Data specification navigator (technical documentation)

1 Abstract

This project addresses the common problem faced by users who are interested in their organization's data but do not fully grasp the whole domain ontology and the underlying technical structure. The project's goal is to act as a conversational intermediary, creating a standalone application that allows users to ask questions in plain language. The final implementation successfully guides the user through the data specification and provides users with executable SPARQL queries, but it still has considerable limitations. The full range of SPARQL syntax is not supported in the current implementation. While the solution is a monolithic application, a key aspect is its modular architecture, ensuring future adaptability and independence from any one specific large language models.

2 Introduction

This document presents the realization of the project named "Helping users navigate data specifications".

2.1 Motivation

The project's motivation is best understood through a common scenario: Tanya needs some specific data from her organization's database. She must visit her organization's "database person" and formulate her question: "I would like to see our employees who started working here this year". This "database person" will then query the database and give Tanya a list of employees. The "database person" could also ask: "Do you want all the employees or only employees from a specific department?". Tanya will answer the question and the two of them can continue in an iterative manner to refine Tanya's query. My project directly addresses this bottleneck by creating a standalone application that acts as a digital intermediary.

The developed application allows the same kind of iterative refinement of the user's query as in the given example, eliminating the need for a human expert (the "database person" in our previous example).

2.2 Dataspecer

Dataspecer is a tool designed for managing and modeling data specifications. It helps users create structured data schemas from ontologies and export them into various formats such as JSON, XML, and CSV. The Dataspecer manager lets users create and manage Dataspecer packages – these packages will be used as data specifications in this project.

3 System design and architecture

The application has a client-server architecture, which comprises of a thin-client frontend and a C# backend.

3.1 Frontend

The user interface (UI) was designed as a thin client to ensure it is lightweight and responsive. It is a single-page application that communicates with the backend via RESTful API calls. The application's user interface is divided into two primary views for user interaction.

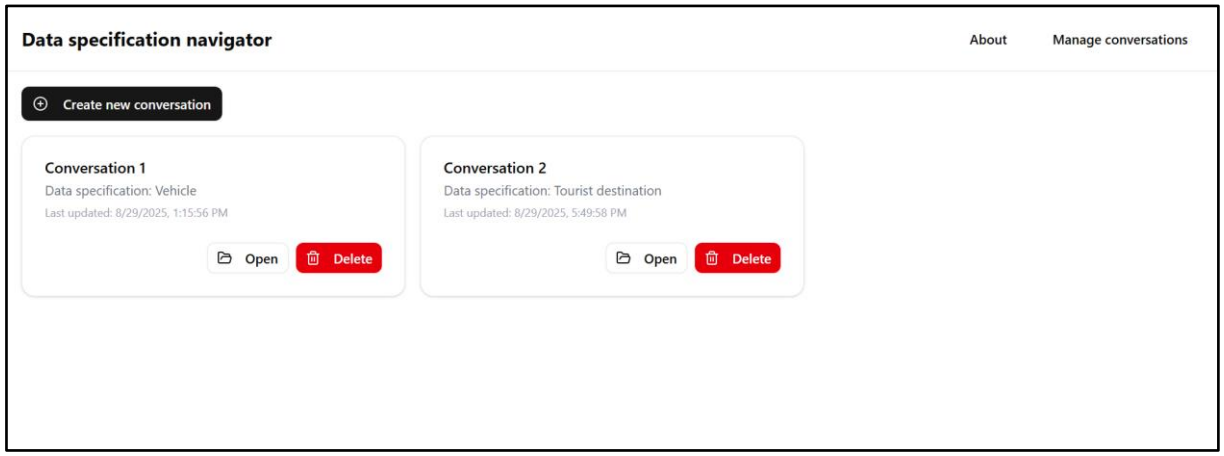
3.1.1 Conversation management view

The conversation management view serves as the main entry point for the application, providing a central hub for users to manage their conversations (see Figure 1).

The UI provides clear visual cues to the user when interacting with the backend:

- Loading conversations: when fetching conversations data, skeleton cards are displayed to indicate that the app is loading conversations.
- A spinning circle while waiting for the backend call that creates a new conversation.
- Whenever a backend call fails or in case of other errors, the UI displays an error message colored red.

Figure 1: Conversation management



3.1.1.1 Core functionalities

Display all conversations: The view retrieves a list of all existing conversations from the backend and displays them in a grid of interactive cards. Each card shows the conversation's **title**, the name of the **data specification** it's based on, and the **last updated** timestamp. If there are not yet any conversations, the view provides the user with a short instruction on how to create a new conversation.

Create a new conversation: Users can create a new conversation in one of two ways:

1. **Manual creation:** By clicking the "Create new conversation" button, a dialog appears that prompts the user to manually enter the Dataspecer package IRI and a title for the new conversation.
2. **Browser redirect:** Conversation creation can also be initiated by a browser redirect from the Dataspecer tool. It detects specific URL parameters (uuid and packageName) to automatically open a new conversation dialog with the Dataspecer package information pre-filled. This feature was implemented to streamline the user experience when navigating directly from the Dataspecer tool.

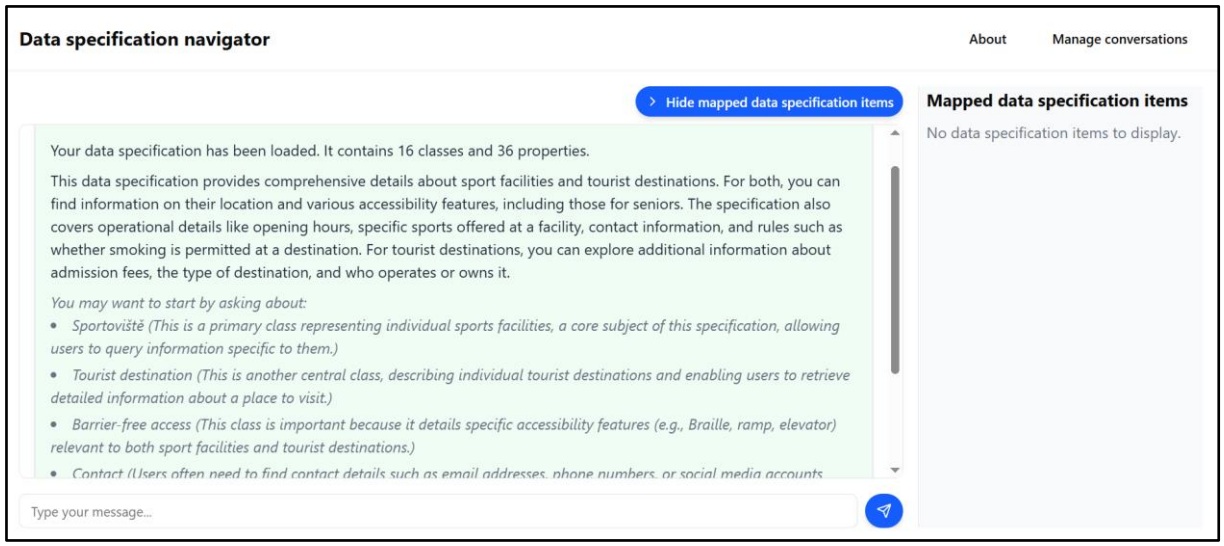
Open a conversation: Each conversation card features an "Open" button that navigates the user to the dedicated Conversation View.

Delete a donversation: Users can permanently delete a conversation by clicking the "Delete" button on its card, which will prompt a confirmation dialog to prevent accidental deletion.

3.1.2 Conversation view

This view is the primary interface for users to interact with the chatbot, emulating the familiar conversational flow of mainstream LLM services (see Figure 2).

Figure 2: Conversation welcome message



3.1.2.1 Core functionalities

Conversation history: The view fetches and renders all messages of the given conversation in chronological order.

User input and submission: The user can type their natural language query into an input field at the bottom of the screen. Upon submission, the UI immediately updates with the user's message and it to the backend via a RESTful API call.

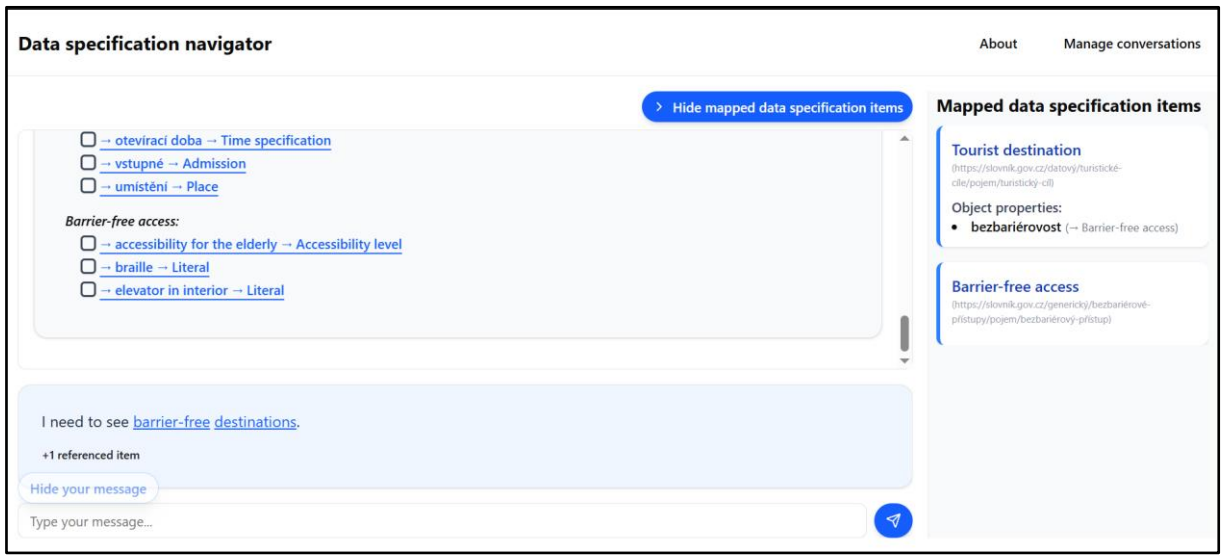
Data specification navigation: The UI provides two key features to help users navigate the data specification (see Figure 3).

1. **Mapped item highlighting:** The backend identifies and highlights words in the user's message that refer to data specification items. The user's message with highlighted words is displayed above the input text field. The UI renders these words as clickable links, and a pop-up dialog provides a summary of the corresponding item.
2. **Query refinement via suggestions:** Each chatbot reply may contain suggested items from the data specification. Clicking on a suggestion opens a dialog with a detailed summary of the item, allowing the user to understand its purpose and relevance. Users can select one or more suggested items from the chatbot's most recent reply to expand their query.

Display a suggested message: Once the user has confirmed selected suggestions, a suggested message is generated by the backend, and the view will display it above the input field. This

verbalization combines the user's message with the selected suggestions, allowing them to preview the expanded query before submission.

Figure 3: Words highlighting and suggestions

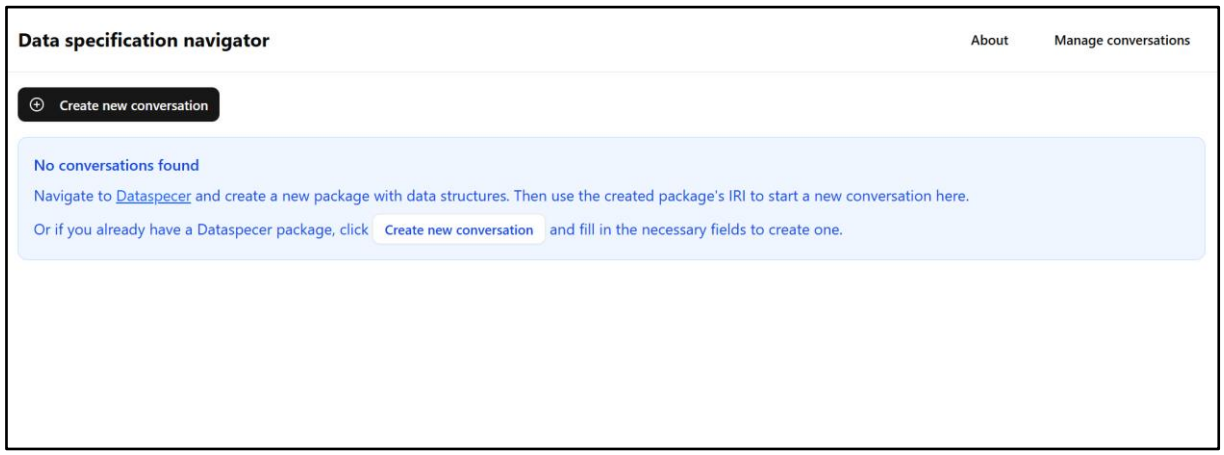


3.1.3 Example workflow

This section provides an example that will present all the important elements of the UI.

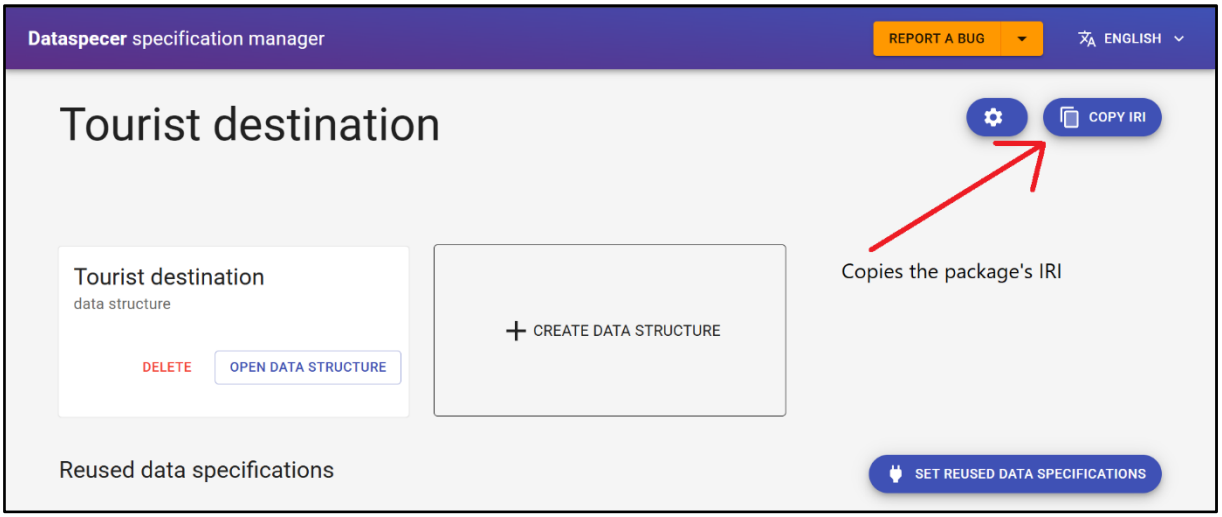
The user navigates to the main page of the Data specification navigator, which is by default the conversation management view (see Figure 4).

Figure 4: Conversation management without any conversations created



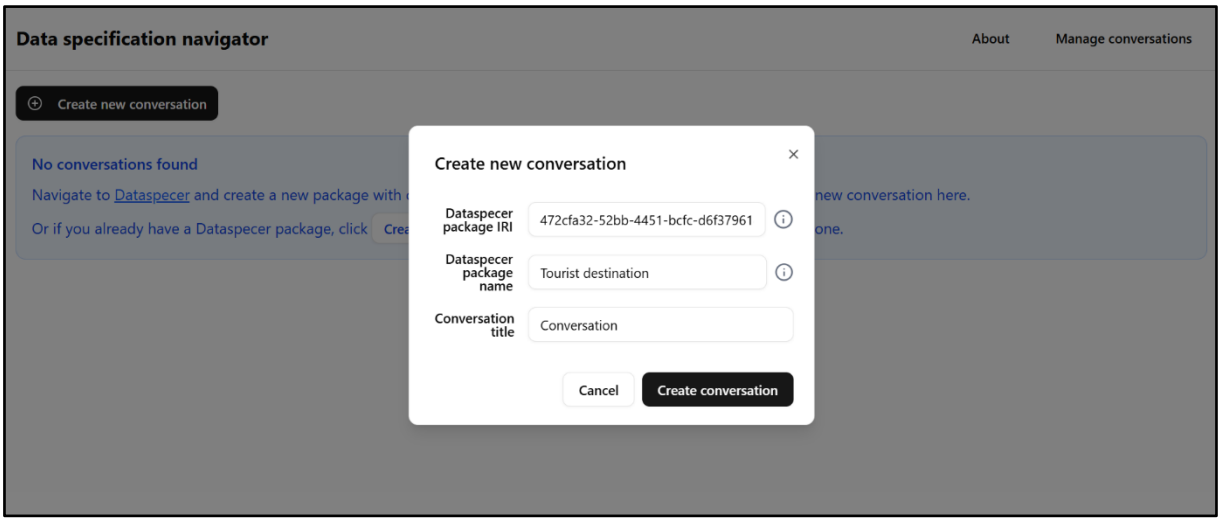
The user follows the instructions and creates a new Dataspecer package. The user then copies the package IRI (see Figure 5).

Figure 5: Dataspecer package UI



The user returns to the Data specification navigator, clicks on the "Create new conversation" and a form appears. The user fills in the necessary information (see Figure 6).

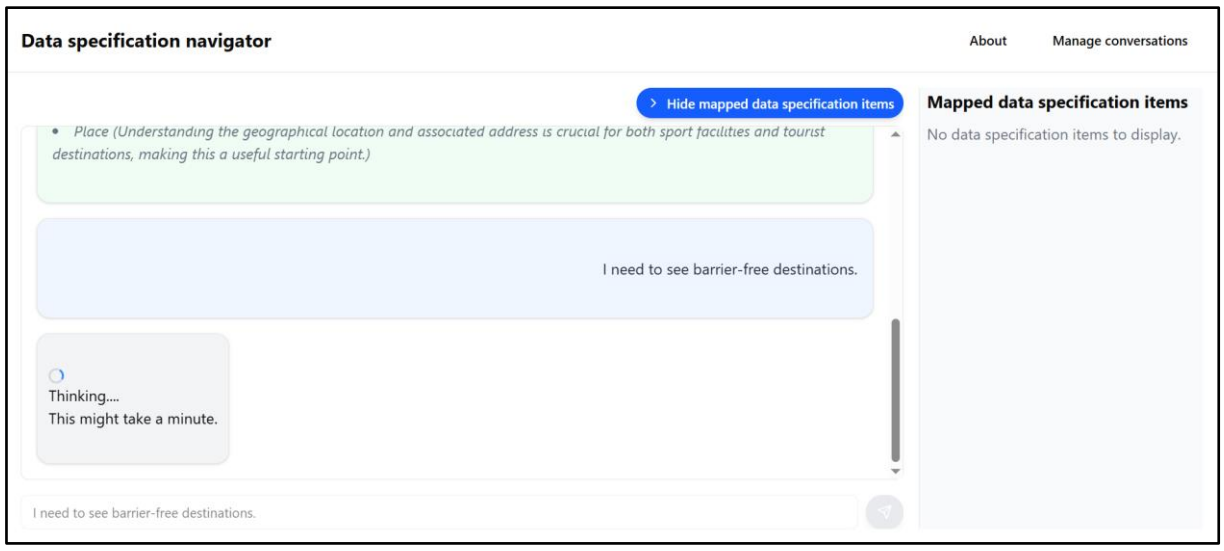
Figure 6: Conversation creation form



The user opens the newly created conversation and sees the welcome message (see Figure 2).

The user types their first message and sends it (see Figure 7).

Figure 7: Waiting for a chatbot reply



The chatbot sends an answer back. The right-side panel is updated with mapped items from the data specification. The reply from the chatbot is long so it is shown in two parts (see Figure 8 and Figure 9).

Figure 8: Chatbot reply - SPARQL query

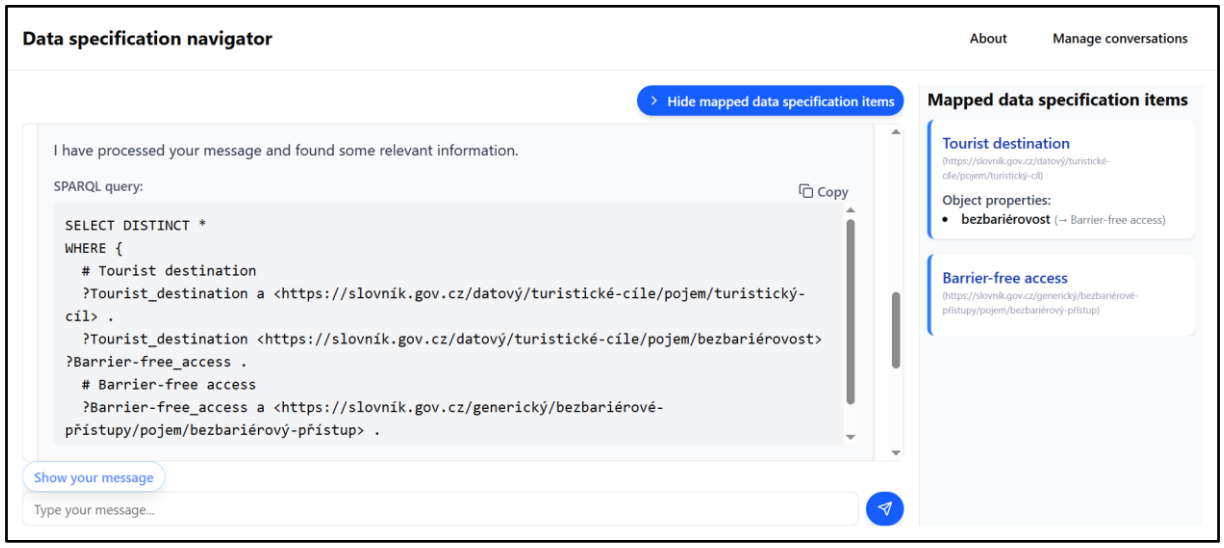
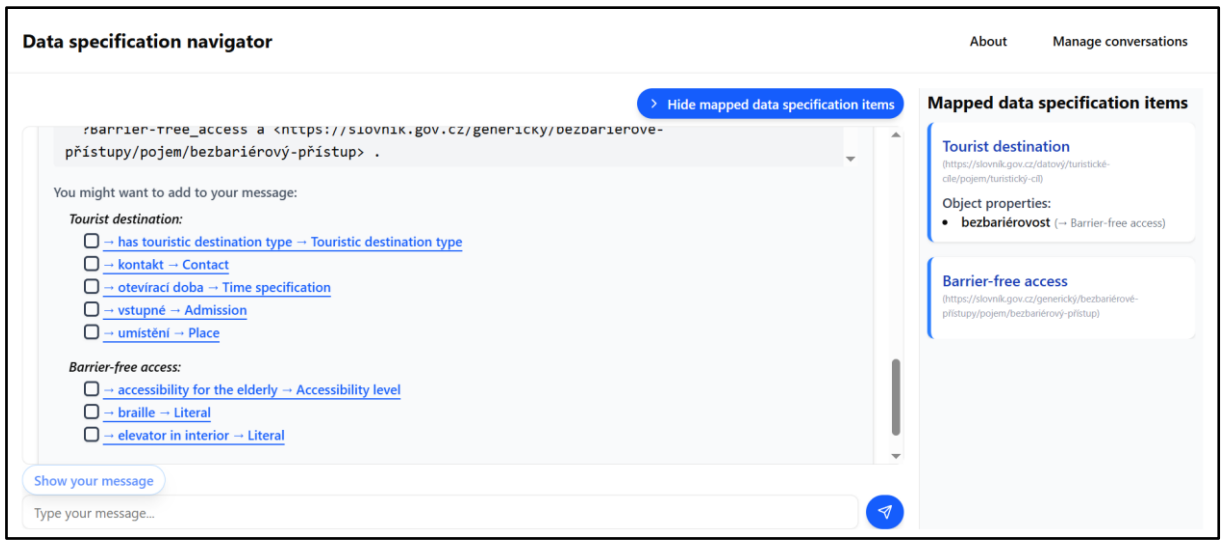


Figure 9: Chatbot reply - suggestions



The toggle “Show your message” shows the user message with the highlighted parts corresponding to the mapped items (see Figure 3).

The user selects some suggested items. The UI shows the button “Add all selected items to my message” on top of the chat (see Figure 10).

The user clicks the button to add all items which prompts the UI to retrieve the suggested message from the backend (see Figure 11).

The UI displays the received suggested message (see Figure 12).

Figure 10: User chooses suggestions

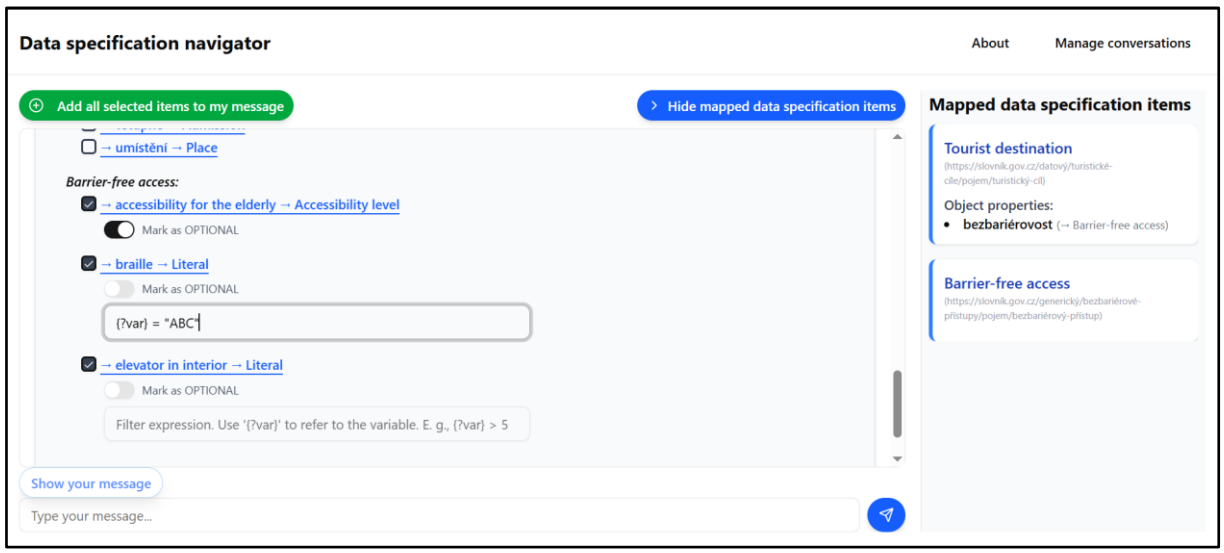


Figure 11: Waiting for a suggested message

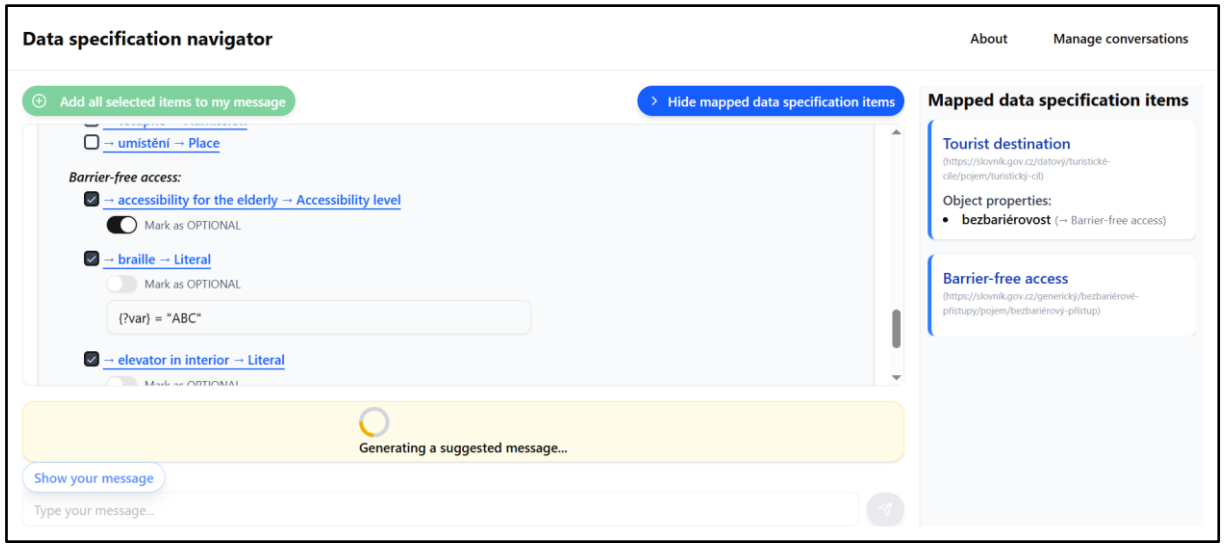
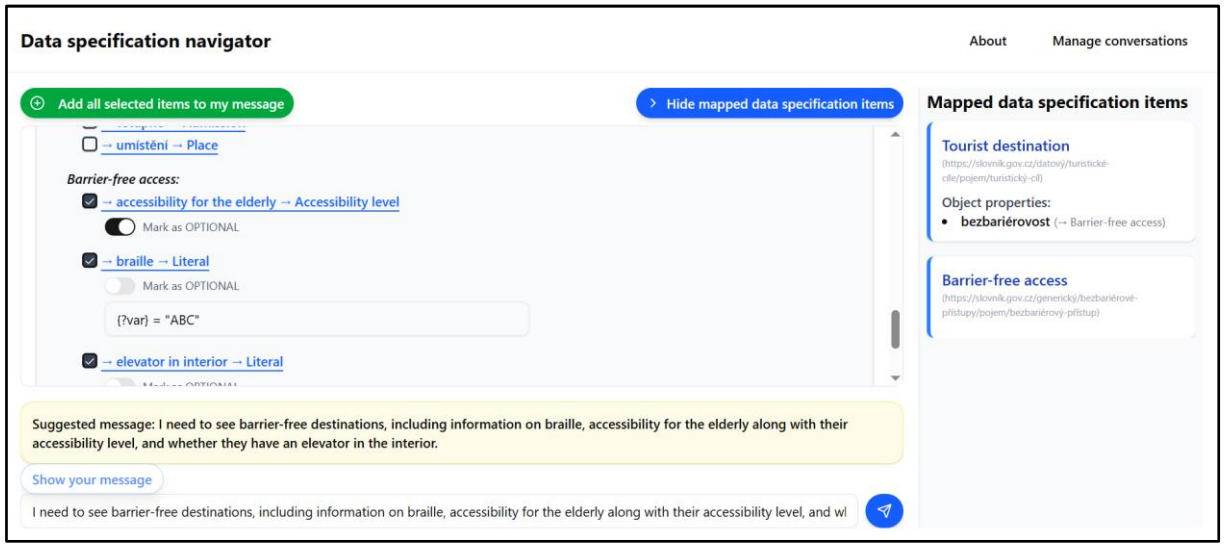


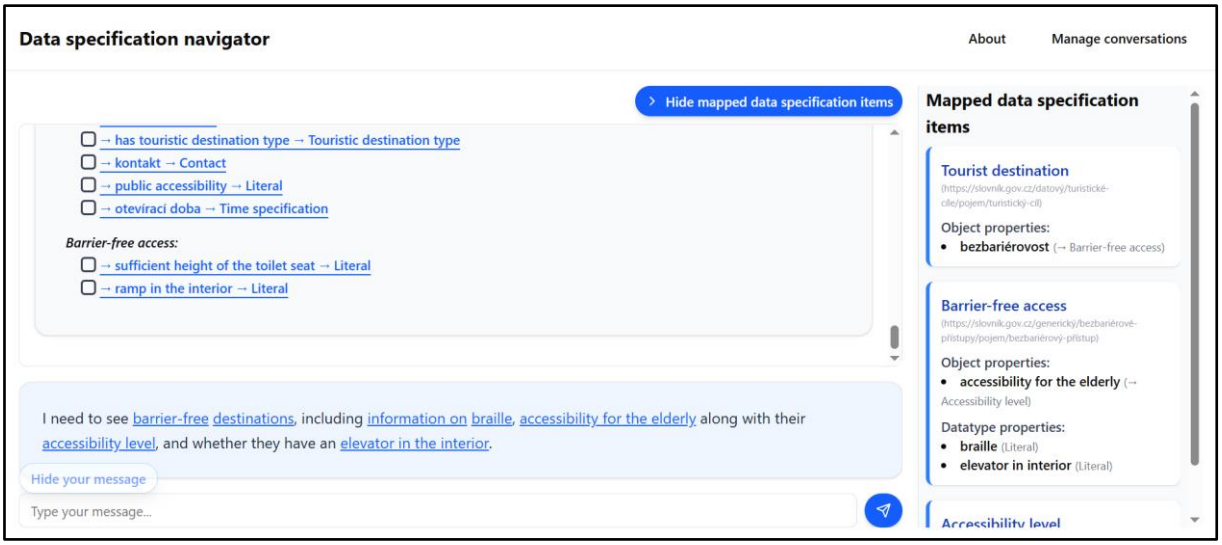
Figure 12: Suggested message displayed



The user sends the suggested message as is. Alternatively, the user can modify the message before sending it.

The chatbot replies with the new answer (see Figure 13). Notice that the right-side panel is again updated with newly mapped items.

Figure 13: User sends suggested message and receives a reply



The conversation flow can continue in this manner until the user is satisfied.

At any point during the conversation, the user may click on a highlighted word in their current message to see a summary for that item. Or they may click on a suggested item to see a summary and a reason for the suggestion (see Figure 14 and Figure 15).

Figure 14: Summary of a mapped item

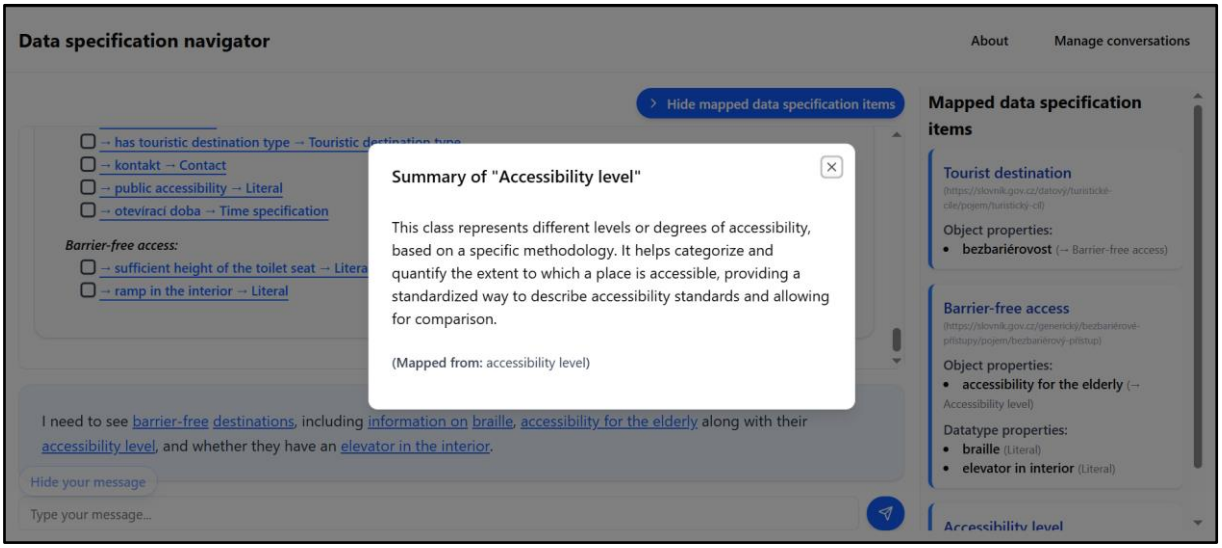
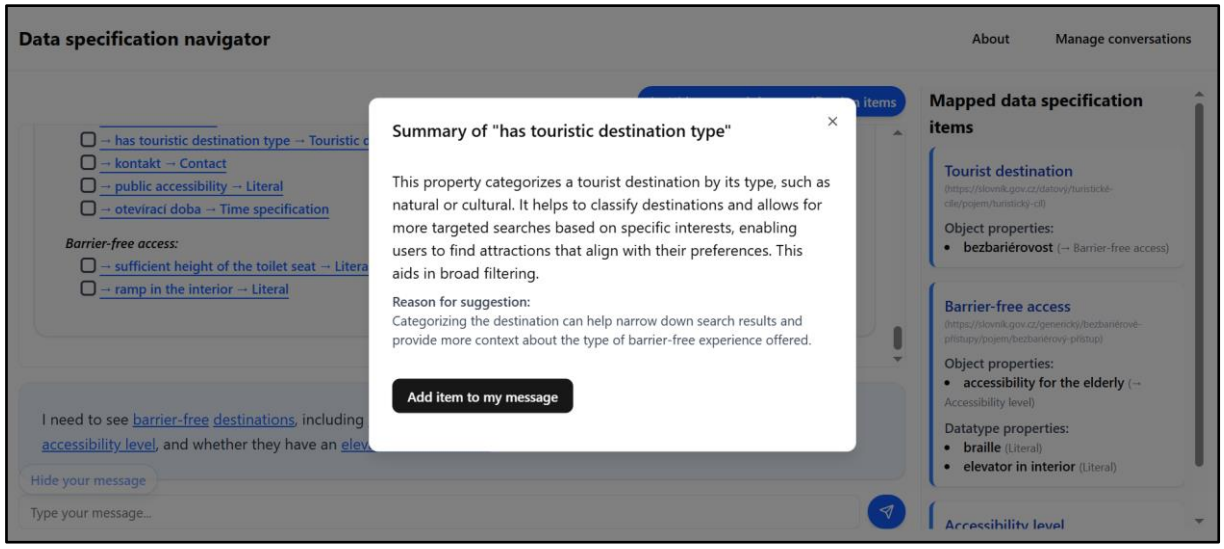


Figure 15: Summary of a suggested item



3.2 Frontend implementation details

The frontend was developed using Vite, React and Typescript. This combination was chosen to provide a modern, efficient, and type-safe development environment. React helps provide a component-based user interface, which allows for a modular and reusable codebase. The use of **TypeScript** provided static typing, which reduces runtime errors and improves code maintainability, especially when handling complex data structures returned from the backend API. **Vite** was utilized as the build tool for its *Hot Module Replacement* during development and its optimized production builds.

3.2.1 Conversation management view

This view is implemented as a React component and it interacts with the backend using RESTful API calls.

Conversation deletion

Upon confirming the delete action, the application performs an optimistic update, immediately removing the card from the UI and then sending a DELETE request to the backend. If the DELETE request fails, the UI will display an error and fetch all conversations again to display them to the user.

Conversation creation by redirect

The functionality for redirecting from Dataspecer directly to the conversation creation dialog is implemented in a forked Dataspecer repository https://github.com/Kwantigon/dataspecer/.

The component looks for the following parameters in the query part of the URL:

- newConversation: If true, the conversation creation dialog is shown to the user.
- Uuid: The IRI of the Dataspecer package.
- packageName: The name of the Dataspecer package

These values are pre-filled into the conversation creation dialog.

3.2.2 Conversation view

This view is a React component with many states, which utilizes the useState function from React.

API calls for new user messages

When the user sends a message, the frontend initiates a sequence of three API calls to the backend.

- It first sends a POST request to log the user's message
- Followed by a GET request to retrieve the system's reply.
- Finally, it makes a third GET request to retrieve the mapped data specification items to display in the right-side panel.

Message rendering

The view renders three distinct message types: welcome, user, and reply messages. The rendering logic for all messages is encapsulated in a separate `MessagesList` React component. This decoupling ensures the code is extensible and easy to maintain, which proved valuable during development as the message structure evolved.

User message highlighting

To show the interactive user message with highlighted words, there is a separate function named renderMessageWithMappedItems, which takes an array of MappedItem objects, which include the indices for the words in the user message that should be highlighted. The method renders these words as a clickable button, which upon clicking, will display a summary of the item.

3.3 Backend

The backend is a monolithic application written in C# using the Minimal API framework. While deployed as a single unit, the system is designed with a layered and highly modular architecture to ensure it is extensible and easy to maintain.

The backend is logically divided into three layers:

1. **Business core layer:** This is the heart of the application, containing all the core business logic. It orchestrates the flow of data, handles conversation state, and processes user requests by interacting with the connectors.
2. **Connectors layer:** This layer is a set of interfaces that abstract communication with external systems. This design ensures that the business core is not directly dependent on any specific external service.
3. **External systems layer:** These are the third-party services that the application depends on, such as various Large Language Models (LLMs) and the Dataspecer tool.

3.3.1 Connectors

This layer defines two main interfaces: ILlmConnector and IDataspecerConnector. This design allows for different implementations to be swapped in without affecting the core business logic.

3.3.1.1 Dataspecer connector

The sole purpose of this connector is to retrieve the data specification files (DSV and OWL) from Dataspecer. It is responsible for fetching these files and returning their contents to the business core layer.

3.3.1.2 LLM connector

To demonstrate the system's independence from any specific LLM, I have created two separate implementations: OllamaConnector and GeminiConnector. Each connector is responsible for a specific LLM and its unique requirements, including prompting and response processing.

3.3.1.3 LLM interaction

The system's interaction with the LLMs is handled through a pair of dedicated components for each connector:

- Prompt constructor (ILlmPromptConstructor): This component transforms the input from the business core into a format optimized for its specific LLM. For example, the LlamaPromptConstructor provides a relevant subset of the data specification to the smaller llama3.3:70b, whereas the GeminiPromptConstructor feeds the entire OWL content of the data specification into its prompt template.
- Response Processor (ILlmResponseProcessor): This component receives the raw output from the LLM, which is typically in JSON format as requested in prompt templates, and converts it into the data structures that the business core layer works with. This ensures the core logic can process the LLM's response consistently, regardless of which LLM was used.

3.3.2 Business core layer

This layer is the heart of the backend, containing the core business logic. It is composed of three main services: a Data Specification Service, a Conversation Service, and a SPARQL Translation Service. Each service has a single, well-defined responsibility, which ensures a strong separation of concerns.

To make the system easier to maintain and debug, only the Data Specification Service and the Conversation Service have the right to store data to the database, ensuring a single point of data entry for persistence. An exception to this rule is the response processor class, which is allowed to fill in the summary for each data specification item whenever the LLM returns it. All other classes can still access the database for reading but must not write to it.

3.3.2.1 Data specification service

This service is responsible for retrieving and processing the data specification that the user wants to work with. It utilizes the Dataspecer connector to retrieve the DSV or OWL files and then extracts the data specification items from those files for use by the other services.

3.3.2.2 Conversation service

This is the most complex service, managing all aspects of the conversation. Its responsibilities include:

- **Conversation creation:** It creates a new conversation and generates an initial welcome message. This message contains a summary of the data specification and suggests possible starting points for the user.
- **Message processing:** When a user sends a new message, this service adds it to the conversation. It then processes the message by mapping the user's natural language to the data specification items and generating suggestions for the user.
- **Reply generation:** The service generates the chatbot's reply message, which includes the mapped items, suggestions, and the SPARQL query.
- **Message preview:** It can verbalize a "suggested message" for the user after they have selected suggested items, providing them with a clear, natural-language preview of the expanded query.

The data specification items successfully mapped during the conversation are stored in the conversation itself, forming a mapped substructure. This substructure represents the relevant subset of the full data specification and serves as the primary input for the SPARQL translation service.

3.3.2.3 SPARQL translation service

This service is solely responsible for generating a SPARQL query from the mapped substructure. This translation happens only once, when the chatbot's reply message is being generated, ensuring that the heavy lifting of query generation is done efficiently on the backend before the reply is sent to the frontend.

3.3.3 Controllers

The final component of the backend are the controllers: IDataSpecificationController and IConversationController. These controllers serve as the entry point for all incoming requests from the frontend and act as an interface to the business core.

Their responsibilities include:

- **Request validation:** They validate incoming data from the frontend, ensuring the data is in the correct format and that any referenced resources, such as conversations or data specifications, exist in the database.
- **Service orchestration:** They orchestrate the business logic by calling the appropriate methods on the service layer. For example, the `ProcessIncomingMessage` method in the IConversationController would coordinate calls to the `AddUserMessageAsync` and `GenerateReplyMessageAsync` methods on the IConversationService.
- **Response adaptation:** They adapt the results from the business services into the specific data format the frontend expects, ensuring a clear and consistent API contract.

3.4 Backend implementation details

This chapter provides a deeper dive into the technical implementation of the backend modules.

3.4.1 Data model

The backend data model is designed to represent and manage the entire conversational flow and its associated data. It is composed of a core set of classes that store conversational history, data specification items, and the relationships between them.

Data specification items

The system models three types of data specification items, which correspond to OWL ontology elements: ClassItem, ObjectPropertyItem, and DatatypePropertyItem. These classes store the following properties for each item:

- Iri: A unique identifier for the item.
- Label: A human-readable name for display in the frontend.
- Type: An enumeration (Class, ObjectProperty, or DatatypeProperty) to easily distinguish the item's role.
- Context properties: The OWL annotation and rdfs comment are stored to provide valuable context for the LLM during prompt construction

Messages

The system defines three types of messages: WelcomeMessage, UserMessage, and ReplyMessage. Each message type shares a core set of properties:

- TextContent: The textual content of the message.
- Conversation: A link to the conversation the message belongs to.
- Timestamp: The time the message was created.
- Sender: An enum value to indicate the source of the message (either system or user).

Specific message types:

- **WelcomeMessage:** Contains a summary of the data specification and initial suggestions to guide the user's first message. There is only one welcome message per conversation.
- **UserMessage:** Represents a message from the user. It contains a reference to its corresponding ReplyMessage.
- **ReplyMessage:** Generated by the system in response to a UserMessage. It contains the final SPARQL query that was translated from the mapped substructure.

Mapping and suggestions

Two helper classes are used to store information about mapped and suggested data specification items, which are not stored directly on the message classes.

- **DataSpecificationItemMapping:** Links a UserMessage to a mapped data specification item. It also stores the MappedWords, which are the exact words from the user's message that the LLM identified as corresponding to this item, enabling highlighting in the frontend.
- **DataSpecificationItemSuggestion:** Links a UserMessage to a suggested data specification item. It also stores a ReasonForSuggestion, which is the LLM-generated explanation for why the item was suggested. This class is designed to recommend properties, not classes, as a class without a connecting property is not a meaningful suggestion in this context.

Conversation management

Conversation: The core class that holds and manages all messages. It ensures messages are stored in chronological order. It also contains the mapped substructure (DataSpecificationSubstructure), which is built during the conversation, a list of UserSelection, and an optional SuggestedMessage string.

DataSpecificationSubstructure: Represents the subset of the data specification that has been mapped during the conversation. It is a list of classes, with their associated properties and their domains and ranges. It also stores the user's choices, such as whether a property is optional or a filter expression for a datatype property. All fields in the classes and properties inside the substructure are stored using simple data types – string and boolean. This was a deliberate choice to make the substructure easily serializable to JSON. The serialized JSON string is stored in the database and sent to the frontend whenever requested so that it can be displayed to the user.

UserSelection: A helper class that captures user-specific choices for a suggested item, such as marking a property as optional or providing a filter expression for a datatype property.

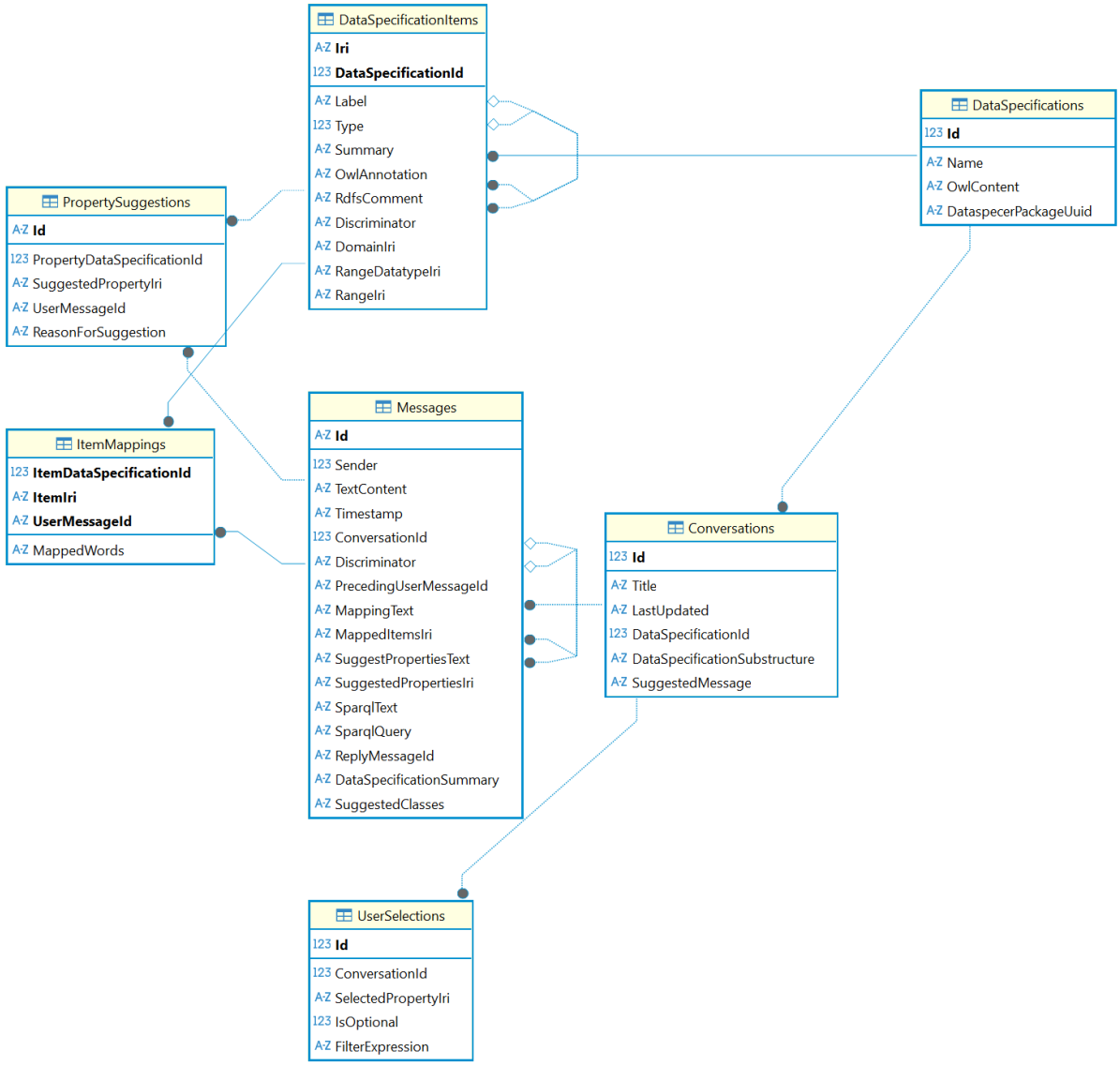
DataSpecification: A helper class that stores the original OWL content, name, and the UUID of the Dataspecer package. It logically groups the data specification items together but does not play an active role in the conversation flow.

The **SuggestedMessage** in the conversation is an optional string that holds a verbalized suggested message. This property is a core part of the mechanism used to distinguish between a new mapping to the full data specification and a mapping to the already established substructure. This functionality is further detailed in the section about the Conversation service.

3.4.2 Database

Data persistence in the system is handled by Entity Framework, utilizing an SQLite database. I chose a “code-first” approach, where the data model classes and their relationships were defined directly in the C# code. This method allows Entity Framework to automatically generate the database schema and handle all SQL queries, abstracting away the complexities of database management. SQLite was selected for its simplicity, as it operates as a single file on the disk, making it ideal for a small-scale, containerized application and simplifying deployment.

Figure 16: Database schema



3.4.3 Connectors

This section discusses the technical details of the various connectors.

3.4.3.1 Dataspecer connector

The DataspecerConnector is the concrete implementation of the IDataspecerConnector interface. It uses the dotnet HttpClient to download the Dataspecer package. It then retrieves either the en/dsv.ttl or en/model.owl.ttl file, depending on the method called. The default Dataspecer URL is **https://tool.dataspecer.com** but this can be changed by modifying the Env:Dataspecer:Url environment variable. This configurability is particularly useful for local Dataspecer deployments.

3.4.3.2 LLM connectors

Two distinct LLM connectors were implemented.

OllamaConnector: This connector uses the OllamaSharp library to interact with a locally deployed LLM instance. By default, it sends prompts to llama3.3:70b at localhost:11434. The connector, as well as its associated prompt constructor and response processor (LlamaPromptConstructor and LlamaResponseProcessor), are specifically tuned for this model. While the model can be changed via the Env:Llm:Ollama:Model environment variable, a different model may not adhere to the expected output format, causing the response processor to fail (e.g., Deepseek's <thinking> block is not handled).

GeminiConnector: This connector is implemented using the Google_GenerativeAI library and requires a valid API key from the ./Secrets/Gemini_api-key.txt file. For the current build, this connector has been commented out of the dependency injection configuration, as the system is only set up to use one LLM at a time. The code is structured to allow easy swapping between connectors but lacks a mechanism to support multiple LLMs simultaneously, a decision made to prioritize development on core features.

3.4.3.3 LLM prompt templates

The system uses six distinct prompt templates, stored as plain text files, to handle different operations. They are called in the following order:

- welcome_message_data_specification_summary.txt: Generate a summary about the data specification and suggest some starting points for the user.
- map_to_data_specification.txt: Maps a user's natural language message to the data specification to identify relevant items.
- get_suggested_items.txt: Suggests additional data specification items that the user might find useful.
- generate_suggested_message.txt: Verbalizes a "suggested message" by incorporating a user's selected suggestions into their original message.
- map_to_substructure.txt: Maps the user's confirmed suggested message to the conversation's mapped substructure, which is used for highlighting on the frontend.
- summarize_data_specification_items.txt: Generates a short summary for mapped or suggested items before they are displayed to the user.

All templates are in Markdown format and follow a similar structure: they begin by assigning a role to the LLM, followed by a description of the inputs, the task, and the required output format.

3.4.3.4 Key prompting strategies for llama3.3:70b vs Gemini

The smaller size of llama3.3:70b required a different prompting strategy compared to larger models like Gemini.

- Data specification handling: Instead of providing the entire OWL file, the Llama prompts pass a flattened JSON list of relevant data specification items to reduce token size and improve

performance. For some operations (e.g., generating a suggested message), the data specification is omitted entirely. For others, like suggesting items, a "local area" around the mapped substructure is provided to the model.

- Consistent data format: To reduce the cognitive load on the LLM, the input data and the requested output are both in a JSON format.
- Explicit output specification: While Gemini prompts list output fields in a list, Llama prompts use an example JSON object or array to explicitly define the output shape.
- Rule reinforcement: Smaller models are more likely to "forget" strict rules in long prompts. Llama prompts repeat key rules (e.g., returning only a raw JSON object) multiple times to ensure the model adheres to the format.

3.4.3.5 Response processing

Assuming the LLM provides a response in the expected format, the response processor, implemented using dotnet JsonSerializer, parses the JSON into a temporary object. It then validates the data against the database and creates the necessary classes for the business core layer. A common issue is that LLMs often wrap JSON output in backticks (e.g., ```json [...]```), so the response processor is designed to automatically detect and remove these characters.

3.4.4 Business core layer

This section discusses the technical details of each module in the business core layer, including data models and service-level implementations.

3.4.4.1 Data specification service

After retrieving the data specification in DSV format, this service uses the dotnetRDF library to convert the DSV syntax into OWL syntax. The conversion algorithm parses each DSV triple and checks the triple's predicate. If the predicate corresponds to an OWL predicate, the algorithm generates a new triple with the corresponding OWL predicate while preserving the original triple's subject and object.

As a fallback, if the Dataspecer connector fails to return the DSV file, this service will ask for the OWL file. If both fail, the service returns null, indicating an error. The preference for the DSV file is due to it containing more information, such as property cardinality, and because some Dataspecer packages may only provide a DSV file.

After a successful retrieval of the OWL representation, this service once again uses the dotnetRDF library to parse the OWL file triple by triple to extract all information on each data specification item. These items are then persisted in the database.

3.4.4.2 Conversation service

This service is responsible for all conversation-related operations.

It is responsible for creating a new conversation and generating a welcome message by calling the LLM to get a summary and initial suggestions for the user.

When the user selects one or more suggested items, this service stores the user's choices and generates a SuggestedMessage for the user. This message is stored in the conversation, which is critical for processing the next user input.

The most complex operation is adding a new user message. The system's logic follows two main scenarios:

New conversation or unmatched suggested message

- If the user message does not match the stored SuggestedMessage or if a SuggestedMessage does not exist (e.g., this is the first user message), the service calls the LLM to map the message to items from the full data specification.
- If at least one item is successfully mapped, it's added to the mapped substructure (DataSpecificationSubstructure in the conversation). When adding properties, their corresponding domain and range classes are also added to prevent "dangling references."
- The service then calls the LLM again to get suggestions based on the mapped substructure and proceeds to generate a reply. If no items are mapped, the service generates a negative response.

Confirmed SuggestedMessage:

- If the incoming message is identical to the SuggestedMessage stored in the conversation, the service concludes that the user has confirmed their previous selections without modification.
- It then adds the user's selections to the mapped substructure.
- The service calls the LLM to map the newly expanded substructure to the words in the user's message, which provides the MappedWords for highlighting on the frontend.
- Finally, it generates new suggestions and proceeds with generating a reply.

The SPARQL query in the reply is generated by calling the SparqlTranslationService with the mapped substructure from the conversation.

When asking for mappings, the LLM might fail to return 'MappedWords' or it might map the same words to multiple items. If 'MappedWords' is empty, we store those mappings without any modifications. In the latter case where some mapped words might overlap, we keep all the mappings but reset the duplicate 'MappedWords' to an empty string.

Similarly, the LLM might fail to return any suggested items, or it might return only two or three items. In this case, there is a fallback mechanism implemented in this service, which randomly picks some properties related to the currently mapped substructure and offers those to the user.

3.4.4.3 Sparql translation service

This service receives a DataSpecificationSubstructure object as input and outputs a SPARQL query as a string. To perform its task, it first converts the substructure into a directed graph. This graph representation makes it easier to traverse the relationships between classes and properties.

The service uses a helper QueryGraph class, where each class in the substructure is a node, and each property is an edge. ObjectProperty edges connect class nodes, while DatatypeProperty edges connect a class node to a leaf node representing a simple data type. Filter expressions and optional flags from the user selections are attached to the corresponding nodes and edges in the graph. The service then traverses this graph to construct the final SPARQL query.

3.4.5 Controllers

There are two main controller interfaces: IDataSpecificationController and IConversationController. These controllers serve as the public API endpoints, handling communication with the frontend and orchestrating calls to the business services. They utilize a set of DTO classes to shape the data for network communication.

- IDataSpecificationController: This controller is implemented but remains unused in the current version. It is a provision for future work to support data specification management,

such as allowing users to upload and manage data specifications within the system, separate from a conversation.
- ConversationController: This is the core controller that handles all communication with the chatbot frontend. It is responsible for deserializing incoming request payloads into the appropriate DTOs, validating them, and then using the results from the ConversationService and database to build and serialize response DTOs for the frontend.

A key part of the ConversationController's job is building the ReplyMessageDTO. In addition to the standard message properties, this DTO contains:

- The generated SPARQL query string.
- A list of mapped data specification items with their summaries and MappedWords for highlighting.
- A list of suggested data specification items, each with a summary and a ReasonForSuggestion.

The controller performs two key transformations to prepare this data for the frontend:

1. **Highlighting calculation:** For each mapped item, the controller calculates the exact character positions of the MappedWords within the user's message. This is what the frontend uses to highlight the relevant text. If a mapped item has no MappedWords, it is still returned to be displayed in a separate section on the frontend.
2. **Suggestion grouping:** The controller uses a helper class called SuggestionsTransformer to group the suggestions. It groups them by the class that the suggestion expands (typically, the suggested property's domain). This allows the frontend to display the suggestions in a logical, structured way, which is crucial for a positive user experience.

3.4.6 Notable API endpoints

The project directory contains a swagger.yml file, which contains the OpenAPI documentation for the full set of available endpoints. This chapter gives an overview of the most important endpoints that the frontend uses to communicate with the backend.

GET /conversations

- Returns all the conversations in the database.
- The conversations only contain basic information: id, title, data specification name and time of the last update.
- The frontend calls this endpoint in the conversation management view to display available conversations to the user.

DELETE /conversations/{conversationId}

- Deletes the specified conversation and all associated resources.
- The frontend calls this endpoint in the conversation management view when the user confirms deletion of a conversation.

GET /conversations/{conversationId}/messages

- Returns all messages in the conversation.
- The frontend calls this endpoint when the user opens up a conversation.

POST /conversations/{conversationId}/messages

- Adds a message to the conversation.
- Returns the newly added user message which also contains the location of the reply message that was generated for it.
- The frontend calls this endpoint when the user sends a message.

GET /conversations/{conversationId}/messages/{messageId}

- Returns the specified message in the conversation.
- The frontend only calls this endpoint to retrieve the reply message from the previous POST /conversations/{conversationId}/messages call.

GET /conversations/{conversationId}/data-specification-substructure

- Returns the conversation's DataSpecificationSubstructure object (which are the mapped items).
- The frontend calls this endpoint to display the mapped items to the user.

4 Deployment

The application is deployed as a single Docker container that contains both the backend service and the frontend. This chapter outlines the prerequisites and steps required for successful deployment.

4.1 Prerequisites

To deploy and run the application, you must have the following installed:

- Docker: For containerizing the application.
- Docker Compose: To build and run the multi-container application with a single command.
- An LLM Instance: The application requires a running Large Language Model (LLM) to function. By default, it is configured to connect to an Ollama instance.

If you don't have Ollama and just want to quickly check out the app, or if you prefer using Gemini over Ollama, consult section 4.4, which describes deploying using Gemini LLM.

4.2 Installation

The installation instructions can be found in the README.md file in the project's repository (link given above). For completeness, they are replicated here.

1. Clone the repository.

git clone https://github.com/Kwantigon/DataSpecificationNavigator

cd DataSpecificationNavigator

2. Change the default Ollama setting. By default, the backend will try to connect to Ollama using the following values:

- Uri: http://host.docker.internal:11434

- Model: llama3.3:70b

This default setting assumes that Ollama is listening on the host machine at port 11434. If Ollama is listening on a different address you must set its URI in the .env file. To change default settings, do the following:

cp .env.example .env

Then replace the dummy values with your own values.

3. Build the images.

docker compose build

IMPORTANT:

The docker-compose.yml specifies mapping of `host.docker.internal` to `host-gateway`. On Linux it means `host.docker.internal` address is likely mapped to 172.17.0.1

If Ollama is listening on 127.0.0.1:11434, then the backend *WILL NOT* be able to connect to it. Make sure Ollama listens on the correct address.

SSH tunneling to Ollama

If the LLM is served by Ollama on a remote server, then you must make sure there is an SSH tunnel to the remote server. The following command creates an SSH tunnel that listens on all interfaces.

ssh -f -N -L 0.0.0.0:11434:localhost:11434 user@remote-server

This is important because if the SSH tunnel is from 127.0.0.1:11434 to the remote server, then the backend cannot reach it from inside the container.

4.3 Running the app

After building the images, run both the frontend and the backend.

docker compose up

By default, the frontend is served at **http://localhost:8080**.

The backend is running at **http://localhost:8080/backend-api**.

You can try sending a GET requrest to **http://localhost:8080/backend-api/hello** to check that the backend is running.

Changing the base URL and port

You can change the base URL by setting the `BASE_URL` value in the .env file. For example, if you want the app to run at **http://localhost:8080/my/custom/path**, set "BASE_URL=/my/custom/path" in the .env file.

Similarly, you can change the default port for the app by setting the APP_PORT value in the .env file.

4.4 Deploying using Gemini

To deploy the application with Gemini instead of the default Ollama connector, you need a Gemini API key.

1. Generate an API key in Google AI studio.

2. Clone and switch to the **gemini** branch in the Git repository.

git clone https://github.com/Kwantigon/DataSpecificationNavigator.git

cd DataSpecificationNavigator

git switch gemini

3. Create a file named **Gemini_api_key.txt** in the directory **DataSpecificationNavigator/backend/DataSpecificationNavigatorBackend/Secrets** and store your API key there.

mkdir backend/DataSpecificationNavigatorBackend/Secrets

echo "[your Gemini key]" > backend/DataSpecificationNavigatorBackend/Secrets/Gemini_api_key.txt

4. Build the docker images.

docker compose build

5. Run the app as described in section 4.3 above.

docker compose up

5 Mapping from specification to implementation

Table 1 shows a mapping from the user story or use case in the original project specification to the implemented features in the project.

Table 1: Features mapping

User story / use case	**Description**	**Implementation**
User Story 1: Choose a Dataspecer package	The user selects a Dataspecer package to use as a data specification.	Implemented in **Conversation Management View**: users can manually create a conversation by entering a Dataspecer package IRI.
User Story 2: Ask questions about data	The user can write natural language questions about data conforming to the chosen specification.	Implemented in **Conversation View**: users type queries in natural language into the chat interface; system maps them to items in the data specification.
User Story 3: Receive a SPARQL query	The application constructs and returns a SPARQL query answering the user's question.	Implemented via **SPARQL Translation Service** in the backend, included in each chatbot reply message. Queries are displayed in the conversation flow.
User Story 4: Expand questions	The user can expand their questions with the help of the application.	Implemented with **Suggested Items** and **Suggested Messages**: users can click suggested items, preview expanded

		messages, and confirm them to refine their query.
Use Case 1: Choosing a data specification	User selects a package in Dataspecer Manager and is redirected to this application, which creates a new conversation.	Implemented in a forked Dataspecer repository. Not integrated into the deployment of the app. The chatbot sends a welcome message into the newly created conversation.
Use Case 2: Sending a message	User writes a question, sends it, and receives an answer with a SPARQL query and related items.	Implemented: API endpoints handle message submission; backend generates replies containing SPARQL + mapped items and suggestions.
Use Case 3: Viewing item summaries	User clicks on a related item to see a summary, with option to add it to their question.	Implemented: highlighted words and clickable suggestions open pop-up dialogs showing summaries.
Use Case 4: Expanding the current question	User selects one or more items, sees a suggested message, and sends it to refine the query.	Implemented: user can select one or more items, backend generates SuggestedMessage and displays in UI for confirmation or editing.

6 Limitations and future work

This chapter discusses the key limitations of the current implementation and outlines possibilities for future development. While the project successfully delivers on its core specification, certain oversights and design choices offer significant opportunities for improvement.

6.1 Limitations

This chapter details the design and implementation choices that restrict the current capabilities of the application while proposing some possible solutions to the posed problems.

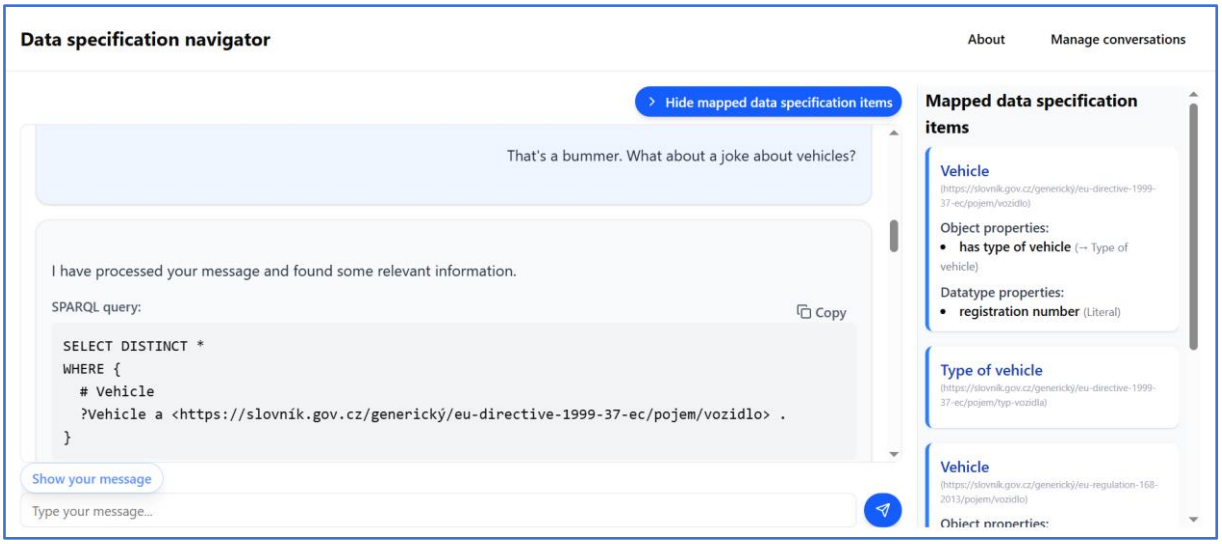
6.1.1 Reliance on LLM performance

The use of smaller LLMs, specifically llama3.3:70b, significantly impacts performance and accuracy. Even with extensive prompt engineering to reduce token count, some operations, like the get_suggested_items prompt, can take several minutes to complete. Furthermore, these models are more prone to hallucinations and may suggest items that are not present in the data specification. A potential solution would be to integrate larger, more capable models.

6.1.2 Ambiguous user messages

The current system struggles with user messages that are semantically unrelated to the data specification but contain words that can be mapped to an item. For example, a query like “What about a joke about vehicles?” will incorrectly map to the “Vehicle” class, as the LLM’s prompt is fine-tuned for a positive mapping case and not a negative one (see Figure 17). This behavior is unintended and can lead to poor user experience. This issue requires fine-tuning the mapping prompt to correctly identify and reject such queries.

Figure 17: Mapping a message unrelated to the data specification



6.1.3 One-to-One word-to-item mapping

The current system imposes a rule that each word or phrase can only map to a single data specification item. This was a deliberate choice to simplify the project's scope, but it limits the chatbot's ability to handle the inherent ambiguity of natural language. The user might envision a different item than the one the LLM maps to, which can be frustrating. A more robust solution would be to present the user with a list of all potential mappings for a given word or phrase, allowing them to select the ones they meant.

It is worth noting that while larger LLMs follow this rule and return only one item for each word or phrase, smaller ones tend to struggle with rules in longer prompts (as discussed in section 3.4.3.4). Smaller LLMs may still return multiple items for the same words. In this case, the backend adds all these mapped items into the mapped substructure and return them to the user.

6.1.4 Limitations in query support

The system is unable to generate correct SPARQL queries for certain cases, such as classes that contain a self-referencing object property. For example, a query about a person who is a "boss of another person" would incorrectly translate into a query about a person who is their own boss. This unintended behaviour stems from the fact that each data specification item is added to the mapped substructure only once. This is a significant oversight that was discovered late in development and would require further analysis and refactoring.

6.1.5 Property sharing

The database schema has an oversight where properties cannot be shared across multiple classes if they have the same IRI. Because each PropertyItem is linked to its domain class, if two different classes (e.g., Person and Dog) share a property with the same IRI (e.g., ex:hasName), the system will only be able to store one instance of that property. This is another limitation that was discovered late and would require a database schema redesign to fix

6.1.6 Duplicate suggestions

The system may sometimes suggest the same property twice in the reply message, once under its domain class and once under its range class. While this was intentionally left as a feature to show the relationship between the two classes, it could confuse users. Similarly, if two different items have the

same label, they are both displayed in the suggestions list (see Figure 18: Vehicle appears twiceFigure 18) and the right-side panel (see Figure 19), which might be confusing to the user.

Figure 18: Vehicle appears twice

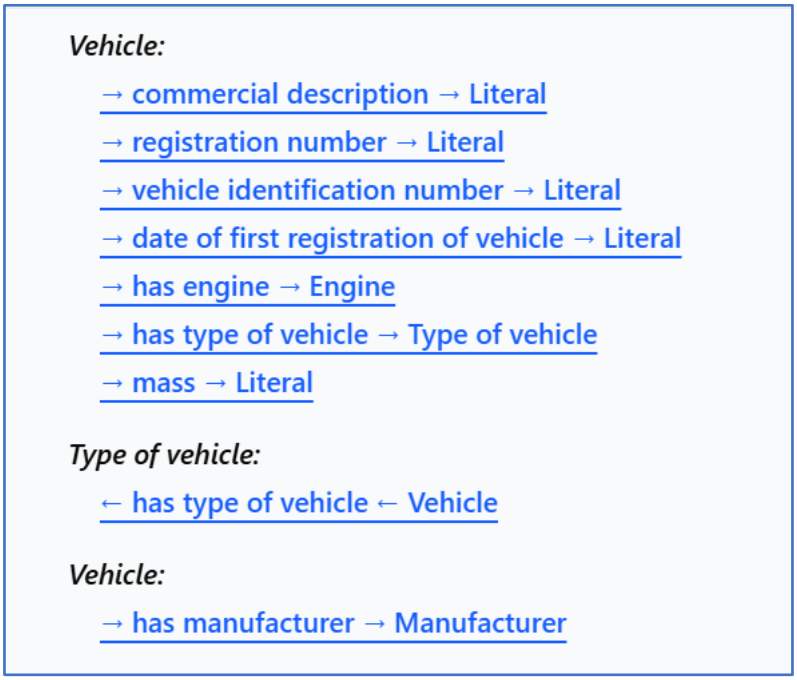
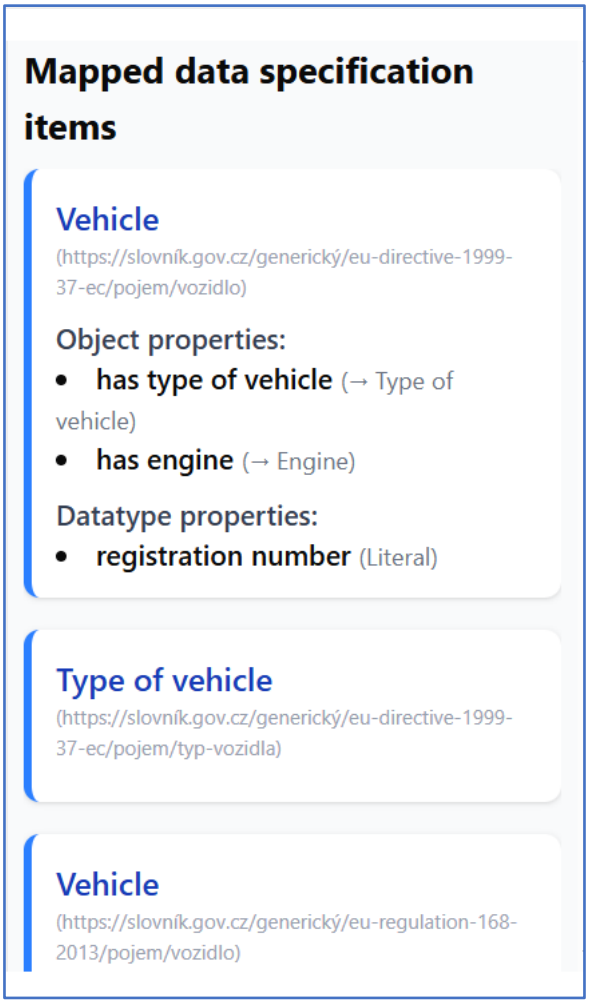


Figure 19: Two vehicle classes



6.2 Future work

Beyond addressing the identified limitations, there is significant potential for further development.

6.2.1 Suggested items

The project can be viewed as a recommender system where the chatbot suggests items to expand the user's query. Currently, suggestions are handled by the LLM with a simple rule ("additional properties from the specification that could expand my question"). A more advanced approach could involve exploring dedicated recommender system techniques to provide more context-aware and personalized suggestions. It would also be beneficial to formally rename "suggested items" to "recommendations" or "recommended items" for greater clarity.

6.2.2 Advanced user message processing

The current system handles follow-up messages by either treating them as a completely new query or by checking for an exact match against the suggested message. A more sophisticated approach would be to incorporate the full conversation context into the LLM's prompt, allowing the system to understand and build upon previous turns even when the user's input is not an exact match. This would enable more fluid and natural conversations.