



Deliverable 2.4

Requirement generator

ITEA4 22003 FireBIM

Deliverable type	Software
Deliverable reference number	ITEA 22003 D2.4
Work Package	WP2
Submission date	2026/09/28
Dissemination level	Public
Abstract	Deliverable 2.4 contains software implementations of the proposed generic framework for digitalisation of fire safety codes (presented in D2.3). The deliverable contains 4 software packages, with the Neuro-Symbolic AI generator being the main software deliverable. Next to this, there is a
Keywords	Fire Safety, Building Codes, Automated Compliance Checking, Digital Building Codes, SHACL



Partner	Author	Role
Eindhoven University of Technology (TU/e) (NL)	Alex Donkers	Semantic web expert
Eindhoven University of Technology (TU/e) (NL)	Ekaterina Petrova	WP2 Lead / BIM expert
Technical University of Denmark (DTU) (DK)	Anne Dederichs	Project Coordinator / Fire Safety Expert



Table of Contents

Executive Summary	Fejl! Bogmærke er ikke defineret.
Table of Contents	3
1. Introduction.....	5
2. Software components.....	6
2.1. Neuro-symbolic Requirement Generator	6
2.1.1. Software structure	8
2.1.2. Application architecture	8
2.1.3. Language-specific natural language processing.....	9
2.1.4. spaCy model management.....	9
2.1.5. Ontology loading.....	10
2.1.6. Ontology context optimisation.....	10
2.1.7. Prompt generation	11
2.1.8. LLM providers	11
2.1.9. Structured generation result.....	11
2.1.10. SHACL generation and validation	12
2.1.11. SHACL-to-Mermaid visualisation	12
2.1.12. Streamlit web application.....	12
2.1.13. Command-line interface.....	13
2.1.14. Standalone script	14
2.1.15. Dependencies and installation	14
2.1.16. Error handling	14
2.1.17. Current limitations.....	15
2.2. Thematic metadata enrichment	15
2.2.1. Purpose	15
2.2.2. Input.....	15
2.2.3. Output.....	15
2.2.4. Internal operation.....	15
2.2.5. Example.....	16
2.3. NLP-based SHACL generator	16
2.3.1. Purpose	16
2.3.2. Input.....	16
2.3.3. Output.....	16
2.3.4. Internal operation.....	16
2.3.5. Example.....	16
2.4. Chatbot and validation notebook	17



2.4.1.	Purpose	17
2.4.2.	Workflow	17
2.4.3.	Notebook structure	17
2.4.4.	Example.....	17
3.	Conclusion	17



1. Introduction

1.1. Objectives of Deliverable 2.4

The primary objective of Deliverable 2.4 is to implement the requirement-generation methods developed in Deliverable 2.3 as reusable software components. The resulting tools support the translation of natural language fire safety regulations into machine-interpretable requirements in the form of SHACL shapes, while retaining intermediate representations such as linguistic features, ontology mappings, and First-Order Logic (FOL).

A secondary objective is to make these methods accessible for practical use and further development. The implementation therefore provides a common generation pipeline through a reusable Python package, command-line interface, standalone script, and Streamlit web application. The software also includes validation and visualisation functionality to support inspection of generated requirements.

1.2. Positioning within Work Package 2 (WP2)

Deliverable 2.4 operationalises the methods established in Deliverable 2.3 and provides the software implementation required to apply them consistently. Figure 1 shows the position of D2.4 within the sequence of WP2 deliverables.

- **Upstream Dependencies:** The software builds on the digitalisation framework, regulation graphs, ontologies, rule-representation methods, and neuro-symbolic generation approach established in D2.3. It uses the FireBIM Building Ontology and related semantic resources developed within WP2 to ground generated requirements in formal domain concepts.
- **Downstream Deliverables:** The generated SHACL requirements and supporting software components provide the technical basis for applying the digitalised regulations in subsequent FireBIM activities. In particular, the generated machine-readable requirements can be incorporated into the rulesets and compliance-checking workflows developed later in WP2 and connected to the project demonstrators.

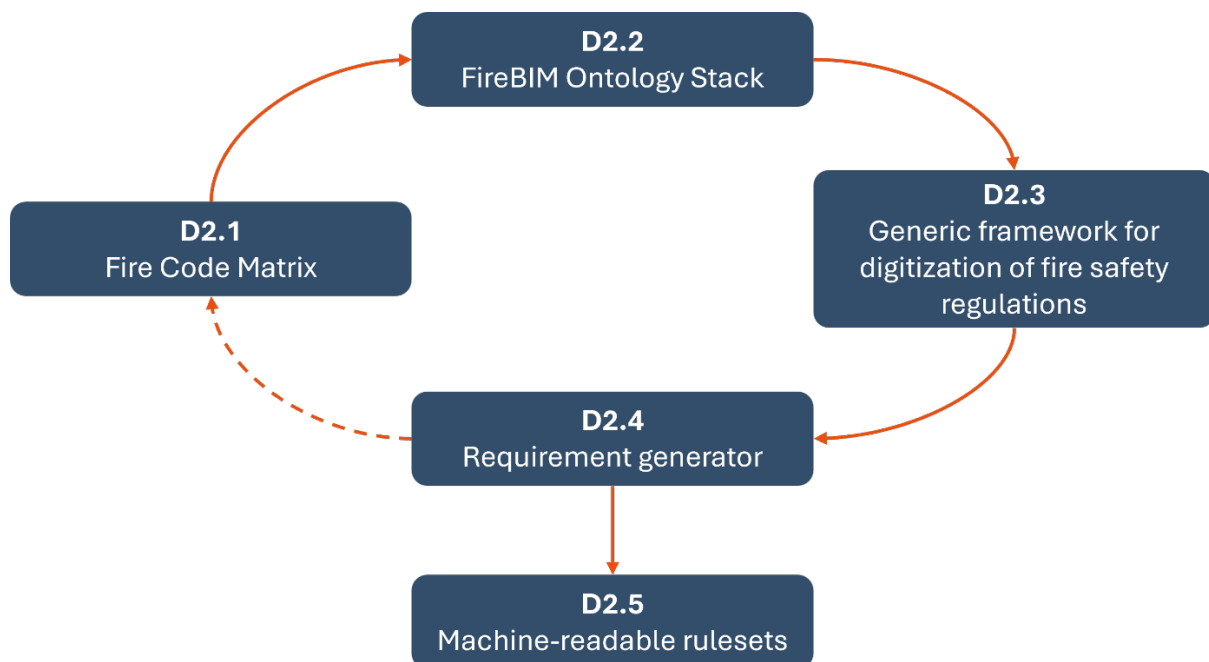


Figure 1 Placement of D2.4 relative to other deliverables in Work Package 2



2. Software components

Deliverable 2.4 provides a set of software components implementing the methods developed within FireBIM for the digitalisation, generation, retrieval, and validation of fire safety requirements. The components are organised as separate tools that can be used independently, while also supporting an integrated workflow from regulatory text to machine-interpretable requirements and compliance checking.

The software components cover four main functionalities: thematic metadata enrichment of regulation graphs, NLP-based generation of SHACL requirements, neuro-symbolic generation of requirements using ontology-grounded intermediate representations, and natural-language interaction with regulation graphs combined with SHACL validation. Each component is accompanied by documentation and example usage, and the implementation is provided as reusable software modules rather than as a single application.

2.1. Neuro-symbolic Requirement Generator

The Neuro-symbolic Requirement Generator implements the sentence-to-SHACL rule generation approach developed in Deliverable 2.3. The software takes a regulatory statement, the language of the statement, and a set of domain ontologies as input and generates a machine-interpretable SHACL shape. In contrast to a direct text-to-SHACL approach, the implementation retains intermediate representations of the interpretation process. These consist of the linguistic features extracted from the regulation, the ontology mappings, and a First-Order Logic (FOL) representation of the regulatory requirement.

The implementation is provided as a reusable Python package, a command-line interface, a Streamlit web application, and a standalone Python script. The same generation pipeline is used by the different interfaces. This avoids maintaining separate implementations for the notebook, the web application, and command-line execution. The software is organised into modules for language-specific natural language processing, ontology handling, LLM interaction, prompt generation, SHACL validation, and SHACL visualisation.



Language

Regulation language

Dutch

spaCy preference: `nl_core_news_lg` (falls back to smaller compatible models if needed)

LLM

Provider

☒ gemini ☐ local

Gemini API key

Model

gemini-3.5-flash-lite

Ontology matches per feature

10

Diagram Mapping FOL SHACL Features Validation Raw response

Antecedent / applicability

Consequent / requirements

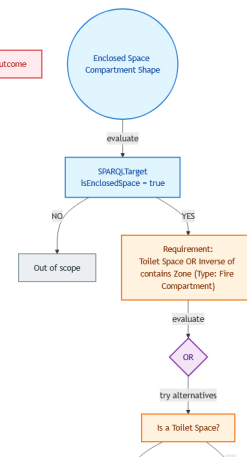
Logical operator

Compliance outcome

Non-compliance outcome

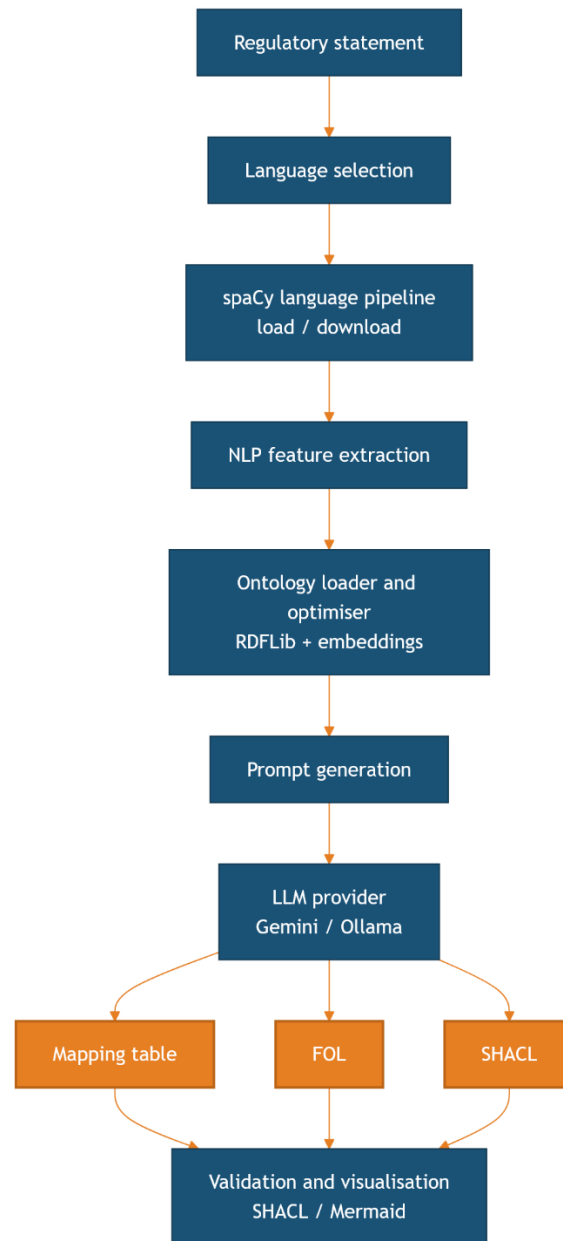
Download Mermaid source

Deploy





2.1.1. Software structure



2.1.2. Application architecture

The implementation follows a layered structure. The **RequirementGenerator** class is the main orchestration layer. It receives the regulatory statement, language, and ontology files and calls the individual processing modules in sequence. The central method is:

```
result = generator.generate(  
    regulation_text=regulation,  
    language="Dutch",  
    ontology_files=ontology_files,  
)
```

The returned **GenerationResult** object contains all intermediate and final results:



```
result.features
result.ontology_matches
result.mapping_table
result.fol
result.shacl
result.validation
result.diagram
result.raw_response
```

The implementation separates the LLM provider from the generation pipeline through an `LLMProvider` interface. The generator therefore does not directly depend on a specific commercial or local model. Two provider implementations are currently available: `GeminiProvider` and `OllamaProvider`.

2.1.3. Language-specific natural language processing

The NLP module extracts linguistic features from the regulatory statement using a spaCy trained pipeline selected according to the language of the regulation. In the initial implementation, the NLP pipeline was fixed to the Dutch `nl_core_news_lg` model. The standalone implementation removes this dependency on a single language and introduces a language parameter as an explicit part of the generation process. The application maintains a mapping between the available regulation languages and corresponding spaCy models. When a language is selected, the application resolves the appropriate model, checks whether it is installed in the active Python environment, and downloads it when necessary. The language model is therefore treated as a runtime dependency rather than as a mandatory installation step for the entire application. This is particularly useful when the same software environment is used for regulations from different countries.

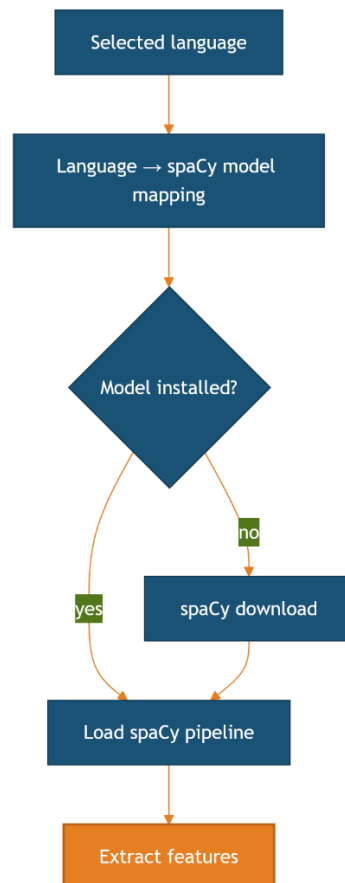
The NLP process itself remains unchanged. Two types of features are extracted:

1. noun chunks;
2. individual content words.

Punctuation, symbols, spaces, and stop words are excluded from the token representation. Noun chunks are retained because domain-specific regulatory concepts are frequently expressed as combinations of nouns and modifiers. The language selection affects the linguistic preprocessing stage only. The remaining stages of ontology processing, LLM interaction, FOL generation, SHACL generation, and SHACL visualisation are handled by the same implementation.

2.1.4. spaCy model management

Language model management is implemented separately from the NLP feature extraction logic. The model resolver receives a language identifier and returns the spaCy pipeline associated with that language. If the pipeline cannot be imported from the current environment, the software invokes the spaCy model downloader and subsequently loads the newly installed pipeline. Conceptually, the process is:



Loaded language pipelines can be cached by the application so that the same model does not need to be loaded repeatedly within a Streamlit session. This approach also changes the installation procedure. The Python package contains the language-independent software dependencies, while the language-specific spaCy model is installed only when it is required.

2.1.5. Ontology loading

The ontology module uses RDFLib to parse one or more Turtle ontology files. Multiple ontology files can therefore be supplied to a single generation run. The ontologies are first merged into a single RDF graph. Resources with `rdfs:label` values and optional comments are extracted as candidate ontology terms. Each extracted term is represented internally by:

```
OntologyTerm(  
    uri=...,  
    label=...,  
    comment=...,  
)
```

This representation is independent of a specific ontology. The generator can therefore use the FireBIM Building Ontology together with additional ontologies where required.

2.1.6. Ontology context optimisation

Loading a complete ontology into an LLM prompt is inefficient and can introduce irrelevant concepts. The implementation therefore reduces the ontology context before the generation step.



The selected ontology terms and the extracted regulatory features are embedded using the Sentence Transformers `all-mpnet-base-v2` model. Embeddings are normalised and compared using cosine similarity. For each extracted feature, the highest-scoring ontology terms are identified. The union of these terms forms the reduced ontology context used by the LLM. The number of retained terms is configurable through the `top_n_ontology_terms` parameter. The software also retains the similarity results. An `OntologyMatch` contains the source feature, ontology URI, ontology label, and similarity score.

2.1.7. Prompt generation

The prompting module constructs the context provided to the LLM. The prompt contains four main parts:

1. the original regulatory statement;
2. the extracted linguistic features;
3. the relevant ontology terms and their URIs;
4. a set of SHACL modelling patterns.

The SHACL pattern library covers common structures encountered in building regulations. The current implementation includes patterns for SPARQL targets, inverse relationships, cardinality constraints, topology relations, numerical restrictions, object-property relationships, logical AND/OR combinations, exceptions, qualified cardinality, and aggregate constraints.

The prompt explicitly separates applicability from requirements. The antecedent of the FOL implication is treated as the applicability condition and is normally mapped to a SHACL target. The consequent represents the requirement and is mapped to SHACL constraints.

2.1.8. LLM providers

The LLM layer provides a common interface for text generation and separates model-specific API functionality from the rule-generation pipeline.

2.1.8.1. Gemini

The Gemini implementation uses the Google GenAI Python SDK. A `genai.Client` is created using the supplied API key, and the selected model is used through the generation API. The Gemini provider uses structured JSON output. The expected response is defined by the Pydantic `LLMGeneration` model.

2.1.8.2. Ollama

The local provider communicates with an Ollama instance through its local API. The selected model is passed to the chat endpoint together with the generated prompt. The local provider also validates the returned object against the same `LLMGeneration` model used by the Gemini provider. The provider abstraction means that the remaining software components do not need to distinguish between Gemini and a local model.

2.1.9. Structured generation result

The generated result contains three main LLM outputs:

```
mapping_table
fol
shacl
```

The mapping table provides the semantic bridge between the regulatory statement and the formal rule representation. The FOL representation provides an intermediate logical representation. It preserves the explicit logic of the regulation before implementation-specific SHACL constructs are introduced. The SHACL field contains



the executable Turtle representation of the requirement. The use of the same structured result model for both cloud and local LLM providers allows the remainder of the software to use one common interface.

2.1.10. SHACL generation and validation

The generated SHACL is returned as Turtle text. Before it is passed to the visualisation component, the software parses the result with RDFLib. A successful parse indicates that the generated output is syntactically valid Turtle. The current syntax validation reports whether parsing succeeded and the number of RDF triples generated. The software also contains a separate validation module using `pySHACL`. This module is intended for validation of a SHACL shape against a building-data graph. The distinction between syntax validation and data validation is deliberate. Syntax validation checks whether a generated shape can be parsed. Compliance validation checks whether building data conforms to the shape. The latter requires a data graph and is therefore not automatically performed for every generation run.

2.1.11. SHACL-to-Mermaid visualisation

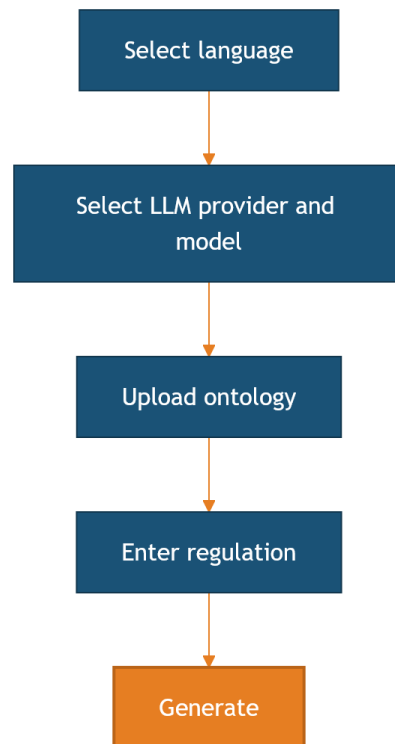
The implementation contains a dedicated `SHACLDiagramGenerator` module for converting the generated SHACL shape into a visual representation. The generator first parses the SHACL Turtle into an RDF graph. It then traverses the structure of the SHACL shape and constructs an intermediate Mermaid flowchart representation. The visualisation distinguishes between the applicability part and the requirement part of the rule. The applicability section is derived from the SHACL target, including `sh:SPARQLTarget`, `sh:select`, and `sh:targetClass`. The consequent is constructed from property shapes, classes, datatypes, cardinality constraints, value restrictions, nested node shapes, qualified value shapes, and logical SHACL constructs. Logical structures are traversed recursively. `sh:and`, `sh:or`, `sh:xone`, and `sh:not` are translated to corresponding logical nodes in the flowchart. The resulting pathway terminates in one of three states:

Complies
Does not comply
Out of scope

Different Mermaid classes are used to distinguish applicability, requirement checks, logical operators, and terminal states. Ontology terms that occur as URI resources can also be linked to their HTTP(S) URI in Mermaid. In the web application, the resulting Mermaid source is rendered in the browser, while the underlying source can also be retained as a .mmd file. The diagram is an explanatory representation of the generated SHACL shape. It is not an alternative rule engine. The SHACL shape remains the source of truth for execution.

2.1.12. Streamlit web application

A Streamlit application provides a graphical interface to the generation pipeline. The input area allows the user to configure the language of the regulation, the LLM provider and model, credentials where required, the ontology files, and the regulatory statement. The language selector is an explicit input because the NLP stage depends on the language of the regulatory text. Selecting a language triggers the corresponding spaCy model resolver. If the required model is not installed, it is downloaded automatically. A typical user interaction is therefore:



The web application hides the model installation and loading procedure from the user. This reduces the amount of manual configuration required when working with regulations in different languages. After pressing the generation button, the application instantiates the requested provider and **RequirementGenerator**, runs the pipeline, and displays the resulting **GenerationResult**. The output is divided into separate views:

- **Diagram**: interactive Mermaid representation of the SHACL logic;
- **Mapping**: ontology mapping table;
- **FOL**: generated logical representation;
- **SHACL**: Turtle representation of the rule;
- **Features**: extracted linguistic features;
- **Validation**: syntax validation status;
- **Raw response**: original structured response returned by the LLM.

2.1.13. Command-line interface

The software also provides a command-line interface through the **firebim-generate** entry point. The CLI accepts:

- LLM provider;
- model;
- language;
- API key where required;
- local LLM endpoint;
- regulatory text file;
- one or more ontology files;
- output directory.

The language parameter is passed to the same NLP model resolver used by the web application. Generation therefore produces the same logical intermediate representations regardless of the selected interface.



2.1.14. Standalone script

For users who do not want to install or interact with the Python package directly, a standalone Python script is also provided. The script exposes the main configuration parameters at the beginning of the file. These include the regulatory statement, language, ontology paths, LLM provider, model, credentials, ontology matching parameters, and output location. The standalone script follows the same processing sequence as the package implementation. The selected language is handled in the same way as in the package and web application: the corresponding spaCy model is resolved and downloaded when necessary. This provides a simple entry point for demonstrations, experiments, and environments where a single Python file is preferable to the package interface.

2.1.15. Dependencies and installation

The implementation is based on standard Python packages for RDF processing, natural language processing, machine learning, LLM interaction, data validation, and web application development. The principal dependencies are:

Component	Technology
RDF parsing	RDFLib
NLP	spaCy
NLP models	spaCy trained pipelines
Semantic embeddings	Sentence Transformers
Similarity calculation	scikit-learn
LLM – cloud	Google GenAI
LLM – local	Ollama
Data validation	pySHACL
Data structures	Pydantic
Web interface	Streamlit
Rule visualisation	Mermaid

The Python dependencies are declared in `pyproject.toml`.

Language-specific spaCy models are not treated as a fixed project dependency. Instead, the application resolves and installs the model corresponding to the selected regulation language when required. This avoids requiring all supported language models to be installed in every environment. A startup script is provided to simplify installation and execution. It creates the Python virtual environment when necessary, installs the Python package, and starts the Streamlit application. Language model installation is handled at runtime by the application.

2.1.16. Error handling

The software validates inputs before execution. Empty regulatory statements and missing ontology files result in explicit input errors. The language resolver checks whether a compatible spaCy model is available before loading it. If the model is missing, the application attempts to install it automatically. The generated LLM output is validated against the Pydantic schema. Generated SHACL is subsequently parsed as Turtle before being passed to the diagram generator. Diagram generation is therefore skipped if the SHACL output is syntactically invalid. The provider abstraction also isolates model-specific API errors from the rest of the generation pipeline. These



mechanisms do not guarantee semantic correctness of the generated rule. They ensure that invalid or incomplete intermediate representations are identified before they are passed to subsequent processing stages.

2.1.17. Current limitations

The current implementation is a research prototype. Although the NLP stage is no longer restricted to Dutch, the quality of linguistic preprocessing depends on the availability and quality of the selected spaCy pipeline. The language resolver also depends on the availability of a compatible spaCy trained pipeline for the selected language. Selecting a language for which no suitable model is available cannot be handled automatically by the current approach. Ontology matching is based on embedding similarity and therefore provides candidate mappings rather than a formal semantic proof that a term is equivalent to the selected ontology concept. The LLM remains responsible for the interpretation and formalisation of the regulatory statement. Consequently, the generated mapping table, FOL representation, and SHACL shape should be reviewed by domain experts. Local LLMs can also differ in their adherence to the requested output structure. The common Pydantic result model detects malformed outputs, but differences in model capabilities can affect the quality of the generated rule. The SHACL-to-Mermaid component provides an explanatory visualisation of common SHACL constructs. Complex or uncommon SHACL patterns may not yet have an equivalent visual representation. The generated SHACL remains the authoritative representation for validation.

2.2. Thematic metadata enrichment

The thematic metadata enrichment tool adds semantic links between regulatory statements and concepts from the FireBIM domain ontology. It extends the regulation graph with thematic metadata, allowing regulatory content to be explored and retrieved based on domain concepts rather than only on the structural organisation of the document.

2.2.1. Purpose

The tool enriches individual regulatory statements with ontology-based thematic tags. These tags support semantic exploration and retrieval of regulations, including queries based on domain concepts such as fire compartments or building functions.

2.2.2. Input

The input consists of regulatory statements retrieved from the regulation graph and a set of ontology concepts against which the statements are evaluated.

2.2.3. Output

The tool adds `fro:isAbout` relations between regulatory statements and relevant ontology concepts. The resulting triples follow the structure:

```
ex:Statement fro:isAbout fbo:Topic
```

where `ex:Statement` identifies the regulatory statement and `fbo:Topic` identifies the associated domain concept.

2.2.4. Internal operation

The implementation evaluates each statement using a zero-shot text-classification approach. The resulting association score is compared with a predefined threshold. When the threshold is exceeded, the corresponding



ontology concept is added to the regulation graph through an **fbo:isAbout** relation. This allows the regulation graph to be queried using semantic concepts in addition to its document structure.

2.2.5. Example

A statement concerning the containment of a space within a fire compartment can be associated with the **fbo:FireCompartment** concept. A subsequent SPARQL query can then retrieve statements related to that concept without requiring the user to know the structure or identifiers of the underlying regulation graph.

2.3. NLP-based SHACL generator

The NLP-based SHACL generator is an early rule-based prototype for transforming a regulatory sentence into a SHACL shape. It uses dependency parsing and noun-chunk analysis to identify the subject, relation, and object of a regulatory statement and maps these components to classes and properties in an RDF/SHACL representation. The implementation is based on spaCy.

2.3.1. Purpose

The tool provides a deterministic NLP-based approach for converting simple regulatory statements into an initial SHACL representation. It was developed as a lightweight alternative to LLM-based rule generation.

2.3.2. Input

The input is a regulatory sentence. The current prototype uses the spaCy **en_core_web_sm** model for linguistic analysis.

2.3.3. Output

The output is a Turtle-formatted SHACL NodeShape containing a target class, a property path, and an object class. The generated terms are represented using FireBIM-style **fbo:** identifiers.

2.3.4. Internal operation

The sentence is first processed using spaCy dependency parsing. The implementation identifies the root of the sentence and its grammatical relations, including the nominal subject and relevant modifiers. Noun chunks and amod and **compound** dependencies are used to construct domain-specific class names. The main relation is then used to construct an RDF property, while the second noun of the relation is converted into an object class. These elements are assembled into a SHACL NodeShape with a **sh:targetClass** and property constraint.

2.3.5. Example

For the statement:

An occupied area is located in a protected sub-fire compartment.

the parser identifies the subject, relation, and object and constructs corresponding **fbo:** classes and a relation. These are then used to generate a SHACL shape with the subject class as target and the identified relation and object class as a property constraint.



2.4. Chatbot and validation notebook

The chatbot and validation notebook provides a training and demonstration environment for querying digital fire regulations and validating SHACL-based rules. It combines natural-language interaction with the regulation graph and Python-based SHACL validation, illustrating how the components developed in FireBIM can be used together.

2.4.1. Purpose

The notebook demonstrates the use of digital regulations from an end-user perspective and provides a reproducible environment for experimenting with rule retrieval and compliance validation.

2.4.2. Workflow

The chatbot accepts a natural-language question and uses an LLM to generate a corresponding SPARQL query. The query is executed against the regulation graph and the retrieved regulatory statements are returned to the user. The generated SPARQL query can also be displayed, providing transparency on how the question was interpreted.

For validation, SHACL shapes retrieved from the regulation graph are applied to a building-data graph using a SHACL validation engine. The notebook can therefore demonstrate both conforming and deliberately violating building data and inspect the resulting validation report. D2.3 describes this workflow using pySHACL and demonstrates the identification of rule violations in example building graphs.

2.4.3. Notebook structure

The notebook is organised as a step-by-step training example. It introduces the required RDF and SHACL data, executes natural-language rule retrieval, performs SHACL validation, and displays the resulting validation reports. This structure allows the individual processing steps to be inspected and modified during experimentation.

2.4.4. Example

A user can submit a question concerning fire compartments and residential functions. The chatbot generates a SPARQL query to retrieve the relevant regulation and associated SHACL shape. The retrieved shape can subsequently be evaluated against an example building graph. Where the building graph deliberately violates the regulation, the notebook displays the corresponding SHACL validation result and identifies the violated constraint.

3. Conclusion

Deliverable 2.4 implements the requirement-generation and rule-exploration methods developed in Deliverable 2.3 as reusable software components. The resulting tools support the enrichment and retrieval of regulatory information, the generation of machine-interpretable SHACL requirements, and their validation against building information.

The neuro-symbolic Requirement Generator provides a common implementation of the sentence-to-SHACL approach, including linguistic preprocessing, ontology-based semantic grounding, FOL formalisation, SHACL generation, validation, and visualisation. The software is exposed through reusable Python components as well as command-line and web interfaces, while the accompanying notebook demonstrates interaction with regulation and building graphs through natural language.

The delivered software constitutes a research prototype rather than a fully automated replacement for expert interpretation. The generated requirements therefore remain subject to domain and semantic review. Within



these boundaries, D2.4 provides the software basis for applying the FireBIM digitalisation approach in subsequent compliance-checking workflows and demonstrators.

3.1. Release

The **Neuro-Symbolic Requirement Generator** has been released as open-source software to support the generation and validation of machine-readable building regulation requirements. The source code is publicly available through the associated GitHub repository (<https://github.com/AlexDonkers/FireBIM-Requirement-Generator/releases/tag/V1.0>) and has been archived on Zenodo to provide a persistent research record and DOI for the released version. In addition, the generator is available as an interactive web application through Streamlit (<https://neurosymbolic-requirement-generator.streamlit.app/>), allowing users to access the functionality without installing the software locally. Together, these release channels provide both a reproducible software artefact for research use and an accessible interface for demonstration and practical use.