ReAct Meets Industrial IoT: Language Agents for Data Access

James T Rayfield, Shuxin Lin, Nianjun Zhou, Dhaval Patel
IBM T.J. Watson Research Center, NY, USA
{jtray@us., shuxin.lin@, jzhou@us., pateldha@us.}ibm.com

Abstract

We present a robust framework for deploying domain-specific language agents that can query industrial sensor data using natural language. Grounded in the Reasoning and Acting (ReAct) paradigm, our system introduces three key innovations: (1) integration of the `Self-Ask` method for compositional, multi-hop reasoning; (2) a multi-agent architecture with `Review`, `Reflect` and `Distillation` components to improve reliability and fault tolerance; and (3) a long-context prompting strategy leveraging curated in-context examples, which we call *Tiny Trajectory Store*, eliminating the need for fine-tuning. We apply our method to Industry 4.0 scenarios, where agents query SCADA systems (e.g., SkySpark) using questions such as, "How much power did B002 AHU 2-1-1 use on 6/14/16 at the POKMAIN site?" To enable systematic evaluation, we introduce **IoTBench**, a benchmark of 400+ tasks across five industrial sites. Our experiments show that ReAct-style agents enhanced with long-context reasoning (`ReActXen`) significantly outperform standard prompting baselines across multiple LLMs, including smaller models. This work repositions NLP agents as practical interfaces for industrial automation. The code/data can be found here.

1 Introduction

The deployment of AI agents to automate complex workflows is accelerating across industries (Jabbour and Reddi, 2024). Recent systems such as SWE-Agent (Jimenez et al., 2023; Tao et al., 2024; Wang et al., 2024) and AIOps-Agent (Shetty et al., 2024) demonstrate the potential of autonomous agents in software engineering and cloud operations. A common foundation for these agents is the **ReAct** framework (Yao et al., 2023; Shinn et al., 2023), which enables agents to interleave reasoning and acting in response to real-time feedback. While ReAct-based systems are promising, adapting them to domain-specific applications, particularly in **Industry 4.0**, poses new challenges related to reliability, domain adaptability, interpretability, and tool integration.

Industry 4.0 focuses on the automation and monitoring of physical assets, such as chillers, compressors, and air handling units (AHUs), to enable predictive maintenance and energy optimization (Yang et al., 2022; Nikitin and Kaski, 2022). In this context, we introduce an **IoT agent** designed to query historical/real-time sensor data from SCADA systems (e.g., SkySpark (Skyspark, 2024)) using natural language. For example, a query such as "How much power was B002 AHU 2-1-1 using on 6/14/16 at the POKMAIN site?" illustrates the agent's intended interaction mode. Even such short queries require reasoning over multiple interdependent domain concepts: assets, sensors, timestamps, and locations, highlighting the need for **tool-aware** and **context-sensitive** language agents.

To meet these demands, we develop a ReAct-centered **multi-agent framework** tailored for industrial environments. Our system integrates three core capabilities:

- **Tool Integration**: Support for complex, multi-parameter industrial tools beyond generic APIs (Langchain, 2024b).
- **Contextual Reasoning**: Handling domain-specific reasoning tasks, such as sensor disambiguation, entity grounding and mathematical reasoning tasks such as last week, max value.
- **Model Evaluation**: Benchmarking LLMs on real-world data access tasks to assess reliability, reasoning depth, and domain robustness.

Our approach demonstrates that language agents can act as effective, domain-aware interfaces for structured industrial data, bridging natural language understanding with sensor-driven systems in Industry 4.0.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 364–382

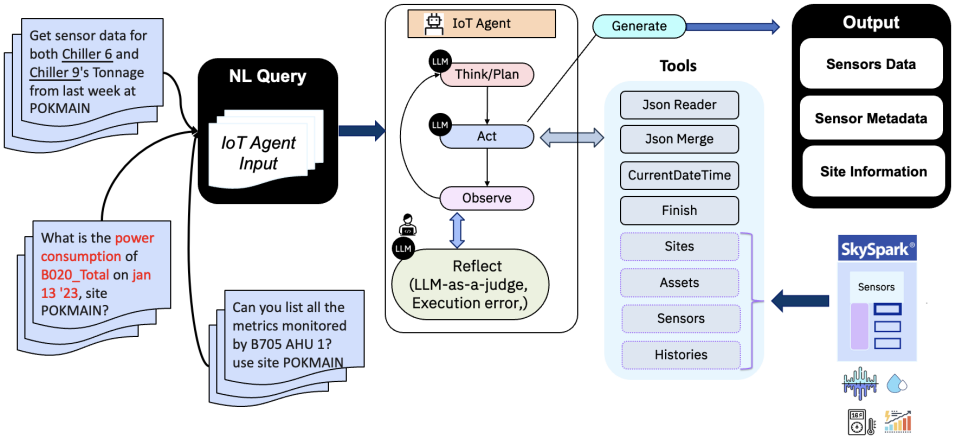

Figure 1: IoT Agent architecture and workflow for processing natural language queries in industrial systems.

While the standard ReAct agent (Yao et al., 2023) is effective for web tasks, it proved insufficient for industrial settings, often exhibiting incomplete reasoning, redundant tool calls, and failures in multi-step composition. To overcome these, we enhance the agent with an iterative `ReAct`, `Review`, and `Reflect` loop combined with the `Self-Ask` strategy (Press et al., 2023), enabling deeper introspective reasoning.

We evaluate our approach on two data center assets, a Chiller and an AHU, using 20 curated queries, and introduce **IoTBench**, a benchmark with 400+ natural language tasks across five industrial sites, incorporating both real and synthetic SCADA-style data (Xu et al., 2023) for diverse input formats.

We also explore **smaller LLMs** (∼8B parameters) for agentic workflows, leveraging new models with 128k-token contexts to transition from few-shot to many-shot in-context learning. Recent studies (Agarwal et al., 2024; Belcak et al., 2025; Dherin et al., 2025) show large contexts can outperform fine-tuning on complex tasks, enabling scalable, low-overhead deployment.

To exploit this, we introduce **Tiny Trajectory Stores**, a lightweight method for curating in-context examples. Combined with our prompt-based strategy, **ReActXen**, we show that long-context prompting significantly boosts reasoning without fine-tuning, enabling LLM agents to handle complex, domain-specific queries and advancing trustworthy, language-driven industrial automation.

2 IoT Agent: System Architecture

Figure 1 illustrates the architecture and workflow of the IoT Agent, which follows a ReAct-style framework to process natural language (NL) queries and generate actionable outputs for industrial monitoring and data retrieval.

The workflow begins with a user-issued natural language query $\mathcal{Q}$, which the IoT Agent $\mathcal{A}$ processes using a structured system prompt $\mathcal{P}$ and a set of available tools $\mathcal{L}$. This results in a multi-step reasoning and action trajectory T, and a final response o_{final}. Formally:

$$T, o_{\text{final}} = \mathcal{A}(\mathcal{Q}, \mathcal{P}, \mathcal{L}),$$

where o_{final} is the final output returned to the user (See Appendix A.1). A distinguishing feature of our system is its **real-time integration** with SkySpark (Skyspark, 2024), an industrial analytics platform that archives up to ten years of sensor and site data across multiple facilities. The IoT Agent uses tools backed by SkySpark to query both real-time and historical data for operational metrics such as `energy usage`, `tonnage`, `temperature differentials`, and `sensor health`.

To facilitate reliable interaction with these platforms, we implement seven custom tools (excluding the "Finish" tool), detailed in Table 1. These include four specialized tools for external data access (e.g., "Histories") and three internal utilities. All tools are implemented using Langchain's tool abstraction layer (Langchain, 2024a). The "Histories" tool presents unique challenges, such as interpreting temporal and string arguments robustly.

As shown in the top-right corner of Figure 1, tool outputs fall into three primary categories: **Sensor Data** (e.g., temperature, power readings), **Sensor Metadata** (e.g., units, source), and **Site Information** (e.g., zone labels, equipment relationships). It

Tool	Description	Parameters (All strings, dates in ISO 8601)
Json Reader*	Reads a JSON file and returns it as a single-line JSON string	`file_name`
Json Merge*	Merges two JSON arrays of the same type into a single-line JSON	`file_name_1`, `file_name_2`
CurrentDateTime*	Returns the current time as an ISO 8601 string and an equivalent English text date and time	None
Finish*	Finishes the agent execution and specifies a return string	Return string
Sites†	Returns a list of sites from SkySpark	None
Assets†	Returns a list of Assets at a Site	`site_name`
Sensors†	Returns a list of sensors for an Asset at a Site	`site_name`, `asset_name`
Histories†	Returns a list of sensor values for the specified Asset(s) at the specified Site	`site_name`, `asset_name_list`, `start (date)`, `final (date)`, `sensor_name`

Table 1: IoT Agent tools, their functionalities, and required parameters. * Supporting tool. † domain-specific tool.

is critical to distinguish between the output of a tool and that of the agent: tools typically return files or structured data, whereas the agent interprets this output to produce concise, human-readable summaries (e.g., data point counts, asset summaries, or file pointers).

3 ReActXen: Design and Implementation

Hallucination remains a critical bottleneck in agentic workflows, particularly when smaller LLMs operate under a ReAct-only paradigm. Our evaluations highlight consistent failures in date-offset reasoning (e.g., misinterpreting "last week"), hallucinated tools or parameters, and premature task termination. Beyond these, agents often fail to decompose multi-step problems, explore alternate toolchains, or match exact strings (e.g., sensor names). Also, models exhibit gaps in commonsense reasoning, such as linking chiller tonnage with energy efficiency, a vital inference in industrial IoT settings. These findings point to the need for more structured reasoning scaffolds and robust reflection phases to mitigate LLM brittleness. To address these, we propose *ReActXen*, a framework built on an "Agent-family" design as shown in Figure 2.

3.1 Agent Roles and Interactions

Building upon the terminology defined in Section A.1, we introduce a suite of specialized agents within the **Agent-Family** ($\mathcal{A}_{\text{family}}$), each contributing to robust and interpretable query resolution.

ReAct with Self-Ask. The *ReAct Agent* ($\mathcal{A}_{\text{react}}$) serves as the core executor, generating a trajectory T in response to a query $\mathcal{Q}$. To enhance reasoning fidelity, especially for structured tasks

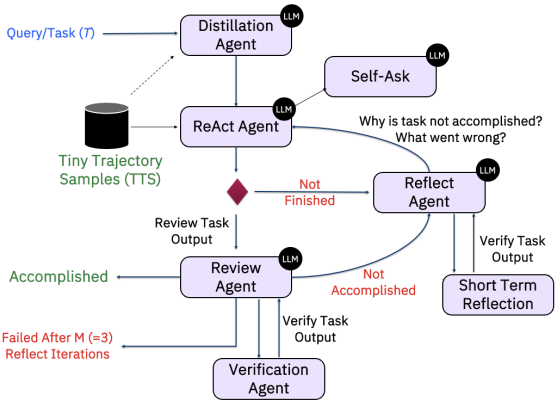

Figure 2: Proposed Architecture : ReActXen

such as mathematical problem-solving or entity disambiguation,$\mathcal{A}_{\text{react}}$ is equipped with a **self-ask** mechanism. Before executing any action $a_i \in T$, the agent emits an internal sub-question r_i to guide subsequent reasoning. This anticipatory query is also answered by LLM and improves trajectory coherence and tool invocation precision. Appendix 12 provide full system prompt.

Review Agent. The *Review Agent* ($\mathcal{A}_{\text{review}}$) functions as a lightweight verifier (See Appendix 13 for System Prompt). Given the triplet $(\mathcal{Q}, T, o_{\text{final}})$, it classifies the outcome as *Accomplished*, *Partially Accomplished*, or *Not Accomplished*, based on whether the final reasoning step r_n sufficiently addresses $\mathcal{Q}$. Using the LLM $\mathcal{M}$, it analyzes the full trajectory and, when necessary, generates feedback incorporated back into the system prompt $\mathcal{P}$ for the next iteration.

Reflect Agent. The *Reflect Agent* ($\mathcal{A}_{\text{reflect}}$) is triggered when $\mathcal{A}_{\text{react}}$ is unable to finish task failures occur or when prompted by $\mathcal{A}_{\text{review}}$. It performs

Algorithm 1 Reinforcement via Multiple Verbal Feedback

Initialize: ReAct agent $\mathcal{A}_{\text{react}}$, Review agent $\mathcal{A}_{\text{review}}$, Reflect agent $\mathcal{A}_{\text{reflect}}$, Distillation agent $\mathcal{A}_{\text{distill}}$, TTS $\mathcal{T}$
Receive query $\mathcal{Q}$
$\mathcal{Q}' \leftarrow \mathcal{A}_{\text{distill}}(\mathcal{Q})$ ▷ Optional distillation
Set memory $\texttt{mem} \leftarrow [\mathcal{T}]$; $t \leftarrow 0$
while $t < T_{\max}$ **do**
  $(\texttt{ans}, T) \leftarrow \mathcal{A}_{\text{react}}(\mathcal{Q}', \texttt{mem})$
  if ans exists **then**
    $\texttt{review}_t \leftarrow \mathcal{A}_{\text{review}}(\langle \mathcal{Q}', \texttt{ans}, T \rangle)$
    if $\texttt{review}_t =$ Accomplished or Error **then**
      break
    end if
  end if
  $\texttt{reflect}_t \leftarrow \mathcal{A}_{\text{reflect}}(\mathcal{Q}', T)$
  Update mem with feedback from $\texttt{reflect}_t$ and $\texttt{review}_t$
  $t \leftarrow t + 1$
end while
Return: Final solution ans

post-hoc introspection over the trajectory T, assessing reasoning steps and tool usage. It outputs targeted feedback, often in the form of strategy shifts or reasoning templates, which are added to the prompt $\mathcal{P}$ to guide future executions. The complete system prompt is given in Appendix 15.

Distillation Agent. The *Distillation Agent* ($\mathcal{A}_{\text{distill}}$) acts as a pre-processor, decomposing complex queries $\mathcal{Q}$ into structured semantic units: variables, constraints, and goals. It generates a distilled prompt $\mathcal{P}' \subseteq \mathcal{P}$, which serves as a scaffold for downstream agents, improving both interpretability and overall problem-solving success (See Appendix 14 for system prompt).

Algorithm 1 formalizes this iterative loop driven by verbal feedback, continuing until task success or a retry threshold is reached.

3.2 Tiny Trajectory Stores (TTS)

The Tiny Trajectory Store is a compact repository of short, efficient trajectories that represent key problem-solving steps. Formally, the TTS is defined as:

$$\mathcal{T} = \{S_1, S_2, \ldots, S_n\}$$

where each trajectory S_i is a sequence of steps $\{s_1, s_2, \ldots, s_k\}$, with $k \leq 3$. These trajectories capture essential actions and reasoning, enabling the agent to quickly adapt, reduce redundancy, and improve decision-making in subsequent tasks.

TTS enhances many-shot in-context learning by organizing examples into three pedagogical categories:

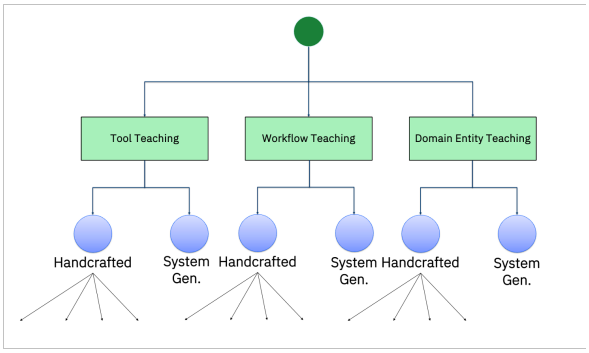

Figure 3: Tiny Trajectory Samples Organization

- **Tool/Skill Teaching:** Focuses on proper tool invocation, including specifying tool names, parameters, and expected outputs. This ensures accurate tool application in problem-solving.
- **Domain/Entity Teaching:** Emphasizes understanding and description of domain-specific entities (e.g., HVAC systems or sensors in industrial IoT). This helps the LLM recognize entities in user queries.
- **Workflow Teaching:** Guides the agent in executing tasks with structured, step-by-step processes to ensure efficient problem-solving.

Each TTS combines handcrafted ($\mathcal{H}$) and system-generated ($\mathcal{G}$) examples, balancing precision with adaptability. Currently, we have prepared 17 handcrafted trajectories: 12 general-purpose (e.g., merging JSON files) and 5 domain-specific (e.g., describing assets or sensors). These examples foster in-context learning and enhance the agent's ability to generalize across tasks. We have included two examples in Appendix 7 and 8. Figure 3 shows the overall organization of TTS.

4 IoTBench: Benchmark Dataset

In this section, we introduce **IoTBench**, a benchmark dataset $\mathcal{D}$ specifically constructed for evaluating agent performance in industrial IoT scenarios. Due to the absence of publicly available datasets tailored to this domain, we adopted a multi-phase strategy that combines hand-crafted and machine-generated data to ensure both quality and diversity. Figure 4 provides an outline of the data generation workflow.

Phase 1: Hand-Crafted Seed Set. We began by constructing 20 high-quality, manually written

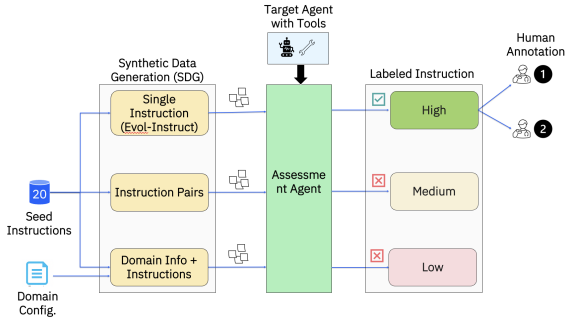

Figure 4: Synthetic Data Generation Pipeline

instructions for the IoT site `POKMAIN`, inspired by TaskBench (Shen et al., 2024). These examples span a range of reasoning challenges, including entity discovery (e.g., sites, assets), temporal queries with relative references (e.g., "last week", "midnight"), multi-asset and multi-sensor comparisons, interval-based versus point-in-time queries, and sensor disambiguation tasks (e.g., comparing readings from Chiller 6 and Chiller 9).

This dataset is denoted $\mathcal{D}_{\text{POKMAIN}}$. To test generalization, we generated $\mathcal{D}_{\text{RCHMAIN}}$ by adapting the original queries to a structurally similar site, `RCHMAIN`, with systematic substitutions of assets (e.g., swapping "AHU" with "Chiller").

Phase 2: Synthetic Instruction Generation. To scale coverage, we used the Evol-Instruct framework (Xu et al., 2024) to generate approximately 300 synthetic instructions based on both single prompts and prompt pairs. These were assessed by an **Assessment Agent**, which labeled each instruction with a confidence level (*High*, *Medium*, *Low*). After manual review, we retained 164 high-confidence instructions, comprising the dataset $\mathcal{D}_{\text{Syn-I}}$. Appendinx 16 provide detailed system prompt. The prompt used to produce new synthetic instruction is given in Appendix 17.

Phase 3: Domain-Specific Augmentation. Next, we enriched $\mathcal{D}_{\text{Syn-I}}$ by embedding domain-specific information such as site metadata, sensor types, and asset hierarchies. This yielded an additional 290 context-rich instructions across three distinct sites: `HQ3SBY`, `HQ1ARM3`, and `AUSWEST`. The final dataset from this phase, $\mathcal{D}_{\text{Syn-II}}$, emphasizes complex, multi-entity scenarios and longer instruction structures.

As evidenced in Table 2, both instruction length and complexity increase substantially from $\mathcal{D}_{\text{POKMAIN}}$ to $\mathcal{D}_{\text{Syn-II}}$, facilitating rigorous evaluation across a spectrum of IoT task difficulties.

Dataset	Size	Site(s)	Avg. Length (Std.)
$\mathcal{D}_{\text{POKMAIN}}$	20	1	12.90 (5.43)
$\mathcal{D}_{\text{RCHMAIN}}$	20	1	12.05 (4.98)
$\mathcal{D}_{\text{Syn-I}}$	164	1	20.83 (6.52)
$\mathcal{D}_{\text{Syn-II}}$	290	3	41.36 (10.17)

Table 2: Summary of IoTBench Datasets

5 Experimental Setup

5.1 Models and Agents

We evaluate diverse language models, including closed-source (`o1` (OpenAI, 2024)) and open-source families like `mistral` (Jiang et al., 2023), `llama` (Touvron et al., 2023), and `granite` (Granite Team, 2024)), selected based on leaderboard performance (Contributors, 2023). Experiments use handcrafted ($\mathcal{D}_{\text{POKMAIN}}$, $\mathcal{D}_{\text{RCHMAIN}}$) and synthetic datasets ($\mathcal{D}_{\text{Syn-I}}$, $\mathcal{D}_{\text{Syn-II}}$), covering varied task complexities. Unless specified, `mistral-large` and `granite-3-8b` are default backends.

We compare planning strategies including CoT (Wei et al., 2023), HuggingGPT (Shen et al., 2023), RAFA (Liu et al., 2024), ReAct-Review (Yao et al., 2023), and ReActXen, covering from plan-execute to reflective agents. For ReActXen, we set `max_react_step=15`, `max_reflect_step=15`, and `temperature=0` (1 for `o1`), running all experiments on a MacBook Pro with GPU backends. We report task completion rate (queries marked `Accomplished` by the Review Agent), efficiency (total token usage, average trajectory length, execution time), and computational cost (API calls per task).

5.2 Results on Baseline Dataset

Figure 5 shows task completion on $\mathcal{D}_{\text{POKMAIN}}$. Models like `o1` and `granite-3-8b` solve up to 14 tasks without reflection, with ReActXen enabling most models to achieve near-complete performance. **Five out of seven** models improve with reflection, highlighting its value. Tasks are marked `Not Accomplished` when exceeding step limits or when outputs are insufficient per Review Agent judgment. Results on $\mathcal{D}_{\text{RCHMAIN}}$ are included in Appendix D.1.

5.3 Reflection Rounds

Figure 6 shows performance as reflection steps increase. Most models improve within 5 steps, e.g., `mixtral-8x7b` rapidly reaches near-optimal performance. Models like `llama-3-8b` show limited

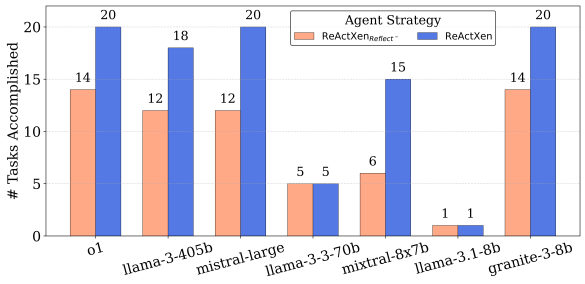

Figure 5: Tasks accomplished on baseline dataset.

gains, possibly due to suboptimal Review Agent outputs (Appendix 10).

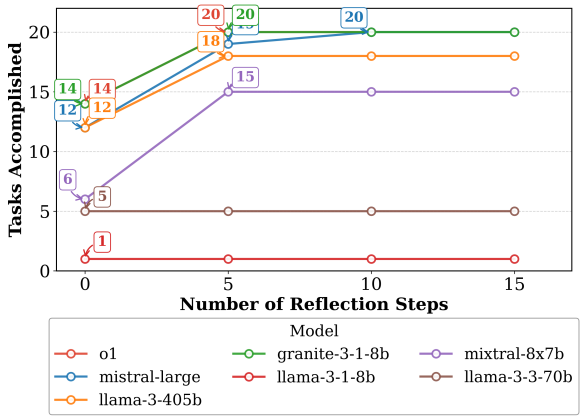

Figure 6: Effect of reflection rounds on task completion.

5.4 Traj. Length and In-context Examples

Figure 7 shows average trajectory lengths; early success (fewer steps) indicates higher efficiency. Without in-context examples, models perform worse; reflective reasoning mitigates this gap (Figures 8, 9), confirming the benefit of Tiny Trajectory Samples (TTS).

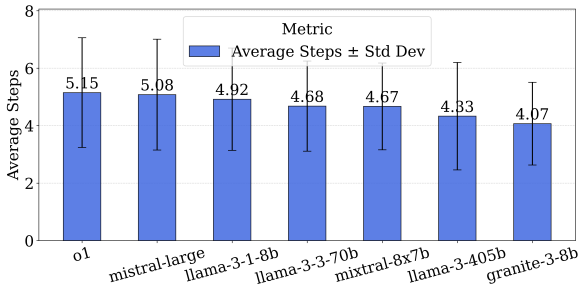

Figure 7: Average trajectory length.

5.5 Execution Strategies

Figure 10 compares strategies. ReActXen achieves the highest task completion, followed by RAFA and ReActXen-Reflect, highlighting the benefit of

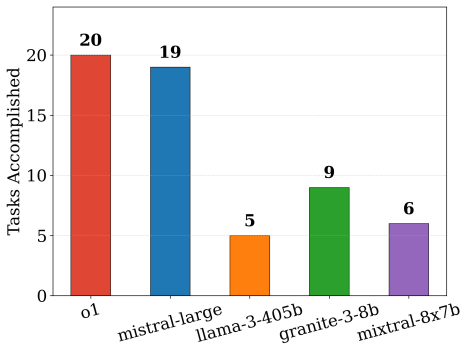

Figure 8: Task completion comparison when no example is included.

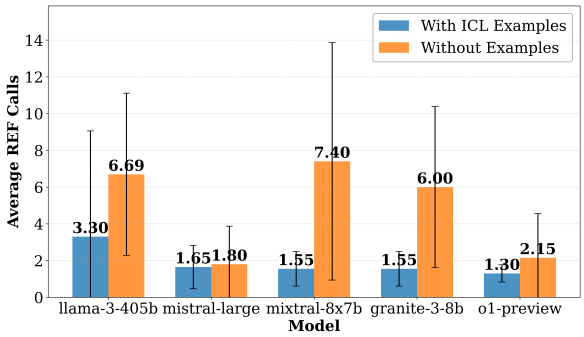

Figure 9: TTS Impact on average reflection rounds

iterative reasoning. CoT and HuggingGPT complete fewer tasks, underscoring limitations in static planning.

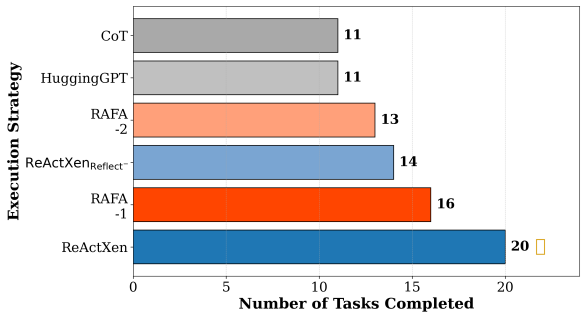

Figure 10: Task completion across agent strategies.

5.6 Synthetic Benchmark

Table 3 summarizes large-scale experiments on $\mathcal{D}_{\text{Syn-I}}$. Reflection significantly boosts task completion (e.g., `mistral-large`: 101→161, `granite-3-8b-instruct`: 88→149), though with increased computational cost. Results on $\mathcal{D}_{\text{Syn-II}}$ are in Appendix D.2.

Model	Tasks Completed		Tokens Sent	Tokens Recv.	API Calls	Proc. Time	Traj. Len
	@Round 1	@Final	Avg	Avg	Avg	Avg	Avg
Experiment 1: Internal Reviewer							
mistral-large	101	161	337k	1854	32	3.2	7.2
granite-3-8b-instruct	88	149	601k	2445	49	5.9	5.8
Experiment 2: External Reviewer							
mistral-large	121	155	270k	1395	25	2.7	6.7
granite-3-8b-instruct	90	140	642k	2721	55	6.6	5.7

Table 3: Summary metrics on $\mathcal{D}_{\text{Syn-I}}$.

6 Research Innovation in TTS

Tiny Trajectory Store (TTS) provides a compact representation of reasoning trajectories, which are essential for supplying high-quality examples to small language models (SLMs). A key research innovation is the automatic generation of compact trajectories from real execution traces exceeding a length of five (Figure 5). Since real traces often have an average length greater than five, this motivates an automated, scalable method to efficiently curate examples. We conducted an experiment on the $\mathcal{D}_{\text{POKMAIN}}$ problem set to quantify this approach. The experiment iteratively expands TTS by automatically generating solved examples, allowing SLMs to leverage TTS for improved problem-solving performance while reducing their reliance on handcrafted examples.

Algorithm 2 Iterative Auto-Generation of TTS

1: **Initialize:** Empty Tiny Trajectory Store $\mathcal{T}$, Problem set $P = \mathcal{D}_{\text{POKMAIN}}$ (size 20)
2: Set unsolved_problems $\leftarrow$ P, max_reflect_step $\leftarrow$ 1, max_examples $\leftarrow$ 17
3: **for** round = 1 to 5 **do**
4: Use current $\mathcal{T}$ as in-context examples (ICL)
5: Solve unsolved_problems using ReAct + Review
6: Record num_solved
7: Add solved problems to $\mathcal{T}$ as new examples via LLM auto-generation
8: Update unsolved_problems $\leftarrow$ unsolved_problems - solved_problems
9: **if** size($\mathcal{T}$) $\geq$ max_examples **or** num_solved = 0 **then**
10: **break**
11: **end if**
12: **end for**
13: **Return:** Solved problems per round, final $\mathcal{T}$

The automation process worked iteratively: problems solved in one round were automatically converted into TTS examples and reused as in-context demonstrations for the next round. Performance began low with zero in-context examples (5 problems solved in Round 1). Over successive rounds, the model reused its own generated examples, with mixed effectiveness, as shown in Table 4.

Round	# of Examples in TTS	# of Problems Newly Solved by granite-3-8b
1	0	5
2	5	4
3	9	2
4	11	3
5	14	0

Table 4: Iterative performance of granite-3-8b.

6.1 Effect of Curated Trajectories

To isolate the effect of curated trajectories based on quantity and quality, we varied the number of hand-crafted examples. Figure 3 shows results with all 17 curated trajectories, while Figure 6 shows the 0-example baseline, where performance drops sharply. Table 5 summarizes the effect of quantity while keeping quality fixed. For all models, performance improves consistently as the number of curated examples increases. The in-context examples in this experiment were hand-curated to ensure high quality.

Model	0	5	12	17
llama-3-405b	5	9	15	18
granite-3-8b	9	11	17	20
mixtral-8x7b	6	8	11	15

Table 5: Performance of different models as the number of curated hand-crafted trajectories increases. Higher number of curated examples improves performance.

7 Conclusion

Our experiments reveal key challenges and gains in reasoning, tool use, monitoring, scanning, and early termination. Integrating Self-Ask and many-shot training with Tiny Trajectory Store (TTS) improved reasoning and tool generalization. Distillation, Reflection, and Review Agents boosted accuracy, efficiency, and reduced redundant executions. Future work will explore model fine-tuning for scaling to 100s of IoT sites, addressing variations in sensor descriptions and asset diversity.

Limitations

A major difficulty with ReActXen is the verification of the result answer. The answer obviously varies with the question and the LLM used, but the validation is complicated:

- Validation of a JSON result – it is relatively easy to validate that the JSON structure is correct
- The asset instance and sensor should also be validated
- The date range should also be validated (if applicable)
- The sensor values should also be validated (if applicable)

We have had some limited success validating these items for a restricted subset of assets. It is not clear how this can be extended to a large study. Perhaps the larger community can help with this task.

References

Rishabh Agarwal, Avi Singh, Lei M. Zhang, Bernd Bohnet, Luis Rosias, Stephanie Chan, Biao Zhang, Ankesh Anand, Zaheer Abbas, Azade Nova, John D. Co-Reyes, Eric Chu, Feryal Behbahani, Aleksandra Faust, and Hugo Larochelle. 2024. Many-shot in-context learning. *Preprint*, arXiv:2404.11018.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small language models are the future of agentic ai. *Preprint*, arXiv:2506.02153.

OpenCompass Contributors. 2023. Opencompass: A universal evaluation platform for foundation models. https://github.com/open-compass/opencompass.

Benoit Dherin, Michael Munn, Hanna Mazzawi, Michael Wunder, and Javier Gonzalvo. 2025. Learning without training: The implicit dynamics of in-context learning. *Preprint*, arXiv:2507.16003.

IBM Granite Team. 2024. Granite 3.0 language models.

Jason Jabbour and Vijay Janapa Reddi. 2024. Generative ai agents in autonomous machines: A safety perspective. *arXiv preprint arXiv:2410.15489*.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues?, 2024. *URL https://arxiv.org/abs/2310.06770*.

Langchain. 2024a. Langchain tools documentation. Accessed: 2025-02-09.

Langchain. 2024b. Wikipedia tool integration. Accessed: 2025-02-09.

Zhihan Liu, Hao Hu, Shenao Zhang, Hongyi Guo, Shuqi Ke, Boyi Liu, and Zhaoran Wang. 2024. Reason for future, act for now: A principled architecture for autonomous LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 31186–31261. PMLR.

Alexander Nikitin and Samuel Kaski. 2022. Human-in-the-loop large-scale predictive maintenance of workstations. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3682–3690.

OpenAI. 2024. o1-preview.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A. Smith, and Mike Lewis. 2023. Measuring and narrowing the compositionality gap in language models. *Preprint*, arXiv:2210.03350.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Preprint*, arXiv:2303.17580.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. Taskbench: Benchmarking large language models for task automation. *Preprint*, arXiv:2311.18760.

Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. 2024. Building ai agents for autonomous clouds: Challenges and design principles. *Preprint*, arXiv:2407.12165.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Skyspark. 2024. Skyspark. https://www.skyfoundry.com/product. Accessed: 2024-09-02.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *arXiv preprint arXiv:2403.17927*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *Preprint*, arXiv:2304.12244.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Hong Yang, Aidan LaBella, and Travis Desell. 2022. Predictive maintenance for general aviation using convolutional transformers. *Preprint*, arXiv:2110.03757.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

A Agent Terminology

A.1 Terminology Definitions

We use following key terms throughout this paper:

- **Query** ($\mathcal{Q}$): A natural language instruction submitted by a user—typically a plant operator—to retrieve or analyze sensor data.
- **Agent** ($\mathcal{A}$): A reasoning and acting module responsible for solving a given query $\mathcal{Q}$. Agents may be specialized for distinct stages in the overall task workflow.
- **System Prompt** ($\mathcal{P}$): A structured input to the LLM, comprising instructions, domain-specific exemplars, and constraints that shape agent behavior. Formally, $\mathcal{P} = \{\text{instructions}, \text{examples}, \ldots\}$.
- **Trajectory** (T): The sequence of reasoning and action steps taken by an agent to solve a query. A trajectory is denoted as:

$$T = \{r_1, a_1, o_1, \ldots, r_{n-1}, a_{n-1}, r_n\},$$

  where r_i is a reasoning step, a_i an action (e.g., tool invocation), o_i the result of the action, and r_n contains the final answer.
- **In-context Learning Example** ($\mathcal{E}_i$): A worked-out demonstration included within the prompt to guide agent reasoning for task T_i.
- **LLM** ($\mathcal{M}$): The large language model serving as the agent's reasoning core. The model maps input x to output y as $\mathcal{M} : x \mapsto y$.
- **Tools** ($\mathcal{L}$): External utilities or APIs the agent can invoke to access data, execute domain-specific operations, or assist in solving subtasks.

A.2 Agent Output

Figure 11 illustrates three representative examples of agent input-output behavior across varying query types. Additionally, the agent performs further analysis on this data, such as identifying maximum values and detecting patterns.

A.3 Why ReActXen Improves performance

Theorem 1 (R.1: Early Feedback Stabilizes Trajectories)**.** *Injecting Review and Reflect feedback early in the reasoning process establishes a stable and influential feedback regime. This significantly enhances the guidance provided to the model and improves final task completion accuracy.*

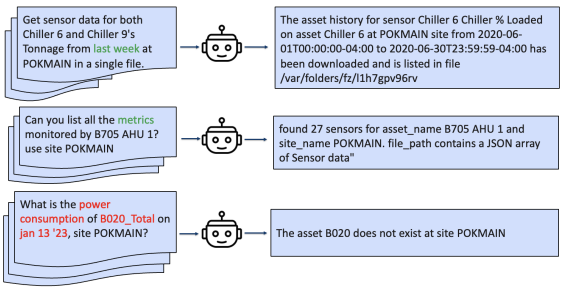

Figure 11: IoT Agent : Input-Output Configuration

Theorem R.1: Injecting Review and Reflect feedback early in the process establishes a stable and influential feedback loop, significantly enhancing the guidance provided to the model.

Let $\mathcal{M}$ denote an auto-regressive language model that generates tokens sequentially, where each token is produced based on previous tokens via an attention mechanism operating over a Key-Value (KV) cache.

Let F_1 and F_2 represent two reflexive feedback regions inserted at the beginning and end of the prompt, respectively. We assume that:

- Tokens in F_1 are accessible to all downstream tokens due to the model's attention pattern, where each token in F_1 influences subsequent tokens for $i \geq t$.
- Tokens in F_2, however, are injected too late in the process, affecting only a narrow subset of trailing tokens, and hence have limited influence on the overall generation.

Thus, the influence of early reflection (F_1) is strictly greater than that of late reflection (F_2):

$$\text{Influence}(F_1) > \text{Influence}(F_2)$$

Proof: This inequality arises from three key factors:

- **Sequential Generation:** In an auto-regressive model, only past tokens influence future generations, thereby amplifying the role of tokens early in the process.
- **KV Cache Persistence:** Early tokens persist in the KV cache and are attended to repeatedly, reinforcing their impact on subsequent tokens.
- **Positional Advantage:** Early tokens serve as foundational anchors, shaping the trajectory of the generation process and guiding later tokens more effectively.

Benchmark	TaskBench (NeurIPS 2024)	ITBench (ICML 2025)	IoTBench (Ours)
Data Generation	Tool Graph + Back-Instruct	Manual	Manual + LLM
Tool Dependency			
Quality Control	LLM Self-critique + Rule-based	Human Verification	Human Verification
Evaluation	Task Decomposition + Tool Selection + Parameter Prediction	ReAct Planning + Tool Selection	ReAct Planning + Tool Selection + Parameter Prediction
Tool Complexity	Single tool to complex tool graph	-	Multiple tools; same tools can be called multiple times
Dataset Scale	17,331 samples	141 scenarios	494 samples
Temporal / Dynamic Query	✗	✗	✓
Name Disambiguation	✗	✗	✓
Tools Output Operation	✗	✗	✓

Table 6: Comparative overview of IoTBench against existing benchmarks. IoTBench combines rigorous human curation, systematic generation, and diverse evaluation, making it a high-quality resource for LLM benchmarking.

Corollary: The early injection of reflective feedback creates a more stable and impactful model behavior, compared to feedback introduced later in the generation process.

B Comparative Overview with Existing Benchmarks

To contextualize the significance of our dataset, we provide a comparative analysis of **IoTBench** against two existing benchmarks: **TaskBench** (NeurIPS 2024) and **ITBench** (ICML 2025). Table 6 summarizes the key differences in data generation, tool usage, evaluation methodology, and other dataset characteristics.

This comparison highlights that IoTBench not only maintains tool dependency and human verification like prior benchmarks, but also incorporates temporal/dynamic queries, name disambiguation, and tool output operations, offering a more comprehensive evaluation framework.

C ReAct Style Dialog in TTS

We provided two examples stored in TTS for domain understabnding, see Table 7 and 8. In Table 8, we queried the system for the list of assets associated with the site "POKMAIN" to understand their representation. After retrieving the asset details in JSON format, we identified five example assets, each containing a site_name and asset_name pair, such as "B706_LC26" and "B012 SCWP 1". The action flow included retrieving the data, reading the JSON file, and presenting these asset examples as the final output.

D Experiment on Remaining Dataset

D.1 $\mathcal{D}_{\text{RCHMAIN}}$

We move from one asset to another. The results (See Table 11) from our experiments demonstrate the significant impact of reflective reasoning and in-context learning on task completion across various language models. Models incorporating the reflection process, such as *ReActXen*, consistently outperformed those without it, achieving higher task completion rates. For instance, *Mistral-large* and *Granite-3-8b-instruct* saw improvements in task completion when reflective reasoning was introduced. However, while these improvements were evident, they came at the cost of increased computational time, underlining the trade-off between performance and efficiency. The results also suggest that the number of reflection steps plays a crucial role, with diminishing returns observed after a certain point. This emphasizes the need for careful optimization of reflection mechanisms to balance task completion and resource utilization.

D.2 Larger Dataset - $\mathcal{D}_{Syn-\text{II}}$

The Table 9 presents a comparison of key metrics for two models: mistral-large and granite-3-8b-instruct. We managed to run only 32 task in 24 hour. mistral-large completed 27 tasks, with average token metrics slightly lower than granite-3-8b-instruct, which completed 24 tasks. The mistral-large model sent an average of 1,048,972 tokens (with a standard deviation of 1,386,043.31), compared to granite-3-8b-instruct's 1,387,515.79 tokens (standard deviation of 1,852,631.64). Both models show significant differences in API and reflection call counts, with granite-3-8b-instruct requiring more reflection calls. The mistral-large

Thought 1	I need to get the list of available sites to answer the question.
Action 1	`sites`
Action Input 1	{}
Observation 1	Available sites: found the following 34 sites, listed in file 4ce1a0e4-....
Thought 2	I now have the list of available sites, which is stored in a file.
Action 2	`Finish`
Action Input 2	The sites are listed in file 4ce1a0e4....

Table 7: ReAct Style Dialog with Thought, Action, and Observation Loop Stored in TTS

Thought 1	I need to get the list of assets for a specific site to understand how they are represented.
Action 1	`assets`
Action Input 1	`site_name=POKMAIN`
Observation 1	`"site_name": "POKMAIN", "total_assets": 5, "file_path": "360fea44-....json",` "message": "found 5 assets for site_name POKMAIN. file_path contains a ..."
Thought 2	I now have the list of assets for site POKMAIN, which is stored in a file.
Action 2	`jsonreader`
Action Input 2	`file_name=360fea44-6bb2-4f41-89d7-9a8313714589.json`
Observation 2	`["site_name": "POKMAIN", "asset_name": "B706_LC26"]...`
Thought 3	I now have the list of assets for site POKMAIN. I will provide ....
Action 3	`Finish`
Action Input 3	Assets are represented with a `site_name` and an `asset_name`. Here are five examples: 1. `"site_name": "POKMAIN", "asset_name": "B706_LC26"` 2. `"site_name": "POKMAIN", "asset_name": "B012 SCWP 1"` 3. `"site_name": "POKMAIN", "asset_name": "B710 FCU 1-39"` 4. `"site_name": "POKMAIN", "asset_name": "B006_LC74"` 5. `"site_name": "POKMAIN", "asset_name": "B006A AHU 2"`

Table 8: ReAct Style Dialog for Asset Representation Query also stored in TTS

model has a lower average processing time of 10.90 minutes compared to 17.18 minutes for granite-3-8b-instruct.

Metric	mistral-large	granite-3-8b-instruct
ModelID	6	8
Total Tasks Completed	27	24
Avg Tokens Sent	1048972.31	1387515.79
Std. Dev.	1386043.31	1852631.64
Avg Tokens Received	6532.38	7326.79
Std. Dev.	8486.77	11303.45
Avg API Calls	109.53	133.94
Std. Dev.	137.66	130.54
Avg REF Calls	4.75	8.94
Std. Dev.	5.12	7.84
Avg Processing Time (min)	10.90	17.18
Std. Dev.	13.52	21.99

Table 9: Summary of model metrics with mean and standard deviation on $\mathcal{D}_{\text{Syn-II}}$.

Model	No Reflect [min]	Reflect [min]
granite-3-2-8b-instruct	16.34	43.88
mistral-large	18.17	54.53
llama-3-1-8b-instruct	16.81	35.58
o1	14.00	470.50

Table 10: Comparison of execution times with and without reflection.

ModelName	Total Tasks Completed	Avg Tokens Sent	Avg Tokens Received
mixtral-8x7b-instruct-v01	14	771161.50 (1052780.44)	4276.80 (5390.68)
mistral-large	18	948762.30 (1827178.31)	4687.25 (8830.64)
llama-3-405b-instruct	16	539524.45 (1269279.89)	3478.65 (7668.36)
granite-3-8b-instruct	16	559777.75 (1035710.01)	2304.85 (4334.83)
llama-3-1-8b-instruct	1	443671.40 (1094015.96)	1635.15 (3881.86)
llama-3-3-70b-instruct	0	204592.15 (526319.34)	1461.65 (3566.10)

Table 11: Summary of model metrics with mean and standard deviation on $\mathcal{D}^{*}_{\text{RCHMAIN}}$.

System Prompt (ReActXen)
Answer the following questions as best you can. You have access to the following tools: {tool_desc} Use the following format: Question: the input question you must answer Thought: you should always think about what to do Action: the action to take, should be one of [{tool_names}] Action Input: the input to the action Observation: the result of the action ... (this Thought/Action/Action Input/Observation can repeat N times) Thought: I now know the final answer Final Answer: the final answer to the original input question ## Here are Guidance on Tool Usage: 1. Default to Self-Ask When Tools Are Missing: Fallback to Self-Ask when tools are unavailable, using logical problem-solving, knowledge or recent interaction (e.g., datetime calculations, math reasoning). 2. Prioritize Step-by-Step Explanations: Provide step-by-step explanations to ensure clarity and transparency in reasoning. 3. Fallback for Common Operations: Manually solve common operations like arithmetic, string manipulations, or date handling when necessary, and validate the solution. 4. Clearly Identify the Steps: Explicitly state when reasoning is used and solve problems step-by-step. { 5. Utilize Agent-Ask for Clarifications: When additional information is needed to resolve the question, use Agent-Ask to query another agent before taking action. Ensure the clarification request is specific and directly addresses missing details in the input question. {Here are some examples: {examples} (END OF EXAMPLES) Here is feedback: {reflections} Question: {question} {scratchpad}

Table 12: System prompt for proposed ReActXen Agent

You are a critical reviewer tasked with evaluating the effectiveness and accuracy of an AI agent's response to a given question or task. Your goal is to determine whether the agent has successfully accomplished the task or has made an unjustified claim of success (hallucinated).

Evaluation Criteria:
1. **Task Completion:**
- Verify if the agent executed the necessary actions (e.g., using tools, downloading files, generating results) to address the question or task.
- The response must produce a meaningful and relevant outcome within the context of the given question.
- Do not make an implicit assumption, such as show the file content when question or task has not asked explicitly.
- If the agent made an internal mistake (e.g., incorrect reasoning or error in intermediate steps) but successfully recovered and completed the task, it should still be considered **Accomplished** as long as the outcome is correct.

2. **Exception Handling:**
- If the agent claims it cannot complete the task due to the unavailability of remote services or resources, confirm whether this is a valid justification.

3. **Hallucination Check:**
- If the agent claims success without executing the required actions or without producing tangible outcomes, identify this as a hallucination.

4. **Clarity and Justification:**
- Ensure the response provides sufficient evidence or explanation to support its claims (success or failure).

Question: {question}
Agent's Thinking: {agent_think}
Agent's Final Response: {agent_response}

Output Format:
Your review must always be in JSON format. Do not include any additional formatting or Markdown in your response.
{
"status": "Accomplished | Partially Accomplished | Not Accomplished",
"reasoning": "A concise explanation for your evaluation.",
"suggestions": "Optional. Actions or improvements for rectifying the response if applicable."
}
(END OF RESPONSE)

Please provide your review based on the given criteria.

Table 13: Critical Reviewer System Prompt

As a highly professional and intelligent expert in information distillation, you excel at extracting essential information from user input query to solve problems. You adeptly transform this extracted information into a suitable format based on the respective type of the issue. If the problem can be generalized to a higher level to solve multiple issues, further analysis and explanation will be provided upon your next response.

Please categorize and extract the crucial information required to solve the problem from the user's input query. Combining these two elements will generate distilled information. Subsequently, deliver this distilled information, based on the problem type, to your downstream task. The distilled information should include:

1. Values and information of key variables extracted from user input, which will be handed over to the respective expert for task resolution, ensuring all essential information required to solve the problem is provided.
2. The objective of the problem and corresponding constraints.
3. ......

Query: {question}
Please distill the information following the format below and cease response after the output of the distilled information. Do not generate any new information which is not provided in the input query.

Distiller Respond:

Distilled Information:

1. Original Question:
- **Question**: Write the input query/question unaltered.
2. Key Information:
- **Variables**: List the key variables extracted from the query.
- **Values**: Any known values or default values (if provided by the user).
... 6. Python Transformation (Optional):
Input parameters:
variable1_name = x
variable2_name = y
......
variableN_name = z

Do not proceed beyond this structured output.
(END OF RESPONSE)

Avoid providing any solution or attempting to answer the problem directly. Your role is only to **extract, categorize, and structure** the information.

Table 14: System Prompt Template for Information Distillation

```
You are an advanced reasoning agent that can improve based on self reflection.
You will be given a previous reasoning trial in which you were given access to
several tools and a question to answer. You were unsuccessful in answering the
question either because you guessed the wrong answer with Finish[<answer>], or
you used up your set number of reasoning steps. In a few sentences, diagnose
a possible reason for failure and devise a new, concise, high-level plan that
aims to mitigate the same failure. Use complete sentences.

Here are some examples:
{examples}

Previous trial:
Question: {question}
{scratchpad}

Reflection:
```

Table 15: Reflection Agent System Prompt

```
You are an evaluator tasked with assessing the feasibility of a given agent solving a specific task based on its available tools, expertise, and past performance. Your objective is to determine whether the agent can successfully complete the task and assign a confidence level based on your analysis.

Evaluation Criteria:
1. Tool Availability: Verify the agent has the necessary tools.
2. Expertise and Experience: Assess relevant expertise or past successes.
3. Task Complexity: Evaluate task difficulty vs. agent capability.
4. Confidence Evaluation: Assign High (80–100%), Medium (50–79%), or Low (0–49%) confidence.
5. Justification: Explain the reasoning behind the assigned confidence.

Task Details:
- Input Task: {input_task}

Agent Details:
- Agent Name: {agent_name}
- Agent Tools: {agent_tools}
- Agent Expertise: {agent_expertise}
- Agent Past Task History: {agent_task_history}

Output Format:
Your response must be in JSON format with the following structure:
{
"confidence_level": "High | Medium | Low",
"confidence_percentage": {confidence_percentage},
"justification": "Concise explanation for confidence level.",
"recommendations": "Optional suggestions for improving agent success."
}
(END OF RESPONSE)
```

Table 16: Task Assessment System Prompt

You are tasked with generating a brand new, unique prompt inspired by two provided prompts. Your goal is to combine elements from both prompts, ensuring that the new prompt belongs to the same domain but is even more unique and rare. The new prompt should reflect the themes of both given prompts in a creative and original way.

Guidelines:
1. **Incorporate Elements from Both Prompts:** Synthesize aspects from both Given Prompt 1 and Given Prompt 2.
2. **Maintain Similar Length and Complexity:** Ensure that the new prompt has a similar length and level of complexity as the two given prompts.
3. **Ensure Coherence and Reasonability:** The new prompt must be logical, reasonable, and coherent.
4. **If a Date or Time is Provided:** Then generate a variation or representation of a new date close to the given date.
5. **If the Prompts Are Diverse:** Avoid mixing the two prompts. Instead, select one prompt for generating a new prompt.
6. **Avoid Direct Phrases:** Do not use the phrases Given Prompt 1, Given Prompt 2, or Created Prompt in the new prompt itself.

Prompt Inputs:
- Prompt 1: {prompt1}
- Prompt 2: {prompt2}

Output Format:
Your response must be in JSON format with the following structure:
{
"created_prompt": "here is your newly generated prompt"
}
(END OF RESPONSE)

Please provide your output based on the given prompt 1 and prompt 2 and the above guidelines.

Table 17: Base Instruction: Two Prompt