}

NormalizedRecord quantitative summary:
{normalized_brief}

Detected violations:
{json.dumps(violations, indent=2)}

Graph to repair:
{owl_text}

Return OWL/Turtle only.
Never return RDF/XML or XML declarations.
```

Appendix C: SPARQL Conformance Checks

This appendix reports the exact SPARQL queries used to compute the six conformance checks $\mathcal{K}$ described in Section 4. All checks share the common prefix block reported below.

C.1 Common Prefix Block.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX iof: <https://spec.industrialontologies.org/
     ontology/core/Core/>
PREFIX qudt: <http://qudt.org/schema/qudt/>
```

C.2 C01: Requirement Typed as `iof:RequirementSpecification`.

```
SELECT ?req ?spec WHERE {
  ?req iof:requirementSatisfiedBy ?spec .
  FILTER NOT EXISTS { ?req a iof:RequirementSpecification
      . }
}
```

C.3 C02: Requirement Has `iof:requirementSatisfiedBy`.

```
SELECT ?req WHERE {
  ?req a iof:RequirementSpecification .
  FILTER NOT EXISTS { ?req iof:requirementSatisfiedBy ?
     spec . }
}
```

C.4 C03: Satisfaction Target Typed as `iof:DesignSpecification` or `iof:PlanSpecification`.

```
SELECT ?req ?spec WHERE {
  ?req iof:requirementSatisfiedBy ?spec .
  FILTER NOT EXISTS {
    { ?spec a iof:DesignSpecification . }
    UNION
    { ?spec a iof:PlanSpecification . }
  }
}
```

C.5 C04: `qudt:QuantityValue` Has a Unit.

```
SELECT ?value WHERE {
  ?value a qudt:QuantityValue .
  FILTER NOT EXISTS { ?value qudt:unit ?unit . }
}
```

C.6 C05: Unit Declares `qudt:hasQuantityKind`.

```
SELECT ?value ?unit WHERE {
  ?value a qudt:QuantityValue ;
         qudt:unit ?unit .
  FILTER NOT EXISTS { ?unit qudt:hasQuantityKind ?qk . }
}
```

C.7 C06: QuantityValue Linked via `isValueExpressionOf-`.

```
SELECT ?value WHERE {
  ?value a qudt:QuantityValue .
  FILTER NOT EXISTS {
    { ?value iof:isValueExpressionOfAtSomeTime ?entity .
     }
    UNION
    { ?value iof:isValueExpressionOfAtAllTimes ?entity .
     }
  }
}
```

References

[1] Pahl, G. and Beitz, W., 1996, *Engineering Design*, Springer London, London, UK.

[2] 2018, "ISO/IEC/IEEE 29148:2018 Systems and software engineering — Life cycle processes — Requirements engineering," doi: 10.1109/IEEESTD.2018.8559686.

[3] Gero, J. S., 1990, "Design Prototypes: A Knowledge Representation Schema for Design," AI Magazine, **11**(4), p. 26.

[4] Avşar, A. Z. and Grogan, P. T., 2024, "Identification of Design Strategies and Their Effects on Performance Outcomes in Pair Parameter Design Tasks," Journal of Mechanical Design, **146**(5), 051401.

[5] Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., Ajagbe, M. A., Chioasca, E.-V., and Batista-Navarro, R. T., 2021, "Natural Language Processing for Requirements Engineering: A Systematic Mapping Study," ACM Computing Surveys, **54**(3), p. 1–41.

[6] Norheim, J. J., Rebentisch, E., Xiao, D., Draeger, L., Kerbrat, A., and de Weck, O. L., 2024, "Challenges in applying large language models to requirements engineering tasks," Design Science, **10**, p. e16.

[7] Aurum, A. and Wohlin, C., 2005, "Requirements Engineering: Setting the Context," *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–15.

[8] Kassab, M., Neill, C., and Laplante, P., 2014, "State of practice in requirements engineering: contemporary data," Innovations in Systems and Software Engineering, **10**(4), p. 235–241.

[9] Goyal, A., Gupta, V., and Kumar, M., 2018, "Recent Named Entity Recognition and Classification techniques: A systematic review," Computer Science Review, **29**, p. 21–43.

[10] Sudhi, V., Kutty, L., and Gröpler, R., 2023, "Natural Language Processing for Requirements Formalization: How to Derive New Approaches?" *Concurrency, Specification and Programming*, Springer International Publishing, p. 1–27.

[11] Lim, S. C., 2022, "A Case for Pre-trained Language Models in Systems Engineering," S.m. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

[12] Colelough, B. C. and Regli, W., 2024, "Neuro-Symbolic AI in 2024: A Systematic Review," *Proceedings of the First International Workshop on Logical Foundations of Neuro-Symbolic AI (LNSAI 2024)*, CEUR-WS.org, Jeju, South Korea, 2024-08-05, pp. 1–12, doi: 10.48550/arXiv.2501.05435.

[13] Drobnjakovic, M., Kulvatunyou, B., Ameri, F., Will, C., Smith, B., and Jones, A., 2022, "The Industrial Ontologies Foundry (IOF) Core Ontology," *Proceedings of the 12th International Workshop on Formal Ontologies Meet Industry (FOMI 2022)*, IOS Press, Bolzano, Italy, 2022-09-12/2022-09-15, pp. 1–13.

[14] FAIRsharing.org, 2026, "QUDT; Quantities, Units, Dimensions and Types," FAIRsharing record, doi: 10.25504/FAIRsharing.d3pqw7, Last edited: 2026-01-12 09:48; Last editor: delphinedauga; Last reviewed: 2025-03-25 10:11.

[15] Failla, L., Rossoni, M., Quirini, M., and Colombo, G., 2025, "Managing lifecycle of product information with an ontology-based knowledge framework," Journal of Industrial Information Integration, **45**, p. 100820.

[16] Failla, L., Rossoni, M., and Colombo, G., 2025, "Ontology-Based Product Lifecycle Management: Insights From a Proof-of-Concept Implementation," IEEE Access, **13**, p. 179295–179314.

[17] Suh, N. P., 1998, "Axiomatic Design Theory for Systems," Research in Engineering Design, **10**(4), p. 189–209.

[18] Stone, R. B. and Wood, K. L., 1999, "Development of a Functional Basis for Design," Journal of Mechanical Design, **122**(4), pp. 359–370.

[19] Ullman, D. G., 2002, "Toward the ideal mechanical engineering design support system," Research in Engineering Design, **13**(2), p. 55–64.

[20] Chandrasegaran, S. K., Ramani, K., Sriram, R. D., Horváth, I., Bernard, A., Harik, R. F., and Gao, W., 2013, "The evolution, challenges, and future of knowledge representation in product design systems," Computer-Aided Design, **45**(2), p. 204–228.

[21] Mavin, A., Wilkinson, P., Harwood, A., and Novak, M., 2009, "Easy Approach to Requirements Syntax (EARS)," *2009 17th IEEE International Requirements Engineering Conference*, Atlanta, Georgia, USA, 2009-08-31/2009-09-04, pp. 317–322, doi: 10.1109/RE.2009.9.

[22] Pohl, K. and Rupp, C., 2011, *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB compliant*, 1st ed., Rocky Nook.

[23] Tiwari, S., Shah, P., and Khare, M., 2022, "NL2RT: A Tool to Translate Natural Language Text into Requirements Templates (RTs)," *2022 IEEE 30th International Requirements Engineering Conference (RE)*, Melbourne, Australia, 2022-08-15/2022-08-19, pp. 262–263, doi: 10.1109/RE54965.2022.00035.

[24] Asudani, D. S., Nagwani, N. K., and Singh, P., 2023, "Impact of word embedding models on text analytics in deep learning environment: a review," Artificial Intelligence Review, **56**(9), p. 10345–10425.

[25] Kabootari, M., Abdeahad, Y., Kheirkhah, Y., and Kheirkhah, E., 2025, "Enhancing software requirements classification: a comparative study of deep learning model integration with embedding techniques," Computing, **107**(10), 1546.

[26] Jp, S., Menon, V. K., Soman, K., and Ojha, A. K. R., 2022, "A Non-Exclusive Multi-Class Convolutional Neural Network for the Classification of Functional Requirements in AUTOSAR Software Requirement Specification Text," IEEE Access, **10**, p. 117707–117714.

[27] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I., 2017, "Attention is All you Need," *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds., Vol. 30, Curran Associates, Inc., Long Beach, CA, USA, 2017-12-04/2017-12-09, pp. 5998–6008.

[28] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H., 2024, "Large Language Models for Software Engineering: A Systematic Literature Review," ACM Transactions on Software Engineering and Methodology, **33**(8), p. 1–79.

[29] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K., 2019, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, eds., Association for Computational Linguistics, Minneapolis, Minnesota, USA, 2019-06-02/2019-06-07, pp. 4171–4186, doi: 10.18653/v1/N19-1423.

[30] Gräßler, I., Preuß, D., Brandt, L., and Mohr, M., 2022, "Efficient Extraction of Technical Requirements Applying Data Augmentation," *2022 IEEE International Symposium on Systems Engineering (ISSE)*, IEEE, Vienna, Austria, 2022-10-24/2022-10-26, pp. 1–8, doi: 10.1109/ISSE54508.2022.10005452.

[31] Schiller, K. A., Haddad, M.-S., and Seibel, A., 2025, "ReqGPT: a fine-tuned large language model for generating requirements documents," Proceedings of the Design Society, **5**, p. 2741–2750.

[32] Holter, M. O. and Ell, B., 2023, "Reading Between the Lines: Information Extraction from Industry Requirements," *Proceedings of the Conference Recent Advances in Natural Language Processing (RANLP 2023)*, INCOMA Ltd., Varna, Bulgaria, 2023-09-01/2023-09-03, pp. 703–711, doi: 10.26615/978-954-452-092-2_076.

[33] Perko, A. and Wotawa, F., 2024, "Evaluating OpenAI Large Language Models for Generating Logical Abstractions of Technical Requirements Documents," *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, Cambridge, United Kingdom, 2024-07-01/2024-07-05, pp. 238–249, doi: 10.1109/QRS62785.2024.00032.

[34] Norheim, J. J. and Rebentisch, E., 2024, "Structuring Natural Language Requirements with Large Language Models," *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, Reykjavik, Iceland, 2024-06-24/2024-06-25, pp. 68–71, doi: 10.1109/REW61692.2024.00013.

[35] Ronanki, K., Cabrero-Daniel, B., Horkoff, J., and Berger, C., 2024, "Requirements Engineering Using Generative AI: Prompts and Prompting Patterns," *Generative AI for Effective Software Development*, Springer Nature Switzerland, p. 109–127.

[36] Doris, A. C., Grandi, D., Tomich, R., Alam, M. F., Ataei, M., Cheong, H., and Ahmed, F., 2024, "DesignQA: A Multimodal Benchmark for Evaluating Large Language Models' Understanding of Engineering Documentation," Journal of Computing and Information Science in Engineering, **25**(2), p. 021009.

[37] Regenwetter, L., Nobari, A. H., and Ahmed, F., 2022, "Deep Generative Models in Engineering Design: A Review," Journal of Mechanical Design, **144**(7), p. 071704.

[38] Grandi, D., Jain, Y. P., Groom, A., Cramer, B., and McComb, C., 2025, "Evaluating Large Language Models for Material Selection," Journal of Computing and Information Science in Engineering, **25**(2), p. 021003.

[39] Sanfilippo, E. M., Kitamura, Y., and Young, R. I., 2019, "Formal ontologies in manufacturing," Applied Ontology, **14**(2), pp. 119–125.
[40] Usman, Z., Young, R., Chungoora, N., Palmer, C., Case, K., and Harding, J., 2013, "Towards a formal manufacturing reference ontology," International Journal of Production Research, **51**(22), p. 6553–6572.
[41] Lin, J., Fox, M. S., and Bilgic, T., 1996, "A Requirement Ontology for Engineering Design," Concurrent Engineering, **4**(3), p. 279–291.
[42] Colombo, G., Mosca, A., and Sartori, F., 2007, "Towards the design of intelligent CAD systems: An ontological approach," Advanced Engineering Informatics, **21**(2), p. 153–168.
[43] Arp, R., Smith, B., and Spear, A. D., 2015, *Building Ontologies with Basic Formal Ontology*, The MIT Press.
[44] Object Management Group, 2019, "OMG Systems Modeling Language (SysML)," Object Management Group, Specification Version 1.6 (formal/19-11-01), Accessed: 2026-02-15.
[45] Edge, D., Trinh, H., Cheng, N., Bradley, J., Chao, A., Mody, A., Truitt, S., Metropolitansky, D., Ness, R. O., and Larson, J., 2025, "From Local to Global: A Graph RAG Approach to Query-Focused Summarization," arXiv2404.16130, https://arxiv.org/abs/2404.16130
[46] Liu, P., Qian, L., Zhao, X., and Tao, B., 2024, "Joint Knowledge Graph and Large Language Model for Fault Diagnosis and Its Application in Aviation Assembly," IEEE Transactions on Industrial Informatics, **20**(6), p. 8160 – 8169, Cited by: 133.
[47] Massoudi, S. and Fuge, M., 2025, "Agentic Large Language Models for Conceptual Systems Engineering and Design," *Volume 3B: 51st Design Automation Conference (DAC)*, American Society of Mechanical Engineers, Anaheim, CA, USA, 2025-08-17/2025-08-20, p. V03BT03A045, doi: 10.1115/DETC2025-168856.
[48] Pan, X., Zhuang, W., Wen, S., Yu, W., Bao, J., and Li, X., 2025, "A context-aware KG-LLM collaborated conceptual design approach for personalized products: A case in lower limbs rehabilitation assistive devices," Advanced Engineering Informatics, **66**, p. 103422.
[49] Wang, Y., Goridkov, N., Rao, V., Cui, D., Grandi, D., and Goucher-Lambert, K., 2023, "Embedding Experiential Design Knowledge in Interactive Knowledge Graphs," Journal of Mechanical Design, **145**(4).
[50] Drobnjakovic, M., 2023, "Guideline for Using QUDT, if were to use with IOF Ontologies," Industrial Ontologies Foundry (IOF), accessed June 25, 2026, https://oagi.atlassian.net/wiki/spaces/IOF/pages/4679696397/Guideline+for+Using+QUDT+if+were+to+use+with+IOF+Ontologies
[51] Falco, R., Gangemi, A., Peroni, S., Shotton, D., and Vitali, F., 2014, "Modelling OWL Ontologies with Graffoo," *The Semantic Web: ESWC 2014 Satellite Events*, Springer International Publishing, p. 320–325.
[52] Cal Poly CubeSat Laboratory, 2022, "CubeSat Design Specification (CDS) Rev. 14.1," California Polytechnic State University, San Luis Obispo, https://www.cubesat.org/s/CDS-REV14_1-2022-02-09.pdf
[53] Centre for Automation and Robotics (CAR), 2019, "D4.1 Definition of Specifications for Robotic Prototype," SUREVEG Project, Project Deliverable D4.1, Work Package WP4: Smart machinery for strip-cropping systems.
[54] Aaltonen, J., Koskinen, K., Laitinen, J., Friman, E., Hakonen, K., Salomaa, T., and Ulmanen, P., 2020, "System Requirements: Robominer Requirement Specification," ROBOMINERS Project, Project Deliverable D3.1, ROBOMINERS Deliverable 3.1.
[55] Lee, J., 1997, "Design rationale systems: understanding the issues," IEEE Expert, **12**(3), p. 78–85.
[56] Burge, J. and Brown, D. C., 2000, "Reasoning with Design Rationale," *Artificial Intelligence in Design '00*, J. S. Gero, ed., Springer, Dordrecht, pp. 611–629.
[57] Leffingwell, D. and Widrig, D., 2000, *Managing software requirements: a unified approach*, Addison-Wesley Professional.
[58] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y., 2007, "Pellet: A practical OWL-DL reasoner," Journal of Web Semantics, **5**(2), p. 51–53.