LLM-Based Composition of Smart Data Services in Shop Floors

Adriano Izzi[1,*], 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*
[2]*Università di Modena e Reggio Emilia, Modena, Italy*

Abstract

The use of Large Language Models (LLMs) has expanded across various domains, simplifying tasks such as coding, querying relational databases, and service composition. In this work, we explore their role in industrial data access, enabling shop-floor operators to issue natural language queries and obtain structured tables integrating data from multiple sources, including machines and information systems. We present a system that takes user queries as input and automatically generates a data processing pipeline that leverages available data services to construct a table satisfying the user's information needs. Our approach is evaluated on a real-world case study, demonstrating that incorporating data service descriptions and prior pipelines leads to accurate results compared to directly applying a state-of-the-art code generation tool.

Keywords

Smart data services, Service composition, Data generation, Large Language Models

1. Introduction

Manufacturing companies generate vast amounts of data from diverse sources, spanning shop floors, warehouses, and administrative offices. These data originate from traditional information systems, such as Enterprise Resource Planning (ERP) systems, and physical sensors embedded in machinery and production environments. Access to these data sources is facilitated through various mechanisms, including direct database connections and Application Programming Interfaces (APIs), often exposed as Web services. We refer to these mechanisms collectively as *smart data services* (or simply *data services*, DSs). By integrating multiple data services, new insights can be extracted, enabling more informed decision-making. We define an *information extraction pipeline* (or simply *pipeline*) as any computational process that *composes* new information by leveraging available data services. Without loss of generality, this composition can be represented in the form of relational tables. Despite their potential, real-time data access remains a challenge on the shop floor, where rapid decision-making is required. Operators often

SEBD 2025: 33rd Symposium on Advanced Database System, June 16–19, 2025, Ischia, Italy
*Corresponding author.
izzi.2048338@studenti.uniroma1.it (A. Izzi); jeringeorge.mathew@uniroma1.it (J. Mathew); monti@diag.uniroma1.it (F. Monti); donatella.firmani@uniroma1.it (D. Firmani); leotta@diag.uniroma1.it (F. Leotta); federica.mandreoli@unimore.it (F. Mandreoli); mecella@diag.uniroma1.it (M. Mecella)
0000-0002-4626-826X (J. Mathew); 0000-0003-3349-7861 (F. Monti); 0000-0003-0358-3208 (D. Firmani); 0000-0001-9216-8502 (F. Leotta); 0000-0002-8043-8787 (F. Mandreoli); 0000-0002-9730-8882 (M. Mecella)

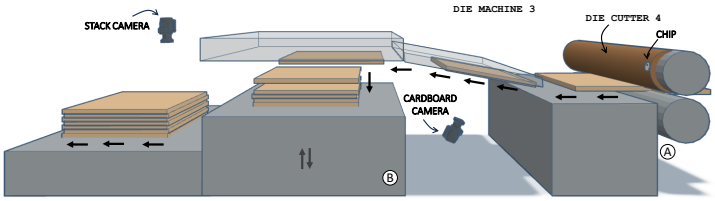

Figure 1: The cardboard manufacturing case study

struggle to retrieve timely information due to evolving data needs, and continuously developing new pipelines to address these needs incurs substantial costs. This issue is particularly critical for non-IT-focused manufacturing companies, which typically outsource data management tasks. While data governance platforms offer partial relief, their reliance on extensive backend data lakes makes them impractical for many organizations, particularly Small and Medium Enterprises (SMEs), which often lack the financial, technical, and human resources required to maintain such systems [1]. Conversely, chat-enabled digital assistants are emerging as an intuitive interface for human-machine interaction, drawing increasing interest from industrial applications [2]. By allowing shop floor operators to interact with data services through natural language queries, these assistants can provide fast, user-friendly, and hands-free access to critical information. Large Language Models (LLMs) have recently demonstrated remarkable capabilities in handling complex queries, particularly when enhanced with external tools such as search engines and databases [3].

Example case study. Consider a cardboard manufacturing plant, illustrated in Figure 1. The production process relies on two primary machines: Ⓐ a die-cutting machine, which cuts cardboard using interchangeable die cutters, and Ⓑ a stacker, which groups cut-out cardboard into *bundles*. The die cutter is equipped with a chip that tracks its speed in rotations per second, while a high-resolution camera monitors the production line, detecting defective cardboards. A second camera, placed after the stacker, records the completion of each bundle. Several data services are involved in this scenario, such as (i) `DS1`, which retrieves the speed of the active die cutter, (ii) `DS2` which identifies the camera monitoring a specific die cutter, (iii) `DS3` which captures an image frame from a specific camera, (iv) `DS4`, which detects defects in a given cardboard image, and (v) `DS5`, which signals when a new bundle is complete. Now, suppose an operator submits the following natural language query:

> **Q**: "For the current session of the die cutter with ID 7, generate a table containing *(i)* the number of defect-free cardboards and *(ii)* those with errors."

In this work we envision a system capable of leveraging LLMs to dynamically construct a pipeline that fulfill a natural language query, composing the required data services and treating sensors, cameras, and other tools as external knowledge sources. Specifically, to answer **Q**, first, `DS2` is invoked to retrieve the ID of the camera monitoring die cutter 7 (e.g., camera 74240). Then, `DS3` captures a frame from camera 74240, recording a sample of the produced cardboard. Finally, `DS3` analyzes the captured image to determine if defects are present.

Contribution. In this work we present a methodology, together with a prototype imple-

mentation, for synthesizing information extraction pipelines in the form of Python scripts, starting from natural language queries. The system utilizes a documented codebase that includes available data services and previously defined pipelines. These pipelines may be manually created or previously generated using our solution.

Outline. The rest of this paper is structured as follows. Section 2 provides background and discusses related work. Section 3 presents the core components of the proposed solution. Section 4 describes technical implementation details. Section 5 reports experimental results. Finally, Section 6 outlines future challenges and presents concluding remarks. This paper builds on our previous work [4], summarizing its key insights and providing an overview of the system and its implementation.

2. Related Works

LLM in data management. LLMs are transforming data management with applications in querying relational databases [5, 6], reasoning over structured data [7, 8], and automating data cleaning [9]. These works typically operate over existing relational data. In contrast, our system constructs relational outputs from scratch by composing data services. The idea of generating tables with LLMs has also been explored in [10], where models enhance database results with external knowledge from their pretraining. In contrast, our approach does not treat the LLM as an information source but as a mechanism to identify and compose data services.

LLM for service composition. LLM Agents [11], which invoke external tools, naturally align with the concept of service composition [12], where complex tasks are fulfilled by combining component services. This capability has been explored in [13], where an agent orchestrates process-execution tools to produce business logic code. More broadly, LLM-based service composition has shown promise [14, 15], though expert supervision remains necessary due to limitations in accuracy and reliability.

LLM for code generation. Due to the structural similarity between code and natural language, LLMs are widely used in software engineering for tasks such as code understanding and synthesis [16, 17, 18]. These models can translate descriptions into executable code, often capturing non-trivial logic [19]. Prominent examples include Codex [20] and AlphaCode [18]. More recent developments, such as LLM Agents and self-reflection techniques [21], have further improved performance in complex code generation tasks.

3. System Architecture

Figure 2 illustrates the architecture of the proposed approach. The process begins when a human operator submits a natural language query ① to retrieve information from an ongoing manufacturing process. Queries can include parameters such as time ranges or specific defect types but do not assume prior knowledge of available data services. Instead, they specify only the required information, without technical details on how to compute it. At the core of the architecture is the *LLM Agent*, which constructs a query-specific prompt for a pre-trained LLM. The prompt is generated based on information retrieved by the *dynamic context retrieval* module.

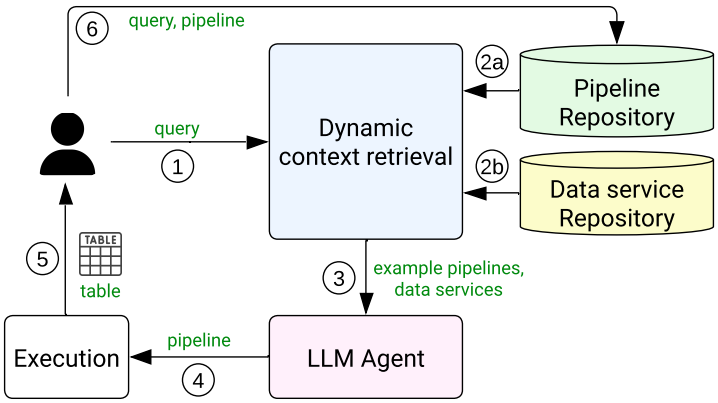

Figure 2: Architecture of the proposed solution

This component identifies ②a relevant example pipelines from the *pipeline repository* and ②b suitable data services from the *data service repository* to help answer the query. Manufacturing assets expose *data services* that range from actuation services (e.g., activating a camera) to data-generating services (e.g., retrieving the current speed of a die-cutting machine). These services operate through various communication protocols, but for simplicity, they are assumed to be callable via function calls. Each service may require parameters and return structured or unstructured outputs. The *data service repository* stores documentation describing each service functionality and usage. The *pipeline repository* contains predefined pipelines, either manually created or derived from previous executions of our solution. Each pipeline is linked to the query it was designed to fulfill and is represented as a software script (Python) that *(i)* produces a structured table as output and *(ii)* utilizes data services. Pipelines can be expressed in different formalisms, such as programming languages or workflow scripting languages [22]. Once the relevant pipelines and data service documentation are retrieved, they are provided ③ to the *LLM Agent* along with the query. This information is incorporated into a structured prompt template, guiding the LLM in generating an appropriate pipeline. The generated pipeline is then executed ④ in an execution environment, such as a Python interpreter. The execution produces a structured table ⑤ that answers 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 for further reuse.

4. Implementation Details

In this section, we outline the implementation details of the proposed solution. A prototype[1] has been developed in Python using the Langchain framework[2]. The LLM Agent is based on OpenAI's GPT-4 (`gpt-4-turbo`)[3].

[1]For repeatability, both the source code and experimental results are publicly available at: https://github.com/jermathew/COSMADS

[2]See https://www.langchain.com/

[3]GPT-4 can be replaced with any other LLM, though adjustments to the prompt may be needed to maintain accuracy.

Dynamic Context Retrieval. The LLM Agent's prompt is built using two key elements: *(i)* relevant past queries and their associated pipelines as few-shot examples, and *(ii)* the set of data services required to answer the input query. As outlined in Section 3, this information is retrieved from the *pipeline repository* and *data service repository*, respectively. The pipeline repository is a vector store that maintains embeddings of solved queries, computed using OpenAI's `text-embedding-ada-002`, along with metadata pointing to the corresponding Python pipelines. For efficient storage and search, we use DocArray's DocIndex[4], which integrates with Langchain. Given a user query, its embedding is computed and the top-K[5] most similar queries are retrieved using cosine similarity. The data service repository lists services that enable access to historical and real-time data. These are typically implemented as Python classes, each including: *(i)* a function that wraps the actual service logic, and *(ii)* a documentation field that describes its functionality, usage, example invocation, and input/output parameters. The LLM Agent relies on this documentation to compose pipelines using the appropriate services. While only the top-K example pipelines are included in the prompt, all data services are made available. This design assumes the set of services remains relatively stable over time, whereas the number of pipelines is expected to grow. Including all services avoids generation errors due to missing service descriptions.

LLM Agent. Our LLM Agent leverages the in-context learning (ICL) capability of LLMs [23]. However, the effectiveness of the generated output depends heavily on the quality of the prompt [24]. In our system, we designed a prompt template, which is populated with information retrieved by the dynamic context retrieval module, following established best practices [25]. These ensure that the task is clearly defined, relevant contextual information is included, demonstrations are provided, and using a model-friendly format style. The prompt consists of both static and dynamically retrieved components and is shown in Figure 3[6]. The invariable sections, highlighted in blue, include (i) a system header that defines the agent's expertise, (ii) a goal description outlining the expected output, (iii) a specification of the structure of data service documentation, and (iv) a set of guidelines that the model must follow when generating its response. The dynamic sections, marked in yellow, are retrieved by the dynamic context retrieval module and contain (i) the set of data services relevant to answering the query and (ii) a selection of example queries with their corresponding pipelines, which serve as few-shot demonstrations. Finally, the user's input query, shown in red, is integrated into the prompt to specify the problem to be solved.

5. Experimental Validation

To evaluate our approach, we adapted a data service repository from a proprietary codebase used in a real-world cardboard manufacturing pilot. For experimentation, service interactions with physical assets were simulated rather than executed on actual devices. The repository contained 12 data services. We also manually defined five queries (`q0`–`q4`), each paired with a corresponding solved pipeline serving as ground truth. These queries, suggested by factory

[4] https://github.com/docarray/docarray
[5] We use $K = 1$ in our experiments.
[6] The full prompt is available at https://github.com/jermathew/COSMADS/blob/main/src/pipeline_chain.py

Prompt template

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

Figure 3: Overview of the prompt template

operators, were rephrased into 10 variants using an LLM to introduce diversity. The queries were designed with increasing complexity, from simple (q0) to complex (q4), based on the number and type of data services involved. Each was ensured to have at least one similar query and pipeline in the repository for retrieval during evaluation.

Evaluation details. We assessed the quality of the tables generated by our approach by comparing them to ground truth data produced from manually written Python scripts. The evaluation was performed at two levels: intentional (schema-level) and extensional (data-level). At the intentional level, we compared the structure of the generated tables with the expected schema, representing the correct relational definition. Schema matching was conducted using the Valentine tool[7] [26], based on the COMA instance-based method [27]. Precision and recall were used to assess schema quality: high precision indicates few incorrect columns (false positives), while high recall reflects the inclusion of most expected columns (low false negatives). Since rephrased queries can lead to varying column names, the matching method accounts for such variations. At the extensional level, we measured the correctness of the data in matched columns using two metrics: *Accuracy (cell)* and *Accuracy (row)*. These capture, respectively, the proportion of correct values at the cell and row level. Importantly, accuracy is computed only over matched columns, meaning high accuracy scores can still be achieved even with lower precision or recall, as long as the content of matched columns is correct.

Evaluation results. The results of our evaluation are summarized in Table 1, with values averaged over the set of query variants. The findings reveal a decreasing trend in the measured metrics as query complexity increases. While schema matching precision remains relatively stable across different queries, recall drops significantly for *q2* and *q3*. Upon manual inspection, we observed that missing columns in the generated tables were often related to timestamps used for defining time windows. In *q2*, despite some missing timestamp columns, accuracy remains high, whereas in *q3*, errors primarily stem from incorrect computations of aggregated results within time windows, leading to a lower accuracy score. The results for *q4* show the

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

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

Table 1
Evaluation results of our approach (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	0.33	1.0	1.0
q3	1.0	0.42	0.54	0.23
q4	0.4	0.31	0.28	0.0

(a) W/o similar pipeline example – *(a)* version

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	0.4	0.6	0.6
q3	0.9	0.32	0.36	0.23
q4	0.6	0.23	0.23	0.14

(b) W/o pipeline example – *(b)* version

Table 2
Ablation study results

lowest overall performance. In this case, the increased complexity of data service composition challenges our solution, leading to inconsistencies in the generated output. Manual analysis revealed that in some cases, the generated tables contained correct data but were poorly structured, for instance, using arrays or textual representations instead of numerical values.

Ablation study. To assess the impact of the dynamic context retrieval component on result quality, we conducted an ablation study. The experiments were performed on the same set of queries, considering two modified versions of our solution. In the first version, (a), instead of providing the LLM Agent with similar query-pipeline examples, we replaced them with unrelated query-pipeline pairs. In the second version, (b), we completely removed query-pipeline examples from the prompt. The results of these two settings are reported in Table 2a and Table 2b, respectively. Both configurations lead to worse performance compared to Table 1, underscoring the importance of query-pipeline few-shot examples in guiding the LLM Agent. Interestingly, version (a), which replaces relevant examples with unrelated ones, does not perform as poorly as one might expect. This suggests that even in an initial deployment phase, where a repository of relevant query-pipeline examples may not yet be available, our solution can still generate reasonably accurate results.

Baseline: GitHub Copilot. We compared our system to an end-to-end baseline using a code-generating LLM, GitHub Copilot[8]. Copilot was prompted to access the documentation of the data services and generate a JSON file containing the tabular data as a response to the query. The evaluation was conducted using Copilot's chat mode within Visual Studio Code[9]. Given a workspace containing the set of implemented tools, we invoked the `@workspace` agent, which employs a meta-prompt to determine relevant information from the workspace

[8]Cf. https://github.com/features/copilot/
[9]Cf. https://code.visualstudio.com/docs/copilot/overview.

Query	Precision	Recall	Accuracy (cell)	Accuracy (row)
q0	1.0	1.0	1.0	1.0
q1	0.7	0.55	0.15	0.1
q2	0.9	0.33	0.8	0.8
q3	1.0	0.45	0.65	0.63
q4	0.2	0.15	0.09	0.0

Table 3
Github Copilot

to assist in answering the query. The results, reported in Table 3, show that while Copilot is capable of generating tables for simpler queries, its performance degrades as query complexity increases. Interestingly, in some cases, Copilot outperforms our approach, likely due to its specialized training on code generation, which gives it an advantage over GPT-4 in certain scenarios. However, integrating Copilot into an automated workflow remains a challenge, as its web APIs are not publicly accessible. Nonetheless, the results suggest that replacing GPT-4 with Copilot within our solution could further enhance performance.

6. Concluding Remarks

In this paper we presented a tool that generates Python scripts from operator queries by leveraging existing data sources to produce tabular outputs. Evaluation on a real-world case study shows that incorporating data service descriptions and prior pipelines improves accuracy over state-of-the-art code generation tools lacking such context. A key challenge lies in ensuring high-quality service documentation, which directly affects LLM Agent performance. While automatic documentation techniques [28] offer promise, enabling the LLM to reliably interpret and apply services remains an open issue, potentially addressable via service profiling or human-in-the-loop refinement [29]. Pipeline generation also poses challenges. Although our system produces pipelines in a single step [30], iterative strategies using Chain-of-Thought reasoning [31] and self-reflection [32, 33, 34] could improve quality. Previewing the resulting table could help catch errors early. Finally, direct execution introduces risks; pipeline simulation using digital twins [35] could offer a safer testing environment.

Acknowledgments

The work of Flavia Monti was supported by the MISE agreement on “Agile&Secure Digital Twins (A&S-DT)”. Jerin George Mathew was financed by the Italian National PhD Program in AI. The work of Donatella Firmani has been partially supported by the HORIZON Research and Innovation Action 101135576 INTEND “Intent-based data operation in the computing continuum”. 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 and Francesco Leotta was partially funded by MICS (Made in Italy—Circular and Sustainable) (PE00000004) Extended Partnership (CUP B53C22004130001) funded by the EU - NextGeneration EU PNRR MUR.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] G. Yilmaz, K. Qurban, J. Kaiser, D. McFarlane, Cost-effective digital transformation of smes through low-cost digital solutions, LoDiSA (2023).

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

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

[4] J. G. Mathew, F. Monti, D. Firmani, F. Leotta, F. Mandreoli, 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.

[5] K. Affolter, K. Stockinger, A. Bernstein, A comparative survey of recent natural language interfaces for databases, The VLDB Journal 28 (2019) 793–819.

[6] G. Katsogiannis-Meimarakis, G. Koutrika, A survey on deep learning approaches for text-to-sql, VLDB J. 32 (2023) 905–936.

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

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

[9] R. C. Fernandez, A. J. Elmore, M. J. Franklin, S. Krishnan, C. Tan, How large language models will disrupt data management, VLDB Proceedings (2023).

[10] M. Saeed, N. De Cao, P. Papotti, Querying large language models with sql, arXiv preprint arXiv:2304.00472 (2023).

[11] A. Parisi, Y. Zhao, N. Fiedel, Talm: Tool augmented language models, arXiv preprint arXiv:2205.12255 (2022).

[12] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, Automatic service composition based on behavioral descriptions, IJCIS 14 (2005) 333–376.

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

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

[15] M. Aiello, I. Georgievski, Service composition in the chatgpt era, SOCA (2023).

[16] 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).

[17] N. Chirkova, S. Troshin, Empirical study of transformers for source code, in: Proceedings of ESEC/FSE 2021, 2021.

[18] 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 378 (2022) 1092–1097.

[19] W. Chen, X. Ma, X. Wang, W. W. Cohen, Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, arXiv preprint arXiv:2211.12588 (2022).

[20] 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).

[21] T. Ridnik, D. Kredo, I. Friedman, Code generation with alphacodium: From prompt engineering to flow engineering, arXiv preprint arXiv:2401.08500 (2024).
[22] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (2004) 3045–3054.
[23] 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).
[24] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, D. C. Schmidt, A prompt pattern catalog to enhance prompt engineering with chatgpt, arXiv preprint arXiv:2302.11382 (2023).
[25] 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).
[26] C. Koutras, G. Siachamis, A. Ionescu, K. Psarakis, J. Brons, M. Fragkoulis, C. Lofi, A. Bonifati, A. Katsifodimos, Valentine: Evaluating matching techniques for dataset discovery, in: ICDE, IEEE, 2021, pp. 468–479.
[27] H.-H. Do, E. Rahm, Coma—a system for flexible combination of schema matching approaches, in: VLDB Proceedings, Elsevier, 2002, pp. 610–621.
[28] J. Y. Khan, G. Uddin, Automatic code documentation generation using gpt-3, in: ASE, 2022.
[29] B. Wang, H. Fang, J. Eisner, B. Van Durme, Y. Su, Llms in the imaginarium: tool learning through simulated trial and error, arXiv preprint arXiv:2403.04746 (2024).
[30] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, E.-P. Lim, Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models, arXiv preprint arXiv:2305.04091 (2023).
[31] 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, Advances in neural information processing systems 35 (2022) 24824–24837.
[32] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, Y. Cao, React: Synergizing reasoning and acting in language models, in: ICLR, 2022.
[33] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, S. Yao, Reflexion: Language agents with verbal reinforcement learning, Advances in Neural Information Processing Systems 36 (2024).
[34] H. Liu, C. Sferrazza, P. Abbeel, Chain of hindsight aligns language models with feedback, arXiv preprint arXiv:2302.02676 (2023).
[35] C. Lo, C.-H. Chen, R. Y. Zhong, A review of digital twin in product design and development, Advanced Engineering Informatics 48 (2021) 101297.