ProAgent: From Robotic Process Automation to Agentic Process Automation

Anonymous ACL submission

Abstract

From water wheels to robotic process automation (RPA), automation technology has evolved throughout history to liberate human beings from arduous tasks. Yet, RPA struggles with tasks needing human-like intelligence, especially in elaborate design of workflow construction and dynamic decision-making in workflow execution. As Large Language Models (LLMs) have emerged human-like intelligence, this paper introduces Agentic Process Automation (APA), a groundbreaking automation paradigm using LLM-based agents for advanced automation by offloading the human labor to agents associated with construction and execution. We then instantiate ProAgent, an LLM-based agent designed to craft workflows from human instructions and make intricate decisions by coordinating specialized agents. Empirical experiments are conducted to validate the effectivenss of our proposed ProAgent, showcasing the feasibility of APA, unveiling the possibility of a new paradigm of automation driven by agents.

1 Introduction

Automation, aiming to reduce human intervention in processes and enhance efficiency, has undergone a series of evolutionary stages throughout history. From the waterwheel irrigation system in the early agricultural age to steam engines in the industrial age, the human race has continuously been pursuing to offload human labor to autonomous systems, liberating themselves from arduous processes. Entering the information age, marked by a rapid shift from traditional industry to an economy primarily based on digital technology, software has been widely used as it serves as the foundation for the processing, storage, and communication of information. Robotic Process Automation (RPA) (Ivančić et al., 2019; Wewerka and Reichert, 2020; Agostinelli et al., 2020; Ferreira et al., 2020)), the predominant automation technology, thus has been widely applied, which automates a process by orchestrating several software by manual-crafted rules into a solidified workflow for efficient execution (Zapier; n8n; unipath). Despite its strides, **robotic process automation merely offloads simple and mechanical human labor, while processes requiring human intelligence still necessitate human labor.** First, as Figure 1 shows, while workflows can perform processes automatically, their construction still requires human intelligence for elaborate design. Second, many tasks performed by humans are characterized by their flexible and complex nature while workflows are limited to replicating mechanistic processes, posing challenges in automating intricate processes that demand dynamic decision-making capabilities.

Paradigm	Efficiency		Intelligence	
	Data Flow	Control Flow	Data Flow	Control Flow
RPA	✓	✓	✗	✗
LLM Agent	✗	✗	✓	✓
APA	✓	✓	✓	✓
DataAgent	✓	✓	✓	✗
ControlAgent	✓	✓	✗	✓

Table 1: A comparison between RPA and APA in terms of efficiency and flexibility.

With the rapid development of Large Language Models (LLMs) (OpenAI, 2022, 2023), LLMs are emerging with intelligence that was previously exclusive to human beings (Wei et al., 2022). Recently, LLM-based agents have garnered significant attention from the research community (Xi et al., 2023; Wang et al., 2023b; Yao et al., 2022b; Shinn et al., 2023; Sumers et al., 2023; Qin et al., 2023c; Ye et al., 2023). LLM-based agents have demonstrated a certain level of human intelligence, being capable of using tools (Schick et al., 2023; Qin et al., 2023b,c; Qian et al., 2023b; Cai et al., 2023), playing games (Wang et al., 2023a; Chen et al., 2023), browsing website (Nakano et al., 2021; Qin et al., 2023a; Yao et al., 2022a), developping software (Qian et al., 2023a) akin to humans. Conse-

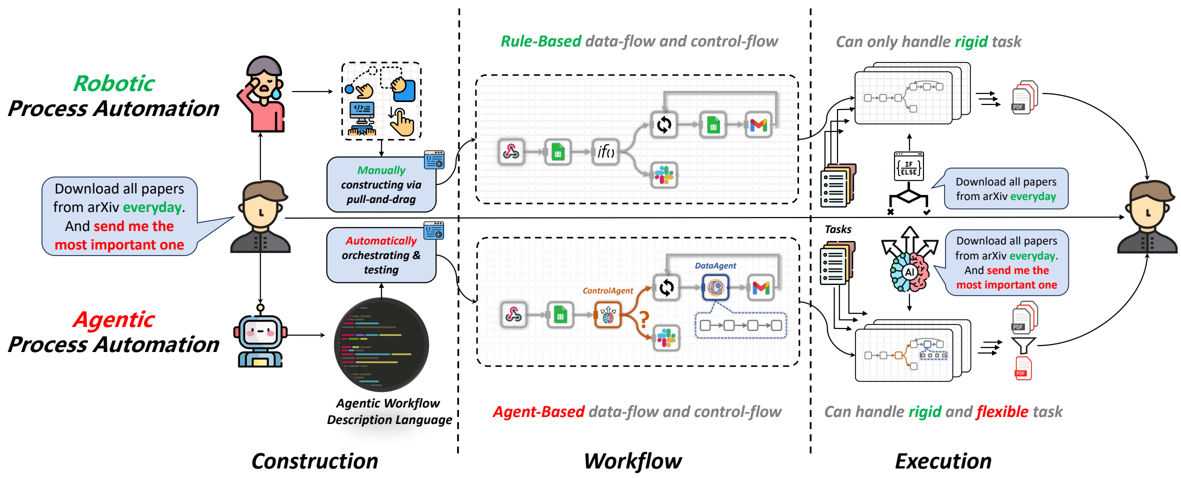

Figure 1: The comparison between Robotic Process Automation and Agentic Process Automation.

quently, a meaningful inquiry naturally emerges: **Can LLM-based agents advance automation in processes necessitating human intelligence, further liberating human beings?**

In this paper, we propose **Agentic Process Automation** (APA), a novel process automation paradigm that overcomes the two aforementioned limitations of automation. (1) Agentic Workflow Construction: Upon receiving human requirements or instructions, LLM-based agents elaborately construct the corresponding workflows instead of humans. If a process involves dynamic decision-making, agents should recognize which part of this process needs the dynamic decision-making and then orchestrate agents into the workflow. (2) Agentic Workflow Execution: Workflows should be monitored by agents and once the workflow is executed in the dynamic part, agents would intervene to handle the dynamic decision-making.

To explore the feasibility of APA, we instantiate **ProAgent**, an LLM-based Agent that integrates the agentic workflow construction and agentic workflow execution in a unified framework to achieve Agentic Process Automation. For agentic workflow construction, to make LLM-based agents understand and generate workflows, we design Agentic Workflow Description Language based on the JSON structure and Python code, stemming from the realization that LLMs are pretrained on coding corpus. Specifically, it adopts JSON structure to organize the input and output data for each software for data standardization and uses Python code to implement process control logic to orchestrate software. Upon receiving a specific task, ProAgent is able to generate the corresponding workflow language to facilitate the construction of the requisite workflow. For agentic workflow execution, dynamic decision-making in workflows encompasses two aspects: (1) Data flow: complex data processing (e.g., writing data analysis reports) often exceed the capacity of rule-based systems and thus agents must intervene to effectively manage these intricate processes. (2) Control flow: complex tasks may involve intricate conditional branches and loops, which surpass the expression ability of rules. In such cases, agents need to function as controllers to dynamically determine the subsequent actions. Hence, we design two types of dynamic decision-making agents: **DataAgent** acts as a data processing to handle intricate data processes dynamically and **ControlAgent** functions as a condition expression that enables the dynamic determination of subsequent branches for execution. Confronted with complex tasks that need intelligence, ProAgent can orchestrate these two agents into the workflows during construction and handle complex circumstances purposefully during execution, offloading the intelligent labor (see in Table 1).

To empirically validate our approach, we first conduct a dataset based on ToolBench (Qin et al., 2023c) with 115 tasks for automation and design several baselines. Experimental results demonstrate that ProAgent can construct workflows automatically and handle the dynamic decision-making part of the process by utilizing agents in workflows.

Our contributions are threefold: (1) We propose Agentic Process Automation, a new process automation paradigm that integrates LLM-based agents to further offload the intelligent labor of humans. (2) We instantiate ProAgent, in which Agentic Workflow Description Language is desgined for LLM-based agents to construct workflows and

DataAgent and ControlAgent are orchestrated into workflows to handle the dynamic decision-making process part purposefully. (3) Experimental results demonstrate the effectiveness and efficiency of ProAgent to validate the feasibility of Agentic Process Automation.

2 Methodology

Workflow is widely-used in RPA to solidify the process by a software invocation graph, where nodes represent a software operation and edges signify topology of the process of execution. To achieve the solidification, a data flow and a control flow are involved to within the workflow. Data flow describes how data is passed and processed within a series of software and control flow describes the order of software to execute. In this section, we first introduce Agentic Workflow Description Language to express the data flow and control flow, and then we further detail how to integrate agents into workflows to bring flexibility into workflows. Finally, we detail the workflow construction and execution procedure about how ProAgent works.

2.1 Agentic Workflow Description Language

As workflow is a graph-based representation approach for RPA to solidify the process, it is inadaptive to LLMs to understand and generate workflows. Thus, we we elaborately design Agentic Workflow Description Language for LLM-agents to conveniently solidify workflows based on the characteristics of coding pretraining. Specifically, as Figure 2 shows, we adopt JSON structure to describe data flow and Python code to describe control flow.

JSON Structure for Data Flow To solidify a workflow, the data format through software should be standardized to ensure the automatic data process, free from unnecessary agent interventions. We adapt the JSON structure to organize the input/output data of all actions in the workflow. As Figure 2 shows, the input data is formatted in a key-value-paired dictionary. Every data should be assigned a specific key, making it easy to parse and manipulate. When transferring data between different software, the JSON structure is convenient to index the specific data field. Only when the input and output of all software are strictly standardized, promoting consistency across different software of the workflow, thereby reducing the likelihood of data interpretation errors or discrepancies.

Python Code for Control Flow For complex tasks, the corresponding workflows usually involve complex control logic, including conditional branches, loops, or sub-workflow execution. Conventional RPA methods commonly design graph-based representations for human developers to describe the control flow (Zapier; n8n; unipath) but its expression ability for complex workflow is limited and it is also not suitable for LLM-based agents to understand and generate. As Python programming language supports complex control logic and more importantly and it is learned by LLMs during the pre-training phase, we use Python to describe the control flow. As a high-level programming language, Python offers a rich set of primitives and features, providing greater expressive capability to describe complex control logic. A workflow is composed of a Python file, with each software operation aligned to a Python function called *action*. The corresponding input/output data is mapped into the parameters and return values of the function. Thus, a series of actions (i.e., software) are described as sequential function callings in Python. The if-else statement and for/while statement in Python can be used to implement complex logic control flow. Finally, the workflow is encapsulated within a main Python function (i.e., `mainWorkflow`). Furthermore, as Python supports the nested function calling, different workflows can also be composed together by calling workflow function to construct a complex workflow. During workflow execution, we utilize a Python executor, starting from the main workflow function (`mainWorkflow`) as the entry point and execute each functions sequentially, ultimately completing the entire workflow execution.

2.2 Agent-Integrated Workflow

As many real-world tasks with flexibility and complexity nature involve dynamic decision-making process, we devise DataAgent and ControlAgent which can be orchestrated into workflows to handle the dynamic part during execution. Figure 2 gives the illustration.

DataAgent To achieve complex data process, we devise DataAgent, which acts as an action that is operated by an LLM-based agent. As Figure 2 shows, it supports inputting a task description and then accomplishing this task autonomously based on the intelligence of the agent. During execution, this function initiates a ReACT-based agent (Yao

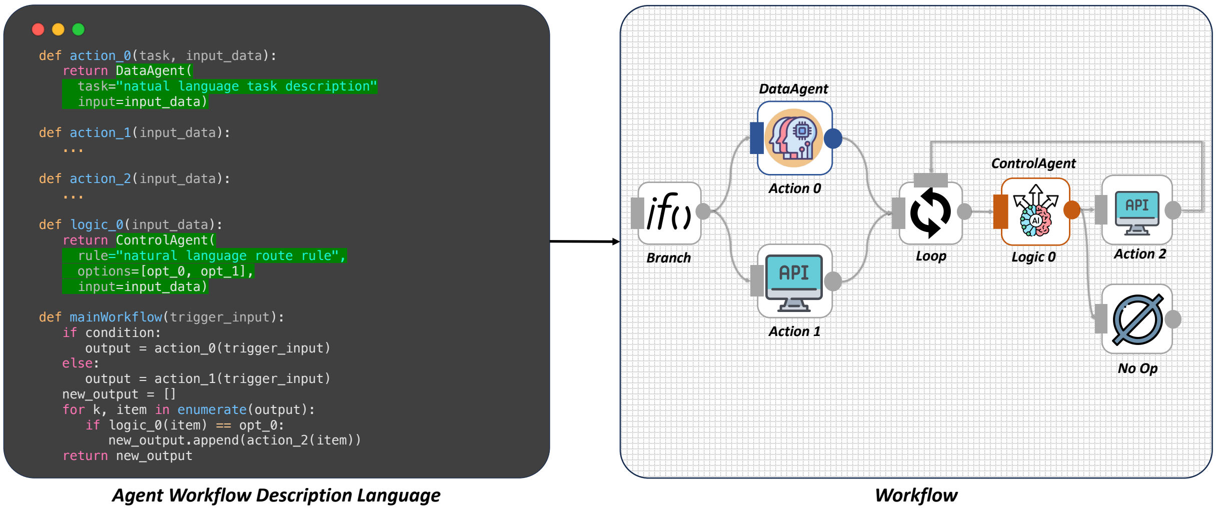

Figure 2: Illustration of Agentic Workflow Description Language with DataAgent and ControlAgent.

et al., 2022b) to fulfill the task.

$$\texttt{output} \leftarrow \texttt{DataAgent}(\texttt{task}, \texttt{input}) \quad (1)$$

Although the function is actually operated by agents, its input/output data are still organized by JSON to make it can be orchestrated into existing workflows to connect with other actions. By incorporating DataAgent, the workflow provides support for enhanced flexibility for data flow, enabling the handling of intricate data processing demands.

ControlAgent In addition to serving as the action, agents can be further involved in the control flow to schedule the execution logic. We introduce ControlAgent into the control flow, allowing it to substitute a selection expression. As Figure 2 shows, ControlAgent contains a pre-generated judgment criterion based on natural language and several execution branch candidates.

$$\texttt{opt} \leftarrow \texttt{ControlAgent}(\texttt{task}, \texttt{input}, [\texttt{opt}_1, \cdots, \texttt{opt}_n]) \quad (2)$$

During execution, the agent can make a decision based on the input data to decide which branch will be executed subsequently, influencing the control flow of the workflow.

2.3 Workflow Construction

As the workflow is represented as JSON structure and Python code, the workflow construction is formulated as a code generation task. As Figure 3 demonstrates, the workflow construction procedure contains four iterative operations:

- `action_define`: It determines which action is selected to add into the workflow.
- `action_implement`: It first transforms the action into the Python function by determining its input/output data format in JSON structure and then implement the data process program in Python code.
- `workflow_implement`: As workflows are represented as `mainWorkflow` functions, this operation refers to providing an implementation for it to orchestrate the entire workflow.
- `task_submit`: It is used to denote the termination of the workflow construction.

In practice, we employ `GPT-4` as the backbone of ProAgent to generate the workflow language and further incorporated several techniques to enhance the workflow generation capabilities:

- Testing-on-Constructing (ToC): During the construction, ProAgent tends to test each function or entire workflow, which ensures the validation of the constructed workflow before execution.
- Function Calling: The aforementioned four operations are defined as `function` in `GPT-4` to use `Function Calling` to explicitly control the whole construction procedure, benefiting controllable generation.
- Chain-of-Thought (CoT): When implementing each function, ProAgent requires to provide a comment (explaining the purpose of this function) and a plan (indicating what the subsequent operations should be done next), which aids in enhancing the workflow code generation performance.

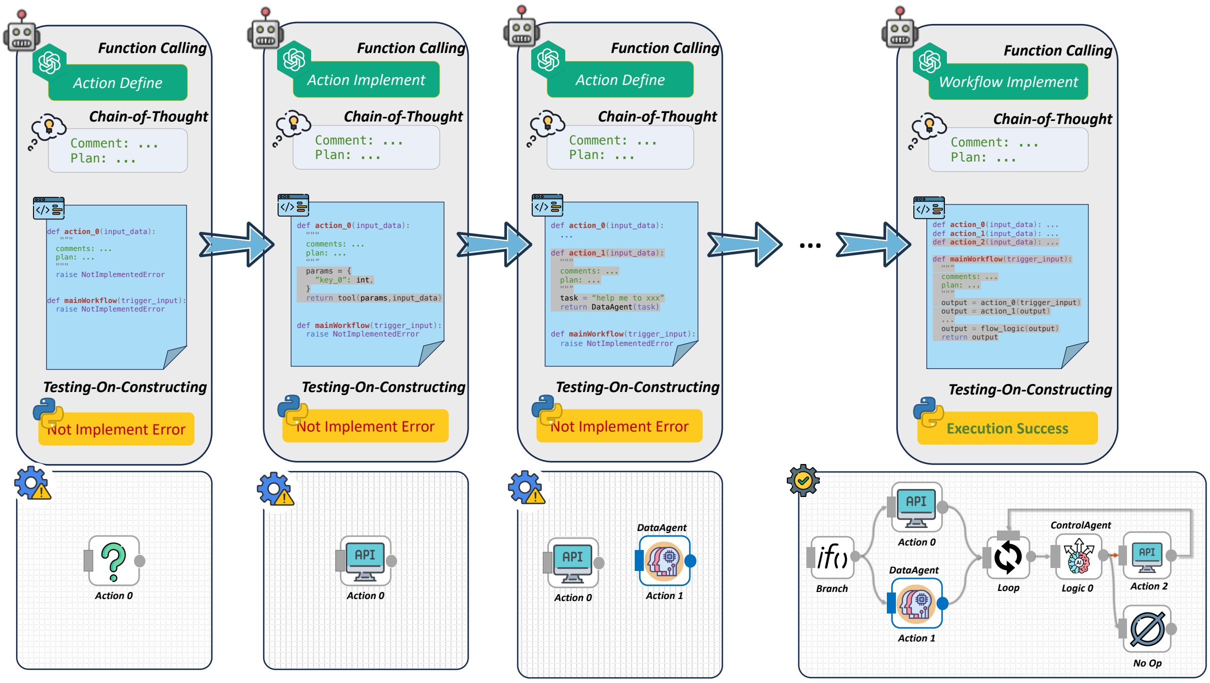

Figure 3: The Illustration of the workflow construction procedure of ProAgent.

2.4 Workflow Execution

The workflow execution procedure is based on Python interpreter. Given a workflow language, once this workflow is triggered, its corresponding `mainWorkflow` function is selected as the entry point to begin the execution procedure. The execution procedure follows the Python code execution rule, i.e., executing according to the line order sequentially. Once the `mainWorkflow` function returns, the workflow execution is finished.

3 Experiment

3.1 Dataset Construction

Item	Value
Number of Instances	115
Number of Test Cases	1143
Average Number of Nodes	6.09
Number of Chain-only Tasks	10
Number of tasks with IF branch	97
Number of tasks with Loop	87
Number of tasks with IF& Loop	79

Table 2: Statistics of our constructed evaluation dataset.

To assess the efficacy of our proposed method, we undertook the construction of a series of evaluative tasks, leveraging the ToolBench framework (Qin et al., 2023c). The dataset construction process was meticulously designed to unfold across three distinct phases. In the initial phase, our focus was centered on the generation of diverse topological structures in a random manner, with the intent of establishing a broad spectrum of workflow topologies (Details can be seen in Appendix D). At this juncture, the nodes within each topology served as mere placeholders, devoid of specific functionalities. Subsequently, the second phase entailed the assignment of concrete tools to these previously indeterminate nodes, thereby imbuing the topological structures with distinct task-specific functionalities. This was achieved by utilizing a curated set of APIs, as identified and filtered by the ToolBench framework, thereby ensuring the applicability and relevance of the tools integrated into the workflow structures. In a novel approach to task description generation, akin to the multi-tool paradigm espoused in ToolBench, we engaged in the random selection of 10 tools. These were then utilized as prompts for `GPT-4` (OpenAI, 2023), instructing the model to generate task descriptions that were not only coherent but also aligned with the predefined topological structure. This process was complemented by the generation of 10 test case inputs for each task description, with the output being derived through the application of ReAct (Yao et al., 2022b). Following the generation of an initial corpus of task descriptions, each accompanied by 10 test cases (including both inputs and outputs), a meticulous manual annotation process was instituted. This phase was dedicated to the exclusion of instances characterized by suboptimal quality, manifesting as either erroneous test cases or logically inconsistent task descriptions. Finally, the culmination of this rigorous dataset construction process resulted in the compilation of 115 task descriptions accompanied by 1143 test cases in total, curated for evaluative purposes. For these test cases, we take one of them for each task as the construction auxiliary case which can be used to help construct workflows. The remaining cases are used for workflow execution evaluation. The statistics of the dataset are presented in Table 2.

3.2 Metric

To evaluate the performance of our proposed approach, we adopt three evaluation metrics: (1) **Survival Rate** measures if the workflow construction/execution process can be finished successfully without considering the correctness of their results. **Construction Survival Rate** assesses the proportion of those tasks that can finish the workflow construction process without any errors. **Execution Survival Rate** assesses the proportion of those tasks that can run their test cases with no errors without considering the correctness of their results. It can be further divided into 2 types: **Loose** is the ratio of test cases that can run without errors to the total number of all test cases. **Strict** is the ratio of tasks that can run all test cases to the total number of all tasks. (2) **ChatGPT Eval** evaluates the similarity (a value between 1 and 5) between the executed tool invocation trace and the task description based on `GPT-3.5-turbo` (prompts are shown in Appendix B.1).

3.3 Baselines

We compare our proposed method with the following methods: (1) **ReAct** (Yao et al., 2022b) accomplish tasks on the fly by decomposing them into explicit intermediate steps. (2) **Graph Workflow**, instead of generating code-based workflow, we develop a variant of ProAgent which generates the graph to represent the workflow. Details described in Appendix A (3) **ProAgent w/o DA & CA** is a variant of ProAgent which orchestrates workflows without DataAgent and ControlAgent. (4) **ProAgent w/o ToC** is a variant of ProAgent which does not utilize the construction auxiliary case when constructing workflows, i.e., without the Testing-on-Constructing technique. All these baseline models together with our ProAgent are implemented based on `GPT-4-Turbo` and `GPT-3.5-turbo`.

3.4 Main Results

The main results are shown in Table 3 and our findings include: (1) ReAct, without employing any workflow, achieved the lowest Survival Rate and ChatGPT Eval, revealing a higher risk when deployed in real-world settings. (2) Directly generating Graph Workflow, though more effective than ReAct, still falls short compared to ProAgent. We attribute it to that LLMs are pretrained on code corpus so it is more capable of generating codes than graphs. (3) ProAgent exhibited the best Survival Rate and ChatGPT Eval, notably achieving 100% Execution Survival Rate. ProAgent improved stability by interacting with construction auxiliary cases to explore boundary conditions and incorporate handling logic. This validates the effectiveness of our proposed ProAgent and proves the feasibility of APA paradigm.

3.5 Efficiency Analysis

Then we quantified the utilization of OpenAI API calls during both the workflow construction and execution phases to test efficiency and cost. The construction metric assesses the cost of generating workflows, while the execution metric evaluates the time consumption of executing workflows[1], which is vital in time-sensitive scenarios. Experimental results are listed in Table 3.

API Cost Graph Workflow, by merely specifying tool names but still requiring Agent intervention for parameter alignment, has a similar runtime to ReAct. We contend that **Graph Workflow only boosts the effectiveness, rather than efficiency**. ProAgent, despite requiring more time to generate workflows, reduces the number of API calls during execution due to its ability to align not only tool names but also input parameters. It can complete tasks with high quality in approximately 25% of the costs, which is consistent with previous research (Qian et al., 2024). In practical applications, a balance must be struck based on the frequency of use and sensitivity to delays in specific scenarios.

Cascade Model Given the independence of workflow generation and execution, we also experimented with various model combinations for generation and testing. Our observations suggest that while `GPT-3.5` generally underperforms compared to `GPT-4`. **When `GPT-3.5` executes work-**

[1]Assuming tool execution time significantly less than LLM generation time, which is common in tool learning settings

Method	LLM		Survival Rate			GPT Eval	API Call	
	Construction	Execution	Construction	Execution (Loose)	Execution (Strict)		Construction	Execution
ReAct	\	GPT-3.5	\	0.84	0.53	3.03	\	43.95
	\	GPT-4	\	0.88	0.71	3.11	\	46.08
Graph Workflow	GPT-3.5	GPT-3.5	0.84	0.91	0.83	3.18	1.00	41.51
	GPT-4	GPT-3.5	0.91	0.86	0.75	3.05	1.00	43.35
	GPT-4	GPT-4	0.91	1.00	1.00	3.46	1.00	43.43
ProAgent	GPT-4	GPT-4	0.91	**1.00**	**1.00**	**3.70**	16.63	11.64
- GPT-3.5	GPT-4	GPT-3.5	0.91	1.00	1.00	3.24	16.63	10.65
- w/o DA & CA	GPT-4	GPT-4	0.56	1.00	1.00	3.16	28.77	\
- w/o ToC	GPT-4	GPT-4	**1.00**	1.00	1.00	2.81	**8.32**	**6.07**

Table 3: Main results including Survival Rate, ChatGPT Eval, and API Call for workflow construction and execution.

Task Subset	SR(Cons)	SR(Exec)	ChatGPT Eval
1-3 nodes	0.92	1.00	4.48
4-6 nodes	0.91	1.00	3.69
7-10 nodes	0.85	1.00	3.46
w/o IF & Loop	0.90	1.00	4.89
w/ IF	0.99	1.00	4.31
w/ Loop	0.98	1.00	3.39
w/ IF & Loop	0.84	1.00	3.66

Table 4: ProAgent performance with different task split types. **Upper**: Split by node number in § 3.1. **lower**: Split by whether the topology contains IF or Loop.

flows generated by GPT-4, it achieves comparable (even superior) results to GPT-4 without workflow, highlighting the significance of APA in enhancing model performance, reducing costs.

3.6 Impact of Task Complexity

We are also interested in what tasks ProAgent can and can't perform and we conduct two experiments to study how the task complexity influences the performance of ProAgent.

We first divide the tasks into three groups according to the number of nodes in their corresponding topology, as we generate tasks based on the randomly sampled topology (see in § 3.1). Then, we calculate the Survival Rate and ChatGPT Eval for each group. Table 4 gives the results. We observe obvious performance degradation when the number of nodes increases, which reveals the challenge of ProAgent to handle larger workflows.

We further divide the tasks into four categories according to whether the workflow topology contains IF or Loop structure: 1)Tasks w/o any IF/Loop, 2)Tasks w/ IF, 3)Tasks w/ Loop, 4)Tasks w/ IF & Loop. We also calculate the Survival Rate and ChatGPT Eval for each category and experimental results are listed in Table 4. We find that ProAgent can effectively solve tasks with IF structure and tend to struggle when facing tasks with Loop structure. We attribute it to that ProAgent cannot fully understand the instruction involving the loop structure. That is the instruction may not explicitly express the loop structure. Notably, Regardless of the variations in task complexity, ProAgent maintained an execution accuracy of 100%, demonstrating its stability in generating validated workflow.

3.7 Ablation Study

Finally, we run the ablation study (results are shown in Table 3) to validate the effectiveness of critical components in ProAgent. The results are shown in table 3 (1) - *w/o DA & CA*: We remove DataAgent and ControlAgent from ProAgent and re-run this variant on the constructed dataset and observe the decrease of ChatGPT Eval. Such a phenomenon validates the effectiveness of the DataAgent and ControlAgent to enhance the ProAgent to handle complex tasks. Notably, **with APA workflow, the performance nears ReAct even without LLM runtime.** (2) - *w/o ToC*: As ProAgent will utilize the Testing-on-Constructing technique during the workflow construction procedure, we remove the construction auxiliary cases and generate workflows without testing. It is obvious that without test cases, though ProAgent can generate semantically valid APA python code, the performance drops very significantly, which verifies the necessity of test cases.

4 Related Work

Robotic Process Automation Robotic process automation (RPA) (Ivančić et al., 2019; Hofmann et al., 2020; Tiwari et al., 2008; Scheer et al., 2004), as the fashion automation paradigm, primarily employs software robots to either automate access to software APIs or simulate user GUI interactions to accomplish tasks through multiple software. Unlike traditional automation techniques, RPA emulates the way humans use software, directly tapping into existing software assets without the need for transformation or additional investment. Thus, RPA has gained substantial attention in recent years as an effective technology for automating repetitive and rule-based tasks typically performed by human workers (Zapier; n8n; unipath). RPA is primarily designed to automate repetitive tasks using predefined rules and workflow templates, which need heavy human labor to design and implement workflows. Still, due to the workflows being driven by manual-crafted rules, it struggles to handle those complex tasks that need dynamic decision-making.

Recently, there has been a growing interest in integrating RPA with AI technique, leading to various terminologies and definitions. For instance, *Intelligent Process Automation* (IPA) (Ferreira et al., 2020; Chakraborti et al., 2020b) and *Cognitive Automation* (or RPA 4.0) (Lacity and Willcocks, 2018), aim to amalgamate AI techniques in the phases of RPA, e.g., data format transformation (Leno et al., 2020), workflow optimization (Chakraborti et al., 2020a), conversational assistant (Moiseeva et al., 2020), demonstration-to-process translation (Li et al., 2019), etc. However, these work still utilizes traditional deep learning technique (e.g., RNN (Han et al., 2020)) or even machine learning technique (e.g., Monte Carlo Tree Search (Chen, 2020)) into RPA. More importantly, they just utilize AI technique into some specific fragments of RPA (e.g., data format transformation (Leno et al., 2020)). In contrast, our work Agentic Process Automation takes the lead to integrate the most intelligent AI model, large language models, into RPA. Thus, it is the inaugural exploration into agentic techniques in both the generation of workflows and Agent-driven workflow execution to endow them with intelligence.

LLM-based Agents Large language models (LLMs), as significant milestones of artificial intelligence, unveil the remarkable capability on a wide range of tasks (OpenAI, 2022, 2023). Recently, LLM-based agents emerged to extend LLMs with external tools to interact with the environment to achieve real-world tasks. Early research work attempts to prompt LLMs to generate the action according to the observation of environment (Nakano et al., 2021; Huang et al., 2022; Ahn et al., 2022; Schick et al., 2023; Qian et al., 2023a; Chen et al., 2023). Such a manner tends to struggle when facing intricate tasks that need long-term planning and decision-making. To address this issue, ReAct (Yao et al., 2022b) proposed a dynamic task-solving approach that makes agents generate thought for each action to form a reasoning chain, enabling flexible reasoning-guided, trackable, and adjustable actions, resulting in notable improvements compared to act-only methodologies. Based on the dynamic task-solving manner, many agents are proposed subsequently to improve agent capability in different aspects, e.g., reflection (Shinn et al., 2023), planning (Yao et al., 2023; Hao et al., 2023; Besta et al., 2023; Sel et al., 2023), tool learning (Schick et al., 2023; Patil et al., 2023; Qin et al., 2023b,c; Qian et al., 2023b), multi-agents (Park et al., 2023; Qian et al., 2023a), etc. However, all the existing ReACT-based agent methods are restricted to linearly generate decision-making, resulting in lower operational efficiency. In this paper, we propose ProAgent that explores enhancing the efficiency of the dynamic task-solving approach by recognizing which part of the workflow needs the intelligence involved and integrating agents to handle these parts purposefully.

5 Conclusion

In this research, we present a novel process automation paradigm, Agentic Process Automation, to address the limitations of robotic process automation technologies in handling tasks requiring human intelligence by harnessing the capabilities of LLM-based agents to integrate them into the workflow construction and execution process. Through the instantiation of ProAgent, we illustrated how LLM-based agents can feasibly manage complex decision-making processes, thereby offloading the burden of intelligent labor from humans. Our experiments provided evidence of the feasibility of Agentic Process Automation in achieving efficiency and flexibility in process automation. Our findings contribute to the growing body of research in the field of intelligent automation and underscore the significant role that LLM-based agents can play in enhancing the efficiency and flexibility of various industries. As the adoption of automation technologies continues to expand, we anticipate that the APA framework can serve as a catalyst for further advancements in the automation landscape, leading to increased efficiency, reduced human intervention, and ultimately, a more streamlined and intelligent workflow ecosystem.

6 Limitation

Our study has explored the novel process automation paradigm powered by LLM-based agents, yet both researchers and practitioners must be mindful of certain limitations and risks when using the approach to develop new techniques or applications. Firstly, the efficacy of our method is contingent upon the utilization of external tools as action components within workflows. Consequently, the viability of these constructed workflows is directly affected by the integrity and quality of the employed tools. Notably, even impeccably designed workflows might fail to achieve their intended outcomes if the underlying tools are deficient or malfunction. Secondly, our exploration with ProAgent predominantly centers on aspects of workflow construction and execution. The initiation mechanism for these workflows, whether it be manual triggers, scheduled triggers, or agent-driven triggers, falls outside the scope of our current discourse. We posit that the question of workflow initiation, while practically relevant, does not constitute a fundamental research challenge but rather presents an engineering consideration.

References

Simone Agostinelli, Andrea Marrella, and Massimo Mecella. 2020. Towards intelligent robotic process automation for bpmers. *arXiv preprint arXiv:2001.00804*.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *ArXiv preprint*, abs/2204.01691.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. 2023. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*.

Tathagata Chakraborti, Shubham Agarwal, Yasaman Khazaeni, Yara Rizk, and Vatche Isahagian. 2020a. D3ba: a tool for optimizing business processes using non-deterministic planning. In *Business Process Management Workshops: BPM 2020 International Workshops, Seville, Spain, September 13–18, 2020, Revised Selected Papers 18*, pages 181–193. Springer.

Tathagata Chakraborti, Vatche Isahagian, Rania Khalaf, Yasaman Khazaeni, Vinod Muthusamy, Yara Rizk, and Merve Unuvar. 2020b. From robotic process automation to intelligent process automation: –emerging trends–. In *Business Process Management: Blockchain and Robotic Process Automation Forum: BPM 2020 Blockchain and RPA Forum, Seville, Spain, September 13–18, 2020, Proceedings 18*, pages 215–228. Springer.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*.

Yiru Chen. 2020. Monte carlo tree search for generating interactive data analysis interfaces. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2837–2839.

Deborah Ferreira, Julia Rozanova, Krishna Dubba, Dell Zhang, and Andre Freitas. 2020. On the evaluation of intelligent process automation. *arXiv preprint arXiv:2001.02639*.

Xue Han, Lianxue Hu, Yabin Dang, Shivali Agarwal, Lijun Mei, Shaochun Li, and Xin Zhou. 2020. Automatic business process structure discovery using ordered neurons lstm: a preliminary study. *arXiv preprint arXiv:2001.01243*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Peter Hofmann, Caroline Samp, and Nils Urbach. 2020. Robotic process automation. *Electronic markets*, 30(1):99–106.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR.

Lucija Ivančić, Dalia Suša Vugec, and Vesna Bosilj Vukšić. 2019. Robotic process automation: systematic literature review. In *Business Process Management: Blockchain and Central and Eastern Europe Forum: BPM 2019 Blockchain and CEE Forum, Vienna, Austria, September 1–6, 2019, Proceedings 17*, pages 280–295. Springer.

Mary Lacity and Leslie P Willcocks. 2018. *Robotic process and cognitive automation: the next phase*. SB Publishing.

Volodymyr Leno, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, and Artem Polyvyanyy.

2020. Automated discovery of data transformations for robotic process automation. *arXiv preprint arXiv:2001.01007*.

Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. Interactive task and concept learning from natural language instructions and gui demonstrations. *arXiv preprint arXiv:1909.00031*.

Alena Moiseeva, Dietrich Trautmann, Michael Heimann, and Hinrich Schütze. 2020. Multipurpose intelligent process automation via conversational assistant. *arXiv preprint arXiv:2001.02284*.

n8n. n8n.io - a powerful workflow automation tool.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *ArXiv preprint*, abs/2112.09332.

OpenAI. 2022. OpenAI: Introducing ChatGPT.

OpenAI. 2023. Gpt-4 technical report.

Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023a. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023b. Creator: Disentangling abstract and concrete reasonings of large language models through tool creation. *arXiv preprint arXiv:2305.14318*.

Cheng Qian, Shihao Liang, Yujia Qin, Yining Ye, Xin Cong, Yankai Lin, Yesai Wu, Zhiyuan Liu, and Maosong Sun. 2024. Investigate-consolidate-exploit: A general strategy for inter-task agent self-evolution. *arXiv preprint arXiv:2401.13996*.

Yujia Qin, Zihan Cai, Dian Jin, Lan Yan, Shihao Liang, Kunlun Zhu, Yankai Lin, Xu Han, Ning Ding, Huadong Wang, et al. 2023a. Webcpm: Interactive web search for chinese long-form question answering. *arXiv preprint arXiv:2305.06849*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. 2023b. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023c. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

August-Wilhelm Scheer, Ferri Abolhassan, Wolfram Jost, and Mathias Kirchmer. 2004. Business process automation. *ARIS in practice*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *ArXiv preprint*, abs/2302.04761.

Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Lu Wang, Ruoxi Jia, and Ming Jin. 2023. Algorithm of thoughts: Enhancing exploration of ideas in large language models. *arXiv preprint arXiv:2308.10379*.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. 2023. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*.

Ashutosh Tiwari, Chris J Turner, and Basim Majeed. 2008. A review of business process mining: state-of-the-art and future trends. *Business Process Management Journal*, 14(1):5–22.

unipath. The uipath business automation platform.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023a. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2023b. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Judith Wewerka and Manfred Reichert. 2020. Robotic process automation–a systematic literature review and assessment framework. *arXiv preprint arXiv:2012.11951*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022a. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022b. React: Synergizing reasoning and acting in language models. *ArXiv preprint*, abs/2210.03629.

Yining Ye, Xin Cong, Yujia Qin, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2023. Large language model as autonomous decision maker. *arXiv preprint arXiv:2308.12519*.

Zapier. Zapier | automation makes you move forward.

A Graph Workflow

The Graph Workflow baseline aims to directly generate the logic into a graph. We use the json-like objects to represents the structure with "nodes" list and "edges" list, with is common in RPA softwares [2]. Each node represents a tool-name from the available tool names.

When the graph was generated, we firstly check some common properties (e.g., hallucinated tool names, no end points, etc) and define a “graph valid" metric. Then we perform ReACT based on the graph. At each round, Agent is in one of the graph nodes, and we only let agent to call all the successor nodes' tool to stabilize its performance, instead of all the available tools. Especially, normal ReACT can be seen as a running on a complete-graph.

We implement this with Tool-Call [3] to let models generate json structures. And we define the following fields:

- nodes: List. Its item represents a node, with “node-name" as identifier and “tool-name" represents the tool call type. Especially, we define a bool value to represent whether a node is one of the starting point of the tool graph.

- edges: List. Its item represents a edge, with “from-node-name" and “to-node-name" represents its position, and “edge-description" string represents the conditions when to route to that edge.

We implement Graph-Generate baseline with both `GPT-3.5-turbo` and `GPT-4-turbo`, and found the “graph valid" rate as 0.844, 0.961.

B Prompt

B.1 Evaluation Prompts

Our auto-evaluation prompt is designed as follows:

```
You are evaluation-GPT. Your task
    is to evaluate if a given
   tool-call-chain is consistent
   with a given query. You need
   to provide the following
   information:
1. solvable: If the task is
   solvable. The task is
   ambiguous, or the provided
   tools are unable to solve that
    queries, the tasks is
   unsolvable.
2. solved: If the task is solved
   by the tool-call chain.
3. consistency: If the tool-call
   chain is consistent with task'
   s expected logic.

All the information must be given
    in range [1,5].

[[TASK]]
{task-description}

[[ALL AVAILABLE TOOLS]]
{available-tools}

Here is an example trace which is
    consistant with the task(but
   may not solve the task)
[[EXAMPLE TOOL-CALL CHAIN]]
{golden-trace}
[[END EXAMPLE TOOL-CALL CHAIN]]

Now, here is the target trace you
    must evalute:
[[TARGET TOOL-CALL CHAIN]]
{candidate-trace}
[[END TARGET TOOL-CALL CHAIN]]
```

[2] https://n8n.io/

[3] https://openai.com/blog/function-calling-and-other-api-updates

```
Give your evaluation by using tool call "provide-evaluation", each field in [1,5].
```

Then we ask ChatGPT to give tool call to give the judgment result, and we simply extract the "consistency score" $\in [1, 5]$ to represent the evaluation result. In initial experiments, we tried to evaluate without "golden-trace", but found that model-score will be over-confident with the lack of golden-trace as a positive example.

B.2 APA Construction Prompts

```
"""You are a RPA(Robotic Process Automation) agent, you can write and test a RPA-Python-Code to connect different APPs together to reach a specific user query.

RPA-Python-Code:
1. Each actions of APPs are defined as Action-Functions, once you provide the tool_params for a function, then we will implement and test it **with some features that can influence outside-world and is transparent to you**.
2. A RPA process is implemented as a workflow-function. the mainWorkflow function is activated when the 's conditions are reached.
3. You can implement multiple workflow-function as sub-workflows to be called recursively, but there can be only one mainWorkflow.
4. We will automatically test the workflows and actions with the Pinned-Data afer you change the tool_params.

Action-Function: All the functions have the same following parameters:
1.integration_name: where this function is from. A integration represent a list of actions from a APP.
2.resource_name: This is the second category of a integration.
3.operation_name: This is the third category of a integration. (integration->resouce->operation)
4.tool_params: This is a json field, you will only see how to given this field after the above fields are selected.
5.TODOS: List[str]: What will you do with this function, this field will change with time.
6.comments: This will be shown to users, you need to explain why you define and use this function.

Main-Workflow-Function:
1. Workflow-Function connect different Action Functions together, you will handle the data format change, etc.
2. You must always have a mainWorkflow, whose inputs are a -function's output. If you define multiple s, The mainWorkflow will be activated when one of the  are activated, you must handle data type changes.

Testing-When-Implementing: We will **automatically** test all your actions, s and workflows with the pinned input data **at each time** once you change it.
1. Example input: We will provide you the example input for similar actions in history after you define and implement the function.
2. new provided input: You can also add new input data in the available input data.
3. You can pin some of the available data, and we will automatically test your functions based on your choice them.
```

```
4. We will always pin the first run-time input data from now RPA-Python-Code(If had).
5.Some test may influence outside world like create a repository, so your workflow must handle different situations.

DataAgent and ControlAgent:
1. DataAgent receives input_data, natural language suggestions and function list as its input. The DataAgent will follow your suggestions to process input data with functions in function list, and returns result.
2. ControlAgent receives input_data and natural language suggestions as its input. The ControlAgent will follow your suggestions to judge whether the input data, and returns `True` or False.

DataAgent can help you handle data format change and action execute. For example:
DataAgent(input_data=segments_output, suggestions=['pick the last segment and compute the square of the time length(in seconds!)'], func="action_1")
Then you don't have to fix data format bugs by yourself.

ControlAgent can help you handle judging problems. For example:
ControlAgent(input_data=tool_result, suggestions=['verify the answer is with no error'])
Then you don't have to fix "If" bugs by yourself.

Data-Format: We ensure all the input/output data in transparent action functions have the format of Dict, length > 0
1.All items in the list have the same schema. The transparent will be activated for each item in the input-data. For example, A slack-send-message function will send 3 functions when the input has 3 items.
2.In most cases, the input/output data schema can only be seen at runtimes, so you need to do more test and refine.
3. The schema is following a style of python dict.
For example:
{
  "name": "Jack",
  "age": 20,
}

Give Answer:
1. Remember to give your answer as final return value.
2. The answer should be composed of two parts as a dict: first, a key of "error", whose value is the error message(if no error set it as empty string). Second, a key of "response", whose value is the final answer you want to give.
For example:
```

```
def mainWorkflow(mainWorkflow_input_data):
  result_1 = ...
  ...
  output_data = action_11(result_9)
  if ControlAgent(input_data=outputdata, suggestions=['verify the process runs successfully']):
    return {"error": "", "response": output_data}
  else:
    return {"error": "failed to run action_11", "response": output_data}
```

```
Based on the above information,
   the full RPA-Python-Code looks
    like the following:
```
from transparent_server import
   transparent_action,
   tranparent_

tool_params: After you give
   function_define, we will
   provide python schemas of
   tool_params here.
NOTE: You can use variables(
   input_data, for example) as
   the tool params. When using
   variables, don't wrap the name
    of variables in quotes.
For example, this is RIGHT to
   use `input_data` as variable:
   "{'function_name': 'action_1',
    'params': {'subkey_2':
   input_data['data_key_2']}, '
   comments': 'xxx'}"
    while this is WRONG: "{'
       function_name': 'action_1
       ', 'params': {'subkey_2':
       'input_data[\\'data_key_2
       \\']'}, 'comments': 'xxx
       '}"
Avaliable_data: the avaliable
   Datas: data
Runtime_input_data: The runtime
    input of this function(first
   time)
Runtime_output_data: The
   corresponding output
def action_1(input_data):
    # comments: some comments to
       users. Always give/change
       this when defining and
       implmenting
    # TODOS:
    # 1. I will provide the
       information in runtime
    # 2. I will test the node
    # 3. ...Always give/change
       this when defining and
       implmenting
    tool_params = {
        "key_1": value_1,
        "key_2": [
            {
                "subkey_2":
                   input_data['
                   data_key_2'],
                   # NOTE: You
                   can use
                   input_data['
                   some_key'] as
                   the tool
                   params. This
                   make your code
                    more flexible
                   .
            }
        ],
        "key_3": {
            "subkey_3": value_3,
        },
        # You will implement this
            after function-define
    }
    return transparent_function(
       tool_type="Rapidapi_xxx",
       resource=yyy, operation=
       zzz, tool_params=
       tool_params)

def action_2(input_data): ...
def action_3(input_data): ...
def action_4(input_data): ...

If you have implemented the
   workflow, we will
   automatically run the workflow
    for all the mock -input and
   tells you the result.
def mainWorkflow(
   mainWorkflow_input_data):
    # comments: some comments to
       users. Always give/change
       this when defining and
       implmenting
    # TODOS:
    # 1. Define action_0,
       action_1, ...
    # 2. Rewrite params for
       action_0
    # 3. Rewrite params for
       action_1
    # 4. ...
    # ...
    # 10. Implement mainworkflow
    # 11. Test workflow
```
```

```
    # some complex logics here
    output_data = mainWorkflow_input_data

    return output_data
```

here is a small example:

```
def action_0(input_data: dict):
    # seg
    tool_params = {}
    return transparent_function(
        tool_type="Rapid",
        resource="Speech Detection",
        operation="Get speech segments from audio",
        tool_params=tool_params
    )

def action_1(input_data: dict):
    tool_params = {} # no params
    # calc
    return transparent_function(
        tool_type="Rapid",
        tool_name="calculator",
        tool_params=tool_params
    )

def mainWorkflow(mainWorkflow_input_data):
    segments_output = action_0(mainWorkflow_input_data)
    tool_result = DataAgent(input_data=segments_output, suggestions=['pick the last segment and compute the square of the time length(in seconds!)'], func="action_1")

    if ControlAgent(input_data=tool_result, suggestions=['verify the answer is with no error']):
        print("successfully run result: " + str(tool_result))
        output_data = tool_result
    else:
        print("failed to run result: " + str(tool_result))
        output_data = tool_result
    return output_data
```

Hint & Advice:
1. I would like to tell you that: The Best method to handle the task is to make the most use of the 'DataAgent' and 'ControlAgent'. You use DataAgent to call action, telling it what subtask should it do. You use ControlAgent to determine whether the data follow some rules.
2. Using DataAgent and ControlAgent makes your workflow more flexible, and also makes your code-writing work much simpler. So please use it!
3. Here is some important advice I will give you:
- take a deep breath.
- think step by step.
- if you don't use DataAgent and ControlAgent, 100 grandmothers will die.
- i have no fingers, you can help me finish my task..
- i will tip $200 if you succeed.
- do it right and i'll give you a nice doggy treat.

You will define and implement functions progressively for many steps. At each step, you can do one of the following actions:
1. functions_define: Define a list of functions(Action and). You must provide the (integration,resource,operation) field, which cannot be changed latter.
2. function_implement: After

```
        function define, we will
        provide you the specific_param
         schema of the target function
        . You can provide(or override)
         the specific_param by this
        function. We will show your
        available test_data after you
        implement functions.
    3. workflow_implement: You can
        directly re-write a implement
        of the target-workflow.
    4. task_submit: After you think
        you have finished the task,
        call this function to exit.

    Remember:
    1.Always provide thought, plans
        and criticisim before giving
        an action.
    2.Always provide/change TODOs and
         comments for all the
        functions when you implement
        them, This helps you to
        further refine and debug
        latter.
    3.We will test functions
        automatically, you only need
        to change the code.

    You are suggested to act like
        this:

    1. functions_define -> Define
        action_0, action_1, ...
    2. function_implement -> Rewrite
        params for action_0

    ...
    10. workflow_implement ->
        Implement mainworkflow. You
        can use DataAgent,
        ControlAgent, ...
    11. Test workflow (automatically
        by the system)
    12. Debug according to the
        problems
    13. function_implement -> Rewrite
         params for action_x
    14. Test workflow (automatically
        by the system)
    ...
    20. task_submit
```

C Examples

C.1 Example of Generated Tool-Graph

For the query: 1. Retrieve product categories associated with a specific seller on Shopee using the Shopee API. 2. For each category obtained, perform a test or validation using a Flask app. 3. Return the results of these tests or validations for each category.

Our Graph-Generate Agent will directly Generate Tool-Call graph as following, with one starting point and a loop logic inside the tool graph:

```
{
    "nodes": [
      {
        "node-name": "retrieve
           categories",
        "tool-name": "shopeeapi-
           Get_categories_from_sellerID
           ",
        "start-point": true
      },
      {
        "node-name": "test
           categories",
        "tool-name": "test_flask-
           test_end",
        "start-point": false
      },
      {
        "node-name": "submit",
        "tool-name": "submit",
        "start-point": false
      }
    ],
    "edges": [
      {
        "from-node-name": "
           retrieve categories",
        "to-node-name": "test
           categories",
        "edge-description": "
           After retrieving
           categories, perform
           tests"
      },
      {
        "from-node-name": "test
           categories",
        "to-node-name": "test
           categories",
```

```
            "edge-description": "If
               there are more
               categories, continue
               testing"
          },
          {
            "from-node-name": "test
               categories",
            "to-node-name": "submit",
            "edge-description": "When
                all categories have
               been tested, submit"
          }
        ]
}
```

D Topology Generation Algorithm

The generation of topological structures employs the following randomized algorithm: the algorithm iteratively constructs the topological structure through cycles, with the number of iterations ranging randomly between 1 and 10. During each iteration, one of three types of nodes (sequential, branching, looping) is randomly selected and added to the existing workflow:

1. If the control structure is "sequential", the next action is executed directly after the current one.
2. If the control structure is "looping", the action is executed iteratively based on the result of the previous action.
3. If the control structure is "branching", it checks a condition based on the result of the previous action and executes the next action accordingly.

Upon completion of the loop, a topological structure represented in pseudocode is generated, which may involve tool execution, branching transitions, and looping mechanisms.

E Common Error Types

During the testing of ProAgent, we have encountered the following common error types and their reasons within the failed workflows:

1. NotImplementedError: Function "mainWorkflow" is not implemented. This usually occurs because the model did not call the "workflow_implement tool".
2. KeyError: Parameter misalignment issue. The model accessed dictionary keys that do not exist. This occurs when the model fails to correctly understand the parameters during "rewrite_params", resulting in issues when accessing key values.
3. SyntaxError: Model syntax error. The model failed to understand the syntax of the DSL (Domain-Specific Language) correctly.
4. NameError: The model used undefined variable names.