Innovating Sales and Planning of Complex Industrial Products
Exploiting Artificial Intelligence

Deliverable 3.7
InnoSale knowledge management components description

Deliverable type:	Document
Deliverable reference number:	ITEA 20054 | D3.7
Related Work Package:	WP 3
Due date:	2024-11-30
Actual submission date:	2024-12-16
Responsible organisation:	:em
Editor:	C. Donges, S. Wendt, T. Scholz (:em)
Dissemination level:	**Public**
Revision:	Final | Version 1.3

Abstract:	InnoSale knowledge management components description. Document describing and explaining the different components developed in WP3 and how it is possible to interact with them. Results will be public as long as no IP issues are affected. (T3.2, T3.2, T3.4, T3.5, T3.6)
Keywords:	

Table_head	Name 1 (partner)	Name 2 (partner)	Approval date (1 / 2)
Approval at WP level	Mario Thron (IFAK)	Anna Räty (MOLOK)	29.11.2024
Veto Review by All Partners			27.12.2024

Editors
Christian Donges (:em)
Thea Scholz (:em)
Sebastian Wendt (:em)

Contributors
Alex Ivliev (TUD)
Nico Herbig (natif.ai)
Yazmin Andrea Pabón Guerrero (UC3M)
Sari Järvinen (VTT)
Arttu Lämsä (VTT)
Tuomas Sormunen (VTT)
Jussi Liikka (VTT)
Kai Huittinen (Wapice)
Mario Thron (ifak)
Sepideh Sobhgol (ifak)
Ahmet Emin Ünal (adesso)
Bilge Özdemir (Dakik)
Frank Werner (Software AG)

Executive Summary

This deliverable D 3.7 is the resulting documentation of D3.1, D3.2, D3.3, D3.4, D3.5 and D3.6. It aims to describe the developed knowledge management components regarding their general use case, the interfaces with data input and output and available functions as well as their basic inner working.

Table of Content

1 Introduction

In WP 3 “Algorithm Design and Implementation” the majority of AI algorithms for the Innosale project get developed, defined in WP1 “Use cases of future sales processes and detailed requirements analysis”. D.3.7 contains a deliverable of a collection of all results of T3.1, T3.2, T3.3, T3.4, T3.5 and T3.6.

The goal is to combine the collected knowledge of all parts within one document. This includes all 18 use cases with a short use case description, the data input, that may be required and data output. Furthermore, it includes the available functions, API calls and user interfaces.

It also describes the inner working of the programs created within this work package.

The use cases are grouped by the following topics:

- Semantic Search
- Knowledge Base
- Customer Segmentation
- Optimal pricing
- Inference Engine

2 D3.2 Semantic Search (NATIF)

2.1 Semantic Inquiry Understanding Using Named Entity Recognition (NATIF)

For Semantic Inquiry Understanding using Named Entity Recognition (NER), we first present the addressed use case, then the technical specifications (Inputs / Outputs, Business Logic, API).

2.1.1 Use case

Customers usually get in contact with the Use Case Partner DEMAG via email / a contact form. In their inquiry, they describe the product specifications. Here, customer expertise differs drastically. Some customers know exactly what they need to specify and which words to use, as they have bought complex products from DEMAG in the past. Others only roughly describe their needs in the language used within their company, sometimes forgetting to specify some relevant information. A sales engineer is therefore needed to understand the customer's intentions and map them to a concrete product configuration. In that process, missing information must be identified, and either must be deducible from other given information or requested from the customer. Once the product configuration is complete, historical projects with similar parameters need to be found, as they can guide the sales engineer in engineering the product and finding a suitable price. Here, a Named Entity Recognition (NER) approach is proposed to automate the inquiry understanding, by extracting relevant knowledge (independent of customer expertise) and transforming it into a structured format, with which historical projects can be searched.

2.1.2 Inputs / Outputs

Input is an inquiry as .eml file (Email format), .txt files (e.g., coming from contact forms), or .pdf files (e.g., printed emails).

The output is a structured JSON, providing the values of the configured crane in overall 33 fields (e.g. the load capacity or power frequency). For each value, a link to the text where this information was extracted from, is provided, so that the sales engineer can easily validate the results in a human-in-the-loop fashion.

2.1.3 Business logic

An in-depth description of the inner workings of the tool can be found in D3.2. Overall, different methods were explored, ranging from (1) directly training a text-based NER, to (2) further pre-training a LLM on domain-specific text and fine-tuning it for the NER task, to (3) exploring a layout-based NER approach that considers the structure of text (bullet points etc.) more closely.

Furthermore, the approach is combined with the inference engine from T3.4, which can deduce information not explicitly stated in the inquiry, as shown in the following **Figure 1**:

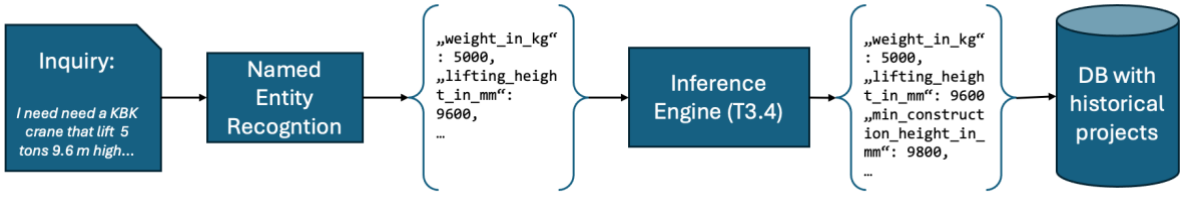


Figure 1: Semantic Inquiry Understanding via NER

2.1.4 Available Functions, APIs, User Interactions

For the integration, an API was developed, that takes an inquiry as input, and outputs the extracted entities, as shown in Figure 2.

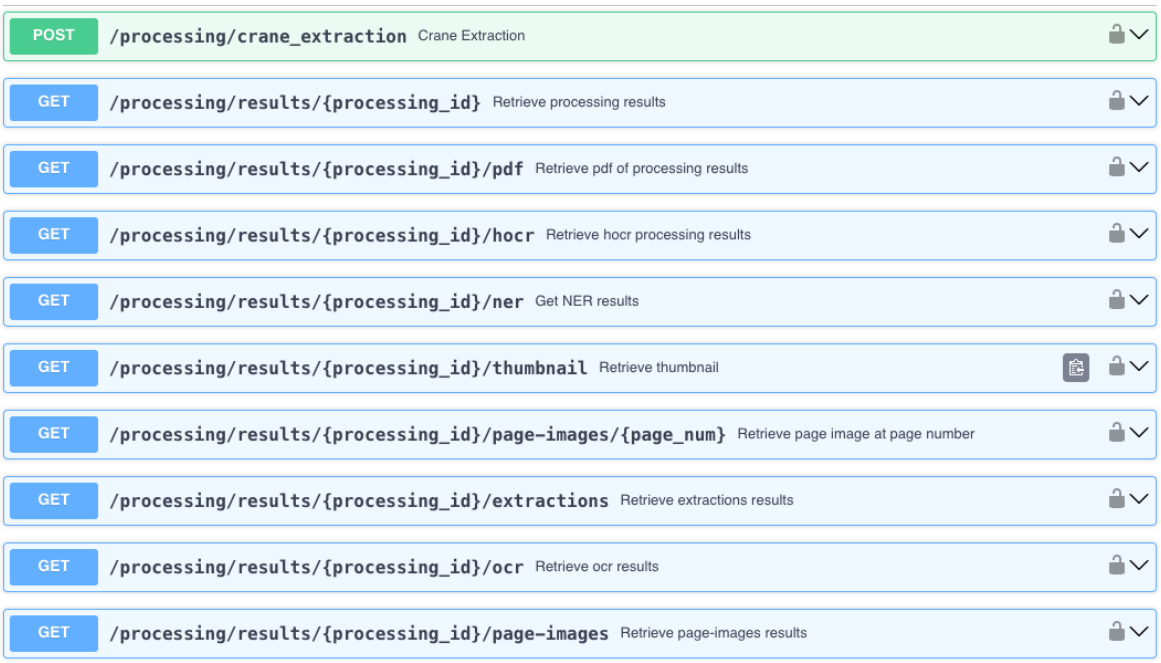


Figure 2: API to transform unstructured inquiries into a structured format

The file simply needs to be POSTed to /processing/crane_extraction. The API can be used synchronously, where the request stays open until the result is ready, or asynchronously, where the request is directly answered with an ID, that can be used to poll the information later on.

In both cases, a structured JSON is returned, containing the entity names as keys, and the extracted values as values.

For authentication, API keys or bearer tokens can be used. Furthermore, the API sticks to the openAPI standard, facilitating the integration into the overall InnoSale solution.

2.2 Semantic Search Within Automatically Transcribed Meeting Notes (DAKIK)

2.2.1 Use case

In the Sheet Metal (SM) use case, employees conduct meetings to discuss customer offers or internal tasks, often in an online setting. These meetings can be recorded, and rather than manually listening to entire recordings, employees can search through automatically transcribed meeting notes. Semantic search, rather than keyword-based search, is used to find relevant discussions, even when the exact words are not remembered, improving efficiency.

2.2.2 Inputs / Outputs

Inputs: Audio files of recorded meetings are uploaded via a user interface, with the option to select the Whisper model for transcription (e.g., large-v2 model fine-tuned for Turkish). Once processed, the meeting transcription is stored in MongoDB.

Outputs: The results of semantic searches are provided in JSON format, showing the most relevant text segments from the transcription, ranked based on similarity to the user's query.

2.2.3 Business logic

The system first transcribes the uploaded audio files using the Whisper model, storing the results in MongoDB. The transcriptions are then segmented and embedded into a vector space using the Universal Sentence Encoder with LangChain. These embeddings are indexed in (Facebook AI Similarity Search) FAISS for fast, similarity-based searches. When a search query is submitted, it is also embedded, and FAISS retrieves the top-matching segments based on cosine similarity, returning them to the user in real-time.

2.2.4 Available Functions, APIs, User Interactions

User Interaction:

Users interact via a web interface where they can upload audio files for transcription and perform semantic searches. Search results are presented on the same interface, with options to view the specific meeting segment or the entire transcription. The interface offers the flexibility to choose different Whisper models based on accuracy and processing time.

APIs:

- Transcription API (`/transcription/upload`): Allows users to upload audio files for transcription. (Figure 3)

- Semantic Search API (`/search/query`): Takes a user query, searches the FAISS vector index, and returns the most relevant meeting segments. (Figure 4)

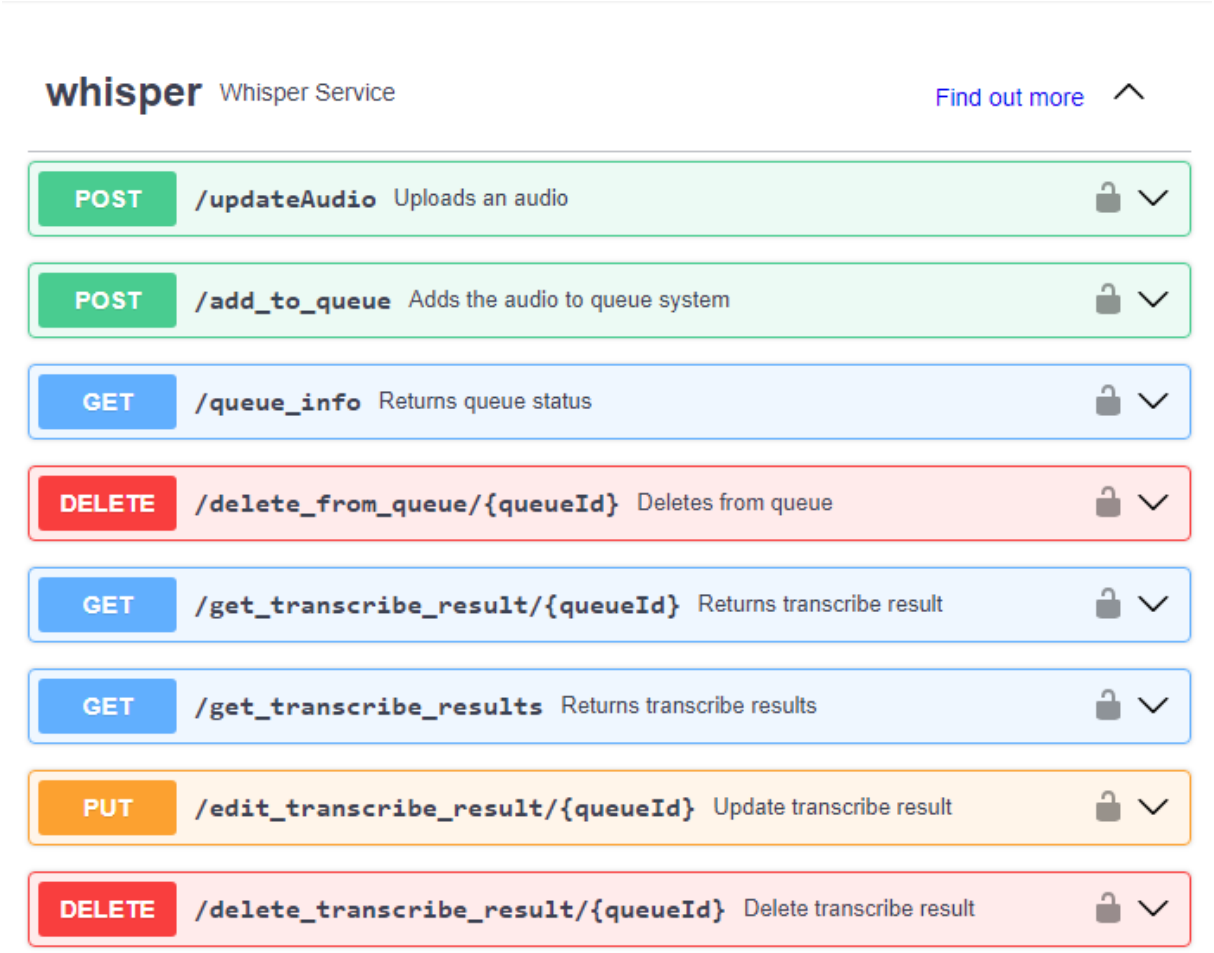


Figure 3: API endpoints for Whisper Service

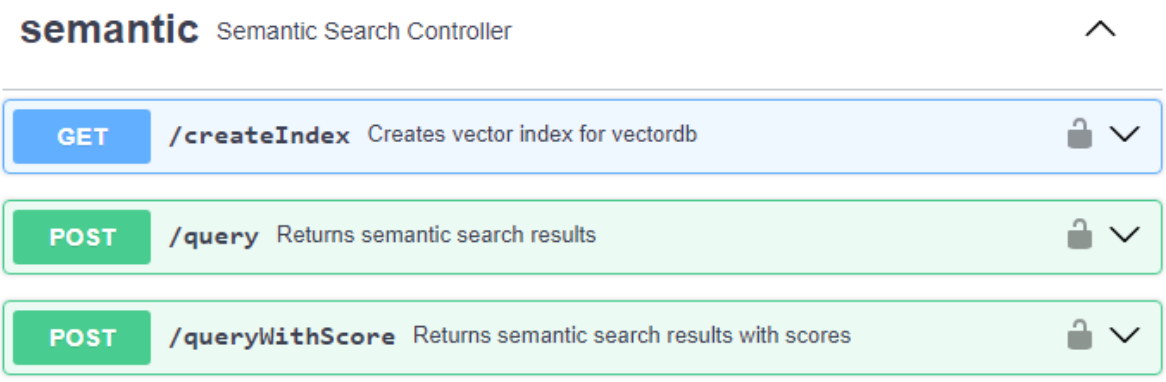


Figure 4: API Endpoints for Semantic Search Service

2.3 Ontology based Semantic Search (IFAK)

2.3.1 Use case

Clients generally reach out to DEMAG, the Use Case Partner, through email or a contact form, outlining their product requirements. Due to divergent levels of customer knowledge, some offer comprehensive descriptions, while others may overlook pertinent details, requiring a sales engineer to decipher and finalize the product configuration process, including identifying absent information and consulting past projects for guidance in product development and pricing. For efficiency, the sales engineer looks for similar projects in the past to derive new technical solutions from proven ones.

2.3.2 Inputs / Outputs

Figure 5 indicates that the input of this technology are inquiry emails of customers, while the output is a sorted list of past projects. In the result list, the most similar projects are provided first.

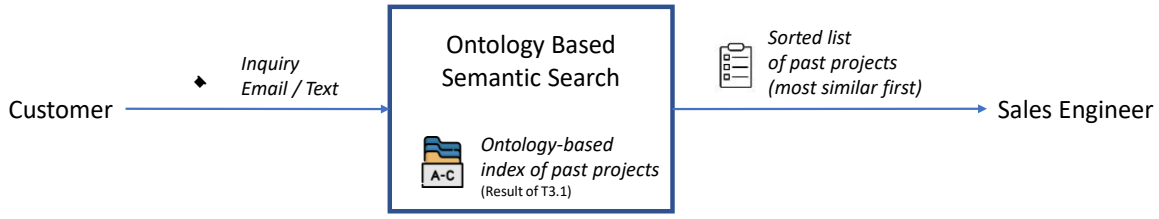


Figure 5: Application of Ontology Based Semantic Search for sales cases

2.3.3 Business logic

Project documents frequently employ technical terminology that differs significantly from the language used by customers in their requests. This discrepancy presents a challenge for effective information retrieval, hindering users from finding relevant information.

An ontology can bridge this gap by establishing relationships (synonyms, generalizations) between expert and layman terms. Semantic search techniques leverage these relationships within the ontology to deliver more accurate and relevant results compared to simple keyword searches.

We explored two semantic search approaches:

- Ontology-based: This approach retrieves documents containing synonyms or generalizations of the user's search terms, even if those exact words aren't present.
- Word Embedding: We compare the ontology-based approach with a word embedding approach to analyse similarities and differences in the retrieved results.

Two main methods connect documents and ontologies:

- Tight Coupling: Documents explicitly reference ontology concepts, simplifying synonym resolution. However, this method requires extensive annotation of documents with semantic information.

- Loose Coupling: Documents aren't bound to a specific ontology, offering flexibility in choosing the most appropriate ontology for the task. However, loose coupling can potentially hinder semantic resolution compared to tight coupling.

Our project adopts a loose coupling approach for its flexibility. Even with loose coupling, connecting document entities to the ontology is crucial for effective semantic search. We leverage shared tokens and entities between documents and existing ontology concepts for annotation, avoiding the need for extensive manual annotation. For detailed information about the ontology structure see section 3.3.

Table 1 and Table 2 show the relations between concepts and existing project files. Existing project files need to be indexed to get search results with high performance. Hence, we initially store the path of each project file, along with a generated ID, in the database. To extract entities from documents, we utilize NER and also perform simple tokenization of the documents. Subsequently, we compare these entities and tokens with concepts in the ontology to establish their linkage.

Table 1: document_index_table

File Name (str)	File ID (int)
DemagCleanedData\DE-262-00467107_01\kfm\Anfrage\dE-262-00467107 OFFER-301645 Hans Mustermann.txt	1
DemagCleanedData\DE-262-00467107_01\kfm\Angebot\AW Ihre Angebotsnummer DE-262-00453901-00 DC-ProCC 2-250 11 H4 V82 380-41550.txt	2

Table 2: concept_document_table

Concept ID (List(int))	File ID (int)
[1]	1
[1, 2]	2
[3 , 5 , 6]	3
[4]	4

To determine the documents within project files that are most similar to our customer inquiry, we propose a method termed "concept frequency". This method calculates the frequency of concepts within each document. Unlike traditional "term frequency", this approach considers all terms related to a specific term, resulting in the same frequency for terms with different relations. Consequently, terms in different languages that are synonyms to specific terms within our customer inquiry receive the same frequency. Thus, when seeking the most similar project files, those containing terms from other languages are also considered. Once concept frequency is calculated, we identify project files containing these concepts. Given that we have already linked concepts and documents in our ontology database, locating them is straightforward. Subsequently, we assign a score to these project files based on the number of concepts they contain. The score is the sum of the frequencies of the concepts within them. The following formula summarizes the aforementioned process.

Let:

- $\boldsymbol{CF(d,c)}$ be the concept frequency of concept ***c*** within document ***d***.
- $\boldsymbol{Count(c,d)}$ be the number of times concept ***c*** appears within document ***d***.
- $\boldsymbol{TotalTokens(d)}$ be the total number of tokens (words or terms) in document ***d***.
- $\boldsymbol{Score(d)}$ be the score assigned to document ***d***.
- $\boldsymbol{Concepts(d)}$ be the set of concepts contained in document ***d***.

Then, the score $\boldsymbol{Score(d)}$ for document $\boldsymbol{d}$ is calculated as the sum of the concept frequencies for all concepts within the document:

$$\boldsymbol{CF(d,c)} = \frac{Count(c,d)}{\text{TotalTokens(d)}}$$

$$\boldsymbol{Score(d)} = \sum_{c \in Concepts(d)}^{\infty} CF(d,c)$$

This formula represents the aggregation of concept frequencies for all concepts within a document, providing a measure of similarity between the document and the customer inquiry based on the shared concepts.

We have created three distinct search functions. The first two adhere to ontology semantic search principles, while the third function employs word embedding techniques.

Search Similar Offers (Ontology based)	scans through concepts within emails and directly compare them with corresponding concepts in the ontology, emphasizing synonym relations.
Extended Search (Ontology based)	scans emails for concepts and matches them against a range of relations in the ontology database, including abstract-specification or part-of connections.
Search Similar Offers (Word Embedding)	generates word embeddings for each project file and the current email. It then compares these two word vectors using cosine similarity. Those project files with the highest similarities will be returned.

2.3.4 Available Functions, APIs, User Interactions

The following figures show the result pages of search by emphasizing synonym relations (Figure 6), abstract relations (Figure 7) and word embeddings (Figure 8).

Project File	Document Score
DemagCleanedData\DE-202-00469403_00\kfm\Anfrage\WG Anfrage B DE-202-00469403_00.txt.txt	0.099
DemagCleanedData\DE-202-00469403_00\kfm\Angebot\DE-202-00469403_00_Black.pdf.txt	0.099
DemagCleanedData\DE-202-00469403_00\techn\Klärung und Notizen\AW Anfrage B DE-202-00469403_00.txt.txt	0.099
DemagCleanedData\DE-202-00469403_00\kfm\Angebot\DE-202-00469403_00_KundenAnonym.rtf.txt	0.086
DemagCleanedData\DE-202-00469403_00\techn\Entwurf\00469403_00_Entwurf_KundeAnonym.rtf.txt	0.086

Figure 6: Similar Offers Emphasizing Synonym Relations

Project File	Document Score
DemagCleanedData\DE-202-00469403_00\kfm\Angebot\DE-202-00469403_00_Black.pdf.txt	0.284
DemagCleanedData\DE-202-00469403_00\kfm\Anfrage\SalesCAD.pdf.txt	0.173
DemagCleanedData\DE-262-00467107_01\kfm\Anfrage\RE Ihre Angebotsnummer DE-262-00453901-00 DC-ProCC 2-250 11 H4 V82 380-41550.txt.txt	0.173
DemagCleanedData\DE-262-00467107_01\kfm\Angebot\AW Ihre Angebotsnummer DE-262-00453901-00 DC-ProCC 2-250 11 H4 V82 380-41550.txt.txt	0.173
DemagCleanedData\DE-262-00467107\Clear Order\03_Angebot\AW Ihre Angebotsnummer DE-262-00453901-00 DC-ProCC 2-250 11 H4 V82 380-41550.txt.txt	0.123

Figure 7: Extended Search Utilizing Other Relation such as abstract-specification relation

Project File	Similarity
DemagCleanedData\DE-202-00469403_00\techn\Kalkulation\00469403_Stï¿½li_Black.pdf.txt	0.919
DemagCleanedData\DE-202-00469403_00\kfm\Anfrage\WG Anfrage B DE-202-00469403_00.txt.txt	0.914
DemagCleanedData\DE-262-00467107_00\kfm\Anfrage\Auftrag 651531_Black.pdf.txt	0.834
DemagCleanedData\NL-003-00451690_00\kfm\Angebot\ABK_Print_00451690_00_Black.pdf.txt	0.833
DemagCleanedData\DE-262-00467107_00\techn\Entwurf\AW DE-262-00467107 OFFER-301645.txt.txt	0.827

Figure 8: Similar Offers Using Word Embeddings

2.4 SentenceBERT-based Semantic Search for Historical Projects (VTT)

2.4.1 Use case

When a sales expert needs additional support for offer creation, a support ticket is used to communicate with back-office support. The back-office prepares manually a customized offer and returns it to the sales expert. Using the semantic search tool the back-office support can easily find historical support tickets with similar content, which can be used to make the process faster and smoother.

2.4.2 Inputs / Outputs

Inputs: Historical Efecte tickets in plain text. New Efecte ticket in plain text.

Outputs: The results of semantic searches are provided in JSON format, showing the most relevant Efecte ticket, ranked based on similarity to the user's query.

2.4.3 Business logic

The aim is for the tool to understand the meaning of the support ticket to identify similar tickets and enable “search with meaning”. The search engine transforms the text to a word-embedding format, compares it with historical data, and identifies relevant tickets using clustering algorithms. The results are presented in a user interface with case information, drawings, relevancy estimation, and voting options for validation. The core of the technical implementation is based on SentenceBERT. The semantic search solution developed by VTT uses SentenceTranformers framework[1] for the utilization of SentenceBERT. The implementation of the semantic search components was done with Python programming language.

2.4.4 Available Functions, APIs, User Interactions

As at the time of this version of the semantic search component's development, the real data from a use case provider was not available; an open data set was used instead. The demonstrator shown in figure below is implemented using a credit card complaint –dataset[2]. The user interface was developed solely for demonstration purposes, but the underlying mechanism implements the SentenceBERT vector transformation and cosine similarity-based search described in the previous section.

[1] https://sbert.net

[2] https://www.consumerfinance.gov/data-research/consumer-complaints/

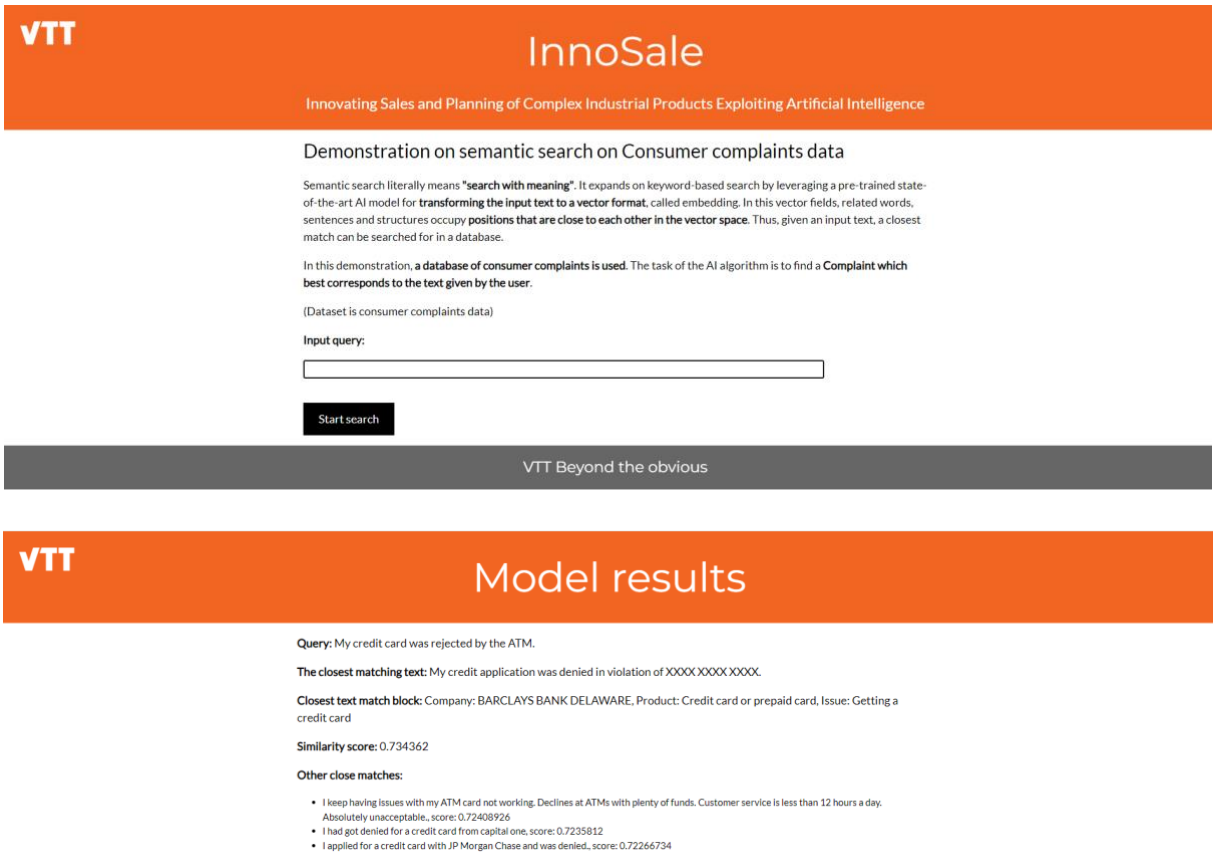


Figure 9: Screenshots of a demonstrator of the VTT semantic search component.

The component will be wrapped as a Docker container. The component can be called by using HTTP POST requests with JSON payloads. The responses are also JSON formatted. When writing this document, the API is not yet implemented but initial specification is presented below in the scope of credit card complaint dataset.

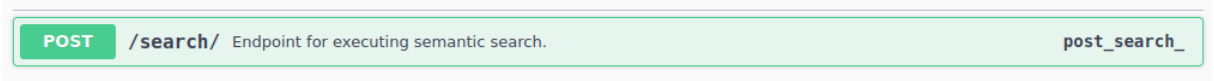


Figure 10: API to perform semantic search with the VTT component.

Example JSON payload for HTTP request to the semantic search component:

{"query": "I have lost my credit card"}

Example JSON payload for HTTP response:

{

"response":

[

{

"document_id": 123,

"matching_text_block": "May have lost my card or it was stolen is it way a to get a new card",

"long_text": "May have lost my card or it was stolen is it way a to get a new card please help me. My XXXX XXXX XXXX credit card # : XXXX and my XXXX XXXX XXXX XXXX

XXXX credit card # : XXXX is lost/stolen, and all the transactions on my credit card accounts is fraudulent transactions.",

"title": "Visa credit card is lost/stolen",

"score": 0.765

}

]

}

2.5 Semantic Search Using an Ontology and a Knowledge Graph (PANEL)

2.5.1 Use case

Customers often require software solutions for their businesses but typically lack the technical expertise necessary to navigate traditional software marketplaces. Platforms like GitHub, Stack Overflow, and Microsoft AppSource usually require users to input specific keywords to search for software, which can be challenging for those without a technical background. Our aim is to allow customers to articulate their requirements in their own words, simplifying the search process and ensuring they find the most relevant software solutions without needing specialized knowledge. The decision to implement semantic search based on ontologies in InnoSale is driven by the need to effectively bridge the communication gap between non-technical user queries and technical software solutions. By enhancing the search process in this way, InnoSale significantly improves accessibility and user satisfaction, catering to a broader audience with diverse technical backgrounds.

2.5.2 Inputs / Outputs

Inputs are natural language requirements as text (txt) files or PDF files. The output is a JSON with the information about the most related digital products.

2.5.3 Business logic

As shown in the following Figure 11, the user inputs a natural language query expressing their software needs. Using NLP techniques, the system extracts not only keywords but also contextual meanings that are then mapped onto specific ontology concepts and relationships defined in the ontology.

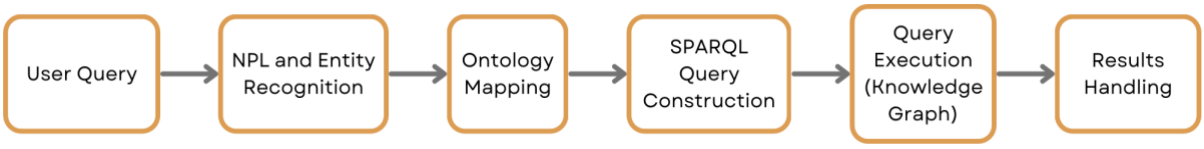


Figure 11: Semantic Search Using an Ontology and a Knowledge Graph Process

We utilized the spaCy NLP library to handle the processing of natural language queries. spaCy provided robust support for various NLP tasks such as tokenization, part-of-speech tagging, and NER. Our approach primarily relied on spaCy's efficient processing pipeline and a set of heuristics we developed to identify and extract key information relevant to our ontology, such as software features, categories, and licensing details.

The ontology, structured in RDF format and described using OWL, includes a variety of classes and object properties that represent different aspects of software solutions. This rich structure allows for detailed mappings and more accurate query formulations.

Each query is analysed to identify which aspects of the ontology it pertains to. For instance, if a user searches for 'CRM software with a flexible license', the system identifies 'CRM' as a Category, 'software' as a Software, and 'flexible license' as a License type.

Using the ontology, natural language queries are decomposed and mapped to these specific classes and their attributes. These mappings facilitate the dynamic construction of SPARQL queries, tailored to search the underlying knowledge graph effectively. The knowledge graph serves as an interconnected repository of data that includes detailed software descriptions and attributes as defined by the ontology. The knowledge graph is regularly updated with new software entries and user feedback, which not only refines current product categorizations

but also enhances the system's learning, making the ontology richer and more aligned with current market trends.

The search algorithm then uses these ontology mappings to execute queries against the knowledge graph. By traversing the relationships in the knowledge graph, the system can retrieve and recommend software products that precisely match the user's expressed needs.

2.5.4 Available Functions, APIs, User Interactions

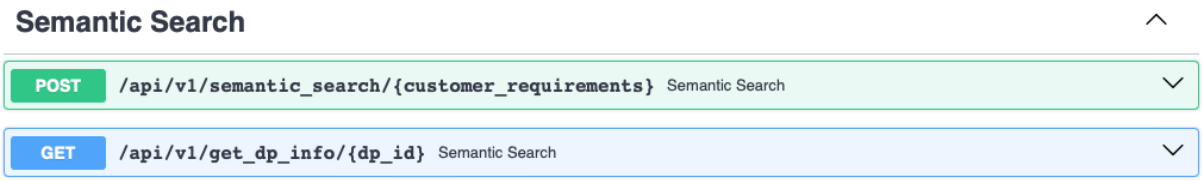


Figure 12: Available APIs

An API (See Figure 12) was developed to process natural language queries from users and return a list of matching digital products. The API endpoints output structured JSON containing information about these digital products. Like other use cases within the InnoSale solution, we adhere to the OpenAPI standard to ensure seamless integration.

3 D3.3 Knowledge Base (DAKIK)

3.1 Historical Data management (Demag, Panel, DAKIK)

3.1.1 Use case

SM (Sheet Metal Stamping): The database in the SM case generally contains numerical data, string and Boolean values, audio files, and 3D part files. Data is stored in table, object, and JSON types. Database management is done through MySQL and MongoDB systems, and data updating is done through Python, which is the backend.

MySQL is a relational database management system in which table structures are predefined and immutable. All records within a given table must adhere to the fixed schema, containing the same set of fields. In such scenarios, the use of SQL is employed for data manipulation and retrieval.

On the other hand, MongoDB is a document-oriented (non-relational) database, offering greater flexibility in terms of data structure. When new data is inserted, corresponding collections are dynamically created, and each record within a collection is not required to possess an identical set of fields. Due to the absence of join operations in MongoDB, the process of retrieving and writing data is generally more straightforward compared to relational databases.

LLE (Material Handling for Light Lifting Equipment): For one of the LLE demonstrators, which is being created as part of UC 1,3,4,5, historical data on past sales cases are required. These historical sales cases are to be mapped and compared with current enquiries coming in as E-Mail requests. This allows similarities to old sales cases to be identified, which should then support the back office in processing the current enquiry.

DPM (Digital Product Marketplace): DPM has a database containing internal digital products from Panel and Softtekt. These internal digital products include the following details:

- requirements specification document,
- class diagram,
- natural language description by the creator,
- tags like programming language, license, platform, quality metrics, etc.

The knowledge base for the digital product marketplace is structured as an ontology. This ontology defines the relationships and classifications between various elements of the digital products. By using an ontology, the system enables semantic mapping of digital products, allowing for advanced querying and reasoning over the data.

Each digital product is stored in a knowledge graph, which models the entities (e.g., digital products) and their relationships as nodes and edges. This graph-based structure allows the system to capture not only the data but also the context and meaning of each element, providing a way to infer new information and improve the discovery and management of digital products.

3.1.2 Inputs / Outputs

SM:

(Inputs) Audio records, 3D metal part files, operation parameters, item parameters.

(Outputs) Part similarity analysis, summarized meeting texts, semantic search results.

LLE:

(Inputs) The data is currently stored in internal company systems to which the partners cannot be given access due to confidential data, a so-called dummy database must be created. This should be a structured database that the partners can access to implement their solutions in the demonstrator.

Figure 13 shows the systems from which data is to be transferred to the dummy database. Firstly, past LLE orders containing information such as reference ID, quantity, price, and currency are filtered via the company's internal SAP system. With the help of the Ref-ID, the corresponding product configurations can be exported from the orders from the Camos system, which contain all the important technical parameters of the ordered product. A certain number of data records are now extracted from these two data sources, merged, and then transferred to the dummy database in a structured manner.

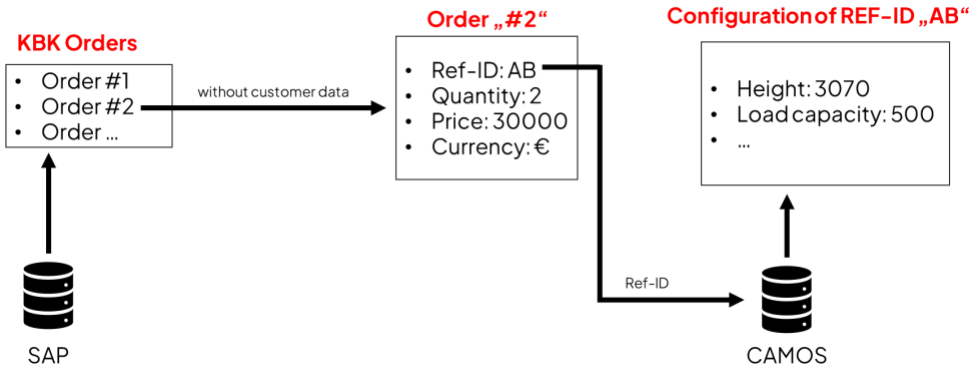


Figure 13: Origin of the historical Datasets from Demag

Another set of data which will be shared outside of the database, directly with the InnoSale partners Natif and ifak, are the E-Mail enquiries from customers. They should be used as historical Sales requests to train their algorithm regarding entity recognition and semantic search.

(Outputs) Specifically, Ifak uses both the data from the database and the data shared from email customer inquiries for its “Semantic Search” approach. IOTIQ will also have visibility into the database as they essentially deal with the integration of the GUI. Also like the TU Dresden, where “rule-based search” is in the foreground. Although Natif will not directly use the data from the database, its solution approach will be used to extract the most important technical parameters from customers' email inquiries using their entity recognition algorithm and automatically transfer them to the required e-project sheet. This is then used, for example, as a basis to automatically fill in missing but essential parameters from historical queries (TU Dresden).

DPM:

(Inputs) JSON with digital product information

(Outputs) OWL ontology

3.1.3 Business logic

SM: The data in the database is read through the backend to be sent to the frontend and transmitted to ReactJS and ThreeJS components. Data is read from the database with the Python backend and presented as endpoints that the frontend can access with the FlaskAPI

system. On the JavaScript frontend, requests are made to endpoints, and data is obtained using the axios system. Python's FlaskAPI system is used in this process.

LLE: The database developed for this use case is a document-based MongoDB, as this was the best fit for the format, the original data was provided in. The initial plan was for Demag to export the needed data and anonymize personal information. This would have been read into the database hosted by :em AG. Because of legal reasons this concept was adapted so that Demag themselves would host the database on their servers. :em AG provides the software as a Docker image, so that Demag can easily start and stop the service as well as read in the initial data needed with a locally run script. The service of Natif is called via an API from Demag's server while TUD's inference tool nemo will run locally on the server. Each partner has the option to access the server hosted by Demag with an external account to develop and test their implementation.

DPM:

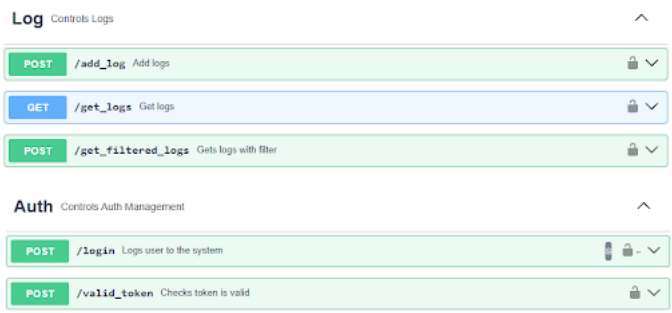
In this use case, the information is stored in a knowledge graph implemented using Neo4J, a graph database management system. To manage the knowledge graph, we have developed an API using FastAPI.

The Knowledge Base in our use case provides information to two main components: the Inference Engine and the Knowledge Acquisition Component (KAC). The interactions are as follows:

1. Inference Engine: This component requires information about all the digital products stored in the knowledge graph. It also has the capability to add new digital products. The API facilitates these interactions by exposing endpoints for querying and updating the digital product information.
2. Knowledge Acquisition Component: This component is responsible for updating or modifying the ontology and its structure to refine and enhance the knowledge graph. The API supports these operations by providing endpoints for ontology management.

3.1.4 Available Functions and APIs

SM: An API was developed. See Figure 14.



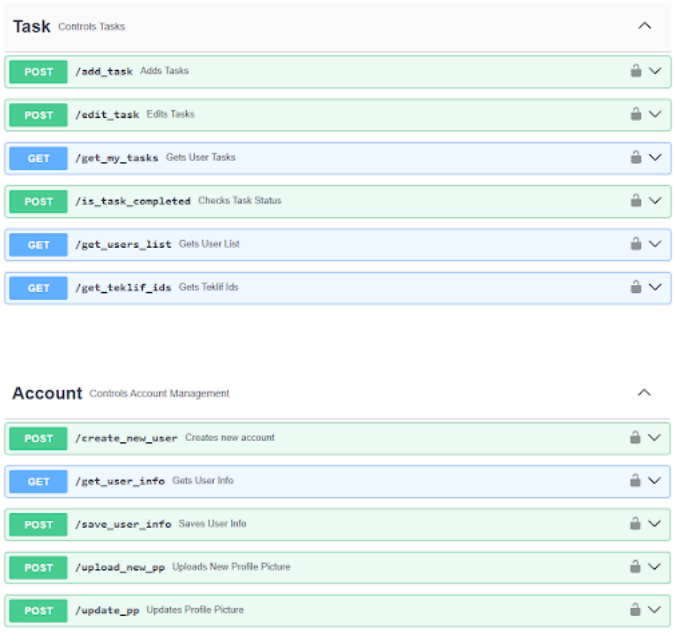


Figure 14: Available APIs

LLE:

An API was developed with FastAPI to enable interaction with the knowledge base which stores historical product configurations and examples of anonymized inquiries. See Figure 15

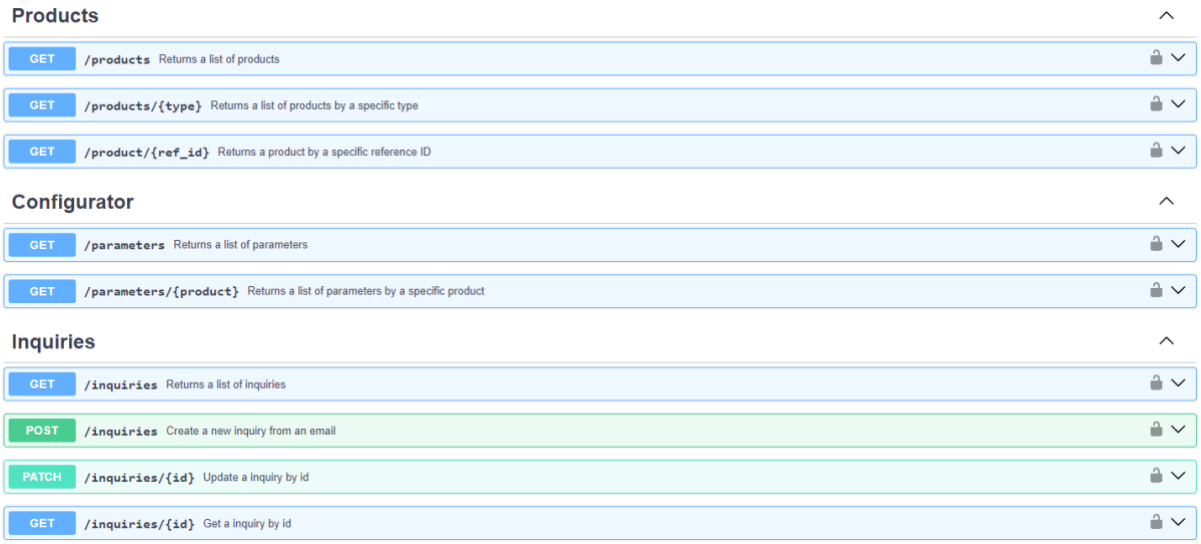


Figure 15: Available API

DPM: The API developed with FastAPI exposes endpoints for interacting with the knowledge base. See Figure 16.

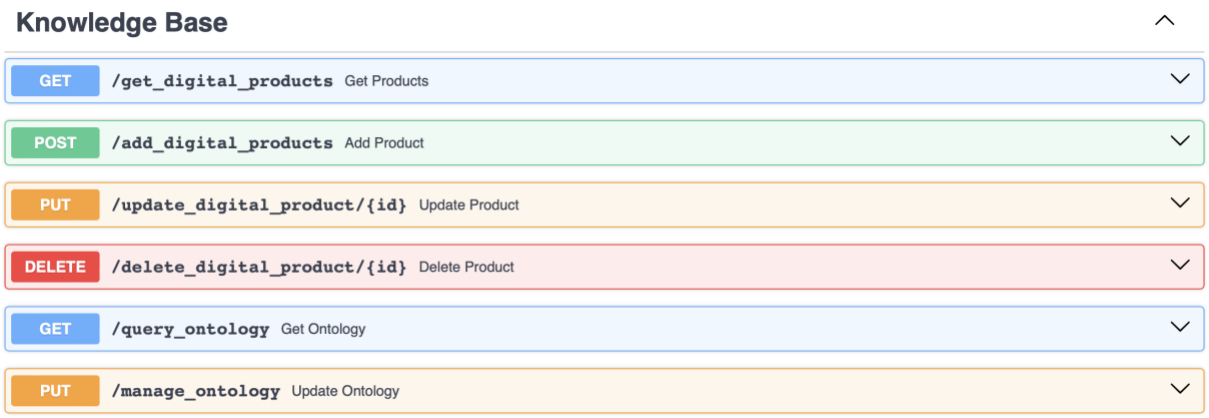


Figure 16: Available API

These endpoints enable seamless communication between the knowledge base, the Inference Engine, and KAC, ensuring that the digital information and the ontology remain accurate and up to date.

3.2 Upper Ontology Development (PANEL)

3.2.1 Use case

This hierarchical structure consists of an upper ontology, which provides a foundational set of concepts applicable in the InnoSale context, and domain ontologies, which cover specific areas relevant to particular contexts. This approach is essential for mapping information into a comprehensive knowledge graph, enhancing data organization and retrieval.

The upper ontology ensures a consistent framework that supports interoperability between different data sources and domains, while domain ontologies address specific contextual needs, enabling precise and relevant data utilization.

The ontology's role in mapping information into the knowledge graph is crucial for organizing and retrieving data efficiently. It ensures that the data is interconnected through defined relationships, which enhances the system's ability to handle complex queries and provides more meaningful search results.

3.2.2 Inputs / Outputs

Inputs:

- Ontology file (RDF/XML, Turtle(.rdf, .ttl))
- Concept and relationship updates (JSON): These updates enable ongoing adjustments and refinements to the knowledge graph to reflect evolving domain requirements.

Outputs:

- Ontology structure (JSON): Upon loading, the system outputs the ontology’s structure in JSON, detailing nodes (concepts) and edges (relationships) that make up the initial graph.

3.2.3 Business logic

For this approach, the data is structured within a graph database, such as Neo4J. This allows for more dynamic and flexible querying of interconnected concepts and relationships, which is particularly beneficial for handling complex queries. The ontology facilitates creating and managing this knowledge graph by defining the nodes (concepts) and edges (relationships) that represent the data.

The following Figure 17 shows the structure of the upper ontology and its branching into domain ontologies:

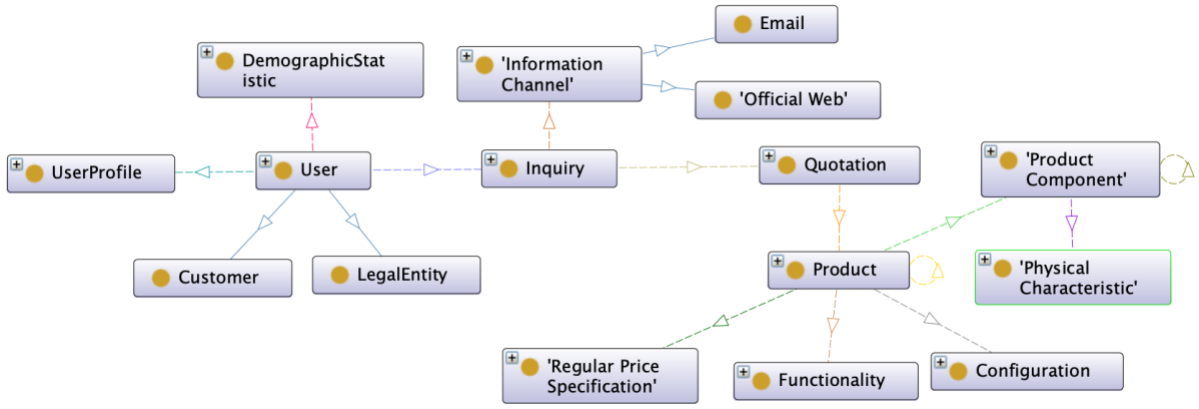


Figure 17: Upper Ontology Structure

3.2.4 Available Functions, APIs, User Interactions

To handle the ontology and knowledge graph effectively, we have an API with functions as `load_ontology', `update_node(node_id, new_data)`, `update_edge(edge_id, new_relationship)`, `save_ontology_graph()`. These functions ensure that the ontology can be dynamically managed and updated, supporting the continuous evolution and refinement of the knowledge graph.

3.3 Development of an Ontology for Semantic Search(IFAK)

3.3.1 Use case

Actor: System Administrator / Knowledge Engineer

Goal: Maintain and update a comprehensive product and product option ontology for efficient information retrieval and project file matching.

Precondition:

- Existing terminologies from Demag (Excel file, Acrolinx database) and Konecranes (Acrolinx database) are available.
- A system for processing and storing the ontology is in place.

Scenario:

1. Initial Ontology Creation: The System Administrator imports existing terminologies from Demag and Konecranes into the ontology system.
2. Terminology Unification: The system merges and standardizes terminology entries, resolving any discrepancies between sources. It incorporates translations and synonyms from the Acrolinx databases.
3. Term Abstraction Integration: The System Administrator adds term abstractions to the unified terminology, enabling representation of product variants and relationships between terms. This transforms the unified terminology into a functional ontology.
4. Optional Ontology Update (Project Files): The System Administrator triggers a process to scan existing project files for terms not present in the ontology. Newly identified terms are added, and their relationships to existing terms are established. A blacklist of frequently occurring but irrelevant words is maintained and updated.
5. Automatic Ontology Update (Inquiry Emails): When a new customer inquiry email arrives:
    a. The system automatically extracts keywords from the email text.
    b. New terms not found in the ontology are flagged for review by the Knowledge Engineer.
    c. The Knowledge Engineer manually adds new terms to the ontology, establishing their relationships as synonyms or term abstractions based on context.

Postcondition: The product ontology is continuously updated and refined, reflecting the evolving vocabulary used by both experts and customers. This ensures accurate information retrieval and efficient matching of customer inquiries with relevant project files.

3.3.2 Inputs / Outputs

Inputs:

- Existing Terminologies:
    - Demag Excel Spreadsheet: Contains terms related to material handling.
    - Konecranes Acrolinx Database: A web-based system shared by Demag and Konecranes, containing terms in multiple languages, along with synonym relationships.
- Project Files: Collection of previous project files in a searchable format.
- Customer Inquiry Emails: Incoming emails from customers expressing their needs and requests.

- Blacklist of irrelevant words frequently appearing in project files.

Outputs:

- Enriched Product Ontology: A structured representation of product-related knowledge, including:
  - Terms and their definitions.
  - Term relationships (mostly synonyms and abstractions, while hyponyms, hypernyms, etc. have not been considered) enabling abstraction and variant identification.
  - Translations in multiple languages.

The output was an SQLite database with a structure as depicted in Figure 18. Here is a short description of the tables:

- **term_table:** Stores terms as string values paired with unique integer termIDs. This allows efficient representation of relations between terms.
- **concept_table:** Links synonyms to the same conceptID, reflecting their semantic equivalence.
- **concept_concept_table:** Captures relationships between concepts, such as "abstract-specific" (e.g., "machine" is abstract to "chain hoist").

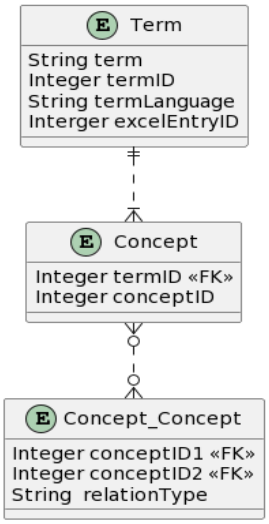


Figure 18: Structure of the Ontology database.

3.3.3 Business logic

3.3.3.1 Unifying Terminologies

The business logic for unifying terminologies from Demag and Konecranes enables a shared understanding of key concepts in the material handling domain. This process involves several crucial steps.

Term Identification and Relationship Extraction

Existing terms from the Acrolinx database are matched with those in the Demag spreadsheet. Synonyms within the Acrolinx database are identified and linked through the conceptID. To detect relationships, we employed two techniques:

A tree-like algorithm identifies parent-child relationships, representing hierarchical structures (e.g., "cover" as an abstract term with specific types like "cover plate" and "cover surface").

We also leveraged a Large Language Model (LLM), specifically the "llama3:8b-instruct-q6_K" model via the Ollama Python library, to analyse relationships between terms. Prompt engineering focused on determining whether one term is a subset of another ("is-a" relationship), with the LLM outputting either "yes" or "no".

Data Integration and Storage

All extracted terms and relationships are stored within a SQLite database using a defined table structure.

Continuous Improvement

This system allows for ongoing refinement, enabling new terms to be added, synonym relationships to be updated, and the LLM-based analysis to be iteratively improved.

3.3.3.2 Optional Update of an Ontology Using Project Files

Existing project files require scanning for uncovered terms to be added to the ontology, potentially relating them to existing entries. This process involves several key steps, outlined as follows.

Preprocessing of Project Files

To initiate this task, project files must first be converted into raw text. Subsequent processing removes unnecessary elements, including:

- Stop words
- Unnecessary numbers
- URLs
- Email addresses
- Non-essential punctuation

A normalization phase ensures spelling accuracy, correcting errors as detected.

Term Extraction Approaches

Two distinct methods can be employed for term extraction from pre-processed text:

Approach 1: Keyword Extraction: This method focuses on extracting statistically significant terms based on factors such as:

- Uppercase/acronym frequency
- Sentence position
- Term frequency within the document
- Co-occurrence of specific terms
- Cross-sentence term appearance

Various algorithms (TKF, TF-IDF, RAKE, YAKE, GRAPH) are available for finding keywords. Notably, YAKE has demonstrated strong results in multilingual keyword extraction.

Approach 2: Comprehensive Term Extraction via Tokenization: For a broader scope beyond just keywords, tokenization divides text into smaller units (tokens), which can be words, characters, or sub words. This approach encompasses all terms within the text.

Ontology Update and Relation Mapping

Regardless of the chosen extraction method, the extracted terms are:

- Compared and matched with existing ontology terms
- Analysed for potential synonym and abstraction relations
- Used to update both the ontology and term-file relationships accordingly.

3.3.3.3 Regular update based on incoming inquiries

Manufacturer-customer communication often involves vocabulary discrepancies between customers and project experts. To bridge this gap, an automated process should update and extend the ontology upon evaluation of each customer inquiry email.

Upon email evaluation, the ontology and relation files are updated with newly encountered words. These additions require manual review to establish relationships, including:

- Synonym terms
- Term abstractions

To facilitate this process, Named Entity Recognition (NER) was applied to sample Demag data. Evaluation results confirm NER's suitability for this task. As a Natural Language Processing (NLP) task, NER involves:

- Identifying named entities within text
- Classifying these entities into specific types, such as: Persons, Organizations, Locations, …

Refer to Figure 19 for a visual representation of the dynamic ontology update process.

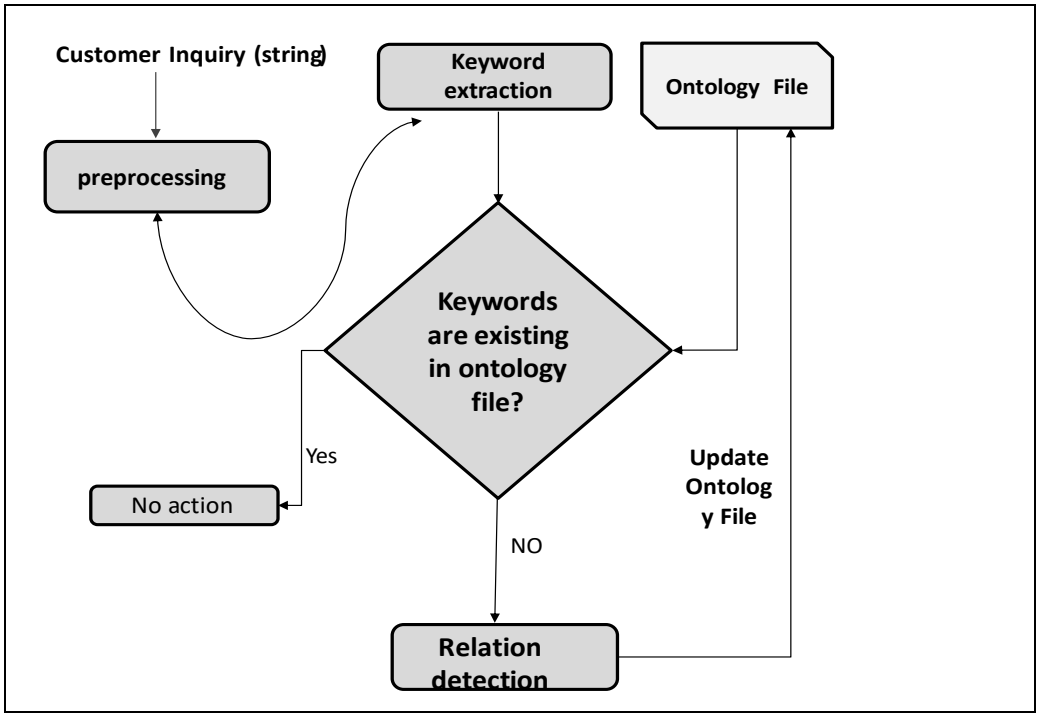


Figure 19: Evaluation of incoming customer inquiries

3.3.4 Available Functions, APIs, User Interactions

The software consists of several Python programs that must be executed separately in a specific order. Here is an overview of how to call each program, their correct execution order, and a brief description:

1. **python database_management.py**: Create a database called OntologyDB.db, which contains all the tables as depicted in Figure 18.
2. **python ontology_creation.py**: Import all terminology from the Demag Excel file (standard terms.xlsx) and TermsAcronix.csv into the database. Combine synonyms and establish "element-of".

3. **python clean_document.py**: Clean the project files by removing unnecessary information. Store the content in a singular format since plural terms are not needed.
4. **python document_processing.py**: Index the documents by saving the path of each document into the database. Afterwards, relate the documents to concepts in the database. "is-a" relations between concepts are being also investigated here because it has dependency on data inside "concept_document_table" which is being filled during execution of document_processing table.

3.4 Fuzzy Logic Rules (IFAK)

3.4.1 Use case

The purpose of the service for managing fuzzy logic rules is to upload and download the rule base for the Fuzzy Control Language Engine (FCLE) as described in section 0. The functionality is now an integrated part of the FCLE and thus, we reference in the following to the comprehensive description of the FCLE.

3.4.2 Inputs / Outputs

The input is an FCL file as defined by IEC 61131-7. An example is provided in section 5.3.1. The output is a message about successful file transfer or a respective error message.

3.4.3 Business logic

We use standard logic for uploading a file to a server. Exceptions are translated into responses in JSON format.

3.4.4 Available Functions and APIs

The API is described as part of section 5.3.4. The routes ‘POST /set_rulebase’ and ‘GET /get_rulebase’ are the relevant parts of the FCLE-API.

3.5 Deductive Reasoning Rules (TUD)

3.5.1 Use case

Expert knowledge that is relevant for suggesting suitable products to customers can be maintained via deductive rules, which are simple if-then statements expressed within a logical formalism. For this purpose, we propose Datalog, extended with various features such as native support for numbers, aggregation, negation, and many built in functions.

3.5.2 Inputs / Outputs

The input is a string containing a list of rules. The output is a message indicating the validity of the given string according to the language specification.

3.5.3 Business logic

The language specification is implemented in Nemo, described in more detail in D3.6.

3.5.4 Available Functions and APIs

The Knowledge Base provides methods for updating and obtaining the rule base.

3.6 Entity Recognition (NATIF)

3.6.1 Use case

The use case for the named entity recognition is described in section 2.1.1. Within the NER component, all this knowledge is modelled implicitly. So instead of learning synonyms, antonyms, and relations like “is_part_of”, NLP algorithms working in a high-dimensional vector space learn such knowledge implicitly from data.

To acquire the expert knowledge, data annotation is needed. For this, the annotation tool shown in Figure 20was used, which offers “types” for relevant entities that shall be extracted on the left and shows the free-form text on the right. Users simply need to select a colour on the left and apply it to the matching text on the right.

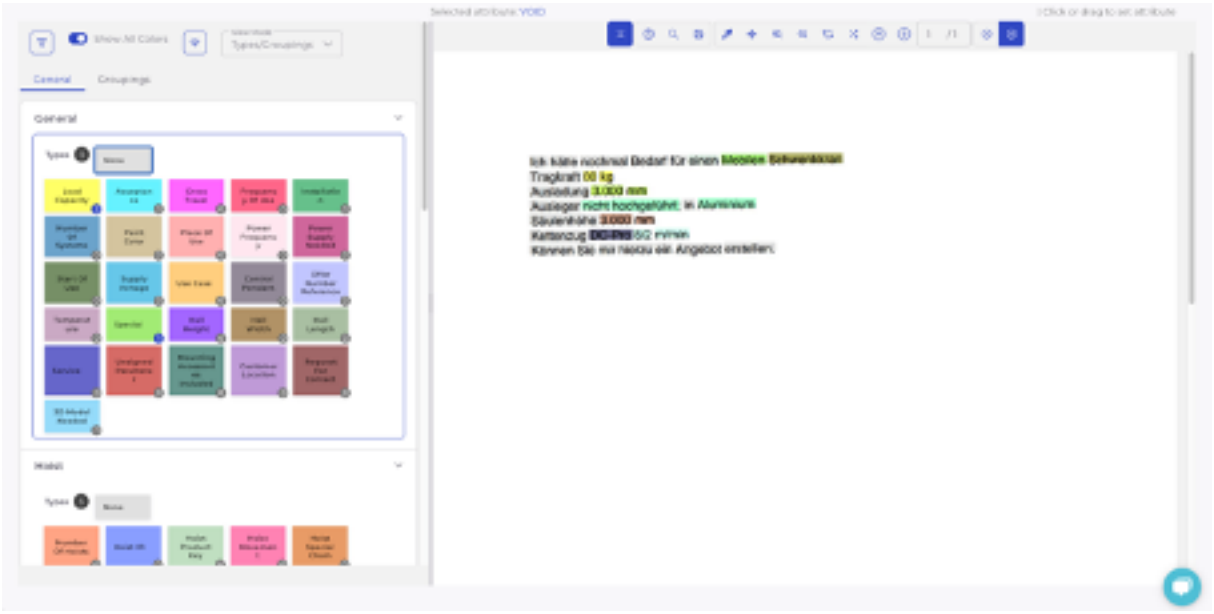


Figure 20: Annotating entities in inquiries

Based on these annotations, NER models can be trained that, given a text, apply the same typing as the user did during annotation.

3.6.2 Inputs / Outputs

Input: customer inquiries to be annotated

Output: annotated inquiries stored in database, that can be used to train AI models automating the task.

3.6.3 Business logic

The tool allows marking text with entities, which are then stored in a database. From there, the machine learning training code can fetch the data to learn mimicking the expert annotator.

3.6.4 Available Functions and APIs

The functions can be summarized as follows:

- Uploading data to be annotated
- Visualizing data as depicted in the above figure
- Selecting entities on the left, and applying that colour on the right via click or drawing
- Storing the annotation

- Loading the next sample to be annotated

That way, experts can easily put their knowledge example-based into our database, so that AI models automating the task can be trained.

3.7 Speech Recognition & Meeting Summarization Parameters (DAKIK)

3.7.1 Use case

Dakik's speech recognition and meeting summarization component is designed to help Ermetal employees search and retrieve key information from recorded meetings without needing to listen to entire recordings. This solution transcribes meetings, analyses them, and summarizes the key points, allowing users to search for specific topics discussed, aiding faster decision-making and information retrieval.

3.7.2 Inputs / Outputs

Inputs:

- Audio files of recorded meetings uploaded via a web interface.

- The user selects parameters for summarization, such as key topics, language models, and summarization settings.

Outputs:

- Transcribed text files stored in MongoDB.

- Summarized meeting content is output in text format and stored in a relational database (MySQL) for further use in semantic searches and meeting reviews.

3.7.3 Business logic

After the user uploads the meeting audio, the Whisper model (fine-tuned for Turkish) processes the audio into text using speech-to-text recognition. This transcription is stored in MongoDB. The system then extracts key points and decisions from the text, applying summarization algorithms that prioritize information based on relevance, using models like Turkish Spacy for NLP. The summarized results are indexed for future searches, allowing users to query the data and quickly retrieve specific information from past meetings.

3.7.4 Available Functions and APIs

User Interaction:

Users can upload audio files, choose summarization settings, and search through the transcriptions and summaries via a user-friendly interface. Results are presented in a structured format, allowing easy access to the key points and full meeting transcripts.

APIs:

- Audio Upload API: Uploads audio files and queues them for transcription. (See Figure 21)

- Transcription API: Processes the audio and stores the text in MongoDB). (See Figure 21)

- Summarization API: Summarizes the transcription based on user-defined parameters, stores the results, and makes them searchable. (See Figure 22)

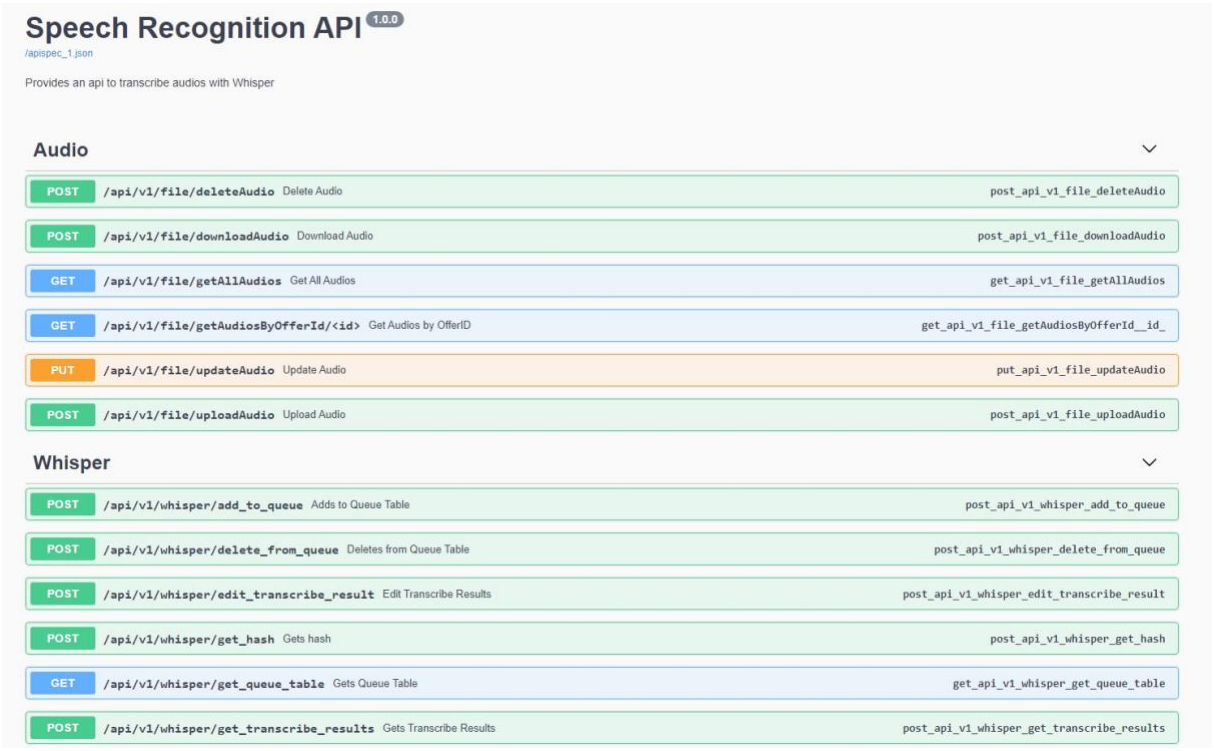


Figure 21: Speech Recognition API

Figure 22: Summarization API

3.8 Similarity Analysis (DAKIK)

3.8.1 Use case

Dakik's similarity analysis component is designed to assist Ermetal in comparing 3D shapes and components for manufacturing purposes. This system uses advanced algorithms to detect similarities between new and existing parts, enabling cost estimations and efficiency improvements by identifying reusable components or designs. The analysis focuses on manufacturing complexity and precision in matching geometric features, which helps sales engineers and product developers to make informed decisions.

3.8.2 Inputs / Outputs

Inputs:
- 3D part files in STL format, along with associated geometric feature data.
- Metadata including dimensions, material type, and other relevant manufacturing parameters.

Outputs:
- A similarity score based on geometric feature comparison and normalized metrics.
- Ranked results displaying the most similar parts from the existing database, with detailed breakdowns of geometric and topological similarities.

3.8.3 Business logic

The similarity analysis begins by extracting geometric features, such as bounding box dimensions and Betti numbers (topological attributes), from the 3D models. These features are normalized using a robust scaler to handle outliers in the data. The system then computes Manhattan distances between feature vectors to determine initial similarity. For more precise alignment, it uses ICP (Iterative Closest Point) algorithms combined with PCA (Principal Component Analysis) to compare the 3D shapes more accurately. The results are ranked based on similarity scores, and parts with the highest matches are returned, enabling efficient reuse of designs in manufacturing. You can find an example in Figure 23.

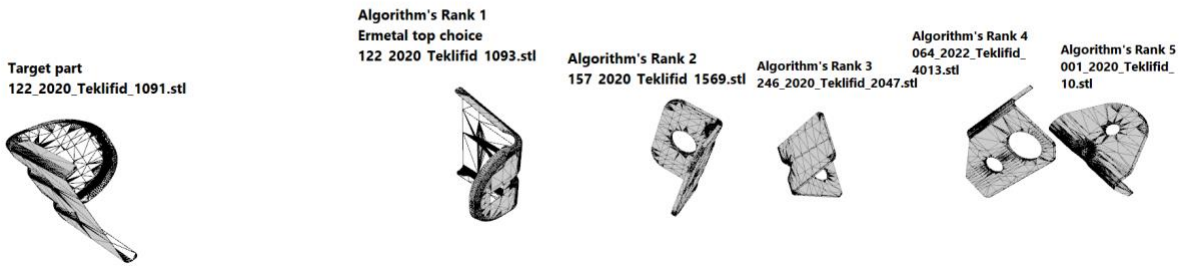


Figure 23: Example Result

3.8.4 Available Functions and APIs

User Interaction:
Users can upload 3D part files and view similarity analysis results through a web interface. The system provides interactive visualizations of the compared parts, along with similarity

scores and suggested reusability. This helps sales engineers and developers quickly identify which existing designs or components can be adapted for new projects.

APIs:
An API was developed. See Figure 24.

Figure 24: Part Similarity API

4 D3.4 Customer Segmentation (VTT)

Customer segmentation is the process of dividing the customer base to subsets based on common features. In the case of data-driven tools, this segmentation requires accurate data about the historical, present, and potential customers, which is often stored in Customer Relationship Management (CRM) systems. The most typical way to segment the customer base is to use simple features such as the domain of the client as well as their geographical location. However, segmentation can also consider various other features that are contained in the CRM systems.

4.1 Use case

In the InnoSale project customer segmentation functionalities are included in multiple use cases. In the *dynamic pricing* use case the goal is to streamline price updates and improve responsiveness to cost changes. Customer segmentation is used to analyse historical sales cases based on customer segments to set pricing accordingly. Customer segmentation relies on data from the CRM system, considering factors like industry, region, and customer ID. Success probability is evaluated using historical data specific to the customer and matching customer segments.

In the use case of *area based product proposal* the available products include a wide range of accessories and options tailored to specific market areas. The InnoSale tool streamlines sales expert orientation, reduces reliance on individual sales experts, and improves offer quality by considering area-specific trends and requirements. The customer segmentation component calculates customer and order similarity. It analyses order history data and compares it to customer information and order configuration details. Based on this analysis, the system identifies similar orders within the specific matching segment. Leveraging the best-matching order, the AI algorithm generates a proposed set of products with configuration parameters.

4.2 Inputs / Outputs

In this and the next sections the customer segmentation functionalities for the area based product proposal use case are presented.

Input: historical order data, customer location and number of households

Output: list of most prominent historical sales case IDs

4.3 Business logic

The data for the area-based product proposal was provided in an Excel spreadsheet. The data is first converted into machine readable format and filtered based on the country and postal code provided by the user. Each offer contains variable number of rows as the number of rows depends on the products included. The data is converted into hierarchical representation where each offer consists of subcomponents that specify the offered product and details in the configuration to enable further processing. Next frequency analysis is applied. The purpose of this step is to give each offer a numerical vector representation. Then the offers presented as vectors are clustered using density-based spatial clustering of applications with noise (DBSCAN) clustering method. For each cluster the most common configurations are searched and given as a result for recommended configurations.

Also, a method for evaluating the performance of the approach was developed. The evaluation is based on dividing the dataset into training and testing data based on time. The training data is used to create recommendations and the testing data is then used to see how many times the recommended configuration was used. This simulates the scenario where the salesperson asks for a recommended configuration based on the historical data collected until the present day and the success of the offer in the future is measured.

Experiments on different dataset for offer outcome prediction are presented in InnoSale D3.4 Customer Segmentation.

4.4 Available Functions and APIs

The product recommendation algorithm is provided as a Docker container. The API is implemented as an HTTP interface:

- HTTP POST request with JSON payload contains customer location information including country and postal code, and the number of households
- Response in JSON contains a list of best matching historical offer IDs, their timestamps (offer creation) and number of times a similar offer was made.

Example of a JSON request:

```
{
  "country":        "Finland",
  "postalCode":       "90100",
  "householdSize":       ”iso”
}
```

Example of a JSON response:

```
{
‟recommendation”:
[
{
  "offerid":                      "Kailame-01",
  "timestamps":                                [
        "2021-06-02T00:00:00.000000000",
        "2022-04-12T00:00:00.000000000",
        "2022-05-13T00:00:00.000000000‟
        ],
  "count":                                    3
},
{
  "offerid":                      "Kailame-31",
  "timestamps":                                [
        "2021-01-03T00:00:00.000000000",
        "2021-02-12T00:00:00.000000000‟
        ],
  "count":                                    2
}
```

5 D3.5 Optimal pricing (adesso)

5.1 3D Shape-based Pricing Strategies (Adesso)

5.1.1 Use case

3D Shape-based pricing service developed by Adesso is designed to be used by Ermetal to increase the efficiency of estimating the costs of new quotations, which can be easily extended into different use cases. This service provides a machine learning-based tool for predicting industrial product prices, specifically focusing on work-hour estimation for labour costs. The component integrates seamlessly with other applications through a dedicated API, enabling efficient and accurate pricing predictions based on 3D model data and operation parameters. This tool helps streamline cost estimation, allowing users to make informed pricing decisions quickly.

5.1.2 Inputs / Outputs

Inputs:

- **3D Model**: *.stl* file, which contains the part's geometric data.
- **Parameters**: Numerical and categorical values, including material thickness, type, surface area, hardness, and additional specifications related to each operation, such as mold dimensions and press types.

Outputs:

- **Predicted Labor Cost**: The output is a floating-point value that represents the estimated work hours required for production. This estimation is used to calculate the total labour cost for each operation.

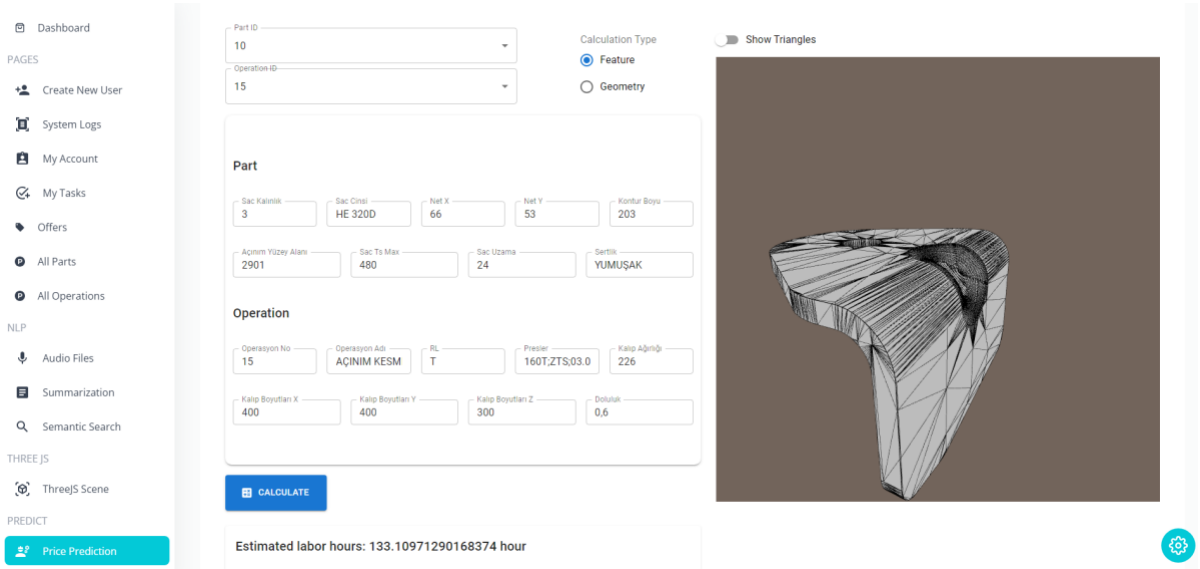


Figure 25: 3D Shape-based Pricing Component User Interface

The user will reach the service via the price prediction component added into the InnoSale user interface, in Figure 25 above, the component developed by Dakik for using the service can be seen. The user can input the data manually or populate the interface by selecting the part if it is already stored in the framework. It is possible to enable and disable the 3d data usage. The 3D model of the part is also shown at the side for supporting experts on decision making. When pressed the calculate button, the prediction is done with the best trained model (according to the validation result) and given as an estimation results.

5.1.3 Business logic

Once the user uploads a 3D model, the system extracts three primary features:

1. **Triangle Count**: Represents the complexity of the part.
2. **Total Surface Area**: Calculated by summing the areas of all triangles in the model.
3. **Volume**: Estimated by creating a voxelized point cloud of the model at a specified resolution.

The extracted features, along with additional parameters, are fed into a machine learning model. After experimenting with multiple algorithms, **LightGBM** was selected due to its superior performance in predicting work hours within the target error rate of less than 10%.

The predicted work hours are converted into a labour cost using a configurable multiplier, allowing the cost to be adjusted based on departmental rates or project-specific requirements. Various department-specific costs are also calculated proportionally to the work hours, providing a detailed cost breakdown for CAD, CAM, assembly, CNC processes, and other activities. This ratio can be set according to the desired rate per work hour, providing a flexible approach to labour cost calculation. (See example in Figure 26)

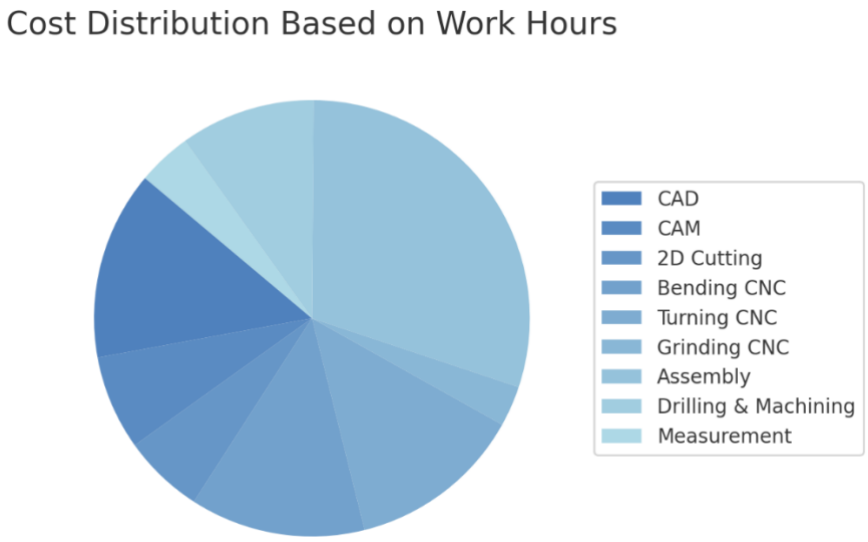


Figure 26: Example cost distribution based on the estimated work-hour value.

There are 2 AI models; 5_fold model and single model (located under saved_models). 5_fold model includes 5 models trained on 5 fold CV, single model is trained on whole data. It should be specified in models.py to which one to use.

5.1.4 Available Functions and APIs

The service is dockerized, and designed to be deployed into on premises as most of the data that will be processed by this component is confidential. If the docker service is deployed into the local machine, it will be running on “http://localhost:5000“, in which the port can be configured.

When the Industrial Price Prediction Tool is up and running, it is possible to access it through the API endpoint. Here's an example of how to make a prediction request using cURL:

```
curl -X POST -H "Content-Type:application/json"
http://127.0.0.1:5000/predict --data "{required_data}"
```

```
curl -X POST -H "Content-Type:application/json"
http://127.0.0.1:5000/costs --data "{required_data}"
```

This will return a JSON response with the predicted price based on the specified work hours.

5.1.4.1 Data Operations API End points

- Upload 3D Model (`POST /upload_3d_model`): Allows uploading of a .stl file containing 3D model data. The path to the file is specified in the request.
- Configure Cost Ratios (`POST /config/ratios`): Configures the multiplier ratios for calculating labour costs per department or process. Users can set ratios for areas such as CAD, CAM, Assembly, etc.
- Upload Tabular Data (`POST /data`): Accepts a CSV file with tabular data and a label column. The data is prepared for further processing.
- Get Dataset Statistics (`GET /data/stats`): Returns statistical information on an uploaded dataset, such as total rows, columns, and missing values.

5.1.4.2 Model Training API End points

- Train Model from Path (`POST /train_from_path`): Trains a new model using provided data files and saves it for future use. This endpoint requires paths to data files and a specified training type.
    - Example input data:

```
{
        "TrainingType" : "5-folds",
        "PartDataPath" : "raw/Parts.xlsx",
        "OperationPath" : "raw/Operations.xlsx",
        "3DFeaturesPath" : "raw/3D_features_pricing.csv",
        "SaveName" : "save_test"
}
```

- Start Training (`POST /train`): Begins the model training process on the uploaded dataset, if available.
- Query Training Status (`GET /train/status`): Retrieves the current status of the ongoing model training process, including progress and completion details.

5.1.4.3 Prediction API End points

- Predict Work Hours and Cost (`POST /predict`): Utilizes a trained machine learning model to predict work hours and calculate labour costs based on 3D model data and additional part/operation parameters.
    - Input Data Scheme:

```
{
  "title": "Industrial Labour Work Man-hour Prediction",
  "type": "object",
  "properties": {
    "part": {
      "type": "object",
      "properties": {
        "PathToSTL": {"type": "string", "description": "Path to the STL file"},
        "OfferNo": {"type": "string", "description": "Offer number or ID"},
        "OfferRevisionNo": {"type": "integer", "description": "Offer revision number"},
        "OfferId": {"type": "integer", "description": "Unique identifier for the offer"},
        "MaterialThickness": {"type": "number", "description": "Material thickness"},
        "MaterialType": {"type": "string", "description": "Type of material"},
        "NetDimensions": {
          "type": "object",
          "properties": {
            "NetX": {"type": "integer", "description": "Net width"},
            "NetY": {"type": "integer", "description": "Net height"}
```

```
          },
          "required": ["NetX", "NetY"]
        },
        "ContourLength": {"type": "integer", "description": "Contour length"},
        "SurfaceArea": {"type": "integer", "description": "Surface area in square units"},
        "MaxStrength": {"type": "integer", "description": "Maximum tensile strength"},
        "Elongation": {"type": "integer", "description": "Material elongation rate"},
        "Hardness": {
          "oneOf": [
            {"type": "null"},
            {"type": "string", "description": "Material hardness level"}
          ]
        }
      },
      "required": ["PathToSTL", "OfferId", "MaterialThickness", "NetDimensions", "ContourLength",
"SurfaceArea", "MaxStrength", "Elongation", "Hardness"],
      "additionalProperties": true
    },
    "operations": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "OfferNo": {"type": "string", "description": "Associated offer number"},
          "OfferRevisionNo": {"type": "integer", "description": "Offer revision number"},
          "OfferId": {"type": "integer", "description": "Unique identifier for the offer"},
          "OperationNo": {"type": "integer", "description": "Operation number"},
          "OperationType": {
            "type": "array",
            "items": {"type": "string"},
            "description": "List of operation types (flexible for custom entries)"
          },
          "RL": {
            "oneOf": [
              {"type": "null"},
              {"type": "string", "description": "Right/Left orientation (customizable)"}
            ]
          },
          "PressTypes": {"type": ["string", "null"], "description": "Type of press equipment used"},
          "MoldDimensions": {
            "type": "object",
            "properties": {
              "X": {"type": "integer", "description": "Mold width"},
              "Y": {"type": "integer", "description": "Mold height"},
              "Z": {"type": "integer", "description": "Mold depth"}
            },
            "required": ["X", "Y", "Z"]
          },
          "MoldWeight": {"type": "integer", "description": "Mold weight"},
          "Density": {"type": "number", "description": "Density or fill ratio of the mold"}
        },
        "required": ["OfferNo", "OfferId", "OperationNo", "OperationType", "MoldDimensions", "MoldWeight",
"Density"],
        "additionalProperties": true
      }
    }
  }
}
```

- Calculate Total Cost (`POST /cost`): Computes the total cost of manufacturing, factoring in parameters like material, coating, and labour.

- Load Model (`POST /load_model`): Loads a previously saved model to use in predictions, allowing flexible model management.
    - Example input data:

```
{
    "SaveName" : "5_fold_simple_op_seed_3"
}
```

- Show Current Model (`GET /models/show_enabled`): Displays the name of the currently loaded model used for predictions.
    - Example output data:

```
5_fold_simple_op_seed_3
```

- Show Available Models (`GET /models/show_available`): Lists all available models that can be loaded for predictions.
    - Example output data:

```
[
    "5_fold_simple_op_seed_3",
    "single_simple_op_seed_3"
]
```

See Figure 27 for available API.

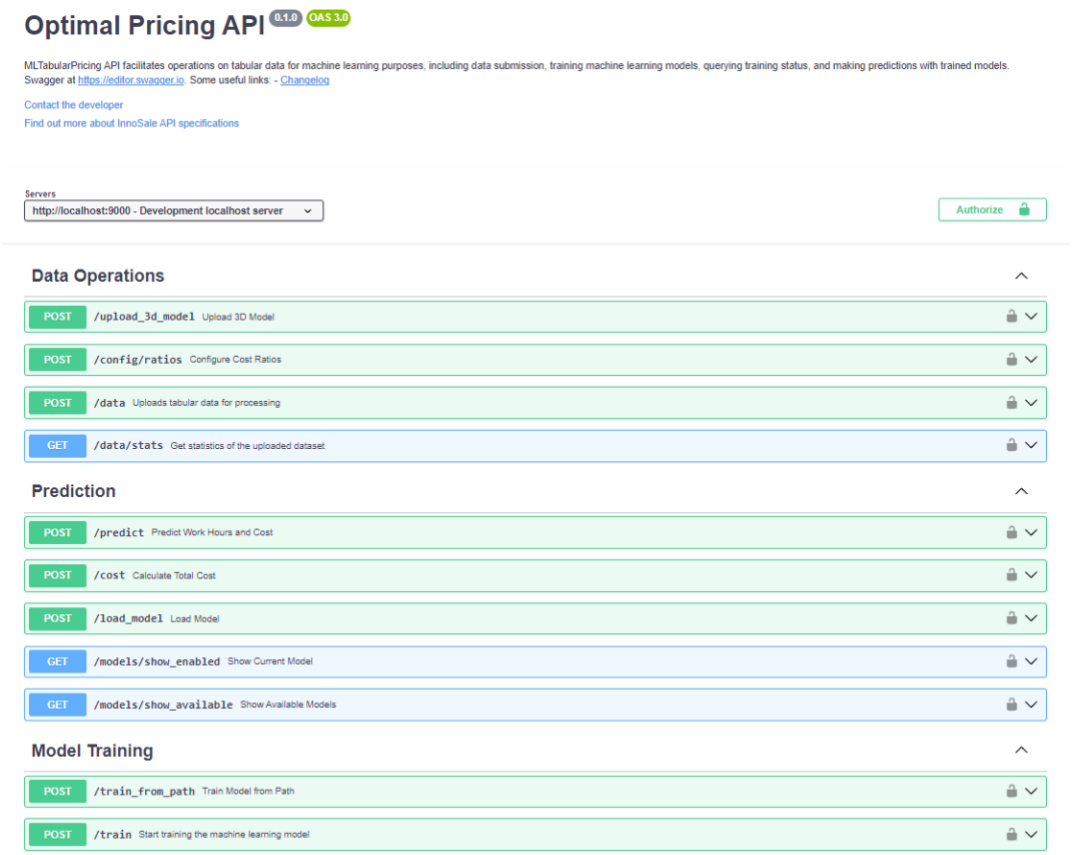


Figure 27: 3D Shape-Based Optimal Pricing API

5.2 Pricing Strategies based on Statistics and Forecasting (Software AG)

5.2.1 Use case

This chapter addresses innovative pricing strategies using artificial intelligence based on statistical analysis and forecasting. In the current workflow for adjusting the master price list, the prices must be defined by the product managers based on the available material, production and engineering costs as well as the target margin. This process is extremely time intensive and inflexible because all the information needs to be found manually from different sources and then implement these adjustments into the master price list. Moreover, a short-term anticipation of material cost development is difficult to achieve. Also, material cost forecast has less precision for long term forecast compared to short term forecast. This causes difficulties for defining the suitable adjustment rate. The AI-based approach pursues the automated adjustment of the master price list for the different product groups in light lifting equipment. Internal data and parameters for each material item are thoroughly analysed in advance and a forecast is created with the help of external data sources. Material consumption, production times, current and future purchase prices from suppliers, stock levels, production and engineering costs as well as logistical costs should be considered. Of particular interest is the cost development of certain volatile materials such as steel, copper and aluminium.

Real-time access to price developments enables an automated process for generating proposals for price adjustments or material surcharges. This approach considers key factors within the material price list and creates the basis for optimal pricing. The efficiency of the entire process is increased as less manual work is required. The improved quality of the forecasts leads to higher price quality and therefore stable margins. Customer centricity is driven by increased price transparency and the provision of convincing arguments for price adjustments. Customers gain an insight into the rational design of prices, which leads to more realistic price perceptions.

The primary objective of developing AI based forecasting service was to develop accurate forecasting models that can reliably predict the future prices of materials we are concerned with in production, aiding in effective decision-making and resource allocation for the production process. Leveraging advanced analytical techniques and machine learning algorithms, we aimed to identify key trends, patterns, and correlations within the dataset to create robust predictive models, by considering various external economic factors such as material prices, interest rates, and inflation, the goal is to provide an insight of potential price movements in the future.

5.2.2 Business logic

All calculations are done using pretrained models, which are accessible by the online service.

5.2.3 Available Functions, APIs, User Interactions

We implemented multiple functions that contributes to the overall functionality of the service that can be invoked by the end user.

5.2.3.1 Status

By invoking this function call: http://innosale.sagresearch.de:8012/status, you can easily check the current status of the server. This call provides real-time information, indicating whether the server is currently running or out of service.

5.2.3.2 Help

When you call http://innosale.sagresearch.de:8012/help, the end user is provided with a detailed list of all the arguments that can be passed to the prediction function, along with concise explanations for each. Here's what the function call returns:


```
{"st37":"Weight of ST37 in KG","p_st37":"Spot price of
ST37","p_high_carbon":"Spot price of High Carbon","alu":"Weight of Alu in
KG","labour":"Labor hours","high_carbon":"Weight of High Carbon in
KG","medium_carbon":"Weight of Medium Carbon in
KG","p_medium_carbon":"Spot Price of Medium
Carbon","p_nodular_cast_iron":"Spot Price of Nodular Cast
Iron","nodular_cast_iron":"Weight of Nodular Cast Iron in
KG","grey_cast_iron":"Weight of Grey Cast Iron in
KG","p_grey_cast_iron":"Spot Price of Grey Cast
Iron","nonalloy_cast":"Weight of Nonalloy Cast in
KG","p_nonalloy_cast":"Spot Price of Nonalloy Cast","months":"Forecasting
period in months (default is 24 months if argument not provided)"}
```


This comprehensive return allows users to understand the parameters they can manipulate and how each one influences the prediction model.

5.2.3.3 Calculate

This is the core functionality of the service we developed. For example, when calling http://innosale.sagresearch.de:8012/calculate/?copper=15&alu=12, you can input material weights, such as copper and aluminium, into the query. The function then predicts and returns the accumulated price of the product over the next two years. This powerful tool allows users to forecast costs with precision based on the materials used.

5.2.3.4 Spot price

The spot price is provided by using the spot-price flag, which is a floating-point number that precedes the alloy type. This flag allows the system to accurately represent the current market price for a specific alloy. By prefixing the alloy with its corresponding spot-price value, the function ensures that the most up-to-date pricing is factored into the overall cost calculations. This parameter is essential for generating precise forecasts and making informed decisions based on real-time market data.

http://innosale.sagresearch.de:8012/calculate/?st37=12&months=12&p_st37=20

5.2.3.5 Plot Price

The price with the above mentioned parameters may also be plotted as a graph (see Figure 28) for better visualization. All parameters are provided as mentioned above, but with the "plot" command.

http://innosale.sagresearch.de:8012/plot/?st37=500&copper=1&p_copper=1&labour=12

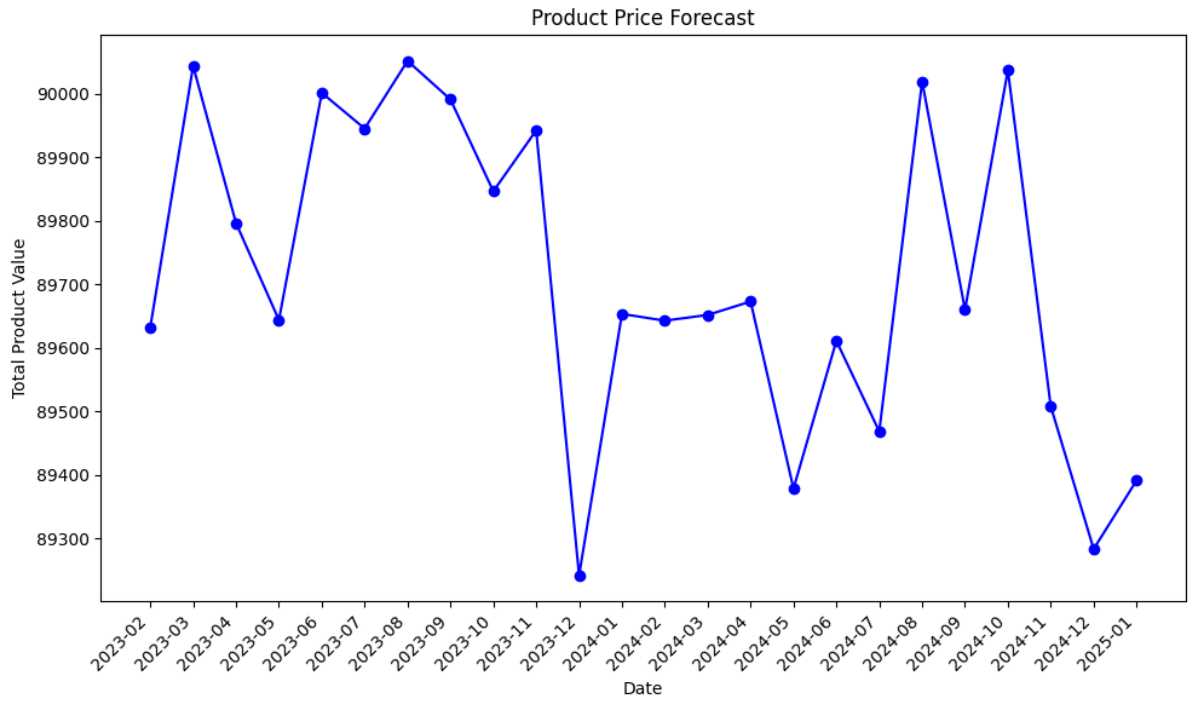


Figure 28: Product Price Forecast

5.3 Fuzzy Logic-based Pricing Strategies (ifak)

5.3.1 Use case

A Fuzzy Control Language Engine (FCLE) has been developed, which is an innovative tool for Fuzzy Logic applications, primarily designed for dynamic pricing systems. Adhering to the IEC 61131-7 standard, FCLE offers versatile use across various industries. It supports flexible rule management and seamless integration, enabling users to customize control strategies by easily modifying rule bases and input configurations. Beyond dynamic pricing, FCLE is suitable for diverse applications requiring fuzzy logic-based control algorithms.

Here is a more detailed list of the features of the FCLE:

- **IEC 61131-7 Compliance**: Is compatible to a subset of the IEC 61131-7 standard for Fuzzy Control Language, allowing for standardized evaluation of function blocks.
- **Comprehensive RESTful API**: Offers a robust API for configuring rule bases, input values, and dynamically triggering the evaluation of fuzzy logic rules.
- **Easy Installation and Setup**: Designed for straightforward installation with minimal configuration, enabling quick deployment.
- **Integration Ready**: Seamlessly integrates with existing GUIs and external systems, suitable for a wide range of industrial applications beyond dynamic pricing.
- **Flexible Rule Base Management**: Supports uploading, modifying, and retrieving rule bases in a standardized format, enhancing adaptability for various control strategies.
- **Dynamic Input Handling**: Capable of processing and evaluating input values from multiple sources, ensuring robust and responsive control algorithms.

An example Fuzzy Logic file, which is used in the test cases of the FCLE, looks like this:

```
FUNCTION_BLOCK DynamicPricing

  VAR_INPUT
    factory_load: REAL;   // Represents the current load on the factory (0 to 100%)
    market_demand: REAL;  // Represents the market demand level (0 to 100%)
    production_cost: REAL; // Represents the cost of production (0 to 100%)
  END_VAR

  VAR_OUTPUT
    pricing_factor: REAL; // The output factor used for dynamic pricing
  END_VAR

  FUZZIFY factory_load
    TERM low := (0, 1) (30, 1) (60, 0);
    TERM medium := (30, 0) (50, 1) (70, 0);
    TERM high := (60, 0) (100, 1);
  END_FUZZIFY

  FUZZIFY market_demand
    TERM low := (0, 1) (25, 1) (50, 0);
    TERM medium := (25, 0) (50, 1) (75, 0);
    TERM high := (50, 0) (100, 1);
  END_FUZZIFY

  FUZZIFY production_cost
    TERM low := (0, 1) (20, 1) (40, 0);
    TERM medium := (20, 0) (50, 1) (80, 0);
    TERM high := (60, 0) (100, 1);
  END_FUZZIFY

  DEFUZZIFY pricing_factor
    TERM low := 0;
```

```
    TERM medium := 50;
    TERM high := 100;
    METHOD: CoGS;
    DEFAULT := 0;
  END_DEFUZZIFY

  RULEBLOCK No1
    AND: MIN;
    ACCU: MAX;

    RULE 1: IF factory_load IS high THEN pricing_factor IS high;
    RULE 2: IF market_demand IS high THEN pricing_factor IS high;
    RULE 3: IF production_cost IS high THEN pricing_factor IS high;
    RULE 4: IF factory_load IS medium AND market_demand IS medium
        THEN pricing_factor IS medium;
    RULE 5: IF factory_load IS low AND market_demand IS low AND production_cost IS low
        THEN pricing_factor IS low;
    RULE 6: IF factory_load IS medium AND market_demand IS low
        THEN pricing_factor IS medium;
    RULE 7: IF production_cost IS medium THEN pricing_factor IS medium;

  END_RULEBLOCK

END_FUNCTION_BLOCK
```

5.3.2 Inputs / Outputs

Basically you need following static data, which can be uploaded by use of the API:

- Rulebase: A fuzzy logic file as provided in the “Use case” section.
- Service configuration: This file is read at start of the service. It contains information about the binding address of the service, the IP port and some variable addresses. Those variable addresses are currently not evaluated. The respective functionality will be added in demonstrators (WP6). Thus currently all variable values appearing as input variables in the rulebase, need to be transferred when calling the ‘/evaluate’ API call.

Here is an example of the service configuration:

```
engine_server:
  server_name: engine_server
  host: 0.0.0.0
  port: 8000
variables:
  factory_load:
    endpoint: http://localhost:8080/get_factory_load
    header_params:
      days: 14
  market_demand:
    endpoint: http://localhost:8080/get_market_demand
    header_params:
      days: 0
      product: KBK
```

5.3.3 Business logic

Let’s explain the business logic by use of an example. We assume to have a fuzzy logic rules base like the following:

IF factory_load(high) THEN pricing_factor(quite_high).

IF customer_buys(frequently) THEN pricing_factor(quite_low).

Here, factory load and customer buys are price influencing parameters. We have to gather them from the IT systems of the manufacturer. Both rules have influence on the target

variable final_price_factor. Therefore, the inference engine needs to be capable to handle conflicting formulas. It is likely, that factory_load and customer_buys but also the target variable final_price_factor have numeric value representations. The parameters need to be transformed into a representation, which is applicable for a fuzzy logic inference engine by fuzzyfication. Vice versa, the fuzzy logic representation of pricing_factor has to be converted back by de-fuzzyfication.

5.3.4 Available Functions, APIs, User Interactions

If you have started the FCLE, then an interactive API documentation is available at:

- Swagger UI: http://localhost:8000/docs
- Redoc: http://localhost:8000/redoc

These interfaces provide detailed information about each endpoint, expected parameters, and response formats. The OpenAPI specification is also available at http://localhost:8000/redoc.

If you have started FCLE on a remote host, then replace “localhost” with the IP address of that host computer. In the following, we describe detailed examples of how to interact with the FCLE API using `curl`. Also here, replace `localhost` with your server’s address if it’s running elsewhere.

1. Evaluate Endpoint

Endpoint: `POST /evaluate`

Evaluates the fuzzy logic rules based on provided input values.

Example Request:

```
curl -X POST "http://localhost:8000/evaluate" \
     -H "Content-Type: application/json" \
     -d '{"input_values": {"factory_load": 75, "market_demand": 60,
"production_cost": 50}}'
```

Note: Make sure that those 3 input values are defined in the rulebase. If necessary parameters are missing, you will get an error. An example rulebase is given in the section “Use case” above.

Headers:

`Content-Type: application/json`

Body:

`input_values`: JSON object containing input variables.

Example Response:

```
{
  "status": "success",
  "timestamp": "2024-09-02T12:34:56Z",
  "request_id": "abcd1234",
  "input_params": {
    "factory_load": 75,
    "market_demand": 60,
    "production_cost": 50
  },
  "output_values": {
    "pricing_factor": 1.23
  },
```

```
  "message": "Pricing factor successfully calculated."
}
```

Explanation:

- Input Variables:
  - `factory_load`: Current load on the factory (0 to 100%, or more with extra shifts).
  - `market_demand`: Market demand level (0 to 100, 50 is normal demand).
  - `production_cost`: Cost of production (0 to 100, 50 is normal production cost).
- Output Variables:
  - `pricing_factor`: Calculated pricing factor based on fuzzy logic rules.

2. Set Rulebase Endpoint

Endpoint: `POST /set_rulebase`

Uploads a new rule base in FCL (Fuzzy Control Language) format.

Example Request:

```
curl -X POST "http://localhost:8000/set_rulebase" \
     -H "Content-Type: application/json" \
     -d '{"rulebase": "'$(base64 -w 0 path_to_your_rulebase.fcl)'"}'
```

Headers:

`Content-Type: application/json`

Body:

`rulebase`: Base64-encoded string of your `rulebase.fcl` file.

Example Response:

```
{
  "status": "success",
  "timestamp": "2024-09-02T12:35:10Z",
  "message": "Rule base successfully updated."
}
```

Explanation:

This endpoint replaces the existing rule base with the provided one.

Ensure your FCL file is correctly formatted and encoded.

3. Get Rulebase Endpoint

Endpoint: `GET /get_rulebase`

Retrieves the current rule base in use.

Example Request:

```
curl -X GET "http://localhost:8000/get_rulebase"
```

Example Response:

```
{
  "status": "success",
  "timestamp": "2024-09-02T12:35:30Z",
  "rulebase": "<base64_encoded_rulebase>",
  "message": "Rule base successfully retrieved."
}
```

Explanation:

The `rulebase` field contains the Base64-encoded FCL content.

Decode using:

```
echo "<base64_encoded_rulebase>" | base64 -d > retrieved_rulebase.fcl
```

4. Set Service Configuration Endpoint

Endpoint: `POST /set_service_conf`

Updates the service configuration.

Example Request:

```
curl -X POST "http://localhost:8000/set_service_conf" \
     -H "Content-Type: application/json" \
     -d '{"service_conf": "'$(base64 -w 0
path_to_your_service_conf.yaml)'"}'
```

Headers:

`Content-Type: application/json`

Body:

`service_conf`: Base64-encoded string of your `service_conf.yaml` file.

Example Response:

```
{
  "status": "success",
  "timestamp": "2024-09-02T12:35:50Z",
  "message": "Service configuration successfully updated."
}
```

Explanation:

- Updates the service configuration parameters.
- Useful for changing server settings or variable endpoints.

5. Get Service Configuration Endpoint

Endpoint: `GET /get_service_conf`

Retrieves the current service configuration.

Example Request:

```
curl -X GET "http://localhost:8000/get_service_conf"
```

Example Response:

```
{
  "status": "success",
  "timestamp": "2024-09-02T12:36:10Z",
  "service_conf": "<base64_encoded_service_conf>",
  "message": "Service configuration successfully retrieved."
}
```

Explanation:

The `service_conf` field contains the Base64-encoded YAML content.

Decode using:

```
echo "<base64_encoded_service_conf>" | base64 -d >
retrieved_service_conf.yaml
```

6 D3.6 Inference Engine (TUD)

6.1 Use case

InnoSale aims to optimize the sales process of specialized and therefore highly customizable products. As their complexity does not allow for a complete representation within a catalogue, customer requests usually arrive as unstructured, free-form text that often omits crucial details. Such cases require substantial back-office support, relying on the expertise of sales engineers to infer missing information and to spot inconsistencies. Towards automating this task, deductive rule-based systems offer clear advantages. By encoding expert knowledge as simple if-then statements, it is possible to validate and to complete customer requests in an understandable and explainable manner. Systems using this approach are easy to maintain and can scale effectively as rules provide a clear, logical structure that is easy to understand, even for people without programming experience.

Existing solutions in this space are either limited in scope [1], do not scale well in terms of performance [2], or are closed-sourced commercial projects [3], [4]. We therefore develop Nemo, an open-source rule-based reasoning toolkit [5], [6]. Its rule-language is based on the recursive query language Datalog, extended with various features to accommodate the use cases of InnoSale. In addition, Nemo provides a convenient web interface and supports the Language Server Protocol enabling efficient rule editing within compatible editors. Furthermore, Nemo can explain why certain facts were inferred, which improves the trustworthiness of the system. Experiments show that it outperforms existing systems on common benchmarks.

6.2 Inputs / Outputs

Nemo expects two kinds of inputs:

- A rule file specifying expert knowledge as if-then rules formatted in the Nemo language, which is described in more detail in Deliverable D3.6
- Relational data in CSV, RDF, or JSON format

Nemo computes the logical consequences of the rules given the provided data and exports it in either CSV or RDF format.

6.3 Business logic

At its core, Nemo is a fast and scalable reasoner that materializes logical consequences using semi-naïve bottom-up evaluation [7]. Existential rules are evaluated using the restricted (also known as standard) chase [8].

Data is maintained in-memory, representing tables as hierarchically sorted, column-based tables, following the design of VLog [1]. However, unlike VLog, Nemo can handle data of various types. This includes fixed-sized values, such as integer or floating-point numbers, which are stored directly, and variable-sized data such as strings, which are mapped to integers and accessed via a dictionary. Data is compressed using Run-Length-Encoding with increments.

Rules are evaluated by compiling them into a series of relational algebra operations such as joins or unions. They are implemented by providing trie iterators [9].

6.4 Available Functions and APIs

The standard way to use Nemo is as a command-line application. After installation (described in more detail in Deliverable D3.6), Nemo can be run via the following command:

```
./nmo <RULES>
```

Executing the above command computes all inferences implied by the provided rule file, which may reference additional input data via import directives, and saves the result according to the export statements. Further options can be provided by the user:

- **export:** Override export directives in the program
- **export-dir:** Base directory for exporting files
- **overwrite-results:** Replace any existing files during export
- **gzip:** Use gzip to compress exports by default; does not affect export directives that already specify a compression
- **import-dir:** Base directory for importing files (default is working directory)
- **trace:** Facts for which a derivation trace should be computed; multiple facts can be separated by a semicolon, e.g. "P(a, b);Q(c)"
- **report:** Control amount of reporting printed by the program

In addition, Nemo provides a Python API, which exposes data import and export, the reasoner, and gives access to the internal rule structure.

6.5 References

[1] D. Carral, I. Dragoste, L. González, C. Jacobs, M. Krötzsch, and J. Urbani, “VLog: A Rule Engine for Knowledge Graphs,” in *The Semantic Web – ISWC 2019*, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, Eds., Cham: Springer International Publishing, 2019, pp. 19–35. doi: 10.1007/978-3-030-30796-7_2.

[2] K. Angele, J. Angele, U. Şimşek, and D. Fensel, “RUBEN: A Rule Engine Benchmarking Framework”.

[3] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee, “RDFox: A Highly-Scalable RDF Store,” in *The Semantic Web - ISWC 2015*, vol. 9367, M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d’Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab, Eds., in Lecture Notes in Computer Science, vol. 9367. , Cham: Springer International Publishing, 2015, pp. 3–20. doi: 10.1007/978-3-319-25010-6_1.

[4] L. Bellomarini, E. Sallinger, and G. Gottlob, “The Vadalog system: datalog-based reasoning for knowledge graphs,” *Proc. VLDB Endow.*, vol. 11, no. 9, pp. 975–987, May 2018, doi: 10.14778/3213880.3213888.

[5] A. Ivliev *et al.*, “Nemo: First Glimpse of a New Rule Engine,” in *Proceedings 39th International Conference on Logic Programming (ICLP 2023)*, E. Pontelli, S. Costantini, C. Dodaro, S. Gaggl, R. Calegari, A. D. Garcez, F. Fabiano, A. Mileo, A. Russo, and F. Toni, Eds., in EPTCS, vol. 385. Sep. 2023, pp. 333–335. doi: 10.4204/EPTCS.385.35.

[6] A. Ivliev, L. Gerlach, S. Meusel, J. Steinberg, and M. Krötzsch, “Nemo: Your Friendly and Versatile Rule Reasoning Toolkit,” in *Proc21st*,

[7] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995. [Online]. Available: http://webdam.inria.fr/Alice/

[8] M. Benedikt *et al.*, “Benchmarking the Chase,” in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, Chicago Illinois USA: ACM, May 2017, pp. 37–52. doi: 10.1145/3034786.3034796.

[9] T. L. Veldhuizen, “Leapfrog Triejoin: a worst-case optimal join algorithm,” Dec. 20, 2013, *arXiv*: arXiv:1210.0481. Accessed: Sep. 25, 2024. [Online]. Available: http://arxiv.org/abs/1210.0481