User Interface and Agent Interface for Online Generation of Knowledge Graph's Competency Questions and Question-Query Training Sets

Yousouf Taghzouti[1], Franck Michel[2], Tao Jiang[3], Louis-Felix Nothias[3] and Fabien Gandon[4]

[1]Univ. Côte d'Azur, Inria, ICN, I3S, France
[2]Univ. Côte d'Azur, CNRS, Inria, I3S, France
[3]Univ. Côte d'Azur, CNRS, ICN, France
[4]Inria, Univ. Côte d'Azur, CNRS, I3S, France

Abstract

Few question-query datasets exist for fine-tuning large language models on tasks such as translating natural language questions into SPARQL queries. While it is often recommended that competency questions and their corresponding SPARQL queries accompany a knowledge graph (KG), this is rarely the case in practice. In this paper, we introduce Q^{2}Forge, a web application designed to support the creation of question-query pairs for any KG. The tool enables users to generate, test, and refine competency questions and their SPARQL counterparts directly within the interface. It employs a retrieval-augmented generation architecture to support a wide range of KGs efficiently. The result is an open-source solution for building reusable question-query datasets applicable to any KG. We also present recent developments around the Model Context Protocol, moving toward the agentification of Q^{2}Forge —enabling natural language interactions in addition to traditional UI-based workflows.

Keywords

Competency Question, SPARQL, LLM, MCP

1. Motivation

Knowledge graphs (KGs) are increasingly being used alongside large language models (LLMs) in a variety of use cases [1, 2, 3, 4, 5, 6, 7]. One of these use cases consists in taking a question in natural language (NL), translate it into the graph's query language, submit the query to the graph, and use the structured response to return a NL answer to the user. Training and evaluating such translation systems requires datasets of curated question-query pairs tailored to a given KG or at to the very least relevant to its domain of interest.

In the case of RDF KGs however, not all KGs come with example NL questions, that we shall call competency questions (CQs), and their SPARQL query counterparts, as can be seen when browsing the KGs in the Linked Open Data Cloud.[1] This lack can be addressed after the publication of the graph, by enabling the construction of a dataset of question-query pairs (hereafter referred to as Q^{2}set). This usually involves three main steps: coming up with CQs tailored to the KG; translating the CQs into SPARQL queries; and then validating and refining the queries to match the expected needs. However, this procedure is time-consuming and requires multiple competencies (expertise in the domain of the KG as well as knowledge in Semantic Web technologies). To address this challenge, in this work we present an intuitive webapplication that guides the user through this procedure and automates the three main steps of generating Q^{2}sets for a given KG while relying on LLMs.

yousouf.taghzouti@univ-cotedazur.fr (Y. Taghzouti); franck.michel@inria.fr (F. Michel); tao.jiang@cnrs.fr (T. Jiang); louis-felix.nothias@cnrs.fr (L. Nothias); fabien.gandon@inria.fr (F. Gandon)
https://youctagh.github.io/ (Y. Taghzouti); https://w3id.org/people/franckmichel (F. Michel); https://orcid.org/0000-0002-5293-3916 (T. Jiang); https://lfnothias.github.io/ (L. Nothias); http://fabien.info/ (F. Gandon)
0000-0003-4509-9537 (Y. Taghzouti); 0000-0001-9064-0463 (F. Michel); 0000-0002-5293-3916 (T. Jiang); 0000-0001-6711-6719 (L. Nothias); 0000-0003-0543-1232 (F. Gandon)

[1]The Linked Open Data Cloud: https://lod-cloud.net/datasets

In the remainder of this paper we first describe in further details the architecture of Q^{2}Forge and the steps involved in each of the main functions. We then present a concrete use case in the metabolomics domain. We finally report on our current developments with the Model Context Protocol (MCP) towards an "agentification" of Q^{2}Forge, where part of the UI interactions can be replaced with NL interactions. Lastly, we discuss the potential and future directions of our approach.

2. Q^{2}Forge Architecture and Workflow

To guide users through the generation of Q^{2}sets for any given KG, we have designed a workflow comprising the following steps: (1) setup of a KG configuration; (2) generation of CQs; (3) generation of counterpart SPARQL queries; and (4) SPARQL queries refinement. Below, we will describe each step, focusing on step (3), where we introduced a Retrieval-Augmented Generation (RAG) mechanism to enhance the automatic SPARQL generation process.

The webapp supports multiple, simultaneous user sessions, where each user may configure their KGs of interest, and concurrently invoke language models at each step of the workflow. The webapp also supports asynchronous interactions, such that a user may launch an action and return only later to check the result. For these reasons, users must create individual accounts, which also enables them to monitor their usage quotas and edit their KG configurations. The webapp's code is open source and available on GitHub.[2] An online prototype is accessible at this URL: https://www.w3id.org/q2forge/.

2.1. KG Configuration and Pre-processing

In the first step, the user creates a configuration by filling in a web form with information about the target KG, such as its name, description, used prefixes, and SPARQL endpoint. If available, the user can add natural language questions and associated SPARQL query examples. These examples shall provide richer context during the query generation and are therefore recommended to improve the performance, but they are not required. The user may typically use the webapp without providing any examples to kick-off the process by generating and curating a first set of question-query pairs, and later on reconfigure the KG with these curated examples.

Furthermore, the performance of LLMs may vary from one task to the other, and users may have their preferred models wrt. various criteria: performance, quality, window size, cost etc. For this reason, a user may configure multiple models and decide which one to use at each step of the process.

The SPARQL query generation workflow uses a RAG approach to improve performance. To do so, different kinds of information must be extracted from the KG, such as ontology classes and properties. These classes and properties, along with their labels and descriptions, are stored and text-embedded. If available, the SPARQL query examples are embedded too and stored separately. As a default, the `nomic-embed-text` embedding model and the FAISS vector store are used, yet users can use different options by editing the configuration.

2.2. CQ Generation

The second step allows the user to generate a number of CQs. A form is filled in automatically if the KG configuration has been set up previously, otherwise the user shall fill it in manually. An LLM is prompted to generate CQs based on the KG description, its schema and any additional information that the user can add in a free text field. To obtain a JSON-structured output, the user can activate a toggle to force the LLM to behave in this way. The user optionally changes the language model to use and launches the generation process. Once the CQs have been generated, the user can download them or save them in the browser's local storage for reuse in further steps.

[2] Github repo: https://github.com/Wimmics/q2forge

2.3. SPARQL Query Generation

In the third step, the user can generate a SPARQL query counterpart for a CQ. Multiple scenarios are implemented in Q²Forge to carry out this task. Figure 1 depitcs one of them and below, we describe each step of this scenario:

- **Initial question:** the scenario is initiated by the user posing a CQ.
- **Question validation:** the question is validated by an LLM to ensure it falls within the scope of the KG. If the CQ is considered invalid, the workflow stops. The validation is performed by an LLM based on the relevance of the question to the graph domain, relying in part on the textual description of the KG during its configuration. However, there is no guarantee that the graph contains everything needed to answer the question, as some necessary concepts may be missing.
- **Question pre-processing:** the CQ undergoes a named entity extraction using Spacy, in preparation for the subsequent RAG steps.
- **Select similar classes:** the most relevant ontological classes are selected by a similarity search between the CQ and the text embedding of the classes computed in the pre-processing step.
- **Get context information about the classes:** for each selected similar class, retrieve a description of the properties and value types used by its instances in the KG.
- **Select similar query examples:** select relevant query examples based on a similarity search between the CQ and the SPARQL query examples embedded in the pre-processing step.
- **Generate query:** generate and submit a contextualized prompt from a template[3], providing the model with the KG configuration and the relevant classes and queries selected in the previous steps.
- **Verify query and retry:** check whether the generated SPARQL query is syntactically correct. If not, use a retry prompt template[4] that includes the context from the previous step in addition to the generated query and the validation error, e.g. syntax error. Then submit the retry prompt to generate a new query. This step can be repeated a configurable maximum number of times.
- **Execute the SPARQL query:** once a valid SPARQL query was generated, submit it to the KG endpoint and get the results.
- The process ends by prompting an LLM to **interpret the results** wrt. the initial question.

Note that the language model used for each step of the scenario (seq2seq or text embedding) can be configured separately. This provides users with greater flexibility and control. Figure 2 shows the SPARQL query generation interface. Past chats can be restored and are shown on the left.

2.4. SPARQL Query Refinement

In the fourth and final step, the user can refine a SPARQL query to match a CQ. This question-query pair can either result from the previous two steps, or the user may paste a handcrafted pair to solely use the refinement step. The refinement process is driven by an LLM judge, and the webapp provides several functionalities to enable this:

- Edit and run a query in a YASGUI editor that supports syntactic highlighting and validation.
- Add known prefixes configured in step 1.
- Extract the qualified (prefixed) names (QNs) and fully qualified names (FQNs) from the query, and obtain their labels and descriptions. This ensures readability for the user, and provides context to the LLM during the judgement step. This is particularly useful when IRIs contain opaque identifiers, for example property http://purl.obolibrary.org/obo/RO_0000056 has the label “participates in" and the description “a relation between a continuant and a process, in which the continuant is somehow involved in the process".

[3] https://github.com/Wimmics/gen2kgbot/blob/master/app/scenarios/scenario_6/prompt.py#L3
[4] https://github.com/Wimmics/gen2kgbot/blob/master/app/scenarios/scenario_6/prompt.py#L32

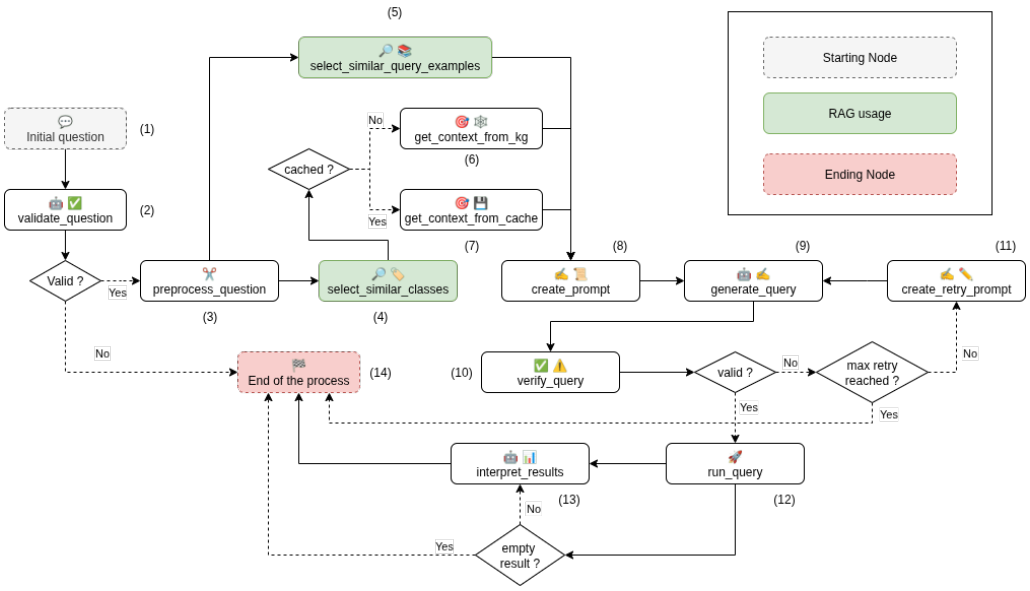

Figure 1: SPARQL Query Generator/Executor workflow

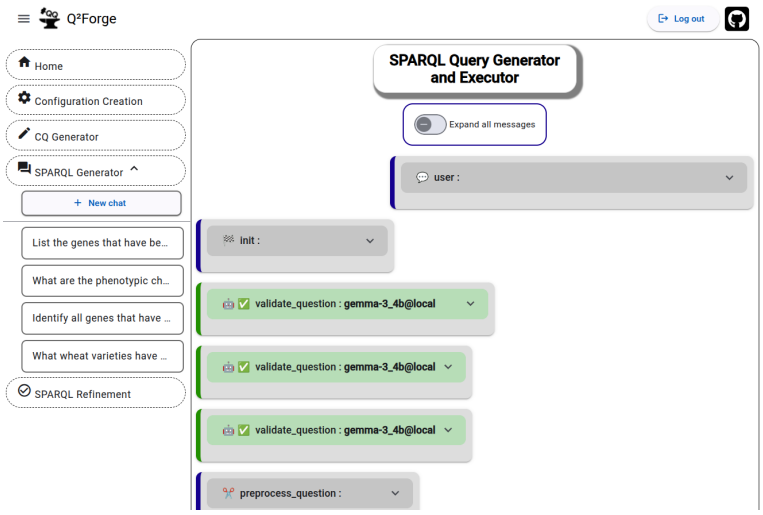

Figure 2: The SPARQL query generator/executor component of the web application showing: (1) the multi-user support; (2) the current chat thread; (3) the past chat history.

- Instruct the LLM to judge whether the query reflects the CQ by providing a score ranging from 0 to 10 and a justification for the score. Note that, again, the LLM used is selected by the user.

With these functionalities, the user can execute and modify the query, extract the QN and FQN, and submit the question-query pair with the augmented context to an LLM for evaluation. This process can be repeated until a satisfactory (semantically valid) query has been crafted. The user can then add the pair to a dataset, and later on export the dataset in various formats to use it in other workflows.

New users who use either the online or local version have access to the D2KAB KG. Once selected, the form for generating the CQs is automatically pre-filled.

3. Application Use Cases

This section presents two on-going use cases where Q^{2}Forge has proven highly beneficial. First, for users unfamiliar with SPARQL, Q^{2}Forge was tested on the IDSM KG in the chemistry and metabolomics domain. Pharmaceutical researchers and metabolomics data scientists used it to generate diverse,

KG	#triple	#cls	#ppt	(1)	(2)	(3)	(4)
D2KAB	27,093,602	590	287	10	132.4	45.6	29.2
IDSM	36,285,192,866	226,809	1,044	2,592.5	146.2	51	40.1

Table 1
IDSM and D2KAB: statistics and time requirements in seconds for Q^{2}Forge steps to execute.

domain-relevant question-query pairs. These included queries related to drug discovery, structure-activity relationships, and biochemical pathway analysis. The tool helped lower technical barriers and showed strong potential for educational use in cheminformatics and bioinformatics labs. Second, Q^{2}Forge was applied to the *Wheat Genomics Scientific Literature Knowledge Graph* [8] from the D2KAB project, which lacked well-defined CQs.[5] Here, it assisted domain experts by automatically generating meaningful CQs and corresponding SPARQL queries. After minor refinements, the generated pairs were valid and aligned well with expert intent.

Our experiments on the IDSM and D2KAB KGs allowed us to determine the time required to execute each stage of the pipeline. Table 1 summarises statistics or each KG, as well as the time taken to (1) compute classes' text embeddings, (2) generate 50 CQs, (3) answer one CQ, (4) extract QNs and FQs and obtain one refinement proposal. The experiments were performed using: an Intel Core Ultra 9 185H × 22 CPU with 64GB of RAM and an NVIDIA RTX 2000 Ada Generation Laptop GPU (8GB). We used `nomic-embed-text` embedding model and the FAISS vector store for the embedding task, and DeepSeek-v3 seq2seq model for all LLM calls. These early experiments suggest that Q^{2}Forge can effectively support both novice users in querying KGs and experts in documenting them, with potential for broader reuse pending further evaluation.

4. Enabling the "Agentification" of Q^{2}Forge Through MCP

Q^{2}Forge is designed as a graphical user interface (GUI) composed of tabs, buttons, and forms for humans to interact. Under the hood, the delivered functions invoke and orchestrate the services of a REST API. This common approach allows seamless integration of these services into third-party workflows. However the emerging *Agentic AI* paradigm radically changes the way applications collaborate, leveraging natural language as the main interface to and between applications. For instance, MCP[6] is an ongoing effort to simplify integration and facilitate collaboration between AI agents and external systems. This reframes the application's functionalities into a standardized, machine-understandable, NL-based interface (by contrast with a regular programmatic interface). Allowing users to "talk to the application" effectively moves common GUI interactions to the realm of natural language. More generally, converting GUI logic into a protocol layer designed for LLMs makes it possible for them to carry out complex functions by dynamically orchestrating multiple services.

We are currently implementing MCP in Q^{2}Forge to expose the REST API services as MCP services. In addition to the visual affordances, the webapp now describes its available services (create a user, activate a KG configuration, ask for the creation of CQs and SPARQL queries etc.) so that an AI agent can dynamically discover and orchestrate them.

Figure 3 illustrates our current experimentation with the *Dive AI* MCP application.[7] The user first asks for the list of KG configurations currently defined in Q^{2}Forge. The selected model reasons on the tools' descriptions it knows about, the interface then displays the tool invoked (`get_current_configurations`) and the response from the model. In a follow-up query, the user asks to generate 4 CQs for the "d2kab" KG configuration. In a first attempt, the model invokes the `generate_questions` service which fails because no configuration has been activated yet. It then retries by first activating the configuration and only then invoking the second service. This succeeds and

[5] https://github.com/Wimmics/WheatGenomicsSLKG/blob/main/SPARQLQueries-JupyterNotebook.ipynb
[6] https://modelcontextprotocol.io/
[7] https://github.com/OpenAgentPlatform/Dive

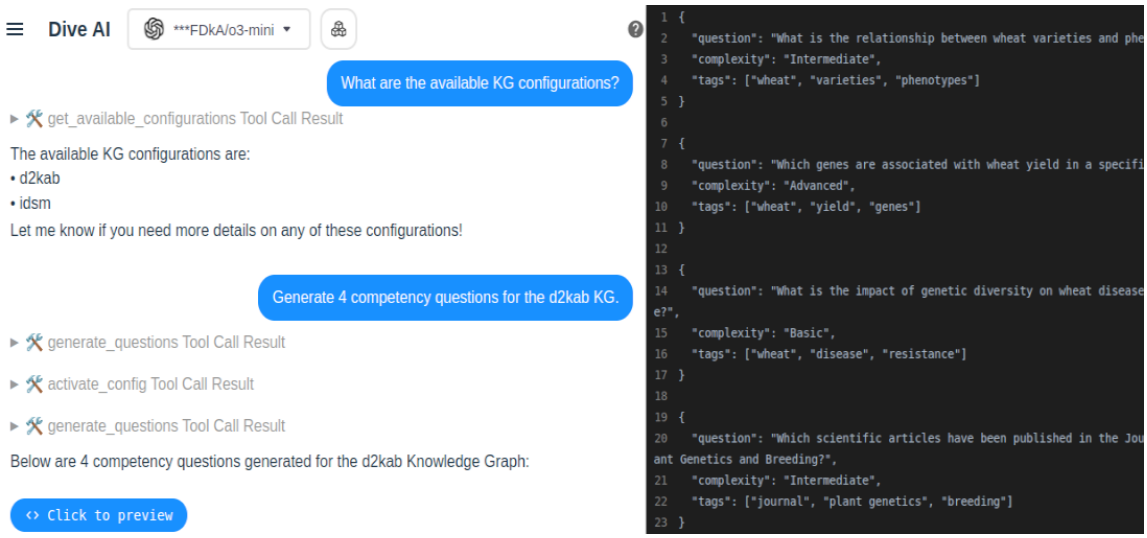

Figure 3: Invocation of the webapp's services through NL using an MCP server.

the generated CQs are displayed on the right-hand pane of the interface. Note that at each invocation, one can check the JSON-RPC trace by clicking on the tool name.

This simple experimentation illustrates two strong paradigm shifts. First, in terms of interaction: similarly to the popups that help users understand the meaning of GUI elements, MCP surfaces tool names and parameters in plain-text descriptors that agents can consume and choose from at runtime. Second, in terms of application design. Instead of hard-coding the workflows associated with clicks, developers define and describe tools exposed over MCP. They need to spend more time commenting precisely the tools so that an agent will know how to use them. Typically the first failed attempt in our experimentation points to a lack of comprehensive documentation of the tools. In this developer-to-agent shift, the loss of control over the workflow appears as a trade-off for the higher flexibility brought by the agent's reasoning capabilities, that make applications capable of fulfilling functions that were not hard-coded.

5. Conclusion

This paper presented Q^{2}Forge, a web application designed to address the challenge of creating question-query sets for any KG. The system introduces a pipeline that first generates CQs in NL, then translates these CQs into SPARQL queries, and finally assists users in refining the queries to export high-quality Q^{2}sets. These outputs can be used for benchmarking, training, and evaluating text-to-SPARQL models, as well as for documenting KGs. In addition to the web interface, we introduced an MCP server that encapsulates the services provided by the application. Our experiments demonstrated the potential for agentifying the process and interfacing with Q2Forge through natural language, paving the way for more flexible and automated workflows.

Future work will focus on several key directions. One area of emphasis is data quality evaluation, including the development of automated validation techniques and human evaluation protocols to assess the relevance and accuracy of generated question-query pairs. We also plan to explore the integration of Q2Forge MCP tools into broader workflows, particularly those requiring the conversion of natural language questions into SPARQL as part of their operations.

Finally, we recognize several limitations of the current system that we aim to address. These include the lack of support for follow-up questions, the absence of non-textual (e.g., visual) interpretations of SPARQL results, and the restriction to a single SPARQL endpoint, which hampers the generation of federated queries. Addressing these challenges will be key to further enhancing the flexibility and applicability of our approach.

Acknowledgments

This work was supported by the French government through the France 2030 investment plan managed by the National Research Agency (ANR), as part of the Initiative of Excellence Université Côte d'Azur (ANR-15-IDEX-01). Additional support came from French Government's France 2030 investment plan (ANR-22-CPJ2-0048-01), through 3IA Cote d'Azur (ANR-23-IACL-0001) as well as the MetaboLinkAI bilateral project (ANR-24-CE93-0012-01 and SNSF 10002786).

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT and DeepL for the following: Grammar and spelling checks. After using these tools/services, the author(s) reviewed and edited the content as needed, taking full responsibility for the publication's content.

References

[1] J. Frey, L.-P. Meyer, F. Brei, S. Gründer-Fahrer, M. Martin, Assessing the Evolution of LLM Capabilities for Knowledge Graph Engineering in 2023, 2025. doi:10.1007/978-3-031-78952-6_5.

[2] J. Lehmann, P. Gattogi, D. Bhandiwad, S. Ferré, S. Vahdati, Language Models as Controlled Natural Language Semantic Parsers for Knowledge Graph Question Answering, 2023. doi:10.3233/FAIA230411.

[3] B. P. Allen, P. T. Groth, Evaluating Class Membership Relations in Knowledge Graphs using Large Language Models, 2024. doi:10.48550/arXiv.2404.17000.

[4] R. Alharbi, V. Tamma, F. Grasso, T. R. Payne, The Role of Generative AI in Competency Question Retrofitting, in: The Semantic Web: ESWC 2024 Satellite Events: Hersonissos, Crete, Greece, May 26–30, 2024, Proceedings, Part I, Springer-Verlag, 2025, pp. 3–13. doi:10.1007/978-3-031-78952-6_1.

[5] F. Ciroku, J. de Berardinis, J. Kim, A. Meroño-Peñuela, V. Presutti, E. Simperl, RevOnt: Reverse Engineering of Competency Questions from Knowledge Graphs via Language Models, Web Semant. 82 (2024). doi:10.1016/j.websem.2024.100822.

[6] K. B. Cohen, J.-D. Kim, Evaluation of SPARQL Query Generation from Natural Language Questions, Proceedings of the Joint Workshop on NLP&LOD and SWAIE: Semantic Web, Linked Open Data and Information Extraction 2013 (2013) 3–7.

[7] D. Edge, T. Ha, C. Newman, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. Osazuwa Ness, J. Larson, From Local to Global: A Graph RAG Approach to Query-Focused Summarization, 2025. doi:10.48550/arXiv.2404.16130.

[8] N. Yacoubi Ayadi, S. Bernard, R. Bossy, M. Courtin, B. G. Happi Happi, P. Larmande, F. Michel, C. Nedellec, C. Roussey, C. Faron, A Unified approach to publish semantic annotations of agricultural documents as knowledge graphs, Smart Agricultural Technology 8 (2024) 43. URL: https://hal.science/hal-04495022. doi:10.1016/j.atech.2024.100484.