




Automating Invoice Validation with Knowledge Graphs: Optimizations and Practical Lessons

Johannes Mäkelburg   
Technical University of Munich, Germany

Maribel Acosta   
Technical University of Munich, Germany

Abstract

To increase the efficiency of creating, distributing, and processing of invoices, invoicing is handled in the form of Electronic Data Interchange (EDI). With EDI, invoices are handled in a standardized electronic or digital format rather than on paper. While EDIFACT is widely used for electronic invoicing, there is no standardized approach for validating its content. In this work, we tackle the problem of automatically validating electronic invoices in the EDIFACT format by leveraging KG technologies. We build on a previously developed pipeline that transforms EDIFACT invoices into RDF know-

ledge graphs (KGs). The resulting graphs are validated using SHACL constraints defined in collaboration with domain experts. In this work, we improve the pipeline by enhancing the correctness of the invoice representation, reducing validation time, and introducing error prioritization through the use of the severity predicate in SHACL. These improvements make validation results easier to interpret and significantly reduce the manual effort required. Our evaluation confirms that the approach is correct, efficient, and practical for real-world use.

2012 ACM Subject Classification Information systems → Electronic data interchange; Information systems → Resource Description Framework (RDF); Information systems → Web Ontology Language (OWL); Information systems → Ontologies

Keywords and phrases Electronic Invoice, Ontology, EDIFACT, RDF, RML, SHACL

Digital Object Identifier 10.4230/TGDK.3.3.2

Category Use Case

Supplementary Material *Software (Source Code)*: <https://github.com/DE-TUM/EDIFACT-VAL> [18]
archived at [swh:1:dir:d820a10d861cfd9e361208220cf7aad3b03ef1a6](https://swh.io/1/dir:d820a10d861cfd9e361208220cf7aad3b03ef1a6)

Funding *Johannes Mäkelburg*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263.

Received 2025-06-05 **Accepted** 2025-10-24 **Published** 2025-12-10

Special Issue Use-Case Articles

1 Introduction

In the current business landscape, efficient financial and administrative practices are paramount for organizations. A key element in these operations is the invoicing process, which requires accuracy, timeliness, and standardization. To meet these demands, organizations are increasingly adopting Electronic Data Interchange (EDI), shifting from traditional paper-based invoices to structured electronic formats. Within EDI, EDIFACT is a widely adopted standard for representing electronic invoices, offering a uniform approach to their creation, distribution, and processing.

While EDIFACT has improved the efficiency of invoicing, a significant gap remains: the validation of EDIFACT invoices is not standardized. This affects business processes that depend on timely and accurate invoice data. One such case is the group purchasing organization Einkaufsbüro Deutscher Eisenhändler (E/D/E), which operates in 30 European countries with varying document regulations and serves as a key domain expert in the development and evaluation of our approach.



© Johannes Mäkelburg and Maribel Acosta;
licensed under Creative Commons License CC-BY 4.0

Transactions on Graph Data and Knowledge, Vol. 3, Issue 3, Article No. 2, pp. 2:1–2:24



Transactions on Graph Data and Knowledge

TGDK Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Although EDIFACT serves as the standard for handling invoices, the lack of a formal constraint language means that validation is often performed manually – a process that is both error-prone and labor-intensive. For example, consider an EDIFACT invoice where the invoice number is missing or the total amount does not match the sum of all line items. Such errors are hard to spot in the condensed EDIFACT syntax and can completely block further processing or lead to incorrect payments. With our KG-based approach, these inconsistencies are made explicit through SHACL validation: the system detects that a legally required identifier is absent or that the financial totals are inconsistent. This allows errors to be uncovered early and ensures that invoices meet both legal and business requirements before they enter downstream processes. To address this, we propose a knowledge graph (KG)-based solution for validating electronic invoices. The core of our approach is to model electronic invoices as RDF graphs. Shifting to the semantic web technologies allows for applying existing open standards and solutions for managing machine-readable data. At the core of the solution is the EDIFACT Ontology, which captures the semantics of EDIFACT message content in RDF. Second, we propose the tool EDIFACT-VAL to validate the content of the original EDIFACT messages. The tool processes the invoices in the EDIFACT format and translates them into XML. XML serves as an intermediate representation since the EDIFACT format is not compatible with existing KG tools. From the XML files, EDIFACT-VAL creates the RDF graphs using the EDIFACT Ontology and the RDF Mapping Language (RML) [4]. EDIFACT-VAL validates the invoice RDF graph using constraints defined in the Shapes Constraint Language (SHACL) [15]. The constraints are created with input from domain experts based on the EDIFACT guidelines.

This manuscript builds on our previous work EDIFACT-VAL v.1 [19] and introduces EDIFACT-VAL v.2, which includes the following novel contributions:

- A new invoice pre-processing approach that reduces integration effort and processing time.
- An error prioritization feature based on *sh:severity*, enabling users to systematically address the most critical validation issues first.
- Insights and lessons learned from real-world applications of EDIFACT-VAL.

We evaluate the extended approach using real-world EDIFACT invoices. The results demonstrate multiple improvements over the previous version of the EDIFACT-VAL. First, the generated RDF graphs offer a more accurate and semantically rich representation of invoice content, resulting in a more reliable and expert-validated foundation for downstream processing. Second, the runtime analysis shows that EDIFACT-VAL v.2 significantly outperforms EDIFACT-VAL v.1, primarily due to an optimized pre-processing and RDF mapping. The RDF graph generation phase alone achieves time savings of up to 66.56%, with the longest runtime observed being just 3.65 seconds. Third, the integration of error prioritization through *sh:severity* enables users to focus on the most critical validation issues, greatly simplifying the interpretation of validation results.

2 Preliminaries

First, we introduce the EDIFACT invoice concept, which forms the basis for the knowledge graphs. Then, we describe the purchase-to-pay ontology, which builds the basis for an EDIFACT ontology.

EDIFACT Invoice. EDIFACT is the most commonly used and most comprehensive international standard for electronic data interchange [13]. EDIFACT is used across almost all business sectors; the individual sectors are delimited in EDIFACT by so-called subsets. The maintenance of the standard is led by the United Nations and the Economic Commission for Europe.

Documents transmitted in EDIFACT are all types of messages of the business processes area. The structure of the messages is based on segments; these, in turn, consist of data elements and data element groups. These three components together are referred to as the EDIFACT elements.

■ **Listing 1** Excerpt of an EDIFACT invoice.

```

1 UNH+1+INVOIC:D:96A:UN:EAN008'
2 BGM+380+4031541+43'
3 DTM+137:20220908:102'
4 NAD+IV+4317784000000::9++Einkaufsbuero DeutscherEisenhaendler:GmbH+EDE Platz 1+Wuppertal++42389+DE'
5 LIN+1++4016671029277:EN::9'
6 PRI+AAA:16.78:::1:PCE'
7 MOA+79:100.68'
8 MOA+124:19.13'
9 UNT+37+1'

```

Listing 1 shows an excerpt of a real-world EDIFACT message. The segments are split into three sections: header-, detail- and summary section. In all three sections, some segments are required, meaning all three sections are always represented in an EDIFACT message. However, there are some segments within the sections that are optional. For example, the header section contains eight mandatory segments and six conditional segments. An example of a mandatory segment is the *NAD segment*, which provides information about one of the organizations involved in the invoice, including its role, name, unique identification number, and address. In contrast, the *PRI segment* is conditional. It specifies the unit price of an item, but this information is considered supplementary since the actual item price is already represented in the *MOA segment*. In the header, general information about the invoice is displayed, e.g., the invoice number (Line 2), the document date (Line 3), and information about the involved organizations (Line 4). Information about the sold items, including the net price (Line 6) or the article number (Line 5), is allocated in the detail section. The summary section contains the total amounts of the invoice, e.g., the total item amount (Line 7) or the total tax amount (Line 8). Above the header and below the summary section are segments allocated for the EDIFACT structure, e.g., the version and type of message (Line 1) or the end character (Line 9). The complexity of an EDIFACT invoice arises from several factors. A single message may include multiple invoices, each with its own header, detail, and summary. The number of segments varies depending on the level of detail. Invoices with many items or rich descriptions have far more segments. Finally, some segments bundle multiple pieces of information, such as the NAD segment (Line 4), which encodes role, name, address, and identifiers of an organization. These characteristics complicate validation, since it requires checking both the structural completeness of the message, such as the presence of all mandatory sections and segments, and the semantic consistency of the data, such as the correctness of totals or the proper assignment of organizational roles. In real-world settings, this structural variability directly impacts invoice handling, where issues have direct consequences: a missing identifier in the header may block legal acceptance, while mismatched totals can delay payment or introduce accounting errors.

P2P-O: Purchase to Pay Ontology. We use the Purchase-to-Pay Ontology (P2P-O) [27] as a foundation for modeling concepts from the EDIFACT standard. P2P-O is an OWL ontology that models semantic representation of invoices based on the *core invoice model* of the European Standard EN 16931-1:2017 [6]. P2P-O provides semantic classes for *item*, *price*, *documentline*, *organization*, *document*, *invoice* and *process*, making it a sound foundation for representing invoices.

3 Approach

Given an electronic invoice in the EDIFACT format, the problem tackled in this work is to validate the correctness of the invoice by representing the invoice as an RDF knowledge graph (KG). Our proposed solution comprises two parts: (1) an ontology to represent EDIFACT invoices (§ 3.1),

■ **Table 1** Competency Questions for the EDIFACT Ontology.

Name	Competency Question	Ontology Class
CQ 0	What invoices are all listed in an EDIFACT message?	<i>E-Invoice</i>
CQ 1	Which organizations are involved in the invoice?	<i>Formal Organization</i>
CQ 2.1	What role does organization <i>S</i> play in the invoice?	<i>AgentRole</i>
CQ 2.2	Which organization is the buyer in the invoice?	<i>AgentRole</i>
CQ 3.1	What information is displayed about the involved organizations?	<i>AgentRole</i>
CQ 3.2	What is the address of the buyer?	<i>AgentRole</i>
CQ 4	What items are sold in the invoice?	<i>Item</i>
CQ 5.1	What information is displayed about the items sold?	<i>Item</i>
CQ 5.2	What is the net price of the items sold in the invoice?	<i>Item</i>
CQ 6.1	What are the invoice details of the invoice?	<i>Invoice-Details</i>
CQ 6.2	What is the invoice amount of the invoice?	<i>Invoice-Details</i>
CQ 6.3	What is the invoice number?	<i>Invoice-Details</i>
CQ 7	What information must be provided so that the file format is valid?	<i>EDIFACT-Structure</i>
CQ 8	To which business process can the invoice be assigned?	<i>E-Invoice</i>

■ **Table 2** Overview of linked ontologies and vocabularies in the EDIFACT Ontology.

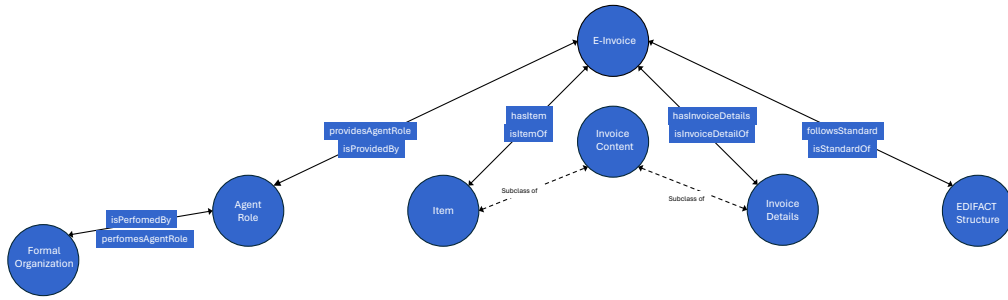
Prefix Name	Prefix	Frequency
agentRole	https://archive.org/services/purl/domain/modular_ontology_design_library/agentrole#	5
dc	http://purl.org/dc/elements/1.1#	2
frapo	http://purl.org/cerif/frapo/	3
schema	http://schema.org/	3
org	http://www.w3.org/ns/org#	1
p2p-o-doc-line	https://purl.org/p2p-o/documentline#	2
p2p-o-doc	https://purl.org/p2p-o/document#	1
p2p-o-inv	https://purl.org/p2p-o/invoice#	3
p2p-o-item	https://purl.org/p2p-o/item#	2
p2p-o-org	https://purl.org/p2p-o/organization#	4
vcard	http://www.w3.org/2006/vcard/ns#	2

based on the concepts presented in Sect. 2, and (2) the EDIFACT-VAL tool to perform the validation of invoices using KG technologies (§ 3.2). In this paper, we introduce a new version of EDIFACT-VAL, called EDIFACT-VAL v.2, which features an improved invoice pre-processing strategy and more tailored RML mappings. These enhancements increase the efficiency of the tool and enable a more specific validation of EDIFACT invoices.

3.1 EDIFACT Ontology

Development Process. The EDIFACT ontology was developed following best practices, like the NeOn method [31] or the formulation of ontology requirements and competency questions (Table 1). These steps were carried out in collaboration with Electronic Data Interchange experts from E/D/E. The development process is detailed in our previous work [19].

Reuse of Ontology Design Patterns and Existing Vocabularies and Ontologies. Following ontology design best practices [31, 23], we reuse existing resources to develop the EDIFACT-Ontology. An overview of the reused resources and the number of reuses can be found in Table 2. We apply the agent role pattern from the Modular Ontology Design Library (MODL) [28] for modelling the participation of organizations in invoices. This is necessary as the same organization



■ **Figure 1** Main concepts of the EDIFACT ontology [19]. Image generated with WebVOWL [17].

can have many roles (seller, supplier, etc.) in the same or different invoices, to which different property values can be associated depending on its role. The ontologies listed in Table 2 also provide adequate solutions for our purpose. Most of them are reused in the class *AgentRole*; other examples are the country code from the Funding, Research Administration, and Projects Ontology [29] or the address of the vCard Ontology [21]. Also, four of the seven different main classes we defined in our EDIFACT Ontology are linked to concepts of these vocabularies or ontologies. For example, the class *FormalOrganization* is linked to the Core Organization Ontology (*org*), the *E-Invoice* to the P2P-O module *document*, and *Item* to the P2P-O module *item*.

Ontology Description. The EDIFACT Ontology is tailored to represent the concepts and fields of the EDIFACT standard. The proposed OWL ontology comprises 31 classes, 22 OWL object properties, 261 OWL data properties, and 6 annotation properties. The ontology was developed using WebProtégé [12]. Figure 1 illustrates the main concept of the EDIFACT ontologies by showing the connections between the different classes. The EDIFACT Ontology can be found under <https://purl.org/edifact/ontology>. Additionally, it has been integrated into the LOV catalogue, available at <https://lov.linkeddata.es/dataset/lov/vocabs/edifact-o>. The classes and some of their most relevant properties for the invoice KGs are discussed in detail in the following. We focus on properties that are essential for representing the structural and semantic aspects of invoices, while omitting less relevant or supplemental properties to maintain clarity and conciseness. The relevant competency questions for each class are provided in Table 1.

- *E-Invoice* This is the main class of the EDIFACT ontology, and its individuals or entities represent electronic invoices. This class is connected to other classes through object properties, including *EDIFACT-Structure* via the property *followsStandard*, *Item* via the property *hasItem*, *InvoiceDetails* via the property *hasInvoiceDetails*, and *AgentRoles* via the property *isProvidedBy* to capture the role of organizations in the invoice. The only data property of this class is *belongsToProcess*, denoting the business process of the invoice.
- *EDIFACT-Structure* This class contains all the information that ensures that the invoice file meets the requirements of the EDIFACT file format. This information appears at the beginning and end of a message and, therefore, does not belong to the content of the individual invoices. As a result, this class and the class *InvoiceContent* are disjoint, specified with the *owl:disjointWith* predicate. However, as this information is part of an invoice, this class is connected to the class *E-Invoice* via the object property *followsStandard*. The data properties of this class include *creationDate*, *dataExchangeCounter*, *messageReferenceNumber*, and *senderIndicator*. Among the datatype properties of the entities of this class, we have *creationDate*, *dataExchangeCounter*, *messageReferenceNumber*, and *senderIndicator*.

- *Invoice Details* This class contains all the information that can only occur in the header- and summary sections of the invoices. Exemplary datatype properties of this class are the document date (*hasDocumentDate*), the document number (*hasDocumentNumber*), the delivery condition (*deliveryCondition*), and the invoice amount (*hasInvoiceAmount*). The connection of the *InvoiceDetails* class to the *E-Invoice* class is done via the object property *hasInvoiceDetails*.
- *Item* This class allows for representing an item listed in the invoice, which is found in the detail section of the invoice. Examples of the datatype properties are the name of the item (*p2p-o-item:Item*), the net price of the item (*hasNetPriceOfItem*), the net weight of an item (*hasNetWeight*), the number assigned to product according to the International Article Numbering Association (*internationalArticleNumber*). Examples of the datatype properties for an item are the name of the item (*p2p-o-item:Item*), the net price of the item (*hasNetPriceOfItem*), the net weight of an item (*hasNetWeight*), the number assigned to a manufacturer's product according to the International Article Numbering Association (*internationalArticleNumber*), etc. We model this relationship as a multi-valued property by defining the object property *hasItem* (and its inverse *isItemOf*) between the *Item* and the *E-Invoice* classes.
- *InvoiceContent* This class allows for modelling the information that can be found in all three sections of the invoices. As a result, the information of the *InvoiceContent* class is defined as the union of the classes *Item* and *InvoiceDetails*, i.e., *InvoiceContent owl:unionOf (Item InvoiceDetails)*. Creating the *InvoiceContent* class ensures the consistency of the ontology by defining properties that apply to all parts of the invoice.
- *AgentRole* According to the guidelines, an organization can, or sometimes must, have many different roles. For example, in the warehousing business, one organization needs to have three roles: *Buyer Role*, *Invoicee Role*, and *Delivery Party Role*. We model these cases using the Role ontology pattern as defined by Shimizu et. al [28] and Grüninger & Fox [8]. In our ontology, the purpose of the class *AgentRole* is to represent the manifold roles an organization can have in an invoice. As a solution, several subclasses have been created, each representing one of these roles. The connection to the *E-Invoice* class exists through the object property *isProvidedBy* with the *E-Invoice* class in the range.
- *FormalOrganization* This class represents the organizations involved in the messages, addressing CQ 1. The properties assigned to the class *FormalOrganization* serve two purposes: assigning the role an organization plays in an invoice through *performsAgentRole*, and providing a globally unique identifier for organizations, i.e., *globalLocationNumber*. In the EDIFACT ontology, the only connection of the class *FormalOrganization* to another class is the class *AgentRole* via the object property *performsAgentRole* with the *AgentRole* class in the range.

3.2 The EDIFACT-VAL Tool

Now that we have modelled the terms of the EDIFACT standard in an OWL ontology (see Sect. 3.1), the EDIFACT-VAL tool processes the electronic invoices in the EDIFACT format to validate their content using KG technologies. Concretely, the tool carries out the following steps (cf. Figure 2). **(1) Invoice Pre-processing:** Translates the invoice files into XML files. **(2) RML Mapping:** Creates RDF knowledge graphs from the invoice XML files using generated RML mappings. **(3) RDF Validation:** Validates the invoice RDF graphs using constraints based on EDIFACT guidelines and reports from domain experts, which are formulated using the Shapes Constraints Language (SHACL) [15].

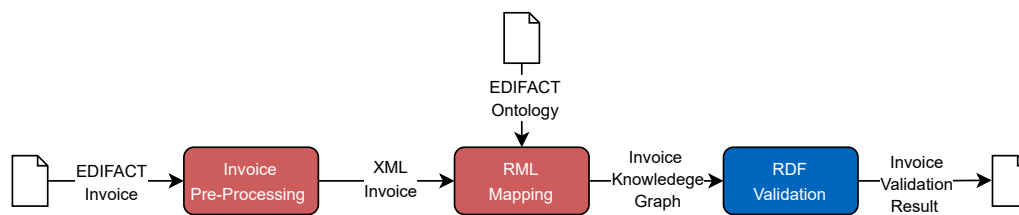
Compared to the old version from our previous work [19], called EDIFACT-VAL v.1, the new version, called EDIFACT-VAL v.2, features a different approach to Invoice Pre-Processing and RML Mapping. The new Invoice Pre-Processing now takes the original EDI messages as input and converts them into more tailored XML invoices. Based on these more refined XML invoices, the RML mapping can also be written in a more efficient and targeted way.

■ **Listing 2** Representation of organizations using *AgentRole* and *FormalOrganization*.

```

1 @prefix invoice: <http://www.ede.com/edifact/invoice#>.
2 @prefix edifact-o: <https://purl.org/edifact/ontology#> .
3 @prefix agentRole: <https://archive.org/services/purl/domain/modular_ontology_design_library/agentrole#> .
4 @prefix frapo: <http://purl.org/cerif/frapo/> .
5 @prefix p2p-o-org: <https://purl.org/p2p-o/organization#> .
6 @prefix vcard: <http://www.w3.org/2006/vcard/ns#> .
7
8 invoice:r999995 a edifact-o:DeliveryPartyRole;
9   frapo:hasCountryCode "DE";
10   edifact-o:hasCity "Wuppertal";
11   edifact-o:hasCountry "Deutschland";
12   vcard:hasStreetAddress "In der Fleute 153";
13   vcard:postalCode "42389";
14   agentRole:isProvidedBy invoice:i999999;
15   p2p-o-org:formalName "E/D/E-GmbH Anlieferstelle L205" .
16
17 invoice:r999996 a edifact-o:InvoiceeRole;
18   frapo:hasCountryCode "DE";
19   edifact-o:hasCity "Wuppertal";
20   edifact-o:hasCountry "Deutschland";
21   vcard:hasStreetAddress "EDE Platz 1";
22   vcard:postalCode "42389";
23   agentRole:isProvidedBy invoice:i999999;
24   p2p-o-org:formalName "Einkaufsuero Deutscher Eisenhaendler GbmH".
25
26 invoice:4317784000000 a edifact-o:FormalOrganization;
27   agentRole:performsAgentRole invoice:r999995, invoice:r999996 ;
28   p2p-o-org:globalLocationNumber 4317784000000 .

```



■ **Figure 2** Overview of EDIFACT-VAL. Components in red have been updated from v1 to v2.

3.2.1 Invoice Pre-Processing

This step aims to prepare the EDIFACT message in a way that meets the requirements for the creation of an RDF graph. Despite that EDIFACT is an open standard, the file format is not yet widely compatible with existing tools, especially the ones related to knowledge graphs. A direct transformation into RDF would require custom parsers for each message variant, whereas XML provides a standardized and flexible basis for subsequent mappings. Consequently, a transformation into a compatible format is required. For both EDIFACT-VAL v.1 [19] and the updated EDIFACT-VAL v.2, we adopted XML as the intermediary format because it combines high flexibility with robust support from a broad ecosystem of processing tools. The updated EDIFACT-VAL v.2 introduces three key improvements over EDIFACT-VAL v.1: (i): direct handling of original EDIFACT invoices (no pre-existing XML needed), (ii): targeted creation of only required XML elements and attributes, and (iii): a simplified saving process that avoids redundant file versions. To demonstrate the differences between the two approach, we examine the handling of the NAD-Segment, shown in Line 4 of Listing 1. There the NAD-segment is displayed with the qualifier *IV*, representing information about the Invoicee of the invoice.

■ Listing 3 Result of Pre-Processing with EDIFACT-VAL v.1.

```

1 <Interchange>
2   <Message>
3     <G_Group_2 id="8" mid="InvoiceDetail1">
4       <S_NAD agentRole="InvoiceeRole" details="Invoice1" id="9" mid="InvoiceDetail1" nid="Item0"
5         organisation="Role2" parent="GGroup2">
6         <D_3035 id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2" parent="NAD">IV</D_3035>
7         <C_C082 id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">
8           <D_3039 id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">+43177840000</D_3039>
9           <D_3055>9</D_3055>
10        </C_C082>
11        <C_C080 id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">
12          <D_3036 id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">+Einkaufsbuero
13            DeutscherEisenhaendler:GmbH</D_3036>
14          </C_C080>
15          <C_C059 id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">
16            <D_3042>EDE Platz 1</D_3042>
17          </C_C059>
18          <D_3164>Wuppertal</D_3164>
19          <D_3251>42389</D_3251>
20          <D_3207 id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">DE</D_3207>
21          <D_PaFu id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">Invoicee</D_PaFu>
22          <D_Country id="9" mid="InvoiceDetail1" nid="Item0" organisation="Role2">Deutschland</D_Country>
23        </S_NAD>
24      </G_Group_2>
25      <Invoicee Item="Item1" Kaeufer="Role1" Lieferanschrift="Role3" Lieferant="Role4" Invoicee="Role2"
26        details="Invoice1" id="1" mid="InvoiceDetail1">1234567000000</Invoicee>
27    </Message>
28 </Interchange>

```

■ Listing 4 Result of Pre-Processing with EDIFACT-VAL v.2.

```

1 <Interchange>
2   <Message>
3     <Party agentRole="InvoiceeRole" details="Invoice1" organisation="IV1">
4       <GLNNumber>+43177840000</GLNNumber>
5       <CodeListResponsibleAgency>9</CodeListResponsibleAgency>
6       <PartyName>Einkaufsbuero DeutscherEisenhaendler:GmbH</PartyName>
7       <Street>EDE Platz 1</Street>
8       <PostalCode>Wuppertal</PostalCode>
9       <CountrySubEntity>42389</CountrySubEntity>
10      <SecondPostalCode>DE</SecondPostalCode>
11    </Party>
12  </Message>
13 </Interchange>

```

(i) Direct Handling of EDIFACT Invoices. A limitation of EDIFACT-VAL v.1 is its requirement for a pre-existing XML representation of the EDIFACT invoice. In contrast, EDIFACT-VAL v.2 directly accepts original EDIFACT invoices, eliminating this dependency and expanding its applicability.

(ii) Targeted XML Creation. EDIFACT messages use qualifiers to secure file compactness by packing several values into a single field. To handle these qualifiers, EDIFACT-VAL v.1 separated such segments into several XML elements, creating a one-to-many correspondence between EDIFACT elements and XML elements. As a result, the size of the XML files increased significantly due to the additional elements and attributes needed to represent this structure.

The key improvement of EDIFACT-VAL v.2 lies in its more targeted creation of XML files: only required XML elements and attributes are generated, reducing unnecessary overhead for further processing. Here, required XML elements are defined as non-empty elements that contain data of interest for the RDF graph transformation process. This includes elements that hold essential information from the EDIFACT message (e.g., invoice details, line items) and elements that aggregate multiple data elements from the EDIFACT structure into a single XML representation.

This focused generation not only reduces the overall number of elements but also leads to fewer nested structures, which in turn minimizes the need for additional attributes to map content to the classes defined in the EDIFACT Ontology (see Section 3.1). Compared to EDIFACT-VAL V.1, we change the numbering of the attributes into more identifiable names like *Item* plus an ascending number for the attribute used to identify the item. This is also the case with organizations, where we also incorporate the role and the invoices inside the message. By focusing only on these required XML elements, EDIFACT-VAL V.2 minimizes overhead and ensures that the generated XML files contain only relevant data for the graph creation process.

(iii) Simplified Saving Process. Another notable improvement of EDIFACT-VAL V.2 is that the XML file now only needs to be saved once before reaching the final version. In contrast, EDIFACT-VAL V.1 required saving the XML file two additional times, introducing further complexity and redundancy. Listing 3 and Listing 4 illustrate the XML-representations of a NAD-Segment with the qualifier *IV* for EDIFACT-VAL V.1 and EDIFACT-VAL V.2, respectively. The comparison demonstrates a significant reduction (53.57%) of the XML elements with EDIFACT-VAL V.2.

3.2.2 RDF Graph Creation for Invoices

EDIFACT invoices can be transformed into RDF representations using our proposed EDIFACT Ontology described in Sect. 3.1. To generate RDF KGs based on a (semi-structured) data source, we define mapping rules. For this, we use the RDF Mapping Language (RML) [4], which provides a way to transform heterogeneous data into the RDF data model. Yet, manually creating RML mapping is a complex task [14], especially when a large number of terms are involved. To ease the definition of the mapping rules, languages like YARRRML [32] and ShExML [7] provide a human-friendly serialization of RML. In this work, we use YARRRML [32] as it allows users to specify simpler mapping rules compared to RML;¹ these mappings are automatically transformed into the RML mappings, later used to transform the invoice XML file into an RDF graph.

YARRRML-Mapping. Our YARRRML mapping contains six mappings, each representing one different class of the EDIFACT ontology. Compared to the EDIFACT ontology, only six of the seven classes are displayed, as the class *InvoiceContent* only has the purpose of providing a domain or range for some resources. When creating each mapping, two types of rules must be defined: (i) rules for defining the data sources, i.e., the XML files of the EDIFACT invoices, and (ii) rules for generating the RDF triples. A notable advantage of EDIFACT-VAL V.2 lies in its more targeted XML construction, as described in the pre-processing step 3.2.1. Due to the less nested structure of the tailored XML file, significantly fewer data source rules are needed. In EDIFACT-VAL V.1, a total of 245 different data source rules needed to be defined, as every different EDIFACT element had a different file path. In contrast, EDIFACT-VAL V.2 requires only four data sources, as the streamlined XML structure consolidates information more effectively. Overall, EDIFACT-VAL V.2 significantly reduces the complexity of the mapping process by consolidating data sources and streamlining XML structures. This improvement is also reflected in the line counts of the mapping files, with 1,690 lines in EDIFACT-VAL V.2 compared to 2,924 lines in the EDIFACT-VAL V.1. Depending on which information is to be displayed in the mappings, the data sources are assigned to the mappings. The rules for generating the RDF triples are divided into two parts. Part one defines the rule for identifying the subject of the RDF triple. In our tool, this is done by

¹ E.g., our YARRRML file has 1,690 lines (2,924 in EDIFACT-VAL V.1) versus 9,734 lines (366,709 in EDIFACT-VAL V.1) in the generated RML file.

■ **Listing 5** SHACL shape for checking the existence of mandatory data and syntactical correctness of EDIFACT invoice elements.

```

1 :InvoiceNumberShape a sh:NodeShape ;
2   sh:targetClass edifact-o:InvoiceDetails ;
3   sh:property [
4     sh:path edifact-o:hasDocumentNumber ;
5     sh:minCount 1 ;
6     sh:maxCount 1 ;
7     sh:datatype xsd:string ;
8     sh:maxLength 12 ;
9     sh:message "Invoice number must be a string with max 12 characters." ;
10    sh:severity sh:Violation ;
11  ] .

```

the added attributes from the preprocessing 3.2.1. This part of the mapping remains consistent across both EDIFACT-VAL V.2 and EDIFACT-VAL V.1. The second part defines the rules for the predicates and objects of the RDF triple. The predicate rule contains the ontology resources, and the object rule contains the identification of the value of the predicate. In the second part of our mapping, the primary difference between EDIFACT-VAL V.2 and EDIFACT-VAL V.1 lies in the updated XML element names resulting from the streamlined XML structure. However, the number of rules for the predicates and objects, as well as their association with the six different YARRRML mappings, remains unchanged.

RML-Mapping. The RML mappings obtained with YARRRML and the invoice XML files are then fed into the RMLMapper² tool, to obtain the invoice RDF graph. At this step, the impact of the streamlined XML structure introduced in EDIFACT-VAL V.2 's pre-processing step (Sect. 3.2.1) becomes evident: the generated RML file in EDIFACT-VAL V.2 has only 9,734 lines compared to 366,709 lines in EDIFACT-VAL V.1, meaning a 97.3% reduction in file size. This underscores how the simplified XML structure directly reduces the complexity and overhead of the transformation process. Exemplary transformation of the *NAD segment* from an EDIFACT message to an RDF graph can be seen in Line 4 in Listing 1 to Lines 17-24 and Lines 26-28 in Listing 2.

3.2.3 Invoice Validation using SHACL

Once the invoice RDF graph has been created, the next step is to validate the knowledge graphs. We use the Shapes Constraint Language (SHACL) [15], the W3C-recommended language, for the validation of RDF graphs. Constraints in SHACL can be specified over specific classes of the ontology using the *sh:targetClass* predicate, and over specific properties of the target class using the *sh:path* predicate. The SHACL validation process itself, as well as the shape graphs used, are the same for both EDIFACT-VAL V.1 and EDIFACT-VAL V.2. In EDIFACT-VAL V.2, we enhanced the SHACL shapes by including the use of the *sh:severity* predicate to classify constraint violations according to their impact on further invoice processing. This allows for a more differentiated constraint validation, distinguishing between errors that could hinder subsequent invoice handling and those that do not affect processing.

In our work, the shapes were developed based on input from domain experts, who also identified which constraints result in crucial violations and which do not. Errors that block further processing of an invoice, for example, when legally required information such as the invoice number or a monetary amount is missing, are classified as *sh:Violation*. By contrast, information that supports but is not essential for processing falls under *sh:Warning*; one case is the specification of quantities

² <https://github.com/RMLio/rmlmapper-java>

■ **Listing 6** SHACL shape for checking the existence of mandatory data and syntactical correctness of EDIFACT invoice elements.

```

1 :BuyerRoleCheckShape a sh:NodeShape ;
2   sh:targetClass edifact-o:E-Invoice ;
3   sh:sparql [
4     a sh:SPARQLConstraint ;
5     sh:message "Each invoice must have exactly one BuyerRole,
6       and the BuyerRole must be linked to a FormalOrganization." ;
7     sh:severity sh:Violation ;
8     sh:prefixes [
9       sh:declare [
10        sh:prefix "edifact-o" ;
11        sh:namespace "https://purl.org/edifact/ontology#"^^xsd:anyURI ;
12      ] ;
13      sh:declare [
14        sh:prefix "agentRole" ;
15        sh:namespace "https://archive.org/services/purl/domain/
16          modular_ontology_design_library/agentrole#"^^xsd:anyURI ;
17      ] ;
18      sh:declare [
19        sh:prefix "p2p-o-inv" ;
20        sh:namespace "https://purl.org/p2p-o/invoice#"^^xsd:anyURI ;
21      ]
22    ] ;
23   sh:select """
24     SELECT $this
25     WHERE {
26       {
27         SELECT $this (COUNT(?buyerRole) AS ?buyerCount)
28         WHERE {
29           $this p2p-o-inv:hasBuyer ?buyerRole .
30         }
31         GROUP BY $this
32         HAVING (?buyerCount != 1)
33       }
34       UNION
35       {
36         $this p2p-o-inv:hasBuyer ?buyerRole .
37         FILTER NOT EXISTS { ?buyerRole a <http://example.com/BuyerRole> . }
38       }
39       UNION
40       {
41         $this p2p-o-inv:hasBuyer ?buyerRole .
42         FILTER NOT EXISTS {
43           ?org a edifact-o:FormalOrganization ;
44             agentRole:performsAgentRole ?buyerRole .
45         }
46       }
47     } """ ;
48 ] .

```

and units of delivered goods. Lastly, details intended purely for internal purposes, such as assigning the invoice to a business process, are considered *sh:Info*, since they do not affect the factual correctness of the invoice.

We distinguish three types of shapes, each with a specific purpose:

(i) Existence of mandatory data. Shapes ensure that the required properties are present in the RDF graph according to the EDIFACT invoice guidelines. Listing 5 (Lines 5 and 6) shows a SHACL constraint for specifying that the *documentNumber* property is mandatory (*sh:minCount* 1) and single valued (*sh:maxCount* 1) for entities of the class *InvoiceDetails*.

(ii) Syntactical correctness of data. Shapes validate the format of the EDIFACT elements, including length, number, and permitted characters. Listing 5 (Lines 7 and 8) shows a SHACL constraint for checking the datatype (*sh:datatype*) and the length (*sh:maxLength*) for the *documentNumber* property of the target class *InvoiceDetails*.

Together, (i) and (ii) ensure the completeness and syntactical correctness of the document number, directly answering Competency Question 6.3 (see Table 1).

(iii) Logical correctness of data. Shapes ensure that the data in the RDF graph is logically consistent and reflects the business logic of an invoice. For example, this includes ensuring that each invoice has exactly one buyer organization linked to it, verifying the logical correctness of the data structure needed to answer Competency Question 2.2 (see Table 1). Such constraints, which involve verifying the existence of a single buyer and ensuring it is properly linked to an organization, can be expressed in SHACL using SPARQL queries, as shown in Listing 6.

4 Evaluation

In our previous work [19], we evaluated the completeness of invoice KG generation and validation with EDIFACT-VAL V.1, including a built-in mechanism to display unknown elements and tests with corrupted invoices. Building on this evaluation, this paper empirically evaluates and compares our approach,

assessing the differences between EDIFACT-VAL V.2 and EDIFACT-VAL V.1, in terms of soundness, performance, and applicability. Concretely, we focus on the following core questions:

- Q1** Are RDF graphs created by EDIFACT-VAL V.1 and EDIFACT-VAL V.2 identical? (§4.2)
- Q2** How does EDIFACT-VAL V.2 influence the runtime performance of EDIFACT invoice processing compared to EDIFACT-VAL V.1? (§4.3)
- Q3** How does EDIFACT-VAL V.2 influence the computational performance of EDIFACT invoice processing compared to EDIFACT-VAL V.1? (§4.4)
- Q4** How does EDIFACT-VAL V.2 influence error analysis by introducing error prioritization? (§4.5)
- Q5** How does EDIFACT-VAL V.2 affect the applicability to real-world business processes? (§4.6)

4.1 Experimental Set Up

Tool Implementation. EDIFACT-VAL is implemented in Python 3 and available online.³ For SHACL validation, we use pySHACL [30], an open-source Python library. EDIFACT-VAL is equipped with 394 SHACL constraints provided by the experts following the EDIFACT standard. All experiments have been conducted on a machine with an AMD EPYC 9224 24-Core CPU, 578 GB of RAM, and running Ubuntu 24.04.1 LTS.

Dataset. We use 44 different real-world EDIFACT messages from 12 different suppliers and 6 different business cases to evaluate the performance of the EDIFACT-VAL tool. The selection of messages has been made together with the EDI experts from E/D/E to have a range of messages representing the day-to-day business workflow. Since each supplier may include different segments and data elements in the invoices, the selected messages represent a wide range of segments and segment combinations. All 394 SHACL constraints matched at least one triple across the 44 EDIFACT messages used in the evaluation, while also all resulted in at least one validation error. This corresponds to a relevance coverage of 100% and a violation coverage of 100%. These constraints, however, do not cover all data represented in the invoice knowledge graphs. Rather, they focus on the most critical aspects of invoice validation for legal compliance with e-invoice regulations, like mandatory standardized organizational data and required financial calculations.

³ <https://github.com/DE-TUM/EDIFACT-VAL>

■ **Table 3** Comparison of RDF graph statistics for each message generated by EDIFACT-VAL v.1 and EDIFACT-VAL v.2. Message names are omitted for privacy.

Message	#Triples		#Nodes		#Properties	
	Old	New	Old	New	Old	New
Message 1	96	96	12	12	67	67
Message 2	401	401	27	27	71	71
Message 3	982	989	70	70	78	79
Message 4	130	132	13	13	68	70
Message 5	143	143	12	12	81	81
Message 6	393	398	31	31	63	65
Message 7	36,170	36,165	2,091	2,091	79	79
Message 8	113	113	11	11	66	67
Message 9	122	122	11	11	76	76
Message 10	111	111	11	11	70	71
Message 11	290	282	20	20	75	76
Message 12	3,152	3,135	209	209	91	91
Message 13	921	920	56	56	73	74
Message 14	440	440	23	23	81	82
Message 15	4,634	4,629	307	307	80	80
Message 16	531	532	29	29	72	74

4.2 Equivalence of the Invoice Generated by EDIFACT-VAL v.1 and v.2

Since the completeness and correctness of EDIFACT-VAL v.1 have already been established in [19], it is sufficient to compare the RDF graphs generated by EDIFACT-VAL v.2 to those of EDIFACT-VAL v.1 using a graph isomorphism check with `rdflib` [16].

First, we loaded the RDF graphs produced by both approaches and compared them directly by checking for graph isomorphism. *Rdflib*'s built-in isomorphism function showed that there were some triples present only in the EDIFACT-VAL v.2 and some only in the EDIFACT-VAL v.1. To identify these triples, we compared key statistics of the graphs, including the number of triples, nodes, and distinct properties, as summarized in Table 3. When comparing the key statistics of different EDIFACT messages, we observed notable variation in the number of generated triples. As discussed in Section 2, this variation arises from the complexity of the messages, which is influenced by the number of contained invoices, the length and detail of their segment structures, and the types of segments used.

After discussions with domain experts to investigate why certain triples were missing in one of the versions, we identified the following reasons: The reasons were split into those explaining why elements appear in the RDF graphs of EDIFACT-VAL v.1 and not in EDIFACT-VAL v.2, and vice versa. We begin by discussing why some elements are present in EDIFACT-VAL v.1 but missing in EDIFACT-VAL v.2.

(i) Reduced Triple Count Due to EDIFACT Syntax Errors. One issue arises from the syntax of EDIFACT messages, where certain characters, such as `+` and `'`, are used to indicate the start of the next element or segment. In some of the evaluated messages, these characters were also used as content within data elements, leading to incorrect splitting of the data elements. In EDIFACT-VAL v.1, the input was an XML version of the invoice, and the in-house translation by E/D/E was able to handle these errors. However, because the use of these symbols as content constitutes an error and should be communicated to the customers, we decided to treat them as errors. This issue frequently occurred in the NAD-Segment, which describes the involved organizations, as some companies include the character `+` in their name. As a result, the syntax of the following data elements, such as postal code or country code, was often missing. This led to a reduction in the number of properties in the RDF graph created for Message 12 by EDIFACT-VAL v.2, as summarized in Table 3.

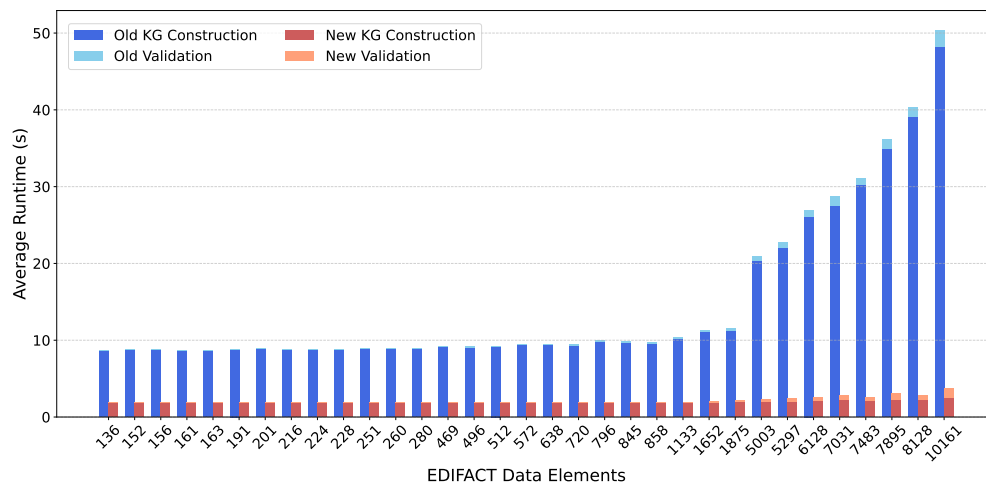
(ii) Reduced Triple Count from Filtering Empty Data Elements. As outlined in Section 3.2.1, EDIFACT-VAL V.2 omits data elements that have no content in the RDF graph. In contrast, EDIFACT-VAL V.1 did not apply this filtering, resulting in the presence of triples with empty objects ("") that are no longer generated in the RDF graph by EDIFACT-VAL V.2. Consequently, this filtering step also reduces the number of properties represented in the RDF graphs created by EDIFACT-VAL V.2, as summarized in Table 3. It is important to note that the magnitude of this reduction varies between messages, since the messages are sourced from different suppliers with differing invoice practices. For instance, some suppliers – such as the supplier of Message 1 – do not include empty data elements, while others – like the supplier of Message 15 – do include them.

(iii) Increased Triple and Property Counts from Improved Segment Allocation. The simpler creation of the XML file in EDIFACT-VAL V.2 has led to a reduction in the number of nested XML elements. As described in Section 3.2.1, this simplification has reduced the number of attributes needed during the pre-processing step to correctly identify and allocate the segments. This is particularly relevant for segments that can occur in all three parts of the message, such as the *RFF-Segment* and the *DTM-Segment*. In EDIFACT-VAL V.1, it was often difficult to correctly allocate these segments because of the complex nested XML structure. Since certain specific combinations of these segments (for example, the valuta date represented by *DTM+209*) only occur in particular parts of the message, precise allocation is crucial and determines whether they are represented in the RDF graph. This improvement in the XML creation process directly affects the completeness of the RDF graph and explains why, in some cases, EDIFACT-VAL V.2 generates more triples and properties than EDIFACT-VAL V.1. This is particularly evident in the results for Messages 3 and 4 in Table 3, where Message 3 has one additional property and Message 4 has two additional properties in the new version of the RDF graph, and therefore also more triples. These findings highlight that the simpler XML creation in EDIFACT-VAL V.2 not only reduces unnecessary data but also ensures that critical segments are correctly represented, contributing to a more complete and accurate RDF graph.

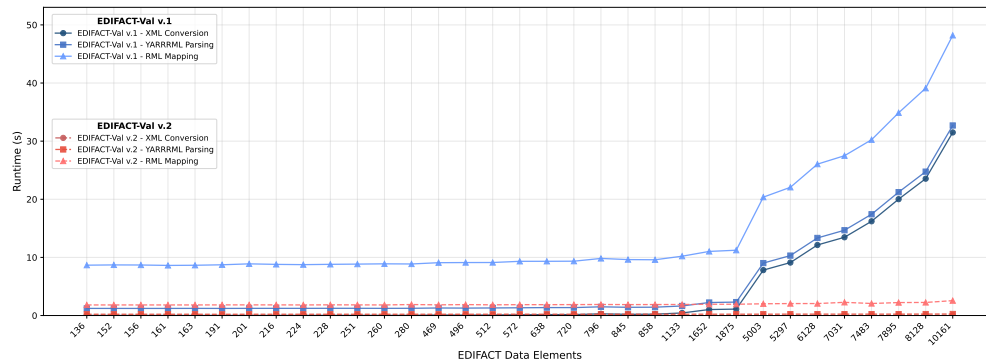
Since the number of triples differs between the RDF graphs generated by EDIFACT-VAL V.1 and EDIFACT-VAL V.2, we cannot claim that the RDF graphs are identical. However, our investigations demonstrate that the RDF graph generated by EDIFACT-VAL V.2 offers an improvement in data representation. Importantly, the number of nodes (entities) remains the same in both graphs, ensuring that all concepts and entities of the invoices are still displayed. At the same time, EDIFACT-VAL V.2 allows us to: (i) omit empty data elements, (ii) identify and exclude errors that were previously undetectable, and (iii) represent information that was previously not included due to the nested structure of the input data. These improvements collectively contribute to a more reliable, accurate, and domain-expert-validated foundation for further processing and analysis of EDIFACT invoices.

4.3 Runtime Performance of EDIFACT-Val

We compare the efficiency of the different versions of EDIFACT-VAL when processing the invoices, i.e., the elapsed time between the tool receiving an EDIFACT message (KG Construction) and producing the validation result (KG Validation). This time includes the generation of the RDF graph and its evaluation using the SHACL constraints. To ensure representative results, we selected one EDIFACT message per supplier, which may contain several invoices with varying numbers of EDIFACT data items. We ran EDIFACT-VAL 10 times on each EDIFACT message using the `hyperfine` [24] command-line benchmarking tool. Figure 3 shows the average runtime per message for both EDIFACT-VAL V.1 (in blue) and EDIFACT-VAL V.2 (in red), with a darker tone representing KG Construction and a lighter tone representing KG Validation. Since the difference



■ **Figure 3** Runtime Comparison: EDIFACT-VAL V.1 versus EDIFACT-VAL V.2.

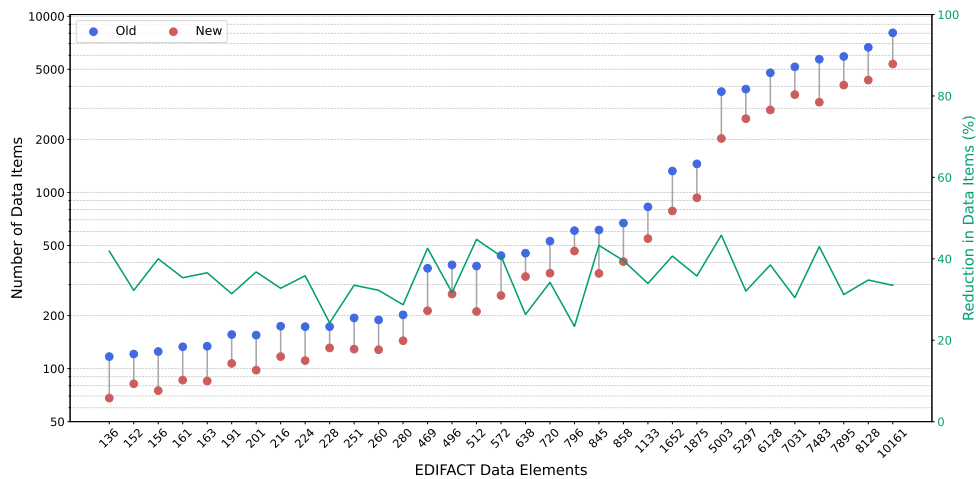


■ **Figure 4** KG Construction Breakdown for EDIFACT-VAL.

between EDIFACT-VAL V.1 and EDIFACT-VAL V.2 lies only in the KG Construction phase, we split the analysis into three parts: (1) general observations, (2) KG construction, and (3) KG validation. This enables us to isolate the performance improvements achieved by EDIFACT-VAL V.2 in the RDF graph generation step.

(1) General Observations. The runtime comparison shown in Figure 3 reveals a significant decrease in the total runtime of EDIFACT-VAL V.2 compared to EDIFACT-VAL V.1. For smaller EDIFACT messages, the runtime with EDIFACT-VAL V.1 was around 8 seconds and is now reduced to about 2 seconds with EDIFACT-VAL V.2. For larger EDIFACT messages, the runtime drops substantially, from 50 seconds with EDIFACT-VAL V.1 to under 4 seconds with EDIFACT-VAL V.2. These observations show that the improvement is consistent across all tested messages and demonstrate that EDIFACT-VAL V.2 effectively handles increasing input sizes. A closer examination of each runtime's composition (Figure 3) shows that this improvement results from the KG construction phase, which is the only part where EDIFACT-VAL V.1 and EDIFACT-VAL V.2 differ.

(2) KG Construction. As noted before, the time spent on KG Construction is where EDIFACT-VAL V.2 is significantly more efficient than EDIFACT-VAL V.1. Looking only at the times for KG Construction, the average percentage of time saved with EDIFACT-VAL V.2 is 82.10%. There are several reasons for this improvement: First, with the tailored pre-processing described in



■ **Figure 5** Number of data items and percent Reduction: EDIFACT-VAL v.1 versus EDIFACT-VAL v.2.

Section 3.2.1, the number of created XML elements is reduced by removing unnecessary elements and avoiding nested structures that do not contribute to the final RDF graph. Figure 5 provides an overview of the number of XML elements generated by EDIFACT-VAL v.2 and EDIFACT-VAL v.1 for each EDIFACT message. This figure shows that, on average, the number of XML elements is reduced by 35.39% across all EDIFACT messages. This reduction means that there are fewer XML elements that need to be checked and processed by the RDF mappings in the following step. As a result, the RDF mappings can be executed more quickly and efficiently, which directly speeds up the KG construction process, as shown in Figure 4. A second improvement of the pre-processing that significantly speeds up the KG construction time is that the created XML files no longer need to be parsed and saved multiple times. In EDIFACT-VAL v.1, the XML files were repeatedly parsed, modified, and saved again, leading to redundant file handling operations. By contrast, EDIFACT-VAL v.2 streamlines this process by parsing, modifying, and saving the XML content in a single step within the pre-processing phase. This eliminates unnecessary operations, reduces memory usage, and improves the overall computational efficiency. As a result, the KG construction process becomes more streamlined and significantly faster, contributing to the overall runtime improvement observed in Figure 4. Third, Section 3.2.2 describes the reduction of both YARRRML-Mapping and RML-Mapping in EDIFACT-VAL v.2. The YARRRML-Mapping was reduced by 42.2% due to fewer needed data sources, which led to a reduction of the RML-Mapping by 97.3%. This substantial reduction directly impacts the KG construction runtime by decreasing the parsing and processing overhead for the RML Mapper, simplifying the graph construction phase, and reducing the number of triple patterns to be executed. Overall, the leaner and more direct mapping instructions in EDIFACT-VAL v.2 make the RDF graph generation faster and more efficient, as reflected in the runtime comparison shown in Figure 3.

(3) KG Validation. Since the RDF graphs created by EDIFACT-VAL v.1 and EDIFACT-VAL v.2 are almost identical, as discussed in Section 4.2, the number of RDF triples is also nearly the same (see Table 3). Consequently, the runtime of the RDF validation with SHACL is also similar in both approaches. Across all tested messages, the validation time ranges from as little as 0.05 seconds for smaller EDIFACT messages to a maximum of 1.2 seconds for the largest EDIFACT messages. On average, the runtime reduction in the KG validation phase is 10.8%, which can be attributed to the slight reduction in triples described in Section 4.2.

Overall, the analysis of the runtime shows that EDIFACT-VAL v.2 is significantly more efficient than EDIFACT-VAL v.1, thanks to its optimized pre-processing and RDF mapping steps. The runtime per EDIFACT element is reduced from 28.53 milliseconds to 9.54 milliseconds, resulting in a normalized runtime reduction of 66.56%. This substantial improvement stems from the more efficient KG construction phase in EDIFACT-VAL v.2, which eliminates redundant processing steps and accelerates the generation of the RDF graph.

4.4 Computational Performance of EDIFACT-Val

Since we use a large-scale machine to conduct the runtime experiments, AMD EPYC 9224 24-Core CPU, 578 GB of RAM, we monitored the CPU and memory usage during the experiments with both EDIFACT-VAL v.1 and EDIFACT-VAL v.2. The results of CPU usage (Figures 6 and 7) and memory usage (Figures 8 and 9) provide insights into the efficiency of both approaches. The average CPU usage is calculated as the mean of all monitored CPU utilization across all cores, while the peak CPU usage corresponds to the maximum recorded value. The same definition applies to the memory usage metric.

The CPU measurements show that EDIFACT-VAL v.1 has a lower average overall usage of about 4.1% compared to 6.5% for EDIFACT-VAL v.2. However, both approaches utilize a small fraction of the available 24-core CPU, with most cores remaining inactive during validation. The peak overall usage is similar at around 12.6%, but the distribution across cores differs: EDIFACT-VAL v.1 occasionally drives individual cores up to 72%, while EDIFACT-VAL v.2 remains below 50% per core, with an average maximum core utilization of 25.32% for EDIFACT-VAL v.1 and 16.17% for EDIFACT-VAL v.2. These results indicate that in both approaches, the workload is unevenly distributed across cores, with a few cores handling short bursts of higher load while most remain underutilized. Detailed per-core utilization values are available in the project's GitHub repository. An unexpected observation in Figure 6 is that the average CPU utilization is lower for the larger EDIFACT message. This occurs because a larger portion of the processing time is spent in components of the approach that are less CPU-intensive, leading to a lower overall average utilization. This behavior is shown in Figure 7. Consequently, despite the slightly higher average load, EDIFACT-VAL v.2 makes more efficient use of the available hardware resources, as it completes execution significantly faster. Although it utilizes the CPU more intensively at a given moment, it holds the CPU for a much shorter time, resulting in lower total CPU time consumption and thus higher overall efficiency.

The results in Figure 8 show a clear difference in memory consumption between the two approaches. The old approach requires on average between 1.2–1.6 GB, with peak usage rising as the number of EDIFACT data elements increases. In contrast, the new approach consistently stays close to 1.0–1.1 GB for both average and peak usage, showing only minor variations even for large messages. On average, this corresponds to roughly a 30% lower memory footprint compared to the old approach. This means that while EDIFACT-VAL v.1 shows a growing memory demand with message size, EDIFACT-VAL v.2 maintains a stable and minimal footprint regardless of input size. When comparing the memory usage over time for one EDIFACT message (Figure 9), it becomes evident that both approaches show relatively stable memory usage during the initial stages, followed by a clear increase towards the end of processing. The main peak in both plots occurs near the completion phase, where the message validation and result aggregation are performed.

While EDIFACT-VAL v.1 exhibits stronger fluctuations and multiple smaller peaks throughout the execution, EDIFACT-VAL v.2 maintains a smoother progression and a lower overall peak, indicating more consistent and efficient memory utilization. Combined with its shorter runtime, this stability makes EDIFACT-VAL v.2 more predictable and better suited for large-scale or resource-constrained deployments.

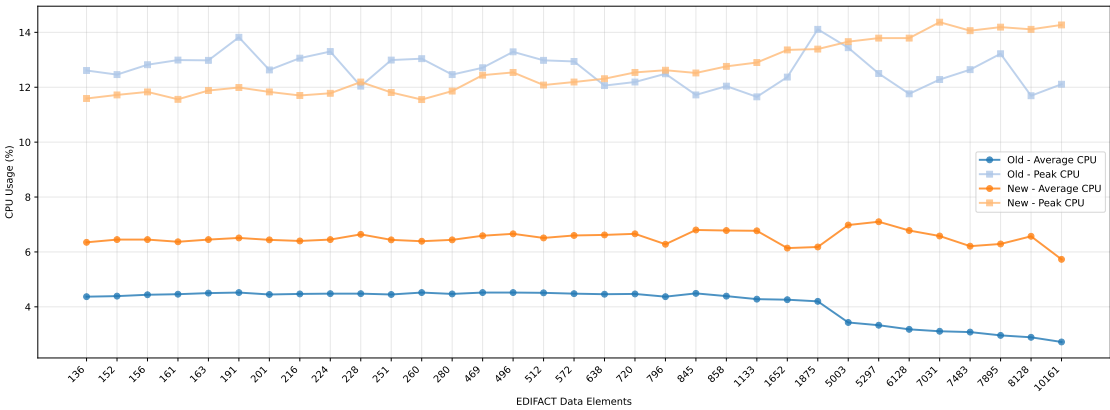


Figure 6 CPU usage comparison for EDIFACT-VAL. Values show the mean utilization across all CPU cores during processing.

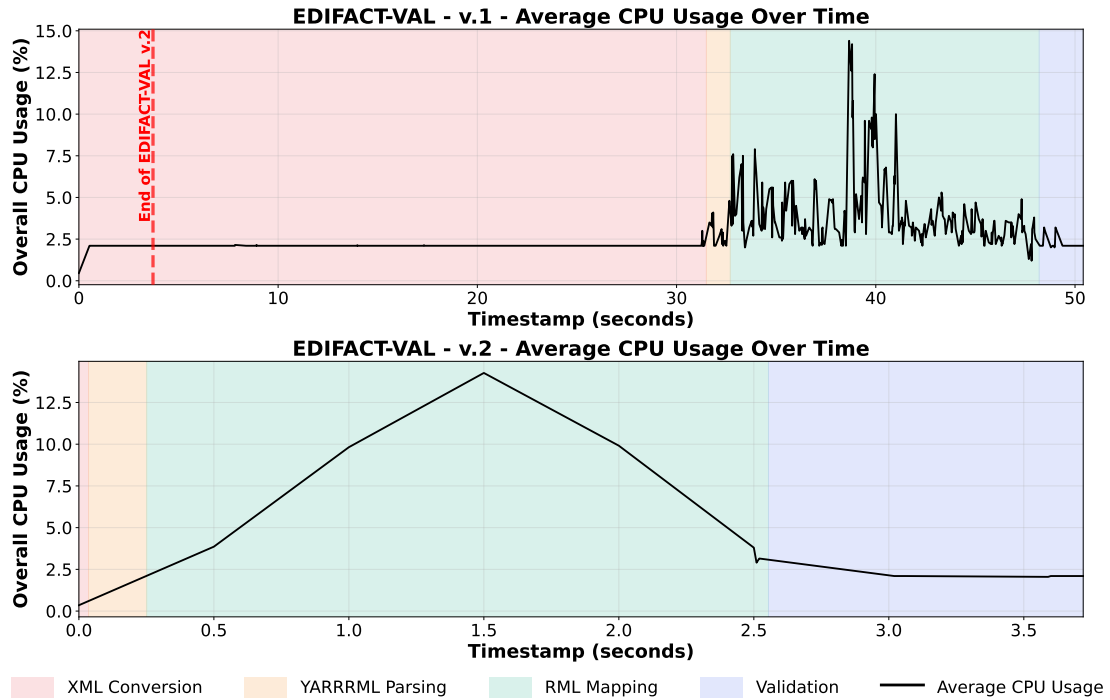


Figure 7 CPU usage timeline for message 10161.

Overall, the results show that while EDIFACT-VAL v.1 consumes slightly less CPU on average, its longer runtimes and higher memory demand make it far less efficient, whereas EDIFACT-VAL v.2 combines faster execution with lower memory usage, offering a more scalable solution.

4.5 Error Prioritization Evaluation

In our previous work [19], we discussed that not all errors have the same severity. Therefore, classifying the different constraints in SHACL using the *sh:severity* predicate became essential. Section 3.2.3 motivates and describes how this classification was implemented. Incorporating

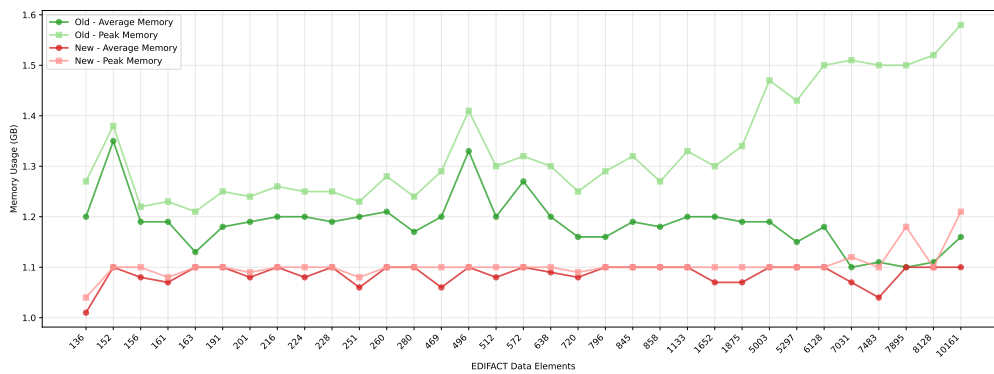


Figure 8 Memory usage comparison for EDIFACT-VAL.

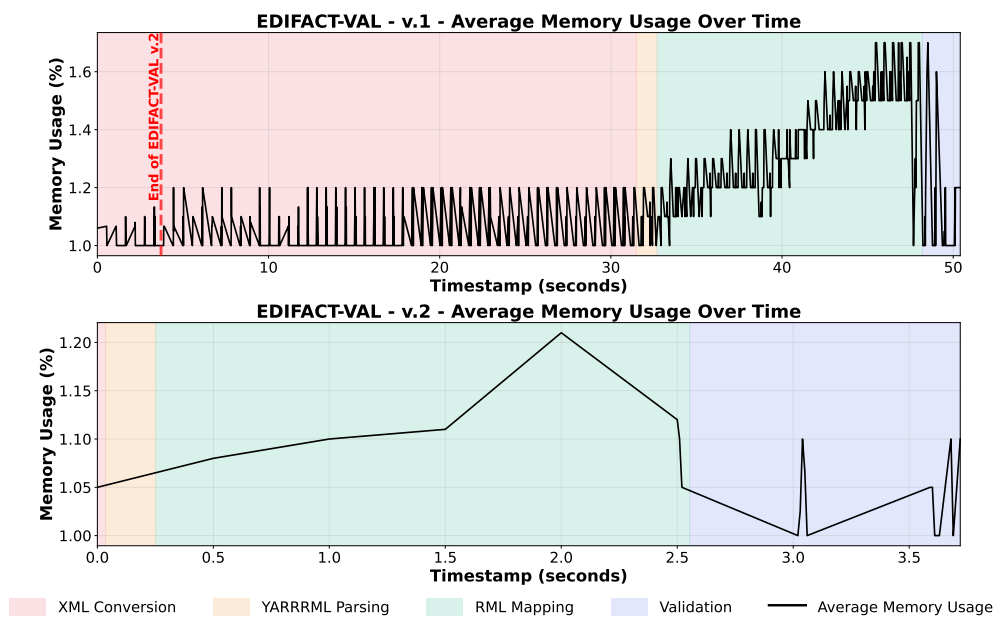


Figure 9 Memory usage timeline for message 10161.

sh:severity simplifies the analysis of validation reports by prefiltering the errors and highlighting the errors that require the most attention. Table 4 presents the results for an exemplary Business Process from [19]. This refinement led to a 100% reduction in the number of violations categorized as *sh:Violation*. Previously, these messages were incorrectly classified as violations, even though they did not affect further invoice processing. This made it difficult for the experts to identify the errors on which they had to work right away, as all types of errors were handled with the same priority. The refined classification makes it significantly easier for domain experts to identify and address truly impactful issues. As the results from EDIFACT-VAL v.2 now show that the errors in the messages are not crucial for the processing of the invoices, the experts can now work on the messages with errors that are crucial for the processing, and work on other issues after resolving the urgent errors. Importantly, prioritizing constraints in the SHACL shapes did not affect the runtime of the validation process.

■ **Table 4** Comparison of RDF graph statistics with and without error prioritization.

	EDIFACT-VAL V.1	EDIFACT-VAL V.2		
Business Process	#Violation	#Violation	#Warning	#Info
BP1	21	15	4	2

Comparison of *sh:severity* options with and without prioritization for an exemplary Business Process.

	EDIFACT-VAL V.1	EDIFACT-VAL V.2		
Message	#Violation	#Violation	#Warning	#Info
Message 1	33	0	33	0
Message 2	6	0	6	0
Message 3	3	0	3	0
Message 4	63	0	63	0

Statistics for each message, showing *sh:severity* options with and without prioritization.

4.6 In-Use Evaluation

Since the development was motivated by the group purchasing organization E/D/E, we also evaluated the applicability of our proposed solution approach to the E/D/E invoice processes. We asked the domain experts to validate the EDIFACT messages following the usual procedure, which is done manually. On average, it takes the experts around 30 minutes to validate one EDIFACT message. In comparison to EDIFACT-VAL V.2 (cf. Figure 3), the longest runtime is in the order of 3.65 seconds, which results in a time saving of around 99.98%.

Beyond runtime, we also assessed applicability with E/D/E domain experts. The constraints and validation reports have already been integrated into their internal testing workflows, where they are used to check newly received invoices before processing. While the tool is not yet deployed as a full production service, E/D/E experts confirmed that the validation results align with their manual procedures and that the prioritization mechanism helps them focus on legally critical errors. This pilot use indicates that the approach is practical and paves the way for continued organizational adoption.

5 Related Work

Ontologies for Electronic Invoices. Schulze et al. [27] propose an ontology, the Purchase-To-Pay Ontology (P2P-O), to represent electronic invoices. P2P-O relies on the European Standard EN 16931-1:2017, intending to provide ontology resources for all mandatory information in a purchase-to-pay process, which includes the invoicing process. In contrast, we aim not only to provide mandatory information for processing an electronic invoice, we also aim to validate the completeness and syntactical correctness of the invoices regarding their specific format. Furthermore, the terms used in the EDIFACT messages are not captured in the P2P-O ontology. This is why the creation of a standard-specific ontology, the EDIFACT ontology, is crucial in this case. In the literature, we also found several ontologies and vocabularies that provide invoice-related definitions. The ones we reuse can be found in Table 2. However, they only present small parts of invoices. For example, schema.org [9] provides terms for the amount of money and currencies. Yet, these vocabularies are not specific enough to model all the EDIFACT terms.

Invoice Validation. Several works have tackled the problem of invoice validation from different perspectives. Emmanuel and Thakur [5] present an approach that implements an EDI invoice validation framework; this work focuses on checking the completeness of invoices (defined as

the number of fields) by comparing actual values to expected values as given in an XML file. The expected value is defined in a rule set. Similarly, our work validates invoices but uses KG technologies and more expressive SHACL constraints formulated by experts and the EDIFACT standard. Other approaches also perform invoice validation but not in the context of EDI. For example, Sál [25] presents a tool to check whether the invoice fields are provided correctly to a digital system w.r.t. to the original invoice in PDF. Other solutions [1, 3, 10] propose processing and classifying invoices in PDF using Machine Learning (ML) models. Similarly, a comprehensive survey by Saout et al. [26] reviews a wide range of extraction and structure recognition techniques for invoice processing, including NLP and graph-based methods. Another approach by Hamri et al. [11] explores information extraction from invoices by studying document anatomy and training models on synthetic data to handle template variants. Additionally, works by Bisetty et al. [2] and Milosheski et al. [22] explore how ERP systems and automated invoicing systems improve operational efficiency through workflow-level automation. All these works check for the correctness of the translation of the invoices into machine-readable formats, rather than the correctness of the original invoices. These approaches greatly differ from EDIFACT-VAL, as it processes invoices that are already in a machine-readable format and validates the correctness of the original EDIFACT invoice. Lastly, the work by Schulze et al. [27] uses SPARQL queries to perform analytical tasks on the invoices. Examples of these include finding items that are sold in large cities. In contrast, EDIFACT-VAL is tailored to check the conformance of the invoices to the EDIFACT standard and other constraints defined by experts.

6 Lessons Learned

The development and application of EDIFACT-VAL in real-world scenarios provided us valuable insights into the use of Knowledge Graph technologies for invoice validation. In particular, we found that a carefully designed Shapes Graph significantly improves usability, especially when dealing with large and complex data graphs. In the following, we elaborate on these aspects.

1. Importance of a Business Graph for Cross-Checking. A lesson learned from our work is the importance of creating a dedicated Business Graph that integrates all included companies and the items sold. This takeaway emerged because we observed inconsistencies in supplier and product information across different invoices, raising concerns about the overall correctness of the data. Without a ground truth for these entities, one can only check the logical correctness within a single invoice, but this local correctness does not necessarily guarantee global correctness across all invoices. By linking supplier data and product details within the Business Graph, discrepancies and missing links can be automatically identified, which significantly reduces manual validation efforts. Furthermore, this integrated view supports robust data quality assurance, as it provides domain experts with a reliable source of truth for verifying the correctness of invoice data. Developing and evaluating such a comprehensive Business Graph will be a focus of our future work.

2. Lessons for Effective Shape Design. When using SHACL to formulate Invoice Guidelines as constraints, we identified several key takeaways. We present these insights starting from the smallest instance (Shapes) and moving up to the largest instance (Shape Graphs).

(i) Naming Conventions for Shape Graphs A prerequisite for meaningful validation results is the ability to trace each violation back to the shape that caused it. To achieve this, SHACL shapes should follow a consistent naming convention. Without such conventions, the system automatically generates blank nodes, which prevents cross-shape querying and limits the

usefulness of the validation report. Establishing a naming convention also increases reusability, since many shapes can be applied across slightly different business processes with minimal adjustments. In the invoicing domain, where processes often differ only in detail, this reusability is particularly valuable. With a well-designed convention in place, shapes can be shared across scenarios, making the validation process both more efficient and more maintainable.

- (ii) **CQ-Centric Shape Modeling** Furthermore, the decision regarding which constraints to include within a single shape is of importance. Our experience has shown that it is more effective to aggregate all constraints that collectively answer a specific competency question within a single shape, rather than fragmenting them based solely on the structural arrangement of the data elements. The shapes presented in Listing 5 and Listing 6 exemplify this approach, as they incorporate all relevant constraints needed to address a particular competency question. Modeling shapes in this manner enables a more systematic and targeted analysis of the validation report, as violations associated with a shape directly indicate the root cause of the issue.
- (iii) **Domain-Specific Shape Graphs** In a company, different departments often have different validation interests. For example, the financial department focuses on the correctness of monetary amounts, while the warehouse department is primarily concerned with the quantities of sold items. In many cases, these interests are disjoint, making it possible to split the shapes accordingly. This separation reduces the runtime of the entire approach, as fewer constraints need to be validated and fewer potential violations need to be processed.

7 Conclusion and Future Work

We presented an updated version of the automatic approach EDIFACT-VAL that assists EDI experts in validating EDIFACT messages using an invoice knowledge graph. In EDIFACT-VAL, EDIFACT messages are transformed into RDF graphs based on the proposed EDIFACT Ontology and RML mappings. These RDF graphs are then validated using SHACL constraints that were developed in collaboration with domain experts. The new version, EDIFACT-VAL v.2, directly processes the original EDIFACT messages as input and employs a tailored pre-processing strategy. This pre-processing not only simplifies the RDF mapping process but also improves the overall data quality and ensures more precise validation of the EDIFACT messages.

Based on our experiments, we observed that the RDF graphs generated by EDIFACT-VAL v.1 and EDIFACT-VAL v.2 are not identical, as the graphs produced by EDIFACT-VAL v.2 represent a more reliable and accurate version of the EDIFACT message. Additionally, EDIFACT-VAL v.2 achieves a runtime reduction of 66.56% compared to EDIFACT-VAL v.1, making it not only more robust but also significantly faster. The evaluation with domain experts further demonstrated that EDIFACT-VAL can substantially reduce manual efforts in the validation process. The collaboration with E/D/E also shows that the approach can be applied beyond synthetic settings. The tool is currently used in pilot workflows, with plans for integration into routine operations. A crucial takeaway from this evaluation is that the effectiveness of automatic approaches like EDIFACT-VAL ultimately depends on the reliability of the validation results. At present, this information is available only in a machine-readable format, making it challenging for human users to analyze and act upon. Future work should therefore focus on tailoring frameworks for automated analysis and a visual presentation of validation results [20] to further support human decision-making.

Future work could extend our solution to support different business documents in various EDIFACT formats, such as purchase order messages (ORDERS) or despatch advice messages (DESADV). Additionally, with the mandatory implementation of electronic invoices in Germany starting in 2025, the adoption of open standards has become increasingly important, particularly

for organizations that find the cost or complexity of traditional EDIFACT implementations challenging. In this context, we hope that our work contributes to the broader adoption of electronic invoice processing based on knowledge graph and semantic web technologies.

Supplementary Material Statement

The source code for this work is publicly available on GitHub under the following link <https://github.com/DE-TUM/EDIFACT-VAL>. Due to data privacy and confidentiality obligations, the dataset used in this study cannot be made available.

References

- 1 Dipali Baviskar, Swati Ahirrao, and Ketan Kotecha. Multi-layout unstructured invoice documents dataset: A dataset for template-free invoice processing and its evaluation using ai approaches. *IEEE Access*, 9:101494–101512, 2021. doi:10.1109/ACCESS.2021.3096739.
- 2 SSSS Bisetty, Aravind Ayyagari, Archit Joshi, Om Goel, Lalit Kumar, and Arpit Jain. Automating invoice verification through erp solutions. *International Journal of Research in Modern Engineering and Emerging Technology*, 12(5):131, 2024.
- 3 Devanshi Desai, Ansh Jain, Dhaivat Naik, Nishita Panchal, and Dattatray Sawant. Invoice processing using rpa & ai. In *Proceedings of the International Conference on Smart Data Intelligence (ICSMDI 2021)*, 2021.
- 4 Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. Rml: A generic language for integrated rdf mappings of heterogeneous data. *Ldow*, 1184, 2014. URL: https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.
- 5 M Emmanuel and Sucheta Thakur. An approach to develop invoice validation framework. *International Journal of Engineering Research & Technology (IJERT)*, 1, 2012.
- 6 EN16931-1:2017. Electronic invoicing-part 1 : Semantic data model of the core elements of an electronic invoice. Technical report, European Committee for Standardization, 2017.
- 7 Herminio García-González, Iovka Boneva, Slawek Staworko, José Emilio Labra-Gayo, and Juan Manuel Cueva Lovelle. Shexml: improving the usability of heterogeneous data mapping languages for first-time users. *PeerJ Computer Science*, 6:e318, 2020. doi:10.7717/PEERJ-CS.318.
- 8 Michael Grüninger and Mark S Fox. The role of competency questions in enterprise engineering. In *Benchmarking—Theory and practice*, pages 22–31. Springer, 1995.
- 9 Ramanathan V Guha, Dan Brickley, and Steve Macbeth. Schema.org: Evolution of Structured Data on the Web. *Communications of the ACM*, 59(2):44–51, 2016. doi:10.1145/2844544.
- 10 Hiruni Gunaratne and Ingrid Pappel. Enhancement of the e-invoicing systems by increasing the efficiency of workflows via disruptive technologies. In *Electronic Governance and Open Society: Challenges in Eurasia: 7th International Conference, EGOSE 2020, St. Petersburg, Russia, November 18–19, 2020, Proceedings 7*, pages 60–74. Springer, 2020.
- 11 Mouad Hamri, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. Document information extraction: An analysis of invoice anatomy. *Applied Computational Intelligence and Soft Computing*, 2024(1):7599415, 2024. doi:10.1155/2024/7599415.
- 12 Matthew Horridge, Rafael S Gonçalves, Csongor I Nyulas, Tania Tudorache, and Mark A Musen. Webprotégé: A cloud-based ontology editor. In *Companion Proceedings of The 2019 World Wide Web Conference*, pages 686–689, 2019. doi:10.1145/3308560.3317707.
- 13 Christian Huemer, Philipp Liegl, and Marco Zapletal. Electronic data interchange and standardization. *Handbook of e-Tourism*, pages 1–29, 2020.
- 14 Ana Iglesias-Molina, David Chaves-Fraga, Ioannis Dasoulas, and Anastasia Dimou. Human-Friendly RDF Graph Construction: Which One Do You Chose? In *International Conference on Web Engineering*, pages 262–277. Springer, 2023. doi:10.1007/978-3-031-34444-2_19.
- 15 Holger Knublauch and Dimitris Kontokostas. Shapes constraint language (SHACL). W3C recommendation, W3C, July 2017. URL: <https://www.w3.org/TR/2017/REC-shacl-20170720/>.
- 16 Daniel Krech, Gunnar Aastrand Grimnes, Graham Higgins, Nicholas Car, Jörn Hees, Iwan Aucamp, Niklas Lindström, Natanael Arndt, Ashley Sommer, Edmond Chuc, Ivan Herman, Alex Nelson, Jamie McCusker, Tom Gillespie, Thomas Kluyver, Florian Ludwig, Pierre-Antoine Champin, Mark Watts, Urs Holzer, Ed Summers, Whit Morriss, Donny Winston, Drew Perttula, Filip Kovacevic, Remi Chateaneu, Harold Solbrig, Benjamin Cogrel, and Veyndan Stuart. RDFLib, January 2025. doi:10.5281/zenodo.6845245.
- 17 Steffen Lohmann, Vincent Link, Eduard Marbach, and Stefan Negru. Webvowl: Web-based visualization of ontologies. In *Knowledge Engineering and Knowledge Management: EKAW 2014 Satellite Events, VISUAL, EKM1, and ARCOE-Logic, Linköping, Sweden, November 24–28, 2014. Revised Selected Papers. 19*, pages 154–158. Springer, 2015. doi:10.1007/978-3-319-17966-7_21.
- 18 Johannes Mäkelburg and Maribel Acosta. EDIFACT-VAL. Software, version 2.0., swhId: swh:1:dir:d820a10d861cfd9e361208220cf7aad3b03ef1a6 (visited on 2025-12-05). URL:

- <https://github.com/DE-TUM/EDIFACT-VAL>, doi:10.4230/artifacts.25246.
- 19 Johannes Mäkelburg, Christian John, and Maribel Acosta. Automation of electronic invoice validation using knowledge graph technologies. In *European Semantic Web Conference*, pages 253–269. Springer, 2024. doi:10.1007/978-3-031-60626-7_14.
 - 20 Johannes Mäkelburg, Zenon Zacouris, Jin Ke, and Maribel Acosta. SHACL Dashboard: Analyzing Data Quality Reports over Large-Scale Knowledge Graphs. In *Proceedings of the 24th International Semantic Web Conference (ISWC 2025)*, Lecture Notes in Computer Science. Springer, 2025. to appear. doi:10.1007/978-3-032-09530-5_6.
 - 21 James McKinney and Renato Iannella. vCard Ontology – for describing People and Organizations. W3C note, W3C, May 2014. URL: <https://www.w3.org/TR/2014/NOTE-vcard-rdf-20140522/>.
 - 22 Filip Milosheski and Aleksandar Karadimce. Improving operational efficiency with an automated invoicing system. *Financial Engineering*, 3:221–230, 2025.
 - 23 Natalya F Noy, Deborah L McGuinness, et al. Ontology development 101: A guide to creating your first ontology, 2001.
 - 24 David Peter. hyperfine, March 2023. URL: <https://github.com/sharkdp/hyperfine>.
 - 25 Jan Sál. Data mining as tool for invoices validation. *IT for Practice 2018*, page 121, 2018.
 - 26 Thomas Saout, Frédéric Lardeux, and Frédéric Saubion. An overview of data extraction from invoices. *IEEE Access*, 12:19872–19886, 2024. doi:10.1109/ACCESS.2024.3360528.
 - 27 Michael Schulze, Markus Schröder, Christian Jilek, Torsten Albers, Heiko Maus, and Andreas Dengel. P2p-o: A purchase-to-pay ontology for enabling semantic invoices. In *The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings 18*, pages 647–663. Springer, 2021. doi:10.1007/978-3-030-77385-4_39.
 - 28 Cogan Shimizu, Quinn Hirt, and Pascal Hitzler. Modl: A modular ontology design library. *arXiv preprint arXiv:1904.05405*, 2019. arXiv: 1904.05405.
 - 29 David Shotton. FRAPO, the Funding, Research Administration and Projects Ontology, April 2017. URL: <http://purl.org/cerif/frapo>.
 - 30 Ashley Sommer and Nicholas Car. pySHACL, January 2022. doi:10.5281/zenodo.4750840.
 - 31 Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Mariano Fernández-López. The neon methodology for ontology engineering. In *Ontology engineering in a networked world*, pages 9–34. Springer, 2011. doi:10.1007/978-3-642-24794-1_2.
 - 32 Dylan Van Assche, Thomas Delva, Pieter Heyvaert, Ben De Meester, and Anastasia Dimou. Towards a more human-friendly knowledge graph generation & publication. In *ISWC2021, The International Semantic Web Conference*, volume 2980. CEUR, 2021. URL: <https://ceur-ws.org/Vol-2980/paper384.pdf>.