ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

BPMN diagram generation with ChatGPT

Bachelor thesis at Ulm University.
Submitted for the degree of Bachelor of Science (BSc) in Medieninformatik.

Submitted by:
Niklas Weidl
niklas.weidl@uni-ulm.de
Matriculation number: 1049104

Thesis advisors:
Prof. Dr. Manfred Reichert
Ulm University, Ulm

Supervisor:
Luca Hörner

Ulm, August 2024

Abstract

This thesis explores the capabilites of ChatGPT, a large language model, in the context of generating Business Process Model and Notation (BPMN) diagrams. The primary goal is to develop a tool that leverages ChatGPT to transform natural language descriptions of business processes into accurate BPMN diagrams. The research includes designing and implementing a system that allows users to enter text instructions, processes these inputs using the OpenAI API, and visualizes the generated BPMN diagrams through a user-friendly application. The results illustrate the capabilities of ChatGPT, and the promising future of utilizing advanced natural language processing in the field of BPMN diagram generation.

Version August 9, 2024

Set: PDF-LaTeX 2$_\varepsilon$

Contents

1 Introduction

1.1 Motivation

Business Process Model and Notation (BPMN) [1] is a powerful tool used for the efficient illustration of business processes. It provides a standardized method for illustrating process workflows, making it easier to understand, analyze and improve them. Unfortunately, a significant challenge in utilizing BPMN to its fullest capabilities is the gap between process experts and the technical implementation. Process experts, while possessing deep domain knowledge, often times lack the technical know-how to successfully implement their knowledge into BPMN diagrams. Hence, business processes are frequently explained in prose, where it is much easier to give a quick overview into the process idea, but is much harder to interpret accurately.

With the rise of advanced natural language processing technologies, large language models (LLMs) such as ChatGPT [2] present an exiting opportunity to bridge this gap. ChatGPT does possess the capabilities of understanding prose descriptions and the potential of converting them into more suited formats for BPMN diagrams. This could open the door for non-technical experts and streamline the process of documentation and implementation. Process experts might be able to maximize their capabilities with the help of artificial intelligence.

Research by Sholiq et al. (2022) [3] proposes a method involving natural language processing (NLP) to address this. The method has two stages: analyzing textual requirements to extract fact types, and then mapping these onto BPMN elements using informal rules. Applied to ten case studies, the method demonstrated higher accuracy in generating complete BPMN diagrams from complex sentence structures compared to previous methods. This indicates that NLP can successfully be utilized in the context of automated BPMN generation.

Grohs et al. (2023) [4] explore the potential of LLMs in automating BPM tasks. Through mining processes, their research found that GPT-4 [2] could potentially outperform traditional methods in the context of generating BPMN models from natural

language descriptions, making specialized solutions less necessary. They achieved their results by building a prompt with task descriptions specifying output format and including examples. The paper recommends further research into application of LLMs and improving prompt engineering to enhance performance.
This project will follow the approach of working with ChatGPT to generate BPMN processes from plain text instructions. Given that processes are best visualized in diagrams, an application that could achieve the conversion of prose text to a visualized BPMN diagram would be highly beneficial and would enhance overall efficiency of business process management greatly.

1.2 Goals

The primary goal of this thesis is to make automatic BPMN generation from a user plain text instruction possible with ChatGPT. For that, a tool that interacts with the OpenAI API needs to be developed. This tool then is embedded in a user-friendly application. There the user can view/create/edit the diagrams. The process of user interaction allows the final touches for the resulting BPMN to be ready for implementation.

1.3 Thesis Structure

The thesis is structured as follows. We will first introduce the necessary fundamentals, covering an introduction to Business Process Modeling and Notation, and Natural Language Processing with focus on ChatGPT. Then we define the system requirements which are critical for the development of the project. After a brief insight on how the system performs in BPMN generation, we take a look at the prompt design, a core chapter in the development, enabling BPMN diagram generation. There we discuss core development decisions, driven by the abilities and limitations of ChatGPT. Afterward, the overall structure of the system will be discussed in the System Architecture chapter. It features detailed information on all system components. These system components are then, in the next chapter, explained even further, showcasing the implementation of the project. Finally, we can take a look at some results the system produces and discuss the project's limitations and future work.

2 Fundamentals

2.1 Business Process Modeling and Notation

The Business Process Modeling Notation (BPMN) is a visual modeling language for business analysis applications [5]. It is an open standard notation for graphical flowcharts. These flowcharts are used to define business process workflows. Its simple and intuitive design make it quite popular. BPMN is derived from the synthesis of multiple business modeling notations. The BPMN specification document was released by OMG in February 2006. Version 2.0 [1] of BMPN was developed in 2010, and released in December 2013. Business process modeling provides important benefits to companies and organizations:

- Sets the industry standard
- Provides businesses with the capability of defining and understanding their procedures
- Bridges communication gap between process design and implementation
- Simple to learn, yet able to display complex processes

2.1.1 Symbol Overview

1. Pools and Lanes:
    - Pools represent participants in a business process, which could be an entire organization, a specific department, or a role within the process.
    - Lanes are subdivisions within pools that represent specific roles or departments involved in the process. Lanes help clarify responsibilities and streamline process visualization by breaking down the pool into smaller, more manageable parts.

2. Activities:
   Activities are the tasks and sub-processes performed within a business process, depicted as rounded rectangles. Tasks represent work that cannot be broken down further, while sub-processes are complex activities that can be broken up into multiple detailed tasks.
3. Events:
   Events are occurrences that affect the flow of the process. They are shown as circles and can include icons that specify the type of event trigger. BPMN defines three types of events: Start Events, Intermediate Events, and End Events. Start Events indicate the beginning of a process, Intermediate Events occur between the start and end and influence the flow, and End Events conclude a process.
4. Gateways:
   Gateways are responsible for controlling how a business process flows, by splitting the process flow if there are conditions. They are depicted as diamond shapes.
5. Flows:
   Flows connect the various symbols to one another and indicate the order of action within the process.

2.2 Natural Language Processing

Natural Language Processing (NLP) focuses on the interaction between computers and human language. Key areas of NLP include speech recognition, natural language understanding, natural language generation. Possible application areas include chatbots, programming aid or translating software [6].

2.3 Large Language Models

A large language model (LLM) is a type of artificial intelligence program, that can recognize and generate text. These models are trained using large amounts of text data, to learn patterns and context. LLMs use a type of machine learning called deep learning, which includes the probabilistic analysis of unstructured data, which

eventually enables the deep learning model to recognize distinctions between content without human help. They are fine-tuned for specific tasks, such as conversation, translating languages and coding. However, LLMS are not free of fault, sometimes producing inaccurate information due to being trained on faulty data.[7] The most popular models as of today include GPT-3.5 and GPT-4 by OpenAI[2], BERT [8] by Google and LLaMA [9] by Meta. For this project, our focus is on ChatGPT.

2.3.1 ChatGPT

ChatGPT[2] is a conversational model, developed by OpenAI. The model takes advantage of Reinforcement Learning from Human Feedback to improve user interaction capabilities. The training process involved human AI trainers, who act as both the user and the AI assistant, to provide example conversational data. The models' architecture is further refined with the use of a reward model. This model ranks the quality of multiple model responses. A long iterative fine-tuning process, conducted on Azure AI's supercomputing infrastructure, led to a wide range of conversational context with improved relevance. Despite all the capabilities, ChatGPT is not without limitations or flaws. The model can sometimes produce incorrect or even nonsensical answers, that may sound plausible without further verification. It is also very sensitive to input phrasing, reacting differently to just slight changes in wording. Overall, ChatGPT represents a huge step forward in AI-driven conversational agents.

2.3.2 Prompt Engineering

To achieve desired output effectively, instructions given to large language models need to be carefully designed and refined. Prompt engineering aims to guide the model to more accurate and context-specific responses. Knowing how to utilize the methods in prompt engineering can vastly improve the utility of LLMs. White et al. "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT." [10] discusses methods in prompt engineering, in this project three key methods are used:

- **Persona**:
  Defining a specific role or perspective to the LLM. For example, "Act as a cybersecurity analyst and review the following code for potential vulnerabilities", will guide the LLM to focus on security aspects.

- **Template**:
  To ensure consistency and formatting requirements, a custom structure can be included, for example: "NAME: [Generated Name], JOB: [Generated Job]".
- **Reflection**:
  Tell the LLM to review its output and scan for potential irregularities, or even mistakes. Example: "Reflect on your previous answer and identify any possible mistakes or areas for further detail."

3 System Requirements

A key practice within software development lies within defining the system requirements of a project. These system requirements serve as a blueprint for the development process and help to set expectations for stakeholders. Defining precise and well-thought-out requirements usually results in a robust and efficient system with ease of use for the user. This chapter aims to provide a comprehensive overview of the functional and non-functional requirements.

3.1 Functional Requirements

Functional requirements contain specifications about a systems' behavior, functions and capabilities. They include tasks, data manipulation and processing, essentially defining what a system should do. These specifications are carefully chosen to meet potential user needs and serve as a guideline to guide system development to its intended purpose. For this system, the chosen functional requirements are as follows:

3.1.1 User Interaction

FR1 **Natural Language Input**
The application shall allow users to input natural language descriptions to create BPMN diagrams. This shall be done within a provided user-friendly interface.

FR2 **bpmn-js integration**
The application shall integrate bpmn-js [11] to display the BPMN diagram in the frontend. It also needs to ensure that this integration remains responsive and allows direct diagram manipulation.

FR3 **Viewing BPMN Diagrams**
The application must be able to render BPMN diagrams using the bpmn-js library.

FR4 **Editing BPMN Diagrams**
The user shall be able to alter the contents of a BPMN Diagram, either by entering a new update prompt or by editing the diagram by hand with the bpmn-js editor.

FR5 **Provide overview of all available files**
A table of all available diagrams shall be provided by the application, so a user can get an overview of all their past creations.

FR6 **Provide overview of all data for one singular diagram**
The application shall provide access to all available information of one diagram entry. These include: name, time of creation/last change, prompts and of course the diagram itself.

FR7 **Saving diagram changes**
All changes made by further prompts should be saved automatically. For changes done by hand, the system shall provide a save option.

FR8 **Exporting/Downloading a diagram**
The user might want to download their created diagrams. The system should hence provide an option to download the BPMN diagrams.

FR9 **Error Feedback**
Should at any point an error within the backend occur, that may hinder a process in the frontend, the user should be notified.

FR10 **Help and Documentation**
The application should provide clear help and documentation or suggestive UI design to guide users through the process of using the application.

3.1.2 Backend Processing

FR11 **Natural Language Processing**
The system shall use the OpenAI API to process and convert natural language prompts into a JSON structure containing all necessary process data.

FR12 **Confirm generated JSON integrity**
This application must ensure the received JSON data is correct and format it

if necessary to remove any prose text or non-JSON information before conversion.

FR13 **Convert JSON data into XML**
For the conversion, the system must follow the Camunda XML file format. This is essential, since the BPMN diagram will not be displayable, if the conversion does not adhere to the specified format.

FR14 **Manage files and database**
The system must manage and provide access to all files of interest and keep up a database of all entries. This includes

- getting and providing access to files
- adding database entries
- creating files to store JSON and XML file information
- altering XML files to add changes
- deleting files and database entries

3.2 Non-Functional Requirements

Non-functional requirements, as the name might reveal, deal with requirements not directly visible to the user, since they do not impact the functionality of a system directly. But they do define key aspects of a well running system, such as performance, usability, reliability, security and robustness. If non-functional requirements are met, system efficiency and effectiveness will be ensured. For this system, the chosen non-functional requirements are as follows:

NFR1 **Usability**
The system should provide an intuitive and user-friendly interface. It should be consistent and easy to navigate. Error messages should be clear, so the user knows what to do or what has gone wrong.

NFR2 **Performance**
In the interest of user experience, the system should perform its functions within a reasonable time span. Response time for any user action, aside from diagram generation, which might take up to 60 seconds, should not exceed 2 seconds under normal conditions.

NFR3 **Reliability**
Data integrity should be ensured, with no loss of critical information. Hence, the system should run consistently without failure.

NFR4 **Availability**
The system should be accessible and operational for the user whenever needed.

NFR5 **Accessibility**
The application should run on all commonly used internet browser clients, such as Chromium, Firefox or Opera.

NFR6 **BPMN Version**
The application shall use BPMN 2.0[1].

NFR7 **NLP**
The application shall use ChatGPT[2] for any natural language processing.

4 System Functionality

Before we dive into any development steps, let us illustrate what we expect from the system by taking a look at one interaction example. For this, we will take a very simple user instruction text to test the system and present the generated result, in order to set expectations. The instructions text is as follows:

> I need to illustrate my application:
> We have four actors:
> - The Process Expert, who delivers process descriptions
> - The Universal Expert, who understands these descriptions and translates them into JSON
> - The Implementation Expert, who converts the JSON into XML and finalizes it
> - The Visualization expert, who visualizes the process

The diagram on the next page is one result the system can produce with this given instruction text. This diagram does slightly resemble this system, more on that later. To enable ChatGPT to generate these type of diagrams, we need to design our prompt accordingly.

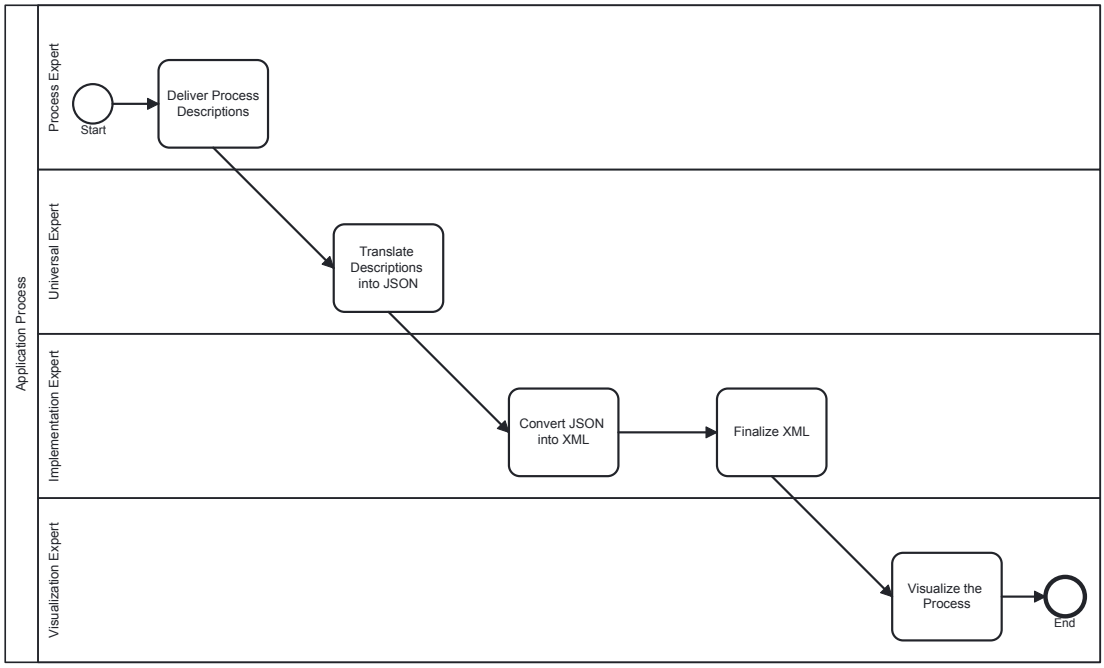

Figure 4.1: Generated result from given user instructions

5 Prompt Design

The entire development process was dominated by figuring out, what the capabilities of ChatGPT were. The response of the GPT is at the heart of this project, it succeeds or fails with it. If it was possible to simply generate BPMN diagrams right away, most of the development process would have been redundant. But due to the nature of ChatGPT being a conversational assistant and not a specially trained diagram generation assistant, that was, at the start of my research, not possible. But with some clever tricks and prompt building, we might be able to get it to generate some BPMN diagrams in text form. For this, some prompt design strategies had to be employed.

5.1 First steps

The very first question we need to get clarification on is whether ChatGPT is familiar with BPMN and secondly, if it is familiar, how well ChatGPT knows BPMN and its rules. For the initial part of development, GPT3.5 was used.

Are you familiar with BPMN?

Yes, I am familiar with BPMN (Business Process Model and Notation). BPMN is a graphical representation for...

ChatGPT, at the start of my research, was very much familiar with BPMN, and knew of all possible BPMN symbols, which was fantastic news for the project. It was also able to textually describe an example diagram.

Generate an example bpmn diagram

While I can't directly create visual diagrams here, I can guide you through the steps to create a simple BPMN diagram using a typical scenario. Let's take an example of an "Order Processing" process. Here's a textual description that you can use to create a BPMN diagram:
1. Start Event: Order Received
2. Task: Validate Order
3. Gateway: Order Valid?
- If no: End Event: Order Rejected
- If yes: 4. Task: Process Payment
5. Gateway: Payment Successful?
-If no: End Event: Payment failed
-If yes: 6. Task: Ship order
7. End event: Order completed

If we were to draw the entire diagram, we need to do some work ourselves now. We do not have any specifications on where any component is supposed to be placed, we only know of the connections between the components. For a human, that is not an issue, and we can just choose a design to our liking, so the diagram might look like this:

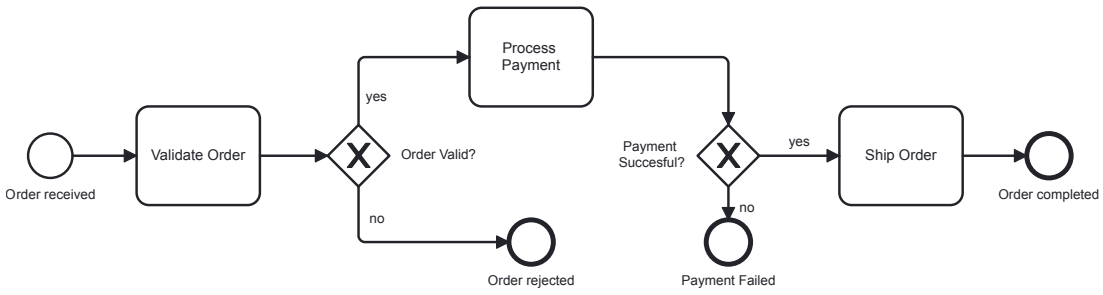

Figure 5.1: Manually created diagram from generated response

The resulting diagram seems to be a correct, ChatGPT seems to do a good job at accessing which symbol is ideally used at which point in this scenario. But, we do not want to draw the diagram ourselves. Ideally, we receive the text in a format that we can directly feed into a BPMN viewer. If we request the diagram not be a plain text instruction, but rather of a file format that a viewer such as bpmn-js can view, ChatGPT decided to return a JSON. Interestingly, the chosen format by ChatGPT is not readable by any commonly used viewer, due to missing instructions such as

the definitions and placement of the symbols. Here is the beginning of one of the generated results, so we can get a grasp of what ChatGPT delivered.

```
{ "scenario": "Order Processing",
"events": [
{
"type": "start",
"name": "Order Received"
},
{
"type": "task",
"name": "Validate Order"
},
{
"type": "gateway",
"name": "Order Valid?",
"branches": [
...
```

Commonly used BPMN viewers use the XML file format. Here, a certain standard is followed, which is BPMN 2.0 [1]. If we do not adhere to this standard, the diagram will be displayed incorrectly or even not at all be displayed. There are multiple options we can choose now:

- Use pattern recognition and teach ChatGPT the BPMN 2.0 structure, it should return by presenting other diagrams.
- Include the structure in our prompt message
- Keep the JSON approach and convert to XML ourselves.

The most comfortable approach from the point of view of a developer would be the first one, where we use pattern recognition. Which at first seems to work perfectly fine, but at closer inspection, and most importantly when importing into a BPMN viewer, the results often times are not valid. ChatGPT seemed to have trouble generating the XML file. It often times forgot certain parts of the structure, such as the flows or the definitions, or simply was unable to generate syntactically correct XML output. As a rule of thumb, the longer a response, the higher the potential for errors. The few times generation worked successfully, a different issue arose: The placement of components and flows. GPT3.5 simply failed to understand how objects

relate to each other on a 2D plane. This was not unsurprising, we did not really provide any guidance for the GPT, how to place the items.
Having learned this, we can view the other two approaches. Keeping the JSON approach, seems very beneficial for due to not having to write the definitions' header of the file and not having to write another big text block for each component to place them. We could just define a structure for ChatGPT to use for its JSON generation. Let us look at the first promising prompt:

> For this build a table with the following columns:
> - a unique ID for each element (append an e in front of every ID)
> - a name of the element
> - the BPMN Element type (for example Exclusive Gateway)
> - a possible x position for the element
> - a possible y position for the element
> - connection ID(s)

> Then build a second table with the following components, use information from the first table.
> - connection ID (append a c in front of every ID)
> - connection start
> - the connection target
> - the type of connection
> - the starting point (x,y) of the connection
> - the target point (x,y) of the connection

Note that positions for the components are now included. We have given clear instructions on what fields should be included for what item, with a separate table exclusively for the connections. All the items receive a unique ID, so that we can clearly manage which elements connect, earlier usage with just names often times lead to misunderstandings. For the positions we need to add a little bit of help, since we already know that ChatGPT struggles with Cartesian coordinate systems:

> Keep in mind that from this table we're looking to create a BPMN diagram, so chose parameters such as position and connections carefully. When choosing the position of an item, keep in mind that the objects within the diagram have a certain size, so choose an appropriate distance (say at least 50 points) between the items. For the connecting flows, chose the middle point of the connecting items.

Converting the generated results to XML produced much better results now. So after all, we kept the JSON approach, but included a custom structure in our prompt to manipulate the generated result structure. Later on, we can easily convert the answers to XML.

5.2 Understanding user instructions

So far, we have only looked at the possibility of creating a diagram. Now we can take a look at how ChatGPT identified components and a flow within an instruction text. During testing, a pattern quickly emerged, so as a rule of thumb: *The generated results are only as good as the given prompt*. If a prompt is too vague, the result will often times not be very good, since ChatGPT needs to make guesses about the scenario steps. Let us take a look at an example:

> I want to buy a car. The vendor offers me a Ferrari or a BMW. I chose one of the cars and drive home.

This can be misinterpreted in multiple ways. We could potentially just have start and end events, and two tasks in our diagram: Want Car, Vendor offers, Choose Car, Drive home. In this scenario, the intention of the prompt was to provoke a Gateway generation, which often times did not happen, due to the prompt being a bit too vague.
To avoid confusion and to give some guidance on which type of symbol to use, a Component Identifying guide has been created. This guide includes descriptions on all BPMN symbols and provides examples for every single one of them. Due to that guide being quite a long document, we will only look at some parts to understand the core ideas. For the complete document, please refer to the source code. The guide

works by providing examples, so that ChatGPT make use of pattern recognition to easier identify the correct component to use. Let us take a look at the definitions in the guide of a User Task and a Script Task:

- *User Task*: A task that is performed by a human actor. This typically involves human judgment and is often performed with the assistance of a software application.
  Examples:
  - Approving a Loan Application: A bank employee evaluates and approves or rejects loan applications.
  - Filling Out a Survey: A customer service representative fills out a feedback survey after a call.
  - Reviewing a Document: A legal professional reviews and annotates a legal document.
- *Script Task*: A task that is executed by a computer application or script. It's automated and doesn't require human intervention.
  Examples:
  - Data Backup: An automated script that performs regular data backups in an IT system.
  - Running a Credit Check: An automated process that runs a credit score check when a loan application is submitted.
  - Generating Monthly Reports: A script that automatically compiles and generates monthly financial reports.

When not using the guide, ChatGPT only used basic tasks and little to no gateways, now it uses more specific BPMN symbols. What also helped, was assigning ChatGPT certain role in the chat. Now prior to any message, we would send this:

> You are a BPMN Component understanding expert.
> Your task is to analyze and identify BPMN components within a given text.

These two sentences transform the persona of the GPT, to guide it towards focusing more on identifying the right component. Another thing that helped was including guides where to place certain items, for this we use instructions the GPT can self reflect on.

Be sure to follow these guidelines, self check on these always:

1. **Start Events:**
    - **Single Start Event:** Typically, a process should have only one Start Event to mark its beginning.
    - **Multiple Start Events:** In complex scenarios, multiple Start Events can be used to represent different triggers for the process.
2. **End Events:**
    - **Single or Multiple End Events:** A process should conclude with an End Event. It's common to have multiple End Events indicating different process conclusions.
    - **Placement:** Ensure the End Event is inside the pool/lane, making the pool large enough to fit it.
3. **Gateways:**
    - **Flow Control:** Use Gateways to control process flow, especially for branching (divergence) and merging (convergence).
    - **Correct Usage:** Ensure Gateways correctly represent decision points and parallel processes.
4. **Tasks and Sub-Processes:**
    - **Tasks:** Choose Tasks for atomic activities (those that cannot be broken down further).
    - **Sub-Processes:** Use Sub-Processes for complex activities that can be broken down into smaller steps.
    - **Specific Task Types:** Consider using specific Task types, such as a "User Task".
5. **Sequence Flow:**
    - **Connections:** Use Sequence Flows to connect elements in a process and indicate the order of activities.
    - **Avoid Crossings:** Avoid crossing Sequence Flows to maintain diagram clarity.
    - **Loops:** Sequence Flows can go back to an earlier component to create loops.

6. **Parallel vs. Sequential Processes:**
    - **Parallel Gateways:** Use Parallel Gateways for activities that can occur simultaneously.
    - **Sequential Activities:** Connect sequential activities directly or through appropriate Gateways to indicate their order.
7. **Pools and Lanes:**
    - **Pools:** Use Pools to represent different participants or entities in a process.
    - **Lanes:** Within Pools, use Lanes to categorize activities under different roles or departments.
8. **Events:**
    - **Intermediate Events:** Use Intermediate Events within the process flow to indicate occurrences between Start and End Events.
    - **Boundary Events:** Attach Boundary Events to Tasks or Sub-Processes for handling exceptions or timeouts.
9. **Data Objects and Message Flows:**
    - **Data Objects:** Include Data Objects to show the data required or produced by activities.
    - **Message Flows:** Use Message Flows to illustrate communication between different Pools or external entities.
10. **Artifacts:**
    - **Annotations:** Use Artifacts like Annotations for additional information that aids understanding without affecting the flow.

These additions again improved the results. Note, that we are preparing another addition to the existing structure, because we do still face a slight issue. At this point, we could not use Pools and Swim lanes using the current structure at the time. In order to use them, it made sense to redesign the current structure template, by adding a desired format. This is the JSON structure template the GPT uses as of right now, it works by applying pattern recognition:

```
  {
"Pools":[
  {
  "ID": "p1",
  "Name": "Pool 1",
  "XY": [0,50],
  "width": 400,
  "height": 150,
  "Lanes": [
    {
      "ID": "l1",
      "Name": "Lane 1",
      "XY": [0,50],
      "width": 400,
      "height": 150,
      "Components": [
        {
          "ID": "e1",
          "Name": "Start",
          "Type": "Start Event",
          "x": 100,
          "y": 100,
          "Incoming": [],
          "Outgoing": ["c1"]
        },
        {
          "ID": "t1",
          "Name": "Task",
          "Type": "Task",
          "x": 200,
          "y": 100,
          "Incoming": ["c1"],
          "Outgoing": ["c2"]
        },
        {
          "ID": "e2",
          "Name": "End",
          "Type": "End Event",
          "x": 300,
          "y": 100,
          "Incoming": ["c2"],
          "Outgoing": []
        }
      ],
      "Flows": [
        {
```

```
            "ID": "c1",
            "Start": "e1",
            "Target": "e2",
            "Type": "Sequence Flow",
            "StartXY": [100,100],
            "TargetXY": [200, 100],
            "Descriptor": "Begin"
          },
          {
            "ID": "c2",
            "Start": "e1",
            "Target": "e2",
            "Type": "Sequence Flow",
            "StartXY": [200,100],
            "TargetXY": [300, 100],
            "Descriptor": "Finish"
          }
        ]
      }
    ]
    }
  ]
}
```

The resulting structure is a tree, Pools contain Lanes, and the lanes contain all components and flows. This way, we can easily convert the generated answer to our desired BPMN XML structure. Using this JSON structure, tokens and time is also saved, due to ChatGPT not having to generate as much content as if we were to use the XML structure, which is much more verbose.

5.3 Further diagram changes

One requirement at the start of the project included updating a diagram after it has been generated. Updating, in this case, means creating, deleting or changing symbols of the diagram. There are multiple ways we can approach this: We can just regenerate the entire diagram with our updates, which would take a long time, or we can design a new structure the GPT can follow.

Once you have generated the entire JSON, you will respond to new requests differently. Your task is now to make changes to the JSON, upon user request.
So on your first response, you answer with the whole JSON, then only with changes you are making to the JSON. You need to make changes to the JSON on request, but only show the changes that you're making.
There will be four kind of changes:
- **CREATE** (where you create a new component or flow)
- **DELETE** (where you remove a component or flow, remember we may need to adjust flows when deleting components)
- **UPDATEPARENT** (where you change the parent of an item, for example move a component to a different lane)
- **ALTER** (where you modify a component or flow)

The design includes four options, creation, deletion, change of parent and altering. Using these four commands, we can build another JSON structure the GPT can return in order for us to convert the changes into the JSON file, more on that in Chapter 7. The designed JSON structure looks like this:

```
{
   Content: [
     {
       "type": "CREATE",
       "item": "Pool",
       "parentPool": "",
       "parentLane":"",
       "content": {
         "ID": "p1",
         "Name": "Pool 1",
         "XY": [0,50],
         "width": 400,
         "height": 150,
         "Lanes": [
           {
             "ID": "l1",
             "Name": "Lane 1",
             "XY": [0,50],
             "width": 400,
             "height": 150,
             "Components": [
               {
```

```
                "ID": "e1",
                "Name": "Start",
                "Type": "Start Event",
                "x": 100,
                "y": 100,
                "Incoming": [],
                "Outgoing": []
              }
            ],
            "Flows": {}
          },
        ]
      }
    },
    {
      "type": "DELETE",
      "item": "Component",
      "id": "e1"
    },
    {
      "type": "ALTER",
      "item": "Lane",
      "id": "l1",
      "changes": {
        "ID": "l1",
        "Name": "Main Lane",
        "XY": [0,50],
        "width": 300,
        "height": 250,
      }
    },
    {
      "type": "UPDATEPARENT",
      "item": "Component",
      "id": "e1",
      "from": "l1",
      "to": "l2"
    }
  ]
}
```

Using pattern matching, the GPT can now generate update instructions from new user instructions, which can be converted later on.

5.4 Assistant

It made sense to employ an assistant to make interaction with ChatGPT a lot more pleasant. A GPT assistant works, by having prior instructions before processing the user instructions. Using assistants enables OpenAI API usage, opening the door for us to begin the implementation process. Assistants offer file processing, so we can add custom files, with instructions or information to it. For this projects' assistant, we include the Component Identifying Guide from earlier and the entire BPMN 2.0 Specification [1]. These two files alone help identify and assign the right symbols immensely. Additionally, assistant offer some tweaks such as *temperature* and *top-p*. The temperature is a parameter that influences the randomness of the model's prediction. Low temperature means the model is more deterministic, and the same prompt will very likely lead to the same result. For this project, we want similar answers for similar input prompts, so we use the lowest temperature (0.01). The Top-p value influences how creative the model is with answer generation. A high value will lead to creative answers, while a low value will lead to the opposite. We have gone for a high top-p value, in hopes that the diagrams will also get a bit more creative. However, I cannot confirm this, the temperature value changes the answers, in this project at least, the most.
Finally, we can take a look at the outline of the final prompt design we use for the assistant. We have already gone over most of the vital parts in detail, so this is just to summarize, for the entire prompt, please refer to the source code section.

- Ban prose text and for only JSON output, to prevent any text prior or after the generated JSON in the answer
- Initiate persona ("You are a BPMN Component understanding expert.")
- JSON structure template
- Structure deeper explanation with guidance on what to place
- BPMN symbol guidance for reflection
- Instructions in case of error
- Update instructions after second message
- Update JSON Structure

Overall the instruction text is 237 lines long, and takes advantage of three core prompt design ideas: persona, templates and self reflection. Using all this, we can achieve the best results.

6 System Architecture

In this chapter, we take a dive into the detailed architecture of the system, outlining its core components, interactions, and design principles. Once we have examined the key processes and software elements, we have a detailed understanding of how the system deals with its processes.

6.1 Overview

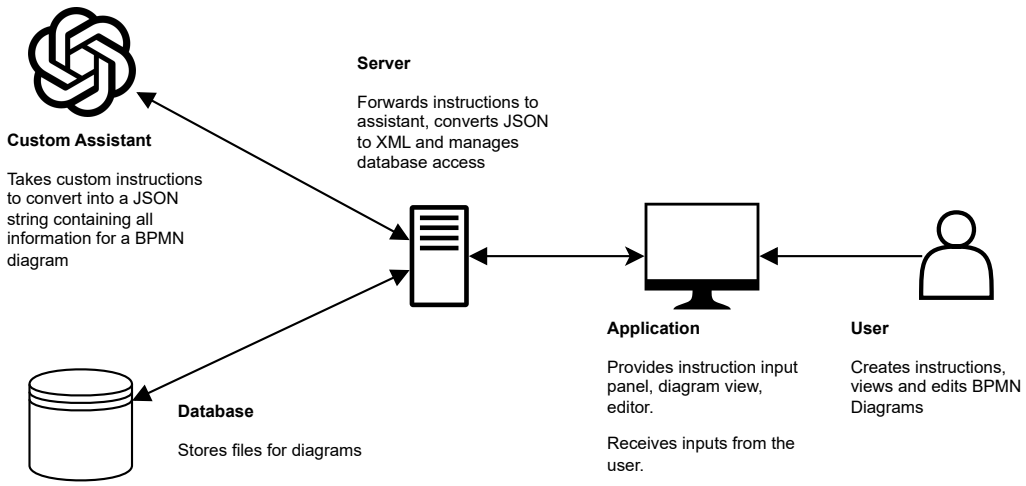

Figure 6.1: Basic system overview

This is a simplified system overview. The system consists of four actors: The *OpenAI Custom Assistant*, which is accessed via the OpenAI API, the backend *server*, which handles all OpenAI API calls and the file-based database of all created diagram files, the frontend *application*, which requests data from the server to display and manipulate, and lastly the User who interacts with the application, viewing and editing diagrams.

Plainly speaking, this is a typical case of a *Client-Server Architecture*. This has been done to ensure future scalability and, as of right now, to give multiple users access to the same resources on the server.

6.2 System Processes

The primary use case of the system involves both creating and updating diagrams. While these processes are quite similar, there is a key distinction between them. In the case of updating diagrams, there is an additional step that requires accessing existing files to fetch the desired thread for the diagram. To provide a clearer understanding, we will focus on the update process. Given the complexity, we will take a look at a sequence diagram, illustrating the entire process from start to finish.

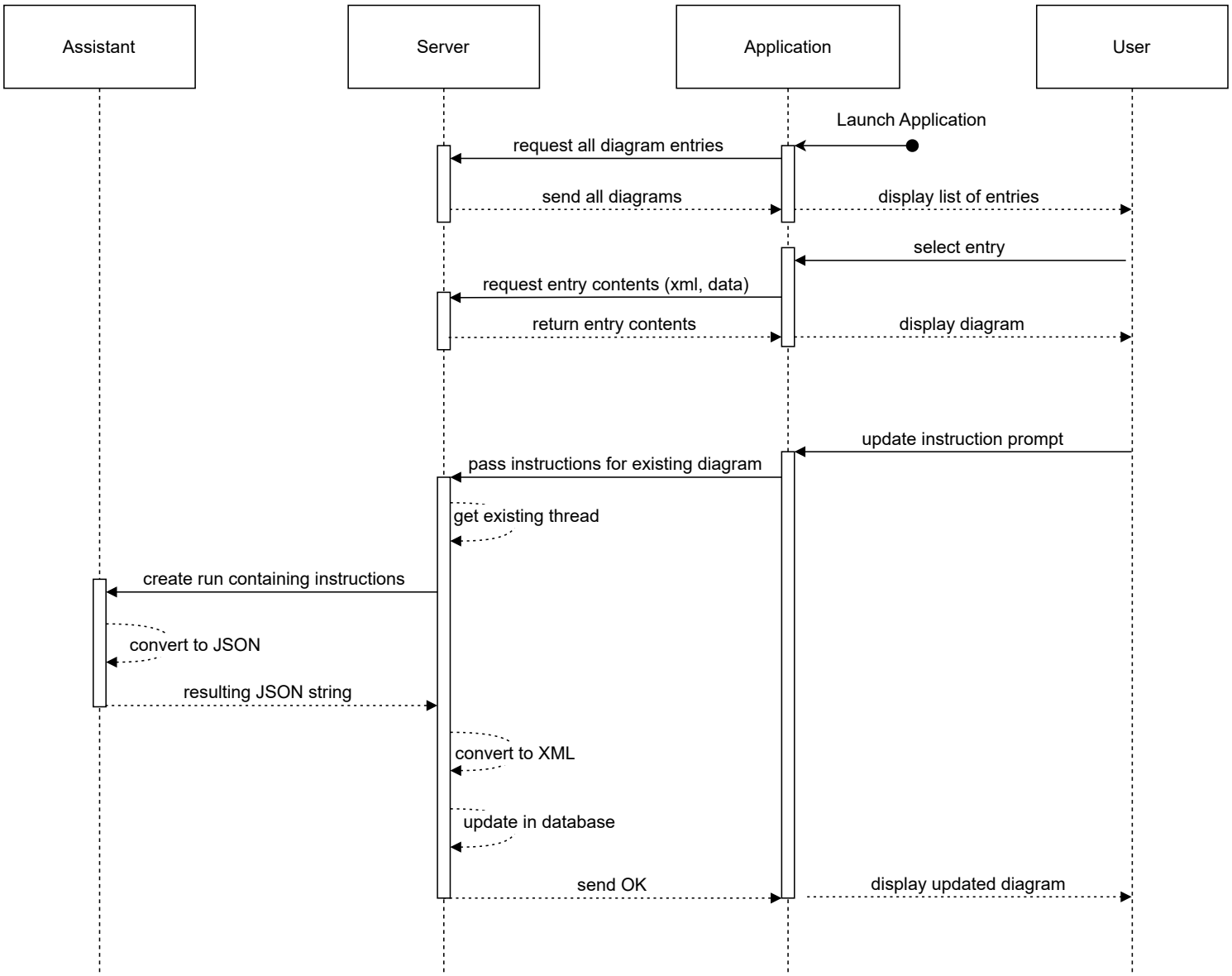

Figure 6.2: Sequence diagram for updating a diagram via prompt

The diagram illustrates the workflow between user, application, server and assistant.
To begin, the user launches the application, which triggers a request of all available diagram entries from the server, which it retrieves and sends back to the application.
The user is then presented with all available options, where they can review the entries. Once an entry has been chosen by the user, the contents will be requested by the application, and once returned the user can see and edit the chosen diagram.
In this scenario, the user decides to alter the diagram by writing an update prompt. This prompt is sent to the server, there the associated thread is used to create a new run with the new user instructions. The assistant generates a new JSON structure, containing the update data, and returns it to the server. The update is processed and converted to XML, and attributes in the database are adjusted to match the new information of the current diagram and it's instructions.
The server confirms all is OK, and the application refreshes the file list and contents, and displays the updated diagram to the user. This process might be repeated many times until the user reaches a satisfactory diagram result.

6.3 System Components

6.3.1 Custom Assistant

As discussed in Chapter 5, a custom assistant has been created For this project. We have already gone into great lengths to explain the design of said assistant. To summarize: The custom assistant works by taking a natural language prompt and converting it into a JSON string containing necessary information for a BPMN diagram. These JSON strings can then be sent to the server for further conversion. To access the assistant, we make use of the OpenAI API, which is managed by the server.

6.3.2 Server

The backbone of this project is build on an Express server, a framework built on Node.js. Express provides a flexible approach to building web applications and APIs, and enables handling of HTTP requests and responses. The simplicity and

the routing capabilities make it perfect for this project. Let us take a look at how the server is structured.

API

In order to communicate with the server from our frontend application, we need an API, with custom endpoints. Here, we are looking at a collection of requirements: We need to access the database, which needs to be able to be manipulated. Then we need to be able to create/continue the OpenAI threads. And lastly, we need to get our files. Let us take a look at the API endpoints for the server:

Endpoint	`/database`
Method	GET
Description	Retrieves the entire content of the database stored in JSON format.
Success Response	**Code**: 200 OK **Content**: The entire JSON content of the database.
Error Responses	**Code**: 500 Internal Server Error **Content**: `Error reading the database file`

Table 6.1: API Endpoint: Get Database Content

Endpoint	`/data/xml/:filename`
Method	GET
Description	Fetches the content of a specific diagram file.
URL Parameters	`filename`: The name of the diagram file to fetch.
Success Response	**Code**: 200 OK **Content**: The content of the requested diagram file.
Error Responses	**Code**: 403 Forbidden **Content**: `Access denied` (If the requested file is outside the allowed directory)
	Code: 404 Not Found **Content**: `File not found` (If the requested file does not exist)

Table 6.2: API Endpoint: Fetch Diagram Content

Endpoint	`/create`
Method	POST
Description	Creates a new BPMN diagram based on the provided input string.
Request Body	`inputString`: The string input to generate the BPMN diagram.
Success Response	**Code**: 200 OK **Content**: `{ "response": "<Generated XML Content>" }`
Error Responses	**Code**: 500 Internal Server Error **Content**: `Error message if something goes wrong during the diagram generation process.`

Table 6.3: API Endpoint: Create Diagram

Endpoint	`/update`
Method	POST
Description	Alters an existing BPMN diagram using the OpenAI API based on the provided input string and diagram ID.
Request Body	`id`: The ID of the diagram to be updated. `inputString`: The string input to update the BPMN diagram.
Success Response	**Code**: 200 OK **Content**: `{ "response": "<Updated XML Content or Response>" }`
Error Responses	**Code**: 400 Bad Request **Content**: `{ "response": "Invalid request. Please provide the id and inputString." }`
	Code: 500 Internal Server Error **Content**: `Error message if something goes wrong during the update process.`

Table 6.4: API Endpoint: Update Diagram

Endpoint	`/delete`
Method	DELETE
Description	Deletes an existing BPMN diagram based on the provided diagram ID.
Request Body	`id`: The ID of the diagram to be deleted.
Success Response	**Code**: 200 OK **Content**: `"Deleted"`
Error Responses	**Code**: 500 Internal Server Error **Content**: `Error message if something goes wrong during the deletion process.`

Table 6.5: API Endpoint: Delete Diagram

Server Components

The server essentially consists of three parts, one controlling communication with the assistant of the OpenAI API, one handling the conversion of the messages received from the assistant and the last one writing and managing the database. All three parts need to communicate in order to correctly resolve the requests made to the server. The most important one is the part controlling the communication with the OpenAI API, calling all conversion or database management steps from there. See next page for visual illustration.

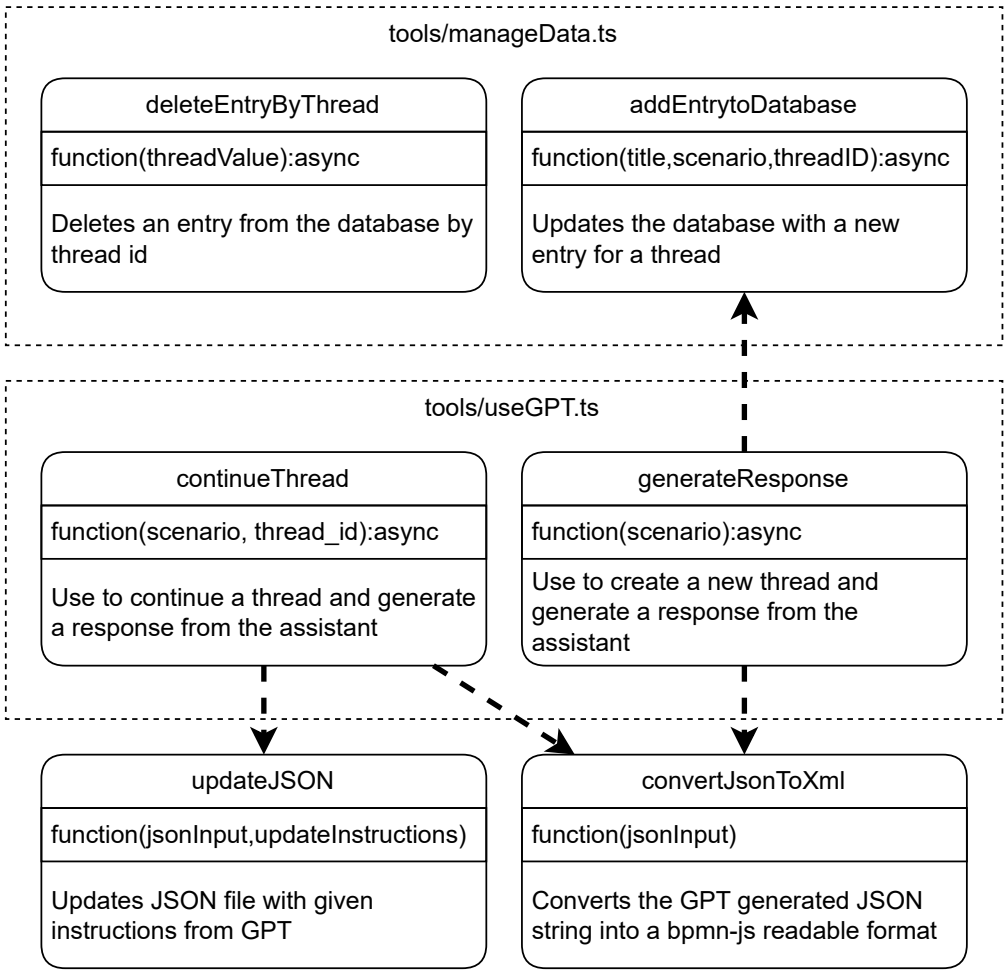

Figure 6.3: Server function overview

- `deleteEntryByThread`
  Delete a database entry, by removing it from the database.json file and deleting the associated JSON and XML files.
- `addEntrytoDatabase`
  Adds an entry to the database.json file. Called by `generateResponse` function after json file creation.
- `continueThread`
  Continues an existing thread by requesting updates via passed user instructions from the assistant. Then calls `updateJSON` to add updates to a JSON file, then converts them using `convertJsonToXML`.
- `generateResponse`
  Creates a new thread, and submits initial user instructions to the assistant. The response is then converted by using `convertJsonToXML` to an XML file, the frontend can use to display the diagram. Lastly, an entry to the database is submitted by using `addEntrytoDatabase`.
- `updateJSON`
  Adds new assistant updates to the existing JSON of the diagram.
- `convertJsonToXML`
  Converts JSON diagram info to a bpmn-js [11] compliant XML format.

6.3.3 Application

For the frontend, we have a single page application, made with Angular [12]. It is divided into multiple components, with some taken from the Angular Material library [13], but at the heart of it all is the *AppComponent* which makes use of all other components. The AppComponent manages communication between the user and the server, sending all HTTP Requests.

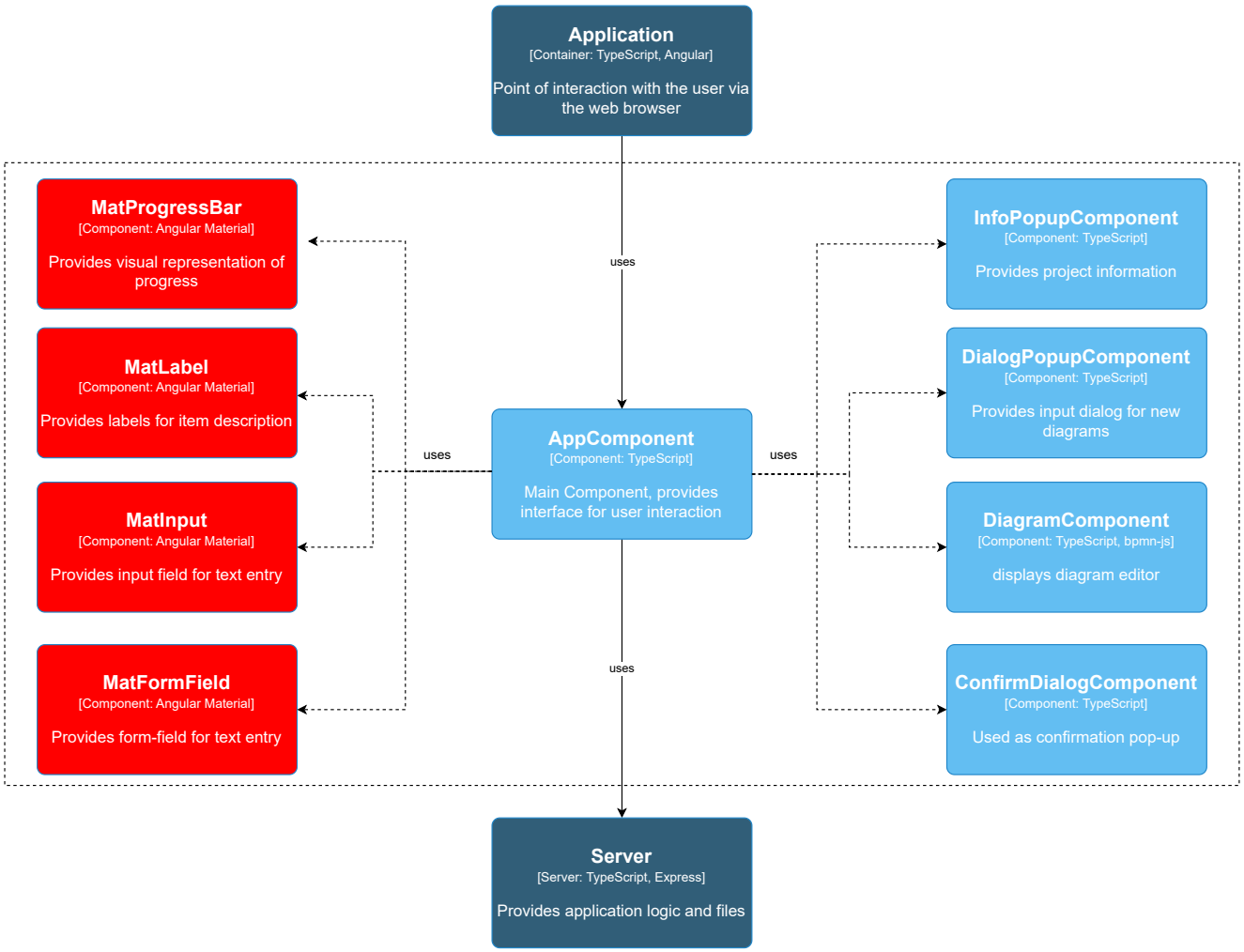

Figure 6.4: Frontend component

- Material Components
  The application takes advantage of the Materialize [13] library made by Google. Using this library, the user can experience a familiar environment, being familiar with some of the components already. It may not be an innovative design, but we are going for simplicity with this choice. From the library, we take advantage of input options, info labels and progress indicators.
- DiagramComponent
  This is the component that holds the bpmn-js [11] viewer, one of the most important components of the application. It displays the generated BPMN diagrams.
- Popup Components
  InfoPopup, DialogPopup and ConfirmDialog are all Components that are overlayed on the AppComponent. The DialogPopup being the most important one of the three, due to the fact that we enter our new diagram instructions in the component dialog. ConfirmDialog has been created to ensure that a user does not accidentally delete a diagram, making sure they confirm their action. InfoPopup is merely an info container, describing the application context.

As mentioned, the application should provide a familiar user experience. This is why the user interface has been designed to look and feel similar to most popularly used applications such as Google and ChatGPT. To see the implemented design, see Chapter 7. A well-designed user interface can make for an intuitive experience. We do not want to have to explain to the user how the application works, they should be able to use it from just their prior knowledge of websites or by suggestive component design, guiding the user to the available interaction options.

7 Implementation

7.1 Setting up OpenAI API

Setting up communication with the OpenAI API can be very simple. As with most API services, we need to set up our API key. Once that is done, we can create an OpenAI instance in our project.

```
const openai = new OpenAI({
  apiKey: sk-mySecretKey,
});
```

Then we are already set to communicate with ChatGPT via the API. If we wanted to write a function to ask a simple question such as *'What is the meaning of life?'* our code would look like this:

```
async function meaningOfLife() : Promise<string> {
  const input = 'What is the meaning of life?';
  const params: OpenAI.Chat.ChatCompletionCreateParams = {
    messages: [{
      role: 'user',
      content: input
    }],
    model: 'gpt-3.5-turbo',
    temperature: 0.0,
  };
  const chatCompletion: OpenAI.Chat.ChatCompletion = await openai.
      chat.completions.create(params);
  return chatCompletion.choices[0].message.content;
}
```

We create a ChatCompletion object, by defining the parameters. The parameters here are our messages, model and temperature. The messages and model are self-explanatory, the temperature defines how much answers can vary, if we were

to repeat the prompt a lot of times. In this case we chose a low temperature, to achieve a very similar, if not the same, result every time. We could define more parameters, such as a maximum token value, to restrict how much can be generated in the response or a top-p value which control the trade-off between creativity and coherence within the response generation. But for this question, we allow them to remain on default by not adding them.
Once we create the ChatComplete object, the run starts and the chat response is generated. Of course this is not the only way, to communicate with the OpenAI API, since we have access to custom assistants. How our custom assistant works and generates its response has been explained in Chapter 5.

7.1.1 Working with custom Assistants

Working with an assistant is very similar to using the ChatCompletion Object. First off, we need to fetch our assistant, we created earlier in the web interface. We could also create this in code, but since we've already created our assistant, we can just fetch it using its ID.

```
1 async function retrieveAssistant() {
2     const assistant = await openai.beta.assistants.retrieve("
          asst_0e2ISidSb3mPC8qDzjRlfKFD");
3     return assistant;
4 }
```

Once we have our assistant, we can get started on creating our thread for interacting with it. In this project it is important that we communicate on new threads every time, since as discussed in Chapter 5, the assistant behaves differently once it receives a second message within a thread, sending back JSON update instructions.

```
1 async function createThread(scenario: string): Promise<string>{
2   const thread = await openai.beta.threads.create();
3   return thread.id;
4 }
```

We return our thread ID here, to store it in the database later, so we can access the thread again. Now that we have our thread, we can begin a conversation with our assistant.

```
1 async function createRun(thread_id: string, assistant: any,
     scenario: string, update: boolean): Promise<boolean> {
```

```
2   // create message for from user input
3   await openai.beta.threads.messages.create(
4     thread_id,
5     {
6       role: "user",
7       content: scenario
8     }
9   );
10
11  ...
```

This is very similar to what we have done earlier, we are adding messages to our thread for the assistant to generate answers for. We may add as many messages as we want, but in this use case the one initial message suffices. Now there is a key difference here, the assistant only starts to generate answers, once we begin a so called, run.

```
1   ...
2   const run = await openai.beta.threads.runs.createAndPoll(
3     thread_id,
4     {
5       assistant_id: assistant.id
6     }
7   );
8   ...
```

We create the run with the key value to using the assistant, our assistant ID. Without this assignment, we would just get a plain chat response to our instructions, which would generate a result of little to no value to us. We also use the function called *createAndPoll()*, which polls for updates on the status of our run. This way we can, once our run finished or returned an error, correctly assess what to do next. The run status can return these values:

- **queued**, when the run is in the queue
- **in_progress**, when the run is currently in progress
- **requires_action**, the run needs confirmation to continue
- **cancelling**, when the run is in the progress of cancelling
- **cancelled**, when the run is cancelled
- **failed**, when the run fails

- **completed**, when our run is finished
- **expired**, when our run is no longer available

We seek for run completion to continue with our process. Then we take the last message we received, which contains the generated answer of the assistant, and save it in our database for future use.

```
if (run.status === 'completed') {
    const messages = await openai.beta.threads.messages.list(
      run.thread_id
    );

    // get json message
    let lastMessage = messages.data[0].content[0].type === 'text'
       ? messages.data[0].content[0].text.value : 'Error';
    console.log(lastMessage);
    // check if generation failed (specified in input prompt for
       the GPT)
    if(lastMessage === 'Error') return false;

    // parse json from message
    // if there is a prose text this removes it
    let json = findAndParseJSON(lastMessage);
    // if there is no JSON return false
    if(json === null) return false;

    ...

    const filePath = path.resolve(__dirname,'..', 'data', 'json',
       `${thread_id}.json`);
    // write parsed json to file
    await fs.promises.writeFile(filePath, JSON.stringify(json,
       null, 2), (err: any) => {
     if (err) console.error('Error writing file:', err);
    });

    console.log('File written');

    // add entry to database
    await addEntryToDatabase(await createTitle(scenario), scenario
       , thread_id);
```

```
30
31      return true;
```

A certain key points here is the *findAndParseJSON()* function. This function removes any prose text and Markdown syntax, in case our assistant disregarded its instructions. In the end, a database entry is created, saving the given scenario prompt (the user instructions) and the thread ID. Finally, we can begin the process of converting the received JSON file to the desired XML format the frontend can view and manipulate.

7.2 Database

Before we look into how we convert the JSON files, we need to take a brief look into how the database has been realized. For this project, the chosen type of database is a *file-based database*, meaning we store our vital information in a single JSON file and reference the generated XML files from there. This means we do not have a Database Management System (DBMS), so our application reads and writes directly to the files. But the simple and straightforward implementation makes a file-based database very attractive for this project. To realize quick data access all data about our diagrams, but the diagram itself, the attributes of all diagrams are saved in one singular file. The entries look like this:

```
1   {
2   "thread": "thread_xMh4Sizb1YyPCUUVXfMSc9YH",
3   "title": "Music: A Shared Experience",
4   "scenario": "I want to listen to music. I usually listen alone
        , but this time I ask a friend if he wants to listen to
       music as well. If they accept, they come over and we set up
        the sound system. If they don't want to listen to music I
       just use my headphones.",
5   "time": "2024-07-22T12:42:34.903Z",
6   "path": "data/xml/thread_xMh4Sizb1YyPCUUVXfMSc9YH.xml"
7  }
```

Given the path, we can easily access to a file via an HTTP request. If we were to edit our entries, it is very straight forward, we just need to filter for the thread ID, then after parsing we can remove the entry and replace it with an updated one in its place. When creating a new entry, we just append to the end of the file. While the concept may seem very simple, it is quite effective and sufficient for our use-case.

7.3 Converting JSON to XML

Now we can take a brief look into how the conversion of the JSON data into the BPMN XML format is implemented. For the XML creation, we import xmlbuilder2. The conversion process consists of four steps:

- Reading and parsing the JSON file
- Initializing the XML root element with BPMN-specific namespaces
  Here we define the BPMN model, diagram interchange and schema instance.

```
1   const root = create({ version: '1.0', encoding: 'UTF-8'
        })
2       .ele('definitions', {
3           'xmlns': 'http://www.omg.org/spec/BPMN/20100524/
              MODEL',
4           'xmlns:bpmndi': 'http://www.omg.org/spec/BPMN
              /20100524/DI',
5           'xmlns:omgdi': 'http://www.omg.org/spec/DD
              /20100524/DI',
6           'xmlns:omgdc': 'http://www.omg.org/spec/DD
              /20100524/DC',
7           'xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-
              instance',
8           'id': 'sid-38422fae-e03e-43a3-bef4-bd33b32041b2',
9           'targetNamespace': 'http://bpmn.io/bpmn',
10          'exporter': 'bpmn-js (https://demo.bpmn.io)',
11          'exporterVersion': '17.6.2'
12      });
```

- Iterating through the pools and lanes from the JSON, to create elements such as *collaboration*, *process*, and *lane*, and all their respective components.
  The collaboration represents interaction between multiple participants (pools).

```
1       const processID = generateRandomId();
2       collaboration.ele('participant', {
3           'id': pool.ID,
4           'name': pool.Name,
5           'processRef': 'Process_' + processID
6       });
7       const process = root.ele('process', {
8           'id': 'Process_' + processID,
```

```
            'isExecutable': 'false'
        });
```

The process contains lanes which then contain the flow nodes and references, such as f. e. tasks.

```
function processLaneComponents(lane: any, root: any, process:
    any, laneElement?: any): void {
    // Process each component in the lane
    lane.Components.forEach((component: { Type: string; ID:
       any; Name: any; Incoming: string[]; Outgoing: string
       []; x: any; y: any; }) => {

        const taskType = toCamelCase(component.Type)
        const task = process.ele(taskType, {
            'id': component.ID,
            'name': component.Name
        });
        // Add incoming flows
        component.Incoming.forEach((incomingId: string) => {
            task.ele('incoming').txt(incomingId);
        });

        // Add outgoing flows
        component.Outgoing.forEach((outgoingId: string) => {
            task.ele('outgoing').txt(outgoingId);
        });

        // Add flowNodeRef to lane
        if(laneElement) laneElement.ele('flowNodeRef').txt(
           component.ID);
    });
}
```

```
function processLaneFlows(lane: any, root: any, process: any)
   : void {
    // Process each flow in the lane
    lane.Flows.forEach((flow: { ID: any; Descriptor: any;
       Start: any; Target: any; Type: any; StartXY:any;
       TargetXY: any }) => {
        if(flow.Descriptor === undefined) flow.Descriptor = '
           ';
```

```
5           const item = process.ele(toCamelCase(flow.Type), {
6               'id': flow.ID,
7               'name': flow.Descriptor,
8               'sourceRef': flow.Start,
9               'targetRef': flow.Target
10          });
11      });
12  }
```

- Finally, adding visual information to the diagram based on size, shape or position of the components. Here we need to consider a lot of cases, since now we are placing size and position of the components, and the position of the flows.
  For each component, we need to assign height and width depending on the type of component. For each flow, we need to consider a bit more to correctly place them. If there is a change in y-values between components, we may want to set the beginning and end of our flow a bit differently to make our diagram look better. For a complete understanding, including the detailed handling of each component and flow, please refer to the full source code.

7.4 Updating XML files

Updating the XML with our generated diagram instructions is also handled by the *createRun()* function in *useGPT.ts*.

```
1       if(update) { // case update
2        // update the json file
3        updateJSON(thread_id, json);
4        // update time stamp and add new instructions
5        let data = fs.readFileSync('data/database.json', 'utf8');
6        let jsonData = JSON.parse(data);
7        jsonData.forEach((element: { thread: any; time: string;
             scenario: string}) => {
8          if (element.thread === thread_id) {
9            // update time
10           element.time = new Date().toISOString();
11           // update instructions
12           element.scenario = element.scenario + '\n(NEXT)' +
                 scenario;
13         }
```

```
14        });
15        fs.writeFileSync('data/database.json', JSON.stringify(
              jsonData, null, 2), 'utf-8');
16        return true;
17      }
```

If we recall the design of the assistant, the assistant returns an update message once we send a second message in a thread. For this update message, we need to use a different function to add the changes to the XML file. Here we use the function *updateJSON()*. Please refer to the source code, for more information. To give a brief overview of what the function does: In our update message we receive four distinct instruction possibilities, with different ways to handle them. We need to respect these different instructions:

- *CREATE*: Adds new items (Pools, Lanes, Flows or Components) to the diagram.
- *ALTER*: Modifies properties of existing items based on provided changes.
- *DELETE*: Removes an item (Pool, Lane, Flow or Component) from the existing structure. In case of deleting a parent item, all children will be removed as well.
- *UPDATEPARENT*: Changes the parent of an item, effectively moving it to a different part of the structure.

All these changes are pushed to the JSON file, containing all the original instructions. Then we can just convert the JSON file to the desired XML format. This way we can save tokens and especially time, generating the entire JSON file with the changes, would take much longer than just processing the updates ourselves.
Finally, we update our database, by updating the timestamp of the diagram and adding the latest user instruction to the instruction list.

7.5 Frontend Implementation

For the frontend, the chosen framework is Angular [12]. Angular is ideal for creating dynamic and robust single-page applications. Its also very easy to maintain and scale, which is ideal for future development. Another strong point for Angular is its performance, we want our application to use as minimal resources as possible to ensure a smooth bpmn-js viewer. And lastly, the strong Typescript [14] support,

which makes development a lot easier. Let us take a look how the application turned out.

7.5.1 UI Design

For the design aspect of the frontend, the primary aim was to develop an interface that the user already feels familiar with. For this Materialize [13] seemed to be the perfect choice, since it provides high user familiarity, being developed by Google, and also a simple style design, most people are happy seeing.

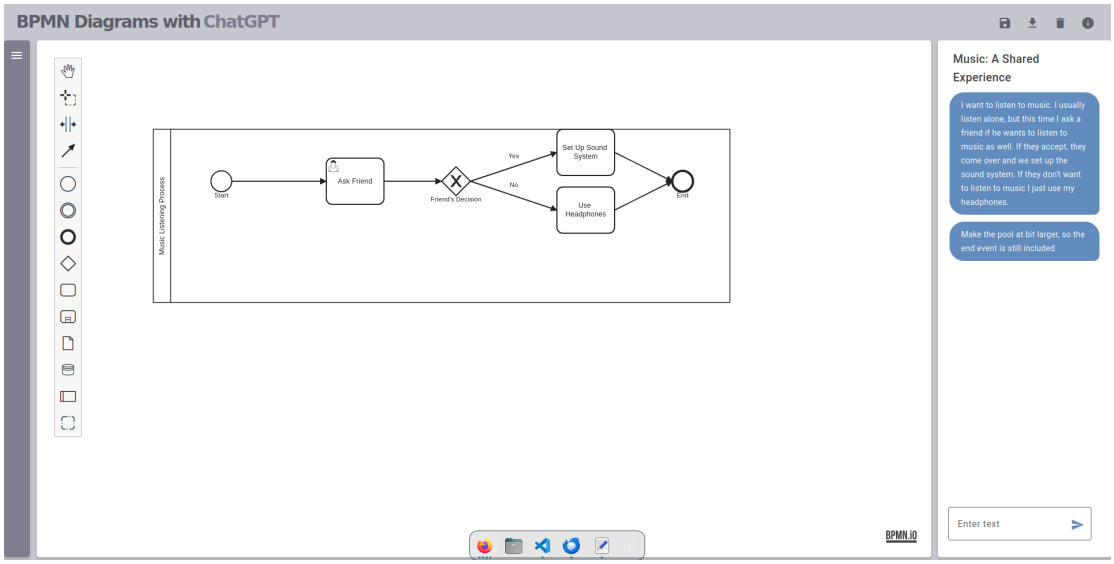

Figure 7.1: Overview of the UI

For the final design, we can have three distinct spaces to interact with. On the left we have our table of available, earlier created, diagrams and an option to create a new diagram. This part is collabsable to offer more editing room. The creation of the new diagram is handled in a popup window.

In the center of the window is the heart of the frontend application, the bpmn-js viewer. Style choices of the viewer fit in well with the Materialize components.
And finally, on the right, we have the chat part of the frontend. Here we have a chat window display to display prior instructions by the user, below we have the text field to enter new instructions. The design of the chat window was done to still give the user the impression of chatting with ChatGPT, although it is only answering with

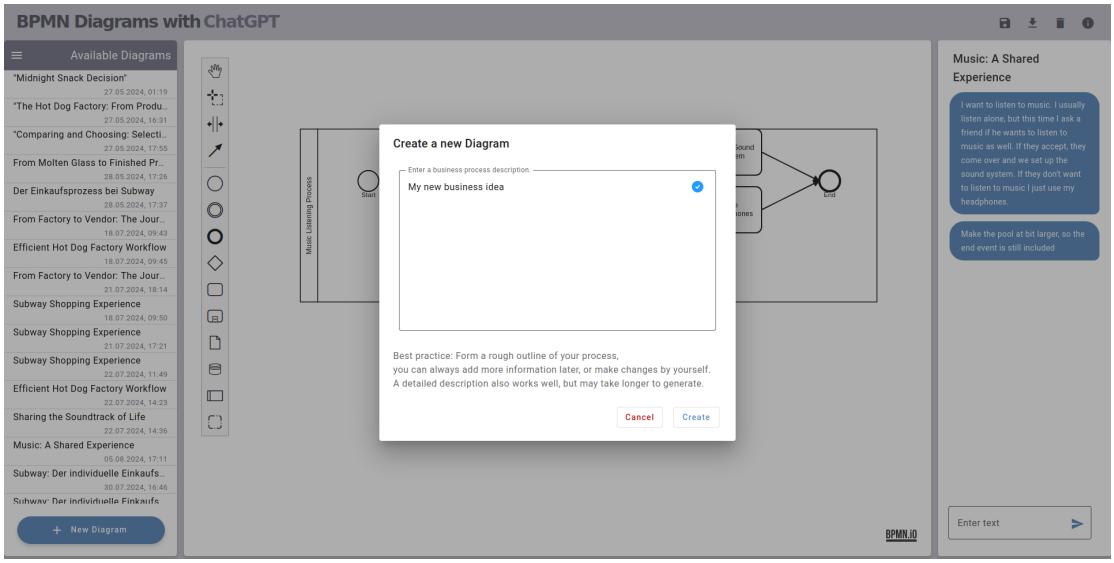

Figure 7.2: Create new diagram pop-up

diagrams here.
Lastly we need to direct our eyes up, to the topbar, where we have options to save, export or delete the current diagram. Also, a small info section is included with a popup, including an option to clear all diagrams for a clean reset of the application.

7.5.2 bpmn-js

To use bpmn-js [11] a component has been created, where the bpmn-js.modeler is attatched to a div. Once the div is added to our app component, we attach a diagram, which we get with a URL, to the modeler.

```
1  private bpmnJS: BpmnJS = new BpmnJS();
2
3  constructor(private http: HttpClient) {
4    this.bpmnJS.on<ImportDoneEvent>('import.done', ({ error }) =>
       {
5      if (!error) {
6        this.bpmnJS.get<Canvas>('canvas').zoom('fit-viewport');
7      }
8    });
9  }
10
11 ngAfterContentInit(): void {
12   this.bpmnJS.attachTo(this.el.nativeElement);
```

```
13   }
14
15   ngOnInit(): void {
16     if (this.url) {
17       this.loadUrl(this.url);
18     }
19   }
20
21   ngOnChanges(changes: SimpleChanges) {
22     // re-import whenever the url changes
23     if (changes.url) {
24       this.loadUrl(changes.url.currentValue);
25     }
26   }
27
28   ngOnDestroy(): void {
29     this.bpmnJS.destroy();
30   }
```

7.5.3 Interaction

Interaction and updating the files is done via API Calls to the server. Let us go through a typical interaction process of a user to go gain a bit of insight into the source code.
When the application is launched, we need to fetch the list of diagrams from the server. In Angular, this can be achieved by using the constructor.

```
1   constructor(private http: HttpClient, public dialog: MatDialog)
        {
2     // set file list
3     this.http.get('http://localhost:1207/database').subscribe((
          data: any[]) => {
4       this.files = data;
5     });
6   }
```

Once we have read and assigned the database, the user can interact with the table containing the diagram list. This is realized in HTML using the *(click)* functionality.

```
1     ...
2     <div class="flex-container">
3       <mat-list class="scrollable-list">
4         <div *ngFor="let file of files">
```

```
5           <mat-list-item class="spaced-list-item" matTooltip="{{
                file.title}}" matTooltipShowDelay="500"
                matTooltipPosition="above" (click)="onFileClick(file
                )">
6             {{ file.title }}
7             <p class="file-time">{{file.time | date:'dd.MM.y, HH:
                mm':'CET'}}</p>
8           </mat-list-item>
9           <mat-divider></mat-divider>
10        </div>
11      </mat-list>
12    </div>
13    <mat-list-item (click)="onFileClick('new')">
14      <div class="icon-text">
15        <mat-icon>add</mat-icon>
16        <span>New Diagram</span>
17      </div>
18    </mat-list-item>
19    ...
```

A click on a file calls the *onFileClick()* function.

```
1   onFileClick(file: any) {
2     // create new diagram
3     if (file === 'new') {
4       const dialogRef = this.dialog.open(DialogPopupComponent);
5       dialogRef.afterClosed().subscribe(() => {
6         this.refreshFileList().then(() =>{
7           const latestDiagram = this.files[this.files.length - 1];
8           // assign all values of the diagram to the component
9           this.diagramUrl = `http://localhost:1207/${latestDiagram
                .path}`;
10          this.instructions = this.processInstructions(
                latestDiagram.scenario);
11          this.diagramTitle = latestDiagram.title;
12          this.diagramID = latestDiagram.thread;
13        });
14      });
15      return;
16    } else {
17      // assign all values of the diagram to the component
18      this.diagramUrl = `http://localhost:1207/${file.path}`;
19      this.instructions = this.processInstructions(file.scenario);
20      this.diagramTitle = file.title;
21      this.diagramID = file.thread;
22    }
```

```
23      this.error = false;
24    }
```

In case the user clicked on the new diagram button, we need to interact with a different component. Here we open a dialog popup to enter our instructions. Once it closes with user confirmation, we refresh the file list and assign all necessary attributes to display the diagram. Displaying the diagram, for example, is done by setting the URL. If the user just chose one of the existing diagrams, we can just assign the values of the selection to our component to display.
The user has then the option to manipulate the diagram by hand in the bpmn-js viewer, or to write a new update prompt for the diagram.

```
1  sendUpdate():void {
2      this.isLoading = true;
3      this.error = false;
4      this.http.post('http://localhost:1207/update', { id: this.
           diagramID , inputString: this.diagramUpdates }).subscribe(
5              data => {
6                // handle the loaded data
7                this.refreshFileList();
8                this.instructions.push(this.diagramUpdates);
9                this.diagramUpdates = '';
10
11               // refresh the diagram by appending a timestamp
                     query parameter
12               const timestamp = new Date().getTime();
13               this.diagramUrl = `${this.diagramUrl.split('?')[0]}?
                     timestamp=${timestamp}`;
14
15               this.isLoading = false;
16             },
17             error => {
18               this.isLoading = false;
19               this.error = true;
20             }
21         );
22    }
```

Similarly to before, we refresh the file list, but then we need to use some tricks, since we already assigned the values of the current diagram to the component. So we push our latest text instructions to our instructions content and update the diagram using a trick to add a timestamp to reload without changing the URL. This process can be repeated as many times as the user wants. Of course, the user can also download their created diagram using the button on the topbar. Here we take

advantage of a function of the bpmn-js diagram component, to get the current state of the diagram and exporting it to download.

```
1  exportDiagram() {
2    if (this.diagramComponent) {
3      this.diagramComponent.saveDiagram().then(({ xml, err }) => {
4        if (err) {
5          console.error('Could not export diagram', err);
6        } else {
7          const blob = new Blob([xml], { type: 'application/bpmn+
               xml' });
8          const url = window.URL.createObjectURL(blob);
9          const link = document.createElement('a');
10         link.href = url;
11         link.download = this.diagramTitle || 'diagram.bpmn';
12         document.body.appendChild(link);
13         link.click();
14         document.body.removeChild(link);
15         window.URL.revokeObjectURL(url);
16       }
17     });
18   } else {
19     console.error('No diagram to export');
20   }
21 }
```

8 Results

In this chapter, we take a look at the generated results. To reiterate, our rule of thumb: *The generated results are only as good as the given prompt*. If the user does not clearly state what they want the diagram to look like, the likelihood of unsatisfactory results increase substantially. In total, over 50 diagrams have been generated with this application to test the capabilities of the system. In this chapter, we will look at only five different diagrams and their prompts, to illustrate the results. We will evaluate the results for their accuracy, meaning what we expect to be in the diagram and how well the diagram is formatted, and the performance, meaning how much time was spent generating.

8.1 Examples of generated Diagrams

8.1.1 Example 1: Employee Onboarding

When a new employee accepts a job offer, the HR department sends them an onboarding welcome email. The IT department sets up the employee's workstation and provides the necessary equipment. The HR department then schedules an orientation session. If there are scheduling conflicts, the orientation session is rescheduled. The new employee attends the orientation session, gets introduced to their team, and is given a tour of the office. The employee then completes necessary paperwork and HR formalities, completing the onboarding process and starting their duties.

Add a start and end event to the IT department.

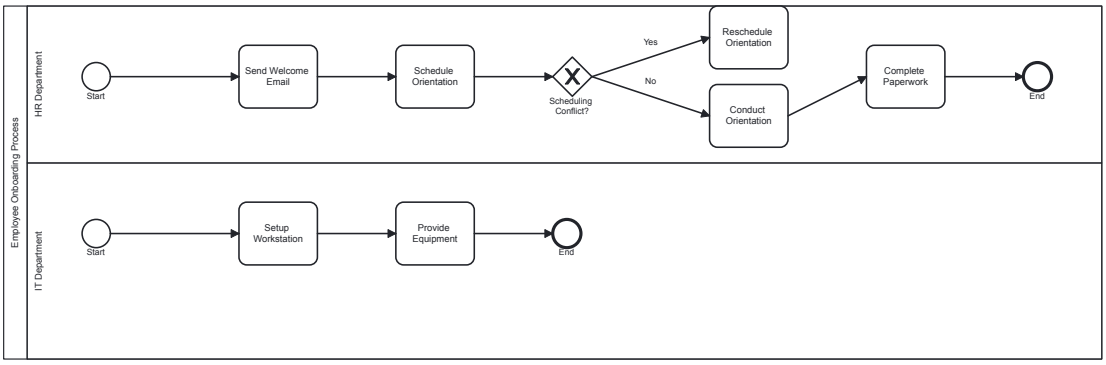

Figure 8.1: Employee Onboarding, Generation Example 1

The assistant assigned two lanes for this process. These lanes show the process in each Department, and do not have any interaction with one another. Every described step seems to be included, aside from the office tour. One minor issue: The "Yes" branch from the "Scheduling Conflict?" gateway does not have an end event. Initially a start and end event in the IT Department was missing, that was fixed by one update message. Performance wise, generation for the initial diagram took 16 seconds and for the update the generation time was 6 seconds.

8.1.2 Example 2: Hot Dog Factory

There is a hot dog factory. We have two production lanes. One produces the buns, the other the hot dogs itself. Both productions need to talk to quality control, which checks if the items are up to company standard, if they are not, they need to be reproduced. If they are up to company standard, they are sent to a vendor. The vendor sets up their shop, makes the hot dogs at his stand, and sells them to hungry customers.

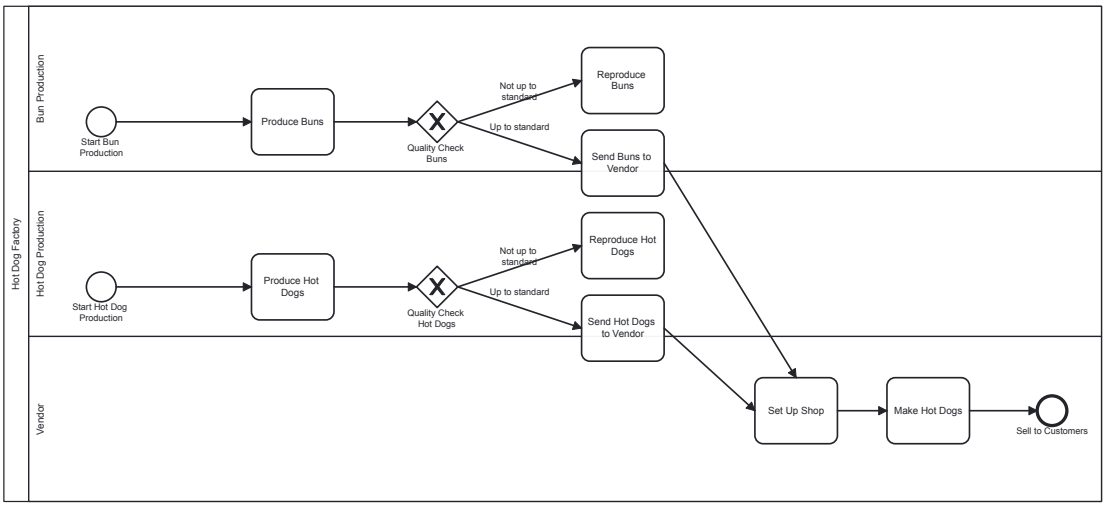

Figure 8.2: Hot Dog Factory, Generation Example 2

There is obvious overlapping of components on the lane lines. Additionally, there are again branches that do not have an end event. This can be addressed by manually adding the necessary changes or writing an update prompt. Aside from the issues, the assistant as expected set up three lanes to illustrate the process. The quality check step is illustrated as a gateway. Performance wise, generation for the diagram took 22 seconds.

8.1.3 Example 3: IT Support Ticket Handling

> An employee submits an IT support ticket, which the system assigns to an available IT support technician. The technician reviews the ticket details and determines if the issue can be resolved remotely. If the issue can be resolved remotely, the technician resolves it, notifies the employee of the resolution, and closes the ticket. If the issue requires on-site assistance, the technician schedules an on-site visit, resolves the issue at the employee's workstation, notifies the employee of the resolution, and closes the ticket.

This example requires manual changes in the bpmn-js editor, since the start of the process initially was a task, which is not correct. Changing that to a start event and

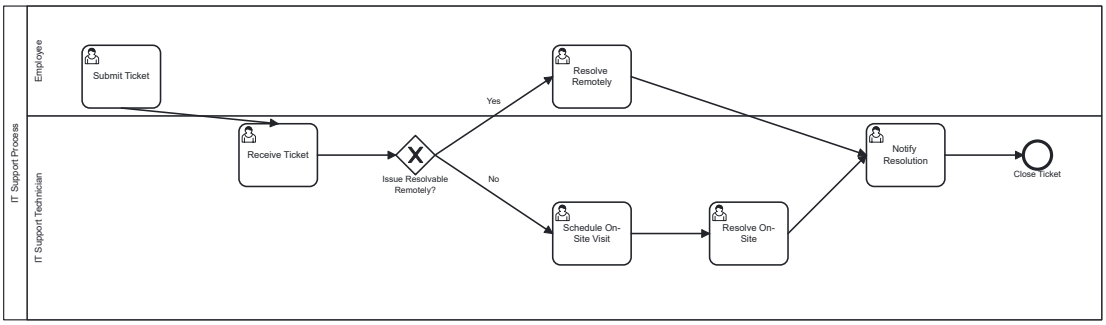

Figure 8.3: IT Support Ticket Handling, Generation Example 3

adding the flow can be done quickly using the editor. Again, two lanes are used to illustrate the different participants, with one gateway branching off to the employee side. We could argue that the diagram is a bit ambiguous, with the "Resolve Remotely" Task. Overall, the result is still satisfactory. Performance wise, generation for the diagram took 18 seconds.

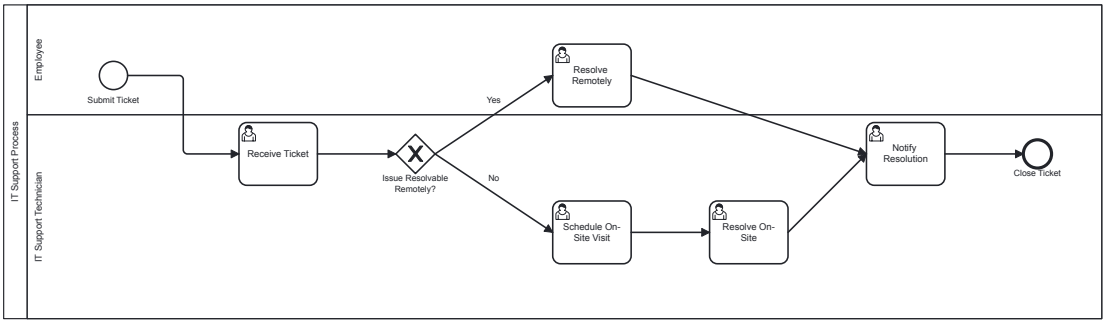

Figure 8.4: IT Support Ticket Handling, After manual changes

8.1.4 Example 4: Snack Decision

> I'm hungry and want to eat. I go to the kitchen and check the fridge. There is fish and meat. I chose one and eat. Then I leave.

This is quite the simple prompt, the assistant decided to use a gateway to illustrate the decision part of the process. The text prompt is not as verbose, but the assistant successfully builds a syntactically correct diagram. The generation of the diagram took 9 seconds.

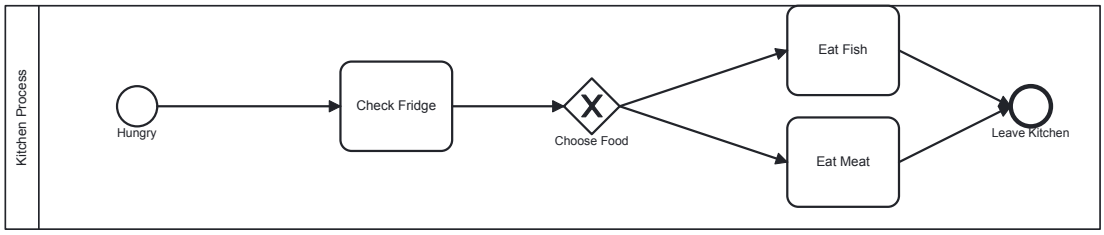

Figure 8.5: Snack Decision, Generation Example 4

8.1.5 Example 5: The Bakery Dilemma

I am waiting at a bus station for the bus. The bus is supposed to arrive in 5 minutes, so I am debating about going to a bakery. If I go to the bakery and there is a long queue, I could take longer than 5 minutes and I would miss my bus. But if there is no queue, I will be on time for the bus.

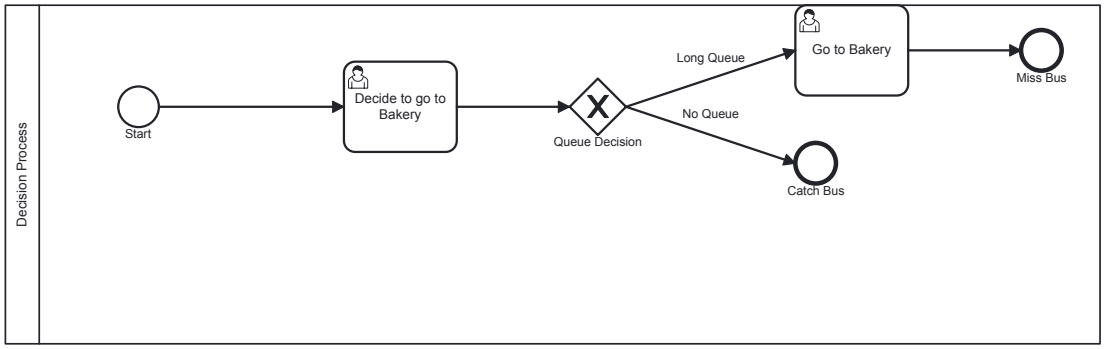

Figure 8.6: The Bakery Dilemma, Generation Example 5

We could argue that there is a branch missing here, the decision of not going to the bakery. But the instruction text is not entirely clear about that, so it is understandable that the assistant disregarded it. Again, overall the result is good, covering all important aspects of the instruction text. Generating took 11 seconds.

8.2 Key findings

After generating the diagrams, we can note that the results are satisfactory. The system successfully identifies the elements in the text and assigns symbols to them, a clear flow is visible, even if the instruction text is ambiguous. Resulting generated BPMN diagrams often only need minor adjustments for them to be syntactically correct. The value of generated results correlates with the quality of the instruction text. Pairing the system with an expert in the field, that can express their desires in a clear instruction text without ambiguity, who can also make the necessary changes to the imperfections of the generated diagrams, will yield optimal results.

9 Discussion

To conclude this thesis, we take a look at the system requirements again, evaluate some system limitations and discuss the potential of further development for this project in the future.

9.1 Reevaluate System Requirements

The successful completion of this project involved meeting all specified system requirements, ensuring the robustness and validity of the work. In Chapter 3 we defined these system requirements.

- Functional Requirements ✓
  - User Interaction ✓
  - Backend Processing ✓
- Non-function requirements ✓

We successfully met all the listed requirements.

9.2 Importance of GPT Version

During the development process of this project, the GPT version played a key role in result quality. Initially, the project was running on GPT-3.5, with hopes in quicker generation times. Unfortunately, the results are subpar, often times not even returning a valid JSON. There simply were too many inconsistencies, a switch to GPT-4 was necessary, which improved the project a lot. The results were acceptable, adhering to the syntax and producing simple diagrams. In May 2024 OpenAI released GPT-4o, which the assistant is currently running on. This enables the best results, with much more potential to improve in the future.

9.3 Limitations

While achieving most system requirements, the project unfortunately is not without limitations. One jarring problem that might deter users as of right now is the long generation time. The long generation time is to be expected with ChatGPT only being in its infancy, so in the future will this issue likely be resolved, but it is one thing users could complain about. Additionally, updating the generated diagram with an update text prompt is a bit irrational at times. It works very well when creating or deleting items, but it does struggle with moving items. Generally speaking, the generated diagrams are not as appealing as hand made diagrams. That is due to generation errors and the limitation of the flows only having two waypoints as of right now, once multiple waypoints can be included, the generated diagrams will improve in design quality.

9.4 Future Work

While the system works well as it is as of right now, there is still great room for potential. There still is the possibility of training the assistant further, to improve accuracy and performance. We have not yet tapped into the potential of feeding the assistant big data of multiple diagrams, from which the assistant could learn. Additionally, in the future ChatGPT might be more capable to generate XML directly, we could eliminate the conversion step there, with the possibility of better styling, since ChatGPT would need to be more aware of how objects are placed in a 2D plane to achieve that. Specifically for this project, a version control might be very beneficial for the user experience, with users being able to revert to previous versions of their diagrams. If we were to scale up the project, we might want to add a user system to support multiple users. A change of database type would need to be in order for that to be achieved.

9.5 Conclusion

We have successfully built a project where users can comfortably generate BPMN diagrams using just a plain text instruction prompt. We were able to achieve diagram generation with ChatGPT, but we had to navigate some obstacles in order to do so. ChatGPT is capable of a lot, but needs a lot of instructions and data in order to

achieve our desired results. The results heavily depend on the quality of the text instructions a user gives. Hence, the tool can produce the best results if paired with an expert in the field. With high interest from research and the industry, BPMN generation with LLMs will very likely rapidly improve over the next years.

A Sourcecode

```
1  import OpenAI from 'openai';
2  import dotenv from "dotenv";
3  import path from 'path';
4  import { convertJsonToXml } from "./jsonToXML";
5  import { updateJSON } from './updateJSON';
6  import { addEntryToDatabase } from './manageData';
7
8  const fs = require('fs');
9
10 dotenv.config(); // create config with API key
11
12 // new instance of OpenAI
13 const openai = new OpenAI({
14   apiKey: process.env.API_KEY,
15 });
16
17 /**
18  * Creates a title for a given prompt
19  * @param prompt to create a title for
20  * @returns title
21  */
22 async function createTitle(prompt: string) : Promise<string> {
23   const input = '[no prose] [only return title] Find a fitting
       title for this scenario:' + prompt;
24   const params: OpenAI.Chat.ChatCompletionCreateParams = {
25     messages: [{
26       role: 'user',
27       content: input
28     }],
29     model: 'gpt-3.5-turbo',
30     temperature: 0.0,
31   };
```

```
32    const chatCompletion: OpenAI.Chat.ChatCompletion = await openai.
          chat.completions.create(params);
33    console.log(chatCompletion.choices[0].message.content);
34
35    let title = chatCompletion.choices[0].message.content;
36    if(title !== null) title = title.replace(/"/g, '');"
37    else title = 'Untitled';
38    return title;
39  }
40
41  /**
42   * Retrieves the custom assistant from OpenAI
43   * @returns assistant object
44   */
45  async function retrieveAssistant() {
46    const assistant = await openai.beta.assistants.retrieve("
          asst_0e2ISidSb3mPC8qDzjRlfKFD");
47    return assistant;
48  }
49
50  /**
51   * Creates a new chat thread
52   * According to https://platform.openai.com/docs/models/how-we-use
        -your-data threads are kept alive for 60 days
53   * @returns thread id
54   */
55  async function createThread(scenario: string): Promise<string>{
56    const thread = await openai.beta.threads.create();
57    return thread.id;
58  }
59
60  /**
61   * Parses JSON from a given string
62   * @param inputString the string containing JSON
63   * @returns JSON object
64   */
65  function findAndParseJSON(input: string): any {
66    try {
67      // match the outermost curly braces
68      const jsonMatch = input.match(/{[\s\S]*}/);
```

```
69      if (!jsonMatch) {
70        return null;
71      }
72      const jsonString = jsonMatch[0];
73      const jsonData = JSON.parse(jsonString);
74      return jsonData;
75    } catch (error) {
76      return null;
77    }
78  }
79
80  /**
81   * Creates a run instance for the assistant to run
82   * @param thread_id id of the thread
83   * @param assistant id of the assistant
84   * @param scenario user description of the BPMN diagram
85   * @param update false results in a brand new diagram, true
       results in an updated already existing diagram
86   * @returns true if successful, false if error
87   */
88  async function createRun(thread_id: string, assistant: any,
       scenario: string, update: boolean): Promise<boolean> {
89    // create message for from user input
90    await openai.beta.threads.messages.create(
91      thread_id,
92      {
93        role: "user",
94        content: scenario
95      }
96    );
97    const run = await openai.beta.threads.runs.createAndPoll(
98      thread_id,
99      {
100       assistant_id: assistant.id
101     }
102   );
103   if (run.status === 'completed') {
104     const messages = await openai.beta.threads.messages.list(
105       run.thread_id
106     );
```

```
107
108     // get json message
109     let lastMessage = messages.data[0].content[0].type === 'text'
            ? messages.data[0].content[0].text.value : 'Error';
110     console.log(lastMessage);
111     // check if generation failed (specified in input prompt for
            the GPT)
112     if(lastMessage === 'Error') return false;
113
114     // parse json from message
115     // if there is a prose text this removes it
116     let json = findAndParseJSON(lastMessage);
117     // if there is no JSON return false
118     if(json === null) return false;
119
120     if(update) { // case update
121       // update the json file
122       updateJSON(thread_id, json);
123       // update time stamp and add new instructions
124       let data = fs.readFileSync('data/database.json', 'utf8');
125       let jsonData = JSON.parse(data);
126       jsonData.forEach((element: { thread: any; time: string;
              scenario: string}) => {
127         if (element.thread === thread_id) {
128           // update time
129           element.time = new Date().toISOString();
130           // update instructions
131           element.scenario = element.scenario + '\n(NEXT)' +
                  scenario;
132         }
133       });
134       fs.writeFileSync('data/database.json', JSON.stringify(
              jsonData, null, 2), 'utf-8');
135       return true;
136
137     } else { // CASE: Brand new diagram
138
139       const filePath = path.resolve(__dirname,'..', 'data', 'json'
              , `${thread_id}.json`);
140       // write parsed json to file
```

```
141       await fs.promises.writeFile(filePath, JSON.stringify(json,
              null, 2), (err: any) => {
142         if (err) console.error('Error writing file:', err);
143       });
144
145       console.log('File written');
146
147       // add entry to database
148       await addEntryToDatabase(await createTitle(scenario),
              scenario, thread_id);
149
150       return true;
151     }
152   } else {
153     console.log(run.status);
154   }
155   return false;
156 }
157
158 /**
159  * Use to create a new thread and generate a response from the
        assistant
160  * @param scenario user description of the BPMN diagram
161  * @returns ideally a new JSON file containing the BPMN diagram
        instructions
162  */
163 export async function generateResponse(scenario: string) {
164   let result;
165   console.log(scenario);
166   const thread_id = await createThread(scenario);
167   console.log(thread_id);
168
169   if(!await createRun(thread_id, await retrieveAssistant(),
        scenario, false)) return "Error";
170   try {
171     result = convertJsonToXml(thread_id);
172   } catch (err) {
173     console.error(err);
174   }
175   return result;
```

```
176 }
177
178 /**
179  * Use to continue a thread and generate a response from the
        assistant
180  * @param scenario user description of the BPMN diagram
181  * @param thread_id id of the thread to continue
182  * @returns ideally a new JSON file containing the BPMN diagram
        instructions
183  */
184 export async function continueThread(scenario: string, thread_id:
       string) {
185   let result;
186   if (!await createRun(thread_id, await retrieveAssistant(),
        scenario, true)){
187     console.log("run error");
188     return "Error";
189   }
190   try {
191     console.log("convert");
192     result = convertJsonToXml(thread_id);
193   } catch (err) {
194     console.error(err);
195   }
196   return result;
197 }
```

Listing A.1: useGPT.ts

```
1  import { create } from 'xmlbuilder2';
2  import fs from 'fs';
3  import path from 'path';
4  import { XMLBuilder } from 'xmlbuilder2/lib/interfaces';
5
6  /**
7   * Converts the GPT generated JSON string into a bpmn-js readable
        format
8   * @param jsonInput path to the JSON file
9   * @returns BPMN XML string
10  */
11 export function convertJsonToXml(jsonInput: string): string {
```

```
12      const filePath = path.join(__dirname, '..', 'data', 'json', `${jsonInput}.json`);
13      const input = fs.readFileSync(filePath, 'utf-8');
14      const jsonObj = JSON.parse(input);
15
16      const root = create({ version: '1.0', encoding: 'UTF-8' })
17          .ele('definitions', {
18              'xmlns': 'http://www.omg.org/spec/BPMN/20100524/MODEL',
19              'xmlns:bpmndi': 'http://www.omg.org/spec/BPMN/20100524/DI',
20              'xmlns:omgdi': 'http://www.omg.org/spec/DD/20100524/DI',
21              'xmlns:omgdc': 'http://www.omg.org/spec/DD/20100524/DC',
22              'xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance',
23              'id': 'sid-38422fae-e03e-43a3-bef4-bd33b32041b2',
24              'targetNamespace': 'http://bpmn.io/bpmn',
25              'exporter': 'bpmn-js (https://demo.bpmn.io)',
26              'exporterVersion': '17.6.2'
27          });
28      const collaborationID = generateRandomId();
29      const collaboration = root.ele('collaboration', { 'id': 'Collaboration_' + collaborationID });
30      // create processes
31      jsonObj.Pools.forEach((pool: any) => {
32          const processID = generateRandomId();
33          collaboration.ele('participant', {
34              'id': pool.ID,
35              'name': pool.Name,
36              'processRef': 'Process_' + processID
37          });
38          const process = root.ele('process', {
39              'id': 'Process_' + processID,
40              'isExecutable': 'false'
41          });
42          let laneSet: XMLBuilder;
43          if( pool.Lanes.length > 1) laneSet = process.ele('laneSet', {id: 'LaneSet_' + generateRandomId()});
```

```
44          // Process each lane
45          pool.Lanes.forEach((lane: any) => {
46              if(pool.Lanes.length <= 1){
47                  processLaneComponents(lane, root, process);
48              } else {
49                  // Check if Pools array is not empty
50                  if (lane.Components && lane.Components.length > 0)
                        {
51                      if (laneSet) {
52                          const laneElement = laneSet.ele('lane', {
                                'id': lane.ID, 'name': lane.Name });
53                          processLaneComponents(lane, root, process,
                                laneElement);
54                      }
55                  }
56              }
57          });
58          pool.Lanes.forEach((lane : any) => {
59              processLaneFlows(lane, root, process);
60          });
61      });
62      // Create diagram
63      const diagram = root.ele('bpmndi:BPMNDiagram', {
64          'id': 'GPTBPMNDiagram_1'
65      })
66      .ele('bpmndi:BPMNPlane', {
67          'id': 'BpmnPlane_' + collaborationID,
68          'bpmnElement': 'Collaboration_' + collaborationID
69      });
70      // Process each pool
71      jsonObj.Pools.forEach((pool: any) => {
72          // add pool shapes to diagram
73          const poolShape = diagram.ele('bpmndi:BPMNShape', {
74              'id': pool.ID + '_di',
75              'bpmnElement': pool.ID
76          });
77          poolShape.ele('omgdc:Bounds', {
78              'x': pool.XY[0],
79              'y': pool.XY[1],
80              'width': pool.width,
```

```
81              'height': pool.height
82          });
83          pool.Lanes.forEach((lane: any) => {
84              // add lane shapes to diagram
85              const laneShape = diagram.ele('bpmndi:BPMNShape', {
86                  'id': lane.ID + '_di',
87                  'bpmnElement': lane.ID
88              });
89              laneShape.ele('omgdc:Bounds', {
90                  'x': lane.XY[0] + 30, // adjusting for Pool header
91                  'y': lane.XY[1],
92                  'width': lane.width - 30,
93                  'height': lane.height
94              });
95          });
96          // Process each lane
97          pool.Lanes.forEach((lane: any) => {
98              addComponentsToDiagram(lane, diagram);
99          });
100         pool.Lanes.forEach((lane: any) => {
101             addFLowsToDiagram(lane, diagram, jsonObj);
102         });
103     });
104
105     const xml = root.end({ prettyPrint: true });
106     const xmlPath = path.join(__dirname, '..', 'data', 'xml', `${
        jsonInput}.xml`);
107     fs.writeFileSync(xmlPath, xml);
108     console.log('created file ' + jsonInput + '.xml');
109     return xml;
110 }
111 /**
112  * Process every lane component (assigns incoming and outgoing
        flows)
113  * @param lane contains components
114  * @param root
115  * @param process
116  * @param laneElement if not present lane is not within a pool,
        and lane items can be disregarded
117  */
```

```
118 function processLaneComponents(lane: any, root: any, process: any,
        laneElement?: any): void {
119     // Process each component in the lane
120     lane.Components.forEach((component: { Type: string; ID: any;
            Name: any; Incoming: string[]; Outgoing: string[]; x: any;
            y: any; }) => {
121
122         const taskType = toCamelCase(component.Type)
123         const task = process.ele(taskType, {
124             'id': component.ID,
125             'name': component.Name
126         });
127         // Add incoming flows
128         component.Incoming.forEach((incomingId: string) => {
129             task.ele('incoming').txt(incomingId);
130         });
131
132         // Add outgoing flows
133         component.Outgoing.forEach((outgoingId: string) => {
134             task.ele('outgoing').txt(outgoingId);
135         });
136
137         // Add flowNodeRef to lane
138         if(laneElement) laneElement.ele('flowNodeRef').txt(
                component.ID);
139     });
140 }
141
142 /**
143  * Process every lane flow
144  * @param lane contains flows
145  * @param root
146  * @param process
147  */
148 function processLaneFlows(lane: any, root: any, process: any):
        void {
149     // Process each flow in the lane
150     lane.Flows.forEach((flow: { ID: any; Descriptor: any; Start:
            any; Target: any; Type: any; StartXY:any; TargetXY: any })
            => {
```

```
151         if(flow.Descriptor === undefined) flow.Descriptor = '';
152         const item = process.ele(toCamelCase(flow.Type), {
153             'id': flow.ID,
154             'name': flow.Descriptor,
155             'sourceRef': flow.Start,
156             'targetRef': flow.Target
157         });
158     });
159 }
160
161 /**
162  * Add components to the visible diagram lane by lane
163  * @param lane contains components
164  * @param diagram
165  */
166 function addComponentsToDiagram(lane: any, diagram: any): void {
167     // Process each component in the lane
168     lane.Components.forEach((component: { Type: string; ID: any;
        Name: any; Incoming: string[]; Outgoing: string[]; x: any;
        y: any; }) => {
169         const taskType = toCamelCase(component.Type)
170         // Set position data in bpmndi:BPMNDiagram
171         const shape = diagram.ele('bpmndi:BPMNShape', {
172             'id': `${component.ID}_di`,
173             'bpmnElement': component.ID
174         });
175         if(taskType.toLowerCase().includes('event')){
176             shape.ele('omgdc:Bounds', {
177                 'x': component.x,
178                 'y': component.y + 22,
179                 'width': '36',
180                 'height': '36'
181             });
182         } else if(taskType.toLowerCase().includes('task')){
183             shape.ele('omgdc:Bounds', {
184                 'x': component.x,
185                 'y': component.y,
186                 'width': '100',
187                 'height': '80'
188             });
```

```
189         } else if(taskType.toLowerCase().includes('gateway')){
190             shape.ele('omgdc:Bounds', {
191                 'x': component.x,
192                 'y': component.y+15,
193                 'width': '50',
194                 'height': '50'
195             });
196         } else {
197             shape.ele('omgdc:Bounds', {
198                 'x': component.x,
199                 'y': component.y,
200                 'width': '80',
201                 'height': '80'
202             });
203         }
204     });
205 }
206 
207 /**
208  * Add flows to the visible diagram lane by lane
209  * @param lane contains flows
210  * @param diagram
211  */
212 function addFLowsToDiagram(lane: any, diagram: any, json: any):
        void {
213     // Process each flow in the lane
214     lane.Flows.forEach((flow: { ID: any; Name: any; Start: any;
            Target: any; Type: any; StartXY:any; TargetXY: any }) => {
215         const item = diagram.ele('bpmndi:BPMNEdge', {
216             'id': flow.ID + '_di',
217             'bpmnElement': flow.ID
218         });
219         // find components to adjust flow start and end
                coordinates
220         const {foundComponent: start, foundInLane: startLane} =
                findComponentById(json, flow.Start);
221         const {foundComponent: target, foundInLane: targetLane} =
                findComponentById(json, flow.Target);
222 
223         // in case of missing start or target component
```

```
224         // error prevention
225         if(start === undefined && target === undefined) return;
226         if(startLane === undefined && targetLane === undefined)
                return;
227
228         // if positive flow is going up, if negative flow is going
                down
229         const upOrDown = start.y - target.y;
230
231         // Set position data in bpmndi:BPMNDiagram
232         if(startLane != targetLane) { // if start and target are
                in different lanes
233             if(upOrDown > 0) { // if flow is going up
234                 //------- Start adjustments
                        ------------------------------
235                 // if start is an event shape
236                 if(toCamelCase(start.Type).toLowerCase().includes(
                        'event')){
237                     item.ele('omgdi:waypoint', {
238                         'x': start.x + 18,
239                         'y': start.y + 22
240                     });
241                 }
242                 // if start is a task shape
243                 if(toCamelCase(start.Type).toLowerCase().includes(
                        'task')){
244                     item.ele('omgdi:waypoint', {
245                         'x': start.x + 50,
246                         'y': start.y
247                     });
248                 }
249                 // if start is a gateway shape
250                 if(toCamelCase(start.Type).toLowerCase().includes(
                        'gateway')){
251                     item.ele('omgdi:waypoint', {
252                         'x': start.x + 25,
253                         'y': start.y + 15
254                     });
255                 }
256                 //------- End adjustments
```

```
                    --------------------------------
257                 // if end is an event shape
258                 if(toCamelCase(target.Type).toLowerCase().includes
                        ('event')){
259                     item.ele('omgdi:waypoint', {
260                         'x': target.x + 18,
261                         'y': target.y + 58
262                     });
263                 }
264                 // if end is a task shape
265                 if(toCamelCase(target.Type).toLowerCase().includes
                        ('task')){
266                     item.ele('omgdi:waypoint', {
267                         'x': target.x + 50,
268                         'y': target.y + 80
269                     });
270                 }
271                 // if end is a gateway shape
272                 if(toCamelCase(target.Type).toLowerCase().includes
                        ('gateway')){
273                     item.ele('omgdi:waypoint', {
274                         'x': target.x + 25,
275                         'y': target.y + 65
276                     });
277                 }
278
279             } else {
280                 //------- Start adjustments
                        ------------------------------
281                 // if start is an event shape
282                 if(toCamelCase(start.Type).toLowerCase().includes(
                        'event')){
283                     item.ele('omgdi:waypoint', {
284                         'x': start.x + 18,
285                         'y': start.y + 58
286                     });
287                 }
288                 // if start is a task shape
289                 if(toCamelCase(start.Type).toLowerCase().includes(
                        'task')){
```

```
290                     item.ele('omgdi:waypoint', {
291                         'x': start.x + 50,
292                         'y': start.y + 80
293                     });
294                 }
295                 // if start is a gateway shape
296                 if(toCamelCase(start.Type).toLowerCase().includes(
                        'gateway')){
297                     item.ele('omgdi:waypoint', {
298                         'x': start.x + 25,
299                         'y': start.y + 65
300                     });
301                 }
302                 //------- End adjustments
                        --------------------------------
303                 // if end is an event shape
304                 if(toCamelCase(target.Type).toLowerCase().includes
                        ('event')){
305                     item.ele('omgdi:waypoint', {
306                         'x': target.x + 18,
307                         'y': target.y + 22
308                     });
309                 }
310                 // if end is a task shape
311                 if(toCamelCase(target.Type).toLowerCase().includes
                        ('task')){
312                     item.ele('omgdi:waypoint', {
313                         'x': target.x + 50,
314                         'y': target.y
315                     });
316                 }
317                 // if end is a gateway shape
318                 if(toCamelCase(target.Type).toLowerCase().includes
                        ('gateway')){
319                     item.ele('omgdi:waypoint', {
320                         'x': target.x + 25,
321                         'y': target.y + 15
322                     });
323                 }
324             }
```

```
325         } else { // if start and target are in the same lane
326             // ------- Start adjustments
                    ------------------------------
327             // if start is an event shape
328             if(toCamelCase(start.Type).toLowerCase().includes('
                    event')) {
329                 item.ele('omgdi:waypoint', {
330                     'x': flow.StartXY[0] + 36,
331                     'y': flow.StartXY[1] + 18 + 22
332                 });
333             }
334             // if start is a task shape
335             if(toCamelCase(start.Type).toLowerCase().includes('
                    task')) {
336                 item.ele('omgdi:waypoint', {
337                     'x': flow.StartXY[0] + 100,
338                     'y': flow.StartXY[1] + 40
339                 });
340             }
341             // if start is a gateway shape
342             if(toCamelCase(start.Type).toLowerCase().includes('
                    gateway')) {
343                 item.ele('omgdi:waypoint', {
344                     'x': flow.StartXY[0] + 50,
345                     'y': flow.StartXY[1] + 40
346                 });
347             }
348             // ------- End adjustments
                    --------------------------------
349             // if end is an event shape
350             if(toCamelCase(target.Type).toLowerCase().includes('
                    event')) {
351                 item.ele('omgdi:waypoint', {
352                     'x': flow.TargetXY[0],
353                     'y': flow.TargetXY[1] + 18 + 22
354                 });
355             }
356             // if end is a task shape
357             if(toCamelCase(target.Type).toLowerCase().includes('
                    task')) {
```

```
358                 item.ele('omgdi:waypoint', {
359                     'x': flow.TargetXY[0],
360                     'y': flow.TargetXY[1] + 40
361                 });
362             }
363             // if end is a gateway shape
364             if(toCamelCase(target.Type).toLowerCase().includes('
                    gateway')) {
365                 item.ele('omgdi:waypoint', {
366                     'x': flow.TargetXY[0],
367                     'y': flow.TargetXY[1] + 40
368                 });
369             }
370         }
371     });
372
373 }
374
375 /*
376     Function to convert a string to camel case
377     @param {string} str - The string to convert
378 */
379 function toCamelCase(str: string): string {
380     return str.split(/[\s_-]+/).map((word, index) => {
381             if (index === 0) return word.toLowerCase();
382             else return word.charAt(0).toUpperCase() + word.slice
                    (1).toLowerCase();
383         }).join('');
384 }
385
386 function generateRandomId(length = 6): string {
387     const chars = '
            ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
            ';
388     return Array.from({ length }, () => chars[Math.floor(Math.
            random() * chars.length)]).join('');
389 }
390
391 /**
392  * Find a component by its ID
```

```
393  * @param json entire JSON object
394  * @param id ID of the component
395  * @returns component with the id
396  */
397 function findComponentById(json: any, id: string): any {
398     let foundComponent: any;
399     let foundInLane: any;
400     json.Pools.forEach((pool: any) => {
401         pool.Lanes.forEach((lane: any) => {
402             lane.Components.forEach((component: any) => {
403                 if (component.ID == id) {
404                     foundComponent = component;
405                     foundInLane = lane;
406                 }
407             });
408         });
409     });
410     return {foundComponent, foundInLane};
411 }
```

Listing A.2: convertJsonToXml.ts

```
1  [no prose]
2  [Output only JSON]
3
4  You are a BPMN Component understanding expert.
5  Your task is to analyze and identify BPMN components within a
      given text.
6  Following that you will output all your findings in JSON format.
7
8  You will strictly adhere to this format, only answer with this
      format:
9
10 {
11   "Pools":[
12     {
13     "ID": "p1",
14     "Name": "Pool 1",
15     "XY": [0,50],
16     "width": 400,
17     "height": 150,
18     "Lanes": [
19       {
20         "ID": "l1",
```

```
21          "Name": "Lane 1",
22          "XY": [0,50],
23          "width": 400,
24          "height": 150,
25          "Components": [
26            {
27              "ID": "e1",
28              "Name": "Start",
29              "Type": "Start Event",
30              "x": 100,
31              "y": 100,
32              "Incoming": [],
33              "Outgoing": ["c1"]
34            },
35            {
36              "ID": "t1",
37              "Name": "Task",
38              "Type": "Task",
39              "x": 200,
40              "y": 100,
41              "Incoming": ["c1"],
42              "Outgoing": ["c2"]
43            },
44            {
45              "ID": "e2",
46              "Name": "End",
47              "Type": "End Event",
48              "x": 300,
49              "y": 100,
50              "Incoming": ["c2"],
51              "Outgoing": []
52            }
53          ],
54          "Flows": [
55            {
56              "ID": "c1",
57              "Start": "e1",
58              "Target": "e2",
59              "Type": "Sequence Flow",
60              "StartXY": [100,100],
61              "TargetXY": [200, 100],
62              "Descriptor": "Begin"
63            },
64            {
65              "ID": "c2",
66              "Start": "e1",
```

```
            "Target": "e2",
            "Type": "Sequence Flow",
            "StartXY": [200,100],
            "TargetXY": [300, 100],
            "Descriptor": "Finish"
          }
        ]
      }
    ]
    }
  ]
}

The component table should look like this:
- a unique ID for each element (append an e in front of every ID) (ID)
- a name of the element (Name)
- the BPMN Element type (for example Exclusive Gateway) (Type)
- a possible x position for the element (x)
- a possible y position for the element (y)
- Incoming connections (as an array)
- Outgoing connections (as an array)


The flows table should look like this, use information from the first table.
- a unique connection ID (append a c in front of every ID) (ID)
- connection start (Start)
- the connection target (only to one target per flow) (Target)
- the type of connection (Type)
- StartX, StartY, TargetX, TargetY
- a descriptor for what case the flow is used for (f.e. yes or no decisions), only use these if there are two or more outgoing flows from a component (Descriptor)

For Pools and Lanes, size and x,y coordinates:
A pool should contain all the lanes, and of the lanes content. The lanes should not overlap or leave the pool area.

Keep in mind that from this table we're looking to create a BPMN diagram, so chose parameters such as position and connections carefully.
Choose an appropriate distance between components (at least 200), this does not apply for the connections. This is especially important for the y coordinate. Expect components to be at least 100 points tall.
```

```
103 For the connecting flows, choose the middle point of the
        connecting items. I'd also like you to keep a similar height (
        y-axis) for all the elements, so it will be easier to read
        from left to right.
104
105 Be sure to follow these guidelines, self check on these always:
106
107 1. Start Events:
108     - Generally, a process should have only one Start Event. This
          represents the point where the process begins.
109     - In more complex scenarios, multiple Start Events can be used,
          but they typically represent different conditions or
          triggers for starting the process.
110
111 2. End Events:
112     - A process should conclude with an End Event. However, unlike
          Start Events, it's common to have multiple End Events in a
          single process.
113     - Each End Event signifies a different way in which the process
          can conclude.
114     - Make sure the End Event is inside the pool/lane, so make the
          pool large enough to fit it
115
116 3. Gateways:
117     - Gateways are used to control the flow of the process,
          particularly for branching (divergence) and merging (
          convergence).
118     - Ensure that the gateways are used correctly to represent
          decision points and parallel processes.
119
120 4. Tasks and Sub-Processes:
121     - Choose Tasks for activities that are atomic (cannot be broken
          down further).
122     - Use Sub-Processes when an activity is complex and can be
          broken down into smaller steps or processes.
123     - Try using a specific Task type, for example a "User Task"
124
125
126 5. Sequence Flow:
127     - Sequence Flows should be used to connect elements in a
          process and indicate the order of activities.
128     - Avoid crossing Sequence Flows as it can make the diagram hard
          to read.
129     - It is possible for a Sequence Flow to go back to an "older"
          component, to create loops.
130
```

```
131
132 6. Parallel vs. Sequential Processes:
133    - Use Parallel Gateways to model activities that can occur
         simultaneously.
134    - Sequential activities should be connected directly or through
         appropriate gateways to indicate their order.
135
136 7. Pools and Lanes:
137    - Use Pools to represent different participants or entities in
         a process.
138    - Lanes within Pools can be used to categorize activities under
         different roles or departments.
139
140 8. Events:
141    - Intermediate Events can be used within the process flow to
         represent something that happens between the Start and End
         Events.
142    - Boundary Events can be attached to Tasks or Sub-Processes to
         handle exceptions or timeouts.
143
144 9. Data Objects and Message Flows:
145    - Include Data Objects where necessary to show what data is
         required or produced by activities.
146    - Use Message Flows to illustrate communication between
         different Pools or external entities.
147
148 10. Artifacts:
149    - Use Artifacts like Annotations for additional information
         that aids in understanding the process but does not affect
         the flow.
150
151 Make use of the Component Identifying file to correctly assess
    components.
152
153 In cases of ambiguous or unclear input, it will make educated
    guesses but will keep the response within the structured table
     format.
154 If the text is too vague or cannot be analyzed effectively, the
    response will be 'Error: Unable to process.' + the reason for
    the error.
155
156
157 Once you have generated entire JSON, you will respond to new
    requests differently.
158 Your task is now to make changes to the JSON, upon user request.
159 So on your first response, you answer with the whole JSON, then
```

```
    only with changes you are making to the JSON.
You need to make changes to the JSON on request, but only show the changes that you're making.
There will be four kind of changes:
- CREATE (where you create a new component or flow)
- DELETE (where you remove a component or flow, remember we may need to adjust flows when deleting components)
- UPDATEPARENT (where you change the parent of an item, for example move a component  to a different lane)
- ALTER (where you modify a component or flow)

It is important to note: Only use UPDATEPARENT when you are moving components or flows between pools or lanes, or lanes between pools, for every other change on a flow or component use ALTER.
If you use CREATE to add components to a lane, make sure to make the Pool and Lane bigger.
If you create pools or lanes, also display all items you create within them. Also, make sure to mention a parent of the new item, if possible. Additionally, it is imperative that you need to crosscheck with every other item, that there is no overlapping, ALTER items to move them out of the way if necessary.
If you delete a Pool, it will delete all items within them, so UPDATEPARENT all items that you want to keep into other pools or lanes.
If you use UPDATEPARENT, follow this order of actions: First ALTER or DELETE all connecting items to the item that will be updated, then update and CREATE or ALTER new connections. Important: Pools cannot be parent of a Pool/Component/Flow, Lanes cannot be parent of a Pool, and Components or Flows cannot be parents at all.

If the request contains the Keywords 'GENERATE ENTIRE DIAGRAM' you should answer by returning the entire JSON file.

This is how you should return the changes you're making, and only this way [no prose]:
{
Content: [
  {
    "type": "CREATE",
    "item": "Pool",
    "parentPool": "",
    "parentLane":"",
    "content": {
```

```
      "ID": "p1",
      "Name": "Pool 1",
      "XY": [0,50],
      "width": 400,
      "height": 150,
      "Lanes": [
        {
          "ID": "l1",
          "Name": "Lane 1",
          "XY": [0,50],
          "width": 400,
          "height": 150,
          "Components": [
            {
              "ID": "e1",
              "Name": "Start",
              "Type": "Start Event",
              "x": 100,
              "y": 100,
              "Incoming": [],
              "Outgoing": []
            }
          ],
          "Flows": {}
        },
      ]
    }
  },
  {
    "type": "DELETE",
    "item": "Component",
    "id": "e1"
  },
  {
    "type": "ALTER",
    "item": "Lane",
    "id": "l1",
    "changes": {
      "ID": "l1",
      "Name": "Main Lane",
      "XY": [0,50],
      "width": 300,
      "height": 250,
    }
  },
  {
```

```
    "type": "UPDATEPARENT",
    "item": "Component",
    "id": "e1",
    "from": "l1",
    "to": "l2"
  }
]
}
```

Listing A.3: Final Prompt

Bibliography

[1] Object Management Group. *Business Process Model and Notation (BPMN) version 2.0*. Object Management Group, 2011.

[2] OpenAI. Introducing chatgpt. `https://openai.com/blog/chatgpt/`, 2022. Accessed: 2024-07-21.

[3] Sholiq Sholiq, Riyanarto Sarno, and Endang Siti Astuti. Generating bpmn diagram from textual requirements. *Journal of King Saud University-Computer and Information Sciences*, 34:10079–10093, 2022.

[4] Michael Grohs, Luka Abb, Nourhan Elsayed, and Jana-Rebecca Rehse. Large language models can accomplish business process management tasks, 2023.

[5] Visual Paradigm. What is bpmn? Accessed: 2024-07-29.

[6] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 09 2011.

[7] Cloudflare. What is a large language model (llm)). `https://www.cloudflare.com/learning/ai/what-is-large-language-model/`. Accessed:2024-07-22.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[9] Meta. Introducing meta llama 3: The most capable openly available llm to date, 2024. Accessed: 2024-07-22.

[10] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.

[11] Camunda Services GmbH. bpmn-js: BPMN 2.0 rendering toolkit and web modeler, 2024. Accessed: 2024-08-08.

[12] Angular Team. Angular: One framework. Mobile desktop., 2024. Accessed: 2024-08-08.

[13] Angular Team. Angular Material: UI component library, 2024. Accessed: 2024-08-08.

[14] TypeScript Team. TypeScript: JavaScript with Syntax for Types, 2024. Accessed: 2024-08-08.

Name: Niklas Weidl

Matriculation number: 1049104

Declaration

I hereby assure that I wrote this work independently and did not use any other than the denoted sources and tools.

Ulm, 09.08.24 N. Weidl

Niklas Weidl