Provider-agnostic knowledge graph extraction from user stories using large language models

Thayná Camargo da Silva[1,*], Leen Lambers[1], Sébastien Mosser[2] and Kate Revoredo[3,†]

[1]Brandenburg University of Technology Cottbus-Senftenberg, Germany
[2]McSCert, McMaster University, Hamilton, Ontario, Canada
[3]Humboldt-Universität zu Berlin, Berlin, Germany

Abstract

In agile software development, it is common to employ user stories to capture requirements. Specifying requirements in structured natural language has the advantage that requirements are easily understood by domain experts. As requirements evolve and become more complex over time, their analysis also becomes more difficult. Requirement specifications in the form of knowledge graphs have been proven to be useful to partially automate this analysis and make it more manageable. There are related works that automate the translation of user stories into knowledge graph representations, making the previous manual translation less error-prone and more efficient. A recent approach of Arulmohan et al. employs large language models (LLMs) to automate the translation and compares it with alternatives based on dedicated Natural Language Processing (NLP). A large experiment revealed that the latter outperformed the LLM-based solution.

Because of the stochastic nature of LLMs, the fact that they evolve over time, and the availability of different LLM providers (i.e., organizations that provide access to an LLM such as OpenAI (GPT) and Meta (LLaMA)), the same experiment run today or with another LLM would generate different results. For this reason, in this paper we present an end-to-end automated approach for an LLM-based translation that is provider-agnostic and can be easily (re)evaluated against a given ground truth. We explain the design and implementation of our solution based on the LangChain framework. We, moreover, present an evaluation script and we report on experiments that we have performed using the script with different LLM providers. We could show that some of the LLMs are indeed able to close the gap compared to a dedicated NLP approach. We conclude the paper with a discussion of the consequences of LLM-based automated requirements processing.

Keywords

Knowledge graphs, Requirements, LLMs, Langchain

1. Introduction

User stories [1] are commonly used in agile software development to capture requirements in semi-structured natural language. They are also known as backlog items, within the Scrum framework [2]. They describe both the functionality desired by specific stakeholders and the value they provide from the perspective of these stakeholders. Typically, they follow the Connextra format [1]:

*As a <**PERSONA**>, I want <**ACTIONS** over **ENTITIES**> so that <**BENEFIT**>.*

User stories have the advantage that any stakeholder can support their specification and validation more easily. When a backlog containing user stories grows, it is difficult to get a structured overview over all requirements, which may lead to development delays [3]. Previous work therefore investigates the translation of user stories into more formal requirement specifications such as domain models or *knowledge graphs* [4, 5], enabling their automated requirement analysis, for example, to detect inconsistencies, overlaps or dependencies. A recent approach [6] uses *large language models (LLMs)*

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10-13, 2025

*Corresponding author.

†Kate Revoredo is funded by the Berliner Chancengleichheitsprogramm (BCP) as part of the DiGiTal Graduate Program.

th.camargodasilva@gmail.com (T. C. d. Silva); leen.lambers@b-tu.de (L. Lambers); mossers@mcmaster.ca (S. Mosser); kate.revoredo@hu-berlin.de (K. Revoredo)
0009-0000-9396-0702 (T. C. d. Silva); 0000-0001-6937-5167 (L. Lambers); 0000-0001-9769-216X (S. Mosser); 0000-0001-8914-9132 (K. Revoredo)

published 2025-12-10

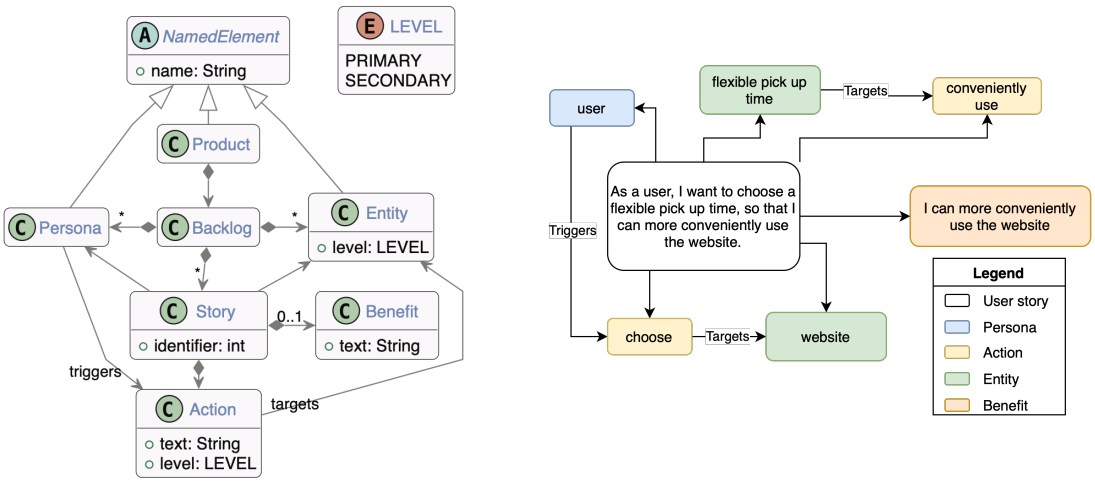

Figure 1: Knowledge graph (KG) architecture for a product backlog [6] (left) and User story as KG (right)

to support knowledge graph extraction from user stories. The extraction is illustrated on a user story example in Figure 1. On the right, a user `Story` is depicted as the central node of the graph, and the colored nodes and further relationships depict how this user story can be decomposed into nodes and edges following the knowledge graph architecture for product backlogs on the left. The `Product` and `Backlog` nodes are omitted for simplicity from the graph on the right, containing only one user story.

Using a curated dataset [7] as ground truth[1], Arulmohan et al. [6] compared their LLM-based solution with Visual Narrator [9], NLP open-source software using techniques such as Part-of-Speech tagging and rule-based extraction, and a Conditional Random Fields (CRF) [10] model, a statistical model that can be used to predict patterns based on their context. The results showed that while Visual Narrator performed reasonably well in identifying personas and actions due to their predictable positions within the text, it struggled with entity extraction, often misidentifying or omitting them. The LLM-based solution demonstrated a substantial improvement over Visual Narrator in all categories. It required significantly less development effort and achieved superior results, particularly in identifying actions. However, their solution was outperformed by the CRF-based model, trained specifically for this task. The LLM-based solution was still considered a promising approach for rapid prototyping.

Because of the stochastic nature of LLMs, the fact that they evolve over time, and the availability of different LLM providers (i.e., organizations that provide access to an LLM such as OpenAI (GPT) and Meta (LLaMA)), the same experiment run today or with another LLM would generate different results. Consequently, we explored in this paper if we can come up with an LLM-based end-to-end automated approach for the knowledge graph extraction that is *provider-agnostic* and can be easily *(re)evaluated against a given ground truth.* We explain the design and implementation of our solution based on the LangChain framework in section 3. LangChain [11], a robust open-source Python framework for developing LLM-powered applications, provides connectivity to various LLM providers and offers a declarative syntax to manage complex LLM interactions. We designed and implemented a custom-made module focusing particularly on knowledge graph extraction. Moreover, our solution relies on LangChain for *defining a prompt template* in advance such that input user stories can be dynamically added to the prompt, *interacting with API via chains* standardizing and automating input feeding to LLMs and processing their outputs, and *using pre-built modules* facilitating various tasks such as configuring LLMs, performing data transformations, and interacting with external databases (e.g., Neo4j [12]). In section 3, we concentrate on explaining the details of our custom-made module as well as prompt template definition. Finally, we also present an evaluation script that enables systematic and automated (re)evaluation against a given ground truth, and we use this script exemplarily in two experiments using different LLM providers in section 4. We could show that some of the LLMs are indeed able to

[1]Based upon a requirements dataset by Dalpiaz et al. [8], consisting of 22 product backlogs containing 1679 user stories.

close the gap compared to the CRF approach [6]. We conclude the paper in section 5 with a summary of our contribution, a discussion of the obtained results and an outlook. This work is based on a master thesis [13], where more details to the approach and its evaluation can be found.

2. Related Works

In this section, we describe the two main streams of research related to ours: *i)* domain modeling using artifacts, such as ontologies and knowledge graphs to represent the requirements extracted from user stories [14]; and *ii)* the use of LLMs to support requirement engineering [15, 16].

For what concerns stream *i)*, A user story modeling ontology was proposed by Mancuso et al. [4], which created a domain-specific modeling language and integrated it into the AOAME modeling tool, resulting in a visual user story. Ladeinde et al. [5] also proposed the use of knowledge graphs to model user stories. Their approach uses NLP techniques to extract the role, goal, and benefit, and model them into an ontology. Arulmohan et al. [6] extracted knowledge from user stories using LLMs, specifically ChatGPT [2], and modeled them into knowledge graphs. The knowledge graph proposed by Arulmohan et al. [6] is considered more comprehensive and flexible compared to the models of Mancuso et al. [4] and Ladeinde et al. [5], as the former lacks general node types, which complicates querying, while the latter extracts only three node types and does not standardize relationship type.

Regarding stream *ii)*, White et al. [17] propose prompt patterns that leverage large language models (LLMs), such as ChatGPT, to facilitate the elicitation and identification of missing requirements. This approach helps capturing user needs more comprehensively during the early stages of software development. Endres et al. [18] explore the use of LLMs to formalize requirements from natural language intent. This approach holds promise for streamlining the transition from user stories to formal specifications, improving clarity, and reducing ambiguity. Arulmohan et al. [6] pioneered a distinct technique, differing from traditional NLP methods, to model user stories by applying LLMs, specifically ChatGPT. While they propose a framework for transforming requirement concepts into a comprehensive domain representation, they do not address the specific challenge of knowledge graph creation.

In this paper, based on the limitations from both streams, we explore the potential of LLMs for extracting knowledge from user stories and representing this knowledge using knowledge graphs.

3. Approach

3.1. Overview

Our solution automates an end-to-end process, from receiving a backlog, over translating the user stories in the backlog into a graph document, to visualizing the translated knowledge graph in the Neo4j [12] graph database. It starts with *configuring the LLM*, a required input of the USGT module (cf. activity 1 in the BPMN model in Figure 2). The LLM configuration specifies the connection to the LLM API via the LangChain framework [11], which offers the possibility of connecting to several LLM providers[3]. Then it *transforms a user story into Document format*, applying a LangChain ready-made function to convert the user story into a Document object type that is required for LLM interaction (cf. activity 2 in Figure 2)). Then our newly developed UserStoryGraphTransformer (USGT) module (cf. activities 3 and 4 in Figure 2) interacts with the selected LLM and generates a Graph Document from the user story Document it is provided with. We will explain in more detail the USGT module in subsection 3.2. Finally, the Graph Document is stored in the Neo4j database by using LangChain's Neo4j integration to establish a connection with the database and ingest the extracted knowledge graph (cf. activity 4 in Figure 2).

An example of the knowledge graph extracted from *"As a student, I want to learn how to code, so that I can build my own projects."* is depicted in Figure 3, the user story represented by the grey, the persona

[2]https://platform.openai.com/docs/models/gpt-4

[3]https://python.langchain.com/docs/integrations/chat/

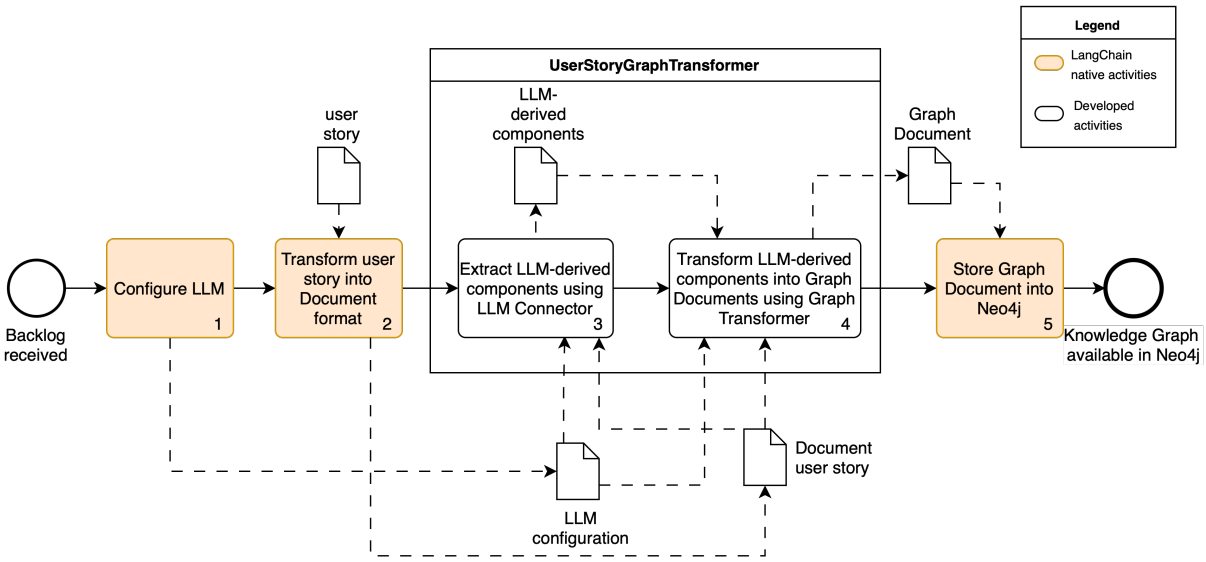

Figure 2: Behavioral overview of our approach.

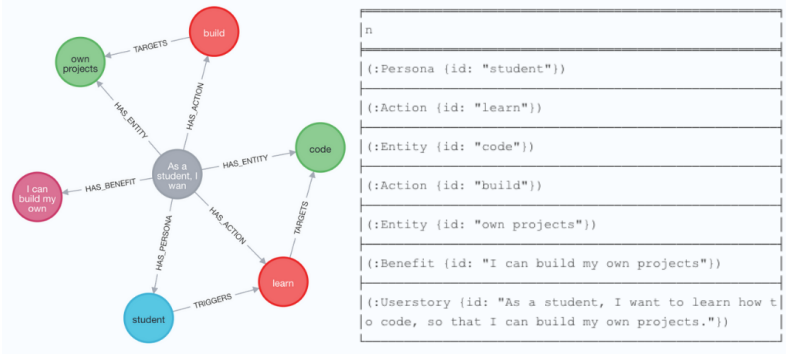

Figure 3: Neo4j snapshot of knowledge graph extracted from example user story.

by the blue, the actions by the red, the entities by the green, and the benefit by the pink node(s).

3.2. UserStoryGraphTransformer Module

We developed the UserStoryGraphTransformer (USGT) module as a specialized module to extract knowledge graphs from user stories using arbitrary Large Language Models (LLMs). LangChain's prebuilt *LLMGraphTransformer* module automates the construction of knowledge graphs from text data using an LLM. The *LLMGraphTransformer* struggled to fully extract nodes and relationships from user stories due to the fine-grained, domain-specific ontology required and the high information density, exceeding the capabilities of general knowledge graph extraction. We customized it to more effectively capture the specific structure and relationships within the user stories. The USGT module processes each user story independently. This decision aligns with the INVEST criteria. This decision is further supported by the related study of Arulmohan et al. [6], where researchers found that when the LLM was provided with a list of user stories, it tended to confuse nodes between different stories and even generated misinformation.

The *USGT module*, as illustrated in the component diagram in Figure 4, comprises two main components: the *LLM Connector* and the *Graph Transformer*.

The *LLM Connector* receives as input a user story and an LLM configuration. The user story is a textual input in a Document format, while the LLM configuration, defined by the user, specifies the LLM provider and the model to be used. This configuration enables communication with the LLM

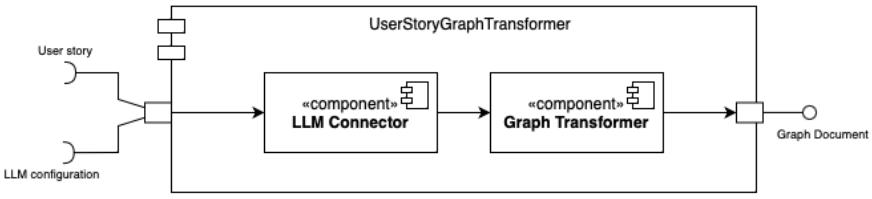

Figure 4: Component diagram of the UserStoryGraphTransformer module.

API by sending requests and receiving responses. Upon receiving these inputs, the LLM Connector uses Prompt Templates to define prompts that guide the LLM to identify and extract the desired nodes and relationships from the user story, and LangChain chains are used to interact with the LLM. As an output, the LLM Connector delivers the LLM-derived components of a knowledge graph, composed of nodes and relationships. The *Graph Transformer* processes the LLM-derived components, enriching it with additional information, and converting nodes and relationships into a Graph Document suitable for ingestion into the graph database. It ensures that the knowledge graph components extracted using the LLM Connector adhere to the ontology constraints and formats. The Graph Document is a special data structure to represent a Knowledge Graph that is required to be ingested by the Neo4j database.

When analyzing the knowledge graph architecture (cf. Figure 1), we identified several opportunities to *streamline the LLM's workload*, saving up resources, such as API costs and processing time, while also improving accuracy. The *userstory* node represents the input itself and, therefore, does not require further processing by the LLM to be represented as a node in the knowledge graph. Instead, it is handled by the Graph Transformer component, by enriching the LLM's extracted nodes with this *userstory* node by default. Additionally, some relationships can be logically inferred from the existing nodes. For instance, if the LLM is able to extract a *persona* node, the *has_persona* relationship can be established with certainty. The same logic applies to *has_action*, *has_entity*, and *has_benefit* relationships. The logical inference of relationships allows the LLM to concentrate its resources on extracting more complex relationships – specifically *triggers* and *targets* – that demand a deeper semantic understanding of the user story content. The *LLM Connector* therefore guides the LLM to only extract the *persona*, *action*, *entity*, and *benefit* nodes and the relationships *triggers* and *targets*. The Graph Transformer is responsible for enriching the knowledge graph with the *userstory* node and for deriving the logically inferrable relationships from extracted nodes from the LLM connector.

3.3. Implementation

We have implemented in Python the end-to-end process as depicted in Figure 2 and released it as open source [19]. The `us_graph_transformer.py` file contains the core logic of the USGT module. To facilitate its use and evaluation, the repository includes several supporting files. `extractor.py` serves as the primary execution script, while `evaluation.py` calculates performance metrics using the ground truth data from `pos_baseline/`. Data management is handled through `template.json` (for input formatting) and the `extracted-user-stories/` and `evaluation/` directories (for storing experimental outputs). We focus here on explaining implementation details of the custom-made USGT module (see subsection 3.2).

Models like ChatGPT 3.5 from OpenAI support function calls and provide structured outputs in a predefined JSON format, simplifying data processing and integration. However, models like Llama 3 by Meta do not have this capability, requiring more explicit prompts and examples to guide the LLM toward a desired output structure. Therefore, the *LLM Connector* defines different types of prompt templates after assessing if the selected language model supports it. For models that support function calls, an output schema template is defined, containing nodes and relationship keys, whose values will be filled out by the LLM response. If the selected model does not support function calls, additional examples and detailed instructions are added to the prompt to guide the LLM to respond in a specific JSON format. Then, the LLM Connector creates two separate prompts (main and benefit prompts):

one for extracting the *persona*, *action*, and *entity* nodes and their relationships, and another solely for extracting the *benefit* node. We made this separation because the first three node types are always present in any given user story, ensuring that their relationships are consistently extracted. In contrast, the *benefit* node is optional and does not participate in the relationships to be extracted by the LLM (*triggers* and *targets*). By isolating the *benefit* node extraction into its own prompt, the extraction performance and consistency were improved. To guide both prompts, the 6 Strategies for getting better results by OpenAI [20] were used as a reference (accessed in November 2024). This guide is composed of high-level principles, strategies, and specific methods, and tactics, that can be used to implement those strategies. The purpose of applying these methods is to improve the chances of the LLM to produce the desired outcome. Nine of the tactics were fully implemented, one was partially implemented, and nine were not relevant or could not be applied in this solution. Key implemented tactics included: instructing the model to adopt a specific persona, using delimiters to clearly define input sections, explicitly specifying task completion steps, providing illustrative examples, and directing the model to answer using provided reference text. The LLM Connector thus constructs two LangChain chains: one for the main and one for the benefit prompts. It sends two requests (one for each chain) to the LLM API and obtains the LLM responses, or LLM-derived components.

The *main prompt* uses two different roles, the system role to set the context, and the human role to provide the input, in this case, the user story. The context part is structured into six different parts: **Overview**: Directs the model to assume the role of a requirements engineer and establishes the task context. **Nodes**: Introduces the concept of a node, describes the three possible node types (Persona, Action, and Entity), and provides detailed instructions for extracting all relevant nodes from the user story. **Relationships**: Specifies what constitutes a relationship, presents the two possible relationship types (TRIGGERS between Persona and the main Action, and TARGETS between Action and Entity), and emphasizes that no other relationships should be extracted. **Coreference Resolution**: Reinforces the importance of maintaining consistency and coherence across extracted nodes and relationships. **Strict Compliance**: Stresses the necessity for the model to adhere strictly to the given instructions. **Example**: Provides an example of a user story, illustrating the extracted nodes and relationships to guide the model's output. We show an excerpt of the main prompt in Listing 1) focusing on node extraction. The complete listing of the prompt is in our repo [19], in the file `us_graph_transformer.py`.

```
system_prompt = (
    """
Knowledge Graph Constructor Instructions \n
## 1. Overview \n
You are a specialized requirements engineer, who understands about scrum framework. Your task is to
    analyze and extract nodes and relationships from user stories to build a knowledge graph.
You have to extract as much information as possible without sacrificing accuracy. Do not add any
    information that is not explicitly in the mentioned user story. \n
## 2. Nodes \n
Nodes represent concepts in a user story. Given a user story, you need to extract: Persona: there is only
    one persona node per user story, introduced as 'As a *persona*,'. Actions: are all verbs in the user
    story that describe what the persona desires to do (e.g. move on, access, have). Extract the verb
    only, without modifiers. Entities: are nouns and each noun must be extracted as a separate entity,
    even if they seem related or grouped. Include any modifiers that clarify the entity (e.g. library
    database, domain).  \n
**Consistency**: Ensure you use available types for node labels, you necessarily extract at least 4 nodes:
    persona, action, entity.\n
**Node IDs**: Never utilize integers as node IDs. Node IDs should be names or human-readable identifiers
    extracted as found in the user story.\n
**Extract all actions and entities**: capture every action and its corresponding entity.\n
**Separate verbs**: consider each verb as a distinct action and its objects as related entities.\n
...
## 6. Example \n
... """
)
default_prompt = ChatPromptTemplate.from_messages([("system",system_prompt),("human",("Use the given
    format to extract information from the following input: {input}" )) ])
```

Listing 1: Main prompt design: node extraction

In addition to the main prompt, five examples were included to to illustrate the expected output format (cf. Listing 2) and guide the LLM in cases where function call support is not available. Thereby, *text* represents the input user story, *head* and *tail* correspond to the extracted nodes, *head type* and *tail type* indicate the type of each node, and *relation* specifies the relationship between them.

```
{
        "text": "As a business owner, I want to give my inputs on the product development.",
        "head": "business owner",
        "head_type": "Persona",
        "relation": "TRIGGERS",
        "tail": "give",
        "tail_type": "Action",
    }
```

Listing 2: Example of expected output when function call is not supported.

The *benefit prompt*, is much simpler, as it has a single objective: to extract the *benefit* node, if present. We refer to our open-source repository [19] (file `us_graph_transformer.py`) for the listing of the benefit prompt. It is structured into the following three components: **Overview**: Instructs the model to assume the role of a requirements engineer and provides context for the task. **Benefit**: Defines what constitutes a benefit sentence within a user story and directs the model to extract it only if it is explicitly stated. **Examples**: Provides two examples of user stories—one where the *benefit* node is present and can be extracted, and another where it is absent, prompting the LLM to return an empty response.

Once the *Graph Transformer* receives the LLM-derived knowledge graph components from the *LLM Connector*, again their processing depends on the LLM's support for function calls. If the LLM supports function calls, the LLM response is already structured into JSON format, and the nodes and relationships can be directly accessed. In the other case, the LLM response cannot be structured into a JSON format. It is a string, and if the LLM followed all the instructions provided, it is possible to parse this string into a JSON format. Both scenarios result in a knowledge graph components object, which contains the nodes and relationships extracted by the LLM. The nodes' list is enriched by adding the input user story as a node. Subsequently, the complete nodes' list is converted into a list of Graph Node objects. As mentioned earlier, the logically inferable relationships from existing nodes *has_benefit*, *has_persona*, *has_entity* and *has_action* are derived and combined with the explicitly extracted relationships from the knowledge graph components before converting them into graph relationships. Finally, the extracted graph nodes and relationships are combined into a graph document.

4. Evaluation

We developed an evaluation script (`evaluation.py` in [19]) that automatically extracts comparison data supporting the users of our solution to answer the following two research questions (RQs).

> *RQ 1. How does the accuracy of knowledge graph extraction of our solution differs when configuring it with different LLMs?*
> *RQ 2. How does the accuracy of knowledge graph extraction of our solution configured with a specific LLM compares with the CRF and Visual Narrator solution as presented in [6]?*

We exemplarily answer both RQs by using the same dataset [7] as employed for experimentation by Arulmohan et al. [6] as ground truth. This means that we used a cleaned version of the annotated dataset [7], representing 87% of the complete version, to ensure comparability of our results with their results (cf. RQ2). Our evaluation script can be easily generalized to work with a different ground truth. The evaluation script's input expects a JSON structure mirroring that of `template.json` in [19], allowing for flexible evaluation against diverse ground truth datasets.

4.1. Evaluation metrics

We use a combination of *multiple-classification* (MC) metrics, such as F-measure and *token-similarity* (TS) metrics, such as BERTScore [21]. The MC metrics evaluate the LLM's ability to categorize texts into various groups, each representing a label. The TS metrics measure the semantic similarity between the LLM's generated text and a reference. The metrics are calculated at the user story level but are averaged across the backlog to provide a broader perspective. This approach ensures that the evaluation considers variations in context and patterns across different backlogs.

We calculate the *MC metrics* recall, precision and F-measure. As a prerequisite it is essential to define what qualifies as a correct output from the LLM. We adopted the criteria outlined by Arulmohan et al. [6], utilizing three comparison modes: *strict, inclusive, and relaxed*. The strict comparison considers a result correct when the LLM produces the exact same response elements as the ground truth, the inclusive comparison adds some flexibility and considers the LLM's output as a superset of the ground truth, and the relaxed comparison ignores adjective qualifiers and considers plurals as singulars as the same. In addition to the experiment by Arulmohan et al. [6], we added the evaluation of the *benefit* node. The MC metrics were calculated for the strict and inclusive modes, but not for the relaxed mode because of missing Part-of-Speech annotations on the ground-truth dataset. Based on the three comparison criteria, each knowledge graph (KG) component generated by the LLM can be classified as follows: **True Positive**: An element that is considered equivalent to the ground truth. **False Positive**: An element that was incorrectly identified by the LLM, as it should not have been included. **False Negative**: An element that should have been identified by the LLM but was not, indicating a missing component.

As *TS metrics* (as opposed to [6]), we chose BERTScore [21] to assess the quality of the extracted nodes. While other metrics like Perplexity, BLEU, ROUGE, and METEOR exist, their reliance on token overlaps and n-gram matches makes them less suitable for our dataset, which contains many single-token actions and entities. Because these lexical similarity metrics do not fully capture semantic relationships, they are less effective when lexical information is limited. BERTScore provides a more relaxed evaluation by using contextual embeddings to capture deeper meaning and assess semantic similarity.

4.2. Experiments and results

Our *first experiment* uses our evaluation script to exemplarily answer RQ 1 for two different LLM providers: the first is Llama 3 8B by Meta, an open-source, cost-free for research purposes and highly powerful model, and the second is GPT-4o mini, the most cost-efficient small model by OpenAI. We have chosen these models because Llama 3 does not support function calls, while GPT-4o mini does support this functionality, therefore demonstrating the versatility of our solution to work with different LLM providers. For the LLM configuration, both models, Llama 3 and GPT-4o mini, were set to a zero temperature. This configuration parameter is necessary to control the stochasticity of the output, making it more consistent and predictable, since the output of the LLM is non-deterministic. Our experiment involved configuring the LLMs and running the `extractor.py` script to apply the USGT module to extract knowledge graphs from all the user stories within the 22 product backlogs, and then evaluating the results against the ground truth.

We only show detailed results from our evaluation in the strict mode in Table 1 and report on average results for the inclusive and relaxed mode (see Figure 5). We refer to the folder `user-story-extractor/evaluation` in our repo [19] for the evaluation details on the other modes. In the strict mode comparison (Table 1), GPT-4o-mini outperforms Llama 3 in extracting all node types except for *benefit*, where Llama 3 shows a slightly higher performance. On average, Llama 3 struggles primarily with *entity* and *action* extraction in this mode. A detailed analysis of the data reveals that Llama 3 frequently mishandles noun qualifiers and verb complements, both of which are critical in this strict mode. Additionally, a significant performance gap is observed in backlog *g02* for *benefit* extraction. Upon closer inspection, many user stories in this backlog did not have a benefit to extract, yet Llama 3 either attempted to extract other parts from the story or produced hallucinations as the *benefit* node.

Moving to the inclusive and relaxed modes, both models show improved scores across all categories

Table 1
Strict comparison of F-Measures for GPT-4o-mini and Llama 3.

Model	Persona F-Measure	Entity F-Measure	Action F-Measure	Benefit F-Measure
GPT-4o-mini	0.998 ± 0.01	0.786 ± 0.06	0.726 ± 0.13	0.853 ± 0.11
Llama 3	0.964 ± 0.05	0.500 ± 0.10	0.617 ± 0.12	0.855 ± 0.13

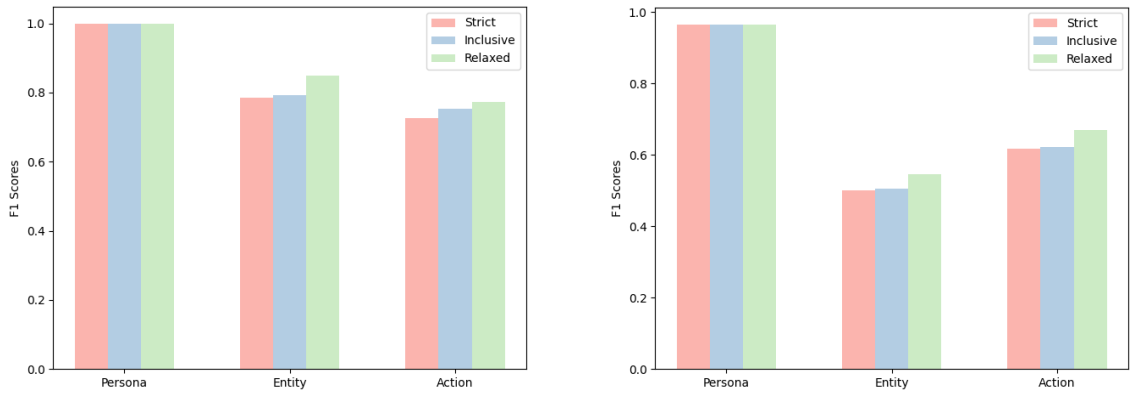

Figure 5: Average comparison of node extraction using GPT-4o-mini (on the left) and Llama3 (on the right)

as expected (see Figure 5). However, GPT-4o-mini continues to lead with a more consistent performance across all categories, maintaining its edge in benefit extraction in the inclusive mode. The BertScore comparison mode brings the semantic alignment perspective, and in this case both models show a good performance (see [19] for concrete evaluation data). Overall, GPT-4o-mini (Figure 5) demonstrates superior performance across all evaluated categories, with consistently higher F-Measure scores. Even though it doesn't have a perfect F-measure when comparing exact string matching, it proves capable in capturing semantic alignment, particularly in the *benefit* node, which poses a great challenge since it has many elements to be extracted. Llama 3, while competitive, exhibits greater variability.

Our *second experiment* aims to answer RQ 2. We used GPT-4 turbo in this experiment as LLM. We conducted it using the same dataset and a zero temperature configuration on the LLM. The earlier study employed Conditional Random Fields (CRF), a tailored machine-learning approach trained on 20% of the dataset. This method outperformed all GPT models (GPT-3.5-turbo-0125, GPT-3.5-turbo-0613, GPT-4-0125-preview, and GPT-4-0613) and the Visual Narrator technique. When comparing CRF to GPT-4-turbo using our solution, CRF still leads in performance; however, the gap has narrowed under the same evaluation metrics, making LLM-based solutions a competitive alternative, in principle.

The results reported for RQ 1 and 2 show that our solution is able to successfully extract nodes and relationships from user stories according to the specific ontology proposed, however the adherence to the ontology, and therefore, the quality of the results depends on the chosen large language model.

5. Conclusion & Discussion

In this work, we focused on making available LLMs for tasks related to requirements extraction, more precisely, the reification of knowledge graphs out of user stories backlogs. By abstracting the task under study from the LLM provider, we offered an easy way to select the *"best"* provider for a given task, as well as extended validation using a reference ground truth. We observed that LLMs have strengths and weaknesses depending on the part of the story to extract, which reinforces the need to be able to seamlessly switch between providers (RQ1). By comparing LLMs to a ground truth and a non-LLM-based approach, we observed that if a supervised approach such as CRF ends up being more efficient in terms of training and computation, LLMs tend to be close to these performances but do not require annotation, which can be helpful in some scenarios.

Threats to validity. One limitation concerns the evaluation against the ground truth. We reused existing benchmarks from non-LLMs approaches to compare our work with the state of practice, leading to an incomplete evaluation. Although the data set was annotated through a rigorous process, the exact annotation of node elements remains somewhat subjective, according to their intrinsic nature. For example, one would consider that out of the text *"I want to add type information to my data"*, one might consider the entity to be the *"type information"*, while others might consider *"data"*, as it includes the former. This subjectivity poses a threat to *construct validity*, as the evaluation might not consistently reflect the true quality or correctness of the extracted knowledge graph components. This is in general mitigated by the operational scale (*e.g.*, thousands of stories) and a choice of evaluation metrics that is robust to small variations. Another limitation is the nondeterministic nature of LLM outputs. Even with model parameters such as the temperature set to zero, responses from the LLM may vary across executions. This variability impacts *internal validity*, as it can lead to inconsistent experimental results, making it difficult to attribute the results solely to the factors being tested. Moreover, it affects the *external validity*, as the same approach might yield inconsistent results when applied in different settings or with alternative data sets. Finally, the evaluation of the knowledge graph focused solely on the extraction of the nodes, as the extraction of relationships was not evaluated in previous comparable work [6].

Discussion. To determine the most assertive approach, it is necessary to consider specific use cases and contexts. For tasks that require constant updates, flexibility, or are part of larger, dynamic systems, *LLMs may offer advantages* and come with the immediate benefit of not necessitating any further annotation or data curation for training. But if this free-lunch approach (no need to build a ground truth) can make sense for prototyping proof-of-concepts, in the long run, it triggers *privacy and security issues*: requirements are often trade secrets, and, as such, using an LLM without proper due diligence might be an issue. This benefit might be impacted if the model has to be fine-tuned. Another issue is related to the difficulty of properly evaluating the outcome. In addition to *lack of determinism*, not having a proper *ground truth* for the tasks considered could alter the evaluation: a lot of research only relies on subjective feedback from interviews to mitigate this, but *quantitative evaluation* cannot be overlooked when integrating LLMs into critical systems [22]. Unfortunately, when taking the quantitative evaluation path, focusing on metrics such as accuracy or F-score might not be sufficient or even reasonable in the long run. For example, in our use case, all LLMs have an accuracy of 1 regarding the identification of the persona. This is highly related to the structure of the Connextra pattern, where an approach taking the nouns located between the third word and the first comma would also have the same accuracy. By only focusing on such metrics, we as researchers do not consider the impact of our decisions on a larger scale. One immediate thing people might think of is the *energy cost* of each request, in addition to the cost of training associated with LLMs. But the *human cost*, for example, is often overlooked: to build the dataset used for their image-generation features, OpenAI exposed Kenian workers to highly disturbing content (including child abuse, bestiality, rape and sexual slavery), for less than $2/h[4]. If, as researchers in software engineering, we are not ethicians, it is still our responsibility to explore alternative solutions before jumping into the LLM train [23] and use it for use cases where other alternatives would produce results as good as the LLM one.

Outlook. An immediate perspective of our contribution is to extend the benchmark efforts and define a comprehensive way of evaluating knowledge graph building, leveraging not only the nodes but also the relations between them. We envision an approach à la Transformer Ranker [24], where the system can automatically guide practitioners to the LLM that best fits their needs. A second perspective is to capture the needs of the task at stake, and provide feedback to practitioners on how relevant LLMs are for such a task, as well as to identify alternative approaches (*e.g.*, heuristic based, supervised) that could be considered for the same objective.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

[4] https://time.com/6247678/openai-chatgpt-kenya-workers/

References

[1] M. Cohn, User stories applied: For agile software development, Addison-Wesley Professional, 2004.

[2] K. Schwaber, J. Sutherland, The scrum guide, Scrum Alliance 21 (2011) 1–38.

[3] G. Lucassen, M. Robeer, F. Dalpiaz, J. M. E. Van Der Werf, S. Brinkkemper, Extracting conceptual models from user stories with visual narrator, Requirements Engineering 22 (2017) 339–358.

[4] M. Mancuso, E. Laurenzi, An approach for knowledge graphs-based user stories in agile methodologies, in: Business Information Research, volume 493 of *LNBIP*, Springer, 2023, pp. 133–141.

[5] A. Ladeinde, C. Arora, H. Khalajzadeh, T. Kanij, J. Grundy, Extracting queryable knowledge graphs from user stories: An empirical evaluation, in: ENASE, SCITEPRESS, 2023, pp. 684–692.

[6] S. Arulmohan, M.-J. Meurs, S. Mosser, Extracting domain models from textual requirements in the era of large language models, in: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, 2023, pp. 580–587.

[7] S. Arulmohan, S. Mosser, M.-J. Meurs, ace-design/qualified-user-stories: Version 1.0, 2023. doi:10.5281/ZENODO.8136975, annotated dataset.

[8] F. Dalpiaz, Requirements data sets (user stories), 2018. doi:10.17632/7ZBK8ZSD8Y.1, dataset.

[9] M. Robeer, G. Lucassen, J. M. E. M. van der Werf, F. Dalpiaz, S. Brinkkemper, Automated extraction of conceptual models from user stories via NLP, in: RE, IEEE Computer Society, 2016, pp. 196–205.

[10] J. D. Lafferty, A. McCallum, F. C. N. Pereira, Conditional random fields: Probabilistic models for segmenting and labeling sequence data, in: ICML, Morgan Kaufmann, 2001, pp. 282–289.

[11] LangChain, 2022. URL: https://github.com/langchain-ai/langchain, last access: 2024-10-10.

[12] Neo4j, 2024. URL: https://neo4j.com/, last access: 2024-06-12.

[13] T. C. da Silva, Extracting Knowledge Graphs from User Stories Using Langchain, Master's thesis, BTU Cottbus-Senftenberg, 2025. doi:https://doi.org/10.26127/BTUOpen-7038.

[14] I. K. Raharjana, D. Siahaan, C. Fatichah, User stories and natural language processing: A systematic literature review, IEEE access 9 (2021) 53811–53826.

[15] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language models for software engineering: A systematic literature review, ACM Trans. Softw. Eng. Methodol. 33 (2024) 220:1–220:79. URL: https://doi.org/10.1145/3695988. doi:10.1145/3695988.

[16] A. Hemmat, M. Sharbaf, S. Kolahdouz-Rahimi, K. Lano, S. Y. Tehrani, Research directions for using llm in software requirement engineering: a systematic review, Frontiers in Computer Science 7 (2025). doi:10.3389/fcomp.2025.1519437.

[17] J. White, S. Hays, Q. Fu, J. Spencer-Smith, D. C. Schmidt, Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, in: Generative AI for Effective Software Development, Springer, 2024, pp. 71–108.

[18] M. Endres, S. Fakhoury, S. Chakraborty, S. K. Lahiri, Can large language models transform natural language intent into formal method postconditions?, Proc. ACM Softw. Eng. 1 (2024). URL: https://doi.org/10.1145/3660791.

[19] T. C. da Silva, Extracting knowledge graphs from user stories using langchain, 2024. doi:10.5281/zenodo.14254058, https://zenodo.org/record/14254058.

[20] OpenAI, Prompt engineering, 2023. URL: https://platform.openai.com/docs/guides/prompt-engineering?ref=blef.fr, last access: 2024-09-07.

[21] T. Zhang*, V. Kishore*, F. Wu*, K. Q. Weinberger, Y. Artzi, Bertscore: Evaluating text generation with bert, in: International Conference on Learning Representations, 2020. URL: https://openreview.net/forum?id=SkeHuCVFDr.

[22] J. J. Norheim, E. Rebentisch, D. Xiao, L. Draeger, A. Kerbrat, O. L. De Weck, Challenges in applying large language models to requirements engineering tasks, Design Science 10 (2024) e16. doi:10.1017/dsj.2024.8.

[23] B. Johnson, T. Menzies, AI Over-Hype: A Dangerous Threat (and How to Fix It), IEEE Software 41 (2024) 131–138. doi:10.1109/MS.2024.3439138.

[24] L. Garbas, M. Ploner, A. Akbik, Transformerranker: A tool for efficiently finding the best-suited language models for downstream classification tasks, 2024. arXiv:2409.05997.