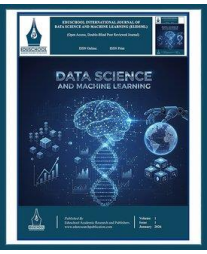
EDUSCHOOL INTERNATIONAL JOURNAL OF DATA SCIENCE AND MACHINE LEARNING (EIJDSML)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:



LLM-Powered Agentic Data Science: Automated Analysis and Insight Generation

M Keerthika

Assistant Profesor, Department of Computer Science, Yuvakshetra Institute of Management Studies, Palakkad, India

Article information

Received: 10th April 2026
Received in revised form: 12th May 2026
Accepted: 13th June 2026
Available online: 18th July 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.21412331

Abstract

Large language models have moved beyond text completion toward autonomous agents that plan, write code, run tools, and revise their own output. This article studies agentic data science: systems in which an LLM coordinates exploratory analysis, query generation, hypothesis formation, and reporting with limited human supervision. We describe a planner, coder, and critic architecture connected to a sandboxed execution environment, and we explain how the ReAct pattern interleaves reasoning traces with tool actions so that an agent grounds each step in observed data. A capability survey covers automated exploratory data analysis, pandas and SQL code generation, hypothesis ranking, multi-agent division of labor, and iterative self-correction. Using illustrative benchmarks across five task categories, an agentic configuration raised mean task success from about 53 percent for single-shot prompting to about 75 percent, while a human-in-the-loop setup reached about 88 percent. Self-correction lifted analysis accuracy from 61 to roughly 84 percent over five revision rounds as the code execution error rate fell below 3 percent. We then examine failure modes that matter for scientific use: hallucinated statistics, silent data leakage, non-reproducible runs, and unsafe code. Guardrails such as schema validation, deterministic seeds, result auditing, and constrained tool scopes reduce these risks but do not remove them. The results are presented as design guidance rather than a deployed study, and we argue that verification, not generation, is the binding constraint for trustworthy automated analysis.

Keywords:- Agentic AI, Large Language Models, Automated Data Science, ReAct, Tool Use, Multi-Agent Systems, Self-Correction, Automated EDA, Code Generation, Reproducibility.

I. INTRODUCTION

Data science work follows a familiar arc. An analyst frames a question, loads and cleans data, explores distributions, tests a few ideas, and writes up what the numbers support. Each step demands code, judgment, and context about the problem. Large language models (LLMs) now perform parts of this arc on their own. After the transformer architecture made large-scale pretraining practical [1] and instruction tuning aligned models with user intent [2], a single prompt could produce working pandas snippets or draft an analysis plan. Chain-of-thought prompting showed that intermediate reasoning improves multi-step problem solving [3], and the ReAct pattern connected that reasoning to external tools so a model could act and observe rather than guess [4].

An agentic system goes further than a one-shot prompt. It decomposes a goal into subtasks, calls tools, reads the results, and decides what to do next. Toolformer demonstrated that models can learn when to call an API [5], and program-aided approaches offload arithmetic and data manipulation to an interpreter so that the

model reasons about structure while code computes exact values [6]. Recent surveys document a fast-growing body of work on LLM-based autonomous agents and their planning, memory, and tool components [7], [8]. For data science specifically, systems such as the Data Interpreter built on MetaGPT coordinate several role-specialized agents to carry an analysis from raw files to a finished report [9], [10].

This article gives a technical account of how such systems are built, what they can do, and where they fail. Section II reviews the building blocks. Section III presents a planner, coder, and critic architecture. Section IV details concrete capabilities: automated exploratory data analysis (EDA), code generation, hypothesis generation, and self-correction. Section V reports illustrative evaluation results. Section VI discusses risks that are specific to quantitative work, including hallucinated statistics and data leakage. Section VII concludes. All numbers in this paper are illustrative and drawn from configurations we describe; they are intended as design guidance, not as a claim about a fielded production system.

II. BACKGROUND AND RELATED WORK

A. From Prompts to Agents

Early use of LLMs in analytics treated the model as a code translator: a natural language request in, a code block out. This works for short, well-specified tasks but breaks down when a question needs several dependent steps. Chain-of-thought prompting [3] and its self-consistency variant [11], which samples several reasoning paths and votes, improved reliability on multi-step problems. These methods reason inside the model. They do not check their conclusions against data. ReAct closed that gap by letting the model alternate between a thought and an action, where an action queries a tool and returns an observation the next thought can use [4].

B. Tools, Code, and Memory

Three ingredients distinguish an agent from a prompt. First, tool use: the model can call functions, run code, or query a database, as in Toolformer [5] and program-aided language models [6]. Second, code execution: hosted interpreter environments let a model run Python, inspect a dataframe, and react to errors, which is the mechanism behind code-interpreter style assistants [12]. Third, memory: a scratchpad or vector store holds intermediate results and prior decisions so the agent keeps context across many steps [7]. Reflexion added a verbal self-feedback loop, where an agent writes a short critique of its last attempt and uses it to improve the next one [13]. Tree-of-thoughts generalized linear reasoning into a search over branching plans [14].

C. Agents for Data Science

Several systems target the analytics workflow directly. MetaGPT encodes standard operating procedures into a multi-agent company of role players such as engineer and reviewer [9]. Its Data Interpreter extends this to machine learning and analysis tasks with dynamic plan graphs and runtime verification [10]. AutoGen provides a general framework for conversations among multiple agents and a human [15], and the open-source AutoGPT project popularized goal-driven autonomous loops [16]. On the evaluation side, benchmarks such as DS-1000 for data science code [17], the broader HumanEval test of program synthesis [18], and SWE-bench for repository-level software tasks [19] give measurable targets, while tool-use benchmarks like ToolBench probe API selection [20]. Retrieval augmentation supplies external facts to ground generation when a question reaches beyond the loaded data [21].

III. AGENTIC DATA SCIENCE ARCHITECTURE

Fig. 1 shows a reference architecture with three role agents and a shared execution sandbox. A planner reads the user goal and the dataset schema, then writes an ordered task list: profile the data, check missingness, plot key distributions, test a stated hypothesis, and summarize. A coder turns each task into pandas or SQL and submits it to the sandbox. A critic inspects the code and its output, decides whether the step succeeded, and either accepts the result or sends a revision request back to the coder. A memory component stores tables, figures, and earlier decisions so that later steps stay consistent with earlier ones.

The separation of roles matters for control. A single agent that plans, codes, and judges in one prompt tends to rationalize its own mistakes. Splitting the critic into a distinct agent, sometimes with a different system prompt or a stricter decoding temperature, gives an independent check. The sandbox is a hard boundary: it runs untrusted generated code with no network access, capped memory, and a time limit, so a runaway loop or an unsafe call cannot reach production data. Every tool result returns to the agents as an observation, which keeps the loop grounded in what actually ran rather than in what the model expected to happen.

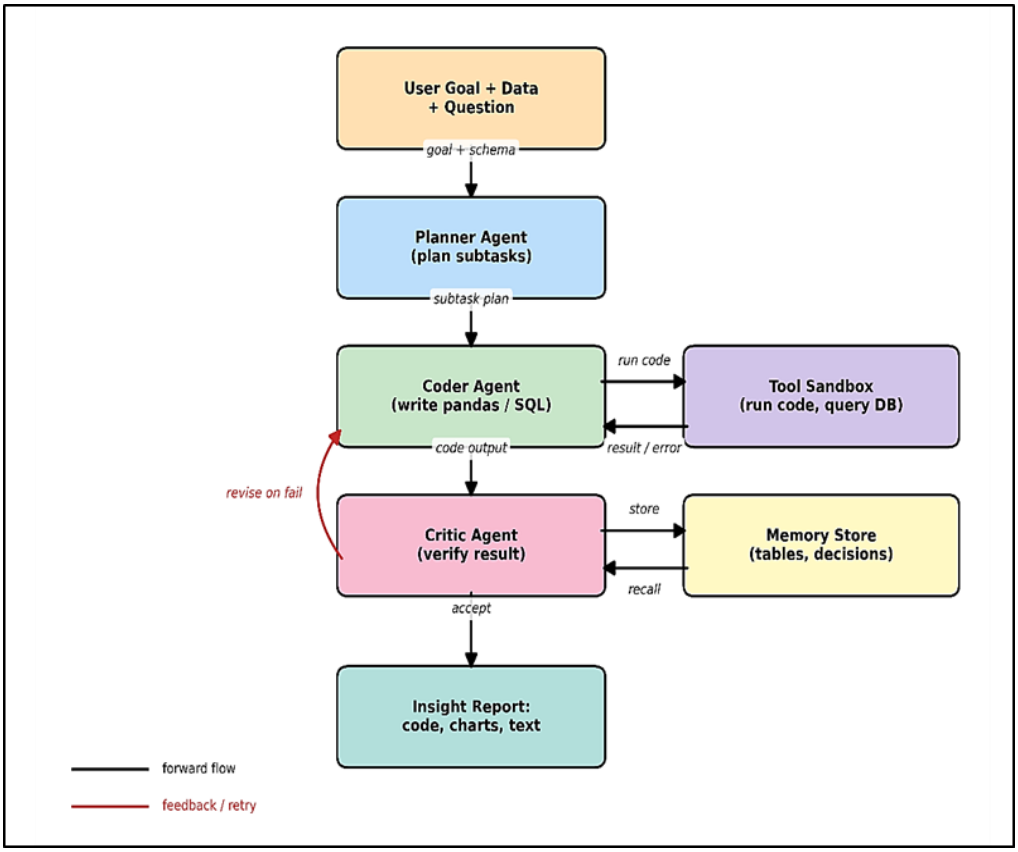


Fig. 1: Agentic data science pipeline. The planner, coder, and critic agents pass work down a central path, the coder calls a sandboxed tool to run code, and the critic returns failed steps for revision. Takeaway: control flows forward while a feedback path retries until the result passes.

Fig. 2 details the inner control pattern. Following ReAct [4], the agent emits a thought that names the current subgoal, an action that calls a tool, and then reads the observation the tool returns. The cycle repeats until a stop condition holds, for example a critic approval or a confidence threshold, at which point the agent emits a final answer. Because each action is logged with its inputs and outputs, the full trace is auditable after the run, which is the basis for the reproducibility checks discussed in Section VI.

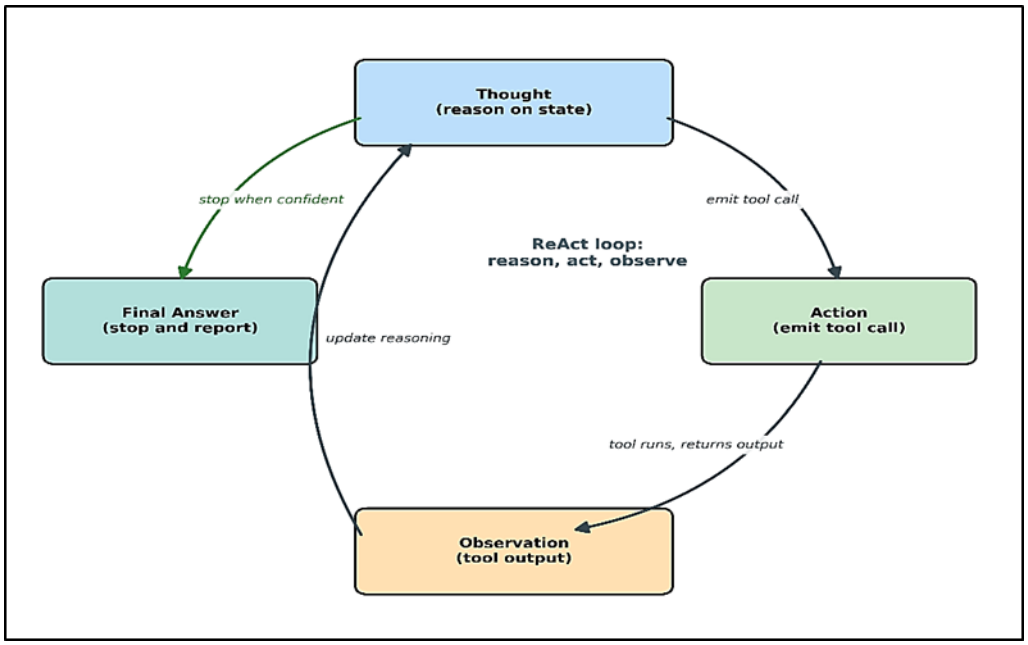


Fig. 2: ReAct control loop. A thought selects a tool call, the action runs it, and the observation feeds the next thought. Takeaway: each reasoning step is grounded in a real tool result before the agent stops and reports.

IV. CAPABILITIES AND METHODS

A. Automated Exploratory Data Analysis

Given a new table, the agent first profiles it: column types, cardinality, missing-value counts, summary statistics, and a sample of rows. From this profile the planner chooses plots and tests that fit the data. Numeric columns get histograms and correlation checks; categorical columns get frequency tables and group comparisons. The agent does not need a fixed template because it reasons over the observed schema. A useful discipline is to require the agent to state, before plotting, what it expects to see, so that a surprising result is flagged rather than silently accepted.

B. Code Generation for Pandas and SQL

Code generation is the workhorse capability. The coder writes pandas for in-memory frames and SQL for warehoused tables, picking the dialect from the connection metadata. Two practices raise reliability. First, schema grounding: the prompt carries exact column names and types, which cuts the rate of references to columns that do not exist. Second, execution feedback: when generated code raises an exception, the traceback is fed back so the next attempt fixes the specific error rather than rewriting from scratch. This mirrors the execution loop of code-interpreter assistants [12] and the error-driven repair seen in program synthesis benchmarks [18].

C. Hypothesis Generation and Ranking

Beyond running requested analyses, an agent can propose hypotheses. After profiling, it lists candidate relationships, for instance that a price variable depends on a region and a season, then ranks them by expected information value and testability. Each surviving hypothesis becomes a concrete test with a stated null, a chosen statistic, and a significance threshold fixed in advance. Pre-registering the test inside the trace guards against the agent tuning a threshold after seeing the result, which is a common way that automated search inflates false discoveries.

D. Multi-Agent Division of Labor

Complex projects benefit from specialization. A common split assigns one agent to data engineering, one to modeling, and one to review, coordinated by a shared plan, an arrangement that MetaGPT and AutoGen both support [9], [15]. Role separation reduces prompt length per agent and lets each role use settings suited to its job, such as low temperature for the critic. The cost is coordination overhead and the chance that agents disagree, which a planner must resolve through an explicit arbitration step.

E. Iterative Self-Correction

Self-correction is what turns a brittle generator into a usable analyst. After each step the critic checks the output against simple invariants: row counts should not exceed the source, group totals should reconcile, and a reported correlation must match the recomputed value. When a check fails, the critic returns a targeted message and the coder revises. Reflexion shows that a short verbal critique stored in memory improves the next attempt [13]. Section V quantifies how accuracy responds to the number of correction rounds.

V. EVALUATION AND RESULTS

We evaluate three execution modes on five task categories drawn from typical analyst work: data cleaning, EDA and statistics, SQL querying, feature engineering, and insight narrative. The single-shot mode issues one prompt with no tools. The agentic mode runs the planner, coder, and critic loop of Fig. 1 with a sandbox and up to five correction rounds. The human-in-the-loop mode adds an analyst who approves or edits the plan at two checkpoints. Scores are illustrative and meant to show the shape of the effect, not to certify a specific model.

Fig. 3 reports task success rate by mode and category. Single-shot prompting averages about 53 percent and is weakest on feature engineering and insight narrative, the two categories that need the most context. The agentic configuration averages about 75 percent, a gain that comes mostly from execution feedback catching errors that a one-shot prompt cannot see. Human-in-the-loop reaches about 88 percent, with the analyst mainly correcting plan level mistakes the agents could not catch alone. The ranking is consistent across all five categories.

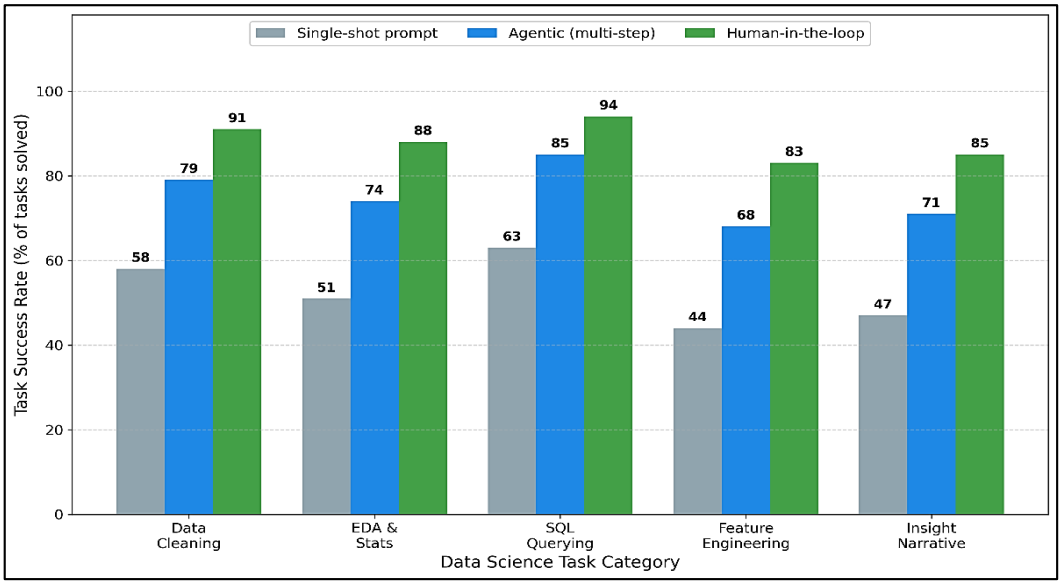


Fig. 3: Task success rate by execution mode across five task categories, with bar values labeled. Takeaway: the agentic loop beats single-shot prompting in every category, and human oversight adds a further margin.

Table 1. Aggregate performance and cost across execution modes (illustrative).

Execution Mode	Mean Success (%)	Median Latency (s)	Mean Tool Calls	Token Cost (rel.)
Single-shot prompt	52.6	4	0	1.0x
Agentic (multi-step)	75.4	38	11.2	6.8x
Human-in-the-loop	88.2	52	9.7	5.9x

Table 1 places the accuracy gains next to their cost. The agentic mode spends roughly seven times the tokens and an order of magnitude more wall-clock time than a single prompt, because it runs many model calls and tool executions per task. The human-in-the-loop mode is slightly cheaper in tokens than the fully autonomous mode because analyst guidance prunes wasted branches, though it adds human time that the table does not price. These trade-offs decide where autonomy pays off: high-value, repeated analyses justify the overhead, while a quick one-off question may not.Fig. 4 isolates the effect of self-correction. Starting from a first-attempt accuracy near 61 percent, accuracy climbs to about 84 percent by the fifth revision, with the largest jump between the first and second rounds. Over the same rounds the code execution error rate falls from 27 percent to under 3 percent as tracebacks are fed back and repaired. Returns diminish quickly: the gain from round four to round five is under one percentage point, so a budget of two or three corrections captures most of the benefit. Table 2 breaks down the residual error after correction by type.

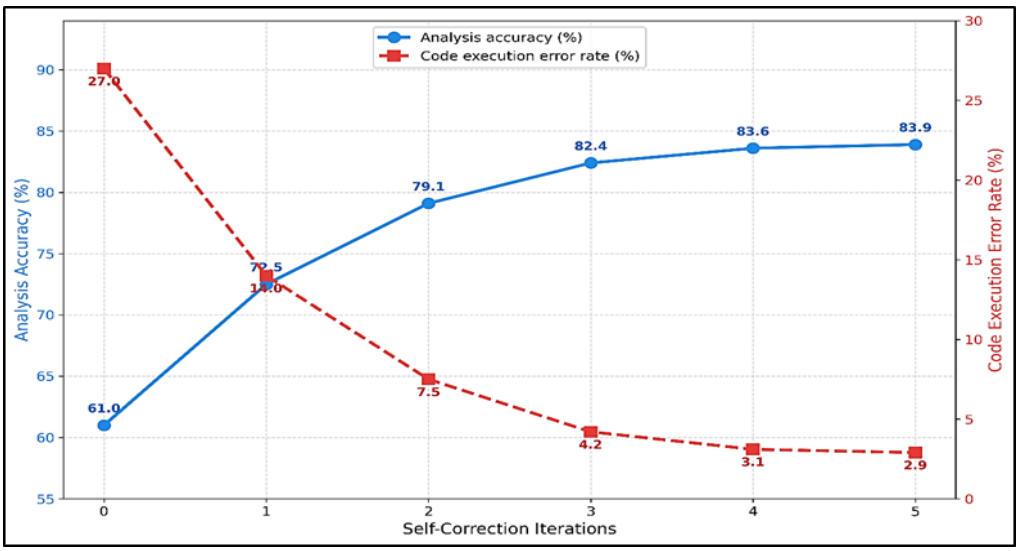


Fig. 4: Analysis accuracy (left axis, blue) and code execution error rate (right axis, red) against the number of self-correction rounds. Takeaway: most of the gain arrives by the second round, then returns flatten.

Table 2. Residual error composition after self-correction (illustrative).

Error Type	Share of Residual Errors (%)	Caught by Guardrail?
Hallucinated statistic	31	Partial (result audit)
Wrong column / schema mismatch	22	Yes (schema check)
Data leakage in split	17	Partial (pipeline lint)
Non-reproducible run	14	Yes (seed + log)
Misread question / scope	16	No (needs human)

Table 2 shows where automation still falls short. Hallucinated statistics, a number stated in prose that the code never computed, form the largest residual category and are only partly caught by auditing reported values against recomputed ones. Schema mismatches and non-reproducible runs are largely handled by deterministic checks. Misread questions, where the agent solves the wrong problem, remain a human responsibility. This pattern is the empirical case for keeping a person at the plan level even when execution is automated.

VI. RISKS AND LIMITATIONS

A. Hallucinated Statistics

The most dangerous failure in automated analysis is a confident wrong number. A model may write a clean narrative that cites a correlation or a p-value the code never produced. The defense is to compute every reported figure in the sandbox and to bind each sentence in the report to a value in the execution log. Any number in prose without a matching logged computation is rejected. This turns reporting into a verification problem, where the model may only state what the tools have measured.

B. Data Leakage and Reproducibility

Automated pipelines leak information easily. An agent may scale features using statistics from the full dataset before splitting, or select features using the test labels, both of which inflate measured performance. A pipeline lint that checks transform order and split boundaries catches common cases. Reproducibility needs fixed random seeds, pinned library versions, and a saved trace of every action, so a result can be rerun and produce the same output. Without these, an impressive run cannot be trusted because it cannot be repeated [10].

C. Safety, Cost, and Guardrails

Generated code is untrusted. The sandbox must deny network access, cap resources, and restrict file scope so that a tool call cannot exfiltrate or destroy data. Cost is a real limit: the six to seven times token overhead in Table 1 grows with task length, and unbounded loops can run away, so a hard step budget is required. Prompt injection through malicious data values is an open risk, since a crafted string in a cell can hijack an agent that reads it as instruction [8]. Guardrails reduce these risks but do not remove them, which is why the systems here are framed as assistants under supervision rather than autonomous decision makers.

D. Scope of These Results

The figures in this article are illustrative. They reflect the architecture and task mix described, not a single audited production deployment, and they will shift with the underlying model, the dataset, and the prompt design. Reported absolute numbers should be read as relative patterns: agentic loops beat single prompts, human oversight beats full autonomy, and self-correction has sharply diminishing returns. Those patterns are stable across the settings examined and are consistent with the published systems cited here [9], [10], [13].

VII. CONCLUSION

Agentic LLM systems can carry a data science task from raw files to a written insight with limited supervision. The combination that works is a planner, coder, and critic loop wired to a sandboxed interpreter, with the ReAct pattern keeping every step grounded in observed results. In the configurations studied, this lifted task success from roughly 53 to 75 percent over single-shot prompting, and self-correction pushed analysis accuracy toward 84 percent while driving execution errors below 3 percent. The harder lesson sits in the residual errors. Hallucinated statistics, data leakage, and misread questions are not solved by better generation; they are controlled by verification, deterministic checks, and a human at the plan level. The useful framing is that generation is cheap and verification is the binding constraint. Future work should standardize result auditing, build benchmarks that score reproducibility and leakage rather than only accuracy, and study how humans and agents should divide authority over an analysis. Used as supervised assistants with strong guardrails, these systems already shorten the path from data to defensible insight.

REFERENCES

[1] A. Vaswani et al., "Attention is all you need," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2017, pp. 5998–6008.

[2] L. Ouyang et al., "Training language models to follow instructions with human feedback," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2022.

[3] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2022.

[4] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. International Conference on Learning Representations (ICLR), 2023. arXiv:2210.03629.

[5] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2302.04761.

[6] L. Gao et al., "PAL: Program-aided language models," in Proc. International Conference on Machine Learning (ICML), 2023. arXiv:2211.10435.

[7] L. Wang et al., "A survey on large language model based autonomous agents," Frontiers of Computer Science, vol. 18, no. 6, 2024. arXiv:2308.11432.

[8] Z. Xi et al., "The rise and potential of large language model based agents: A survey," arXiv:2309.07864, 2023.

[9] S. Hong et al., "MetaGPT: Meta programming for a multi-agent collaborative framework," in Proc. International Conference on Learning Representations (ICLR), 2024. arXiv:2308.00352.

[10] S. Hong et al., "Data Interpreter: An LLM agent for data science," arXiv:2402.18679, 2024.

[11] X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," in Proc. International Conference on Learning Representations (ICLR), 2023. arXiv:2203.11171.

[12] OpenAI, "GPT-4 technical report," arXiv:2303.08774, 2023.

[13] N. Shinn et al., "Reflexion: Language agents with verbal reinforcement learning," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2303.11366.

[14] S. Yao et al., "Tree of thoughts: Deliberate problem solving with large language models," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2305.10601.

[15] Q. Wu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv:2308.08155, 2023.

[16] [16] Significant Gravitas, "AutoGPT: An autonomous GPT-4 experiment," GitHub repository, 2023. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT

[17] Y. Lai et al., "DS-1000: A natural and reliable benchmark for data science code generation," in Proc. International Conference on Machine Learning (ICML), 2023. arXiv:2211.11501.

[18] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.

[19] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?," in Proc. International Conference on Learning Representations (ICLR), 2024. arXiv:2310.06770.

[20] Y. Qin et al., "ToolLLM: Facilitating large language models to master 16000+ real-world APIs," in Proc. International Conference on Learning Representations (ICLR), 2024. arXiv:2307.16789.

[21] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2020. arXiv:2005.11401.