International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue: 06 | June - 2024 SJIF Rating: 8.448 ISSN: 2582-3930

PlanLLM: Planning based Large Language Agents to Address Data Manipulation Queries, Using Local Large Language Models

Abdu Rehaman Pasha Syed
Department of ISE
R. V. College of Engineering®
Bengaluru, India

Dr. Kavitha S.N
Department of ISE
R. V. College of Engineering®
Bengaluru, India

Srivathsan Varadharajan
Data Scientist
Philips
Bengaluru, India

***Abstract*—Planning architecture are a powerful set of tools utilised to realise step wise execution of queries. These execution steps are often arranged in a topological order, with a fixed execution cycle. Generation and creation of a consistent set of plans for execution of a query, require several calls to Large Language engine, and can turn out to be highly uneconomical for industry use case. In this proposed system, a planning architecture that utilises local Large Language instances through Ollama client and Kubernetes pods to generate instruct model based responses, as well as a code generator and executor engine relying on coding based Large Language Models, including CodeGemma and CodeLlama are studied as an alternative to API based cloud solutions. This can increase the overall control of data flow for several use cases, as well as provide an exhaustive tool for data visualisation and manipulation tasks. This setup also provides a general architecture for using planning based agents for other such sequential use cases.**

***Index Terms*—Large Language Models, Kubernetes, Ollama, CodeGemma, CodeLlama, Planning agents.**

I. Introduction

Data visualisation and manipulation tasks are cumbersome to be handled by an Large Language Model(LLM) agent. It requires understanding of the data schema, processing of user query, intermediate action required by the user as well as a set of guided steps to aid in building complex queries. Planning agents prove to be a very powerful tool for such use cases. These agents rely on Reason and Act(ReAct) based steps and produce an understanding of queries at multiple levels based on complexities. This produces the basic understanding of a plan in these agents.

Through construction of a well connected plan, the agents can always follow a guided step for level based execution. This involves steps that deconstruct the data schema, produce a series of steps for execution of manipulation operations and finally visualise the data for quicker insights. Usual architectures involve utilising a single LLM agent, that can produce quicker and efficient responses of textual queries, however queries requiring data manipulation and visualisation are difficult due to the inherent requirement of generating code and executing the entire query. To address this different multi-agent options are possible, which reduce high level tasks into lower sub tasks, that can be executed more efficiently. In this system, an architecture to produce efficient data retrieval operations, along with utilising local LLM instances are introduced, that provides a much higher data security implementation, and can be extended to produce results on a consistent basis.

II. Literature Review

The paper titled ”Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning” [1] presents a method for few-shot grounded planning for embodied agents using a large language model (LLM), specifically focusing on the ALFRED dataset. The authors introduce the LLM-Planner, which, despite using less than 0.5% of paired training data, achieves competitive performance with recent baselines that are trained using the full training data. Existing methods struggle to complete any task under the same few-shot setting. The LLM-Planner can generate high-quality, high-level plans grounded in the current environment with minimal labeled data. The work has significant implications for the development of versatile and sample-efficient embodied agents capable of quickly learning many tasks [1]. The authors acknowledge the limitations of existing work, which may hinder larger-scale applications beyond their narrow evaluation setting. They suggest enhancing the ability to learn new tasks with a few training examples and emphasize the importance of careful prompt design and other techniques for better in-context learning introducing an innovative approach with the LLM-Planner that utilizes pre-trained LLMs for grounded planning[1]. This demonstrates the potential for LLMs to drastically reduce the data requirements for training effective planning systems. Complementing this work, Singh et al.,[3] proposed TWOSTEP, a multi-agent task planning system that combines classical planners with LLMs to efficiently handle two-agent tasks [2]. The evaluation across various domains showed that TWOSTEP outperformed single-agent and multi-agent PDDL formulations in both planning time and execution efficiency, particularly when dealing with partially independent subgoals. These studies suggest that incorporating LLMs into the planning process could significantly enhance the capabilities of embodied agents,

facilitating more efficient learning and planning in complex environments. Continued research in this area may focus on optimizing planning time efficiency and execution length variability, further refining the integration of LLMs with task planning methodologies. Developing RAG based models over traditional fine-tuning approaches, a pivotal advancement in integrating user provided documents with LLMs. The document outlines three successive paradigms—Naive, Advanced, and Modular RAG—each representing an evolutionary step in refining the RAG framework. The results from this study underscore how RAG, when combined with other AI methodologies such as fine-tuning and reinforcement learning, extends the capabilities of LLMs, enabling them to leverage both parameterized and non-parameterized knowledge sources effectively [4]. The convergence of RAG and fine-tuning methodologies has shown promising outcomes, particularly in tasks where context understanding and external knowledge retrieval are crucial. However relying on fine-tuning is an expensive undertaking that can be avoided for general purpose structured documents related tasks. The RAG framework enhances the LLM's ability to dynamically retrieve and generate information from extensive databases, providing a more nuanced and informed output than could be achieved through fine-tuning alone, but with a much lesser cost. This has significant practical implications, as it allows for more adaptable and contextually aware AI systems [4,5]. Despite its progress, there is room for improvement in terms of robustness and handling extended contexts. Moreover, the future scope of RAG is expanding into multimodal domains, paving the way for its principles to be adapted for interpreting and processing various forms of data, such as images, videos, and code. This anticipated evolution of RAG points to its vast applicability and potential in creating more sophisticated and versatile AI deployments [4]. The recent development of the Forward-Looking Active REtrieval (FLARE) method marks a significant stride in the field of generative language models (LMs). FLARE is designed to enhance content generation by anticipating future content through an innovative retrieval-augmented generation process. By generating a temporary next sentence and using it as a query, FLARE retrieves relevant documents to inform and refine the subsequent generation of text [4,5]. In a comprehensive evaluation spanning four long-form knowledge-intensive generation tasks and datasets, FLARE has demonstrated its capability to outperform baseline models across all measured metrics. The robustness of FLARE is evidenced by its superior or competitive performance, underscoring the method's effectiveness in leveraging retrieved information to improve content generation [5]. Despite the successes of FLARE, the document identifies a clear direction for future research. Improving the generation of search queries by LMs, utilizing task-generic retrieval instructions, and exemplars remains a challenging frontier. The substantial gap between FLARE's current capabilities and the complexity of question decomposition suggests a rich potential for further advancements in retrieval-augmented generation techniques [5,6]. Through these attempts, developing of a robust system capable of handling document extraction through RAG and a planner architecture to address query response for structured relational data will address shortcomings as well as provide an economical system for the industry [7,8].

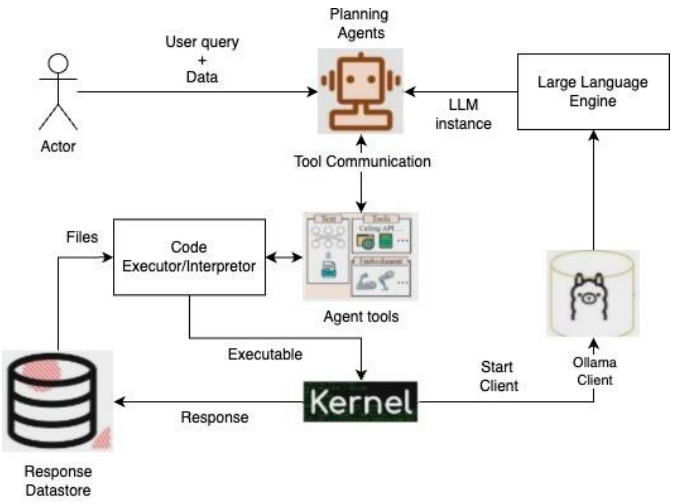

Fig. 1. Architecture of the Proposed System

III. Proposed System

A. Architecture

The architecture can be seen in Figure 1. In the overall architecture, the system components and the flow between them is shown. The system kernel provides the necessary computation and backend assistance for the entire system to be used on. The user pushes the structured data to a data-store, while providing the query to the planning agent. The planning agent then creates an initial plan to follow through and complete the query [9]. Based on the type of document, the planning agent is provided with a choice of using the retrieval augmented generation module for contract extraction. To create the plan an LLM instance is required, which is provided in two ways, one through the cloud infrastructure using their APIs, and a local instance through Ollama Client. If the plan requires data manipulation or visualisation, the planner contacts the code executor/interpreter. The executor then creates a code instance and runs it on the kernel. The response is then stored in a response data-store that is accessible by the user and the agent [10].

B. Methodology

The aim of the proposed architecture is to avail the use of local LLM models within the scope of planning and code execution, making it secure and cost effective for industry wide usage.

1) Setting up the planner: The planner agent is the crux of the entire architecture, and as such needs adequate framework support to run. Initially, Ollama client is used to setup local Llama 3 70 billion parameters model, and it is used as an instruct model. The LLM is provided with examples as a

source of generation json format based plans. These plans are to be strictly followed as per the example json file, failing to notice could cause json parser errors. Through the instruct model, the provided user queries are setup and different prompt engineering techniques such as few shot queries, self reflection and ReAct were used to setup the planner. ReAct proved to be a better choice for the planner as it provided a simple and less computationally expensive prompt example engineering, compared to the others. Through the ReAct prompting example json formats were provided to the planner.

```
pulling manifest
pulling 392f2ba7a9be... 100% ▕▏ 5.0 GB
pulling 097a36493f71... 100% ▕▏ 8.4 KB
pulling 109037bec39c... 100% ▕▏  136 B
pulling 65bb16cf5983... 100% ▕▏  109 B
pulling 043e4545e532... 100% ▕▏  483 B
verifying sha256 digest
writing manifest
removing any unused layers
success
>>> Generate a python code for reading a dataframe named df
```python
import pandas as pd

# Read the dataframe from a CSV file
df = pd.read_csv('filename.csv')
```

Fig. 2. Ollama client used to setup LLM instance.

2) Setting up the code generator: The planner agent communicates a series of steps that are required to be executed part by part, sequentially, to realise the entire execution of the query. This was addressed through setting up another LLM which was specialised in code generation. The choice of programming language was chosen as Python, as it provided easier access to libraries that can visualise data. CodeGemma and CodeLlama were used as the code generator LLMs for the code generator agent. CodeGemma was chosen based on better outputs for the example problem set. An example query is provided, that gives the python code and that is fed to the kernel for execution. Based on the execution result, the output is stored in a session location.

3) Linking the planner and the code generator: The setup of the planner provides a sequential step based plan, that can provide instructions to the code generator. If a step requires a specific code for manipulation of the data, then it raises a call to the code generator. The code generator then proceeds to generate a python code to address the step requirements. Once the step is addressed the execution is begun. The output is saved in a file location, and the result of the code generator step is closed. The planner again begins its execution.

C. Understanding the flow of data

The data flow diagrams describe the proposed data flow architecture to produce a simple and robust planning system. It is divided here into two levels. Level-0 gives the entire component and its interaction with the external entity, i.e., the user. Level-1 provides the breakdown of the entire component into individual modules that will provide the necessary response to the user.

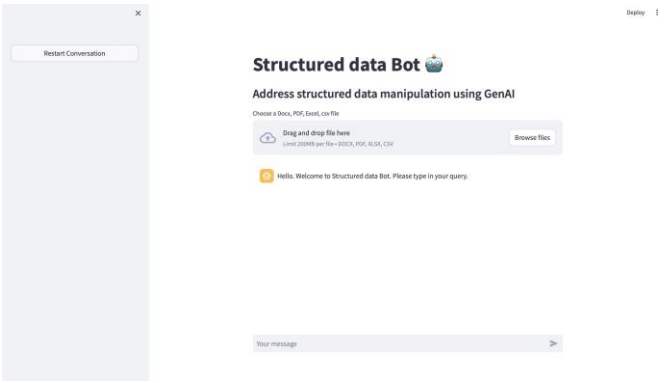

Fig. 3. Frontend to load the files and query

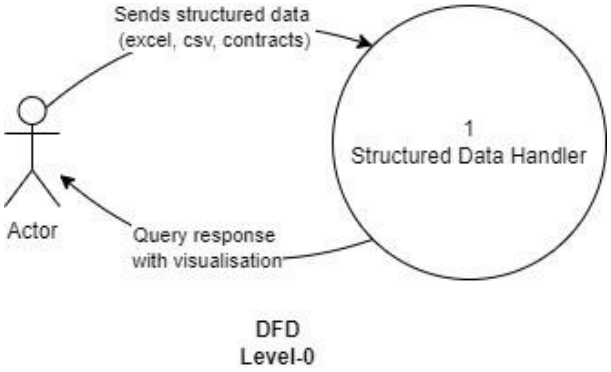

Fig. 4. DFD Level-0

Level-0 provides the entire component as structured data handler. This component accepts structured data and user query as input. Structured data includes contracts and csv,excel,json files etc., and it finally provides with a query response. Based on the query requirements, the user is also provided with the visualisation of the data if required.

Level-1 splits the structured data handler component into 5 major components. The Planner agent forms the most important component, that can plan and execute the user query. It sends the contract file to be indexed by the vector database, which will be utilised by the RAG module. A data parser module is provided to the entire system, to aid in parsing information and sending it in readable format to the user. Execution agent accepts the data manipulation plan from the planner agent, and performs the code execution. The executed response is then provided to the data parser module. Finally, for the planner agent and execution agent to work, the LLM instance is provided by the LLM/Ollama instance setup module.

IV. Result & Discussion

To realise the system a simple plotting query for an incident dummy data was used. The dummy data consisted of incident
```

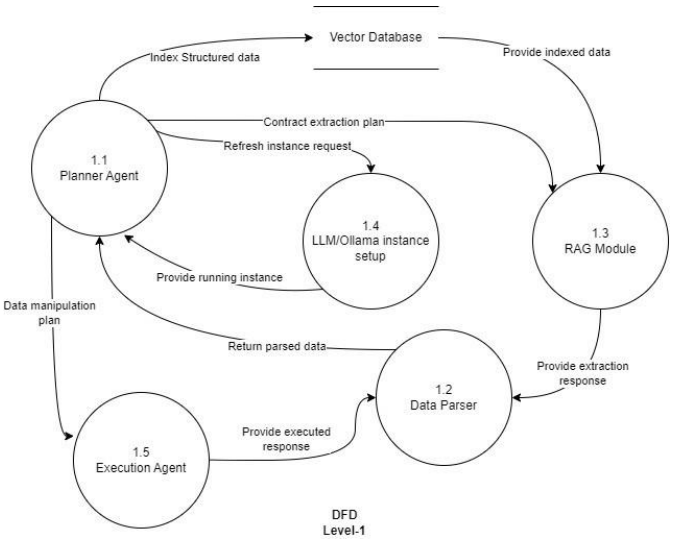

Fig. 5. DFD Level-1

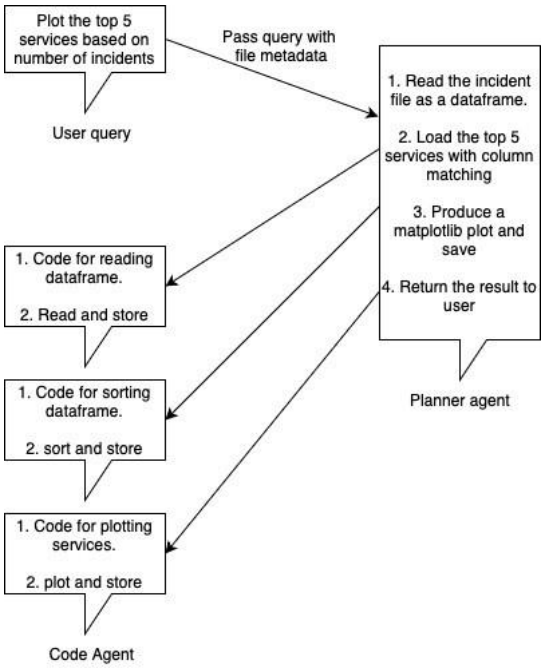

Fig. 6. Generating plan for the sample incident file

name, occurrence, time limit, date and department the incident belonged to. Sample queries, requesting the plot of top 5 services based on number of incidents were sent to the planning agent.

The sample file, is then loaded into the planner. The base step involves differentiating the file into contract or data based file. This is done through a document loader read, and searching for xlsx or csv content. If the planner fails to obtain any xlsx or csv content it loads the plan for RAG extraction of the contract summary and source metadata. However, in the proposed system this data is loaded as a xlsx data and the planning steps proceed through the call to local llm agent. Based on the requirement of the step, the agent proceeds with call to judge whether the step requires code while generating the plan itself. The field, ”requires_code” is set to trigger on as seen in Figure 7, and the step, query and the file with its metadata is passed on to the code agent.

Along with explaining the steps to the user, the planner agent produces json snippets that are loaded as steps into a steps list. This list is tentative, and based on results the agent can be called again to refine it’s plan. Now the first step is passed onto the code generator agent.

The code generator is provided with filler query, were the json ”step” field of the step is loaded and appended in the query. This query is then passed to the CodeGemma model.

In the backend, the code is then parsed and saved as a .py file ready to be executed. Then within the kernel it is loaded and executed, and the session is saved in a local folder accessible by the planner.

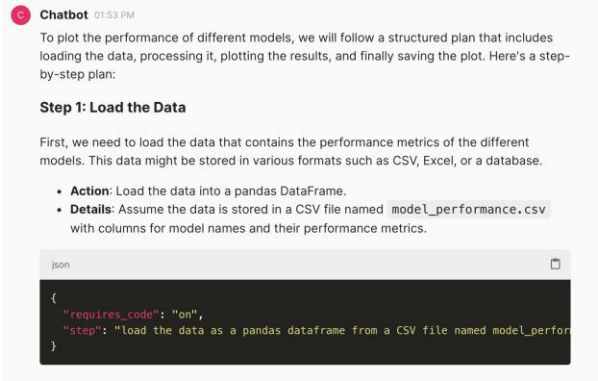

Fig. 7. The planner agent explaining the steps to the user

```
>>> Generate a python code to load in the consolidated
_data.xlsx as a dataframe. Make sure you produce it as
 a json format, with the cod
... e field containing the code, and the to field cont
aining to the kernel
```python
import pandas as pd
import json

# Load the consolidated_data.xlsx file as a dataframe
df = pd.read_excel('consolidated_data.xlsx')

# Create a dictionary with the code and to fields
data = {
    'code': df['Code'].tolist(),
    'to': df['To'].tolist()
}
```

Fig. 8. The code generator generates the code in the backend

Running through all the steps, provides a final plot output. Based on the complexity of the task, the agent can be tuned to provide better examples. If any query is executed improperly,
```

an example can be made out of the prompt and the expected response, which creates a refined few-shot learning process for the LLM.

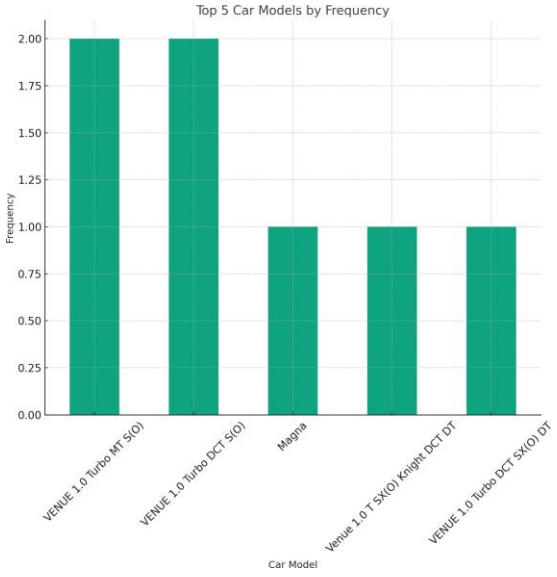

Fig. 9. Plot of the top 5 frequent models of cars

This proposed system proves to be more secure in terms of data handling for industrial usage, as it relies on executing queries based on local large language models, which provides much more control over the data flow [12]. It is possible to horizontally scale the system by utilising more powerful infrastructure, providing industries with flexible cost management options [13].

Through a simpler instruct model, the code generator agent is able to handle most of the data manipulation and visualisation queries. This system is more robust, and reduces hallucination, compared to single agent models, that do provide the necessary execution of data manipulation, and often exceed rate limits set due to excessive calls. By splitting the work of code generation to a second agent, it provides more flexibility to produce plans and well structured steps.

V. Conclusion

Data manipulation and visualisation is a difficult task to handle for many LLMs. By utilising 2 agent model, that involves a planning agent and a code generation agent, it provides higher flexibility and a more robust system to address data manipulation queries. Furthermore use of local LLMs, through Ollama client, it provides local llm instances that are isolated and can be setup for industrial usage. This provides much stronger data security and efficient data flow handling. Through instruct based LLMs, producing plans are much simpler. These plans can be extended to include other step based querying attempts to address generic queries at a larger scale [14]. Code generator agent is specialised in producing python snippets that address the step query. This provides a lower level breakdown of the task requirement. This breakdown is essential in execution of the queries, as providing high level tasks to the code generator can result in highly complex coding requirements, which the code generator might fail to address consistently.

References

[1] Guan, Lin, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. ”Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning.” In Advances in Neural Information Processing Systems, pp. 79081-79094. Curran Associates, Inc., 2023.

[2] Song, Chan Hee, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. ”LLM-Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models.” Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), pages 2998-3009, October 2023.

[3] Singh, Ishika, David Traum, and Jesse Thomason. ”TwoStep: Multi-agent Task Planning using Classical Planners and Large Language Models.” arXiv preprint arXiv:2403.17246 (2024).

[4] Gao, Yunfan, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. “Retrieval-Augmented Generation for Large Language Models: A Survey.” arXiv preprint arXiv:2312.10997, 2024.

[5] Jiang, Zhengbao, Frank F. Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. “Active Retrieval Augmented Generation.” arXiv preprint arXiv:2305.06983, 2023.

[6] Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y.-J., & Huang, G. (2024). ExpeL: LLM Agents Are Experiential Learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17), 19632-19642. DOI: 10.1609/aaai.v38i17.29936.

[7] Zhu, Yuqi, Shuofei Qiao, Yixin Ou, Shumin Deng, Ningyu Zhang, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, and Huajun Chen. ”KnowAgent: Knowledge-Augmented Planning for LLM-Based Agents.” arXiv preprint arXiv:2403.03101, 2024.

[8] Ji, Zhenlan, Daoyuan Wu, Pingchuan Ma, Zongjie Li, and Shuai Wang. ”Testing and Understanding Erroneous Planning in LLM Agents through Synthesized User Inputs.” arXiv preprint arXiv:2404.17833, 2024.

[9] Dagan, Gautier, Frank Keller, and Alex Lascarides. ”Dynamic Planning with a LLM.” arXiv preprint arXiv:2308.06391, 2023.

[10] Mei, Kai, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. ”LLM Agent Operating System.” arXiv preprint arXiv:2403.16971, 2024.

[11] Wang, Zihao, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Shawn Ma, and Yitao Liang. ”Describe, Explain, Plan and Select: Interactive Planning with LLMs Enables Open-World Multi-Task Agents.” In Advances in Neural Information Processing Systems, vol. 36, 2024.

[12] Pan, Bo, Jiaying Lu, Ke Wang, Li Zheng, Zhen Wen, Yingchaojie Feng, Minfeng Zhu, and Wei Chen. ”AgentCoord: Visually Exploring Coordination Strategy for LLM-Based Multi-Agent Collaboration.” arXiv preprint arXiv:2404.11943, 2024.

[13] Huang, Xu, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. ”Understanding the Planning of LLM Agents: A Survey.” arXiv preprint arXiv:2402.02716, 2024.

[14] Zhang, Zheying, Maruf Rayhan, Tomas Herda, Manuel Goisauf, and Pekka Abrahamsson. ”LLM-Based Agents for Automating the Enhancement of User Story Quality: An Early Report.” arXiv preprint arXiv:2403.09442, 2024.