Data Service Composition in Cyber-Physical Systems Adopting LLMs

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—The increasing adoption of Large Language Models (LLMs) has significantly lowered technical barriers across various domains, enabling tasks such as code generation, database querying, and service composition. This work explores the potential of LLMs to facilitate human access to heterogeneous data in Cyber-Physical Systems (CPS). The proposed system interprets natural language queries and, by leveraging in-context learning, dynamically synthesizes data integration pipelines that compose data services, thus enabling access to information from multiple sources. The execution of a pipeline generates a table that fulfills the input query, effectively integrating and structuring the retrieved data. This paper presents the design and implementation of the proposed solution and its evaluation using a real-world case study and the BIRD benchmark dataset. The results demonstrate its effectiveness in improving data retrieval, data service design, and execution in CPS.**

***Index Terms*—Service composition, Data services, Large Language Models, Cyber-Physical Systems**

I. Introduction

A Cyber-Physical System (CPS) is a complex integration of computational and physical processes where embedded computers and networks continuously interact with and control physical components [1]. These systems rely on sensors to collect data from the physical environment, process that data through software, and then use actuators to influence physical operations in real-time. CPSs are characterized by their ability to facilitate seamless communication between digital and physical elements, often leveraging networked communication technologies such as the Internet of Things (IoT). They frequently incorporate autonomous decision-making, using data analytics and machine learning to optimize performance and efficiency. These systems play a crucial role in various safety-critical domains, including healthcare, transportation, smart infrastructure, and industrial automation. However, a prominent feature of CPSs is the fact that data is available in the form of isolated information islands, which do not follow a common schema, often requiring data integration pipelines to provide useful information to the final users.

For example, manufacturing companies handle vast amounts of diverse data from multiple departments, including the shop floor, warehouse, and administrative offices. These data sources range from traditional information systems like Enterprise Resource Planning (ERP) systems to sensor-generated data from machines and physical spaces. Access to this data occurs through various means, such as direct database connections or Application Programming Interfaces (APIs), often delivered as Web services [2]. We refer to these access mechanisms as *smart data services* (or simply data services – DSs).

Individual data services can be combined to extract information, leveraging the integration of multiple data sources. We use the term *information extraction pipeline* (or simply *pipeline*) to refer to any computational process that *composes* new information from existing data services. Without loss of generality, the outcome of this composition can be represented using relational tables.

Despite the immense potential, the information required for decision-making on the shop floor, where value is created, changes rapidly, leaving operators with limited access to timely data. Continuously developing new pipelines to address these unpredictable data needs can be costly, particularly for non-core IT companies like manufacturing firms, which often rely on external suppliers for data management. Data governance and management platforms can help alleviate this issue to some extent, but maintaining the large back-end data lakes they require is often impractical due to a lack of expertise, human resources, and computational power. This is especially challenging for Small and Medium Enterprises (SMEs), which cannot bear the costs of such solutions [3].

On the other hand, chat-enabled digital assistants are emerging as a new form of interaction between humans and machines, attracting growing interest in the industrial sector [4]. By allowing shop floor operators to interact using natural language and providing quick, intuitive, and potentially hands-free access to data, these assistants have the potential to

streamline, accelerate, and enhance product-related activities. Large Language Models (LLMs), in particular, have recently shown impressive abilities to tackle complex questions, especially by utilizing external knowledge and tools (e.g., search engines, DBMS, etc.) to support their responses [5].

In this paper, we present a methodology, together with a prototype implementation, for synthesizing information extraction pipelines, in the form of Python scripts, starting from natural language queries and a documented *codebase* consisting of the data services available in the CPS. The methodology is first quantitatively evaluated on the BIRD [6] dataset using different LLMs for code generation. Then, we present a qualitative evaluation of an industrial case study.

This paper is organized as follows. Section II introduces background and related works. Section III introduces our real-world case study to exemplify the intended contribution. In Section IV, we discuss the main components of the proposed solution. In Section V, we provide technical details. Experimental results are presented in Section VI. Finally, an outline of future challenges together with concluding remarks are presented in Section VII

II. Background and related works

LLMs are advanced computational models with massive parameter sizes excelling in understanding and generating sentences in natural language [7]. Many are the state-of-the-art large pre-trained language models (PLMs) currently available, including BERT [8], GPT [9], [10], LLaMA [11], Mistral [12], which, however, are not domain-specific and can produce untruthful information (commonly called hallucinations), when used in specialized domains like healthcare, finance and the one presented in this work. The development of a domain-specific model requires a proper (large) dataset and significant computing power for LLM training (or fine-tuning, transfer-learning) which, however, may not always be possible due to the unavailability of such requirements. One key feature of LLMs is the In Context Learning (ICL) ability, which enables the LLMs to generate more coherent and contextually relevant responses based on a given context or prompt [10]. In this context, prompt engineering has emerged as an approach to interact with LLMs, where the users design specific prompt text, sometimes following specific templates [13], to guide LLMs in generating desired responses [14], [15]. In the following, we review LLM literature relevant to this work.

LLM Agent. An emerging approach to solving complex problems and mitigating hallucinations is the development of LLMs as *agents*, also referred to as tool-augmented LLMs or LLM Agents, which can generate reasoning plans involving the invocation of external services (tools) [5], [16]–[18]. These tools can access external information (e.g., documents, calendars) or affect the virtual or physical world (e.g., robotic arm) [19]. The general technique leverages ICL by including in the prompt *(i)* tool demonstrations, *(ii)* tool documentation, or *(iii)* even a combination of both [5]. Tool demonstrations consist of providing examples that show how a tool can be used to accomplish certain tasks in combination with other tools. On the other hand, tool documentation contains general usage details about what the tool can be used for and how the LLM can interact with the tool. In this paper, we adopt the LLM Agent concept to develop our proposed approach. A key distinction from existing works is that the tools, which in our case are the data services, represent cyber-physical assets rather than general-purpose tools (e.g. search engines). These assets operate within a cyber-physical system, leading to potential dependencies that must be considered when formulating and executing the reasoning plan. Additionally, another significant difference lies in the generated output. While the above works can produce various types of textual data, our focus is on generating structured (i.e. relational) data.

Code generation. LLMs are widely employed in software engineering tasks such as code comprehension and generation due to the similarities between natural language and programming languages [7], [20], [21]. In this context, an LLM is given a natural language description of a problem and generates an executable code in a specific programming language to solve it. The impressive performance of LLM has also motivated the research on their usage for generating code that encodes the necessary logic to answer a given question [22]. Recently, a variety of code-generating LLMs have been introduced, including OpenAI Codex [23], Deepmind AlphaCode [21], Codegen [24] and CodeT [25]. These models benefit from the availability of vast amounts of code-related datasets that are publicly available including GitHub and Stack Overflow. Leveraging these models, many LLM-powered tools have also been developed for integration into Integrated Development Environments (IDEs) to assist programmers in their daily tasks, such as GitHub Copilot[1], Amazon CodeWhisperer[2], Tabnine[3], and Cursor[4]. Advanced LLM techniques such as the usage of the LLM agent paradigm and the self-reflection [26] have further demonstrated the improvement of the accuracy of these models on code generation [27]. In this paper, we compare the proposed solution with GitHub Copilot, showing that our approach has substantial advantages with respect to plain code generation.

Data management. LLMs are rapidly transforming data management [28], with a wide range of methods emerging for querying relational databases (also called Text-to-SQL techniques) [29]–[31], reasoning over structured data [32]–[34] and performing data cleaning operations [35], [36]. While these approaches primarily focus on querying relational databases with a known schema, our goal is to extract data, in form of tables, from independent data sources where a schema is not available. The lack of a schema complicates the integration as the relationship between data sources is not explicit.

A potentially related, though completely different, task is to query the LLM knowledge base by using SQL queries [37].

[1]https://github.com/features/copilot
[2]https://aws.amazon.com/q/developer
[3]https://www.tabnine.com
[4]https://www.cursor.com

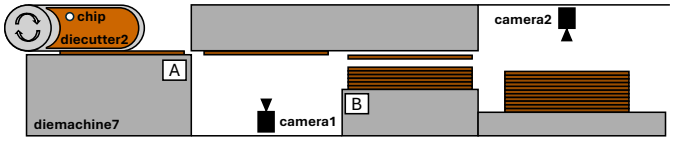

Fig. 1: The cardboard manufacturing case study

Our goal differs in this case as *(i)* our knowledge base is external to the LLM and *(ii)* we do not make use at any level of SQL as an intermediate language.

Service composition. As explained in [38], service composition addresses the situation when a complex task, also called *target service*, cannot be realized by calling a single service, but a composite service, obtained by combining "parts of" available *component services*, might be used. This definition somewhat aligns with the definition of LLM Agent which leverages external tools to derive an output.

Classical service composition approaches present several limitations like the manual definition of the target and component services and then computation complexity that is implicit when symbolic reasoning is involved. As expressed in [39], [40], LLMs can help address many of these limitations, although expert supervision is still necessary due to potential inaccuracies. Authors in [41], for example, propose to use LLMs to select and call single services to address specific needs. However, an LLM Agent, which leverages external tools, can be thought of as a service composition engine to integrate information from several different data services.

The authors in [42] implement a human-in-the-loop approach engaging an LLM Agent and employing few-shot techniques for prompt definition to compose services into an executable code script that produces structured data. However, the approach heavily relies on examples provided by humans and the experimental evaluation is limited.

Authors in [43] take advantage of this ability to develop a solution where an LLM Agent composes external tools able to perform process activities, resulting in code representing a business process (a.k.a. orchestration). Similarly, authors in [44] proposed an LLM-based solution that generates service compositions based on the development of a prompt containing semi-structured service documentation. However, the focus in these last two works is more on the synthesis of automatized processes where steps represent action, whereas the data aspect, central to our work, is neglected.

III. Real-world case study

We consider a real-world case study in the smart manufacturing domain related to a cardboard box factory. Smart manufacturing represents a recent trend where advanced technologies, such as Artificial Intelligence (AI) and Industrial IoT (IIoT), come into play to increase the efficiency and agility of traditional manufacturing processes [45]. In this domain, the amount of data produced and consumed is huge and originates from disparate sources.

Figure 1 illustrates the main elements of the manufacturing process in a cardboard box factory. The core production involves two key machines: A a *die cutting machine* (`diemachine7` in the figure), which cuts the cardboard using a *die cutting tools* (`diecutter2` in the figure), and B a *stacker*, which organizes the cut-out cardboard into bunches. The process is monitored through different sensors: *(i)* each die cutting tool is equipped with a `chip` that computes the speed as rotations per second, *(ii)* a high-resolution camera (`camera1` in the figure) captures the printed side of the produced cardboards and identifies defective pieces, and *(iii)* a second high-resolution camera (`camera2` in the figure) records the non-printed side of the produced bunches of cardboard and identifies defective bunches.

In such a scenario, we have 12 data services that wrap different data sources such as cameras, chips, and databases. Specifically, camera-related data services provide information such as captured cardboard frames and defect analysis results, chip-related data services provide data regarding the actual functioning of the die machine, and database-related data services offer information about the factory floor, including equipment, components, and employees.

Let us consider the following natural language query **Q** in the context of this case study.

> **Q**: *"Generate a table presenting details on the ongoing operation of diecutter 2. The table should include for each of the next 5 bunches: (i) error presence, (ii) specific error types if applicable, (iii) current operating speed, and (iv) current operating temperature of the die cutting tool."*

The aim of the proposed methodology is to realize a pipeline producing a relational table that satisfies the query **Q** by composing available data services and treating the data sources as external knowledge that can be queried and processed via the available data services.

In particular, among the available data services, the pipeline must integrate: `DS1`, returning the id of the camera streaming the production of a die cutting tool; `DS2`, capturing a frame of a bunch of cardboard; `DS3`, returning the defects identified on a bunch; `DS4`, returning the id of the chip installed on the die cutting tool; and `DS5`, returning the operational data from the chip installed on the die cutting tool. To answer query **Q**: *(i)* `DS1` and *(ii)* `DS4` are initially invoked to retrieve the id of the camera (e.g., 2564) and the id of the chip (e.g., 9231532) streaming the production of die cutting tool 2, *(iii)* `DS2` captures a frame of bunch of cardboard from camera 2564, *(iii)* `DS3` detects possible defects in the captured frame, and *(iv)* `DS5` tracks the current speed and temperature from the chip 9231532. An example of the output data generated from query **Q** is reported in Table I[5].

IV. System Architecture

The proposed approach heavily relies on LLMs and leverages LLM Agents and tools to streamline the generation of

[5]The relative generate pipeline is available at https://anonymous.4open.science/r/F8B6/src/ground_truth/in_action/q4/pipeline.py

TABLE I: Output example of query **Q**

bunch	has_error	error_type	speed	temperature
1	True	Missing hole	12000	28.7
2	False	–	11992	28.8
3	True	Measure	11999	28.8
4	True	Measure	12016	28.7
5	False	–	12021	28.6

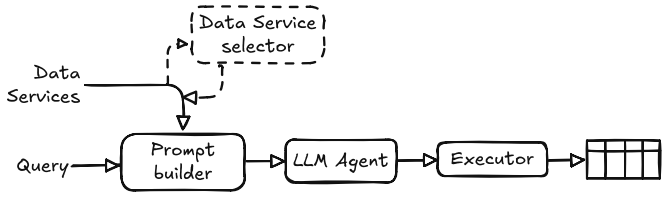

Fig. 2: Architecture of the proposed solution

structured data from a natural language query.

We assume a CPS provides data services ranging from operational/actuating services (e.g., turning the camera on and off) to services that generate data (e.g., retrieving the current rotation speed of the die machine from the embedded chip). These data services can be accessed using different paradigms and communication protocols. In our case, we assume they can be invoked through function calls that wrap the actual calling mechanism. Specifically, a data service can expect a set of parameters and return an output, which may be either structured or unstructured. Furthermore, we assume each data service has a related *documentation*, i.e., a textual description of the functioning and usage of a data service, including details of its parameters and the resulting output. We expect such documentation, which can be derived from standard specifications like OpenAPI, WSDL, and gRPC, to be put in the devised prompt.

In general, a data service retrieves historical or real-time data and can be combined with other data services within a pipeline to generate new data based on an input query. Real-time data is often associated with executing a specific operation (e.g., capturing a frame from a camera sensor). Consequently, any company asset that exposes services supporting operational functionalities can be considered a data service. In the example case study, one such data service is the camera asset, which captures frames (i.e., `DS2`).

As illustrated in Figure 2, the architecture of the proposed approach consists of four main components: the *Prompt builder*, the *LLM Agent*, the *Executor*, and an additional component, the *Data Service selector*.

The workflow begins with a natural language query, which is provided as input to the *Prompt builder*. This component is responsible for structuring the prompt that will be passed to the *LLM Agent*. The prompt is organized according to a specific prompt template[6]. The key part of the prompt includes the documentation of the data services that are used to answer the input query.

Once the prompt is constructed, it is fed to the *LLM Agent*, which generates a pipeline fulfilling the input query. Pipelines, that utilize data services to produce structured data outputs, such as tables, can be defined using different modeling formalisms, including programming languages or scientific workflow scripting languages [46]. The generated pipeline is then executed by the *Executor*, which can be an interpreter for the chosen formalism. This execution produces a table containing the data answering the query.

An additional component, the *DS selector*, can be employed during the workflow execution. This component operates at the beginning of the process. It takes the natural language query as input and selects only the relevant data services from the full set available, which are necessary to answer the query. The output from the *DS selector* is then passed to the *Prompt builder*, ensuring that only the documentation for the required data services is included in the prompt.

V. Implementation details

In this section, we introduce a prototype of the system described in Section IV and outline the implementation details of its components. The prototype is implemented in Python, using the Langchain framework[7]. The system is designed to facilitate modular integration of different LLM models. New models or updated versions of pre-existing models can be easily integrated provided that they are supported by Langchain. The specific models currently supported are detailed in the evaluation section (cf. Section VI). Anyway, the prototype can be easily extended with the various models integrated with Langchain. For transparency and reproducibility, both the source code and the experimental results are publicly available at https://anonymous.4open.science/r/F8B6.

A. Prompt builder

The prompt builder is responsible for formatting the contextual information required by the *LLM Agent* into a prompt according to a template in order to generate the pipeline. The template includes *(i)* the input query, *(ii)* the available data services, *(iii)* optional sample data from each data service, and *(iv)* additional contextual or domain-specific information, referred to as *evidence*, which may help the LLM in constructing the pipeline. For example, an evidence might specify the timestamp format used by a particular data service or describe the data types associated with certain values.

These components are structured within a *prompt template*, which provides explicit instructions for the *LLM Agent*, adhering to widely recognized best practices [47]. An overview of the prompt template is shown in Figure 3. The blue highlighted sections denote fixed elements of the prompt, including *(i)* a system header outlining the agent's capabilities, *(ii)* a description of the target task, *(iii)* the format specifications for data service documentation, data samples, and evidence, and *(iv)* a set of guidelines that the agent's output must adhere to.

[6]For a detailed description of the prompt template, refer to Section V-A.

[7]Cf. https://www.langchain.com/

Prompt template

```
<LLM AGENT EXPERTISE>
Query: {query}
<GOAL DESCRIPTION>
{data_services}
<DATA SERVICES DOCUMENTATIONS STRUCTURE>
evidence
<EVIDENCE STRUCTURE>
data_samples
DATA SAMPLES STRUCTURE
<GUIDELINES>
Answer:
```

Fig. 3: Overview of the prompt template

The yellow highlighted sections represent query-dependent elements, such as the set of data services available for integration and the corresponding sample data. The red highlighted sections indicate the primary input data, namely the natural language query and any optional evidence to be incorporated into the prompt.

B. Data Service selector

Each data service is implemented as a Python class[8], consisting of two main components: *(i)* a callable function that wraps the existing service of the asset and encapsulates the execution logic, enabling operations such as collecting frames from `DS2`, and *(ii)* a class variable storing the service documentation. Notably, this documentation corresponds to what is traditionally referred to as a service description in service composition.

At its core, the documentation is structured as a JSON object containing *(i)* a detailed description of the functionality provided by the data service, *(ii)* a list of input parameters along with their data types, *(iii)* the expected output type of the data service call, and *(iv)* a one-shot example demonstrating how to invoke the data service.

In practice, the number of available data services typically exceeds those needed to resolve a given query. For instance, in the case study presented in Section I, only five data services are required to answer query **Q**, out of the twelve available. To prevent the *LLM Agent* from being overloaded with irrelevant data services, as outlined in Section IV, our system incorporates an optional component, the *Data Service selector*. The *Data Service selector* is implemented using an LLM, which is prompted with *(i)* the input query, *(ii)* the complete list of available data services, and *(iii)* the same evidence used in the *LLM Agent* prompt template. Based on this input, the LLM returns the subset of data services necessary to answer the query.

[8]An exemplary data service documentation can be found in https://anonymous.4open.science/r/F8B6/src/data_services/camera1.py

C. LLM Agent and pipeline execution

Our *LLM Agent* uses the In Context Learning capability of LLMs [10] to generate a pipeline. The pipeline itself is structured as a Python script, where data services are treated as libraries that are imported and invoked to interact with and retrieve data. Once the prompt builder gathers the required information and fills the prompt template, the resulting prompt is passed to the *LLM Agent* to generate a pipeline that answers the query. However, in general, the quality of the generated output heavily depends on both the quality of the provided prompt [13] and the base LLM model used for the *LLM Agent*.

After generating the Python script, additional processing is performed to address potential syntax errors when calling data services. Specifically, we extract the data service invocations from the generated code and verify whether the provided parameters match the expected input of the corresponding data services. This is achieved by employing the embeddings generated by the BERT model [8] to compare the parameter names used in the service calls with the set of valid parameter names extracted from the data service documentation. If a mismatch is detected, the incorrect parameter name is replaced with the closest matching one, provided it scores above a predefined similarity threshold.

Once refined, the final Python script is stored in a temporary file, executed using the Python interpreter, and the resulting tabular output is collected and presented to the user.

VI. Experimental validation

A. Datasets

We evaluate our approach in two distinct settings: *(i)* an *benchmark* setting, where data services are automatically generated using an LLM, and *(ii)* an *in-action* setting, which consists of manually curated data services and queries. The benchmark setting allows for a large-scale quantitative evaluation across a diverse range of queries and data services, while the in-action setting is designed to reflect real-world applications by carefully constructing queries and data services that align with practical use cases.

Benchmark setting. We use BIRD [6], a widely used *benchmark* for evaluating text-to-SQL models. The BIRD dataset comprises 12.751 natural language questions, each paired with its corresponding SQL query, covering 95 databases across 37 domains (e.g., healthcare, education, hockey). The dataset is divided into a training split and a development split; for our experiments, we focus on the training split.

As remarked in Section II, our task is significantly different from text-to-SQL. However, the BIRD dataset offers an unmatched amount of natural language queries with the corresponding SQL query, which can be used as a gold standard to evaluate our method.

Each database in the training split consists of a variable number of tables (ranging from 2 to 65) and is associated with *(i)* a set of natural language questions (ranging from 6 to 458), *(ii)* ground truth SQL queries corresponding to each question, and *(iii)* an *evidence* field containing additional information for

query disambiguation, such as column types and formatting details.

To adapt BIRD-SQL to our task, we treat each database as a collection of data services, where each table is wrapped as an individual data service. This means we are completely ignoring interrelational constraints, which are instead fundamental in text-to-SQL. These data services are designed to accept a set of column names and SQL-like operators (e.g., `EQUAL`, `GREATER`, etc.) that define the subset of rows and columns to retrieve. Upon receiving these parameters, the data service constructs and executes an SQL query to extract the relevant tuples from the table.

The data services and their corresponding documentation are generated automatically by prompting an LLM with *(i)* the table schema, *(ii)* related tables identified through foreign keys, *(iii)* instructions on structuring the wrapper function, and *(iv)* formatting guidelines for the data service documentation. The LLM returns a Python module implementing the data service, enabling direct access to the table.

For our experiments, we select a subset of the BIRD-SQL training split, consisting of 27 databases with a diverse range of tables per database (from 2 to 34). From each database, we sample up to 25 random queries for evaluation, totaling 648 queries.

In-action. We evaluate our approach using a set of data services adapted from a proprietary codebase deployed in a real-world cardboard manufacturing environment, as part of a smart manufacturing pilot project. For the purpose of our experiments, these data services have been simplified by simulating interactions with physical assets rather than directly interfacing with industrial devices. In this setting, we consider the 12 data services described in Section III, which capture the functionalities and operations of the physical assets used in the cardboard box manufacturing case study.

To assess system performance, we define a set of 5 manually curated queries (`q0`-`q4`), each paired with a manually designed ground truth pipeline. These queries, similar to query **Q** in Section III, were originally suggested by operators working in the manufacturing factories where the proprietary codebase is deployed. For each query, we generate 10 additional variants by rephrasing the original query using an LLM, resulting in a total of 50 queries[9].

The queries are structured to progressively increase in complexity based on how data services must be composed to compute the final output. For instance, `q0` represents the simplest query, requiring interactions with only three data services, whereas `q4` involves a more intricate pipeline that integrates six data services to produce the final response.

B. Metrics

We evaluate the two main stages of our approach separately: the data service retrieval stage, handled by the *Data Service selector*, and the pipeline generation stage, performed by the LLM Agent. Each stage is assessed using a distinct set of metrics.

Data service retrieval. To evaluate the effectiveness of the *Data Service selector*, we use *recall* and *precision*, two standard metrics in the information retrieval literature. Given a query q, the set of necessary data services D^* required to resolve q, and the set of retrieved data services $\hat{D}$ returned by the *Data Service selector*, we define (data service) recall and (data service) precision as follows:

$$recall_{DS} = \frac{|\hat{D} \cap D^*|}{|D^*|}, \quad precision_{DS} = \frac{|\hat{D} \cap D^*|}{|\hat{D}|}$$

We use $recall_{DS}$ to measure the proportion of required data services successfully retrieved for a given query, while $precision_{DS}$ quantifies the proportion of retrieved data services that are actually relevant, indicating how many unnecessary data services were included in the selection. While both metrics are important, achieving $recall_{DS} = 1$ is crucial, as missing even a single relevant data service can prevent the pipeline from retrieving all necessary information, leading to an incomplete or incorrect output table.

Pipeline generation. To evaluate the quality of the pipelines generated by the LLM Agent, we compare the resulting table with the ground truth table. In the benchmark setting, the ground truth table is obtained by executing the SQL query associated with each natural language query. In the in-action setting, the ground truth table corresponds to the output produced by the manually designed pipeline for each query.

The evaluation is conducted at two levels: *intentional*, where we compare the schema of the generated table to the ground truth schema, and *extensional*, where we assess the row-level content of both tables.

At the intentional level, we evaluate the generated schema against the ground truth schema, which represents the ideal relational structure the system should produce. For this, we employ the Valentine tool[10] [48] for schema matching, using the COMA instance-based method [49]. We compute $precision_{schema}$ and $recall_{schema}$ to assess matching quality. A high schema precision value indicates a low false positive rate, meaning the generated table contains only relevant columns. A high schema recall value suggests a low false negative rate, indicating that most of the expected columns are correctly included. Since different queries may result in schema variations due to rewording, the Valentine COMA instance-based method accounts for such differences.

At the extensional level, we measure *instance accuracy*, which evaluates whether the generated table contains the same data as the ground truth. Specifically, we compute the fraction of rows in the generated table that match those in the ground truth table, denoted as $accuracy_{row}$. Notably, instance accuracy is computed only on matched columns, meaning that a high accuracy score can still be achieved even if schema precision and recall are low, provided that the correctly matched columns contain high-quality data.

[9]The full list of evaluated queries is provided at https://anonymous.4open.science/r/F8B6/src/queries/in_action_queries.json

[10]Cf. https://github.com/delftdata/valentine

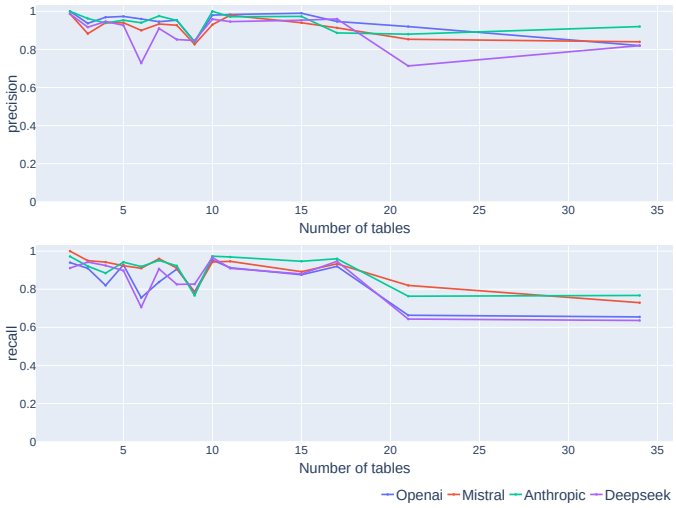

Fig. 4: *Data Service selector* performance on BIRD

Evaluation methodology. We assess the performance of our system using four different models as the base LLM in the *LLM Agent* component: OpenAI GPT-4o, Mistral Large, Anthropic Claude 3.5, and Deepseek Chat.

For the benchmark setting, we use the *evidence* field provided in BIRD to populate the corresponding placeholder in the prompt template (see Figure 3). In contrast, no additional evidence is provided to the *LLM Agent* in the in-action setting. Additionally, in the benchmark setting, we employ GPT-4o to automatically generate the documentation for the data services.

The evaluation of the data service retrieval stage is conducted exclusively in the quantitative setting, as it involves multiple databases, each containing a varying number of data services. This enables us to analyze the impact of the total number of data services on retrieval performance in terms of $recall_{DS}$ and $precision_{DS}$.

Pipeline generation is evaluated in both settings. Each LLM is provided with the prompt template described in Section V-A to generate the pipeline. The resulting pipeline is then executed, and the output is compared against the ground truth to assess it.

C. Experimental results (quantitative)

Data service retrieval. We present the epxerimental results in Figure 4. The reported $precision_{DS}$ and $recall_{DS}$ values are averaged across queries associated with the same database and further aggregated by grouping databases with the same number of tables (i.e., data services). Overall, for all models, we observe a general trend where increasing number of tables leads to a decrease in both $precision_{DS}$ and $recall_{DS}$. Upon further analysis, we identify three primary causes of low performance: (i) selecting data services with typos or formatting inconsistencies (e.g., referring to a data service as `player_module` instead of `Player`), (ii) retrieving non-relevant services that resemble relevant ones, often due to shared column values from foreign key relationships, resulting in reduced precision (e.g., the first row in Table II), and (iii) failing to retrieve essential data services needed to resolve the query, leading to lower recall (e.g., the second row in Table II).

This suggests that while the *Data Service selector* performs well when filtering from a smaller set of data services, maintaining accuracy becomes more challenging as the number of data services increases. In such cases, prioritizing recall over precision is essential to ensure that all relevant data services are retrieved, even at the cost of including some non-relevant ones.

Among the evaluated models, Deepseek Chat exhibits greater fluctuations in $precision_{DS}$ compared to the others, which demonstrate more stable results ranging between 0.8 and 1.0. Claude 3.5 and GPT-4o perform slightly better than Mistral Large in this regard.

In terms of $recall_{DS}$, similar fluctuations are observed across different dataset sizes for all models. However, Claude 3.5 and Mistral Large consistently outperform GPT-4o and Deepseek Chat, particularly in databases with a larger number of tables, where the performance gap becomes more pronounced.

Pipeline generation. Figure 5 presents the performance of the pipeline generation stage for each LLM across different BIRD databases, sorted by increasing number of tables. The results are averages across queries related to the same database.

Unlike the *Data Service selector*, for this task, increasing the number of tables, i.e., data services in the bench mark setting, provided to the *LLM Agent* does not necessarily degrade performance. Instead, we observe mixed results, with some models performing better on databases with a higher number of tables compared to others with fewer tables. For instance, on the `law_episode` database, which contains 6 tables, all models performed worse across all metrics compared to the `books` database, which consists of 15 tables.

Overall, $precision_{schema}$ and $recall_{schema}$ exhibit significant variation across databases and models, ranging from a minimum of 0.08 for both metrics (Deepseek Chat on `public_review_platform`) to a maximum of 0.96 for $precision_{schema}$ and 0.91 for $recall_{schema}$ (Claude 3.5 on the `human_resources` database). Similarly, $accuracy_{row}$ also shows considerable variability, with values ranging from 0.08 (Deepseek Chat on `public_review_platform`) to 0.83 (GPT-4o on `craftbeer`).

The drop in performance across all metrics can be attributed to two main factors: (i) errors in pipeline generation, resulting in empty tables and leading to a score of zero across all metrics, and (ii) variations in data arrangement within the generated tables, which can over-penalize actual performance. Regarding the first issue, we note that while syntax errors were relatively infrequent (generally around 10% across most datasets), they led to empty or null tables, resulting in a score of zero across all metrics and thus impacting overall performance.

An example of the second issue is illustrated in Figure 6, which shows a query from the `chicago_crime` database alongside the corresponding ground truth table and the table generated by our system. In this case, the Valentine tool cor-

TABLE II: Example queries from `soccer_2016` where the *Data Service selector* exhibits low $recall_{DS}$ or $precision_{DS}$

Query	Ground truth DS	Retrieved DS	Recall	Precision
"Count the matches with a total of two innings."	WicketTaken	BatsmanScored, WicketTaken, BallbyBall	1	0.33
"Which country is umpire TH Wijewardene from?"	Umpire, country	Umpire	0.5	1

Fig. 5: Performance of the *LLM Agent* across different databases (sorted by increasing number of tables) in the BIRD benchmark

rectly matches the `district_name` column in both tables, resulting in $precision_{schema} = 1$ and $accuracy_{row} = 1$. However, the absence of the `district_no` column in the generated table, despite not being explicitly required by the query, causes $recall_{schema}$ to drop to 0.5.

Another interesting example of this issue is shown in Figure 7. For this query, the `SELECT` clause of the ground truth SQL query contains a `SUM` operator but since no aliases is specified for the aggreagated value, the entire aggregation expression is used as the column name. The Valentine tool matches this column with the `community_area_name`, resulting in a $recall_{schema} = 1$. When computing $accuracy_{row}$, we only consider the matching columns between the two tables, which results in an $accuracy_{row} = 0$ despite the generated table contains the data that the query asks for.

Finally, while no single model consistently outperforms the others across all databases, some models tend to perform better or worse than others depending on the number of tables in the database.

To highlight this, we ranked the models for each database and each metric, assigning a rank of 1 to the best-performing model and 4 to the worst. We then averaged these rankings

Query: "At which district did the multiple homicide case number JB120039 occur?"

(a) Exemplary query from `chicago_crime`

district_no	district_name
3	Grand Crossing

(b) Ground truth table

district_name
Grand Crossing

(c) Output from our approach

Fig. 6: Exemplary query and results from `chicago_crime` resulting in low $recall_{schema}$

Query: "Among the crimes in Woodlawn, how many of them happened in January, 2018?"

(a) Exemplary query from `chicago_crime`

SUM(CASE WHEN T1.community_area_name = 'Woodlawn' THEN 1 ELSE 0 END)
444

(b) Ground truth table

community_area_name	crime_count	date_range
Woodlawn	444	January 2018

(c) Output from our approach

Fig. 7: Exemplary query and results from `chicago_crime` with $accuracy_{row} = 0$

across databases with the same number of tables. The results are shown in Figure 8.

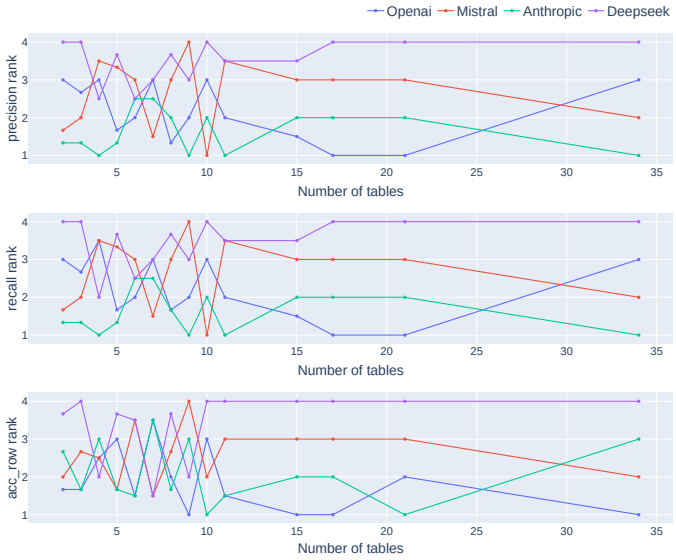

Fig. 8: Average ranking of models across databases in the BIRD benchmark

In terms of $recall_{DS}$ and $precision_{DS}$, Claude 3.5 generally achieves the best performance for databases with up to five tables. However, within this same range, results for $accuracy_{row}$ are more mixed, with no clear leading model.

For databases with a larger number of tables (10 or more), the rankings reveal clearer patterns across all metrics. GPT-4o and Claude 3.5 consistently among the best-performing models, while Deepseek Chat consistently ranks lowest across all metrics. Interestingly, for the largest database in our experimental setting (with 34 tables), Mistral ranks as the second-best model. However, since this is the only database of its size, this result may not be fully representative.

D. Experimental results (in-action)

For this setting, we compare our system against an end-to-end baseline using a code-generating LLM, namely GitHub Copilot, which is prompted to access the documentation of the data services and generate a JSON file containing the tabular

Fig. 9: Performance of the *LLM Agent* in the in-action setting

data as a response to the query. Copilot is utilized via the chat mode in Visual Studio Code[11]. Within a workspace containing the set of implemented tools, we invoke the `@workspace` agent, which employs a meta-prompt to determine the relevant information needed to answer the query. The results for this setting are presented in Figure 9.

Across all models, we observe that simpler queries (`q0`-`q1`) yield high and comparable performance across all metrics.

However, for more complex queries (`q2`, `q3`, and `q4`), the results become more varied. On `q2`, all models achieve high $recall_{schema}$ and $precision_{schema}$, except for GPT-4o, which exhibits poor performance in $recall_{schema}$, performing worse than GitHub Copilot. Nevertheless, in terms of $accuracy_{row}$, all models outperform the baseline, with Claude 3.5 achieving the best results despite performance variations across models.

On `q3`, GPT-4o, Claude 3.5, and Mistral Large produce accurate results, consistently outperforming GitHub Copilot across all metrics. In contrast, Deepseek Chat frequently generates pipelines with syntax errors, leading to zero scores across all metrics for this query.

[11]Cf. https://code.visualstudio.com/docs/copilot/overview.

For $\mathtt{q4}$, Mistral Large emerges as the best-performing model across all metrics, despite a relatively low score in $accuracy_{row}$. Deepseek Chat provides accurate results in terms of $precision_{schema}$, performing comparably to Mistral Large, but struggles on other metrics. The remaining models perform poorly overall. GitHub Copilot and GPT-4o achieve similar scores in $recall_{schema}$ and $precision_{schema}$, while Claude 3.5 records the lowest performance among all models.

E. Discussion

Overall, the *Data Service selector* component demonstrated strong performance when dealing with smaller sets of data services (up to 15), with a gradual decline in accuracy as the number of data services increased. As highlighted in the previous section, optimizing recall is crucial for improving accuracy, since missing even a single relevant data service can cause the pipeline generated by the *LLM Agent* to fail or produce incorrect results. Another challenge with larger sets of data services is the limited context window of LLMs. While most state-of-the-art models support context windows exceeding 100k tokens, they may still struggle with thousands of data services. A potential solution could involve a RAG approach, where an initial filtering step using embeddings is followed by a refined selection using an LLM.

For the *LLM Agent*, results in the quantitative setting were mixed. For some databases we observed consistently accurate results across all models, while others posed challenges, with some or all models struggling. Besides errors introduced by our approach (e.g. syntax errors leading to empty pipelines), part of this variability stems from the nature of the BIRD benchmark, which is designed for text-to-SQL tasks rather than our specific use case. As illustrated in Figure 6 and Figure 7, we observed cases exist where our system produces correct results yet receives low performance scores due to differences in schema alignment. A possible mitigation strategy could involve manually or automatically generating queries with multiple ground truth tables to account for these nuances or refining our evaluation metrics to better accommodate variations in column arrangements and value ordering.

Finally, we observe that our system produces far more accurate results in the in-action setting compared to the quantitative setting. This is partly because the queries in the in-action setting were manually designed to reflect real-world applications, whereas BIRD is a text-to-SQL benchmark. Additionally, the choice of the underlying model in the *LLM Agent* plays a crucial role, particularly for complex queries. This may be due to the fact that we used the same prompt for all models; tuning the prompt for each model based on their specific failure cases could potentially yield more consistent performance, an aspect we plan to explore in future work.

VII. Conclusions and Future Works

This paper presents a tool that processes queries from a CPS environment and automatically generates a Python script. This script utilizes existing data sources, accessible as services, to produce a table that meets the user's informational needs. The proposed approach is first evaluated quantitatively using the BIRD dataset, yielding promising results, especially taking into account the rapid advancements in LLMs, which continuously improve their reasoning capabilities. Also, results are influenced by the fact that the BIRD dataset is originally intended for a different task, and queries suffer from the inherent ambiguity of natural language query, which is eased in text-to-SQL by the availability of a data schema. Additionally, we applied our methodology to a real-world industrial case study, demonstrating that incorporating available data service descriptions in a structured prompt leads to better outcomes than simply using a state-of-the-art code generation tool. However, the possible use of such tools in production scenarios requires some considerations resulting in several future research challenges.

First, the quality of the results heavily depends on the accuracy and completeness of data service documentation. Therefore, significant effort must be dedicated to crafting documentation that the *LLM Agent* can effectively utilize. Similar challenges have arisen in traditional service composition, which requires semantic service descriptions. While automatic code documentation techniques [50] could help address this issue, ensuring that the *LLM Agent* correctly interprets and applies data services based on their documentation and demonstrations remains an open challenge. In this context, *profiling* data services could be beneficial. This could involve allowing the LLM to generate hypothetical usage scenarios, learn from failures, similar to the approach in [51], or incorporating a human-in-the-loop strategy, where users define example use cases and refine the documentation as needed.

Building on the previous point, the proposed approach aims to generate an entire pipeline in a single step, similar to other methods such as [52]. An alternative strategy is to construct the pipeline iteratively using paradigms like Chain of Thought [53]. Both approaches can be further enhanced through *self-reflection*-based techniques [54]–[56], which enable the *LLM Agent* to analyze its own output and refine it incrementally. Additionally, generating a preview of the final table can be beneficial in two key ways: *(i)* identifying potential syntax and runtime errors that might obstruct table generation, and *(ii)* allowing users to spot missing or incomplete information in the table without requiring direct access to the pipeline.

Finally, our system executes the generated pipeline immediately after its creation. While this is generally safe, errors in the pipeline, whether partial or complete, could result in inadequate monitoring of manufacturing sessions. To mitigate this risk, a pipeline simulation strategy could be beneficial. One potential approach is the use of *digital twins*, which are virtual replicas of industrial assets. Digital twins have been employed both to enhance product development and to monitor the performance of physical assets [57].

REFERENCES

[1] W. Wolf, "Cyber-physical systems," *Computer*, vol. 42, no. 03, pp. 88–89, 2009.

[2] X. Zhou, Z. Feng, S. Chen, X. Xue, and H. Wu, "Governance of data service marketplace under service ecosystem," in *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 2024, pp. 65–72.

[3] G. Yilmaz, K. Qurban, J. Kaiser, and D. McFarlane, "Cost-effective digital transformation of smes through low-cost digital solutions," *LoDiSA*, 2023.

[4] S. Colabianchi, A. Tedeschi, and F. Costantino, "Human-technology integration with industrial conversational agents: A conceptual architecture and a taxonomy for manufacturing," *JIII*, 2023.

[5] C.-Y. Hsieh, S.-A. Chen, C.-L. Li, Y. Fujii, A. Ratner, C.-Y. Lee, R. Krishna, and T. Pfister, "Tool documentation enables zero-shot tool-usage with large language models," *arXiv preprint arXiv:2308.00675*, 2023.

[6] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo *et al.*, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[7] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *TIST*, 2023.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[9] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *NeurIPS 2020*, 2020.

[11] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[12] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[13] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[14] D. Dai, Y. Sun, L. Dong, Y. Hao, Z. Sui, and F. Wei, "Why can gpt learn in-context? language models secretly perform gradient descent as meta optimizers," *arXiv preprint arXiv:2212.10559*, 2022.

[15] H. Zhao, H. Chen, F. Yang, N. Liu, H. Deng, H. Cai, S. Wang, D. Yin, and M. Du, "Explainability for large language models: A survey," *ACM TIST*, 2023.

[16] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *ICML*, 2022.

[17] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *arXiv preprint arXiv:2305.15334*, 2023.

[18] A. Parisi, Y. Zhao, and N. Fiedel, "Talm: Tool augmented language models," *arXiv preprint arXiv:2205.12255*, 2022.

[19] G. Mialon, R. Dessì, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Rozière, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz *et al.*, "Augmented language models: a survey," *arXiv preprint arXiv:2302.07842*, 2023.

[20] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of ESEC/FSE 2021*, 2021.

[21] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[22] W. Chen, X. Ma, X. Wang, and W. W. Cohen, "Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks," *arXiv preprint arXiv:2211.12588*, 2022.

[23] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[24] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[25] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," in *ICLR*, 2022.

[26] M. Renze and E. Guven, "Self-reflection in llm agents: Effects on problem-solving performance," *arXiv preprint arXiv:2405.06682*, 2024.

[27] T. Ridnik, D. Kredo, and I. Friedman, "Code generation with alphacodium: From prompt engineering to flow engineering," *arXiv preprint arXiv:2401.08500*, 2024.

[28] A. Quamar, V. Efthymiou, C. Lei, F. Özcan *et al.*, "Natural language interfaces to data," *Foundations and Trends® in Databases*, vol. 11, no. 4, pp. 319–414, 2022.

[29] Z. Gu, J. Fan, N. Tang, L. Cao, B. Jia, S. Madden, and X. Du, "Few-shot text-to-sql translation using structure and content prompt learning," *PACMMOD*, vol. 1, no. 2, 2023.

[30] G. Katsogiannis-Meimarakis and G. Koutrika, "A survey on deep learning approaches for text-to-sql," *VLDB J.*, vol. 32, no. 4, pp. 905–936, 2023.

[31] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G. T. Kakkar, Y. Gan, A. Saberi, F. Ozcan, and S. O. Arik, "Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql," *arXiv preprint arXiv:2410.01943*, 2024.

[32] J. Jiang, K. Zhou, Z. Dong, K. Ye, W. X. Zhao, and J.-R. Wen, "Structgpt: A general framework for large language model to reason over structured data," *arXiv preprint arXiv:2305.09645*, 2023.

[33] Z. Wang, H. Zhang, C.-L. Li, J. M. Eisenschlos, V. Perot, Z. Wang, L. Miculicich, Y. Fujii, J. Shang, C.-Y. Lee, and T. Pfister, "Chain-of-table: Evolving tables in the reasoning chain for table understanding," *ICLR*, 2024.

[34] P. Ma, R. Ding, S. Wang, S. Han, and D. Zhang, "Insightpilot: An llm-empowered automated data exploration system," in *EMNLP*, 2023, pp. 346–352.

[35] R. C. Fernandez, A. J. Elmore, M. J. Franklin, S. Krishnan, and C. Tan, "How large language models will disrupt data management," *VLDB Proceedings*, 2023.

[36] Z. A. Naeem, M. S. Ahmad, M. Eltabakh, M. Ouzzani, and N. Tang, "Retclean: Retrieval-based data cleaning using llms and data lakes," *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 4421–4424, 2024.

[37] M. Saeed, N. De Cao, and P. Papotti, "Querying large language models with sql," *arXiv preprint arXiv:2304.00472*, 2023.

[38] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, "Automatic service composition based on behavioral descriptions," *IJCIS*, vol. 14, no. 04, pp. 333–376, 2005.

[39] R. D. Pesl, M. Stötzner, I. Georgievski, and M. Aiello, "Uncovering llms for service-composition: Challenges and opportunities," in *ICSOC*. Springer, 2023.

[40] M. Aiello and I. Georgievski, "Service composition in the chatgpt era," *SOCA*, 2023.

[41] D. Bianchini, M. Garda, M. Melchiori, and A. Rula, "Leveraging large language models for data service discovery," in *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 2024, pp. 1097–1099.

[42] J. G. Mathew, F. Monti, D. Firmani, F. Leotta, F. Mandreoli, and M. Mecella, "Composing smart data services in shop floors through large language models," in *International Conference on Service-Oriented Computing*. Springer, 2024, pp. 287–296.

[43] F. Monti, F. Leotta, J. Mangler, M. Mecella, and S. Rinderle-Ma, "Nl2processops: Towards llm-guided code generation for process execution," in *BPM*. Springer, 2024.

[44] R. D. Pesl, C. Mombrey, K. Klein, D. Zyberaj, I. Georgievski, S. Becker, G. Herzwurm, and M. Aiello, "Compositio prompto: An architecture to employ large language models in automated service computing," in *International Conference on Service-Oriented Computing*. Springer, 2024, pp. 276–286.

[45] A. Kusiak, "Smart manufacturing," *International journal of production Research*, vol. 56, no. 1-2, pp. 508–517, 2018.

[46] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a

tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[47] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[48] C. Koutras, G. Siachamis, A. Ionescu, K. Psarakis, J. Brons, M. Fragkoulis, C. Lofi, A. Bonifati, and A. Katsifodimos, "Valentine: Evaluating matching techniques for dataset discovery," in *ICDE*. IEEE, 2021, pp. 468–479.

[49] H.-H. Do and E. Rahm, "Coma—a system for flexible combination of schema matching approaches," in *VLDB Proceedings*. Elsevier, 2002, pp. 610–621.

[50] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3," in *ASE*, 2022.

[51] B. Wang, H. Fang, J. Eisner, B. Van Durme, and Y. Su, "Llms in the imaginarium: tool learning through simulated trial and error," *arXiv preprint arXiv:2403.04746*, 2024.

[52] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim, "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models," *arXiv preprint arXiv:2305.04091*, 2023.

[53] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *NEURIPS*, vol. 35, pp. 24 824–24 837, 2022.

[54] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *ICLR*, 2022.

[55] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[56] H. Liu, C. Sferrazza, and P. Abbeel, "Chain of hindsight aligns language models with feedback," *arXiv preprint arXiv:2302.02676*, 2023.

[57] C. Lo, C.-H. Chen, and R. Y. Zhong, "A review of digital twin in product design and development," *Advanced Engineering Informatics*, vol. 48, p. 101297, 2021.