Applications of LLMs in Information Systems

Jerin George Mathew[0000-0002-4626-826X] and
Flavia Monti[0000-0003-3349-7861]

Abstract The integration of Large Language Models (LLMs) into information systems is revolutionizing data management and business process automation. This chapter presents two complementary approaches that leverage LLMs to enhance intelligent data processing and workflow execution. By enabling the seamless integration of heterogeneous data and increasing the adaptability of business process management systems, these approaches highlight the potential of LLMs to optimize and advance information systems. Implementation examples provide practical guidelines for developing LLM-based applications for real-world scenarios, illustrating how LLMs can drive efficiency, automation, and intelligent decision-making within information systems.

1 Introduction

The increasing adoption of Large Language Models (LLMs) in various domains has transformed how data is processed, retrieved, and utilized. In the context of industrial applications and business process automation, LLMs offer significant potential for enhancing the efficiency and adaptability of available application systems, also referred to as information systems.

This chapter presents two complementary works that leverage LLMs for data synthesis and process execution. The first work, *COSMADS* (COmposing SMArt Data Services) [5], addresses the challenge of data accessibility and integration in industrial settings. Manufacturing environments generate vast amounts of heterogeneous data, often siloed across different systems. COSMADS facilitates data retrieval through the dynamic composition of information extraction pipelines in response to natural language queries. An LLM agent and a structured repository of data services and past pipelines enable efficient on-demand data processing, bridging the

Sapienza Università di Roma, Rome, Italy,
e-mail: `{mathew,monti}@diag.uniroma1.it`

gap between operators and digital resources. The second work, *NL2ProcessOps* [8], focuses on business process automation by extracting structured process operations from natural language process descriptions. Traditional business process modeling techniques require predefined workflows, which may lack flexibility in dynamic environments. ProcessOps employs an LLM-based approach to translate textual process descriptions into executable scripts, ensuring rapid process adaptation. Tool descriptions and context-aware retrieval mechanisms support the generation of Python-based process workflows that can be executed within business process management systems. The chapter presents such works focusing on the functioning of the developed LLM-based solutions, if the reader wants to get more details on the evaluation please refer to the related full papers.

Together, these works showcase the potential of LLMs in facilitating intelligent data management and process automation. COSMADS emphasizes data retrieval and synthesis in industrial settings, while ProcessOps extends these capabilities to business process execution, demonstrating how LLMs can drive automation across different operational domains.

2 Data-on-demand in the industrial domain with LLMs

In modern manufacturing environments, the ability to access and synthesize data on demand is increasingly vital for maintaining operational efficiency and agility. From shop floors to administrative offices, manufacturing companies generate vast amounts of heterogeneous data, including sensor readings, production metrics, and machine statuses. However, this data often resides in siloed systems, limiting its accessibility and utility. In this section, we introduce *COSMADS* – COmposing SMArt Data Services [5], a tool that synthesizes information extraction pipelines, in the form of Python scripts, starting from natural language queries and a documented *codebase* consisting of available data services and possibly other, previously defined, information extraction pipelines. Such pipelines, in particular, can be either manually defined or formerly generated with *COSMADS*.

2.1 The *COSMADS* Architecture

Figure 1 depicts the architectural components of *COSMADS*. A new execution is spawned as soon as a human operator specifies a natural language query (1) to retrieve information from the ongoing manufacturing process. A query can be parametric, meaning that it can provide a set of input arguments (e.g., a time range, or a specific kind of defect to be monitored). The reader can imagine the human operator to have little or zero knowledge about the available data services. As a consequence, it can be supposed that the query only expresses the required information without technical details on how to compute it.

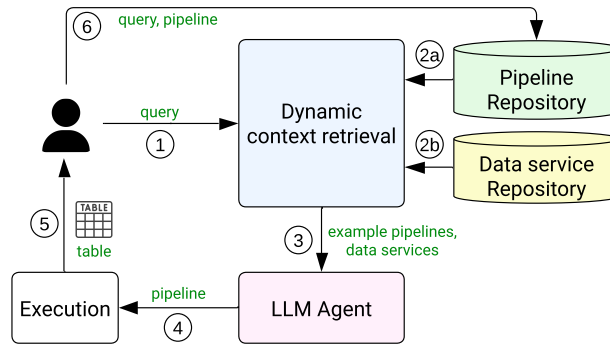

Fig. 1 *COSMADS* high-level architecture [5]

The core of the architecture is represented by an *LLM Agent* that instructs a pre-trained LLM by feeding a query-specific prompt, which is built according to the output of the *dynamic context retrieval* component. This module analyzes the query and retrieves (2a) a set of example pipelines from a *pipeline repository* and (2b) a set of data services from a *data service repository* that can be used to answer the query.

Data services exposed by manufacturing assets in a factory can range from operational/actuating services (e.g., turning on the camera) to services that generate data (e.g., getting the current speed of the die cutting machine using its embedded chip). Data services can be accessed using different paradigms and communication protocols, but for simplicity, the authors assume they can be called through function calls wrapping the actual calling mechanism. A data service can expect a set of parameters and returns an output, which can be either structured or unstructured. For each data service, the *data service repository* contains a *documentation*, i.e., a textual description of the functioning and usage of a data service.

The *pipeline repository* consists of all the pipelines already available. These pipelines can be manually defined or obtained from previous executions of *COSMADS*. Each pipeline is associated with the query it fulfills. In general, pipelines can be defined using various modeling formalisms, such as programming languages and scientific workflow scripting languages [9], among others. In *COSMADS*, pipelines are software (Python) scripts that *(i)* produce a table as an output, and *(ii)* make use of data services.

Examples of pipelines and documentation of relevant data services are fed (3) to the *LLM Agent* together with the original query. Such input data is incorporated into a prompt, according to a specific prompt template. The output pipeline generated by the *LLM Agent* is then sent (4) to the execution module (e.g., the Python interpreter).

The execution of the pipeline finally (5) produces a table answering the query. If the human operator thinks the produced pipeline can be helpful as a future reference for future queries, the pipeline together with the originating query can be stored (6) in the pipeline repository.

2.2 Implementing COSMADS

A Python-based prototype of *COSMADS*[1] has been implemented using the Langchain framework. The base LLM model utilized for the LLM Agent relies on GPT-4 (`gpt-4-turbo`) by OpenAI[2].

Dynamic context retrieval

The contextual information required for the prompt of the LLM agent consists of *(i)* relevant previous queries with corresponding pipelines to be used as *few-shot examples*, and *(ii)* the set of data services needed for solving the input query. As discussed in Section 2.1, this information can be obtained from the *pipeline repository* and the *data service repository* respectively.

The *pipeline repository* is implemented as a vector store containing the vector representations of queries already solved. Each vector contains the embedding of the `query` and some metadata, including the path to the Python script containing the `pipeline` that solves that query. An example of a query that is embedded into a dense vector is provided below.

Example of stored query and pipeline metadata

```
1 {
2     "query": "Consider the next 5 carboard of the first diecutter.
    ↪  Generate a table containing: (i) the number of cardboards with
    ↪  no defects, (ii) those with errors, (iii) how many fold errors,
    ↪  and (iv) how many hole errors.",
3     "metadata":{
4         "pipeline": "pipelines/04_07_2024-21_43_24.py"
5     }
6 }
```

The embedding is computed by applying the `text-embedding-ada-002` model by OpenAI. For the vector store, the authors rely on DocArray's DocIndex[3], which is well integrated into Langchain and allows us to efficiently access stored data. The implementation details of the pipeline repository are provided below, omitting or simplifying some methods for the sake of clarity.

Pipeline repository implementation

```
1 from langchain_community.vectorstores import DocArrayInMemorySearch
2 from langchain_core.documents import Document
```

[1] `https://github.com/jermathew/COSMADS`

[2] The base LLM model for the LLM Agent can be any LLM model available in the literature. Some changes in the prompt may be necessary if another model is selected.

[3] Cf. `https://github.com/docarray/docarray`

```
3  import json
4
5
6  class PipelineStore():
7
8      def __init__(self):
9          json_setup = json.load(open('queries_pipelines.json'))
10         queries = json_setup.keys()
11         self.docs = []
12         for q in queries:
13             self.docs.append(
14                 Document(
15                     page_content = json_setup[q]['query'],
16                     metadata = {"pipeline":json_setup[q]['pipeline']}
17                 )
18             )
19
20     def embed_docs(self, embedding_function):
21         self.embedding_function = embedding_function
22
23         self.db =
           ↪ DocArrayInMemorySearch.from_documents(documents=self.docs,
           ↪ embedding=embedding_function)
24
25     def search(self, query):
26         best_result = None
27         best_match = self.db.similarity_search(query)
28         best_result = best_match[0]
29         return {'api_name': self.__class__.__name__, 'input': query,
           ↪ 'output': best_result, 'exception': None}
```

The vector store is implemented as a Python class called `PipelineStore`. During instantiation (lines 8–18), the `PipelineStore` reads a json file containing the list of available pipelines and the query from which they stem. Each query-pipeline pair in the json file is formatted similarly to the example provided above. Each query is then wrapped into a `Document`, a built-in class from the Langchain library that provides a unified interface for managing and manipulating documents (e.g. generating an embedding).

The `embed_docs` method (line 20) contains the logic for embedding each `Document` parsed from the json file during initialization. The method takes an `embedding function` parameter, i.e. a function that maps a string to a dense vector using an embedding model and uses it to embed each query and store it into DocArray, using the `DocArrayInMemorySearch` class. The latter represents a wrapper class for DocArray that allows easy storage of embeddings and performing queries.

The retrieval procedure is implemented in the `search` method (line 25). The core part is the `similarity_search` function, which takes a query as a parameter and (i) transparently embeds it into a dense vector and (ii) returns a sorted list of similar queries in descending order of cosine similarity. Particularly, given a natural

language query, the *dynamic context retrieval* component computes its embedding and retrieves the top-K similar queries from the vector store.

The `PipelineStore` can be initialized and the `text-embedding-ada-002` model by OpenAI can be specified as the embedding function using the `OpenAIEmbeddings` class, as shown in the code snippet below:

Specify an embedding function for DocArray

```python
from langchain_openai import OpenAIEmbeddings

self.pipeline_store = PipelineStore()
openai_key = None # replace with your OpenAI API key
embedding_function = OpenAIEmbeddings(model="text-embedding-ada-002",
    api_key=openai_key)
self.pipeline_store.embed_docs(embedding_function)
```

To simplify interaction with the `PipelineStore` and allow for convenient configuration, the system introduces a wrapper class, `PipelineManagerDB`.

The PipelineManagerDB wrapper class

```python
class PipelineManagerDB:

    def __init__(self, openai_key):
        self.pipeline_store = PipelineStore()
        embedding_function =
            OpenAIEmbeddings(model="text-embedding-ada-002",
            api_key=openai_key)
        self.pipeline_store.embed_docs(embedding_function)

    if __name__ == '__main__':
        openai_key = None # replace with your OpenAI API key

        tools_manager = PipelineManagerDB(openai_key=openai_key)
```

This wrapper simplifies the initialization of the vector store by enabling direct specification of the embedding function, which, in our case, is OpenAI's `text-embedding-ada-002` model.

The set of data services needed for solving the input query is stored in the *data service repository*. A data service is used to retrieve historical or online data. Online data is often associated with the execution of some operation (e.g., taking a picture). As a consequence, any asset of the company exposing services to support its operational functionalities can be considered a data service. Referring to the example case study, an example of data service is the service related to the camera asset which captures frames (i.e., `DS3`). In *COSMADS*, each data service is realized as a Python class having two main components: *(i)* a function wrapping the existing service of the asset and representing the actual execution logic, and *(ii)* a class variable containing the documentation. The LLM Agent relies on the documentation of the data services, which summarizes their capabilities, how they need to be used,

a one-shot example, and a specification of the input and output parameters. The data service documentation of the camera asset capturing frames is provided below. Notably, this documentation corresponds to what in traditional service composition was referred to as service description.

Data service contextual information

```
"brief_description": "Data service that, given the id of a camera1,
↪ provides a frame captured from that camera1.",
"detailed_description":
    """Data service that, given the id of a camera1, provides a frame
    ↪ captured from that camera1.
    In general instances of camera1 point downwards to a conveyor belt
    ↪ of a specific production line that transports single cutout
    ↪ cardboards produced by a specific die machine.
    The data service takes a single parameter, namely the id of the
    ↪ camera1 (an integer) and returns a frame captured from that
    ↪ camera1 as a numpy matrix.
    The matrix is a 2D array having a shape of (1080, 1920, 3) where
    ↪ 1080 is the height, 1920 is the width and 3 is the number of
    ↪ channels (RGB).
    Example usage:
    - If the id of the camera1 is 123, then the data service would be
    ↪ called as follows:
    camera1_id = 123
    frame = GetFrameFromCamera1.call(camera1_id=123)
    # assuming the frame is a numpy matrix
    print(frame.shape)  # (1080, 1920, 3)
    Things to keep in mind:
    - The refresh rate of the camera is 1 second, i.e. the frame is
    ↪ updated every second, so if the data service is called multiple
    ↪ times within a second, it will return the same value.
    - The frame is a numpy matrix, so avoid trying to access it as a
    ↪ dictionary.""",
"input_parameters": ["camera1_id:int"],
"output_values": ["frame:np.matrix"],
"module": "camera1"
```

Noteworthy, while the *dynamic context retrieval* select only K example pipelines to be included in the prompt, all data services are considered. This approach is justified by the assumption that the number of services in the entire repository remains relatively stable, as the codebase of a company typically grows slowly and is also relatively small compared to the maximum prompt length allowed by the LLM. Conversely, the number of pipelines is expected to grow more significantly over time, as new pipelines are added to the repository, either through manual definition or automatic generation by *COSMADS*. Also, this excludes the possibility an incorrect pipeline is generated, simply because information about needed data services is not available.

LLM Agent

The LLM Agent leverages the ICL ability of LLMs [2]. The quality of the output though is strongly dependent on the quality of the provided prompt [13]. For *COSMADS*, in particular, a *prompt template* is designed to be filled with the output of the *dynamic context retrieval* module, which follows the most common best practices [14]. These include *(i)* expressing the goal task clearly, *(ii)* including contextual information, *(iii)* providing demonstrations, and *(iv)* utilizing model-friendly format style. The prompt template is provided below[4]:

Prompt template

```
<LLM AGENT EXPERTISE>
Query: {query}
<GOAL DESCRIPTION>
{data_services}
<DATA SERVICES DOCUMENTATIONS STRUCTURE>
<GUIDELINES>
Here examples of pipelines that may help you in generating a new
↪  pipeline:
Query: {example_query}
Pipeline: {example_pipeline}
...other examples...
Answer:
```

The invariable parts of the prompt include *(i)* a system header describing the skills of the agent, *(ii)* a description of the specific goal to be fulfilled, *(iii)* the description of the structure of the documentation of the data services, and *(iv)* a set of guidelines the output of the agent must respect. It also contains the *dynamic context retrieval* output, i.e., the set of data services the pipeline can call, and the example queries. Finally, the input natural language query to be answered is also provided.

The implementation of the LLM Agent is provided below.

Implementation of the LLM Agent

```
1  from langchain.prompts import ChatPromptTemplate
2  from langchain.schema import BaseOutputParser
3  from langchain_openai import ChatOpenAI
4
5  class PipelineGeneratorAgent:
6      """The agent that designs the pipeline."""
7
8      def __init__(self, openai_key):
9          """Initialize the agent."""
10         # define the prompt
11         prompt_template = TEMPLATE
```

[4] The full prompt is available at `https://github.com/jermathew/COSMADS/blob/main/src/pipeline_chain.py`.

```
12          self.prompt = ChatPromptTemplate.from_template(prompt_template)
13          # define the LLM
14          self.llm = ChatOpenAI(model="gpt-4-turbo",
15                                api_key=openai_key,
16                                temperature=0.0)
17          # define the output parser
18          self.output_parser = CustomOutputParser()
19
20      def get_chain(self):
21          # generate the python function
22          agent_chain = self.prompt | self.llm | self.output_parser
23          return agent_chain
24
25
26  class CustomOutputParser(BaseOutputParser):
27      """The output parser for the LLM."""
28
29      def parse(self, text: str) -> str:
30          text = text.strip("\n")
31          text = text.strip()
32          # count how many ``` are in the text
33          back_count = text.count("```")
34          if back_count != 2:
35              print(text)
36              raise ValueError("The string should contain exactly two
                ↪ triple backticks")
37          code = text.split("```")[1]
38          code = code.strip()[len("python"):].strip()
39          return code
```

The LLM Agent is implemented as a class called `PipelineGeneratorAgent` and its inner logic is defined in the `get_chain` method (line 20). This method implements the chain as a linear sequence of tasks, consisting of *(i)* passing the prompt to the model, *(ii)* generating the pipeline, and *(iii)* parsing its output. This sequence of steps is defined in line 22 using LCEL. The final part of the pipeline involves a custom output parser which is used to parse the output generated by the LLM Agent. The prompt, the LLM, and the output parser are defined in the constructor of the `PipelineGeneratorAgent` (lines 8–18). The prompt is set up in line 11 according to the previously described prompt structure and is then wrapped into a `ChatPromptTemplate`, a utility class to format the prompt for OpenAI's GPT models. The constructor also creates a `ChatOpenAI` object which is a general wrapper for OpenAI models and is set up with `gpt-4-turbo`. Finally, a `CustomOutputParser` object (defined in lines 26-39) is initiated, which strips off character delimiters from the Python code generated by the LLM Agent.

The COSMADS LCEL chain

We have already seen an example of a chain in the `PipelineGeneratorAgent` class:

The chain used in PipelineGeneratorAgent

```python
def get_chain(self):
    # generate the python function
    agent_chain = self.prompt | self.llm | self.output_parser
    return agent_chain
```

This simple composition demonstrates how LangChain chains can be built using modular components. The LangChain Expression Language (LCEL) extends this idea by structuring more complex chains that involve multiple sequential and parallel operations. It introduces components such as `RunnableLambda` for single-step transformations and `RunnableParallel` for concurrent execution, enabling efficient and flexible task orchestration. LCEL also allows for the dynamic composition of LLM-driven workflows, making them well-suited for integrating retrieval, generation, and execution steps, as seen in the *COSMADS* system.

We now present the high-level functioning of *COSMADS* through its LCEL chain. The main implementation is encapsulated in the `COSMADS` class, detailed below.

The COSMADS class

```python
import sys
from pathlib import Path
import dotenv
import os
from langchain.schema.runnable import Runnable, RunnableLambda,
↪  RunnableParallel, RunnablePassthrough

# append the path to the parent directory to the system path
import sys
sys.path.append(str(Path(__file__).parent.parent.parent))

from pipeline_manager_db import PipelineManagerDB
from pipeline_chain import PipelineGeneratorAgent
from runner_chain import PipelineRunner

INTERMEDIATE_RESULTS_FILEPATH = Path(__file__).parent /
↪  "temp_pipeline.py"

class COSMADS:
    def __init__(self):
        dotenv.load_dotenv()
        OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")

        self.pipeline_manager = PipelineManagerDB(OPENAI_API_KEY)
        self.generator = PipelineGeneratorAgent(OPENAI_API_KEY)
```

```
24          self.runner = PipelineRunner()
25
26
27
28      def get_chain(self) -> Runnable:
29          generator_chain = self.generator.get_chain()
30          runner_chain = self.runner.get_chain()
31
32          generator_chain_output = {
33              "pipeline": generator_chain,
34              "inputs": RunnablePassthrough()
35          }
36
37          runner_chain_output = {
38              "output": runner_chain,
39              "inputs": RunnablePassthrough()
40          }
41
42          chain = ...
43
44          # return the chain
45          return chain
46
47
48  if __name__ == "__main__":
49      llm = COSMADS()
50      query = "" # your query here
51      result = llm.get_chain().invoke(query)
52      print(result["output"])
```

The *COSMADS* class includes a `get_chain` method that consists of a `Runnable` object implementing the *COSMADS* processing flow depicted in Figure 1. The chain can be called using the `invoke` method (line 54) by passing an input query and will execute the *COSMADS* system, including the generation of the Python script and its execution.

The execution of the generated pipeline is managed by the `PipelineRunner` class, which is responsible for running the generated Python script and processing its results. We provide the implementation details of the `PipelineRunner` class below:

PipelineRunner implementation

```
1   class PipelineRunner:
2
3       def run_pipeline(self, pipeline_filepath: str) -> dict:
4           execution_ok = False
5
6           cwd = Path.cwd()
7           os.chdir(Path(pipeline_filepath).parent)
8           execution_result = os.system(f"python {pipeline_filepath}")
9
```

```
10          if execution_result == 0:
11              execution_ok = True
12
13          os.chdir(cwd)
14          return execution_ok
15
16      def parse_pipeline_result(self, pipeline_result_filepath: str) ->
        ↪  dict:
17          with open(pipeline_result_filepath, "r") as f:
18              result = json.load(f)
19
20          result = pd.DataFrame(result)
21          result = tabulate(result, headers='keys', tablefmt='psql')
22          return result
23
24      def get_chain(self) -> Runnable:
25          runner_chain = (
26              RunnableLambda(lambda x: {
27                  "execution_ok":
                    ↪  self.run_pipeline(x["pipeline_filepath"])
28              })
29              | RunnableBranch(
30                  (lambda x: x["execution_ok"], RunnableLambda(lambda x:
                    ↪  self.parse_pipeline_result(
31                  str(PIPELINE_RESULT_FILEPATH)
32                  ))),
33                  (RunnableLambda(lambda x: "The pipeline did not run
                    ↪  successfully"))
34              )
35          )
36          return runner_chain
```

The `PipelineRunner` class consists of three core methods. The `run_pipeline` method (line 3-14) executes the pipeline script in its designated directory and returns a boolean indicating whether the execution was successful. If the execution completes without errors, the output of the pipeline is stored in a JSON file. The `parse_pipeline_result` method (line 16-22) loads the result into a Pandas DataFrame and formats it as a table for display. The `PipelineRunner` is integrated into the COSMADS LCEL chain through its `get_chain` method, which defines the execution workflow. The chain consists of a `RunnableLambda` that triggers the pipeline execution, by invoking the `run_pipeline` method, followed by a `RunnableBranch` that checks if the execution was successful. If the pipeline runs successfully, the output is parsed using the `parse_pipeline_result` method and returned, otherwise, an error message is displayed.

The implementation details of the whole *COSMADS* chain are provided below. Additionally, Figure 2 provides its visual explanation.

LCEL implementation of COSMADS processing flow

```
chain = (
    # Step 1: Retrieve similar pipelines from the Pipeline repository
    RunnableLambda(lambda x: {
        "query": x,
        "pipeline_search":
        ↪ self.pipeline_manager.pipeline_store.search(x),
        }
    )
    # Step 2: Retrieve the the most relevant data services
    # for this query from the Data Service repository
    | RunnableLambda(
        lambda x: {
            "query": x["query"],
            "example":
            ↪ self.get_example(x["pipeline_search"]["output"]),
            "data_services": self.get_data_services()
        }
    )
    # Step 3: Parse the retrieval results from both repositories
    | RunnableLambda(
        lambda x: {
            "query": x["query"],
            "data_services": x["data_services"][0],
            "data_services_list": x["data_services"][1],
            "example_query": x["example"][0],
            "example_pipeline": x["example"][1],
        }
    )
    # Step 4: Generate the pipeline using the LLM Agent
    | generator_chain_output
    # Step 5: Store the pipeline into a temporary Python script file
    | RunnableParallel(
        gen = RunnableLambda(lambda x: {
            "query": x["inputs"]["query"],
            "data_services": x["inputs"]["data_services"],
            "example_query": x["inputs"]["example_query"],
            "example_pipeline": x["inputs"]["example_pipeline"],
            "pipeline": x["pipeline"]
        }),
        exe = RunnableLambda(lambda x:
            self.save_intermediate_result_to_json(x["pipeline"],
            ↪ x["inputs"]["data_services_list"])
        )
    )
    | RunnableLambda(lambda x: {
        "inputs": x,
        "pipeline_filepath": str(INTERMEDIATE_RESULTS_FILEPATH)
    })
    # Step 6: Execute the pipeline (i.e. the temporary file)
    | RunnableParallel(
        inputs = RunnableLambda(lambda x: {
```

```
49                "query": x["inputs"]["gen"]["query"],
50                "data_services": x["inputs"]["gen"]["data_services"],
51                "example_query": x["inputs"]["gen"]["example_query"],
52                "example_pipeline": x["inputs"]["gen"]["example_pipeline"],
53                "pipeline": x["inputs"]["gen"]["pipeline"],
54            }),
55            output = runner_chain_output
56        )
57        # Step 7: Parse the pipeline output
58        | RunnableLambda(lambda x: {
59            "query": x["inputs"]["query"],
60            "data_services": x["inputs"]["data_services"],
61            "example_query": x["inputs"]["example_query"],
62            "example_pipeline": x["inputs"]["example_pipeline"],
63            "pipeline": x["inputs"]["pipeline"],
64            "output": x["output"]["output"],
65        })
66    )
```

The *COSMADS* LCEL chain is structured as a sequence of `RunnableLambda` and `RunnableParallel` components, the two foundational abstractions in LangChain. `RunnableLambda` is designed for single-step transformations or computations. It wraps a callable function or lambda, providing a flexible way to define and execute individual tasks within a chain. For example, it can retrieve data, transform inputs, or parse outputs, all while maintaining a clear and isolated scope for each operation. In the context of *COSMADS*, `RunnableLambda` is used to perform tasks like querying repositories and parsing retrieval results. `RunnableParallel`, on the other hand, facilitates the concurrent execution of multiple independent tasks. It is particularly useful in scenarios where different processes can run simultaneously without interdependencies, enhancing the efficiency of the workflow.

The chain begins with a `RunnableLambda` (line 3) that queries the pipeline repository to retrieve pipelines relevant to the input query. This step leverages existing knowledge by identifying similar pipelines to use as examples in the generation process. Following this, another `RunnableLambda` (line 10) fetches data services from the data service repository that are most relevant to fulfilling the query. These two steps ensure that the *COSMADS* system gathers the necessary contextual information for creating a new pipeline.

Once the repository outputs are retrieved, a third `RunnableLambda` (line 18) parses and organizes the results, wrapping the query, example pipelines, and data services into a structured format. This parsed information is then fed into the LLM Agent, which is implemented as part of a `RunnableLambda` (line 28). The agent uses the provided context to generate a Python script that represents the pipeline for fulfilling the query.

The generated pipeline is then stored using a `RunnableParallel` component (line 30), which handles both writing the pipeline to a temporary Python file and saving metadata about the inputs and selected data services. Subsequently, another

RunnableParallel component (line 42) executes the stored pipeline while capturing its outputs concurrently.

Finally, a RunnableLambda (line 47) processes the output of the executed pipeline, structuring it into a relational table or other desired formats based on the query. The chain concludes with this step, delivering the final result to the user.

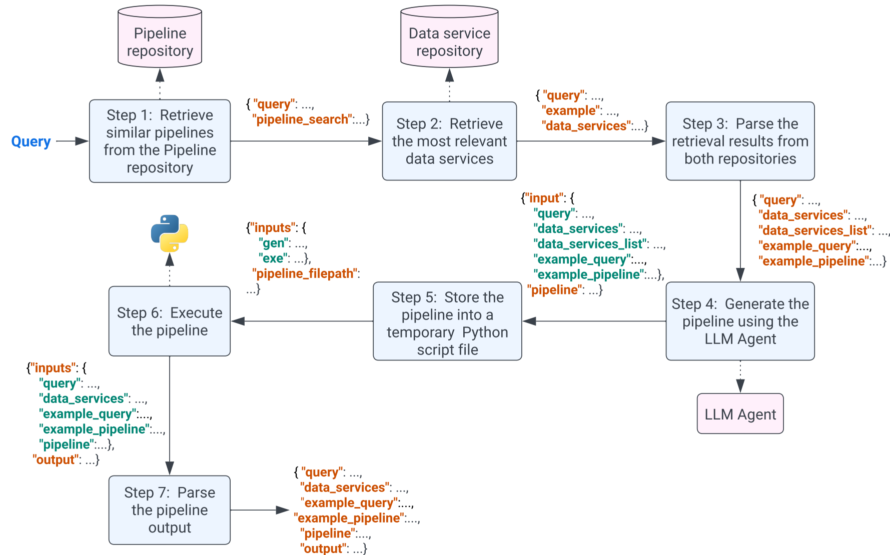

Fig. 2 *COSMADS* LCEL chain explained

3 Extracting Process Operations with LLMs

The complexity of modern business processes often demands rapid and efficient solutions for translating textual descriptions into executable process operations. Traditional approaches to business process management rely heavily on predefined models, making them inflexible to evolving requirements. In this section, we describe NL2ProcessOps [8], an LLM-based approach for generating a process script from a textual process description. In practice, the script is a Python code containing invocations to external tools for executing the tasks of the given process. Each tool is characterized by a description that provides a textual representation of operations offered to execute the tasks. Tool documentations are prompted to the LLM to generate the script. To overcome the limitation of the input context length of LLMs, which cannot incorporate too much information, we consider the most appropriate tool descriptions and fit them within the prompt length.

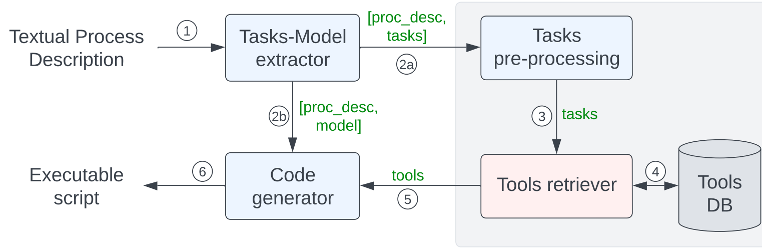

Fig. 3 Architecture of the NL2ProcessOps [8]

3.1 The NL2ProcessOps architecture and pipeline

The goal of NL2ProcessOps is to provide a solution for process operations (ProcessOps), aiming to simplify the development and deployment of processes, similar to how DevOps simplifies the development and deployment of general software. Specifically, once a new process definition is available, the involved operations include *(a)* extracting a process model from the description, defining the legal execution traces similarly to a program, *(b)* binding of each task to a software module implementing that task, and *(c)* defining the data flow, i.e., how data artifacts are manipulated by the tasks.

The generation of the *process script* from a textual process description is broken down into multiple *stages* supported by LLMs and chained together as follows: *(i)* extraction of tasks and control flow from the textual process description (operation *(a)* of ProcessOps), *(ii)* retrieval of relevant tools corresponding to the extracted tasks (operation *(b)* of ProcessOps), and *(iii)* generation of the process script implementing the process (operation *(c)* of ProcessOps). Figure 3 illustrates the components of NL2ProcessOps and their interactions. The numbers in the circles represent the order of the performed operations.

Stage *(i)* consists of a textual process description given as input ① to the *Tasks-Model extractor*. This component is an LLM prompted to extract the tasks and the control flow of the process and generates the model representation as a Mermaid.js [3, 4]. The list of tasks paired with the textual process description (i.e., `[proc_desc, tasks]`) is input (2a) to the *Tasks pre-processing* component. Concurrently, the process model and the textual process description (i.e., `[proc_desc, model]`) are input (2b) to the *Code generator* component.

Stage *(ii)* (depicted in gray in Figure 3) is inspired by the RAG concept to retrieve the relevant tools for the particular textual process description. The *Tasks pre-processing* component employs an LLM to refine the descriptions of the extracted tasks based on the textual process description. The refined list of tasks is then processed ③ by the *Tools retriever* component. This component interacts ④ with the vector database *Tools DB* and retrieves the most similar embedded tools offering the most suitable operation for each embedded task. *Tools DB* stores vectors consisting of the embeddings of the descriptions of the tools.

The list of retrieved tools is fed ⑤ into the *Code generator* component to initiate stage *(iii)*. The *Code generator* LLM. The *Code generator*, given the textual process description, process model and the list of tools (and their operations) implementing the process tasks, generates ⑥ a Python code – *process script* – embedding the control and data flows and implementing the process.

The integration of LLMs within the proposed approach is essential for several reasons. LLMs excel at processing natural language, making them ideal for extracting tasks and control flow from textual process descriptions. This first stage is critical as it forms the basis for the subsequent stages of the pipeline. Without accurate extraction of tasks and control flow, subsequent stages would lack the necessary information for generating meaningful outputs. The retrieval of relevant tools highly relies on the quality of the extracted tasks. An incorrect set of tools would affect the generation of the process script, leading to incorrect data and control flows. Finally, LLMs excel in generating high-quality code from a description. In this case, the textual process description guides the code generation, supported by the control flow and tools information derived from the previous stages.

3.2 A running example

Let us consider a real-world example in the Smart Manufacturing domain. Smart Manufacturing is a modern trend where cutting-edge technologies such as Industrial Internet of Things (IIoT) and Artificial Intelligence (AI) play pivotal roles in enabling quality enhancement, optimization and automation of production processes [10]. In this domain, the integration of the proposed solution, paired with PEE and enterprise systems like Manufacturing Execution System (MES) and Enterprise Resource Planning (ERP) enables the orchestration and execution of specific processes in a quick and efficient way [12].

> **Example** The automatic calibration process of cardboard production consists of continuously capturing a photo of the cardboard being produced. Each photo is analyzed to check if all the markers identified are ok. If markers are not ok, the calibration process continues. If the markers are ok, the speed of the die-cutting machine is set to 10000 RPM and the process ends.

The *Example* describes the automatic calibration process in cardboard production. Cardboard production is a manufacturing process that involves a die-cutting machine for the transformation of raw cardboard into printed cut-out cardboard sheets for the packaging industry. When starting a new order, calibration is needed to guarantee quality before proceeding with the production.

Figure 4 depicts the input-output of each of the NL2ProcessOps components over the *Example*. The numbers in circles are tightly connected to those in Figure 3. Figure 4 reports the process model represented as Mermaid.js (green colored) and the list of (refined) tasks (blue and red colored) extracted from *Example*, the list of tools operations (purple colored) retrieved, and the process script – Python code (black colored). All the artifacts are available at the provided link [5].

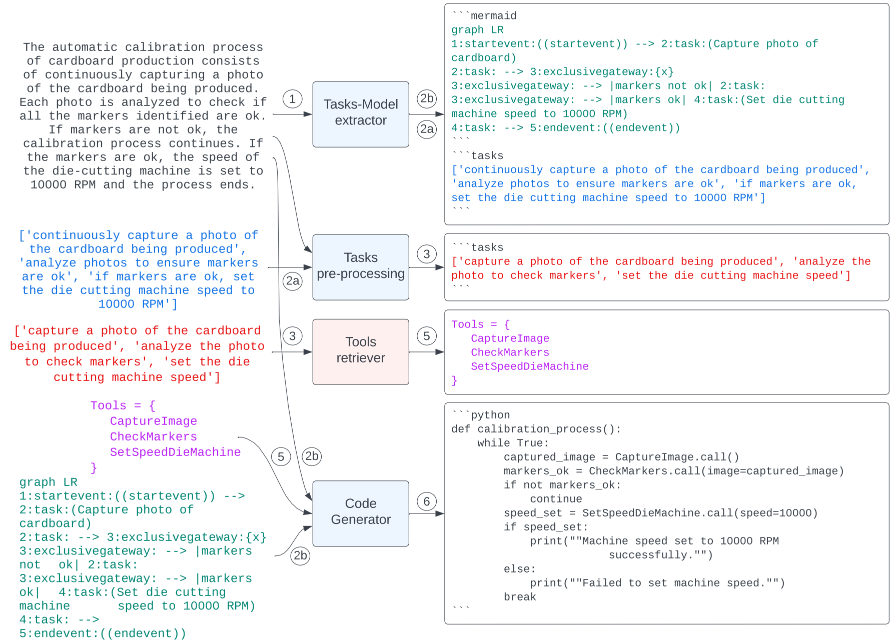

Fig. 4 Input-output of each component of NL2ProcessOps on *Example*

3.3 Realization

The authors have realized a prototype[5] of NL2ProcessOps [7]. The prototype is developed in Python and built on top of Langchain[6], a framework for constructing LLM-powered applications facilitating the creation, management and chaining of prompts. The base LLM model utilized for the three LLM-based components is GPT-4 (`gpt-4-0125-preview`) from OpenAI [1] with temperature set to 0, implying a more deterministic LLM mode.

Tools retriever

Tools are central to such approach as they support task execution in a service-oriented way, i.e., offering operations for specific tasks. As proposed in [6], the LLM relies solely on the documentation of these tools which outlines their capabilities, usage instructions, and outcomes. An excerpt from the documentation for the `SetSpeedDieMachine` tool of the *Example* is provided below. It consists of a general description of its unique operation, parameters details, and outcome description.

[5] Cf. `https://github.com/iaiamomo/NL2ProcessOps`.

[6] Cf. `https://www.langchain.com/`.

Example of tool documentation

```
1 SetSpeedDieMachine = {
2     "description": "Set the speed of the die-cutting machine.",
3     "more_details": "It takes as input the speed. It returns a boolean
    ↪  value, True if the speed has been set, False otherwise.",
4     "input_parameters": ["speed:int"],
5     "output_parameters": ["speed_set:bool"],
6     "tool": "die_machine" }
```

The *Tools DB* is a vector database containing the vector representations of the descriptions of the available tools. Each vector constitutes the embedding of the following information: `tool_name` and `description`, where the `description` is extracted from the tool documentation. An embedding is a sequence of numbers that represent information and enable better comprehension of relationships between similar information. The `text-embedding-ada-002` model from OpenAI[7] is utilized to compute these vectors. The embeddings are usually managed in vector DBs which enable a set of capabilities, including indexing, distance metrics and similarity search. As vector DB ChromaDB[8], which is open-source and well integrated with Langchain, is employed. ChromaDB enables the implementation of RAG-based approach basing similarity search on cosine distance (where a lower score indicates better similarity).

In the *Example*, for the refined extracted task description *"set the speed of the die cutting machine"*, the similarity search identifies `SetSpeedDieMachine` tool operation as the most similar. Consequently, the *Die machine* tool is the most suitable for task execution.

The similarity search outputs a set of most similar tools with corresponding score results. The module guarantees that the useful tools are among those retrieved. In practice, given a task, the *Tool retriever* selects one or more tools based on their score values. The set of tools is then processed by the *Code generator* LLM that considers only those necessary for the specific case.

The tools retriever is implemented as a Python class responsible of managing tools, which are implemented as Python classes themselves. An example of tool is provided below:

Sample tool implementation

```
1 class ConfigureAssemblyLine:
2     description = {
3         "description": "Configure the assembly line.",
4         "more details": "This tool takes no input and does not return
        ↪  anything.",
5         "input_parameters": [],
6         "output_parameters": [],
```

[7] Cf. https://platform.openai.com/docs/models/embeddings.

[8] Cf. https://www.trychroma.com/.

```
7            "actor": "assembly_line"
8        }
9
10       def call():
11           return
```

A simplified version of the class implementing the *Tool Retriever* is provided below.

Embedding tools in NL2ProcessOps

```
1   from langchain_community.vectorstores.chroma import Chroma
2
3   class ToolStore():
4
5       def embed_tools(self, embedding_function):
6           self.embedding_function = embedding_function
7           self.db = Chroma.from_texts(self.tools, embedding_function)
8
9       def search(self, keywords):
10          best_match = self.db.similarity_search_with_score(keywords)
11          for i, match_elem in enumerate(best_match):
12              if i == 0 and match_elem[1] >= 0.5:
13                  break
14              elif i > 0 and match_elem[1] <= 0.3:
15                  tool_name = match_elem[0].page_content.split(' ')[0]
16                  file_name = match_elem[0].page_content.split(' ')[1]
17                  api_info = self.extract_input_output(tool_name,
                    ↪  file_name)
18                  list_match.append(api_info)
```

The `embed_tools` method (line 5) makes usage of the `Chroma.from_texts` method (line 7), which takes a list of tool descriptions and converts them into embeddings, which are then indexed in ChromaDB for similarity-based searches.

The search method (line 9) performs retrieval by calculating the cosine similarity between the query embeddings and the stored tool embeddings. It uses ChromaDB's `similarity_search_with_score` function to fetch tools whose descriptions closely match the query. If the similarity score exceeds predefined thresholds, the corresponding tools are added to the result set.

In this realization, embeddings serve as a bridge between textual process descriptions and tool metadata. For example, a textual task description like "adjust machine speed to 10,000 RPM" is embedded and compared against stored tool descriptions. The similarity search may retrieve a match like the `SetSpeedDieMachine` tool, which provides the exact operation required.

Table 1 Prompt information of the LLM-based components.

	Tasks-Model extractor	Tasks pre-processing	Code generator
(a)	BPM expert	BPM expert	-
(b)	Extract the control flow in terms of a process model and the list of tasks	Rephrase the tasks descriptions	Generate a Python code
(c)	Description of BPMN elements	-	Tools descriptions and guidelines
(d)	Yes	Yes	No
(e)	Set of custom rules for the process model representation	-	Python program structure
(f)	Textual process description	Textual process description and extracted tasks	Textual process description and process model

Prompt engineering

The prompt is a guide for the model, instructing it on relevant information and desired output formatting. The quality of the LLM output directly correlates with the quality of the provided prompt [13].

The proposed approach consists of three different LLM-based components, each specialized in a particular task, i.e., extraction of the process tasks and model, pre-processing of the extracted tasks and generation of Python code. Each component is characterized by a specific prompt. Each prompt is characterized by all (or some) of the following parts: *(a)* the role the LLM plays that helps in controlling the output style [11], *(b)* a clear description of the task to be performed, *(c)* additional information (context) to aid the LLM in generating better responses, *(d)* few examples to teach the LLM, *(e)* type and format of the desired output and *(f)* the input data used by the LLM to compute the response. Table 1 presents detailed information regarding each part of the prompt for the three LLM-based components. All the prompts are available at the provided link[5].

NL2ProcessOps implementation

The implementation of NL2ProcessOps leverages LangChain to integrate LLMs into a pipeline for generating process scripts from natural language descriptions. This section explains the key components of the implementation, focusing on how tasks, tools, and code generation are orchestrated.

The `ProcessLLM` class implements the core logic of NL2ProcessOps. It integrates multiple components, including task modeling, code generation, and tool management, by leveraging LangChain's `Runnable` utilities and custom LLM modules.

The ProcessLLM class

```python
class ProcessLLM:
    def __init__(self, model="gpt-3.5-turbo", openai_key=None,
    ↪  temperature=0.0):
        self.model_tasks_llm = MermaidLLM(model, openai_key,
        ↪  temperature=temperature)
        self.task_llm = TaskRetrieverLLM(model, openai_key,
        ↪  temperature=temperature)
        self.code_llm = CodeLLM(model, openai_key,
        ↪  temperature=temperature)

        embedding_function =
        ↪  OpenAIEmbeddings(model="text-embedding-ada-002",
        ↪  api_key=openai_key)
        self.tools_store = ToolStore(openai_key)
        self.tools_store.embed_tools(embedding_function)
```

The `ProcessLLM` class initializes various components, including (i) `MermaidLLM`, which extracts process models from descriptions, (ii) `TaskRetrieverLLM`, which identifies tasks within the process, and (iii) `CodeLLM`, which generates Python code based on tasks and tools. These classes are implemented as standard LangChain pipelines, adhering to the standard chai structure of: `self.prompt | self.model | self.output_parser`. The `ProcessLLM` class also sets up ChromaDB (line 8) for semantic retrieval of the tools through the `ToolStore` class.

The `get_chain` method, which is provided below, defines the overall workflow of the `ProcessLLM` class. It orchestrates the sequence of operations using LangChain's `Runnable` utilities, including `RunnableLambda`, `RunnablePassthrough` (which directly propagates inputs without modification), and `RunnableBranch` (enabling conditional branching based on predicate functions).

The ProcessLLM chain

```python
def get_chain(self):
    model_tasks_llm_chain_output = self.model_tasks_llm_parser()
    task_llm_chain_output = self.task_llm_parser()
    code_llm_chain_output = self.code_llm_parser()

    general_chain = (
        RunnableLambda(lambda x: {
            "model": x["model"],
            "tools": self.tools_prompt_parser(x["tasks"]),
            "input": x["input"],
        })
        | code_llm_chain_output
        | RunnableBranch(
            (lambda x: not x["error_python"], RunnableLambda(lambda x:
            ↪  self.parse_output(x))),
            (lambda x: "There are some errors in the python code.")
        )
    )
```

```
18
19      chain = (
20          model_tasks_llm_chain_output
21          | task_llm_chain_output
22          | RunnableLambda(
23              lambda x: {
24                  "tasks": x["tasks"],
25                  "has_tasks": self.is_list_of_tasks(x["tasks"]),
26                  "input": x["input"],
27                  "model": x["model"],
28              }
29          )
30          | RunnableBranch(
31              (lambda x: x["has_tasks"], general_chain),
32              (lambda x: "Your process description does not contain any
                 ↪  task.")
33          )
34      )
35      return chain
```

This method begins by defining three parsing subchains for task modeling (line 2), task retrieval (line 3), and code generation (line 4). The `model_tasks_llm_parser` chain processes the input to extract tasks, the `task_llm_parser` chain retrieves additional task details, and the `code_llm_parser` chain generates Python code based on the tasks and tools. These subchains are constructed using LangChain's RunnableLambda for inline computations and RunnablePassthrough to propagate inputs.

The heart of the workflow lies in the two branches created using RunnableBranch. The first branch (lines 30–33) checks whether tasks are present in the process description. If tasks are found, the `general_chain` is executed, which:

1. Integrates task descriptions, tools, and process inputs,
2. Generates Python code using the `code_llm_parser` (line 12),
3. Validates the generated code with the second branch (lines 13–16). It either parses the generated code for output or returns an error message if issues are detected.

If no tasks are found, the branch directly outputs a message stating, "Your process description does not contain any task".

Finally, the `ProcessLM` provides a CLI-based interactive interface for users to input natural language process descriptions, generate Python code based on these descriptions, and optionally execute the generated code. Below is a simplified version of the main function:

Running NL2ProcessOps (simplified)

```
1  if __name__ == "__main__":
2      # Load environment variables and retrieve the OpenAI API key
3      dotenv.load_dotenv()
```

```
4     OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
5     model = "gpt-4-0125-preview"  # Define the LLM model to use
6
7     # Instantiate the ProcessLLM pipeline
8     llm = ProcessLLM(model, OPENAI_API_KEY)
9
10    while True:
11        # Prompt the user to input a process description
12        input_text = input("Enter a process description (or press Enter
          ↪  to skip): ")
13        if input_text == "":
14            continue
15
16        # Process the input and generate Python code
17        result = llm.get_chain().invoke({"input": input_text})
18        print(f"Generated Code:\n{result}")
19
20        # Ask if the user wants to execute the process or provide
          ↪  another description
21        user_choice = input("Execute the process? (y)\nEnter a new
          ↪  description? (p)\nQuit? (q): ")
22        if user_choice == "y":
23            try:
24                # Validate and execute the generated Python code
25                print("Executing the process...")
26                if os.system("python -m py_compile
                  ↪  llm_process_code.py") != 0:
27                    print("The generated code contains syntax errors.")
28                else:
29                    os.system("python llm_process_code.py")
30                    print("Process executed successfully.")
31            except Exception as e:
32                print(f"Error during execution: {e}")
33        elif user_choice == "p":
34            continue  # Loop back for a new process description
35        else:
36            print("Exiting...")
37            break
```

4 Concluding remarks

This chapter presented two complementary LLM-based solutions, i.e., COSMADS for intelligent data integration and NL2ProcessOps for automated business process execution, highlighting how LLMs enhance data accessibility and streamline workflow automation in real-world scenarios. By enabling seamless integration of heterogeneous data sources and dynamically generating executable workflows from natural language descriptions, these approaches demonstrate the significant impact of LLMs in bridging the gap between human input and machine execution. The presented implementation examples illustrate the practical applicability of these

methods in real-world scenarios, offering valuable insights into the development of LLM-based applications for enhancing information systems across diverse domains. In this context, challenges remain, including ensuring the reliability, interpretability, and security of LLM-generated outputs. Addressing these challenges will be key to unlocking the full potential of LLMs in information systems.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. NeurIPS 2020 (2020)
3. Klievtsova, N., Benzin, J.V., Kampik, T., Mangler, J., Rinderle-Ma, S.: Conversational process modelling: State of the art, applications, and implications in practice. In: BPM 2023 Forum (2023)
4. Klievtsova, N., Benzin, J.V., Kampik, T., Mangler, J., Rinderle-Ma, S.: Conversational process modeling: Can generative ai empower domain experts in creating and redesigning process models? arXiv preprint arXiv:2304.11065 (2024)
5. Mathew, J.G., Monti, F., Firmani, D., Leotta, F., Mandreoli, F., Mecella, M.: Composing smart data services in shop floors through large language models. In: International Conference on Service-Oriented Computing, pp. 287–296. Springer (2024)
6. Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A., et al.: Augmented language models: a survey. arXiv preprint arXiv:2302.07842 (2023)
7. Monti, F., Leotta, F., Mangler, J., Mecella, M., Rinderle-Ma, S.: NL2ProcessOps (2024). DOI 10.5281/zenodo.11219809
8. Monti, F., Leotta, F., Mangler, J., Mecella, M., Rinderle-Ma, S.: Nl2processops: Towards llm-guided code generation for process execution. In: BPM. Springer (2024)
9. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17), 3045–3054 (2004)
10. Popkova, E.G., Ragulina, Y.V., Bogoviz, A.V.: Industry 4.0: Industrial revolution of the 21st century, vol. 169. Springer (2019)
11. Shanahan, M., McDonell, K., Reynolds, L.: Role play with large language models. Nature (2023)
12. Thalmann, S., Mangler, J., Schreck, T., Huemer, C., Streit, M., Pauker, F., Weichhart, G., Schulte, S., Kittl, C., Pollak, C., et al.: Data analytics for industrial process improvement a vision paper. In: CBI 2018. IEEE (2018)
13. White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C.: A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 (2023)
14. Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al.: A survey of large language models. arXiv preprint arXiv:2303.18223 (2023)