Frontiers in Computing and Intelligent Systems
ISSN: 2832-6024 | Vol. 18, No. 2, 2026

Large Language Model-Driven Data Analysis Agents: A Systematic Review of Architectures and Code Generation Strategies

Yueming Lan *

School of Business, Monash University Malaysia, Subang Jaya, 47500, Malaysia
*** Corresponding author Email:** ylan0014@student.monash.edu

Abstract: Autonomous data analysis, once the domain of experienced programmers, is being reshaped by large language models (LLMs) — a shift this paper examines through a structured systematic review. Models like GPT-4, Claude, and LLaMA have reached a level of proficiency in natural language understanding and code generation that now makes it feasible to embed them inside agent-based pipelines capable of planning, executing, and interpreting analytical tasks with little human intervention. To bring conceptual order to this rapidly growing body of work, the review proposes a two-dimensional taxonomy organized around two design choices: system architecture (single-agent versus multi-agent) and code generation modality (Python-centric versus SQL-centric). Within this framework, three lines of research emerge as dominant: systems built around a single execution agent, multi-agent collaborative pipelines, and text-to-SQL generation architectures. Drawing on representative work published at leading venues between 2023 and 2025, each line is assessed with respect to planning strategies, execution feedback loops, and error-correction mechanisms. A cross-cutting analysis finds that how a system handles execution feedback — rather than whether it uses one agent or several — is the factor that most reliably separates stronger from weaker designs. The paper closes by mapping out three underexplored problems: verifying code across modalities, developing unified evaluation benchmarks, and extending multi-agent coordination into SQL-oriented settings.

Keywords: Large Language Model; Data Analysis Agent; Multi-agent System; Text-to-SQL; Code Generation; Autonomous Data Science.

1. Introduction

Data analysis now sits at the center of how most large organizations operate — banks run credit risk models, hospitals scan patient records for early disease signals, retailers tune recommendation engines through continuous behavioral analysis. None of this is new. The bottleneck that has not gone away is access: doing any of it properly still requires some mix of programming fluency, statistical training, and domain knowledge. That combination is uncommon, and it remains a genuine obstacle for many practitioners [1].

LLMs trained on large corpora of text and code have begun to soften that barrier in ways that were not practical even a few years ago. Transformer-based models demonstrate a genuinely useful ability to parse natural language instructions and translate them into runnable programs [2], and researchers have moved quickly to exploit this. A substantial body of work now exists around agent-based systems that can take an analytical question, break it into sub-tasks, write and run code to address each one, examine the intermediate outputs, and package the results into a readable summary. The appeal is real: rather than requiring a data scientist to write every query by hand, these systems let domain experts pose questions in plain language and receive executable analyses in return.

1.1. Review of Prior Surveys

Existing survey efforts cover parts of this territory but none positions itself to address the two-axis taxonomy proposed here. Wang et al. [1] provide the most comprehensive general treatment, organizing LLM-based autonomous agents around four cognitive components — profiling, memory, planning, and action — across settings as varied as robotics, web navigation, and game playing. Data analysis surfaces as one application domain among many, which means the survey has no occasion to separate single-agent from multi-agent designs within this space, or to treat Python-based computation and SQL querying as distinct technical traditions that deserve their own axis. Sun et al. [3] take a narrower view, examining agents built specifically for statistical practice and data science, but they organize their analysis around pipeline stages (cleaning, exploration, modeling, reporting) rather than the architectural and code-generation distinctions that matter here. Text-to-SQL gets only a brief mention, treated as peripheral rather than as a coordinate dimension for comparison.

Two additional works approach the same space from adjacent angles and deserve mention. Luo et al. [4], contributing a SIGMOD 2026 tutorial, lay out a capability ladder that runs from simple natural-language-to-SQL converters up through fully autonomous analytical systems — a useful lens for gauging system maturity but not a systematic review of architectures, and it does not map agent type against code modality. Li et al. [2] bring a database-systems perspective, cataloguing LLM interventions across query optimization, data cleaning, and interface design, though agent orchestration receives only a brief treatment and no comparative framework of its own is developed.

Taken together, the four works confirm that this is an active and fast-moving field — but they also expose a structural gap: none of them treats architecture type and code-generation modality as two independent dimensions that can be crossed against each other. That gap is not merely presentational. Two

systems sharing the same architecture — say, two multi-agent designs — can behave in fundamentally different ways depending on whether they generate Python or SQL; and two SQL-centric systems can diverge sharply depending on whether work is handled by a single agent or coordinated across several. A single-axis survey structure, regardless of which axis it picks, tends to hide that interaction.

This paper addresses that gap directly. The proposed two-dimensional matrix organizes systems by architecture (single-agent vs. multi-agent) and code generation strategy (Python-centric vs. SQL-centric), and within that structure identifies three principal research streams for analysis: single-agent code execution systems, multi-agent collaborative pipelines, and text-to-SQL generation frameworks. For each stream, representative work published at top venues is examined with particular attention to task planning, execution feedback, and error correction. A cross-cutting discussion in Section 6 then pulls out execution feedback as the primary differentiator and identifies the open problems worth pursuing next.

The paper proceeds as follows. Section 2 sets out the classification framework. Sections 3 through 5 examine the three research streams in turn. Section 6 draws cross-stream comparisons, and Section 7 concludes.

2. Classification Framework

A shared vocabulary is needed before comparing systems across streams. The two-dimensional taxonomy adopted here is built around the two design decisions that, available evidence suggests, shape the behavior of LLM-driven analysis agents most consequentially.

2.1. Architecture Dimension

The first axis concerns system architecture. At the most basic level, this comes down to how many LLM-backed agents are doing the work. A single-agent design concentrates everything inside one model instance — intent parsing, planning, code generation, execution, interpretation — all within the same context window. The advantages are largely practical: the reasoning trace is self-contained, there is nothing to coordinate between agents, and when something goes wrong it is usually not hard to find where.

A multi-agent design distributes those responsibilities across two or more specialized roles — a planner, a coder, a reviewer — that communicate through defined channels. Specialization can raise output quality, but it also introduces overhead: coordination takes time, and poorly designed protocols can produce message loss or role confusion that a single-agent system never has to contend with.

2.2. Code Modality Dimension

The second axis describes what kind of code the system generates. Python-centric pipelines emit scripts that call libraries such as pandas, scikit-learn, and matplotlib — tools well suited to open-ended exploratory analysis where the analytical path cannot be fully planned in advance.

SQL-centric pipelines instead map natural language questions to structured queries that run against relational databases. Schema understanding and join reasoning replace general-purpose computation as the central challenge, and these systems fit naturally into enterprise settings where analytical questions can be expressed as set operations over normalized data.

2.3. The Resulting Taxonomy

Crossing the two axes produces four quadrants: single-agent Python, single-agent SQL, multi-agent Python, and multi-agent SQL. The published literature does not fill these evenly. Single-agent and multi-agent Python systems are well represented; SQL-oriented work is mostly single-agent or lightly pipelined. The multi-agent SQL quadrant is, at least among the systems surveyed here, the least populated — a point the discussion in Section 6.3 revisits.

Table 1. Representative systems mapped onto the proposed two-dimensional taxonomy.

	Python-Centric	**SQL-Centric**
Single-Agent	Data Interpreter [5]; DS-Agent [6]; TaskWeaver [7]	DIN-SQL [8]; CHASE-SQL [9]
Multi-Agent	MetaGPT [10]; LAMBDA [11]; MatPlotAgent [12]	MAC-SQL [13] (relatively underexplored)

3. Single-Agent Code Execution Systems

In a single-agent design, one LLM-backed controller handles the entire workflow: planning, code generation, execution monitoring, and error recovery, all within the same context window. There is no inter-agent messaging, no protocol to design, and one unified reasoning trace to follow. Three systems illustrate how much design space that constraint still leaves open.

3.1. Data Interpreter

Data Interpreter [5] (ACL 2025) organizes the analytical process as a directed acyclic graph (DAG) in which each node corresponds to one atomic coding step. After the LLM writes code for a node, the system runs it in a sandboxed environment and routes the resulting stdout, stderr, and variable states back into the prompt as context for the next iteration. Failures trigger re-examination of the error trace and regeneration of the offending code block, a cycle that repeats up to a configurable limit. Decomposing a complex request into fine-grained nodes means that Data Interpreter confines the impact of any single error — a mistake in one node does not automatically contaminate downstream steps — and makes selective re-execution possible without rerunning the entire pipeline. Independent nodes can also be executed in parallel, which can cut wall-clock time substantially for analyses with branching structure.

3.2. DS-Agent

DS-Agent [6] (ICML 2024) takes a retrieval-based path rather than a planning-based one. Before tackling a new task, the system queries an external knowledge base for solved cases that are structurally similar to the current request, then injects those solution patterns into the prompt. This case-based reasoning (CBR) approach reduces reliance on in-context few-shot examples — which can be brittle when the

task involves domain-specific choices like feature engineering on tabular data or hyperparameter tuning for gradient-boosted models. On Kaggle-style machine learning benchmarks, CBR-augmented generation outperforms unassisted prompting by a meaningful margin, with the gains concentrated in precisely the tasks where prior solved examples offer the most transferable patterns: feature construction and model selection.

3.3. TaskWeaver

TaskWeaver [7] is organized around extensibility. Users register custom Python functions as named tools that the LLM can call; a planner then decomposes each user request into a sequence of such plugin invocations, interspersed with any free-form code needed to connect them, and a code-generation module turns that plan into a runnable script. Critically, a stateful session manager preserves variable bindings across turns, so that a later query in a conversation can pick up where an earlier one left off. For organizations maintaining proprietary analytical functions that must not be modified, this plugin-first architecture is a natural fit.

3.4. Summary of Single-Agent Approaches

Taken together, these three systems suggest that a single agent with the right scaffolding

4. Multi-Agent Collaborative Systems

The core idea is that dedicated, specialized agents can apply more focused attention to their assigned sub-tasks than a single generalist agent handling everything

4.1. MetaGPT

MetaGPT [10] (ICLR 2024) maps the analytical process onto a software engineering team. Separate agents handle product management, architecture design, coding, and testing; each operates under a standard operating procedure (SOP) that defines acceptable inputs and expected outputs; a shared message board routes inter-agent communication; and a global workspace accumulates intermediate artifacts. Applied to data analysis, the architecture agent draws up the analytical plan, the coder agent implements it, and the tester agent checks outputs against assertions. Evaluations show that catching coding errors at the tester-agent stage — before they propagate into downstream interpretation — reduces cascading failures relative to single-agent baselines. That said, more active roles means more coordination overhead, and on compact, well-defined tasks the system can end up slower than a single well-prompted agent.

4.2. LAMBDA

LAMBDA [11], which appeared in the Journal of the American Statistical Association in 2025, is aimed squarely at statistical analysts who are not necessarily programmers. Three specialized agents collaborate: an innovator that selects appropriate statistical methods, a code generator that translates the chosen methodology into runnable scripts, and an inspector that checks both methodological soundness and code correctness before surfacing results to the user. Evaluations on tasks drawn from statistical practice find that this tripartite structure produces analyses that align more closely with expert statistician judgments than single-agent alternatives do. The separation of method selection from code generation is what most clearly distinguishes LAMBDA — systems that assume the analytical method is already decided and focus only on implementation are a different category.

4.3. DA-Code and MatPlotAgent

Two further systems round out the multi-agent picture by addressing complementary sub-problems. DA-Code [14] (EMNLP 2024) is a benchmark rather than an analytical system — its contribution is a standardized evaluation suite covering data wrangling, statistical modeling, and visualization tasks across diverse domains. Before DA-Code existed, comparing different agent systems was complicated by the fact that they were typically tested on incompatible task sets; the benchmark provides a common measurement surface.

MatPlotAgent [12] (ACL 2024) focuses on just one part of the analytical pipeline: visualization. A plotting agent writes matplotlib code; a critic agent evaluates the resulting figure on aesthetic quality, data-ink ratio, and informational clarity; and the two iterate in a generate-critique-revise loop until quality thresholds are met. The narrow focus makes an important broader point: multi-agent collaboration does not have to span the entire workflow to be useful. Even a single, well-scoped sub-task can benefit from splitting generation from evaluation.

4.4. Summary of Multi-Agent Approaches

Multi-agent coordination pays off when tasks are complex enough, and long-running enough, to justify the overhead. The literature is clear, however, that this benefit is conditional. On tightly scoped, well-defined tasks, message-passing overhead and role negotiation can actually slow things down relative to a single well-configured agent. The advantage becomes most pronounced for open-ended, multi-step analyses where the ability to isolate errors in specific roles — and review them with specialized focus — translates into measurable quality gains.

5. Text-to-SQL Generation Systems

Text-to-SQL systems occupy a specialized corner of the LLM-driven analysis space. Rather than producing general-purpose Python, they convert natural language questions into SQL queries that run directly against relational databases. The scope is narrower, but the technical demands are different in character: accurate schema comprehension, multi-table join reasoning, and sensitivity to query optimization constraints are required in ways that open-ended Python generation does not surface as directly.

5.1. DIN-SQL

DIN-SQL [8] (NeurIPS 2023) introduced a decomposition approach that has since influenced the design of many successor systems. Rather than asking the LLM to generate a SQL query in one shot, DIN-SQL chains several sub-tasks: the model first identifies relevant tables and columns (schema linking), then classifies the query type (simple, join, nested, etc.), and finally assembles the SQL statement step by step. On the Spider benchmark, this sequential approach produces substantially higher accuracy than end-to-end prompting. A self-correction step, in which the model inspects its own draft for syntactic and logical problems before finalizing it, adds another layer of reliability.

5.2. MAC-SQL

MAC-SQL [13] (COLING 2025) extends DIN-SQL's decomposition logic into a multi-agent architecture, explicitly occupying the multi-agent SQL quadrant that most text-to-

SQL research leaves empty. A schema selector agent first narrows the relevant portion of the database schema, passing a focused context to a drafter agent; the drafter produces an initial SQL statement; and a refiner agent executes it, examines runtime errors and suspicious empty result sets, and iterates until the query is both syntactically valid and semantically plausible. This three-phase pipeline achieves competitive results on cross-database benchmarks and stands as one of the clearer demonstrations that multi-agent coordination can be applied meaningfully within the SQL domain.

5.3. CHASE-SQL

CHASE-SQL [9] (ICLR 2025) takes a different route entirely, generating multiple candidate queries rather than refining a single one. Diverse prompting strategies — chain-of-thought, divide-and-conquer, and direct translation — each produce a candidate SQL statement; a preference-optimized selector, trained on query quality annotations, then ranks them and picks the most promising for execution. The motivation is a specific failure mode that sequential refinement is prone to: once a model commits to a plausible-but-wrong initial query, iterative revision often cannot escape it. Generating a diverse pool of candidates and selecting among them sidesteps this anchoring problem.

5.4. Benchmarks and Evaluation Resources

Progress in text-to-SQL has been shaped by the evaluation resources available to the field. The BIRD benchmark [15] (NeurIPS 2023) has proven particularly influential: unlike earlier benchmarks, it uses large-scale, real-world databases with complex schemas and deliberately ambiguous natural language questions, exposing system weaknesses that tidier test sets tend to hide. Gao et al. [16] provide a comprehensive empirical comparison of LLM-based text-to-SQL methods under reproducible conditions, helping to standardize performance reporting across research groups. Hong et al. [17] survey the next-generation database interface landscape more broadly and document evaluation gaps that remain open, including limited coverage of multi-dialect SQL and cross-lingual queries.

6. Discussion

Single-agent code execution, multi-agent collaboration, and text-to-SQL generation all rest on the same foundation — an LLM doing the core reasoning — but they differ substantially in how they organize the work that surrounds that core. Comparing them along several dimensions surfaces both consistent patterns and persistent blind spots.

6.1. Execution Feedback as a Core Differentiator

Looking across all three paradigms, one factor appears more consistently linked to task performance than others: how execution feedback is handled — that is, what a system does with the information produced after code or a query runs. Single-agent systems like Data Interpreter close a tight execute-observe-revise loop inside a single context window, appending error messages and intermediate results to the conversation history. Multi-agent systems hand that interpretation to a dedicated reviewer or inspector agent. Text-to-SQL systems apply schema-aware validation and runtime error analysis. In each case, how well this feedback loop is designed seems to correlate with task success more than the architecture choice itself does. For researchers building new systems, this suggests treating feedback loop design as a primary concern rather than something to address during implementation.

6.2. Architecture Selection Is Task-Dependent

Architecture selection, then, should follow task characteristics — how decomposable the task is, how high the error rate is likely to be at each stage — rather than any general assumption that more agents means better results.

6.3. The Multi-Agent SQL Gap

MAC-SQL [13] is the main exception — and it is worth noting how unusual that makes it.

6.4. Evaluation Fragmentation

Progress is also slowed by the absence of benchmarks that span all three paradigms. Python-centric work is evaluated on Kaggle-style challenges, custom task suites, or the DA-Code benchmark [14]; text-to-SQL work converges on Spider, BIRD [15], and related cross-database sets. No widely used benchmark requires a system to handle a complete analytical session that starts with a SQL query, loads the results into a data frame, applies statistical transformations, and produces a visualization — yet that is what real-world analysts do routinely. Building such cross-modality benchmarks would make comparisons across research streams much more meaningful and would better reflect what deployment actually requires.

6.5. Notable Domestic Contributions

Several contributions from Chinese research institutions deserve specific mention in this context. SiriusBI [18], a joint project between Tencent and Peking University reported in VLDB proceedings, achieves SQL accuracy exceeding 93% on internal business intelligence tasks by coupling schema-aware prompting with execution-guided refinement inside a production-grade architecture. It should be noted that this figure is derived from proprietary internal benchmarks and has not been validated against publicly reproducible evaluation suites. Table-GPT [19], developed at Microsoft Research, takes a fine-tuning approach rather than a prompting one, adapting LLMs specifically for tabular reasoning tasks; the performance gains on table understanding and manipulation are substantial, and it is included here for its technical relevance rather than geographic origin. The BIRD benchmark [15] has itself become a global reference point, providing more realistic test conditions than earlier alternatives. Taken together, SiriusBI and BIRD in particular fill gaps in the research landscape — one on the side of industrial deployment at full pipeline scale, the other on evaluation infrastructure.

7. Conclusion

This review has examined LLM-driven data analysis agents through a two-dimensional taxonomy organized by architecture type and code generation modality. Three research streams were identified and analyzed in detail: single-agent systems relying on DAG planning, case-based retrieval, or plugin registries; multi-agent pipelines that distribute analytical work across specialized roles via structured messaging; and text-to-SQL architectures that use decomposition, diverse candidate generation, and preference-

based selection to convert natural language into database queries.

Table 2. Comparative summary of representative systems across key design dimensions (end-to-end analysis systems only; DA-Code and MatPlotAgent are excluded as they serve specialized rather than full-pipeline roles).

System	Venue	Architecture	Code Modality	Planning	Feedback
Data Interpreter	ACL 2025	Single	Python	DAG	Execute-revise loop
DS-Agent	ICML 2024	Single	Python	CBR retrieval	Case-guided retry
TaskWeaver	arXiv 2023	Single	Python	Plugin chain	Stateful session
MetaGPT	ICLR 2024	Multi	Python	SOP roles	Tester-agent review
LAMBDA	JASA 2025	Multi	Python	Tripartite	Inspector agent
DIN-SQL	NeurIPS 2023	Single	SQL	Decomposition	Self-correction
MAC-SQL	COLING 2025	Multi	SQL	Three-phase	Runtime validation
CHASE-SQL	ICLR 2025	Single	SQL	Multi-path	Candidate selection

A few things emerge fairly consistently from this analysis. How a system handles execution feedback seems to matter more for task outcomes than whether it uses one agent or several — which suggests feedback loop design deserves more explicit attention in future work. Multi-agent coordination offers real benefits for open-ended, complex tasks, but introduces coordination costs that can be counterproductive on simpler, well-scoped ones. Within the taxonomy, the multi-agent SQL quadrant is underexplored relative to what enterprise database settings seem to require. And the field still lacks a benchmark that integrates Python computation and SQL querying within a single analytical workflow, which limits how meaningfully the three paradigms can be compared.

As LLM capabilities continue to develop — longer context, more reliable code generation, stronger multi-step reasoning — the systems reviewed here will likely evolve in ways that are not easy to predict in advance. The taxonomy in this paper is not meant as a definitive classification. It is more of a working frame: one way to organize what has been built so far, and to make room for what comes next.

Acknowledgments

The author would like to thank the instructors and teaching assistants for their guidance throughout the research process.

References

[1] Wang, L., Ma, C., Feng, X., et al. (2024). A survey on large language model based autonomous agents. Frontiers of Computer Science.

[2] Li, G., Zhou, X., & Zhao, X. (2024). LLM for data management. Proceedings of the VLDB Endowment, 17(12), 4213–4216.

[3] Sun, M., Han, R., Jiang, B., et al. (2025). A survey on LLM-based agents for statistics and data science. The American Statistician.

[4] Luo, Y., et al. (2026). Data agents: Levels, state of the art, and open problems. SIGMOD Tutorial.

[5] Hong, S., Lin, Y., Liu, B., et al. (2025). Data interpreter: An LLM agent for data science. In ACL Findings.

[6] Guo, S., Deng, C., Wen, Y., et al. (2024). DS-Agent: Automated data science by empowering LLMs with case-based reasoning. In ICML.

[7] Qiao, B., Li, L., Zhang, X., et al. (2023). TaskWeaver: A code-first agent framework. arXiv:2311.17541.

[8] Pourreza, M., & Rafiei, D. (2023). DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In NeurIPS.

[9] Pourreza, M., Li, H., Sun, R., et al. (2025). CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In ICLR.

[10] Hong, S., Zheng, X., Chen, J., et al. (2024). MetaGPT: Meta programming for multi-agent collaborative framework. In ICLR.

[11] Sun, M., Han, R., Jiang, B., et al. (2025). LAMBDA: A large model based data agent. Journal of the American Statistical Association.

[12] Yang, Z., Zhou, Z., Wang, S., et al. (2024). MatPlotAgent: Method and evaluation for LLM-based agentic scientific data visualization. In ACL Findings.

[13] Wang, B., Ren, C., Yang, J., et al. (2025). MAC-SQL: A multi-agent collaborative framework for text-to-SQL. In COLING.

[14] Huang, Y., Luo, J., Yu, Y., et al. (2024). DA-Code: Agent data science code generation benchmark for LLMs. In EMNLP.

[15] Li, J., Hui, B., Qu, G., et al. (2023). Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs. In NeurIPS.

[16] Gao, D., Wang, H., Li, Y., et al. (2024). Text-to-SQL empowered by large language models: A benchmark evaluation. Proceedings of the VLDB Endowment, 17(5), 1132–1145.

[17] Hong, Z., Yuan, Z., Zhang, Q., et al. (2025). Next-generation database interfaces: A survey of LLM-based text-to-SQL. IEEE Transactions on Knowledge and Data Engineering.

[18] Jiang, J., Xie, H., Shen, S., et al. (2025). SiriusBI: A comprehensive LLM-powered solution for data analytics in business intelligence. Proceedings of the VLDB Endowment, 18(12), 4860–4873.

[19] Li, P., He, Y., Yashar, D., et al. (2024). Table-GPT: Table fine-tuned GPT for diverse table tasks. Proceedings of the ACM on Management of Data, 2(3), 176.

[20] Grosnit, A., Maraval, A., et al. (2024). Large language models orchestrating structured reasoning achieve Kaggle grandmaster level. arXiv:2411.03562.