Innovating Sales and Planning of Complex Industrial Products
Exploiting Artificial Intelligence

Deliverable 3.3
Knowledge Base

Deliverable type:	Software
Deliverable reference number:	ITEA 20054 | D3.3
Related Work Package:	WP 3
Due date:	2024-05-31
Actual submission date:	2024-07-17
Responsible organisation:	Dakik Software
Editor:	Bilge Özdemir
Dissemination level:	Confidential
Revision:	Final | Version 1.0

Abstract:	Knowledge base develops data association models and Bayesian models to find correlations and dependencies between product variants and related products that are not identified by the experts. Knowledge base provides the T3.6 Inference Engine with data and validation rules to be specified in the project.
Keywords:	Historical data, Sales data, Database management, Document-oriented database, Relational database, Semantic search, Relation extraction, Fuzzy logic rules, Pricing rules, Entity recognition, Speech recognition, Meeting summarization, Deductive reasoning

Table_head	Name 1 (partner)	Name 2 (partner)	Approval date (1 / 2)
Approval at WP level	ADESSO	TARAKOS	2024-06-14
Veto Review by All Partners			2024-07-17

Editor

Bilge Özdemir (DAKIK)

Contributors

Stefan Ellmauthaler (TUD)
Alex Ivliev (TUD)
Sepideh Sobhgol (IFAK)
Thomas Marus (DEMAG)
Bilge Özdemir (DAKIK)
Mehtap Öklü (DAKIK)
Mert Daloğlu (DAKIK)
Talha Rehman Abid (DAKIK)
Sebastian Wendt (:em AG)
Mario Thron (ifak)
Nico Herbig (Natif)

Executive Summary

Knowledge Base plays a vital role in the InnoSale by serving as a centralized repository to capture and formalize expertise across various sales domains. It leverages ontology-based rules to represent product variants, configuration constraints, and dependencies, incorporates historical sales data from partners, structured in databases like MongoDB and MySQL, and accessible via APIs. Knowledge Base also supports managing fuzzy logic rule sets for situational pricing, providing APIs to upload, retrieve, and modify these rules. Overall, it establishes a robust foundation by integrating data sources, formalizing domain knowledge through ontologies and rules, and providing structured access to this curated knowledge for intelligent sales decision support.

Table of Content

Figures

1 Introduction

Knowledge Base serves as a repository for domain-specific knowledge and provides interfaces for the Inference Engine to access data and validation rules. KB acts as a facade, providing a unified interface to access various storage APIs implemented by different InnoSale partners. Its primary role is to consolidate and centralize access to the diverse knowledge sources and services developed within the project.

Knowledge Base incorporates historical data management from various use cases, outlining the database structures, data formats, and technologies used for storing and retrieving relevant information.

A significant aspect of the Knowledge Base is the ontology development process. It employs two complementary approaches: a semi-automatic approach for constructing ontologies and the utilization of upper/domain ontologies. The semi-automatic approach combines synonym terms to form concepts and defines relationships between these concepts. The upper/domain ontology approach establishes a hierarchical structure, with an upper ontology providing a foundational set of concepts and domain ontologies addressing specific contextual needs.

Knowledge Base also includes fuzzy logic rules, which encapsulate nuanced knowledge for determining optimal pricing for complex products. These rules are crafted by experts, taking into account various influence factors such as customer relationships, competitive pressure, and manufacturer workload.

Additionally, it covers deductive reasoning rules using the Nemo language, entity recognition techniques, speech recognition and meeting summarization parameters, and similarity analysis methods. These components contribute to the overall functionality and decision-making capabilities of the Knowledge Base.

2 Historical Data Management

Knowledge Base provides an effective management of historical data provided by the project's use case partners. This historical data, encompassing details such as past sales orders, product configurations, customer inquiries, and project files, serves as a valuable foundation for the InnoSale project. The following section explores the various types of data shared by partners, how this data is structured within databases, and the mechanisms in place to facilitate access to this information for other components through well-defined APIs.

The historical data management approach aims to consolidate and organize these diverse data sources, enabling the Knowledge Base to provide a comprehensive and integrated view of the available information. By standardizing data formats and structures, Knowledge Base can effectively support data association models, Bayesian models, and other analytical techniques to uncover hidden correlations and dependencies, further enriching the knowledge base over time.

2.1 LLE

Database input data/facts

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.

As this 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 1 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.

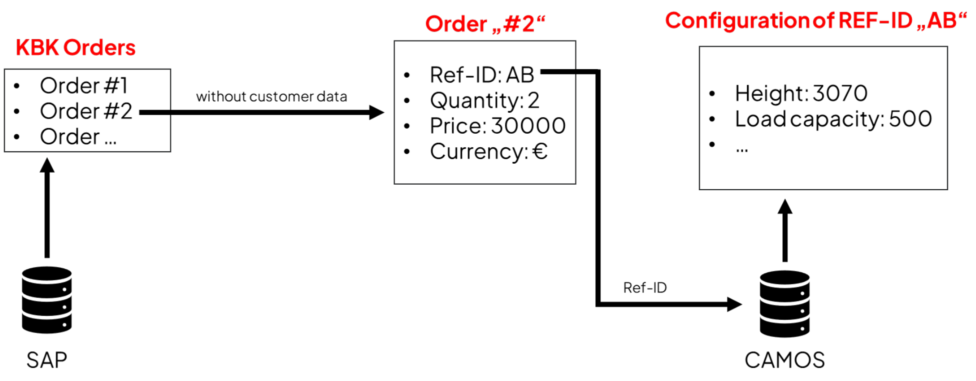


Figure 1: Origin of the historical Datasets from Demag

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.

Hata! Başvuru kaynağı bulunamadı. shows the entire planned data flow within the first demonstrator. For the planned demonstrator, the dummy database will primarily serve the purpose of giving partners access to some of our historical sales cases and thus being able to apply their solution approaches. Each partner ultimately accesses the data for their own purposes and tries to contribute their solution to the demonstrator. 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).

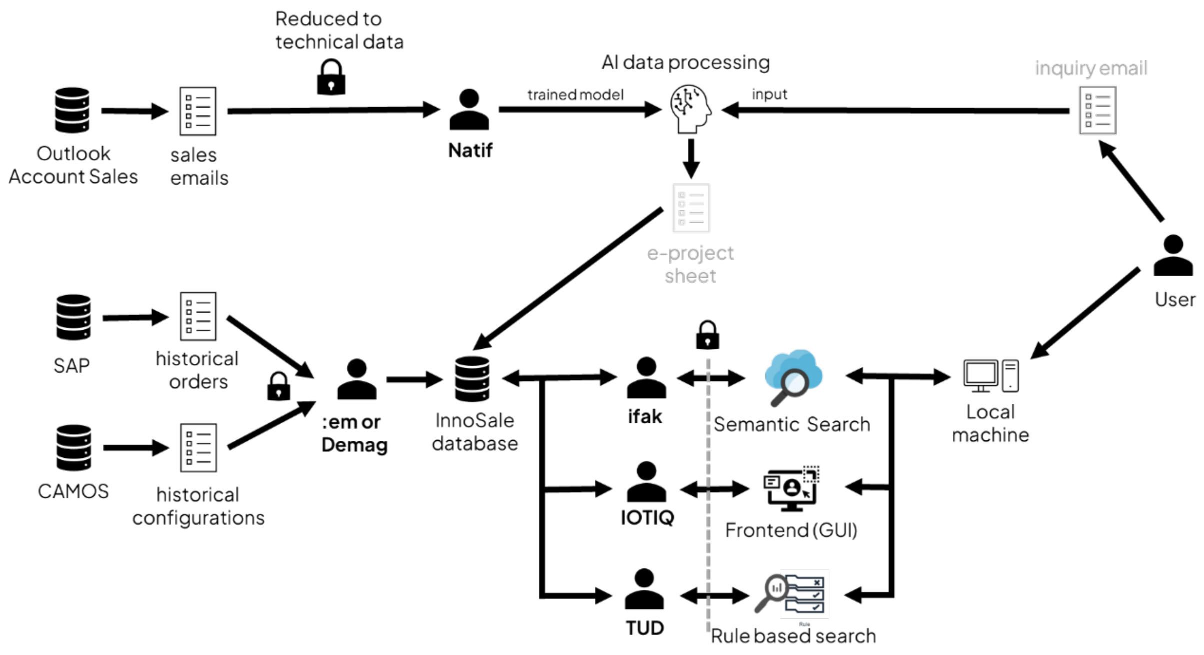
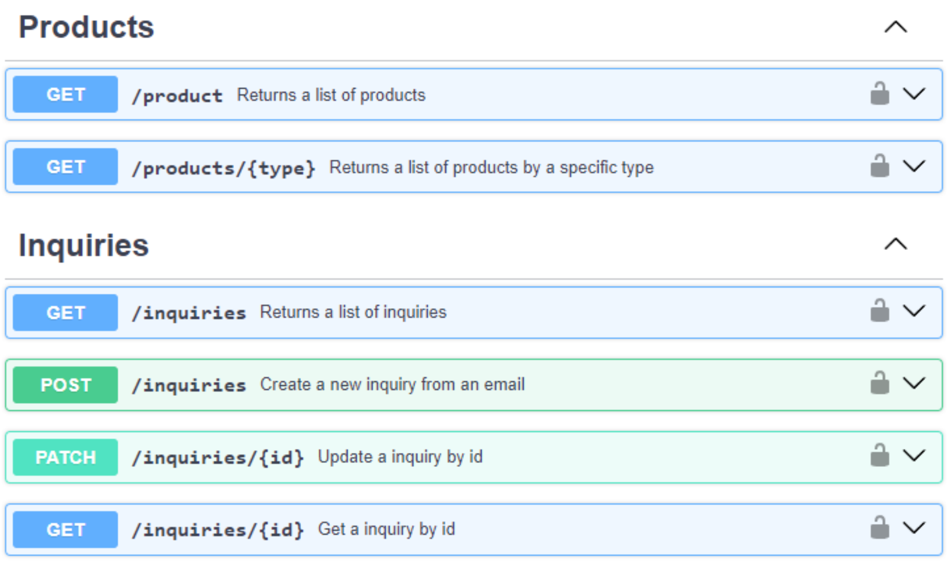


Figure 2: Dataflow for LLE-UC 1,3,4,5

In addition, for another demonstrator regarding situational pricing, knowledge from Demag's sales engineers is made available for the fuzzy logic approach to define variables and fuzzy logic rules. This information will not be part of the earlier described database and will be provided directly to ifak, due to the reason it is not for interest of the other partners.

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 will provide 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 data is then made accessible to IFAK, IOTIQ and TUD secured via a company specific API token for each partner.


Products

GET /product Returns a list of products

GET /products/{type} Returns a list of products by a specific type

Inquiries

GET /inquiries Returns a list of inquiries

POST /inquiries Create a new inquiry from an email

PATCH /inquiries/{id} Update a inquiry by id

GET /inquiries/{id} Get a inquiry by id

2.2 CTO

Knowledge Base for CTO supplies the Inference Engine with pertinent data, including records of previous service tickets, material costs, and details about customer environments. Additionally, Knowledge Base receives supplementary background information from external data sources through the "update background information" interface, facilitated by the knowledge acquisition component. However, no specific services are being developed within this particular component.

2.3 WM

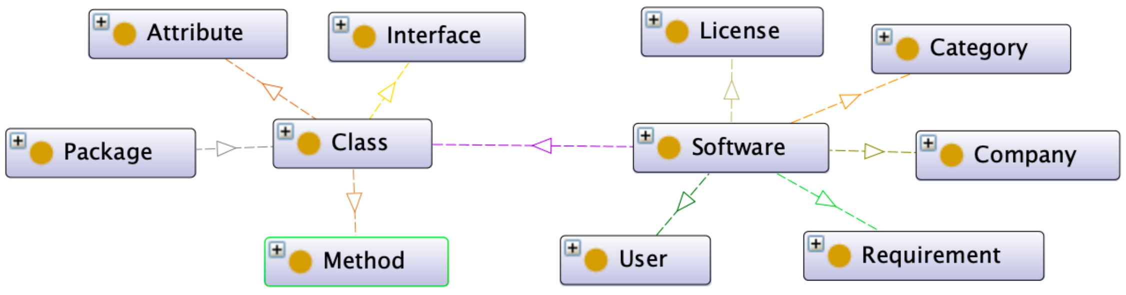
Knowledge Base in WM use case acts as a repository, supplying the Inference Engine with information such as past sales records, customer details, campaign data, and applicable rules and regulations. The Summium CPQ configurator platform developed by Wapice serves as the database for this purpose. Knowledge Acquisition Component is responsible for populating the Knowledge Base with the required data from external sources. However, no specific services are being developed within this component itself.

2.4 DPM

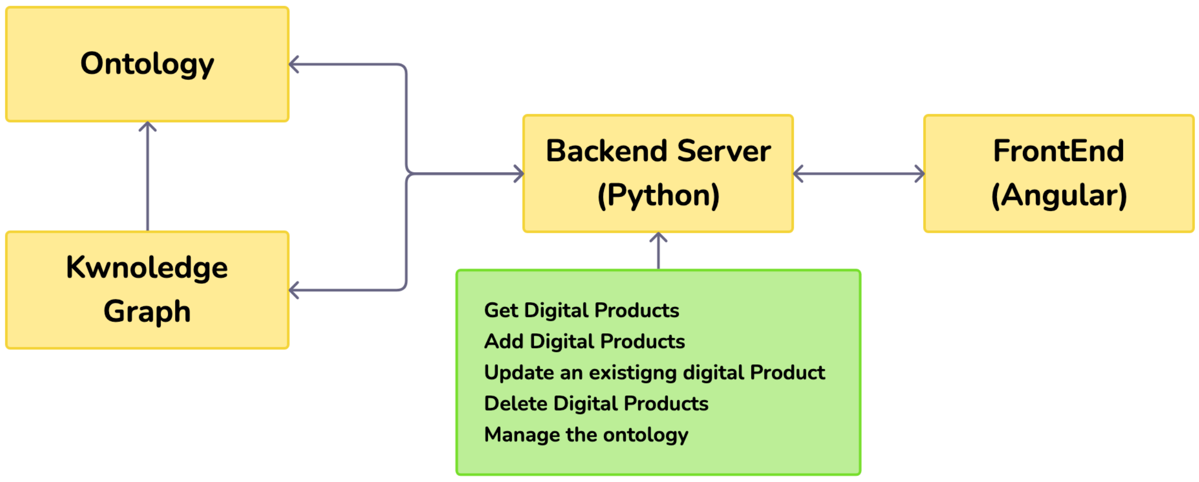
Digital Product Marketplace has a database containing internal digital products from Panel and Softtekt. These internal digital products include the following details:

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

The ontology is structured as follow:



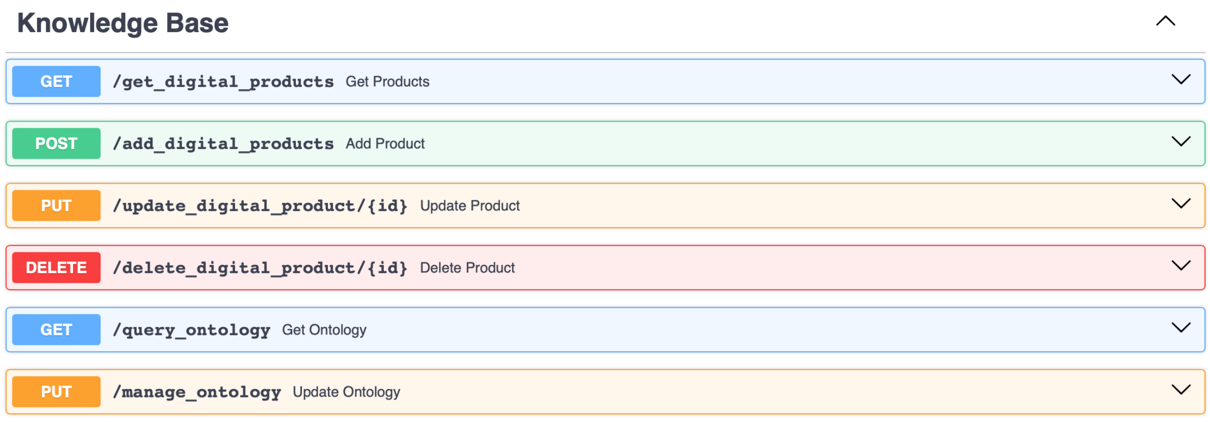
In this use case, the information is stored in a knowledge graph implemented using Neo4J, a graph database management system. To manage the knowledge graph, we have developed an API using FastAPI.



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.

The API developed with FastAPI exposes endpoints for interacting with the knowledge base:


Knowledge Base

GET `/get_digital_products` Get Products

POST `/add_digital_products` Add Product

PUT `/update_digital_product/{id}` Update Product

DELETE `/delete_digital_product/{id}` Delete Product

GET `/query_ontology` Get Ontology

PUT `/manage_ontology` Update Ontology

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.

2.5 SM

The database in 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.

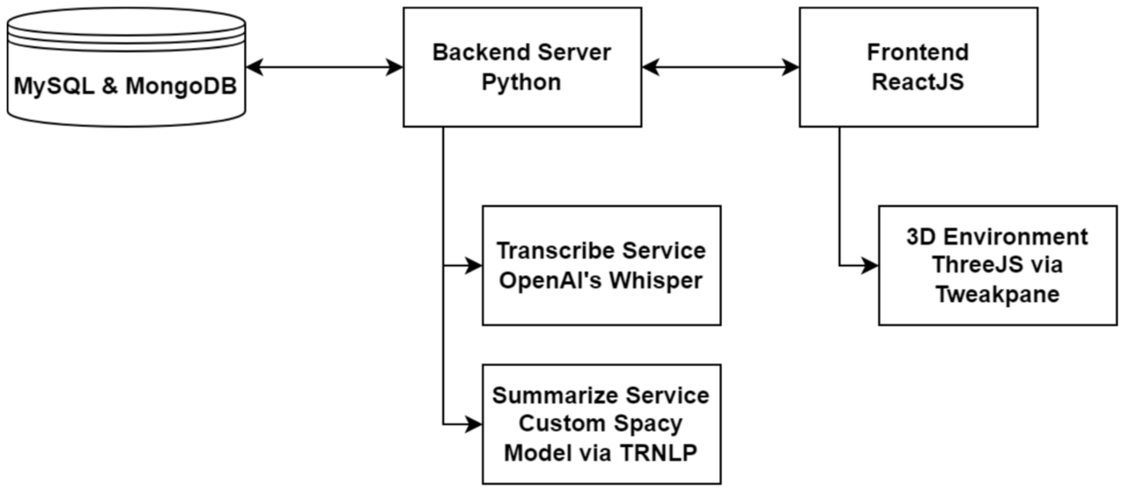

MySQL & MongoDB
Backend Server Python
Frontend ReactJS
Transcribe Service OpenAI's Whisper
3D Environment ThreeJS via Tweakpane
Summarize Service Custom Spacy Model via TRNLP

Figure 3 Technologies Used

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.

3 Ontology Development

We have created an ontology to bridge the gap between the terminology used by sales engineers and customer inquiries. This ontology will be used for the semantic search of project files and historical data. Details on the ontology approach can be found in D3.1, while information on its use in ontology-based semantic search is available in D3.2.

We employ two complementary approaches to maximize the effectiveness in the InnoSale use cases: Semi-Automatic approach of constructing an Ontology and Upper/Domain Ontology.

3.1 Semi-Automatic approach of constructing an Ontology

The following figures provide an overview of the data structure of the ontology. They illustrate how synonyms combine to form concepts and how these concepts can have various types of relationships with each other, including "element-of" and "abstract-specification." As previously mentioned, the details on relation extraction are covered in D3.1.

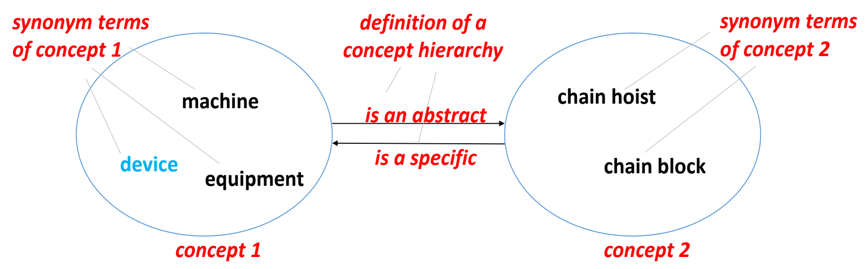


Figure 4: Terms, Concepts, Relations

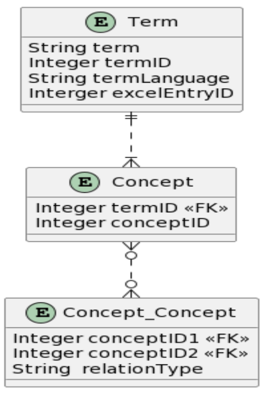


Figure 5: Ontology Data Model

The data structure is stored in Sqlite, which functions as a relational database. This database could be stored within the knowledge base component. Therefore, some functions need to be developed to access the database for loading it, editing the data, and saving the updated database back into the knowledge base. Assuming there is a user interface that provides access to the knowledge base, at least two functions need to be implemented to utilize the ontology database. These functions could be `load_ontology()` and `save_ontology(sqlite_database_file)`. To efficiently update the database, we often need several quick accesses. Therefore, when we load the database, we should be able to load it onto a local computer, edit it, and then save it back into the knowledge base all at once. This process provides quicker and faster access. Given that the database file is small, loading the entire file should not be problematic.

Usage examples

A) Load Ontology Example

To load the ontology from the knowledge base, you can use the `curl` command to access the HTTP-based REST-API. The following example demonstrates how to use the `load_ontology` route:

```
# Download the ontology using curl
curl -X GET http://localhost:8080/api/v1/ontology \
   -H "Authorization: Bearer YOUR_ACCESS_TOKEN" \
   -H "Accept: text/plain" \
   -o ontology_base64.db

# Decode the downloaded base64 file
base64 --decode ontology_base64.db > ontology.db
```

B) Save Ontology Example

To save the updated ontology back into the knowledge base, you can use the `curl` command to access the `save_ontology` route:

```
# Encode the ontology file in base64
base64 ontology.db > ontology_base64.db

# Upload the encoded ontology file using curl
curl -X POST "http://localhost:8080/api/v1/ontology" \
   -H "Authorization: Bearer YOUR_ACCESS_TOKEN" \
   -H "Content-Type: text/plain" \
  -d @ontology_base64.db
```

Configuration

The configuration of the knowledge base service is managed via a YAML file. Below is an example configuration:

```
server:
  port: 8080
  basePath: /api/v1

knowledgeBase:
  ontologyPath: /data/ontology
  security:
    oauth2:
      authorizationUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/auth
      tokenUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/token
      clientId: ontology-knowledgebase
      clientSecret: your-client-secret
```

The “server” section contains typical HTTP server information. The “ontologyPath” is the location of the file ontology_base64.db at the storage server (the knowledge base). It is a relative path starting at the root of the server executable. The security part contains typical OAuth2 information.

Formal Specification

The formal specification of the ontology management API uses the OpenAPI (version 3.0.0) standard. Below is the example specification for the ontology routes. Therein, you need to update the server’s URL and the security server URLs accordingly:

```
openapi: 3.0.0
info:
  title: Ontology Management API
  version: 1.0.0
  description: API for managing ontology in the knowledge base.
servers:
  - url: http://localhost:8080/api/v1
paths:
  /ontology:
    get:
      summary: Load the ontology
      operationId: loadOntology
        responses:
        '200':
          description: Ontology retrieved successfully
          content:
            text/plain:
              schema:
                type: string
                description: Base64 encoded ontology SQLite database file
        '400':
          description: Bad request
        '403':
          description: Forbidden
        '404':
          description: Not found
        '500':
          description: Internal server error
      security:
        - oauth2: []
  /ontology/save:
    post:
      summary: Save the ontology
      operationId: saveOntology
      requestBody:
        required: true
        content:
          text/plain:
            schema:
              type: string
              description: Base64 encoded ontology SQLite database file
      responses:
        '200':
          description: Ontology saved successfully
        '400':
          description: Bad request
        '403':
          description: Forbidden
        '500':
          description: Internal server error
      security:
        - oauth2: []
components:
  securitySchemes:
    oauth2:
      type: oauth2
      flows:
        password:
          authorizationUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/auth
          tokenUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/token
          scopes: {}
```

3.2 Upper/Domain Ontology

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.

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.

Several functions need to be developed to handle the ontology and knowledge graph effectively, 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.

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

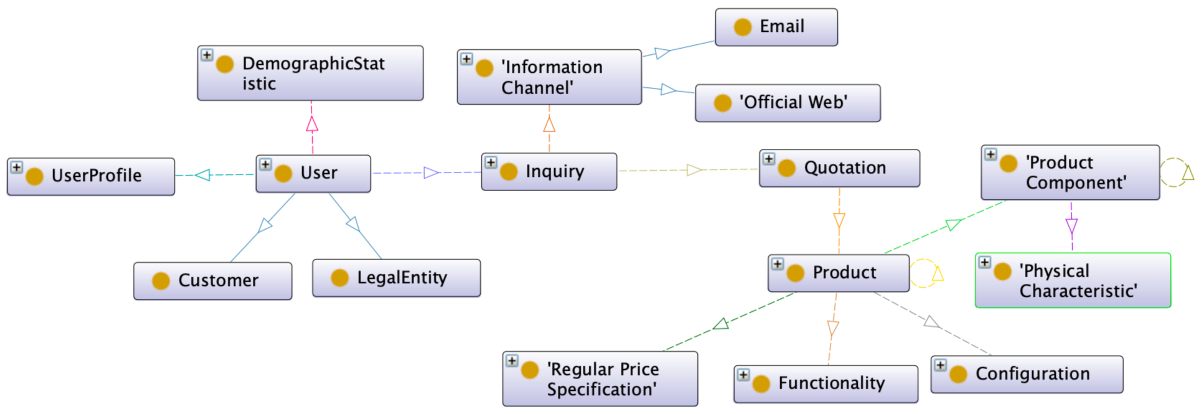


Figure 6 Upper Ontology Structure

4 Fuzzy Logic Rules

In the competitive landscape of industrial sales, determining the optimal pricing for complex products requires an intricate understanding of various influence factors. These factors include the duration of the customer relationship, competitive pressure in the customer's region, and the current workload of the manufacturer. To encapsulate this nuanced knowledge, fuzzy logic rules are employed. These rules are crafted by the CEO, the head of sales or other sales experts, leveraging their expertise to adapt master prices for specific customers. It is comparable with writing discount tables in a more traditional sales process. The purpose of this chapter is to describe the management of fuzzy logic rule sets within the InnoSale project, specifically focusing on the knowledge base that provides functionality to store and retrieve these rules as indicated in deliverable D2.2. The rule sets are text files with a syntax according to IEC 61131-7 (Fuzzy Control Language).

The knowledge base is implemented as a set of HTTP-based REST services. These services can be deployed on a single computer, across multiple computers, or within cloud storage environments. The following sections provide a detailed description of the service interface for managing fuzzy logic rules, including examples, configuration guidelines, and a formal OpenAPI specification.

Usage examples

A) Uploading a Fuzzy Logic Rule Set:

To upload a fuzzy logic rule set to the knowledge base, you can use the following `curl` command. The rule set should be encoded in base64 before uploading.

First, create a file containing your fuzzy logic rules, encode it in base64, and upload it:

```
# Encode the file in base64
```

```
base64 rules.fcl > rules_base64.fcl

# Upload the encoded rule set using curl
curl -X POST http://localhost:8080/api/v1/fuzzy-rules/PricingRules \
    -H "Authorization: Bearer YOUR_ACCESS_TOKEN" \
    -H "Content-Type: text/plain" \
    -d @rules_base64.fcl
```

Of course, you need to adapt “localhost” and the port “8080” to the network address of the concrete Knowledge Base service.

B) downloading a Fuzzy Logic Rule Set:

To download a fuzzy logic rule set from the knowledge base, you can use the following `curl` command. The downloaded rule set will be in base64 format, which you can then decode.

```
# Download the rule set using curl
curl -X GET http://localhost:8080/api/v1/fuzzy-rules/PricingRules \
    -H "Authorization: Bearer YOUR_ACCESS_TOKEN" \
    -H "Accept: text/plain" \
    -o downloaded_rules_base64.fcl

# Decode the downloaded base64 file
base64 --decode downloaded_rules_base64.fcl > downloaded_rules.fcl

# View the decoded rules
cat downloaded_rules.fcl
```

Configuration

The configuration of the knowledge base service is managed via a YAML file. This file includes parameters such as the relative path on the knowledge base server. Below is an example configuration file:

```
server:
  port: 8080
  basePath: /api/v1

knowledgeBase:
  fuzzyRulesPath: /data/fuzzy-rules
  security:
    oauth2:
      authorizationUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/auth
      tokenUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/token
      clientId: fuzzy-logic-knowledgebase
      clientSecret: your-client-secret
```

Port and base path are typical HTTP server parameters. The fuzzyRulesPath is a relative path to where the Fuzzy Logic rule sets are stored. The security section contains information about how to address to the authentication and authorization service like a Keycloak server.

Formal Specification

The following section provides the OpenAPI specification for the knowledge base service, detailing the routes and HTTP messages.

```yaml
openapi: 3.0.0
info:
  title: Fuzzy Logic Rule Management API
  version: 1.0.0
  description: API for managing fuzzy logic rules in the knowledge base.
servers:
  - url: http://localhost:8080/api/v1
paths:
  /fuzzy-rules/{ruleSetName}:
    post:
      summary: Store a new fuzzy logic rule set
      operationId: storeFuzzyRuleSet
      parameters:
        - name: ruleSetName
          in: path
          required: true
          schema:
            type: string
      requestBody:
        required: true
        content:
          text/plain:
            schema:
              type: string
              description: Base64 encoded fuzzy logic rule set string
      responses:
        '200':
          description: Rule set stored successfully
        '400':
          description: Bad request
        '403':
          description: Forbidden
        '404':
          description: Not found
        '500':
          description: Internal server error
      security:
        - oauth2: []
  /fuzzy-rules/{ruleSetName}:
    get:
      summary: Load a fuzzy logic rule set by name
      operationId: loadFuzzyRuleSet
      parameters:
        - name: ruleSetName
          in: path
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Rule set retrieved successfully
          content:
            text/plain:
              schema:
                type: string
                description: Base64 encoded fuzzy logic rule set string
        '400':
          description: Bad request
        '403':
          description: Forbidden
        '404':
          description: Not found
        '500':
          description: Internal server error
      security:
        - oauth2: []
components:
  securitySchemes:
    oauth2:
      type: oauth2
      flows:
        password:
```

```
        authorizationUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/auth
        tokenUrl: http://my_keycloak.com:8000/realms/InnoSale/protocol/openid-connect/token
        scopes: {}
```

In Innosale, we will always use the constant character string “PricingRules” as value for the parameter “ruleSetName”. This makes the service universally usable for other kinds of Fuzzy Logic rule sets.

5 Deductive Reasoning Rules

One solution to realize ontology-based data access (OBDA) on ontologies and related formats to store knowledge and facts is a rule-based, declarative, and logic-based approach. The Nemo language [1] follows that approach and enhances the reasoning capabilities of Datalog with native support for numbers, various aggregation functionalities, and many built in functions one would expect from imperative and relational query languages. This approach allows the Nemo system to refine available knowledge, identify structurally missing information, and produce explainable answers.

Expert knowledge can transfer in a straightforward way into maintainable and easy to change declarative rules. This allows for repeated use and further insights to the modelled knowledge. Expert systems and business rule systems build upon such methods but are usually less efficient or versatile for the end user of them. The advantage of the Nemo language is a simple, yet powerful, language with a well maintained developer environment, consisting of a language server for code highlighting and syntactical error checking as well as a robust method to explain solutions.

Nemo language

The Nemo language builds upon the declarative rule language Datalog and extends it with many features of modern query and rule languages. In Datalog, expert knowledge can be represented through simple and easy-to-understand if-then statements. Such rules are evaluated by matching the if-part with existing data and deriving the corresponding conclusion of the then-part. Unlike traditional database query languages, Datalog allows rules to depend on each other recursively, enabling more complex queries and a more sophisticated manipulation of data. It therefore achieves a perfect balance between simplicity and expressiveness. In the following, we show example usage of the Nemo language to represent knowledge for the Light-Lifting Domain.

```
is_a("KBK Aluline", "KBK Light Crane") .
is_a("KBK Light Crane", "Crane") .
max_load("KBK Light Crane", 3200) .

max_ load (?Crane, ?Weight)
    :- is_a(?Crane, ?Category), max_load(?Category, ?Weight) .
is_a(?A, ?C) :- is_a(?A, ?B), is_a(?B, ?C) .
```

In the above program, we see part of a concept hierarchy describing the “subclass of” or “instance of” relationship between different types of cranes. Such hierarchies form the basis of many ontologies. Furthermore, the program includes the maximum load capacity for a specific type of crane. Rules are read from right to left, where the right side encodes its condition while the left side contains its conclusion. Variables are denoted with a question mark. Intuitively, the first rule states that if a category of cranes has a given maximum load capacity, then every crane belonging to that category must also have the same load capacity. The second rule demonstrates the recursive computation of the is_a relationship. The above ruleset therefore derives that KBK Aluline is a crane with a maximum load capacity of 3200 kg.

Nemo has support for many different datatypes – including integer, string, language-tagged string, single and double precision float, and boolean – as well as related functions. The latter include many functions known from other programming languages, such as SQRT (square root of a number) or STRLEN (length of a string), as well as standard arithmetic operators. Functions can be nested arbitrarily and may occur anywhere in a rule. Moreover, comparison operators like < or != can be used provided that variable bindings are sufficiently determined by other parts of the rule's condition. Nemo is dynamically typed and allows any type of data to occur in any position, without requiring a fixed schema. Functions such as STRLEN are not defined for all types of inputs, and rules carry the implicit condition that they only apply to variable bindings for which all functions are defined. The following example shows a simple use case where a request is marked as invalid, if the requested load is greater than the maximum load that can be handled by the requested type of crane:

```
invalid(?Request) :-
    request_load(?Request, ?Load), request_crane(?Request, ?Crane),
    max_load(?Crane, ?Max), ?Load > ?Max .
```

The Nemo language also features existential rules, which allow the user to assert the existence of certain patterns. Such rules are useful to determine the validity of a given ontology or may be used to verify the completeness of a customer request. The following rule, for example, requires that every crane of type KKB Light Crane includes a hoist. Existential variables are marked with an exclamation mark.

```
has_part(?Crane, !Hoist), type(!Hoist, "hoist")
    :- is_a(?Crane, "KBK Light Crane") .
```

An important feature of Nemo's rule language are aggregates. Supported are the most common aggregates: maximum, minimum, sum and count. The next example determines the maximum load capacity of a crane by computing the minimum load capacity of each of its parts.

```
max_load(?Crane, #min(?Load))
    :- max_load(?Part, ?Load), part_of(?Crane, ?Part) .
```

Finally, Nemo supports (stratified) negation using the following syntax:

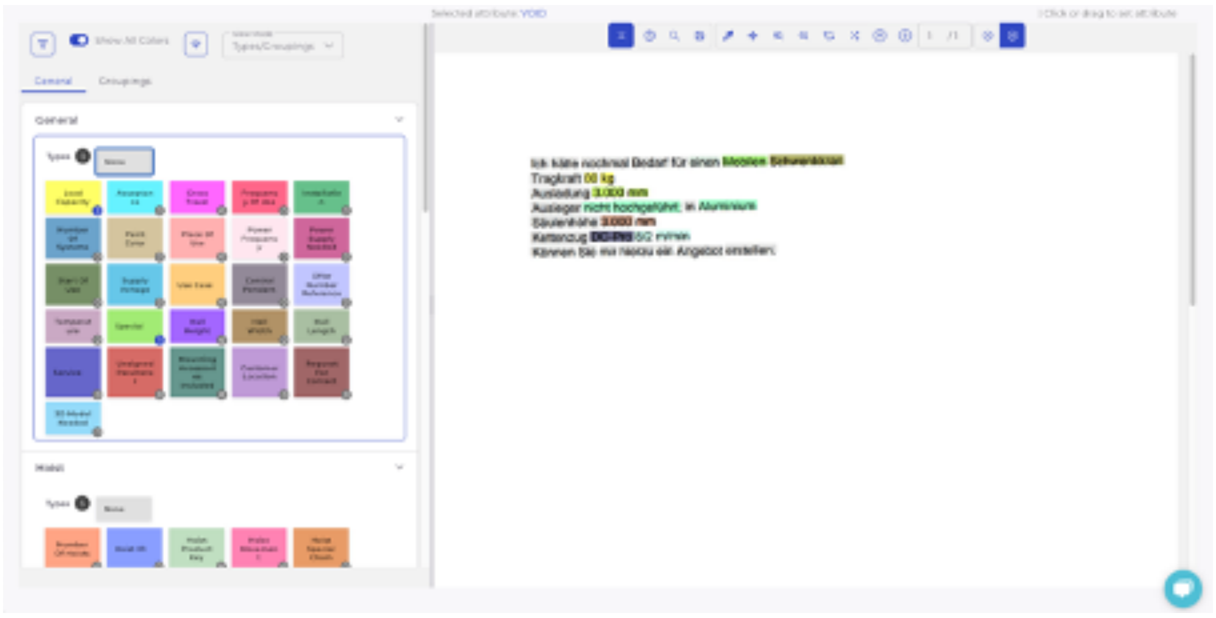
```
valid(?Request):- ~invalid(?Request) .
```

Stratified negation intuitively means that there are no negation cycles allowed.

6 Entity Recognition

Within InnoSale, Named Entity Recognition (NER) is performed to extract relevant technical specifications from customer e-mail inquiries. The NER learns different formulations used from novice customers buying a complex product for the first time, to expert customers that can precisely specify what they need. Within the NER component, all this knowledge is modeled 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 that knowledge, data annotation is needed. For this, the annotation tool shown in Figure 5 was used, which has 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.



Ich hätte nochmal Bedarf für einen Mobilen Schwenkkran
Tragkraft 00 kg
Ausladung 3.000 mm
Ausleger nicht hochgeführt, in Aluminium
Säulenhöhe 3.000 mm
Kettenzug [illegible] 8/2 m/min
Können Sie mir hierzu ein Angebot erstellen.

Figure 7 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. Taking the NER output and transforming it into a structured format (here JSON) thus allows the automatic deduction of technical requirements from free-form textual e-mail inquiries. This allows automatically filling in e-project sheets, thus, significantly speeding up the time to offer. Furthermore, deductive reasoning rules as described in Section 5, can be used on top of the NER output to deduce further knowledge not explicitly stated in the inquiry. Thus, the implicit modeling of the NER module can be extended by the explicit modeling via reasoning rules, leading to an overall system that understands what the user explicitly stated and also what he did not state explicitly but which can be deduced from it.

7 Speech Recognition & Meeting Summarization Parameters

The integration of advanced Natural Language Processing (NLP) technologies is a pivotal component of the InnoSale project, aimed at transforming spoken content from project meetings into actionable, precise textual transcripts. Knowledge Base in this case, employs different models for transcription and language processing as well as semantic analysis and summarization. It also stores all audio files and transcribed summaries, maintaining a robust and searchable database that supports quick data retrieval and historical analysis.

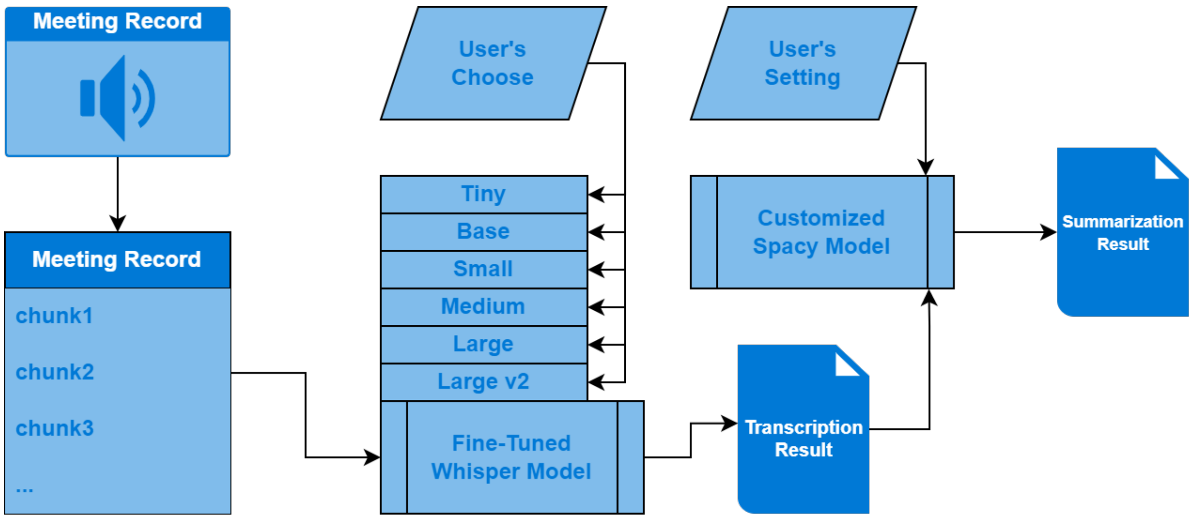


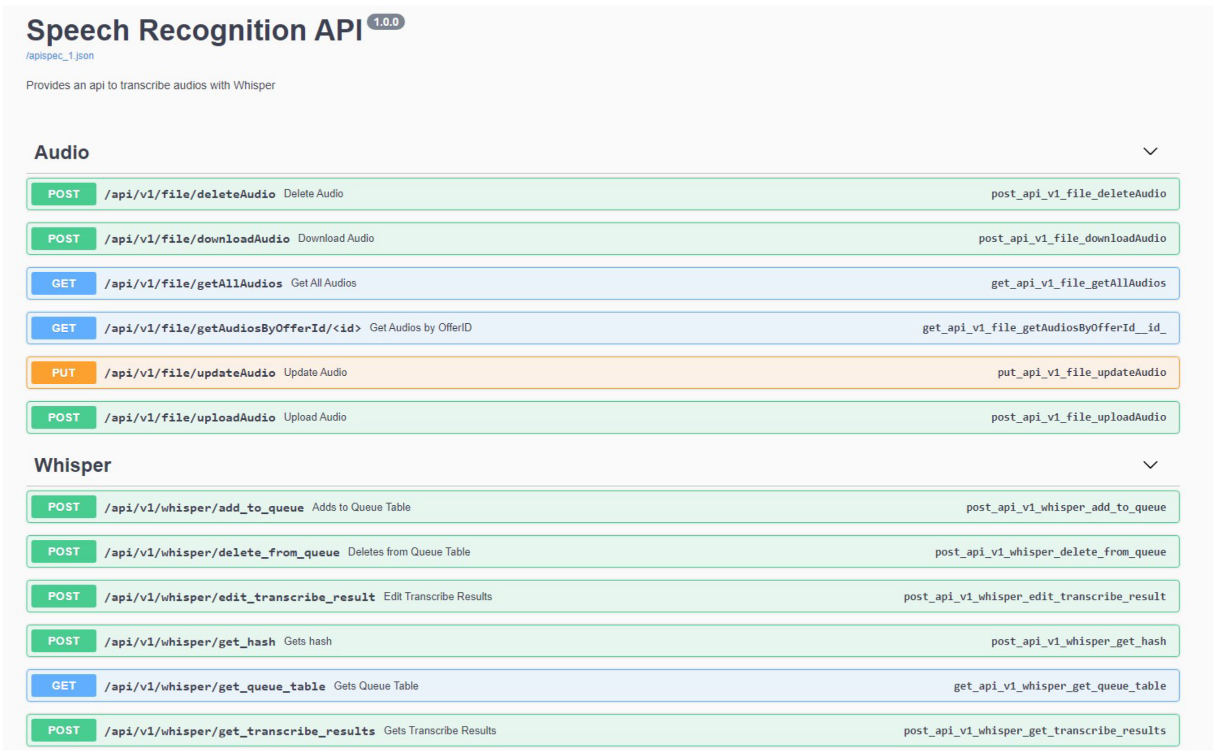
Figure 8 Flow Chart Diagram

We are utilizing OpenAI's Whisper Large-v2 model, specifically fine-tuned to recognize and transcribe the Turkish language to address the unique challenges presented by the language. This model captures spoken content with high accuracy, which is essential given Turkish's agglutinative nature, where suffixes significantly alter meanings and grammatical functions. The fine-tuning process includes training on a diverse dataset received from Ermetal that includes industry-specific terminology and various dialects to ensure comprehensive understanding and accuracy.

Post-transcription, the Turkish Spacy model applies advanced NLP techniques to analyse the structure and semantics of sentences. This model prioritizes crucial information, employing relevance algorithms such as Jaccard and Cosine similarities to identify and highlight the most important information. The system extracts key decisions, technical specifications, and actionable insights from lengthy discussions, condensing them into concise, easy-to-digest summarized texts. This not only saves time but also ensures that critical data influencing quote accuracy is highlighted.

API Specification

Audio recordings meetings are uploaded to the system through the interface developed within the scope of the InnoSale project. Voice recordings uploaded to the system are stored on the server. Additionally, a table containing information about the uploaded voice recordings and their file locations are stored in the MySQL database. The voice recording to be transcribed is selected through the interface developed within the scope of the InnoSale project and the transcribing process is started. When the transcribing process of the voice recording is completed, the results are stored in the MySQL database and can be viewed via the interface.



Speech Recognition API 1.0.0
/apispec_1.json

Provides an api to transcribe audios with Whisper

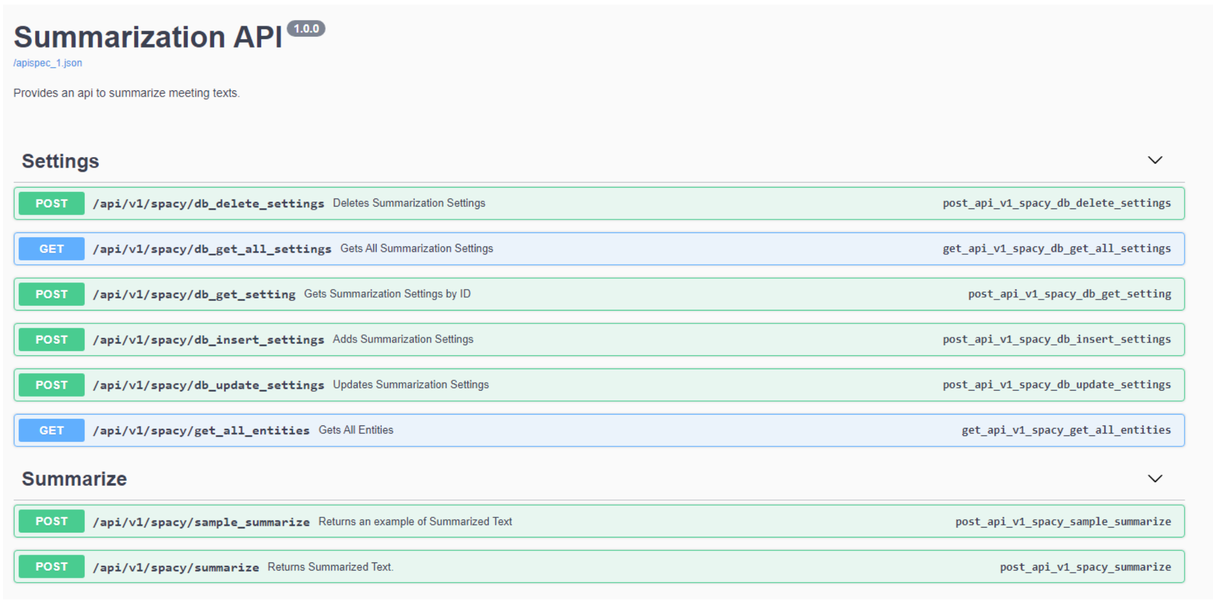
Audio

POST /api/v1/file/deleteAudio Delete Audio post_api_v1_file_deleteAudio
POST /api/v1/file/downloadAudio Download Audio post_api_v1_file_downloadAudio
GET /api/v1/file/getAllAudios Get All Audios get_api_v1_file_getAllAudios
GET /api/v1/file/getAudiosByOfferId/<id> Get Audios by OfferID get_api_v1_file_getAudiosByOfferId__id_
PUT /api/v1/file/updateAudio Update Audio put_api_v1_file_updateAudio
POST /api/v1/file/uploadAudio Upload Audio post_api_v1_file_uploadAudio

Whisper

POST /api/v1/whisper/add_to_queue Adds to Queue Table post_api_v1_whisper_add_to_queue
POST /api/v1/whisper/delete_from_queue Deletes from Queue Table post_api_v1_whisper_delete_from_queue
POST /api/v1/whisper/edit_transcribe_result Edit Transcribe Results post_api_v1_whisper_edit_transcribe_result
POST /api/v1/whisper/get_hash Gets hash post_api_v1_whisper_get_hash
GET /api/v1/whisper/get_queue_table Gets Queue Table get_api_v1_whisper_get_queue_table
POST /api/v1/whisper/get_transcribe_results Gets Transcribe Results post_api_v1_whisper_get_transcribe_results

Summarization parameters are created according to the content of the meeting, ensuring a more accurate summarization of the language model. These parameters are set and saved by the user via the InnoSale interface. The saved parameters are stored in the MongoDB database for later use. Then, the summarization process is performed by selecting a voice recording that has been transcribed and a parameter set stored in the database. The summarized result obtained as a result of this process is displayed on the interface.



Summarization API 1.0.0
/apispec_1.json

Provides an api to summarize meeting texts.

Settings

POST /api/v1/spacy/db_delete_settings Deletes Summarization Settings post_api_v1_spacy_db_delete_settings
GET /api/v1/spacy/db_get_all_settings Gets All Summarization Settings get_api_v1_spacy_db_get_all_settings
POST /api/v1/spacy/db_get_setting Gets Summarization Settings by ID post_api_v1_spacy_db_get_setting
POST /api/v1/spacy/db_insert_settings Adds Summarization Settings post_api_v1_spacy_db_insert_settings
POST /api/v1/spacy/db_update_settings Updates Summarization Settings post_api_v1_spacy_db_update_settings
GET /api/v1/spacy/get_all_entities Gets All Entities get_api_v1_spacy_get_all_entities

Summarize

POST /api/v1/spacy/sample_summarize Returns an example of Summarized Text post_api_v1_spacy_sample_summarize
POST /api/v1/spacy/summarize Returns Summarized Text. post_api_v1_spacy_summarize

8 Similarity Analysis

We are utilizing a combination of 3D shape analysis, advanced algorithms, and integrated data systems to provide a thorough assessment of manufacturing complexities and directly influences accurate cost estimations.

We propose a novel algorithm [2] in KB that combines geometric feature extraction, normalization techniques, and advanced point cloud registration methods to enhance the accuracy and robustness of 3D shape similarity detection.

Our analysis first starts with geometric feature extraction. The initial step involves calculating the dimensions of the oriented bounding box (OBB), which helps standardize the comparison of 3D models by eliminating variances caused by different orientations. The algorithm computes the OBB for each mesh, which encapsulates the shape within the smallest box aligned with its principal axes, providing an orientation-invariant representation of the shape's dimensions. This ensures that all models are assessed on a common ground, facilitating more consistent cost predictions.

Persistent homology is employed to compute Betti numbers, which are topological attributes that count the number of loops and holes in a structure. Betti numbers, particularly b1, which represents the number of loops or holes in the mesh, offers insights into a shape's topological complexity. Understanding these aspects is crucial as they significantly influence the manufacturing process and associated costs, particularly for complex designs that require sophisticated techniques.

To mitigate the influence of outliers in the geometric feature distributions, a robust scaler is employed for normalization. This scaler focuses on the median and interquartile range, ensuring that the scaling is not unduly affected by extreme values, which are common in geometric data.

After normalization, the Manhattan distances between the feature vectors of different shapes are computed to establish an initial similarity ranking, with smaller distances indicating closer matches.

A pre-processing stage is employed and the top 50 potentially similar shapes from the initial ranking are prepared for further analysis. This involves scaling the point clouds to a standardized size and aligning them with respect to their principal axes.

For precise model alignment and comparison, KB uses an advanced ICP algorithm enhanced with Random Sample Consensus (RANSAC) and Principal Component Analysis (PCA). This method is critical for handling the top 50 similar parts to achieve precise alignment and similarity assessment. For parts with a high aspect ratio (resembling long poles), the algorithm relies solely on the initial Manhattan distance metric for similarity assessment, bypassing the ICP alignment due to potential inaccuracies with elongated shapes.

Lastly, before running the ICP algorithm, PCA is used to optimally align and scale the shapes, ensuring an accurate initial alignment for the subsequent ICP iterations.

This approach provides our use case partner a robust framework for accurate and efficient 3D shape similarity detection across various applications. The sample results which compare Ermetal's domain experts and algorithm's results can be found below:

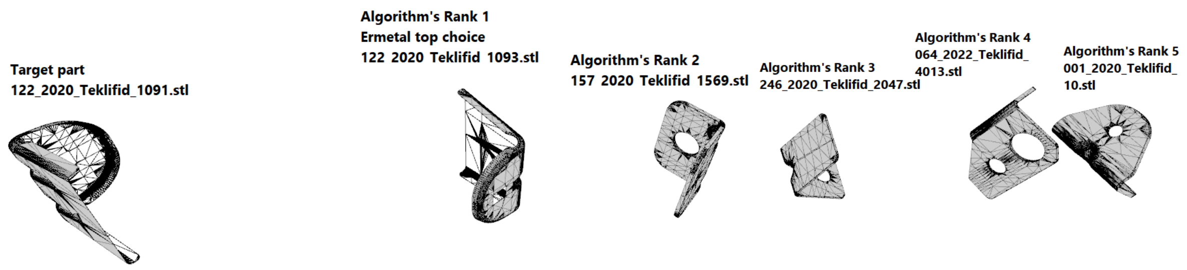


Figure 9 Similarity Results #1

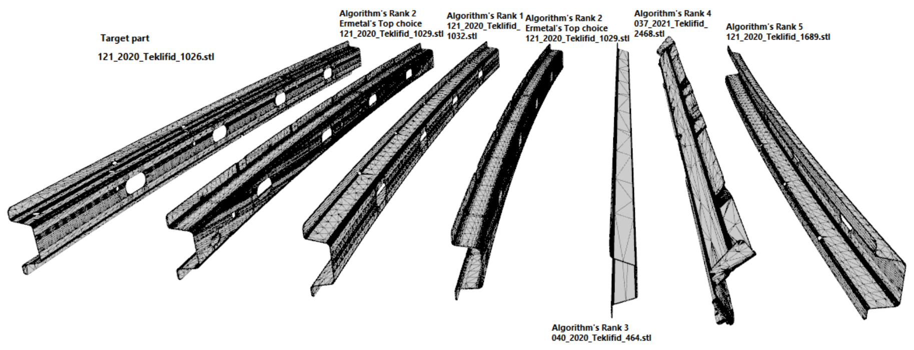


Figure 10 Similarity Results #2

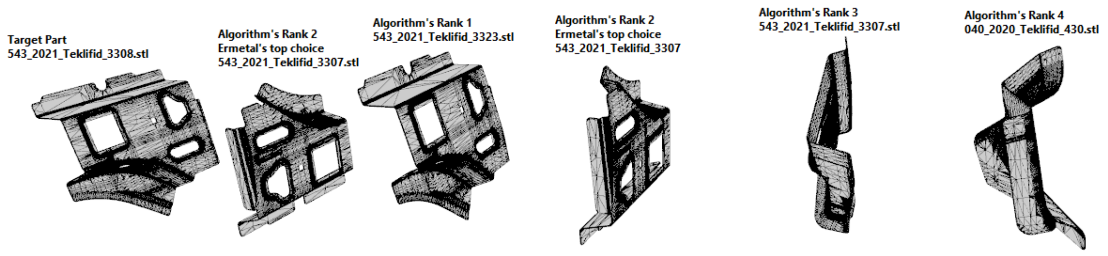


Figure 11 Similarity Results #3

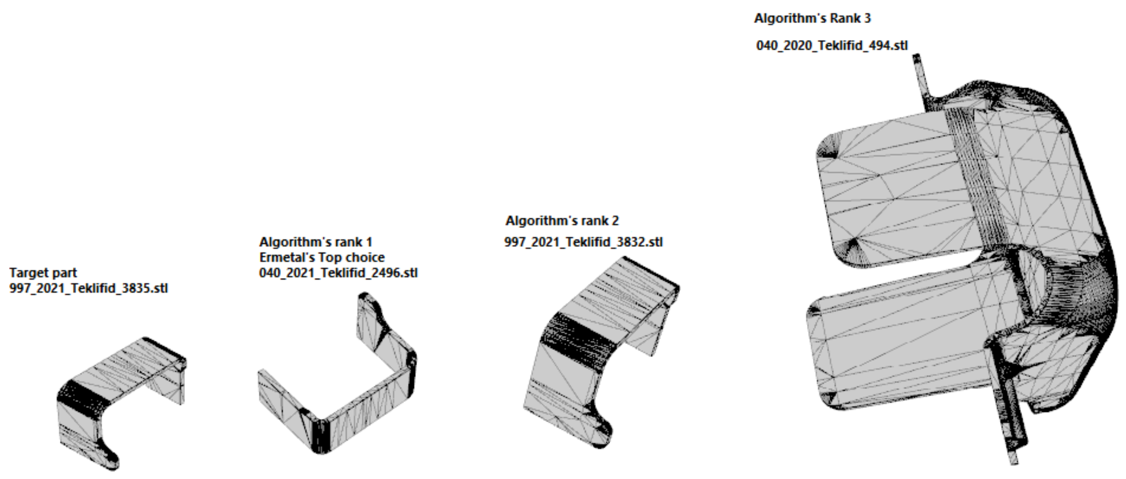
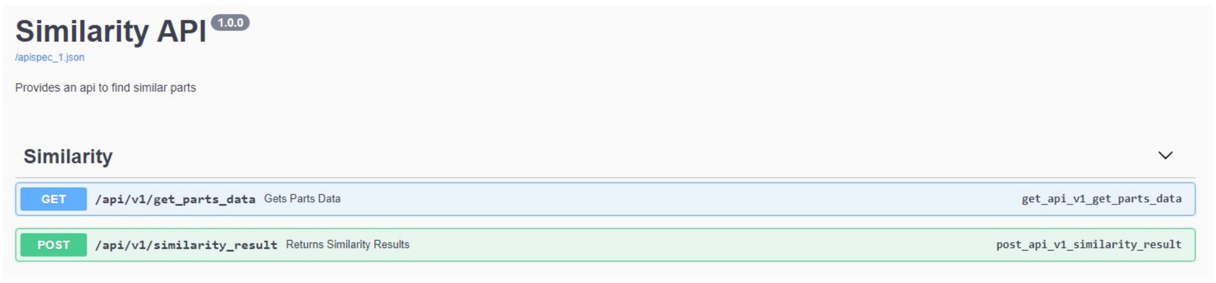

Target part
997_2021_Teklifid_3835.stl
Algorithm's rank 1
Ermetal's Top choice
040_2021_Teklifid_2496.stl
Algorithm's rank 2
997_2021_Teklifid_3832.stl
Algorithm's Rank 3
040_2020_Teklifid_494.stl

Figure 12 Similarity Results #4

API Specification

The file location and attributes of the STL parts added before in the system are stored in the MySQL database. The similarity algorithm works in two different ways. Similarity calculation is made according to the geometric shape of the parts (comparative score calculation with the ICP algorithm over STL files) and the cosine distance between the feature vectors. At the same time, the interface also includes the feature of filtering parts with a certain range of attributes before running the similarity algorithm.



Similarity API 1.0.0

/apispec_1.json

Provides an api to find similar parts

Similarity

GET `/api/v1/get_parts_data` Gets Parts Data `get_api_v1_get_parts_data`

POST `/api/v1/similarity_result` Returns Similarity Results `post_api_v1_similarity_result`

9 Abbreviations

LLE	Light Lifting Equipment
CTO	Industrial Cranes
WM	Waste Management
DPM	Digital Product Marketplace
SM	Sheet Metal Stamping
KB	Knowledge Base
API	Application Programming Interface
SQL	Structured Query Language
JSON	JavaScript Object Notation
MySQL	My Structured Query Language
OAuth	Open Authorization
YAML	Yet another markup language
HTTP	Hypertext Transfer Protocol
REST	Representational state transfer
KAC	Knowledge Acquisition Component
UC	Use Case

10 References

[1] Abid, T. R., Yıldız, C., Erten, A. E., & Kaya, K. (n.d.). *Enhancing Mesh and Point Cloud Similarity Detection through Geometric Features and ICP.* Dakik Software Technologies & ERMETAL.

[2] Alex Ivliev, Stefan Ellmauthaler, Lukas Gerlach, Maximilian Marx, Matthias Meißner, Simon Meusel, Markus Krötzsch: **Nemo: First Glimpse of a New Rule Engine** In Enrico Pontelli, Stefania Costantini, Carmine Dodaro, Sarah Gaggl, Roberta Calegari, Artur D'Avila Garcez, Francesco Fabiano, Alessandra Mileo, Alessandra Russo, Francesca Toni, eds., *Proceedings 39th International Conference on Logic Programming (ICLP 2023)*, volume 385 of EPTCS, 333--335, September 2023