Composing Smart Data Services in Shop Floors through Large Language Models

Jerin George Mathew[1], Flavia Monti[1], Donatella Firmani[1], Francesco Leotta[1], Federica Mandreoli[2], and Massimo Mecella[1]

[1] Sapienza Università di Roma, Rome, Italy
{mathew,monti,leotta,mecella}@uniroma1.it, donatella.firmani@uniroma1.it
[2] Università degli Studi di Modena e Reggio Emilia, Modena, Italy
federica.mandreoli@unimore.it

Abstract. Recent years have witnessed an ever-growing use of Large Language Models (LLMs) to lower the technical barrier for several tasks, ranging from coding to querying relational databases to composing services. In this work, we focus on using LLMs to simplify access to data in the industrial scenario, by allowing humans operating on the shop floor to submit a query in natural language and then materializing a table integrating data gathered from different data sources including machines and information systems. In particular, we introduce *COSMADS*, which takes as input a query from an operator on the shop floor and automatically synthesizes a pipeline that leverages existing data sources accessible as services (data services), to compose a table output fulfilling the user's information need. The proposed solution is evaluated using a real case study, showing that results obtained by taking into account available data service descriptions and previous pipelines outperform those obtained by naively employing a state-of-the-art code generation tool.

Keywords: Smart data services · Service composition · Data generation · Large Language Models

1 Introduction

Manufacturing companies have access to massive amounts of disparate data sources spread across various departments, from the shop floor to the warehouse and the administrative offices. These include traditional information systems (e.g., Enterprise Resource Planning systems - ERPs) and sensors provided by machines and physical spaces. These data can be accessed through various mechanisms, including direct access (e.g., a connection to a DBMS) and Application Programming Interfaces (APIs) provided, for example, in the form of Web services. We refer to these mechanisms as *smart data services* (or simply *data services* – DSs).

Single data services can be combined to extract information, benefiting from the integration of multiple data sources. We use the term *information extraction pipeline* (or simply *pipeline*) to denote any computing procedure that *compose*

new information from available data services. Without loss of generality, we can consider this composition as representable by means of relational tables.

Despite this huge potential, the kind of information needed to make decisions on the shop floor, where value is created, changes quickly, leaving operators with limited access to timely data. Continuously developing novel pipelines to address these unpredictable data needs can incur significant costs, especially for non-core IT companies, such as manufacturing ones, that usually rely on an external workforce for data management. Data governance and management platforms can mitigate this problem only partially: maintaining large back-end data lakes they require is often unfeasible due to a lack of know-how, human, and computational resources. This is especially true for Small and Medium Enterprises (SMEs), which cannot sustain the costs of such solutions [32].

On the other hand, chat-enabled digital assistants are emerging as a novel form of interaction between humans and machines, garnering increasing interest in the industrial sector [9]. By enabling shop floor operators to interact using natural language and offering fast, intuitive, and potentially hands-free access to data, these assistants have the potential to streamline, expedite, and improve product-related activities. Large Language Models (LLMs), in particular, have recently demonstrated remarkable capabilities to solve complex questions, especially through the use of external knowledge and tools (e.g., search engines, DBMS, etc.) to ground their responses [12].

Example case study. Consider the case of a cardboard box factory, as shown in Figure 1. The core production process consists of two main machines: Ⓐ a *die cutting machine* that cuts the cardboard by means of interchangeable *die cutters* and Ⓑ a *stacker*, which groups the cut-out cardboard into bunches. Each die cutter features a chip that tracks its speed in rotations per second. A high-resolution camera records the stream of cardboard and detects the defective ones. A second camera is placed after the stacker and records when a bunch is ready for the subsequent processing steps. Several data services are available in this scenario, such as: `DS1`, returning the speed of the currently installed die cutter; `DS2`, returning the id of the camera streaming the production of a die cutter; `DS3`, capturing a cardboard frame from the camera with a given id; `DS4`, returning the defects encountered in cardboard; and `DS5`, detecting the completion of a new bunch.

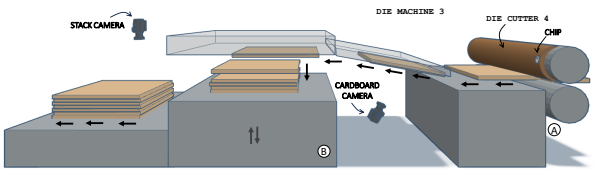

Fig. 1. The cardboard manufacturing case study

Suppose the operator proposes the following natural language query:

> **Q**: "Consider the current session of the die cutter with id 4. Generate a table containing *(i)* the number of cardboards with no defects and *(ii)* those with errors."

We can imagine a system that builds, leveraging LLMs capabilities, a pipeline returning a relational table that satisfies the query **Q** by composing available

data services and treating the data sources (e.g., cameras, sensors, tools) as external knowledge that can be queried and processed. To answer query **Q**: *(i)* `DS2` is invoked to retrieve the id (e.g., 54239) of the camera streaming the production of die cutter 4, *(ii)* `DS3` captures a frame from camera 54239 of cardboard produced, and *(iii)* `DS4` detects possible defects in the captured cardboard frame.

Contribution. In this paper, we present *COSMADS* – COmposing SMArt Data Services, 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*.

Outline. This paper is organized as follows. Section 2 introduces background and related works. In Section 3, we discuss the main components of *COSMADS*. In Section 4, we provide technical details. Experimental results are presented in Section 5. Finally, an outline of future challenges together with concluding remarks are presented in Section 6.

2 Background and related works

LLM in data management. LLMs are rapidly transforming data management, with a wide landscape of methods for querying relational databases [1,13], reasoning over structured data [28,18] and performing data cleaning operations [11]. These works are orthogonal to our approach, where we envision a scenario where relational data are constructed from scratch rather than queried. The idea of creating tables via LLM was recently introduced in [24], where the authors use LLMs not only to query a relational database but also to potentially augment the selected records by tapping the information in the LLM, which has been in turn extracted from massive corpora of text documents during training. In our case, the LLM is not used as a source of information but as a way to identify and compose data services.

LLM for code generation. Due to the similarity between natural language and source code, LLMs are extensively used in software engineering tasks like code understanding and generation [5,8,16]. In this setting, the LLM is prompted to take a natural language description of a problem and generate an executable code in a certain programming language to solve the problem. The surprising performance has also prompted research on using LLM to generate code to express the necessary logic to answer a given question [7].

Recently, a variety of code-generating LLMs have been introduced, including Codex [6], and AlphaCode [16]. These models feature billions of parameters and benefit from the availability of vast amounts of code-related datasets that are publicly available. The use of the LLM agent paradigm and self-reflection has further improved the accuracy of these models on code generation [23].

LLM for service composition. An LLM Agent [21] which leverages external tools can be thought of as a service composition engine. As explained in [3],

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. Such a definition perfectly aligns with the definition of LLM Agent which leverages external tools to derive the output. Authors in [19] take advantage of such an ability to develop a solution where an LLM Agent composes external tools performing process activities, resulting in a code representing a business process (a.k.a. orchestration). Recently, LLMs have also been investigated in the service composition field [22,2] with promising results but still requiring domain expert supervision due to potential inaccuracies in the produced output.

3 COmposing SMArt Data Services

Figure 2 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). We can imagine the human operator to have little or zero knowledge about the available data services. Thus, we assume the query only specifies the required information, omitting technical details on how to compute it.

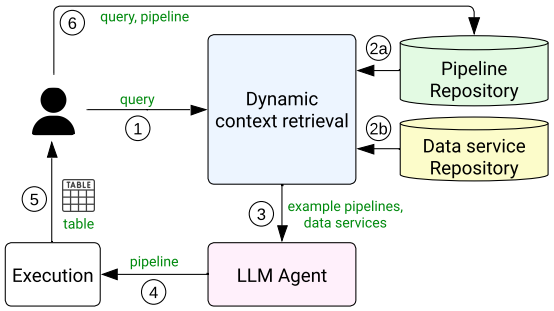

Fig. 2. *COSMADS* high-level architecture

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, we 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 DS.

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 [20], 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.

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.

4 Realizing COSMADS

In this section, we describe the realization details of *COSMADS*, of which a prototype[3] has been implemented in Python leveraging the Langchain framework[4]. The base LLM model utilized for the LLM Agent relies on GPT-4 (`gpt-4-turbo`) by OpenAI[5].

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 3, 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 the metadata, which include the path to the Python script containing the `pipeline` that solves that query. The embeddings are computed by applying the `text-embedding-ada-002` model by OpenAI. For the vector store, we rely on DocArray's DocIndex[6], which is well integrated into Langchain and allows to efficiently access stored data.

Given a natural language query, the *dynamic context retrieval* component computes its embedding and retrieves the top-K similar queries from the vector store. Similarity is computed according to cosine similarity.

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.,

[3] For repeatability, both the source code and the experimental results are openly available at the following link: `https://github.com/jermathew/COSMADS`

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

[5] Some changes in the prompt may be necessary if another LLM model is used.

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

taking a picture). As a consequence, any asset of the company exposing services to support its operational functionalities can be considered a data service. In our example case study, an example of data service is the service related to the camera asset which capture 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. Our 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. Notably, this documentation corresponds to what in traditional service composition was referred to as service description.

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. Our LLM Agent leverages the ICL ability of LLMs [4]. The quality of the output though is strongly dependent on the quality of the provided prompt [30]. For *COSMADS*, in particular, we designed a *prompt template*[3] to be filled with the output of the *dynamic context retrieval* module, which follows the most common best practices [33]. These 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 documentations of the data services, and *(iv)* a set of guidelines the output of the agent must respect.

5 Experimental validation

In order to evaluate our approach, we used a data service repository adapted from a proprietary codebase deployed in a real cardboard manufacturing domain, as part of a smart manufacturing pilot project. The data services have been simplified for our experiments, by simulating the interaction with the physical assets instead of really interacting with actual industrial devices. In particular, the repository consisted of 12 data services. Moreover, we defined 5 queries with corresponding solved pipelines to be used as examples in the LLM Agent prompt and stored them in the *pipeline repository*.

We measured performance results on 5 manually defined queries. The queries were suggested by operators working in the manufacturing factories where the proprietary codebase is deployed. For each query, we produced a total of 10 variants by rephrasing the original query text using an LLM. The queries are defined by increasing their complexity (i.e., q0 represents a simple query, while q4

corresponds to a more complex one) by means of how DSs need to be composed to compute the output, and such that at least one similar query and corresponding solving pipeline are stored in the *pipeline repository*.

Evaluation details. We evaluated the quality of the tables generated by *COSMADS* by comparing them with ground truth data generated from manually-designed Python scripts. The evaluation has been performed at two levels, i.e., intentional (at the schema level) and extensional (at the objects/tuples level).

At the intentional level, we assessed the generated table by comparing its schema to the ground truth schema, i.e., the correct relational definition that the *COSMADS* should ideally provide to the user. This has been done by employing Valentine tool[7] [15] for schema matching based on COMA instance-based method [10]. *Precision* and *recall* metrics are computed for the matching results. A high precision value indicates a low rate of false positives, meaning the generated table contains the correct columns among those it generated. A high recall value indicates a low rate of false negatives, meaning the generated table contains most of the correct columns that should have been included. Note that in general, the columns of the generated schema and the ground truth schema may have different wordings due to different rephrasing of the queries. Valentine COMA instance-based considers such aspects.

At the extensional level, we computed the *instance accuracy* by assessing if the resulting match between the ground truth and generated tables contains the same data. We computed the instance accuracy by measuring if the tables have both the same cells (*Accuracy (cell)*) and the same rows (*Accuracy (row)*). Noteworthy, the accuracy value is computed only on the matched columns. This means that, in principle, high values of accuracy can be achieved with low values of precision and recall if only a subset of the columns are matched but the quality of them is high.

Evaluation results. We report the results of the evaluation in Table 1. Results are averaged over the set of variants for each query.

Table 1. Evaluation results of *COSMADS* (w/ similar query-pipeline example)

Query	Precision	Recall	Accuracy (cell)	Accuracy (row)
q0	1.0	1.0	1.0	1.0
q1	1.0	1.0	1.0	1.0
q2	1.0	0.66	1.0	1.0
q3	1.0	0.67	0.83	0.80
q4	0.9	0.81	0.76	0.55

The results indicate a decreasing trend in the measured metrics as queries become more complex. While the precision of the schema matching remains relatively stable despite query complexity, recall significantly drops for *q2* and *q3*. By manually inspecting the results, we found that missing columns in the generated tables often relate to the timestamps of the time windows. In those cases, accuracy is high for *q2* but not for *q3*, where the main issue lies in incorrect computation of aggregated results within time windows. Metrics of *q4* result to be the worst. In this case, indeed, the composition of the data services is

[7] Cf. `https://github.com/delftdata/valentine`

more complex, and *COSMADS* does not always produce accurate results. Manually inspecting the results, in some cases, generated tables contain correct data but are badly structured, e.g., usage of arrays or textual representation instead of numerical data.

6 Concluding remarks

In this paper, we introduce *COSMADS*, a tool that takes as input a query from an operator on the shop floor and automatically synthesizes a Python script that leverages existing data sources accessible as services to produce a table output fulfilling the user's information need. The proposed solution is evaluated using a real case study, showing that results obtained by taking into account available data service descriptions and previous pipelines outperform those obtained by naively employing a state-of-the-art code generation tool. The employment of the tool is not straightforward though, opening to a set of research challenges.

In first place, the good quality of results strongly depends on the quality of data services documentation. As a consequence, considerable effort must be devoted to defining data services documentation that can be effectively exploited by the LLM Agent. Similar issues happened in the past for traditional service composition, which required semantic service descriptions. Whereas, to this aim, automatic code documentation techniques can be used [14], assessing whether the LLM Agent correctly uses data services and pipelines based on their documentation and demonstrations is an open question. In this scenario, a *profiling* of data services might be needed. Such profiling could involve letting the LLM imagine plausible scenarios where the data services can be used, and letting the LLM fail and learn from its errors, similarly to [26], or using a human-in-the-loop approach where users define exemplary tool usage scenarios and refine the documentation if necessary.

Linked to the previous point, the proposed approach aims at generating an entire pipeline at once, similar to other proposed approaches such as [27]. An alternative is to generate it iteratively using paradigms such as Chain of Thoughts [29]. Both approaches can be further refined using *self-reflection* based strategies [31,25], that allows the LLM Agent to reason over the generated output and refine it progressively. Generating a preview of the final table can be useful to *(i)* detect potential syntax and runtime errors that can hinder the generation of the table and *(ii)* allow the user to find potential missing or incomplete information in the table without providing direct access to the pipeline.

Also, after generating the pipeline, *COSMADS* immediately executes it. While this is generally safe, if the generated pipeline is partly or fully wrong, these could lead to manufacturing sessions that are not well monitored. To this aim, pipeline simulation strategy could help. A possible approach involves *digital twins*, which represent digital replicas of the industrial assets, and have been used either to improve the product development or for monitoring its underlying physical assets [17].

Acknowledgments

Jerin George Mathew is financed by the Italian National PhD Program in AI. The work of Favia Monti was supported by the MISE agreement on "Agile&Secure Digital Twins (A&S-DT)". The work of Donatella Firmani has been partially supported by SEED PNR Project "FLOWER" "Frontiers in Linking records: knOWledge graphs, Explainability and tempoRal data", Sapienza Research Project B83C22007180001 "Trustworthy Technologies for Augmenting Knowledge Graphs" and HORIZON Research and Innovation Action 101135576 INTEND "Intent-based data operation in the computing continuum". The work of Francesco Leotta was partially supported by project SERICS (PE00000014) Extended Partnership under the PNRR MUR program funded by the EU - NextGenerationEU. The work of Federica Mandreoli was partially funded by the Horizon Europe project "WASABI: White-label shop for digital intelligent assistance and human-AI collaboration in manufacturing" (GA No. 101092176). The work of Massimo Mecella is partially funded by MICS (Made in Italy—Circular and Sustainable) (PE00000004) Extended Partnership (CUP B53C22004130001) funded by the EU - NextGenerationEU PNRR MUR.

References

1. Affolter, K., Stockinger, K., Bernstein, A.: A comparative survey of recent natural language interfaces for databases. The VLDB Journal **28**, 793–819 (2019)
2. Aiello, M., Georgievski, I.: Service composition in the chatgpt era. SOCA (2023)
3. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic service composition based on behavioral descriptions. IJCIS **14**(04), 333–376 (2005)
4. 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)
5. Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., et al.: A survey on evaluation of large language models. TIST (2023)
6. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
7. Chen, W., Ma, X., Wang, X., Cohen, W.W.: Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. arXiv preprint arXiv:2211.12588 (2022)
8. Chirkova, N., Troshin, S.: Empirical study of transformers for source code. In: Proceedings of ESEC/FSE 2021 (2021)
9. Colabianchi, S., Tedeschi, A., Costantino, F.: Human-technology integration with industrial conversational agents: A conceptual architecture and a taxonomy for manufacturing. JIII (2023)
10. Do, H.H., Rahm, E.: Coma—a system for flexible combination of schema matching approaches. In: VLDB Proceedings. pp. 610–621. Elsevier (2002)
11. Fernandez, R.C., Elmore, A.J., Franklin, M.J., Krishnan, S., Tan, C.: How large language models will disrupt data management. VLDB Proceedings (2023)
12. Hsieh, C.Y., Chen, S.A., Li, C.L., Fujii, Y., Ratner, A., Lee, C.Y., Krishna, R., Pfister, T.: Tool documentation enables zero-shot tool-usage with large language models. arXiv preprint arXiv:2308.00675 (2023)
13. Katsogiannis-Meimarakis, G., Koutrika, G.: A survey on deep learning approaches for text-to-sql. VLDB J. **32**(4), 905–936 (2023)
14. Khan, J.Y., Uddin, G.: Automatic code documentation generation using gpt-3. In: ASE (2022)
15. Koutras, C., Siachamis, G., Ionescu, A., Psarakis, K., Brons, J., Fragkoulis, M., Lofi, C., Bonifati, A., Katsifodimos, A.: Valentine: Evaluating matching techniques for dataset discovery. In: ICDE. pp. 468–479. IEEE (2021)

16. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al.: Competition-level code generation with alphacode. Science **378**(6624), 1092–1097 (2022)
17. Lo, C., Chen, C.H., Zhong, R.Y.: A review of digital twin in product design and development. Advanced Engineering Informatics **48**, 101297 (2021)
18. Ma, P., Ding, R., Wang, S., Han, S., Zhang, D.: Insightpilot: An llm-empowered automated data exploration system. In: EMNLP. pp. 346–352 (2023)
19. Monti, F., Leotta, F., Mangler, J., Mecella, M., Rinderle-Ma, S.: Nl2processops: Towards llm-guided code generation for process execution. In: BPM. Springer (2024)
20. 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)
21. Parisi, A., Zhao, Y., Fiedel, N.: Talm: Tool augmented language models. arXiv preprint arXiv:2205.12255 (2022)
22. Pesl, R.D., Stötzner, M., Georgievski, I., Aiello, M.: Uncovering llms for service-composition: Challenges and opportunities. In: ICSOC. Springer (2023)
23. Ridnik, T., Kredo, D., Friedman, I.: Code generation with alphacodium: From prompt engineering to flow engineering. arXiv preprint arXiv:2401.08500 (2024)
24. Saeed, M., De Cao, N., Papotti, P.: Querying large language models with sql. arXiv preprint arXiv:2304.00472 (2023)
25. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems **36** (2024)
26. Wang, B., Fang, H., Eisner, J., Van Durme, B., Su, Y.: Llms in the imaginarium: tool learning through simulated trial and error. arXiv preprint arXiv:2403.04746 (2024)
27. Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R.K.W., Lim, E.P.: Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. arXiv preprint arXiv:2305.04091 (2023)
28. Wang, Z., Zhang, H., Li, C.L., Eisenschlos, J.M., Perot, V., Wang, Z., Miculicich, L., Fujii, Y., Shang, J., Lee, C.Y., Pfister, T.: Chain-of-table: Evolving tables in the reasoning chain for table understanding. ICLR (2024)
29. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al.: Chain-of-thought prompting elicits reasoning in large language models. NEURIPS **35**, 24824–24837 (2022)
30. 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)
31. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K.R., Cao, Y.: React: Synergizing reasoning and acting in language models. In: ICLR (2022)
32. Yilmaz, G., Qurban, K., Kaiser, J., McFarlane, D.: Cost-effective digital transformation of smes through low-cost digital solutions. LoDiSA (2023)
33. 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)