BrickLLM: A Python library for generating Brick-compliant RDF graphs using LLMs

SoftwareX 30 (2025) 102121

Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

BrickLLM: A Python library for generating Brick-compliant RDF graphs using LLMs

Marco Perini [a], Daniele Antonucci [a], Rocco Giudice [b], Marco Savino Piscitelli [b,*], Alfonso Capozzoli [b]

[a] Institute for Renewable Energy, Eurac Research, 39100 Bozen/Bolzano, Italy
[b] Department of Energy "Galileo Ferraris", TEBE Research Group, BAEDA Lab, Politecnico di Torino, Corso Duca degli Abruzzi 24, Turin, 10129, Italy

ARTICLE INFO

Dataset link: https://github.com/EURAC-EEBgroup/brick-llm

Keywords:
Brick
RDF
Large Language Models
Portability
Energy Management and Information Systems

ABSTRACT

One of the key challenges of Energy Management and Information Systems in buildings is related to the lack of interoperability, due to the absence of standardization of the underlying data models. In recent years, there has been a growing interest in using ontology-based metadata models to address this issue, as they offer a structured approach to organize and share information across diverse systems (e.g. Brick ontology). However, the creation of ontology-based metadata models is often a labor-intensive task that requires specific domain expertise, hindering the practical use of such data models. For this reason, in this work the BrickLLM Python library is introduced, which addresses this issue by generating Brick-compliant Resource Description Framework graphs through Large Language Models, automating the process of converting natural language building descriptions into machine-readable metadata. The library supports both cloud-based APIs (e.g., OpenAI, Anthropic, Fireworks AI), local models (e.g. LLaMa3.2, etc.) and evenfine-tuned ones. This paper explores the architecture, key functionalities, and practical applications of BrickLLM, showcasing its potential impact on the future of building systems monitoring and automation.

Code metadata

Current code version	v1.2.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00607
Permanent link to Reproducible Capsule	https://zenodo.org/records/14039358
Legal Code License	BSD-3 Clause License
Code versioning system used	git
Software code languages, tools, and services used	Python, LangChain, LangGraph, RDFLib, PyShacl, Poetry
Compilation requirements, operating environments & dependencies	Requires Python 3.x, dependencies managed via Poetry
If available Link to developer documentation/manual	https://github.com/EURAC-EEBgroup/brick-llm/wiki
Support email for questions	daniele.antonucci@eurac.edu

1. Motivation and significance

The increasing digitalization of buildings has amplified the importance of Energy Management and Information Systems (EMIS) in optimizing energy usage, improving system performance, and enhancing occupant comfort [1]. However, as building systems grow more complex, EMIS solutions face significant interoperability and scalability challenges. Modern buildings, particularly large commercial and institutional ones, comprise a range of interconnected systems, such as HVAC, lighting, monitoring, and security systems, that must work in an efficient interoperable way [2]. To ensure broad applicability, the EMIS solution must be adaptable for deployment across different buildings. This flexibility is crucial given that available monitored data is often unstructured and challenging to interpret, demanding substantial effort and expertise to align it with a standardized naming convention or metadata schema [3]. In many cases these solutions are developed from scratch without leveraging any standardized or formalized data models,

* Corresponding author.
E-mail address: marco.piscitelli@polito.it (Marco Savino Piscitelli).

https://doi.org/10.1016/j.softx.2025.102121
Received 12 November 2024; Received in revised form 28 February 2025; Accepted 3 March 2025
Available online 17 March 2025

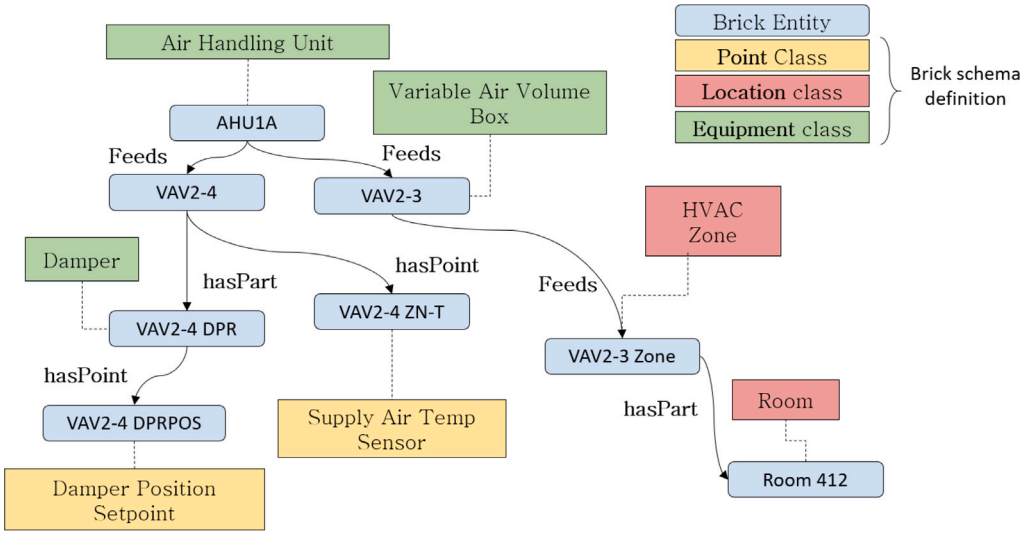

Fig. 1. Example of modeling an AHU-VAV system using the Brick ontology.
Source: Adapted from [7].

creating significant challenges when attempting to implement the same EMIS application in different buildings or to integrate new features in the analytical process. Without a clear, machine-readable representation of building data, this process becomes manual and error-prone [4], leading to inefficiencies, higher costs, and longer deployment time.

Semantic metadata models are structured frameworks that offer a promising solution for these issues. They, in fact, provide a machine-readable, consistent format to describe building equipment (such as energy systems, sensors, controllers, etc...) and their relationships, enabling seamless integration across different building systems [5]. A common approach to implement semantic metadata models is through the Resource Description Framework (RDF). RDF is a graph-based data model that represents entities (such as building rooms and sensors in a monitoring infrastructure) as nodes, with their relationships defined as edges [6]. Semantic metadata models are fundamentally structured around ontologies, which rigorously define domain-specific concepts, relationships, and hierarchies in a standardized and formalized framework. This standard vocabulary ensures that various EMIS platforms can interpret and use data consistently. A leading ontology for building systems is "Brick", an open-source framework designed to standardize the description of physical and logical building entities [7]. Several studies in the literature highlight the advantages of implementing a semantic layer based on Brick ontology to enhance the portability of data-driven energy management strategies in buildings. This approach has demonstrated its effectiveness across various application fields, including advanced system control [8] and fault detection and diagnosis [9]. For reference, a visual example on how to semantically describe a building HVAC system using Brick ontology is provided in Fig. 1.

Despite the well recognized importance of structured metadatada models for EMIS portability and interoperability, their creation is often a labor-intensive task that requires specific domain expertise. Manually creating these metadata models is time-consuming, posing a major barrier to widespread adoption, especially for complex or large-scale buildings. Additionally, the expertise needed to understand both the domain and the related ontology restricts the pool of people qualified to contribute to this process. In recent years, the development of powerful Large Language Models (LLMs) has opened new possibilities for non-expert user to generate semantic metadata models in a fast and accurate way [10,11]. By prompting LLMs with natural language descriptions, users can automate the creation of these models. However, since LLMs are not typically domain-specific, they require careful instructions to produce meaningful and accurate outputs.

To the best of the authors knowledge, no literature specifically addresses the use of LLMs for generating semantic metadata models in the building domain. A few attempts have been made in other areas with the aim to automate the construction of ontology-based metadata models. For instance, van Cauter et al. used local LLMs to extract metadata models from maintenance annotations using the Maintenance Short Texts scheme [12]. Cao et al. developed AutoRD, a framework to construct metadata models on rare diseases using medical ontologies via LLM [13]. Eventually, Ding et al. used LLMs to develop a framework that constructs both ontologies and metadata models on specific domain. They automatically construct the domain-specific ontology using Wikipedia to find the entities, which are then linked in a graph-based model employing LLMs [14].

Although various approaches have been explored in other fields, there remains a significant gap in the automatic generation of ontology-based metadata models specifically suited for building energy management and information systems. As a consequence this work introduces *BrickLLM*, a Python library that employs LLMs to generate complete RDF graphs for structuring building and energy system data and metadata using the Brick ontology. The library format provides a structured and modular approach, ensuring maintainability, scalability, and ease of use. Furthermore, it supports collaboration allowing users to contribute and improve it suggesting updates, fixes, or new features to the original project. By leveraging LLMs, *BrickLLM* allows to create comprehensive RDF representations by providing a textual description of physical, logical and virtual assets in a building (including HVAC, lighting and others systems) and the relationships between them. This approach significantlly reduces the time, cost, and expertise required to create a robust metadata model of a building, making the process accessible to a broader audience and enhancing the portability and scalability of advanced data-driven EMIS solutions in the building market. The semantic correctness of the RDF graph generated is ensured by built-in validation tools that use the SHACL shapes [15] defined by the Brick ontology.

2. Software description

BrickLLM is a Python library designed to automate the generation of metadata models for applications in the building domain exploiting Brick as reference ontology. The software simplifies the process of converting natural language descriptions of buildings and energy systems into machine-readable metadata in the form of RDF graphs. By leveraging LLM agents, *BrickLLM* enables users to generate comprehensive

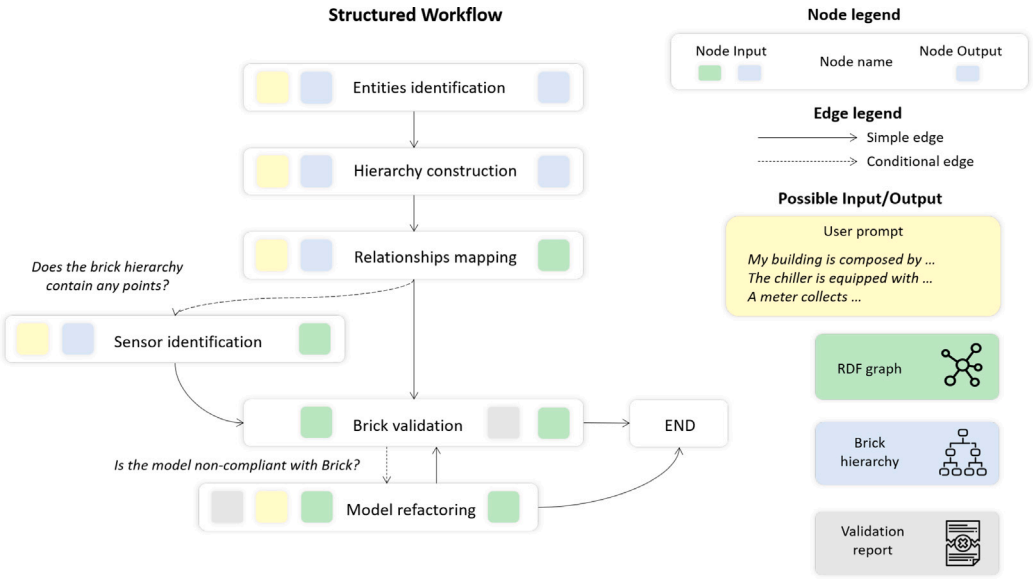

Fig. 2. Structured workflow methodology.

RDF representations of building structures and their energy systems, significantly reducing the time and expertise required for metadata modeling. *BrickLLM* is highly versatile, supporting both cloud-based APIs (e.g., OpenAI [16], Anthropic [17], Fireworks AI [18]) and local open-source models (e.g., LLaMA 2 [19], LLaMA 3 [20]), as well as custom fine-tuned models available as .gguf files. In addition to RDF graph generation, the library includes built-in validation tools to ensure semantic correctness by verifying the generated graphs against the Brick schema using SHACL shapes.

2.1. Software architecture

BrickLLM is built on top of LangChain[1] and LangGraph.[2] LangGraph is a powerful orchestrator that enables the creation of LLM agents, pipelines or workflows quickly and efficiently, by using a graph structure. It allows specifying execution flows, nodes, conditional edges, feedback loops, interrupts, and even implementing a human-in-the-loop approach. These components are compiled into graphs that share a common state, i.e. a Python dictionary, across all nodes and edges, ensuring data continuity and consistency. Each node in LangGraph can be viewed as a 'black box' that takes an input, processes it, and generates an output, which is stored in the common state. These nodes execute specific tasks using LLMs, which can either be local models or accessed via APIs from multiple providers, including OpenAI, Anthropic, and Fireworks AI.

Figs. 2 and 3 show the two main workflows developed in *BrickLLM*.

In the following it is described how a graph-based workflow is executed in more detail:

- The input starts as a natural language prompt that describes the building or energy system that need to be semantically described. This prompt is processed through a series of nodes in a predefined order, orchestrated by LangGraph. BrickLLM at its core supports two different workflows of semantic metadata model generation: the *instructed workflow* and *structured workflow*, represented respectively in Figs. 2 and 3. In Section 2.2 both workflows will be described.
- Each node performs a specific function using LLMs. The function is specified in the instructions provided to the model and is different in each node.
- The data flows between nodes in the form of dictionary, i.e. the common state, which allows easy extraction, transformation, and final serialization to RDF.

2.2. Software functionalities

The library provides two main workflows for generating Brick-compliant RDF graphs starting from a natural language description of building and energy systems: the *instructed workflow* and the *structured workflow*. The use of one or the other depends on the capabilities of the LLM present in the different nodes.

The instructed workflow, shown in Fig. 3, is particularly suitable when fine-tuned models are used, along with the instructions on which the model was trained. In this case, the process is very simple, as it involves generating an RDF graph using a one-shot generation, by means of the node *One-shot generation*, since the model has been instructed to generate the graph directly from a description of the building or energy system. After the generation, the model is validated through the node *Brick validation*, which evaluates whether the RDF graph aligns with the Brick schema. In this case, the validation of the graph to ensure Brick compliance is performed in an open-loop fashion. Specifically, if the fine-tuned LLM fail in generating a valid graph, it is regenerated by running the initial prompt multiple times, with the maximum number of iterations set by the user. As a result the process does not incorporate feedback from previous validation issues. This limitation arises because the model, being relatively small, lacks the capability to refactor the generated model by integrating such feedback.

On the other hand, the structured workflow, is more general and therefore suitable when nodes are controlled by general-purpose LLMs such as GPT4, Claude3.5, etc. In this case, to produce a robust output, several steps must be followed, each represented by a node. The steps, also with the indication of the specific Python module that covers it, are:

- Entity identification (`get_elements.py` node): The LLM extracts relevant entities from the user's prompt, mapping them to the corresponding high-level entities in the Brick schema. All Brick entities mentioned in the prompt are identified and returned.

[1] https://github.com/langchain-ai/langchain
[2] https://github.com/langchain-ai/langgraph

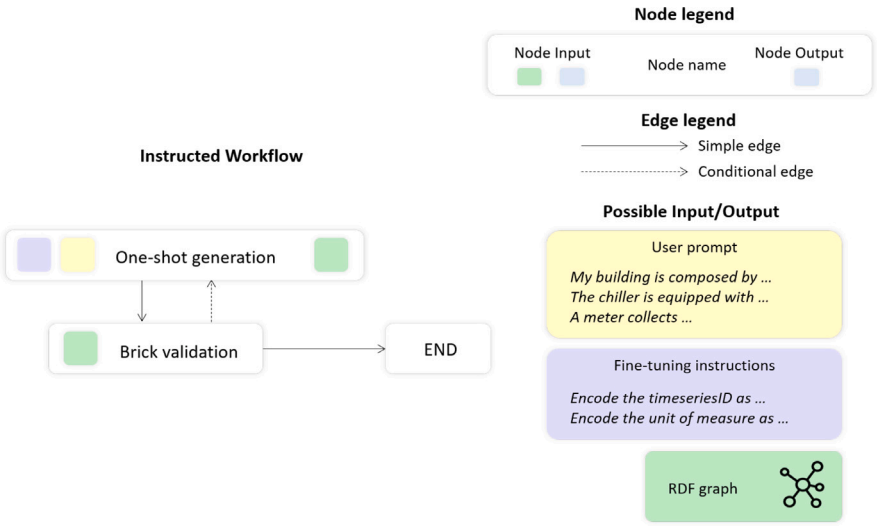

Fig. 3. Instructed workflow methodology.

- Hierarchy construction (`get_elem_children.py` node): The identified entities are structured into a parent–child hierarchy, creating a JSON representation that reflects their relationships. In this way, a subset of the entire Brick ontology is constructed.
- Relationships mapping (`get_relationships.py` node): this node identify the semantic elements in the user prompt, assign the corresponding Brick type and the appropriate relationship between entities. Finally, the graph is instantiated and all entities and relationships are added.
- Sensor identification (`get_sensors.py` node): If entities of subclass Point are detected, their information is expanded to include attributes like timeseriesID unit of measures, database storage information or other relevant properties. The rationale behind dedicating a specific node to sensor identification is due to the greater complexity involved in modeling sensors compared to other entities.
- Brick Validation (`validate_schema.py` node): The RDF graph is validated against Brick's SHACL shapes. If the validation fails, the graph is regenerated through the `model_refactoring.py` node for a defined number of iterations until it passes validation. The default value of the number of iterations is 3, avoiding incurring endless loops during the generation process.
- Model refactoring (`model_refactoring.py` node): If the validation check fails, this node receive the validation report provided by the previous node and, together with the original user prompt and the RDF graph generated, rebuild a new RDF graph considering the feedback provided.

By following this structured approach, the probability of hallucinations or incorrect/inefficient generations is drastically reduced, due to the fact that each node is responsible for a very precise and limited action and the Brick knowledge is injected gradually.

2.3. Software implementation

The project structure of *BrickLLM* is organized to ensure modularity and ease of maintenance. Fig. 4 reports a schematic description of the folder structure.

In particular, the main modules are:

- **edges/**: contains edge definitions used in graph construction, such as validation conditions (`validate_condition.py` or `validate_condition_local.py`).
- **graphs/**: defines the graphs that are executed, such as `instructed_graph.py` and `structured_graph.py`, which are the two main orchestrators.
- **helpers/**: contains the utility modules to aid with LLM interactions (`llm_models.py`) and prompt management (`prompts.py`).
- **nodes/**: contains the various nodes, i.e. the agents, used in the graph execution, each performing a specific function like extracting Brick elements (`get_elements.py`, `get_elem_children.py`), extracting semantic elements from the prompt and relating them (`get_relationships.py`), validating the model (`validate_model.py`), refactoring the model (`refactoring_model.py`), or directly generate the RDF graph (`one_shot_generation.py`) using a fine-tuned model.
- **ontologies/**: holds data related to ontologies, including `Brick.ttl`, which defines the Brick ontology, and `brick_hierarchy.json`, which stores a JSON representation of the ontology hierarchy.
- **utils/**: contains utility functions, including functions to extract Brick ontology information (`get_hierarchy_info.py` or `query_brickschema.py`), or parsing the RDF graph from the string generated by the LLM (`rdf_parser.py`).
- **docs/**: documentation files for the library.
- **examples/**: Example scripts that demonstrate the use of BrickLLM, including how to interact with different LLMs (`example_anthropic.py`, `example_custom_llm.py`, `example_fine_tuned_llm.py`).

BrickLLM is implemented primarily in Python. The key technologies used include:

- **LangChain**: Used to manage LLM interactions, allowing the library to interpret user inputs accurately.
- **LangGraph**: Implements a graph-based workflow, facilitating modular execution of various stages in the RDF generation process.
- **RDFLib** and **PyShacl**: These libraries are used for RDF data manipulation and validation.
- **Poetry**: Handles dependency management and ensures a consistent environment for developers and users.

The implementation also leverages multiprocessing for executing different graph nodes concurrently, enhancing performance when processing large or complex building descriptions.

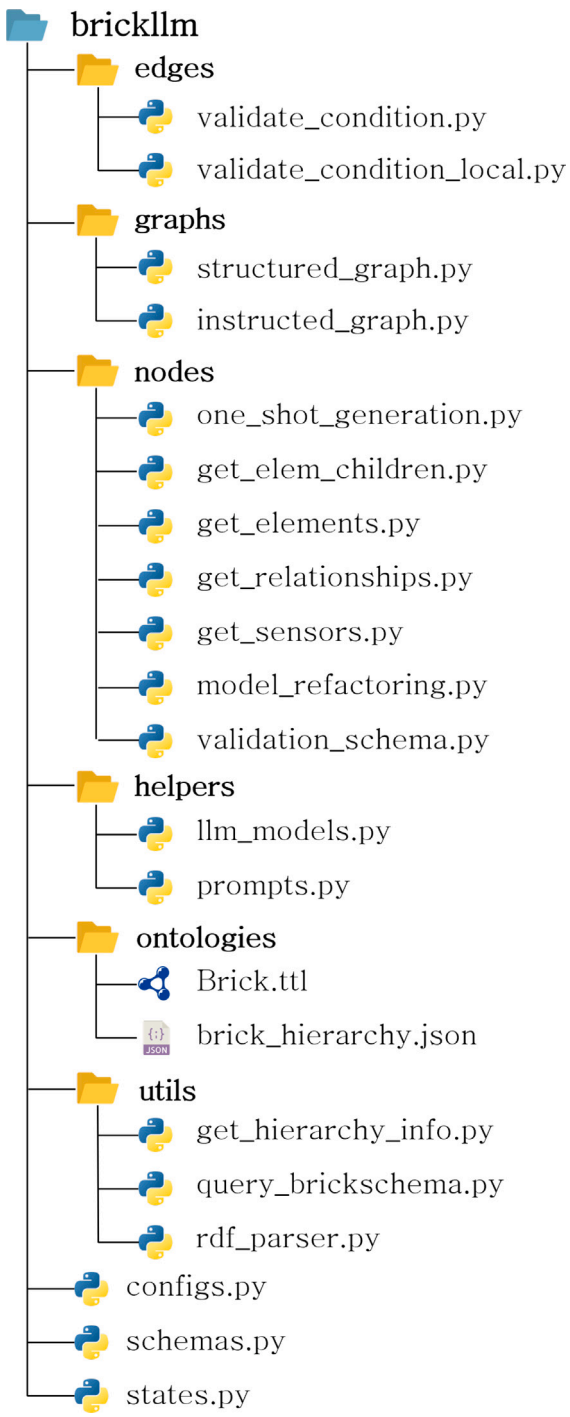

Fig. 4. Schematic description of the folder structure.

2.4. Software validation

To validate the performance and reliability of *BrickLLM*, a series of tests were conducted by generating semantic models for different case studies and evaluating the results. The validation focuses exclusively on the *structured workflow*, as the success of the *instructed workflow* is highly dependent on the fine-tuning process, which is outside the scope of this research.

The case studies used for validation exhibit increasing complexity in terms of graph dimension and the number of distinct Brick entity types. The considered case studies are configured as follows:

1. **Case study 1:** A building with a photovoltaic (PV) system. Both the building and PV system include meters that monitor power sensors (RDF length: 21 triples; Brick entities: 7).
2. **Case study 2:** A building composed of rooms and zones. Each room is equipped with fan coil units fed by a chiller, which is monitored by meters and sensors (RDF length: 47 triples; Brick entities: 11).
3. **Case study 3:** An Air Handling Unit (AHU) consisting of various components such as cooling coils, dampers, and fans, all monitored by sensors on both the AHU and its equipment (RDF length: 167 triples; Brick entities: 25).

For the sake of completeness, the prompt for each case study provided as input to the structured workflow is reported in Appendix A.

The validation process introduced in this study unfolds over the following three steps:

- **Correctness of the RDF Generation (RDF validation):** In this step the RDF graph is validated from the syntax point of view within three iterations.
- **Brick Compliance (Brick validation):** In this step the generated model is validated by checking its adherence to the specifications of the Brick ontology.
- **Coherence with Building Description (SHACL validation):** In this step the generated model is validated against SHACL shapes manually developed by expert users that reflect the entities and relationships described in the prompt provided to the LLM. The SHACL shapes, in this case, represent a reference ground-truth on which to compare the generated model. A minimum level of flexibility in constraint definitions was considered, as entities can often be represented in different ways (e.g., `brick:Meter` `brick:meter` `brick:Chiller` is equivalent to `brick:Chiller` `brick:isMeteredBy` `brick:Meter`, or a thermal zone may be labeled as `brick:Zone` or `brick:HVAC_Zone`).

For each case study, a descriptive prompt was executed 100 times using the *gpt-4o-mini* model, and all three validation steps were evaluated. The results, shown in Fig. 5, demonstrate that the final success rate (SHACL validation) is heavily influenced by the complexity of the semantic model, in terms of both the number of triples and the diversity of Brick entities. Simpler configurations, such as Case study 1 and Case study 2, achieved high success rates of 100% and 87%, respectively. In contrast, while Case study 3 maintained a high success rate in generating Brick-compliant RDF graphs (i.e., 98%), its semantic correctness declined due to the increased graph complexity. It is important to note that the choice of the LLM significantly impacts the results obtained. In this study, a smaller but optimized LLM tailored for instruction-based tasks was used, rather than the most powerful available model. This choice was made based on the suitability of the selected LLM for the specific task considered. The observed variability in semantic coherence results (SHACL valid) can also be attributed to the inherent non-deterministic nature of LLMs. These models operate as probabilistic generators, meaning their output may vary even when provided with identical input. This characteristic underscores the variability in results and is a fundamental aspect of their behavior.

3. Illustrative example

In this section, an example is provided to illustrate the generation of a building metadata model through a structured workflow. However, more examples on how to use the different workflows with different LLMs can be found in the original repository of the software.

The Illustrative examples uses the *structured workflow* to generate the RDF graph of a building that contains a defined topology composed by floors and rooms, and each room is monitored by sensors, whose data are collected by a meter.

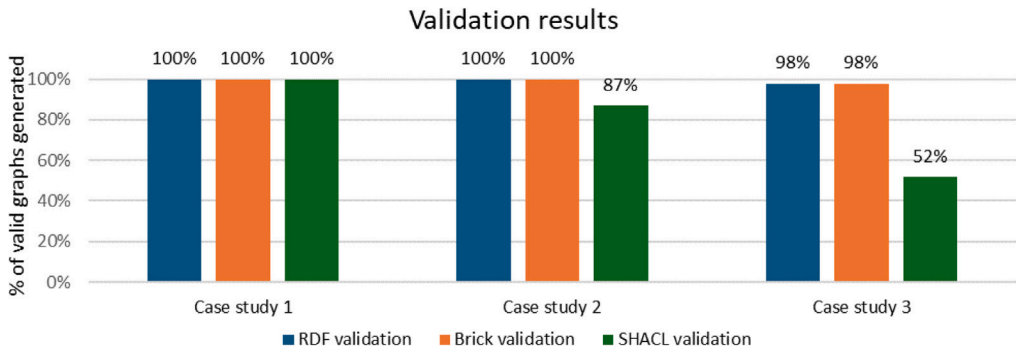

Fig. 5. Results of the validation process of the structured workflow on the different case study configurations.

```
from brickllm.graphs import StructuredGraph

building_description =
"The␣building␣is␣composed␣by␣2␣floors.
Each␣floor␣is␣composed␣by␣1␣room.
The␣room␣at␣the␣first␣floor␣has␣three␣sensors:
–␣Temperature␣sensor␣(C)␣(ID:␣"hfws–xdf2");
–␣Humidity␣sensor␣(ID:␣"regs–452z");
–␣CO2␣sensor␣(PPM)␣(ID:␣"gwq2–FH53).

The room at the second floor has two sensors:
– Temperature sensor (C) (ID: "542c–743s");
– CO2 sensor (PPM) (ID: "deqz–63sr).

A␣general␣meter␣is␣installed␣in␣the␣building,
and␣collects␣the␣data␣of␣all␣the␣sensors␣in␣the
building."

# Create an instance of StructuredGraph with a
# predefined provider
custom_model = ChatOpenAI(temperature=0,
model="gpt–4o–mini")
building_graph = StructuredGraph(model=custom_model)

# Display the graph structure
building_graph.display()

# Run the graph
result = building_graph.run(prompt=building_
description, stream=False)

# Print the result
print(result)

# Save the result to a file
building_graph.save_ttl_output("my_building.ttl")
```

Once finished, the output is the following RDF graph in Turtle format:

```
@prefix bldg: <urn:Building#> .
@prefix brick: <https://brickschema.org/schema/
Brick#> .
@prefix ref: <https://brickschema.org/schema/Brick/
ref#> .
@prefix unit: <http://qudt.org/vocab/unit/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

bldg:Building.1 a brick:Building ;
  brick:hasPart bldg:Floor.1,
    bldg:Floor.2 ;
  brick:isMeteredBy bldg:General_Meter.1 .

bldg:Floor.1 a brick:Floor ;
  brick:isLocationOf bldg:Room.1 .
bldg:Floor.2 a brick:Floor ;
  brick:isLocationOf bldg:Room.2 .

bldg:General_Meter.1 a brick:Meter ;
  brick:hasPoint bldg:CO2_Sensor.1,
    bldg:CO2_Sensor.2,
    bldg:Humidity_Sensor.1,
    bldg:Temperature_Sensor.1,
    bldg:Temperature_Sensor.2 .

bldg:Room.1 a brick:Room ;
  brick:hasPoint bldg:CO2_Sensor.1,
    bldg:Humidity_Sensor.1,
    bldg:Temperature_Sensor.1 .

bldg:Room.2 a brick:Room ;
  brick:hasPoint bldg:CO2_Sensor.2,
    bldg:Temperature_Sensor.2 .

bldg:CO2_Sensor.1 a brick:CO2_Sensor ;
  brick:hasUnit unit:PPM ;
  ref:hasExternalReference
  [ a ref:TimeseriesReference ;
     ref:hasTimeseriesId "gwq2-FH53"^^xsd:string ] .

bldg:CO2_Sensor.2 a brick:CO2_Sensor ;
  brick:hasUnit unit:PPM ;
  ref:hasExternalReference
  [ a ref:TimeseriesReference ;
     ref:hasTimeseriesId "deqz-63sr"^^xsd:string ] .

bldg:Humidity_Sensor.1 a brick:Humidity_
Sensor ;
  ref:hasExternalReference
  [ a ref:TimeseriesReference ;
     ref:hasTimeseriesId "regs-452z"^^xsd:string ] .

bldg:Temperature_Sensor.1 a brick:Temperature_
Sensor ;
  brick:hasUnit unit:DEG_C ;
  ref:hasExternalReference
  [ a ref:TimeseriesReference ;
     ref:hasTimeseriesId "hfws-xdf2"^^xsd:string ] .

bldg:Temperature_Sensor.2 a brick:Temperature_
Sensor ;
  brick:hasUnit unit:DEG_C ;
```

```
ref:hasExternalReference
[ a ref:TimeseriesReference ;
   ref:hasTimeseriesId "542c-743s"^^xsd:string ] .
```

As shown in the output provided by BrickLLM, it accurately identified all entities mentioned in the prompt, including a `Building` with two `Floors`, each containing a `Room`, and a `Meter` that aggregates data from all sensors in the building. The sensors – two `CO2_Sensors`, one `Humidity_Sensor`, and two `Temperature_Sensors`) – were correctly assigned to their respective rooms, incorporating both unit measurements and time-series references as specified in the prompt. Additionally, all relationships identified by the LLM were semantically correct. In this case, the LLM linked `Rooms` and `Floors` using the `isLocationOf` relationship, though a `hasPart` relationship would also have been acceptable.

4. Impact

As discussed in Section 1, scalability and interoperability challenges are crucial in modern EMIS software in the building domain. Ontology-based metadata models are an essential solution, yet creating them requires domain expertise in both building systems and metadata modeling, a combination of skills that is not so common among building professionals.

Previous efforts like BuildingMOTIF have introduced template-based solutions to accelerate Brick model development [21]. While helpful, these approaches still require considerable expertise, limiting accessibility for non-experts.

The developed Python library aims to address these challenges by providing a pragmatic and automated solution to generate semantic metadata models for buildings and energy systems. Through the use of LLM agents, users can automatically generate RDF models from natural language descriptions, eliminating the need for manual input or predefined templates. This simplifies the connection between building-related data sources and portable energy management applications, enabling non-expert users to generate robust, interoperable models more efficiently and in a flexible manner.

The proposed RDF generation opens up new research directions, particularly in the benchmarking and evaluation of different workflows for generating semantic metadata models in the building domain. By making the repository available to the research community, collaboration in expanding the current set of workflows is promoted (currently limited to two), encouraging advances in the field.

5. Conclusions

In this paper, *BrickLLM* was presented, a Python library designed to automate the generation of semantic metadata models for buildings and their energy systems using the Brick ontology. By leveraging LLMs, *BrickLLM* allows users to generate RDF graphs from natural language descriptions, significantly reducing the time and expertise required for metadata modeling. The tool supports both cloud-based APIs and local open-source models, making the generation flexible to the user's capabilities. By democratizing access to ontology-based metadata modeling, *BrickLLM* addresses the critical challenges of scalability and interoperability in modern energy management systems.

CRediT authorship contribution statement

Marco Perini: Writing – original draft, Software, Methodology, Conceptualization. **Daniele Antonucci:** Writing – review & editing, Validation, Supervision. **Rocco Giudice:** Writing – original draft, Software, Methodology, Conceptualization. **Marco Savino Piscitelli:** Writing – review & editing, Validation, Supervision. **Alfonso Capozzoli:** Writing – review & editing, Validation, Supervision.

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.

Acknowledgments

This research was supported by Eurac Research and Politecnico di Torino.

This article has been prepared under the MODERATE project which has been funded by the European Union's Horizon Europe innovation program under Grant Agreement No. 101069834. The work of Marco Savino Piscitelli was carried out within the DM 1062/2021 and received funding from the Italian Ministry of University and Research (MUR) in the framework of the project FSE REACT-EU - PON Ricerca e Innovazione 2014–2020. The work of Rocco Giudice was made in the framework of the project PNRR-NGEU which has received funding from the Italian Ministry of University and Research (MUR) – DM 351/2022. Views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or MUR. Neither the European Union nor the granting authorities can be held responsible for them.

Appendix A

In the following are provided the prompts used to describe the three considered case studies tested for validating the structured workflow implemented in *BrickLLM*.

Case study 1 description prompt:

"A PV array is composed by a PV panel. The PV panel feeds a building. The building is metered by a building electrical meter. The building electrical meter collects data of an electric power sensor. The PV array is metered by an electrical meter. The electrical meter of the PV array collects data of an electric power sensor. The power sensor on the electrical meter has the following timeseries ID: "abfs-qerz-53zw". The power sensor of the building electrical meter has the following timeseries ID: "24sa-4t21-342a"".

Case study 2 description prompt:

"A building has two HVAC zones. The first HVAC zone is composed of two rooms. The second HVAC zone is composed of one room. In each room there is a fan coil unit. A chiller feeds all the fan coil units. The chiller is equipped with a meter. The meter collects data from an electrical power sensor, a current sensor, a voltage sensor and a chilled water flow sensor. The timeseries ID of the sensors and the unit of measures are:

- *"chiller_power" in kilowatt (kW);*
- *"chiller_current" in ampere (A);*
- *"chiller_voltage" in volt (V);*
- *"chilled_water_flow" in cubic meters; per second (m^3/s)"*

Case study 3 description prompt:

"The building is composed by 5 HVAC zones. An Air Handling Unit feeds all the HVAC zones. Each HVAC zone has a zone air temperature sensor. The Air Handling Unit is composed by the following equipment: a cooling coil, a supply fan, a return fan, an outside damper and a return damper. Each equipment has the following sensors:

- *The cooling coil has a valve position sensor;*
- *The outdoor air damper has a damper position sensor;*
- *The return air damper has a damper position sensor;*
- *The return fan has a speed setpoint and a speed status;*
- *The supply fan has a speed setpoint and a speed status;*

The Air Handling Unit is equipped with the following sensors:

- *a supply air temperature sensor;*

- *a return air temperature sensor;*
- *an outside air temperature sensor;*
- *a mixed air temperature sensor;*
- *a supply air temperature setpoint;*
- *an operating mode status;*
- *a supply air flow sensor;*
- *a return air flow sensor;*
- *an outside air flow sensor".*

Data availability

The BrickLLM library is publicly available on GitHub: https://github.com/EURAC-EEBgroup/brick-llm.

References

[1] Kramer H, Lin G, Granderson J, Curtin C, Crowe E, Tang R. Synthesis of year three outcomes in the smart energy analytics campaign. Report, Berkeley, CA: Lawrence Berkeley Laboratory; 2019.

[2] Bergmann H, Mosiman C, Saha A, Haile S, Livingood W, Bushby S, et al. Semantic interoperability to enable smart, grid-interactive efficient buildings. In: 2020 summer study on energy efficiency in buildings. 2020, http://dx.doi.org/10.20357/B7S304.

[3] Chiosa R, Piscitelli MS, Pritoni M, Capozzoli A. A portable application framework for energy management and information systems (EMIS) solutions using Brick semantic schema. Energy Build 2024;323:114802. http://dx.doi.org/10.1016/J.ENBUILD.2024.114802.

[4] Benfer R, Müller J. Semantic digital twin creation of building systems through time series based metadata inference – A review. Energy Build 2024;321:114637. http://dx.doi.org/10.1016/j.enbuild.2024.114637.

[5] Pritoni M, Paine D, Fierro G, Mosiman C, Poplawski M, Saha A, et al. Metadata schemas and ontologies for building energy applications: A critical review and use case analysis. Energies 2021;14(7). http://dx.doi.org/10.3390/en14072024.

[6] World Wide Web Consortium (W3C). RDF - resource description framework. 2014, URL https://www.w3.org/RDF/.

[7] Balaji B, Bhattacharya A, Fierro G, Gao J, Gluck J, Hong D, et al. Brick: Metadata schema for portable smart building applications. Appl Energy 2018;226:1273–92. http://dx.doi.org/10.1016/j.apenergy.2018.02.091.

[8] de Andrade Pereira F, Paul L, Pritoni M, Casillas A, Prakash A, Huang W, et al. Enabling portable demand flexibility control applications in virtual and real buildings. J Build Eng 2024;86:108645. http://dx.doi.org/10.1016/J.JOBE.2024.108645.

[9] Mavrokapnidis D, Fierro G, Korolija I, Rovas D. A programming model for portable fault detection and diagnosis. In: Proceedings of the 14th ACM international conference on future energy systems. New York, NY, USA: Association for Computing Machinery; 2023, p. 127–31. http://dx.doi.org/10.1145/3575813.3595190.

[10] Zhang L, Chen Z. Opportunities and challenges of applying large language models in building energy efficiency and decarbonization studies: An exploratory overview. 2023, http://dx.doi.org/10.48550/arXiv.2312.11701.

[11] Zhu Y, Wang X, Chen J, Qiao S, Ou Y, Yao Y, et al. LLMs for knowledge graph construction and reasoning: Recent capabilities and future opportunities. 2024, http://dx.doi.org/10.48550/arXiv.2305.13168.

[12] Van Cauter Z, Yakovets N. Ontology-guided knowledge graph construction from maintenance short texts. In: Proceedings of the 1st workshop on knowledge graphs and large language models. Stroudsburg, PA, USA: Association for Computational Linguistics; 2024, p. 75–84. http://dx.doi.org/10.18653/V1/2024.KALLM-1.8.

[13] Cao L, Sun J, Cross A. AutoRD: An automatic and end-to-end system for rare disease knowledge graph construction based on ontologies-enhanced large language models. 2024, http://dx.doi.org/10.48550/arXiv.2403.00953.

[14] Ding L, Zhou S, Xiao J, Han J. Automated construction of theme-specific knowledge graphs. 2024, http://dx.doi.org/10.48550/arXiv.2404.19146.

[15] World Wide Web Consortium (W3C). Shapes constraint language (SHACL). 2017, URL https://www.w3.org/TR/shacl/.

[16] OpenAI. OpenAI. 2024, https://openai.com. [Accessed 04 November 2023].

[17] Anthropic. Anthropic. 2024, https://www.anthropic.com/. [Accessed 04 November 2023].

[18] AI F. Fireworks AI. 2024, https://fireworks.ai/. [Accessed 04 November 2023].

[19] Touvron H, et al. Llama 2: Open foundation and fine-tuned chat models. 2023, arXiv:2307.09288, URL https://arxiv.org/abs/2307.09288.

[20] Dubey A, et al. The llama 3 herd of models. 2024, http://dx.doi.org/10.48550/arXiv.2407.21783, arXiv:2407.21783.

[21] Fierro G, Saha A, Shapinsky T, Steen M, Eslinger H. Application-driven creation of building metadata models with semantic sufficiency. In: Proceedings of the 9th ACM international conference on systems for energy-efficient buildings, cities, and transportation. New York, NY, USA: Association for Computing Machinery; 2022, p. 228–37. http://dx.doi.org/10.1145/3563357.3564083.