Future Generation Computer Systems 176 (2026) 108185

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

EXASAGE: The first data center operational data analysis assistant

Junaid Ahmed Khan *, Martin Molan, Andrea Bartolini

DEI Department, University of Bologna, Bologna, Italy

ARTICLE INFO

Keywords:
Data centers
Operational Data Analytics (ODA) monitoring frameworks
Large Language Models (LLMs)
Resource Description Language (RDF)
Knowledge Graph (KG)
Virtual Knowledge Graph (VKG)
Natural language query generation

ABSTRACT

Data centers increasingly rely on Operational Data Analytics (ODA) for real-time insights from vast streams of telemetry data. They typically utilize NoSQL databases for scalability and to handle diverse data types, which results in unstructured data representations and poses significant challenges for data retrieval and interoperability. Indeed, the lack of standardization, combined with schema flexibility and complex data structures, makes it difficult for system administrators to write and execute queries, ultimately complicating the automation of data retrieval tasks. Pre-trained Large Language Models (LLMs), with their latent knowledge, promise a ready-to-use AI-driven data interoperability layer, enabling data retrieval through natural language input. However, they often generate inaccurate or hallucinated query code when handling heterogeneous data sources and complex data structures. In this paper, we present EXASAGE, the first operational data analysis assistant for ODA, which leverages a Knowledge Graph (KG)-based approach to provide an AI-driven interoperable layer that addresses LLM limitations and simplifies data retrieval tasks in data center facilities through a prototype implementation. EXASAGE employs an LLM-based query generator to convert natural language into SPARQL queries (native to KGs), executed at a graph database endpoint, along with a virtual KG approach that dynamically generates KGs with only the data relevant to the user's input query, significantly reducing the storage overhead associated with a fully materialized KG in such large-scale telemetry systems. In evaluations on 1000 user input queries, EXASAGE achieved a 93.6 % accuracy in generating correct SPARQL code and retrieving correct answers, significantly outperforming the 25 % accuracy of NoSQL/SQLite queries, which frequently exhibited severe hallucinations. Furthermore, SPARQL queries were generally more concise and demonstrate shorter inference and execution times compared to NoSQL/SQLite queries. The average end-to-end time for a single execution cycle was 12.77s, making it suitable for interactive, non-critical operational data analysis tasks. The maximum observed storage overhead across all generated virtual KGs was just 52.62 MiB.

1. Introduction

The uptake of generative Artificial Intelligence (AI) in the mass market is driving an unprecedented computational demand, surpassing traditional computational challenges by several orders of magnitude and growing at a rate of 4-5x annually [1]. While data centers and high-performance computing (HPC) systems were once a niche focus for economic competitiveness, scientific leadership, and national security, they have now become the backbone of the AI-driven economy, acting as the "shovels" of the generative AI gold rush.

McKinsey's 2023 report highlights that industry leaders, on average, allocate 20 % of their digital budgets to AI technologies across various sectors, with this percentage expected to rise in the coming years [2]. As AI investment and significance grow, so does the need for robust hosting infrastructure. The rising demand for computationally intensive AI models has directly led to significant investment in HPC capabilities. Datacenter Dynamics reports that AI-driven HPC investments reached $36 billion USD in 2023, with 2024 projections indicating a 24.4 % increase [3]. This surge in demand and investment intensifies the challenges of efficient and sustainable HPC operations.

Today's best practices are based on Internet of Things (IoT) holistic monitoring frameworks, which convey real-time telemetry data from the different constituting layers and components (ICT operation, facility, and user-support) into a centralized database for dashboard and post-mortem data-driven analysis [4,5]. In 2020, authors of [6] reported that 9 of the largest supercomputing centers worldwide were deploying alongside their supercomputing infrastructure holistic monitoring frameworks based on IoT and big data technologies to efficiently operate the HPC clusters - in the field known as Operational Data Analytics (ODA) Frameworks [7]. Among all interviewed sites there was consensus on the overall architecture composed of distributed IoT sensors and data publishers, a message bus, a NoSQL database, and Grafana dashboards for visualization. Furthermore, while researchers have been focusing on applying AI and

* Corresponding author.

https://doi.org/10.1016/j.future.2025.108185
Received 15 January 2025; Received in revised form 3 June 2025; Accepted 29 September 2025
Available online 14 October 2025

Machine Leaning (ML) to the collected telemetry data [4], their usage is currently limited to real-time data visualization with customized dashboards [5] with a recent attempt to build augmented reality (AR) visualizations [8].

So far, the direct utilization of the telemetry data by stakeholders (system administrators, facility managers, user support teams, policymakers, and end users) is hindered by the lack of structure in the NoSQL database, which requires not only expertise in the HPC domain but also an intimate knowledge of the architecture of the monitoring system itself [9] – thus the defacto limiting their usage to pre-defined dashboards and AR visualizations. We must underline that this is an intrinsic issue of ODA and in general of IoT monitoring frameworks that rely on NoSQL databases, which becomes more prominent with the growing number and diversity of sensors and components. There is a clear need for a data interoperability layer that facilitates the knowledge extraction hiding the domain knowledge from the end user.

In [10], Khan et al. propose a Knowledge Graph (KG) for ODA where sensor readings were manually transferred from an IoT time-series database into a triplestore (GraphDB) according to a schema defined by an ODA-specific ontology. While the data ingestion process requires manual effort, the KG representation facilitates simpler data queries by leveraging the explicit schema provided by the ontology and the expressive power of the SPARQL query language. In contrast, traditional NoSQL approaches for ODA often require multiple sub-queries and manual integration across different data sources to answer complex questions. This process can be time-consuming and typically demands significant domain expertise to write and coordinate these queries effectively. The KG's ontology explicitly models these relationships between data elements, which can help simplify query formulation and reduce the number of manual steps needed. However, users still need some familiarity with ontology concepts and SPARQL to fully exploit these benefits.

Given the remaining challenges in writing and executing SPARQL queries even with a well-defined KG and ontology, a key question emerges: Can we enhance user accessibility and query efficiency by combining generative AI and pre-trained Large Language Models (LLMs) with KG data representations to create an AI-driven query assistant for ODA systems?

Motivated by this, we propose a framework that: (i) maps the heterogeneous ODA monitoring data into a unified graph representation, and (ii) leverages an LLM "informed" by the KG ontology to automatically generate and execute SPARQL queries from natural language input. This AI-driven assistant serves as an intuitive interface layer within existing ODA infrastructures, reducing the expertise barrier, enhancing the effectiveness and responsiveness of data retrieval processes, and fostering interoperability through consistent, schema-aware access to diverse data sources via the shared logical data model of the ontology used within the proposed framework.

1.1. Contributions

In this paper, we propose EXASAGE, the first operational data analysis assistant for ODA in data centers. EXASAGE is a Large Language Model (LLM)-based tool that provides an AI-driven interoperable layer for on-demand access to data collected at HPC facilities, generating graph database query codes to support timely, non-critical operational analysis. This paper presents the following innovations:

- We extend the standard architecture of ODA systems in data centers and HPC environments by introducing an AI-driven data interoperability layer that employs an LLM as a tool for query generation. This framework functions as an on-demand data access assistant that generates graph database query codes. The proposed ODA ontology-an extension of [10]-facilitates interoperability and serves as the knowledge base for the LLM.
- We evaluated EXASAGE against direct LLM use for NoSQL/SQLite query generation-the standard language for ODA and IoT monitoring-using a set of standard day-to-day end-user queries from a survey, augmented with random ones. EXASAGE achieved 93.6 % accuracy in query code generation, higher than the 92.96 % human accuracy reported in [11], while the NoSQL/SQLite approach reached only 25 % accuracy over 1000 queries.
- We demonstrate the impracticality of storing a complete Knowledge Graph (KG) for the time-series data of ODA, which incurs a 745× storage overhead (see Table 3). To address this limitation, we proposed a Virtual Knowledge Graph (VKG) generator that dynamically populates the graph database with only the data relevant to a user's input prompt. The maximum storage overhead observed across 1000 generated VKGs was only 52.62 MiB, demonstrating the scalability of the approach.

All code and resources for this study are available in the project's Git repository, which can be found at the following link: https://gitlab.com/ecs-lab/exasage

2. Related work

Despite recent advancements in generative AI, LLMs still exhibit some crucial limitations that are inherent to the technology itself: 1) LLMs are susceptible to hallucinations and serving of wrong or made-up information, and 2) they are limited to providing information that they were trained on. To contrast these limitations, researchers are focusing on: 1) contrasting hallucinations by grounding the reasoning abilities of the LLMs with a KG and 2) connecting the LLM models with the relevant information base.

For grounding an LLM with a KG, an ontology is essential to define the KG's structure, including its vocabularies and relationships. Since our primary objective is efficient ODA for data centers, the first step is to develop an ontology tailored for data centers. Thus in this section, we will begin by reviewing existing ontologies for data centers. Following this, we will discuss the two state-of-the-art approaches for (1) contrasting hallucinations and (2) connecting LLMs with an information base. Finally, we will outline recent advancements in LLM-based query generation techniques.

2.1. Ontologies for data centers

The Resource Description Framework (RDF) is an industry standard for developing ontologies and constructing KGs. RDF organizes data into triples, consisting of a subject (resource), a predicate (relation), and an object (either another resource or a value). These triples create a graph structure where subjects and objects act as nodes and predicates serve as the edges connecting them. An ontology provides a formal data representation, utilizing this triple pattern to capture complex relationships between data points, which enables the KG to store domain-specific knowledge effectively. These KGs can be queried using SPARQL, an RDF-specific querying language. SPARQL's triple-pattern matching enables precise query formulation, allowing users to easily leverage intricate relationships within the graph and perform complex queries that would otherwise require domain expertise.

Existing ontologies for data centers generally fall into two main categories: component-driven and data-driven approaches. The majority of currently available ontologies are component-driven, heavily emphasizing cataloging data center infrastructure and inventory rather than supporting dynamic data querying for telemetry information. Oscar Corcho et al. [12] highlight the lack of a comprehensive, unified data center ontology. They propose the DevOpsInfra ontology, modeling high-level ICT infrastructure entities-configurations, items, resources, and resource groups-and their relationships by integrating data from configuration and IT service management systems. Similarly, the proposed NORIA-O ontology [13] models network infrastructure components, network activities, and maintenance operations, focusing on

event tracking and incident management. Both ontologies are primarily component-centric, emphasizing static infrastructure elements and their relationships. While NORIA-O incorporates the SSN Ontology [14], its focus remains on event tracking, which differs from our objectives. In contrast, our use case targets ODA, centered on dynamic telemetry data from data center operations; thus, its logical data model (ontology) must be constructed around time-series metrics and real-time monitoring data. Gabriel G. Castañé et al. [15] propose an ontology integrating HPC and cloud systems; however, its emphasis on HPC-cloud relationships and inadequacy to model the underlying telemetry properly limit its utility for our use case. Similarly, Liao et al. [16] introduce an ontology intended to apply FAIR (Findable, Accessible, Interoperable, Reusable) principles to training datasets and AI models on heterogeneous supercomputers, which does not align directly with our focus on telemetry data of the data center facility itself. Furthermore, Kousha et al. [17] developed an HPC ontology for job script submission and AI tool integration but did not address telemetry data access. Likewise, Tuovinen et al. [18] presented an HPC ontology aimed at creating a unified query framework for various time-series storage solutions, but their focus is on storage structuring rather than on telemetry data querying.

Some works adopt a data-driven approach [10,19]. The ontology in [19], designed specifically for the SURF supercomputing facility. While it integrates high-level metadata and low-level operational metrics to support their ODA framework, its strong coupling to facility-specific data and infrastructure has limited its adoption and generalizability to other data center environments. In contrast, the ontology in [10], developed by the authors and extended in this work, models essential hardware and software components with fewer classes and properties. Although evaluated on a limited set of queries, it provides a more manageable and extensible foundation for data center ODA.

2.2. Grounding LLMs with KGs

LLMs have shown high adaptability and zero-shot performance in various downstream tasks and real-world applications, such as question-answering and text generation. The limitation of those models shared across all their variations is that they cannot return grounded knowledge. Grounded knowledge is factual, real-world information that is both accurate and reliable [20]. Instead, LLMs are, by their design, capable of generating plausible-sounding (or most probable) text, but they have no guarantees of providing factual information. As the LLMs are trained in a next-token prediction task on large amounts of human-generated data, they can give coherent and plausible-sounding responses that might be misleading and incorrect. This phenomenon in LLMs is sometimes called LLM hallucination [21].

While this behavior may be acceptable and even expected in certain applications, there are scenarios where explicitly encoding critical information becomes imperative. Recent research identifies KGs as the most promising method for storing essential factual information [22,23]. This structured storage approach enables LLMs to directly utilize the encoded information [20].

2.3. Retrieval Augmented Generation (RAG) and tool use

Besides hallucinations, another limitation of LLMs is their ability to provide current and up-to-date information. Without extensions, LLMs can only generate responses based on their training set. Updating the training set is model re-training, which is expensive and cannot be performed at regular intervals because of its significant costs [21].

Various strategies have been proposed to integrate the LLMs with external databases where additional information can easily be added. Depending on the structure of the data, the external database integration can be a part of a Retrieval-Augmented Generation (RAG) or integrated as an SQL integration.

The general structure of the RAG system consists of the connection between an LLM and a vector database. Each database entry (usually a section of a text document) is passed to the LLM embedding layers. In a vector database, the embedding of the document is stored alongside the document itself. After each query is created, a query embedding is compared to the vector database entries, and relevant database entries are selected. These entries are then passed as context to the LLM model [24].

RAG has been implemented across various applications and has allowed the LLMs to access and take textual documents into context. The RAG structure, however, does not provide an answer for other types of data information. For instance, integrating the LLM with a high-velocity, real-time monitoring system - such as the one employed at contemporary HPC systems [9] - requires the ability of the LLM to process large quantities of numerical data as well as interact with the existing software infrastructure correctly. The ability of the LLM to interface with the existing software infrastructure is called the LLM tool use [25].

Integrated with the LLMs, the tools extend the model's capabilities beyond language manipulation. Examples of integrated tools are mathematical functions, multimodal tools, and database query systems. The challenges associated with the tool use are tool integration (the connectivity layer between the LLM and the tool itself) and the training techniques (how to communicate the information about the tool used to the LLM itself) [25].

2.4. LLM and query generation

Several studies have explored the application of LLMs in generating SQL and SPARQL queries. This sub-section reviews related works that explore how LLMs are applied to these tasks, highlighting key methods and challenges.

2.4.1. SQL Generation

Generating accurate SQL query code remains a significant challenge in natural language processing (NLP). This difficulty stems from the inherent complexity of user queries and the requirement for a deep understanding of the underlying database schema. Advances in machine learning (ML) and artificial intelligence (AI), particularly through deep neural networks, have shown considerable promise in addressing these challenges. Pre-trained language models have also demonstrated strong capabilities in SQL generation. However, as modern databases become increasingly complex-due to large-scale datasets and the integration of data from advanced systems such as data centers-the limitations of relying solely on pre-trained language models for SQL generation have become more evident.

Recent advancements and the broad industry adoption of LLMs have revealed their substantial capabilities in understanding natural language. As a result, LLMs are now extensively studied for their potential in SQL generation. This implementation generally involves three key steps: (1) processing the natural language user prompt to capture its underlying semantics; (2) providing the LLM with the database schema to enable accurate mapping of prompt components to database elements; and (3) generating SQL code by combining insights from the user prompt and database schema [26].

Several strategies have been explored to enhance SQL generation accuracy, ranging from in-context learning, where the LLM is given context with either zero-shot [27] or few-shot [28] approaches-both of which have been shown to improve accuracy, to the more advanced chain-of-thought (CoT) prompting technique. CoT involves a step-by-step reasoning process that guides LLMs through structured deduction, encouraging more accurate and logical responses [29]. Each of these methods has demonstrated significant improvements in SQL generation performance.

Despite these advances in SQL generation using LLMs, their performance still falls short in practical real-world applications. Jinyang Li et al. [11] found that even state-of-the-art LLMs, like GPT-4, achieve only around 54.89 % accuracy in SQL generation when tested in real-world settings, a notable gap compared to human accuracy at 92.96 %.

2.4.2. SPARQL Generation

The integration of LLM with KG empowers knowledge-driven applications, with a natural extension being the generation of SPARQL queries by LLM, as SPARQL is a core technology for accessing KGs. This raises the question: what are the capabilities of LLMs in generating SPARQL queries from natural language input prompts?. In this subsection, we outline various research efforts on SPARQL query generation.

In [30] and [31], the authors use GPT-variants to generate SPARQL queries. Specifically, in [30], the focus is on improving entity linking for more accurate query generation, but they achieve only 62.7 % accuracy with a three-shot approach, using just three examples. In contrast, Kovriguina et al. [31] uses GPT-3 employing a one-shot approach with relevant context provided to the LLM. They tested their approach on three datasets, achieving an F1-macro of 67.07 on one KG, which leaves room for improvement, while the scores for the other two KGs were 28.75 and 15.01, indicating challenges with the generalization of their approach.

Furthermore, in [32], Meyer et al. assess the capabilities of state-of-the-art LLMs, including GPT, Gemini, and Claude, using a set of openly available KG benchmarking datasets. Their evaluation focuses on tasks such as Text-to-SPARQL(T2S), SPARQL-to-Answer(S2A), SPARQL-Syntax-Fixing(SSF) and Text-to-Answer(T2A). The study reveals that while LLMs excel at fixing SPARQL syntax errors with minimal difficulty, they struggle with generating correct *SELECT* statements, especially in complex queries. Similarly, in [33], the authors also explore ChatGPT's ability to translate natural language into SPARQL (T2S). Their framework, called Auto-KGQAGPT, utilizes the strategy of using a selected fragment of the KG as input to an LLM, significantly reducing the number of tokens. They evaluated their framework on 15 questions, achieving an accuracy of 93 % in correctly generating SPARQL queries using GPT-4. While the evaluation demonstrated promising results, it was conducted on a small test set with a much smaller KG (121 nodes and 183 edges), compared to the much larger KG for ODA data, which can contain upto billions of nodes and edges.

In [34], Li et al. address the challenge of declining result quality in real-world scenarios where there are limited high-quality annotated data. To tackle this, they employ fine-tuning of lightweight SPARQL generation models, where they first generate a training set of synthetic data transforming templated SPARQL queries into natural language questions using LLMs. Additionally in [35], the authors present a scholarly Knowledge Graph Question and Answering (KGQA) to generate SPARQL queries using few-shot training method to achieve high accuracy with an F1-score of 99 % on SciQA - one of the Scholarly-QALD-23 challenge benchmarks [36]. Their method employs a RAG strategy, where relevant question-SPARQL pairs are retrieved and used to prompt an LLM for SPARQL generation. They used Vicuna-13B5 - an open-source LLM, a descendent of Llama [37] by Meta AI. While their approach achieves high accuracy in SPARQL generation, it is confined to the scholarly domain, where the data is organized and structured, and the questions are domain-specific. In contrast, in ODA, the data is heterogeneous and unstructured, presenting a greater challenge and necessitating more intricate SPARQL queries. Additionally, ODA requires real-time data access, further complicating the task.

LLMs have shown promising results in understanding natural language questions using prompting strategies such as zero-shot and few-shot prompting [38–41]. In zero-shot prompting, the model performs tasks without prior examples, relying solely on the task description, whereas few-shot prompting provides a small number of examples alongside the task description to guide the model. In this paper, we aim to evaluate the effectiveness of these prompting strategies for generating SPARQL queries for data center facilities as an additional interoperable layer to address complex ODA queries.

The remainder of this paper is organized as follows: Section 3 introduces the proposed EXASAGE framework; Section 4 presents the experimental results; and Section 5 concludes the paper.

3. EXASAGE Framework

Fig. 1 presents the proposed EXASAGE framework. The end-user submits their query as a natural language input prompt, which is first sent to the *(1) Input Validator*. This component analyzes the user input using a rule-based approach to extract relevant entities according to the end-user's intent, and then verifies the validity of these entities against a set of ten predefined rules. If the input is determined to be invalid, the system returns an error notifying the user of the reasons for the invalid input. Conversely, if the input is valid, the system proceeds with the following downstream processes concurrently:

1. Generation of the SPARQL query that retrieves the answer to the user's query, using the *(2) LLM-Query Generator*, which combines the natural language input prompt with the *(3) ODA Ontology* and a set of few-shot examples as context. The generated SPARQL query is then passed to the *(5) Query Refinement* component for the query refinement process, where minor syntactic and logical corrections are applied based on common errors observed in LLM-generated queries;
2. The *(4) Virtual Knowledge Graph (VKG) Generator* takes as input the extracted entities from the *(1) Input Validator* to construct a virtual KG containing only the data necessary to answer the user's query, referred to as the *VKG*. This VKG is then sent to the *(6) Graph Database* to be stored and made available for query execution.

Following the successful completion of the two concurrent preprocessing steps, the refined query is sent to the SPARQL endpoint of the *(6) Graph Database* for execution. The *(7) Query Validator* then inspects the response from the endpoint. If no errors are detected, the resulting answer is returned to the end-user. However, if an error is encountered, the *Query Validator* triggers a retry mechanism, prompting the *(2) LLM-Query Generator* to regenerate the SPARQL query from scratch. The regenerated query undergoes the *(5) Query Refinement* process and is subsequently resubmitted for execution. This validation and regeneration cycle is repeated for up to three attempts to resolve the error and produce a valid, executable query.

3.1. Input Validator

The Input Validator contains two algorithm implementations: (1) *Entities Extraction* from the user input prompt, and (2) *Input Validation*. This component determines whether the user input prompt is valid according to system requirements and checks that all necessary information is provided by the user to enable successful execution of downstream tasks.

Algorithm 1 shows how to extract key entities from a the user input prompt. These entities belong to specific categories defined by the proposed ODA ontology (detailed in Section 3.3), including node, rack, job, metric, plugin, and time markers (start and end time). Since telemetry data is time-based, the user input prompt may mention time intervals directly. The algorithm starts by converting the question to lowercase and creating a dictionary to hold the extracted entities. Each category in the dictionary has two keys: present (to indicate if it appears in the question) and value (to store its content). The algorithm goes through each category, checks if it appears in the question using regular expressions, and updates the dictionary accordingly. If node mappings are available, a helper function can standardize node names based on the facility's naming rules. In the end, the algorithm returns a dictionary with all the found entities.

Once the entities are identified, the output from Algorithm 1 is passed to the second Algorithm 2 within the *Input Validator*. Algorithm 2 performs a structured validation of the extracted entities against a set of ten predefined rules to determine their suitability for further processing. It systematically evaluates the presence and completeness of key entity categories-including metrics, jobs (with or without identifiers), temporal intervals, nodes, racks, and plugins-prioritizing valid combinations

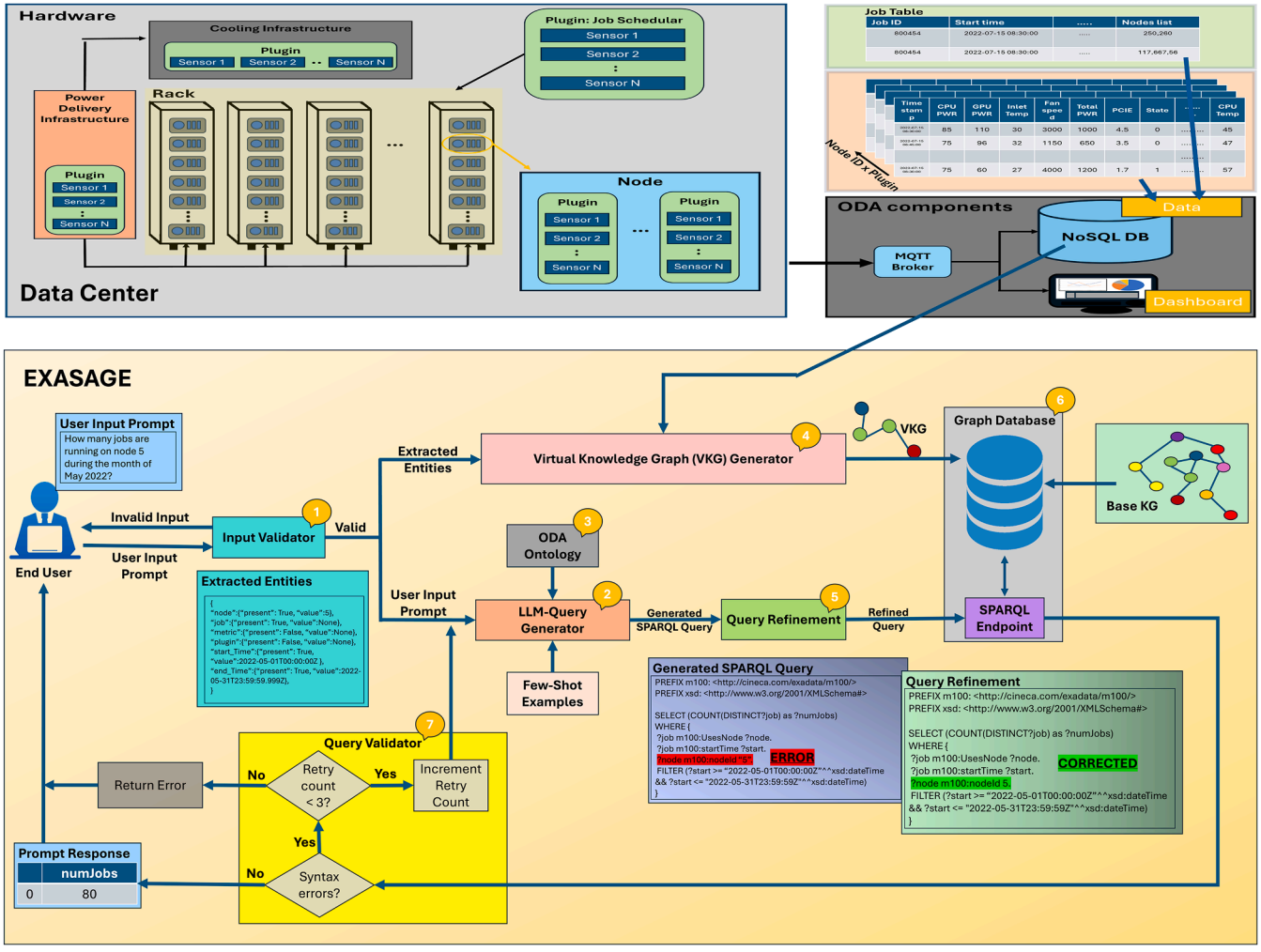


Fig. 1. EXASAGE: The first data center operational data analysis assistant, shown as a block diagram.

Algorithm 1: Entity extraction.

Input: *user_input_prompt*, *plugin_metric_map*, optional *node_mappings*
Output: *entities* dictionary with detected entity categories and values

Convert *user_prompt* to lowercase;
Define categories: {`node, rack, job, metric, plugin, start_time, end_time`};
Initialize *entities*[*category*] with `present = False, value = None`;
foreach *category in categories* **do**
  if `category` *keyword detected in user_input_prompt* **then**
    entities[*category*][*present*] ← `True`;
    `val` ← extract category value using pattern matching;
    if *val found* **then**
      entities[*category*][*value*] ← `val`;
      if *category* = `node` *and node_mappings provided* **then**
        Map node using *node_mappings*;
      if *category* = `metric` **then**
        `plugin` ← match plugin and metric from *plugin_metric_map*;
        entities[*plugin*][*present*] ← `True`;
        entities[*plugin*][*value*] ← `plugin`;
return *entities*;

that enable meaningful interpretation of the user input. The algorithm outputs a boolean flag indicating the validity of the user input prompt, alongside a detailed validation report in the form of a dictionary. This report either confirms successful validation with explanatory messages or provides explicit error codes and diagnostic information identifying missing or inconsistent elements.

3.2. LLM-query generator

This component uses an LLM model to generate SPARQL query code corresponding to the user input prompt. It takes the user query in natural language and combines it with the structure of the virtual KG, represented by the ODA ontology in Turtle (.ttl) format (detailed in Section 3.3), along with a set of six few-shot examples as additional context. The ODA ontology grounds the LLM with factual information as the logical data model, and the few-shot examples enable the LLM to learn effective data retrieval strategies for a few specific types of ODA complex query. The text snippet below illustrates the format used for the few-shot examples.

```
Prompt: <user input query as natural language>

SPARQL Query:
PREFIX .........

SELECT ?..
WHERE {
....
}
```

The complete input prompt to the LLM for query generation task is provided in the text snippet below.

Algorithm 2: Input validation.

Input: *entities* dictionary
Output: `(is_valid, validation_report)`
Extract presence flags: *has_metric*, *has_job*, *job_id*, *has_start*, *has_end*, *has_node*, *has_rack*, *has_plugin*;
$TIME\_NOTE \leftarrow$ " Times must follow ISO 8601 UTC format (e.g., 2024-06-01T12:00:00Z)."
if *has_plugin and only has_plugin present* **then**
  return `True, { "message": "Valid: plugin only." }`
if *(has_node or has_rack) and only these present* **then**
  return `True, { "message": "Valid: node/rack only." }`
if *has_metric* **then**
  if *has_start and has_end* **then**
    return `True, { "message": "Valid: metric with full time range." }` + $TIME\_NOTE$ }
  if *has_job and job_id* $\neq$ *None* **then**
    return `True, { "message": "Valid: metric with job ID." }`
  return `False, { "error_code": "metric_missing_job_and_time", "message": "Invalid: metric needs job or full time." }` + $TIME\_NOTE$, `"details": { "missing_fields": missing of "job", "start time", "end time" } }`
if *has_job* **then**
  if *job_id* $\neq$ *None* **then**
    return `True, { "message": "Valid: job with ID." }`
  if *has_start and has_end* **then**
    return `True, { "message": "Valid: job reference with full time."` + $TIME\_NOTE$ }
  return `False, { "error_code": "job_missing_id_and_time", "message": "Invalid: job missing ID and time."` + $TIME\_NOTE$, `"details": { "missing_fields": missing of "job ID", "start time", "end time" } }`
if *no entities present* **then**
  return `False, { "error_code": "no_entities_present", "message": "Invalid: no recognizable entities found."` + $TIME\_NOTE$ }
return `False, { "error_code": "unrecognized_combination", "message": "Invalid input: unrecognized combination."` + $TIME\_NOTE$, `"details": { key: val.present for` *key*, *val* `in` *entities* `} }`

```
You are a HPC center engineer. Your HPC center has a knowledge graphbased on the schema set by the ontology. Your task is to write SPARQL queries of the prompts that are provided to you. Utilize the ontology provided as context for accuracy. Follow all the logical paths set by the ontology between different classes and use proper syntax and use the proper xsd type set by the ontology for each property. Do not exceed the maximum tokens limit when providing response.

Ontology:
<KG in .ttl format>
## **Some example prompt implementations**
Example 1:
...
...
Example 6:
...
Sensor Metadata:
<Text which includes the types of sensors located in the facility>

Using the ontology, example query implementations, and sensor metadata, please write a SPARQL query to address the following prompt:
**Prompt:**
{user_input_prompt}
**SPARQL Query:**
```

3.3. ODA ontology

Fig. 2 presents the ODA ontology proposed within the EXASAGE framework. This ontology is based on the RDF framework and was developed through a data-driven, methodical design process. It serves as the schema for constructing the ODA KG, providing a formal and comprehensive representation of data collected within a data center facility. The ontology comprises 8 classes, 8 object properties defining relationships among these classes, and 17 data properties describing their attributes. It encapsulates the specialized vocabulary of ODA and the specific knowledge base of a data center, covering the key hardware and software components and their interactions.

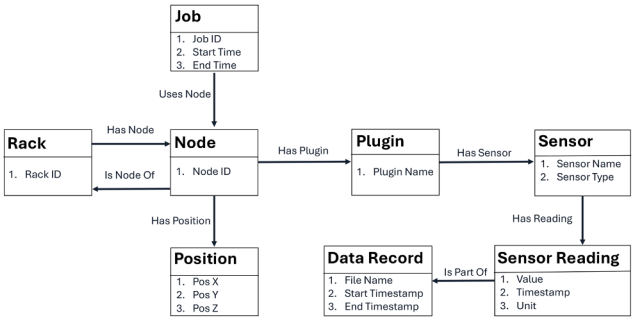


Fig. 2. ODA ontology with its classes, and properties.

3.3.1. Ontology design process

The ontology design followed an iterative methodology inspired by the *Ontology Development 101* guide by Noy and McGuinness [42]. We began by defining the domain and scope through a set of competency questions (see Table 1), which represent a set of common use cases and queries expected from the end users, primarily facility managers and engineers. These competency questions were developed collaboratively with a panel of four domain experts, including data engineers and managers from CINECA[1], Italy's largest supercomputing center, as well as specialists involved in developing the ODA monitoring framework currently deployed there.

We reviewed several existing ontologies related to HPC and data center infrastructures, including DevOps-Infra [12], NORIA-O [13], mOSAIC [15], and SEAS [43]. While these ontologies provide useful models of component relationships and infrastructure layouts, they do not adequately address the telemetry of ODA within data centers, which is the primary focus of this work. For example, a central component within a data center is the submitted job, which is executed on selected compute nodes according to the facility's resource allocation policy. This feature is missing in the existing ontologies. Since no existing ontology fully met our requirements, we decided to develop a new operational data-driven one specifically tailored to ODA, with improving interoperability with established vocabularies planned as future work.

We enumerated key domain concepts such as *Rack*, *Node*, *Position*, *Plugin*, *Sensor*, *Sensor Reading*, *Data Record*, and *Job*. Using these key terms, we defined the core classes of the ontology. For each class, we defined object and data properties to express relationships and attributes. For instance, a *Rack* has a `hasNode` property linking it to *Node* instances, and an identifying data property of `rackId`. Finally, we assigned domain and range constraints to all properties to formalize semantics and ensure ontology consistency.

3.4. Virtual Knowledge Graph (VKG) generator

The scale of the telemetry data collected already at a medium-sized data center facility can reach a million unique sensors with a sampling frequency of around 20s for every unique sensor [9], making it impractical to convert the entire time-series data into RDF for storage. We tested this approach and found it resulted in more than a 745x increase in storage size compared to a NoSQL representation (see Table 3). Therefore, we needed a solution that would allow us to leverage the benefits of the KG approach without sacrificing practicality in terms of storage size.

There exists a practical solution known as the **virtualization of KGs)**. In this approach, a virtual RDF graph is created on top of existing

[1] https://www.cineca.it/it

Algorithm 3: Virtual Knowledge Graph (VKG) Generator.

Input: *entities*, optional *node_mappings*
Output: VKG with RDF triples

$key\_entities \leftarrow \{e \mid entities[e][present] = \texttt{True}\}$;
if $key\_entities \subseteq \{rack,\ node\}$ **then**
  return;
if $job \in key\_entities$ **then**
  if $entities[job][value] \neq None$ **then**
    $jobId \leftarrow entities[job][value]$;
    Retrieve job data for $jobId$ from ODA NoSQL DB;
  else
    $startTime \leftarrow entities[start\_time][value]$;
    $endTime \leftarrow entities[end\_time][value]$;
    Retrieve job data in range $[startTime, endTime]$ from ODA NoSQL DB;
  Add RDF triples to VKG for job attributes ($jobId$, $startTime$, $endTime$);
  if *node_mappings exists* **then**
    Apply to node names;
  Add job-node relationships to VKG;
if $metric \in key\_entities$ **then**
  $metric \leftarrow entities[metric][value]$;
  $plugin \leftarrow entities[plugin][value]$;
  if $start\_time,\ end\_time \in key\_entities$ **then**
    $startTime \leftarrow entities[start\_time][value]$;
    $endTime \leftarrow entities[end\_time][value]$;
  else
    Use job's time range as $startTime$, $endTime$;
  if $entities[node][value] \neq None$ **then**
    $node \leftarrow entities[node][value]$;
    Retrieve metric data for $node$ in $[startTime, endTime]$ from ODA NoSQL DB;
  else
    Retrieve all metric data in $[startTime, endTime]$ from ODA NoSQL DB;
  if *node_mappings exists* **then**
    Apply to node names;
  foreach $node$ **do**
    Add RDF triples to VKG for node, plugin, and sensor;
  foreach $reading$ **do**
    Add RDF triples: ($value$, $timestamp$, $unit$) linked to corresponding sensor;
return VKG;

data sources, eliminating the need for data duplication or extensive storage requirements [44]. This method enables the integration and querying of heterogeneous data sources, while offering the flexibility and scalability required to handle the large volumes of data typically found in data center environments. While tools such as *ontopic.ai*[2] offer an accessible, no-code environment for KG construction and support virtual-KG implementations, they-along with similar platforms-currently focus on connecting to relational databases and other structured data sources (e.g., PostgreSQL, MySQL, Oracle). However, they do not natively support direct connections to NoSQL databases, which represent the preferred storage solution in ODA due to their scalability and schema flexibility [6]. As a result, we implemented a custom algorithm for VKG generation, specifically tailored to the characteristics and requirements of our ODA use case.

This component uses virtualization to generate a virtual knowledge graph (*VKG*) containing only the data needed to answer a user query. The VKG generator takes as input the entities extracted by the *(1) Input Validator* using Algorithm 1 and applies Algorithm 3 to dynamically construct the relevant *VKG*. Algorithm 3 presents the pseudocode for this dynamic VKG construction. The algorithm initiates by extracting key entities flagged as present from the output of Algorithm 1. In the absence of such entities, the algorithm terminates with an error. When only topological entities such as racks or nodes are present, VKG generation is not required, as queries of this nature can be answered directly via the *Base-KG* (see Section 3.6). For job-related entities, the algorithm retrieves job-specific data from the ODA DB connection using templated query codes populated with values extracted from the input *entities* dictionary, leveraging either a job identifier or a defined temporal interval based on the available information. It then constructs RDF triples to represent the job and establishes semantic links to associated compute nodes, applying facility-specific node mappings when applicable. In cases where metric entities are present, the algorithm first verifies the existence of a valid time range before acquiring sensor readings. These readings are retrieved from the ODA DB connection using the same templated approach, either for specified nodes or for all nodes if none are designated. RDF triples encoding the sensor data are generated and linked to the relevant plugins and sensors. Finally, the fully constructed VKG is returned as the output of the process.

3.5. Query refinement

Given the inherent randomness in the LLM's generation process, there is a risk of producing inaccuracies, commonly referred to as hallucinations. This issue is discussed in the related works section, and specifically in Section 2.4.2 regarding SPARQL query generation, where grounding the LLM with a KG is proposed as a mitigation strategy. However, hallucinations can still occur when generating SPARQL queries with an LLM. This was also the case in our work: upon analyzing the errors of LLM-generated SPARQL queries (see Table 5), we identified six common error patterns that could be effectively corrected using Python's regex matching: (1) numeric literals were often written as quoted strings or included incorrect datatype annotations, (2) triples with the `m100:HasNode` property had incorrect domain and range assignments-the domain should be *Rack* and the range *Node*, but queries used arbitrary variables that did not reflect this, (3) triples with `m100:IsNodeOf` had wrong domain and range, where the domain should be *Node* and the range *Rack*, yet inconsistent variables were used, (4) triples with `m100:HasSensor` appeared without the corresponding `m100:HasPlugin` triples, causing incomplete relationships according to the data model in the proposed ontology. (5)`HasRack` property used which doesn't exist in the proposed ontology, (6) variables in the `SELECT` clause were separated by commas, such as in `SELECT ?var1, ?var2, ?var3`‘, which can lead to syntax errors.

Algorithm 4 presents the pseudocode for the query refinement component, that addresses these six common identified wrong patterns. It applies a series of targeted transformations to the generated SPARQL query using Python's regex matching. The process begins by normaliz-

Algorithm 4: Query Refinement.

Input: `Generated SPARQL query`
Output: `Refined query`

Apply regex substitution on `Generated SPARQL query` to replace numeric literals written as quoted strings or with incorrect datatype annotations;
Replace all occurrences in `Generated SPARQL query` of `m100:HasNode` triples with incorrect domain/range by `?rack m100:HasNode ?node`;
Replace all occurrences in `Generated SPARQL query` of `m100:IsNodeOf` triples with inconsistent domain/range by `?node m100:IsNodeOf ?rack`;
if *`Generated SPARQL query` contains a `m100:HasSensor` triple* **then**
  Replace it in `Generated SPARQL query` with `?plugin m100:HasSensor ?sensor`;
  if *`Generated SPARQL query` does not already contain a `m100:HasPlugin` triple* **then**
    Insert `?node m100:HasPlugin ?plugin` before the `HasSensor` line in `Generated SPARQL query`;
Replace all occurrences of the invalid `HasRack` property in `Generated SPARQL query` with `?node m100:IsNodeOf ?rack`;
Identify the `SELECT` clause in `Generated SPARQL query` and remove any commas between variables;
Return the modified `Generated SPARQL query` as the `Refined query`;

[2] https://ontopic.ai/en/

ing numeric literals, replacing quoted strings and incorrect datatype annotations with plain numeric values. Next, it corrects improper domain and range usage for the `m100:HasNode` and `m100:IsNodeOf` properties by enforcing the correct subject-object variable structure consistent with the ontology. The algorithm also rewrites any `m100:HasSensor` triple to ensure the subject is a plugin and, if a corresponding `m100:HasPlugin` triple is missing, inserts it to be coherent with the ontology. Furthermore, any use of the nonexistent `HasRack` property is replaced with a valid `m100:IsNodeOf` triple. Finally, the `SELECT` clause is normalized by removing commas between variables to conform to SPARQL syntax. The output of this algorithm is returned as the `Refined query`.

3.6. Graph database: Triplestore for RDF data

The preferred storage backend for KG is a triplestore[3]-a specialized database designed to store and manage RDF triples. There are two types of data in a data center facility: (1) temporal data, comprising telemetry streams collected from sensors, and (2) static data, which captures the structural and spatial configuration of the facility. This includes the layout of system racks, the positioning of compute nodes, installed components and sensor metadata.

While queries involving temporal data are handled by the *(4) VKG Generator* component of the proposed framework, additional support is needed for queries concerning static data-for example, “What sensors are installed on compute node XYZ¿‘. To address this, we introduce the *Base-KG*, a pre-loaded knowledge graph containing static metadata about the data center’s topology and components. In the proposed ODA ontology (see Section 3.3), the classes `Rack`, `Node`, `Position`, and `Sensor` provide the concepts required to model this static metadata.

For standalone deployment, the selected triplestore must operate as a self-contained service. We chose GraphDB[4] by Ontotext, which supports large-scale data ingestion, offering advanced reasoning capabilities, and provides a built-in SPARQL endpoint-making it a suitable backend for the requirements of our framework.

3.7. Query validator

To ensure the generation and execution of syntactically valid SPARQL queries produced by the language model, the *(7) Query Validator* component serves as the final stage of the EXASAGE framework. Once the refined query is submitted to the SPARQL endpoint of the *(6) Graph Database* for execution, the *(7) Query Validator* inspects the response for syntax errors. If the query executes successfully, the result is returned directly to the user. In the event of a syntax error, the component initiates a bounded retry loop, redirecting control back to the *(2) LLM-Query Generator* to regenerate the SPARQL query from scratch. Each regeneration uses the original natural language user input along with the full ontology and few-shot examples in the prompt context. This process is repeated up to a maximum of three times, with each attempt treated as a fresh generation rather than an incremental correction. If all attempts fail, the system halts and returns an error message prompting the user to revise or clarify their query. This mechanism ensures reliability and robustness while explicitly avoiding infinite loops when repeated attempts fail to correct the underlying errors.

4. Experimental results

In this section we evaluate the performance of EXASAGE framework by (i) assessing the correctness of the output query code and the query result for each user input prompt, (ii) the execution time for the different EXASAGE components (LLM inference, VKG generation, and query execution at SPARQL endpoint, (iii) the VKG storage cost, and (iv) an input and output token analysis to further evaluate the proposed framework’s efficiency with different prompting strategy.

4.1. Experimental setup

All the experimental results reported in this paper have been conducted in the following computing resources: (1) We used the Examon ODA framework deployed at CINECA during the production of the Marconi100 supercomputer (M100). Examon runs in the CINECA Ada Cloud resources and uses Cassandra as a NoSQL database, with KairosDB as the time-series database extension. (2) We run the Graph database in a server featuring an Intel Xeon E5-2630 v3 CPU with an x86_64 architecture comprising 16 physical cores across two sockets. It includes a 256 KiB L1 data cache, a 256 KiB L1 instruction cache, a 2 MiB L2 cache, and a 20 MiB L3 cache, operating with a base frequency of 2.40 GHz and a maximum frequency of 3.20 GHz. (3) We run all the remaining EXASAGE components in a system featuring an Intel Xeon 8358 CPU featuring 32 cores operating at 2.6 GHz. It is equipped with 512 GB of DDR4 RAM running at 3200 MHz and an NVIDIA Ampere A100 GPU with 64 GB of HBM2 memory.

For the *(2) LLM-Query Generator*, we used the Meta Llama-3 8B parameter pre-trained model running on-premises, employing the instruct model variant. The evaluation criterion assigns a value of "1" if the model returns a correct answer to the user input prompt, even if the response includes additional redundant but related information. If the model returns an incorrect answer or triggers an error at the SPARQL endpoint of the *(6) Graph Database*, it is assigned a value of "0". Accuracy of the framework is calculated as the percentage of correct responses over the total number of queries. This assessment was performed through manual inspection of all generated SPARQL query codes as well as verification of the retrieved answers from the *(6) Graph Database*.

4.1.1. Baseline: NoSQL/SQLite query generation

To evaluate the proposed EXASAGE framework, we compare it with the current standard practices in data centers, specifically the use of NoSQL databases for scalability and flexibility in handling diverse data sources [6]. This comparison involves assessing the KG approach with data access through SPARQL generation, against the generation of state-of-the-art NoSQL query codes. These NoSQL queries, often based on SQLite, adopt SQL-like structures (e.g., *SELECT*, *WHERE*, and *DESCRIBE*), but are tailored for NoSQL databases. Henceforth, we will denote these as “*NoSQL/SQLite query*” in the remaining manuscript and they will represent the collective data center ODA query access strategies. Furthermore, given that the telemetry data collected in data centers and IoT systems are inherently time-series, these NoSQL/SQLite queries often include specifications for time-periods to support focused analysis. We further denote the start and end of these time-periods with the keywords “*TSTART*” and “*TSTOP*” respectively.

4.1.2. LLM-query generator prompting strategy

To evaluate the capabilities of our chosen pre-trained LLM for query generation and determine the most effective prompting strategy, we implemented two approaches-zero-shot and few-shot learning-applied to both SPARQL and NoSQL/SQLite query generation. In zero-shot evaluation, we test the LLM’s ability to generate complex ODA queries without additional training, assessing its pre-existing query generation capabilities. This allows us to compare its performance against few-shot learning, where the LLM is provided with contextual examples to improve the accuracy of generated queries.

The contextual information provided to the LLM for each type of generation includes the following:

- SPARQL query generation: In the zero-shot approach, only the ODA ontology (see Section 3.3) in ‘.ttl‘ format, which provides the LLM with the underlying structure of the KG and its specific vocabularies, is used. While in the few-shot approach, a set of six example implementations would be added with the ODA ontology in the context.

[3] https://en.wikipedia.org/wiki/Triplestore
[4] https://www.ontotext.com/products/graphdb/

Table 1
ODA input query archetypes.

No.	Prompt
1	Give me all the nodes present in rack X
2	Give me a list of all the racks
3	Give me the position of node X
4	Give me the list of plugins
5	What nodes were used by the job XXXXXXXX?
6	What is the average power used by the job XXXXXXXX?
7	How many jobs are running on node X during the month of May 2022?
8	What is the min, max and avg temperature of node X when it is in use during the month of May 2022?
9	Give me a list of sensors which are of type "power"
10	Give me a list of the jobs running and the nodes they used during the month of May 2022

- NoSQL/SQLite query generation: Since the data at a data center facility is unstructured, there is no formal description available for direct use. To ensure a fair comparison, we compiled a textual description of key concepts relevant to the data center. This includes specific instructions addressing different aspects of complex ODA queries, such as the format for writing NoSQL/SQLite query codes tailored to the Examon instance deployed at the CINECA. In the zero-shot approach, only this textual description is included in the context. While in the few-shot learning approach, a set of five example implementations would be added alongside the textual description in the context.

4.1.3. User input queries

To characterize the performance of the different LLM-Query Generator configurations, we use two different sets of user-defined queries: (i) ODA input query archetypes (Table 1)- obtained by a survey conducted on end users, specifically system administrators, user support managers and facility managers with a progressive increase in complexity; (ii) randomly generated queries, obtained by providing to OpenAI ChatGPT model the list of all different metrics collected by M100 Examon and the ODA input query archetypes and to incorporate the different metrics' names into an arbitrary number of input prompts similar to these archetypes. For experimentation, we used the month of May 2022, as the quality of telemetry data during this period was observed to be high [9]. In all queries where time intervals were not explicitly specified, we defaulted to using the start and end timestamps of May 2022. We conducted the tests using a total of one thousand queries, categorized as follows:

- **Spatial information** The first **203 prompts** are spatial information queries and cannot be implemented in NoSQL/SQLite queries, as such information is typically absent from current ODA monitoring frameworks. Therefore, they will be used exclusively for evaluating SPARQL queries. Their inclusion provides additional benefits, such as enabling graph neural network-based workflows that require spatial information to generate graph tensors. Specifically, the anomaly prediction framework for data centers which uses graph neural networks [45].
- **Job-related insights** The next **207 prompts** include queries related to the average, maximum, and minimum job power, job duration, and jobs resources utilization. These insights are essential for optimizing resource allocation, monitoring performance, and improving energy efficiency in data centers.
- **Mixture** The remaining **590 prompts** focus on sensor metrics, covering details about sensors on a node, the frequency of specific metrics exceeding thresholds, and instances where a node surpasses a threshold. These threshold values were arbitrarily selected for all queries.

To assess the differences in phrasing among the randomly generated queries, we calculated the cosine similarity between the initial set of 10 prompts and the remaining 990. We utilized TfidfVectorizer[5] from the Scikit-learn library to transform the prompts into numerical vectors based on Term Frequency-Inverse Document Frequency (TF-IDF) weighting. The average cosine similarity between the initial ten ODA input query archetypes and the subsequent 990 randomly generated queries was found to be 0.069. This low average similarity indicates significant differences in prompt phrasing, despite the underlying prompt question potentially being similar.

4.2. Zero-shot vs few-shot

This experiment aims to determine which prompting strategy yields the highest accuracy in query generation.

We utilized the ODA input query archetypes (see Table 1) for this evaluation. Results are reported in Table 2, which demonstrates that SPARQL generation consistently outperforms NoSQL/SQLite query generation in all cases. In the few-shot with context scenario, SPARQL generation achieves a perfect score of 10 out of 10 correct prompts. An error analysis was conducted for each experiment to identify and categorize the errors observed across different prompts.

4.2.1. Zero-shot (with context)

In this configuration, SPARQL generation resulted in six correct prompts out of ten. However, errors were observed in the following prompts:

- Prompt 1: A minor syntax error occurred where an integer literal was incorrectly enclosed as a string.
- Prompts 6, 7, and 8: The LLM failed to follow the correct logical pathway as defined by the ontology (Node →Plugin →Sensor →Sensor Reading; see Section 3.3), instead jumping directly from Node to Sensor, leading to output answer errors.

In contrast, the NoSQL/SQLite query generation faced more significant issues. While prompts 4, 5, and 6 were correctly generated, the remaining prompts contained major errors:

- Prompt 7: The LLM used an incorrect timestamp format (NoSQL/SQLite query accepts "DD-MM-YYYY HH:MM:SS") and included an erroneous data fetch statement.
- Prompt 8: Logical errors were present along with the wrong timestamp format, and incorrect aggregation statements for minimum, maximum, and average values were generated.
- Prompt 9: Severe hallucinations occurred, with the LLM creating a non-existent data source.
- Prompt 10: Similar issues as prompts 8 and 9 were observed.

4.2.2. Few-shot (without context)

In this configuration, the LLM achieved 90 % accuracy for SPARQL generation. This outcome indicates that the LLM can comprehend the underlying structure of the KG based on the few-shot examples provided. Furthermore, the LLM successfully avoids the error of jumping directly from Node to Sensor Reading, adhering to the correct logical pathway identified in the zero-shot analysis. This improvement results from the LLM learning this explicit pathway from one of the few-shot examples. Only one prompt contained an error, which is detailed as follows:

- Prompt 9: The LLM hallucinated by introducing a non-existent entity into the generated query, leading to an incorrect response. This error occurred because, without access to the ontology, the LLM lacked grounding in the complete data structure of the ODA KG.

[5] https://scikitlearn.org/1.5/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

Table 2
Zero-shot vs. Few-shot Output Analysis. The dash (-) indicates that the prompt is not possible to implement in NoSQL/SQLite query. '1' denotes a correctly generated query code, while a '0' indicates an incorrectly generated query code.

Prompt	Zero-shot (with context)		Few-shot (without context)		Few-shot (with context)		Query execution time at endpoint	
	SPARQL	NoSQL query /SQLite	SPARQL	NoSQL query /SQLite	SPARQL	NoSQL query /SQLite	SPARQL	NoSQL query /SQLite
1	0	–	1	–	1	–	77.07 ms	-
2	1	–	1	–	1	–	67.13 ms	-
3	1	–	1	–	1	–	70.15 ms	-
4	1	1	1	0	1	1	47.76 ms	19.05 ms
5	1	1	1	1	1	1	82.33 ms	191.74 ms
6	0	1	1	1	1	1	47.03 ms	694 ms
7	0	0	1	1	1	1	90.21 ms	131 s
8	0	0	1	0	1	0	2.02 s	15120 s
9	1	0	0	0	1	0	33.86 ms	9.49 ms
10	1	0	1	0	1	0	5.53 s	133 s
Total correct	6	3	9	3	**10**	4		
% correct	60 %	43 %	90 %	43 %	**100 %**	57 %		

In contrast, for NoSQL/SQLite query generation, the LLM correctly generated responses for prompts 5, 6, and 7. However, it encountered issues in other prompts, including:

- Prompts 4 and 9: The LLM hallucinated by generating incorrect or non-existent data sources.
- Prompt 8: Incorrect data fetch statements and improper timestamp format were used.
- Prompt 10: Although the LLM correctly identified the data sources, it made a logical error by misclassifying nodes used by jobs as jobs, rather than accurately identifying them as nodes.

4.2.3. Few-shot (with context)

In this configuration, the LLM successfully generates all 10 prompts correctly for SPARQL. Since the complete ontology was provided as context to the LLM, it eliminates the error of introducing entities or data sources that do not exist in the ODA KG. In contrast, the NoSQL/SQLite query generation shows some improvement, but the final accuracy remains at only 57 %. While prompts 4, 5, 6, and 7 are correctly implemented, the LLM continues to make logical errors in the remaining prompts. Specifically:

- Prompt 8: The query should first access the job table to determine when the node was in use. However, the generated query incorrectly retrieves all sensor metrics for the specified time-period without filtering for job activity.
- Prompt 9: The generated query suffers from hallucinations, introducing non-existent entities into the response.
- Prompt 10: The generated query contains both an incorrect data access statement and an invalid key for a python language dictionary access.

4.2.4. Query execution time analysis

The query execution times reveal a significant disparity between SPARQL and NoSQL/SQLite query, with SPARQL achieving faster execution times across most queries. In contrast, NoSQL/SQLite query performance varies greatly, with some queries taking substantially longer to execute. For instance, Query 7 takes 163.67s, and Query 8 takes a staggering 15,120s in NoSQL/SQLite query, compared to just 90.21ms and 2.02s for SPARQL, respectively. Although the NoSQL/SQLite query outperforms SPARQL in the execution of queries 4 and 9, the difference is in the millisecond range and thus minimal. The longer duration for NoSQL/SQLite query is primarily due to the multiple data retrievals required from different data sources to produce the final output. In contrast, SPARQL benefits from the structured relationships defined by the ontology (see Section 3.3) within the KG, allowing it to navigate these data retrieval processes more efficiently. This results in significantly faster query responses, as illustrated in Table 2.

Table 3
Storage size comparison between NoSQL (Parquet-compressed) and ODA KG (Turtle) formats for IPMI plugin telemetry data collected in May 2022.

Metric	Value
Total data instances	11.37 billion
NoSQL DB (Parquet-compressed)	4.00 GiB
ODA Knowledge Graph (Turtle)	2.91 TiB
Turtle-to-Parquet size ratio	745×

4.2.5. Key takeaway

The analysis consistently demonstrates that SPARQL query generation outperforms NoSQL/SQLite query generation across all evaluated scenarios, achieving a perfect score in the few-shot analysis with context. This underscores the importance of grounding the LLM with an ontology, as evidenced by the reduced performance in the few-shot (without context) scenario. Furthermore, the results indicate that the LLM is better suited for generating SPARQL queries, underscoring the compatibility and effectiveness of integrating LLMs with KGs and SPARQL, as opposed to the NoSQL/SQLite approach. Finally, the key takeaway is that the LLM performs optimally in the few-shot scenario with context for both NoSQL/SQLite and SPARQL query generation. Therefore, this configuration is adopted for the proposed framework and subsequent evaluations.

4.3. Motivation for the adoption of virtualization of Knowledge Graphs (KG)

Table 3 compares the storage requirements for 31 days of IPMI plugin telemetry collected from the M100 system [9], contrasting a NoSQL-based representation using Parquet compression with a KG constructed using the ODA ontology (see Section 3.3) and serialized in Turtle (.ttl) format. During this period, a total of 11.37 billion data instances were recorded. While the Parquet-compressed NoSQL dataset occupies just 4.00 GiB, the fully materialized RDF graph requires 2.91 TiB of storage-a 745× increase. This dramatic growth highlights the impracticality of fully materializing RDF data for large-scale telemetry environments. Such excessive storage overheads motivated the adoption of a virtualization approach through the proposed *Virtual Knowledge Graph (VKG) Generator* (see Section 3.4). The VKG enables on-demand graph construction and scalable semantic integration, allowing efficient query execution without the need to persist the entire RDF dataset in storage.

4.4. Evaluation on randomly generated queries

Evaluating only 10 queries is insufficient to comprehensively demonstrate the proposed framework's performance. In real-world data cen-

Table 4
Accuracy of the *LLM-Query Generator*, *Query Refinement*, and *Query Validation* components in EXASAGE on randomly generated prompts. The table reports the number of total prompts, correct and wrong queries, and overall accuracy for each query type.

EXASAGE Component	Query Type	Total Prompts	Correct Queries	Wrong Queries	Accuracy [%]
LLM-Query Generator	NoSQL/ SQLite	797	199	598	25
	SPARQL	1000	723	277	72.3
Query Refinement	SPARQL	1000	907	93	90.7
Query Validation	SPARQL	1000	936	64	93.6

ters, hundreds or even thousands of queries are executed daily on telemetry. To better establish the framework's efficiency and accuracy, we extend the evaluation to a total of 1000 queries. The first 10 queries were analyzed in Section 4.2, while the remaining 990 queries, which were randomly generated as user inputs, are the focus of this section. For completeness, the results are reported collectively for all 1000 queries. Additional details about the randomly generated queries are provided in Section 4.1.3. Table 4 summarizes the accuracy results of the proposed framework.

4.4.1. LLM-query generator

Inspecting the NoSQL/SQLite query results revealed that the most accurate queries typically involved basic statistical operations, such as calculating the max, min, and avg of a sensor metric. This trend can likely be attributed to the inclusion of an example query about average job power in the few-shot context, which guided the LLM in generating these types of queries accurately. However, accuracy decreased when the input prompts were more complex-either not directly part of the few-shot examples or not covered by the ten ODA input query archetypes. These included both variations of the original archetypes and entirely new types of questions, such as "When did a node's temperature exceed a specified value of X?", "How many times did the metric XYZ exceed the threshold T?", "What is the duration of the job XYZ?", "When did node X power consumption exceed threshold T?", "When did node X temperature first exceed threshold T?", and "Identify racks with nodes being used by a job in specific time periods.". In such cases, the NoSQL/SQLite outputs often contained major errors, including incorrect data retrieval statements, formatting issues, and logical mistakes.

In contrast, SPARQL generation showed higher performance, with accuracy 2.89× that of NoSQL/SQLite query generation. Table 5 presents a classification of errors found in SPARQL queries generated by the *LLM-Query Generator*, divided into syntax, logical, and combined error types. Of the 277 total errors, the majority (234) were logical in nature. Among these, 188 involved incorrect references to entities or relationships in the ODA ontology-such as mismatches in node-to-rack, node-to-reading, or reading-to-sensor relations-indicating difficulties in mapping natural language queries to the underlying data model. Other logical issues included incorrect literal type assertions, missing or malformed timestamp filters, and the use of invalid namespaces for sensor identifiers. Syntax errors (18 in total) consisted of malformed FILTER expressions, improper placement of structural clauses such as GROUP BY, ORDER BY, or LIMIT within WHERE, and errors in the construction of the SELECT clause. An additional 25 errors involved a combination of both syntax and logical mistakes. Notably, the SPARQL generator was more robust to prompt variations due to the presence of the logical data model-i.e., the ODA ontology-in the LLM's context. This provided the model with semantic grounding, enabling it to generalize beyond fixed query templates and accurately interpret inputs that were semantically related but syntactically diverse.

Table 5
Statistical distribution of syntax and logical errors identified in SPARQL queries generated by the *LLM-Query Generator*.

Error Type	Number of Errors	Error Description
Syntax	18	1) Errors in FILTER statements. 2) Wrong use of GROUP BY / ORDER BY / LIMIT. 3) Errors in SELECT clause.
Logical	234	1) Incorrect entity or property references (e.g., wrong relations such as node-to-rack, node-to-reading, node-to-sensor, reading-to-sensor). 2) Incorrect literal type assertions. 3) Missing timestamp filter; missed filters. 4) Wrong namespaces for sensor names.
Syntax + Logical	25	Combination of both syntax and logical errors
Total Errors	277	

Table 6
Comparison of query validation approaches using LLMs with and without reinforcement learning-inspired prompting. The table shows the number of syntax errors identified, how many were corrected, remaining errors, and the percentage of errors successfully corrected.

Query Validation Approach	LLM Model	Syntax Errors	Errors Corrected	Errors Remaining	Error Correction [%]
Retry mechanism max 3 attempts	Llama 3 8B	43	29	14	67.4
Reinforcement learning-inspired prompting	Llama 3 8B	43	13	30	30.2
Reinforcement learning-inspired prompting	GPT-4o	43	19	24	44.2

4.4.2. Query refinement

The significant drop in accuracy for NoSQL/SQLite query generation, along with the major errors observed in the generated queries, indicates that minor adjustments are unlikely to produce substantial improvements. Although natural language processing techniques may correct some mistakes, the underlying logical issues suggest only limited gains in accuracy. In contrast, SPARQL-generated queries are more amenable to refinement, offering clearer opportunities to improve overall accuracy through a refinement process.

In the proposed framework, the SPARQL queries generated by the *LLM-Query Generator* are passed to the *Query Refinement* component, which applies corrections based on the most common errors identified in Table 5. This refinement process employs simple pattern matching and replacement, resulting in negligible time overhead-averaging less than 10 microseconds per query.

The *Query Refinement* component significantly enhanced the accuracy of the framework, increasing the number of correctly generated queries from 723 to 907 (see Table 4), corresponding to an 18.4 % improvement. Among the remaining 93 incorrect queries, 50 exhibited logical errors, such as missing filter statements, while 43 contained syntax errors, with 25 of these involving both syntax and logical issues.

4.4.3. Query validation

Table 6 presents a comparison of query validation techniques, focusing on syntax error correction performance. The methods evaluated include a retry mechanism with up to three attempts using the Llama 3 8B model, and a reinforcement learning-inspired prompting approach, which incorporates the faulty SPARQL query as a negative example within the prompt to guide correction. The reinforcement learning-

inspired approach was tested on both Llama 3 8B and GPT-4o models. The comparatively poor performance of LLama 3 8B compared to GPT-4o may be attributed to its smaller model size, which limits its ability to reason over errors, follow complex prompt structures, or generalize from negative examples. GPT-4o performed moderately better due to its stronger reasoning and instruction-following abilities. However, since logical errors still remain, the reinforcement learning-inspired prompting approach method alone is not enough to achieve high query accuracy.

This approach successfully corrected 29 out of 43 syntax errors, 25 of which also contained logical errors. While all syntax issues were resolved, 14 logical errors remained unresolved in those queries. The 29 corrected queries produced correct answers on the graph database, increasing the total number of correct queries to 936 out of 1000 (see Table 4). Interestingly, all syntax errors were resolved within three retry attempts: 35 were fixed on the first attempt, 7 required two attempts, and only one needed three attempts. Overall, this represents a 2.9 % improvement over the *Query Refinement* component and a 21.3 % improvement compared to the initial output of the *LLM-Query Generator*.

4.5. Performance evaluation

In this section, we provide a comprehensive performance evaluation of EXASAGE. This includes a comparison of input and output token counts, as well as LLM inference time using the *LLM-Query Generator* component, across SPARQL and NoSQL/SQLite query generation. These measures are used solely as performance metrics to evaluate the behavior and computational effort of the LLM during query generation, and do not imply equivalence in expressivity or vocabulary usage among these languages. Additionally, we assess other key performance aspects of EXASAGE, such as query execution time at the SPARQL endpoint of the *Graph Database*, the time required to generate the VKG using the *Virtual Knowledge Graph (VKG) Generator* component, and its characteristics, including the number of nodes, edges, and storage size in bytes. All measurements are conducted with the specific aim of quantifying the execution time of a single complete cycle of the proposed framework in response to one user input prompt, thereby providing a holistic view of its end-to-end performance.

Fig. 3 illustrates the distribution of data points for each of these performance metrics using violin plots[6]. Specifically, Subfigure 3a compares the key performance metrics between SPARQL and NoSQL/SQLite query generation by the *LLM-Query Generator*. Subfigure 3b highlights the additional time overheads associated with using the EXASAGE framework, while Subfigure 3c visualizes the distribution of VKG attributes. Each plot includes the median (represented as a white dot), the interquartile range (IQR) as a thick central bar, and whiskers showing the full data range. The violin's shape reflects the density of data points across values, offering insights into the distribution's characteristics, such as skewness and variability.

4.5.1. Input tokens comparison

The input tokens for SPARQL generation and NoSQL/SQLite query generation exhibit notable differences. For SPARQL generation, the input token count averages 3906.90, with a standard deviation of 13.24, ranging from a minimum of 3896 to a maximum of 3,936. In contrast, NoSQL/SQLite query generation demonstrates a significantly lower average of 2934.80 tokens, with a standard deviation of 14.33, and a range from 2921 to 2962.

Notably, the token counts for both SPARQL and NoSQL/SQLite query generation are similar across all randomly generated queries, with both exhibiting low standard deviations. This consistency is attributed to the fact that the majority of the token count is derived from the context and few-shot examples, which remain constant across all queries. The slight variation primarily stems from the actual text in the end user's input prompt.

4.5.2. Output tokens comparison

In SPARQL generation, the average output token count is 195.26, with a standard deviation of 64.81, ranging from a minimum of 66 to a maximum of 512. In contrast, NoSQL/SQLite query generation exhibits a higher average output token count of 350.01, with a standard deviation of 108.31, and a range from 65 to 512.

This demonstrates that SPARQL outputs are generally more concise; however, some variability exists, with longer outputs occurring less frequently, as indicated by the lighter tails of the plot. In contrast, the wider shape of the NoSQL/SQLite query plot reflects greater variability and higher output token counts, suggesting that its outputs are typically longer and more diverse compared to those of SPARQL generation. Additionally, the broader top of the NoSQL/SQLite distribution indicates a higher likelihood of encountering longer output tokens compared to SPARQL.

4.5.3. LLM Inference time comparison

The inference times for SPARQL generation average 11.09s, with a standard deviation of 2.57s, ranging from a minimum of 6.11s to a maximum of 26.22s. In contrast, NoSQL/SQLite query generation exhibits a higher average inference time of 16.09s, with a standard deviation of 4.20s, and a range from 5.02s to 23.54s.

The plot for inference time illustrates that, for SPARQL generation, the majority of inference times are concentrated around the median, reflecting a high degree of consistency in performance. In contrast, the plot for NoSQL/SQLite query generation shows a wider distribution, suggesting that inference times vary significantly, with some instances taking considerably longer than others. This variability may be attributed to the inherent complexity of generating NoSQL/SQLite queries, which could involve more diverse processing steps compared to the more structured nature of SPARQL generation.

4.5.4. VKG Generation time

The VKG generation time shows an average of 8.05s, with substantial variability indicated by a high standard deviation of 15.85s. The minimum generation time is 0.81s, while the maximum reaches 200.51s, suggesting occasional outliers. Most VKG generation times cluster between 1.22s and 9.30s, as indicated by the IQR.

4.5.5. VKG Storage cost

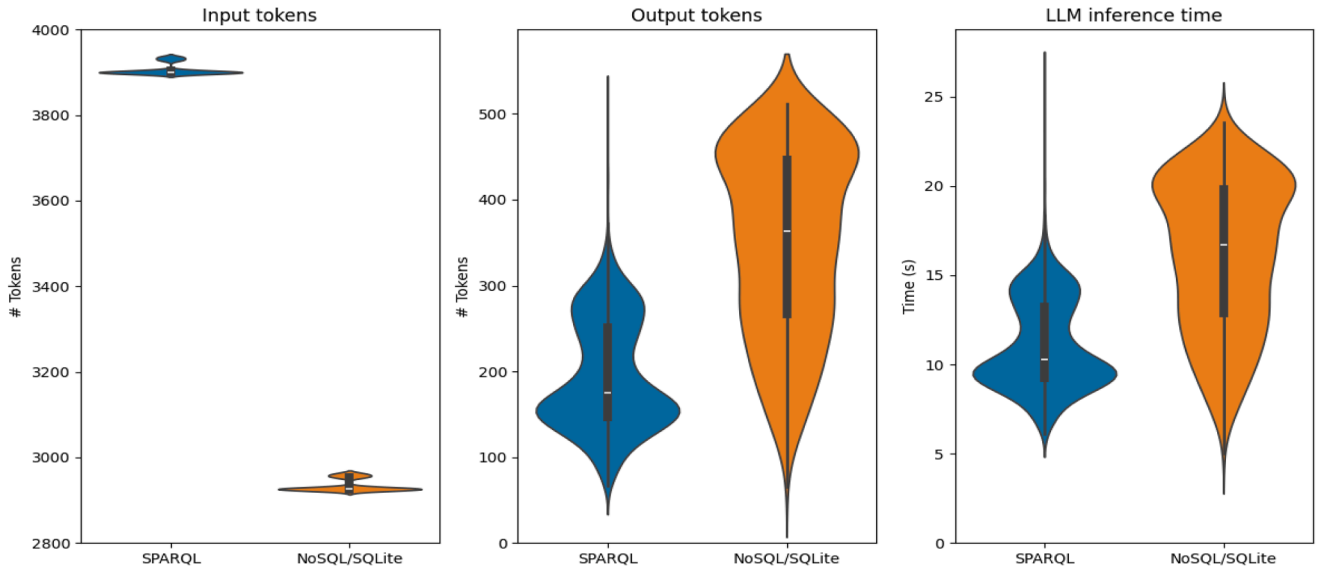
To determine the VKG storage cost, we calculated the size of all generated VKGs for the user input queries. The size is directly proportional to the number of nodes and edges of the VKG. On average, the VKGs contain 9758 nodes, with a high standard deviation of 63,395, indicating significant variation. The minimum number of nodes is 4, while the maximum reaches 750,851, suggesting the presence of outliers. The IQR reveals that 75 % of the data falls between 77.5 and 5,449.25 nodes, with a median of 2499 nodes.

The average number of edges is 17,623, with a standard deviation of 118,478, showing considerable variability. The minimum number of edges is 4, and the maximum is 13,770,197, reflecting occasional extremes. The 25th percentile is 149.25 edges, the 75th percentile is 8945 edges, and the median is 4,034.5 edges, indicating that most edge counts are concentrated in the lower ranges.

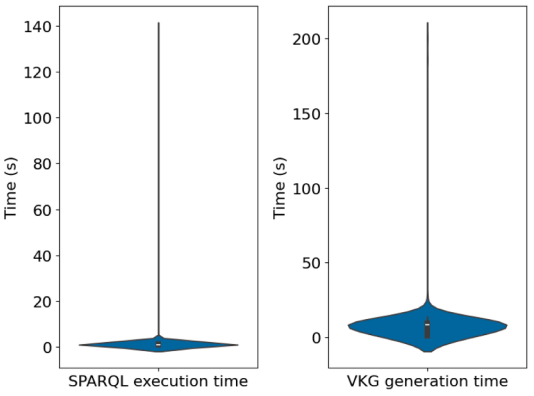
Finally, the mean storage size of the VKGs is 720,462 bytes, with a standard deviation of 4,807,915 bytes, which also suggests significant variation. The minimum size is 282 bytes, and the maximum size is 55,175,610 bytes, highlighting the presence of large outliers. The IQR indicates that 75 % of the sizes fall between 5,571.25 bytes and 365,151 bytes, with a median size of 161,197 bytes, showing that most sizes are concentrated in the lower to middle range.

Interestingly the maximum storage overhead, introduced by the creation of a VKG, among the tested queries is just 52.62 MiB - which makes

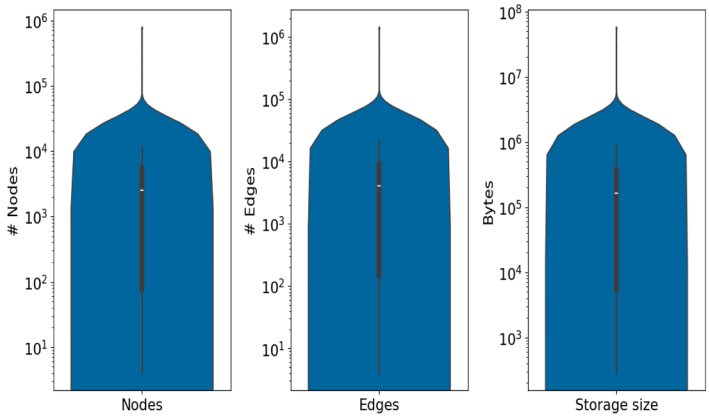
[6] https://en.wikipedia.org/wiki/Violin_plot



(a) Performance metrics comparison of LLM-Query Generator in SPARQL vs. NoSQL/SQLite query generation



(b) EXASAGE additional time overhead metrics: SPARQL execution at endpoint and VKG generation time



(c) Data distrbution of VKG attributes in Log scale

Fig. 3. Violin plots showing the distribution of performance metrics for SPARQL generation versus NoSQL/SQLite query, including input/output tokens, LLM inference time, query execution time for SPARQL, and Virtual Knowledge Graph (VKG) generation time as well as VKG attributes (#nodes, #edges and storage size).

it negligible compared to the extremely large storage overhead of a fully materialized ODA KG.

4.5.6. Query execution time at SPARQL endpoint

The query execution times at the endpoint show a mean of approximately 1.05s, with a standard deviation of 0.70s. Execution times range from a minimum of 0.29s to a maximum of 13.04s. The IQR indicates that 50 % of the queries are executed between approximately 0.80s and 1.16s. This narrow range suggests consistent performance for most queries. However, the presence of outliers, though less extreme than in broader distributions, indicates that a small number of queries may still take noticeably longer to complete.

4.5.7. Query validator time overhead

The *Query Validator* operates as a check that inspects the output of the SPARQL endpoint of the *Graph Database* for syntactic errors. In most cases (95.7 %), this check completes in (<10μs) and has a negligible impact on the overall execution cycle time. However, in rare failure cases (4.3 %), the *Query Validator* triggers a retry of the LLM inference for SPARQL generation using the *LLM-Query Generator* followed by *Query Refinement*, and finally a retry of the query execution at the SPARQL endpoint. Table 7 summarizes the retry distribution and associated overheads across 1000 randomly generated queries. On average, the retry mechanism incurs an overhead of 0.63s per query.

Table 7
Retry distribution and time overhead in *Query Validation*. The table reports the number of retry attempts, how many queries required that number of retries, the average time per retry attempt (in seconds), and the resulting total overhead (in seconds).

Retry Attempts	Number of Queries	Time per Retry Attempt [s]	Total Overhead [s]
1	35	12.14	424.9
2	7	24.28	169.96
3	1	36.42	36.42
Total	43	-	631.3

4.5.8. Single input execution cycle

Fig. 4 shows the time diagram for the average single input execution cycle of the proposed EXASAGE framework, based on the mean duration of the following processes: *LLM Query Generation and Query Refinement* (LLM-QG + QR), *VKG Generation*, VKG storage, query execution, and *Query Validation*. For clarity, the *Input Validator* is omitted from the timeline, as it incurs negligible latency (approximately 0.6ms

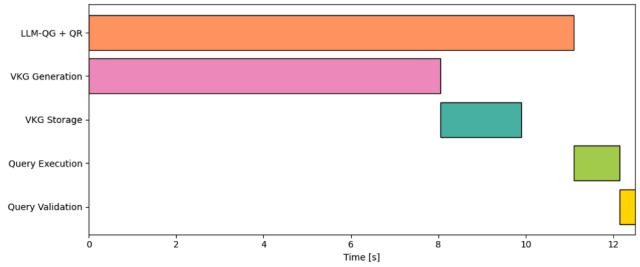


Fig. 4. Single input cycle execution - Time diagram for the proposed framework.

and only initiates the subsequent processes once the input is deemed valid. Similarly, the *Query Validator* typically requires (<10μs) to verify syntactically valid SPARQL queries, which constitute the majority of cases. However, in the rare instances (4.3 %) identified in Section 4.5.7, the *Query Validator* triggers retries that cumulatively add an average overhead of 0.63s per query across the evaluated queries. The average *Query Validator* time is highlighted in yellow in Fig. 4.

The execution cycle begins (assuming valid input) with the concurrent processes of LLM generation and query refinement, which take on average 11.09s, where the time for query refinement is also negligible (<10μs). The process of VKG generation takes on average 8.05s, and VKG storage on the *Graph Database* takes on average 1.85s, amounting to a total of 9.90s. Since the next stage depends on the completion of these two processes, query execution starts at 11.09s and finishes at 12.14s, taking 1.05s on average. Finally, to account for the rare cases of syntactic errors detected by the *Query Validator*, which add an average overhead of 0.63s per query, the total average single input execution cycle time reaches 12.77s.

5. Conclusion

In this paper, we propose EXASAGE, the first operational data analysis assistant for data centers that combines a knowledge graph (KG) approach with a pre-trained LLM to provide an AI-driven data interoperability layer over ODA monitoring data. EXASAGE supports on-demand access by using an LLM-based query generator to translate natural language into SPARQL queries, executed at a GraphDB endpoint. We evaluated EXASAGE on 1000 queries-including 10 ODA archetypes and 990 randomly generated prompts-against NoSQL/SQLite-based query generation. EXASAGE achieved 93.6 % accuracy, far outperforming NoSQL/SQLite's 25 %, which often suffered from hallucinations. SPARQL queries were also significantly more concise, with lower output token counts, and executed faster, despite requiring more input tokens. These results demonstrate that integrating LLMs with KGs based approaches enables accurate, efficient, and schema-aware data access from natural language-capabilities that are not achievable with LLMs and NoSQL/SQLite alone in complex heterogeneous environments, such as data centers and IoT monitoring systems.

Future Work. Future efforts will focus on enhancing interoperability with established vocabularies, either through extension modules or by integrating concepts from related ontologies such as DevOps-Infra [12], SEAS [43], and NORIA-O [13]. Additionally, we aim to reduce the end-to-end latency of the proposed framework to support real-time analysis for time-critical queries and to develop a broader set of queries to further evaluate and strengthen the framework's capabilities.

Extending EXASAGE to general IoT applications. EXASAGE enables natural language access to time-series operational data stored in NoSQL databases, demonstrated on data center use cases but applicable broadly across IoT. While relational databases support complex LLM-generated queries, LLMs struggle with NoSQL/SQLite queries in IoT. EXASAGE innovates by translating NoSQL/SQLite data into a (virtual) knowledge graph, allowing LLMs to generate more accurate, syntactically correct graph queries.

This translation approach is domain-agnostic, enabling EXASAGE's application across varied IoT fields. Adapting EXASAGE to a new domain requires developing a domain-specific ontology and, if needed, a VKG generator for high-frequency time-series Fig. 1.

The ontology, in RDF format, formalizes domain knowledge with tailored terminology. Updating the ontology allows easy adaptation of few-shot query examples, while Input Validator and VKG generator rules must be fine-tuned accordingly. Other core components remain unchanged, making EXASAGE a flexible, efficient tool for diverse IoT analytics.

CRediT authorship contribution statement

Junaid Ahmed Khan: Resources, Writing – original draft, Validation, Investigation, Data curation, Writing – review & editing, Visualization, Methodology, Formal analysis, Conceptualization; **Martin Molan:** Project administration, Writing – original draft, Methodology, Conceptualization, Writing – review & editing, Investigation; **Andrea Bartolini:** Resources, Investigation, Conceptualization, Writing – review & editing, Supervision, Project administration, Funding acquisition, Validation, Methodology.

Data availability

The manuscript contains a Git repository link that contains all the resources and data for viewing or reproducing the results.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This research was partly supported by EuroHPC JU SEANERGYS (g.a. 101177590), HE EU DECICE (g.a. 101092582), HE EU GraphMassivizer (g.a. 101093202), and Spoke "FutureHPC & BigData" of the ICSC-Centro Nazionale di Ricerca in "High Performance Computing, Big Data & Quantum Computing", funded by the EU - NextGenerationEU. We also thank CINECA for their collaboration and for providing access to their computing resources.

Appendix A. Query Complexity Analysis Experiment

To better understand the factors contributing to these variations in execution time, we define a weighted scoring model to evaluate the complexity of the SPARQL queries generated from our randomly generated input prompts. This model is based on five common query features observed in our dataset-projections, triple patterns, filter expressions, joins, and aggregations-which collectively capture the primary sources of computational effort during query evaluation.

Since not all features contribute equally to query complexity, we assign weights to each feature based on their relative computational cost, ordered from lowest to highest:

```
projections < triple patterns < filter expressions < joins
                                < aggregations
```

The overall complexity score $C(q)$ for a query q is computed as a linear combination of these weighted counts:

$$C(q) = 1 \cdot P_q + 2 \cdot T_q + 3 \cdot F_q + 4 \cdot J_q + 5 \cdot A_q$$

To evaluate whether this query complexity score corresponds to actual execution costs, we plotted execution time against the computed complexity scores. Fig. A.1 presents a scatter plot comparing the query

Table A.1
Weighted SPARQL query features and their computational justifications.

Feature	Weight	Technical Justification
Projections (P)	1	Minimal overhead; applied after evaluation
Triple Patterns (T)	2	Linear index scans; moderate cost
FILTERs (F)	3	Evaluated post-matching; increased CPU usage
Joins (J)	4	Intermediate result generation; high memory cost
Aggregations (A)	5	Requires grouping and materialization

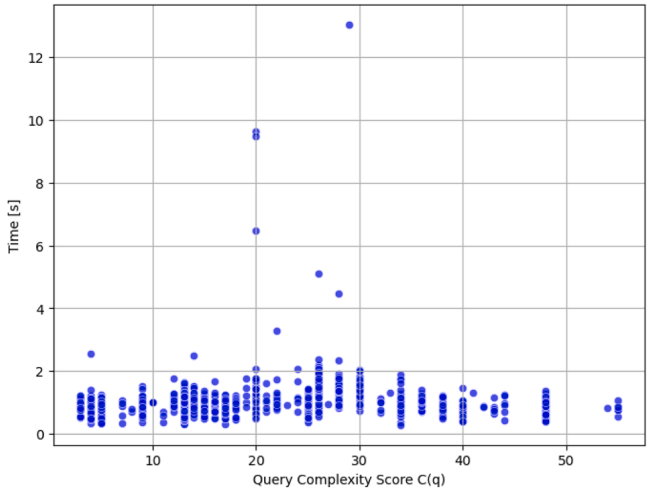


Fig. A.1. Scatter plot of query complexity score $C(q)$ vs. Query execution time for 1000 randomly generated queries.

complexity score $C(q)$ and query execution time for each of the 1000 randomly generated queries. The plot shows that execution time remains relatively stable across a wide range of complexity scores, with the majority of queries completing in under 2s. Although a few outliers-primarily within the mid-range complexity scores (20–30)-exhibit significantly longer execution times, no clear positive correlation is observed between the query complexity score $C(q)$ and query execution time.

References

[1] J. Sevilla, T. Besiroglu, B. Cottier, J. You, E. Roldán, P. Villalobos, E. Erdil, Can AI scaling continue through 2030?, 2024, (https://epochai.org/blog/can-ai-scaling-continue-through-2030). Epoch AI Research Report.

[2] McKinsey, The state of AI in 2023: generative AI's breakout year, 2023, (https://tinyurl.com/46dynvz6). McKinsey & Company.

[3] Georgia Butler, 2023 Second-largest investment year in decade for data centers - report, 2023, (https://tinyurl.com/hp74ndf9). DatacenterDynamics.

[4] A. Borghesi, A. Burrello, A. Bartolini, ExaMon-X: a predictive maintenance framework for automatic monitoring in industrial IoT systems, IEEE Internet Things J. 10 (4) (2023) 2995–3005. https://doi.org/10.1109/JIOT.2021.3125885.

[5] A. Netti, M. Ott, C. Guillen, et al., Operational data analytics in practice: experiences from design to deployment in production HPC environments, Parallel Comput. 113 (2022) 102950.

[6] M. Ott, W. Shin, et al., Global experiences with HPC operational data measurement, collection and analysis, in: 2020IEEE International Conference on Cluster Computing, 2020.

[7] A. Netti, W. Shin, M. Ott, T. Wilde, N. Bates, A conceptual framework for HPC operational data analytics, in: 2021IEEE International Conference on Cluster Computing (CLUSTER), 2021, pp. 596–603. https://doi.org/10.1109/Cluster48925.2021.00086

[8] J. Athavale, C. Bash, W. Brewer, M. Maiterth, D. Milojicic, H. Petty, S. Sarkar, D. Milojicic, Digital twins for data centers, Comput. 57 (10) (2024) 151–158. https://doi.org/10.1109/MC.2024.3436945

[9] A. Borghesi, C. Di Santi, M. Molan, et al., M100 ExaData: a data collection campaign on the CINECA's Marconi100 tier-0 supercomputer, Sci. Data 10 (2023) 288. https://doi.org/10.1038/s41597-023-02174-3

[10] J.A. Khan, M. Molan, M. Angelinelli, A. Bartolini, et al., ExaQuery: proving data structure to unstructured telemetry data in large-scale HPC, in: Companion of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE '24 Companion, Association for Computing Machinery, New York, NY, USA, 2024, pp. 127–134. https://doi.org/10.1145/3629527.3652898

[11] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K.C.C. Chang, F. Huang, R. Cheng, Y. Li, Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs, in: Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23, Curran Associates Inc., Red Hook, NY, USA, 2024.

[12] O. Corcho, D. Chaves-Fraga, J. Toledo, J. Arenas-Guerrero, C. Badenes-Olmedo, M. Wang, H. Peng, N. Burrett, J. Mora, P. Zhang, A high-level ontology network for ICT infrastructures, in: A. Hotho, E. Blomqvist, S. Dietze, A. Fokoue, Y. Ding, P. Barnaghi, A. Haller, M. Dragoni, H. Alani (Eds.), The Semantic Web – ISWC 2021, Springer International Publishing, Cham, 2021, pp. 446–462.

[13] L. Tailhardat, Y. Chabot, R. Troncy, NORIA-O: an ontology for anomaly detection and incident management in ICT systems, in: The Semantic Web: 21st International Conference, ESWC 2024, Hersonissos, Crete, Greece, May 26-30, 2024, Proceedings, Part II, Springer-Verlag, Berlin, Heidelberg, 2024, p. 21–39. https://doi.org/10.1007/978-3-031-60635-9_2

[14] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W.D. Kelsey, D. Le Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, K. Taylor, Ontology paper: the SSN ontology of the W3C semantic sensor network incubator group, Web Semant. 17 (2012) 25–32.

[15] G.G. Castañé, H. Xiong, D. Dong, J.P. Morrison, An ontology for heterogeneous resources management interoperability and HPC in the cloud, Future Gener. Comput. Syst. 88 (2018) 373–384. https://doi.org/10.1016/j.future.2018.05.086

[16] C. Liao, P.-H. Lin, G. Verma, T. Vanderbruggen, M. Emani, Z. Nan, X. Shen, HPC Ontology: towards a unified ontology for managing training datasets and AI models for high-performance computing, in: 2021IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC), 2021, pp. 69–80. https://doi.org/10.1109/MLHPC54614.2021.00012

[17] P. Kousha, V. Sathu, M. Lieber, H. Subramoni, D.K. Panda, Democratizing HPC access and use with knowledge graphs, in: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 243–251. https://doi.org/10.1145/3624062.3624094

[18] T. Lauri, S. Jaakko, Ontology-based framework for integration of time series data: application in predictive analytics on data center monitoring metrics, in: Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2021) - KEOD, INSTICC, SciTePress, 2021, pp. 151–161. https://doi.org/10.5220/0010650300003064

[19] S. Suman, X. Chu, D. Niewenhuis, S. Talluri, T. De Matteis, A. Iosup, Enabling operational data analytics for datacenters through ontologies, monitoring, and simulation-based prediction, in: Companion of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE '24 Companion, Association for Computing Machinery, New York, NY, USA, 2024, p. 120–126. https://doi.org/10.1145/3629527.3652897

[20] X. Zhang, A. Bosselut, M. Yasunaga, H. Ren, P. Liang, C.D. Manning, J. Leskovec, GreaseLM: graph REASoning enhanced language models for question answering, in: International Conference on Learning Representations, 2022.

[21] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, T. Liu, A survey on hallucination in large language models: principles, taxonomy, challenges, and open questions, ArXiv abs/2311.05232 (2023). https://api.semanticscholar.org/CorpusID:265067168.

[22] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, X. Wu, Unifying large language models and knowledge graphs: a roadmap, arXiv preprint arXiv:2306.08302 (2023).

[23] S. Ji, S. Pan, E. Cambria, P. Marttinen, P.S. Yu, A survey on knowledge graphs: representation, acquisition, and applications, IEEE Trans. Neural Netw. Learn. Syst. 33 (2) (2021) 494–514.

[24] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuťtlér, M. Lewis, W.-t. Yih, T. Rocktäschel, et al., Retrieval-augmented generation for knowledge-intensive nlp tasks, in: Advances in Neural Information Processing Systems, 33, 2020, pp. 9459–9474.

[25] Z. Guo, S. Cheng, Y. Wang, P. Li, Y. Liu, Tool learning with large language models: a survey, 2024, https://arxiv.org/abs/2405.17935. 2405.17935

[26] Z. Hong, Z. Yuan, Q. Zhang, H. Chen, J. Dong, F. Huang, X. Huang, Next-generation database interfaces: a survey of LLM-based text-to-SQL, 2024, https://arxiv.org/abs/2406.08426. 2406.08426

[27] N. Rajkumar, R. Li, D. Bahdanau, Evaluating the text-to-SQL capabilities of large language models, 2022, https://arxiv.org/abs/2204.00498. 2204.00498

[28] M. Pourreza, D. Rafiei, DIN-SQL: decomposed in-context learning of text-to-SQL with self-correction, 2023, https://arxiv.org/abs/2304.11015. 2304.11015

[29] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, 2023, https://arxiv.org/abs/2201.11903. 2201.11903

[30] D. Bustamante, H. Takeda, SPARQL generation with entity pre-trained GPT for KG question answering, 2024, https://arxiv.org/abs/2402.00969. 2402.00969

[31] L. Kovriguina, R. Teucher, D. Radyush, D. Mouromtsev, SPARQLGEN: one-shot prompt-based approach for SPARQL query generation, 2023.

[32] L.-P. Meyer, J. Frey, F. Brei, N. Arndt, Assessing SPARQL capabilities of large language models, 2024, https://arxiv.org/abs/2409.05925. 2409.05925

[33] C.V.S. Avila, V.M.P. Vidal, W. Franco, M.A. Casanova, Experiments with text-to-SPARQL based on chatGPT, in: 2024 IEEE 18th International Conference on Semantic Computing (ICSC), 2024, pp. 277–284. https://doi.org/10.1109/ICSC59802.2024.00050

[34] Z. Li, S. Fan, Y. Gu, X. Li, Z. Duan, B. Dong, N. Liu, J. Wang, FlexKBQA: a flexible LLM-powered framework for few-shot knowledge base question answering, in: Proceedings of the AAAI Conference on Artificial Intelligence, 38, 2024, pp. 18608–18616.

[35] T.A. Taffa, R. Usbeck, Leveraging LLMs in scholarly knowledge graph question answering, 2023, https://arxiv.org/abs/2311.09841. 2311.09841
[36] S. Auer, D.A.C. Barone, C. Bartz, E.G. Cortes, M.Y. Jaradeh, O. Karras, M. Koubarakis, D. Mouromtsev, D. Pliukhin, D. Radyush, I. Shilin, M. Stocker, E. Tsalapati, The sciQA scientific question answering benchmark for scholarly knowledge, Sci. Rep. 13 (1) (2023) 7240. https://doi.org/10.1038/s41598-023-33607-z
[37] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al., LLaMA: open and efficient foundation language models, arXiv preprint arXiv:2302.13971 (2023).
[38] T.B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D.M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, in: Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS '20, Curran Associates Inc., Red Hook, NY, USA, 2020.
[39] J. Baek, A.F. Aji, A. Saffari, Knowledge-augmented language model prompting for zero-shot knowledge graph question answering, in: B. Dalvi Mishra, G. Durrett, P. Jansen, D. Neves Ribeiro, J. Wei (Eds.), Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE), Association for Computational Linguistics, Toronto, Canada, 2023, pp. 78–106. https://doi.org/10.18653/v1/2023.nlrse-1.7
[40] W. Chen, Large language models are few (1)-shot table reasoners, arXiv preprint arXiv:2210.06710 (2022).
[41] T. Sorensen, J. Robinson, C.M. Rytting, A.G. Shaw, K.J. Rogers, A.P. Delorey, M. Khalil, N. Fulda, D. Wingate, An information-theoretic approach to prompt engineering without ground truth labels, arXiv preprint arXiv:2203.11364 (2022).
[42] N.F. Noy, D.L. McGuinness, Ontology development 101: a guide to creating your first ontology, 2001, (https://tinyurl.com/2mukx364). Stanford Knowledge Systems Laboratory, Technical Report KSL-01-05.
[43] M. Lefrançois, Planned ETSI SAREF extensions based on the W3C&OGC SOSA/SSN-compatible SEAS ontology patterns, in: Workshop on Semantic Interoperability and Standardization in the IoT (SIS-IoT), Amsterdam, Netherlands, 2017, p. 11. https://hal-emse.ccsd.cnrs.fr/EMSE-01638275.
[44] G. Xiao, L. Ding, B. Cogrel, D. Calvanese, Virtual knowledge graphs: an overview of systems and use cases, Data Intell. 1 (3) (2019) 201–223. https://doi.org/10.1162/dint_a_00011
[45] M. Molan, M.S. Ardebili, J.A. Khan, F. Beneventi, D. Cesarini, A. Borghesi, A. Bartolini, GRAAFE: graph anomaly anticipation framework for exascale HPC systems, Future Gener. Comput. Syst. 160 (2024) 644–653. https://doi.org/10.1016/j.future.2024.06.032