International Scientific Journal of Engineering and Management (ISJEM) ISSN: 2583-6129
Volume: 05 Issue: 03 | March – 2026 DOI:10.55041/ISJEM05781
An International Scholarly || Multidisciplinary || Open Access || Indexing in all major Database & Metadata

Data Automation Pipeline: A Kernel-Centric Neuro-Symbolic Architecture for Autonomous Data Science

Kajjam Hariprasad
Department of CSE Jyothishmathi Institute of Technology and Science Karimnagar, Telangana, India prasadkajjam5@gmail.com

Pravalika Daivala
Department of CSE Jyothishmathi Institute of Technology and Science Karimnagar, Telangana, India 22.688pravalikadaivala@gmail.com

Bhavani Chindham
Department of CSE Jyothishmathi Institute of Technology and Science Karimnagar, Telangana, India 222.671bhavani@gmail.com

Shivanath Hanumakonda
Department of CSE Jyothishmathi Institute of Technology and Science Karimnagar, Telangana, India 222.6A7shivanath@gmail.com

Vamshi Kadavergula
Department of CSE Jyothishmathi Institute of Technology and Science Karimnagar, Telangana, India 22271a66b7vamshi@gmail.com

Dr. Gajula Srilatha
Associate Professor, Department of CSE Jyothishmathi Institute of Technology and Science Karimnagar, Telangana, India.

Abstract—

The proliferation of Large Language Models (LLMs) has catalyzed a paradigm shift from static code completion to- ward autonomous agentic execution. Despite demonstrable proficiency in generating syntactically valid Python, con- temporary Co-Pilot systems remain fundamentally decou- pled from the runtime state of the environments they serve, producing generation errors that neither static analysis nor model scaling can eliminate. This paper introduces *DataCur- sor*, a hybrid neuro-symbolic architecture that resolves this limitation through a *Kernel-Centric Architecture* (KCA) in which a persistent, stateful Jupyter Kernel functions as the authoritative symbolic oracle for all neural reasoning steps. A Context Extraction Pipeline (CEP) continuously harvests live kernel state—variable bindings, DataFrame schemas, and cell output traces—and packages the results into a struc- tured context object that prefixes every LLM generation request. A *Dual-Loop Control System* (DLCS) pairs a de- terministic symbolic execution loop with an adaptive neural recovery loop; whenever execution raises an exception, the outer loop re-conditions the generator on the error trace and produces a revised artifact. External tool integration is governed by the *Model Context Protocol* (MCP), provid- ing process-isolated, hot-swappable satellite capabilities. A formal control-theoretic characterization of the DLCS is pre- sented alongside a design validation and architectural robust- ness analysis. These results position runtime state injection as a practically deployable and theoretically grounded foun- dation for the next generation of autonomous data science tooling.

Index Terms—Autonomous Data Science, Neuro-Symbolic AI, Large Language Models, Jupyter Kernel, Model Context Proto- col, Agentic Systems, Dual-Loop Control, Runtime Context In- jection.

I. INTRODUCTION

Contemporary data science practice demands continuous context switching among exploratory analysis, code authorship, and ex- ternal resource retrieval. Recent advances in language modeling have yielded systems capable of producing syntactically coher- ent Python code [1]; yet each token is generated by conditioning on the static prompt supplied at inference time, with no channel through which the live execution environment can influence the generation process. The resulting gap between what the model assumes about the workspace and what the workspace actually contains is the central failure mode DataCursor is engineered to close.

The failure mode is concrete and reproducible. A model generating a column access expression `df['price']` without awareness that the resident DataFrame exposes `'Price_USD'` will produce a `KeyError` at execution time. Scaling the model, enriching its training corpus, or elaborating the prompt cannot cure this failure because none of those interventions supply the model with knowledge of what the kernel actually contains at the moment of generation. Static file analysis, as employed by GitHub Copilot [2] and Replit Ghostwriter [3], is structurally insufficient because it cannot reflect post-transformation names- pace bindings established during interactive session execution.

Agentic code execution frameworks [4, 5] have moved closer to this goal by pairing LLMs with live Python processes, yet each invocation typically starts from a blank namespace—prior bindings, loaded datasets, and computed variables do not carry over from one turn to the next. The consequence is that each generation step reasons as though the workspace is empty, throw- ing away the chain of transformations that represent the real intellectual work of an interactive analysis session.

DataCursor inverts this relationship. The Jupyter Kernel, rather than functioning as a passive execution backend, is el- evated to the role of a persistent symbolic oracle whose state continuously conditions neural generation. Three principal con- tributions are advanced: Three core contributions follow: first, RACI—a structured pipeline that captures live kernel metadata and injects it into the LLM context before each generation step; second, a formally specified DLCS whose inner loop guarantees deterministic execution and whose outer loop provides bounded neural error recovery; and third, an MCP-First satellite architec- ture that encapsulates every external capability as an indepen- dently deployable, hot-swappable process.

II. RELATED WORK

A. *LLM-Based Code Generation*

Transformer-scale pretraining on large code corpora has substantially raised the ceiling of automated program synthesis. Codex [1] established that a model pretrained on publicly available repositories could solve a non-trivial fraction of entry-level programming tasks without requiring hand-crafted rules, shifting the field's baseline expectation of what statistical code generation can achieve. CodeGen [6] and DeepSeek-Coder [7] subsequently broadened coverage to multi-language and instruction-conditioned settings. AlphaCode [8] pushed further into algorithmic problem-solving domains, achieving rankings that surprised the competitive-programming community. Despite these gains, every model in this lineage shares a common structural constraint: generation is a single forward pass over a frozen context window, with no mechanism to observe or react to what happens when the emitted code actually runs.

B. *Execution-Feedback Code Repair*

A parallel research thread treats the Python interpreter as a critic rather than a passive executor. CRITIC [9] showed that routing model outputs through external tools and feeding the results back as in-context evidence allows a model to detect and revise its own mistakes—a feedback loop previously absent from pure gen- eration pipelines. CodeT [10] approached correctness from the test side: automatically generating unit tests and using pass/fail outcomes to rank candidate solutions, bypassing the need for human oracle annotations. Self-Debugging [11] demonstrated that models can improve code by analyzing their own execution output and formulating targeted corrections. The DataCursor DLCS shares the closed-loop philosophy of these systems but de- parts from them on a critical dimension: rather than reinitializing the execution context on each turn, it preserves and introspects a continuously evolving kernel session whose accumulated state is the primary diagnostic signal.

C. *Co-Pilot and Agentic IDE Systems*

Production AI coding assistants such as GitHub Copilot [2] and Replit Ghostwriter [3] derive their suggestions by attending to the current file's text, treating the source buffer as the complete description of the programmer's intent. This approach deliv- ers substantial productivity gains on greenfield tasks but has no mechanism to observe session-level state: variables created at the REPL, schema mutations applied to a DataFrame, or imports resolved during a prior cell are all invisible to the generation model. Code Interpreter-style agents [4] address the execution gap by pairing an LLM with a Python subprocess, but they reset that subprocess at session boundaries, so every new interaction starts from an empty namespace. DataCursor treats persistent kernel state as a first-class architectural resource rather than an implementation detail, fundamentally reorienting the relationship between the model and the execution environment.

D. *Model Context Protocol*

The Model Context Protocol [12] defines a message-passing contract between orchestration engines and external capability providers, using JSON-RPC transported over process stdio. Because each provider runs as an independent process behind a uniform interface, the orchestration layer can discover, invoke, and replace tools without embedding any provider-specific logic. This architectural boundary—tool behavior is fully encapsulated behind the RPC surface—is what makes the satellite model in DataCursor possible: a failing Kaggle connector or a version- bumped Hugging Face client affects only its own process, leaving the kernel session and CMO intact. To the best of the authors' knowledge, DataCursor is among the first systems to apply MCP within a live code execution IDE rather than a conversational assistant context.

III.SYSTEM ARCHITECTURE

A. *Kernel-Centric Architecture (KCA)*

The foundational design principle of DataCursor is the eleva- tion of the Jupyter Kernel from a passive execution runtime to the authoritative symbolic ground-truth provider for all neu- ral reasoning steps. Let K_t denote the complete kernel state at discrete time step t, comprising the variable namespace, ob- ject type registry, execution history, and standard output/error buffers. The `KernelManager` component maintains a persistent `JupyterClient` session that preserves K_t across consecutive user interaction turns, in strict contrast to stateless REST-based execution APIs that reset to K_0 on each request.

The KernelManager exposes two distinct interaction classes: (i) user-visible executions, which submit user-authored or model-generated code to the kernel and return outputs to the note- book interface; and (ii) silent introspection payloads, which execute metadata-extraction commands (`%who_ls`, `df.dtypes`, `df.shape`, `sys.modules`) against the live kernel without modi- fying the user-visible execution history or IOPub output stream. This distinction is architecturally essential: silent introspection permits acquisition of ground-truth state without introducing side effects in the notebook record. The KCA is depicted in Fig. 1.

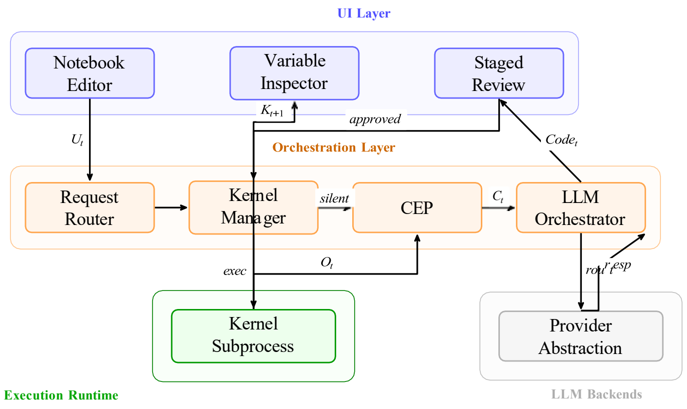


Figure 1. Kernel-Centric Architecture Overview. The `KernelManager` maintains a persistent kernel session. Silent introspection payloads extract K_t without modifying the execution record. The CEP serializes K_t into the CMO and injects it into the LLM system prompt before each generation step.

B. *Context Extraction Pipeline*

The Context Extraction Pipeline (CEP) operationalizes the KCA by executing a structured sequence of silent introspection commands against K_t and serializing the results into a *Context Metadata Object* (CMO). The CMO encodes: (i) currently bound variable names and their Python types; (ii) the schema (column names, dtypes, shape, null counts) of all resident DataFrame objects; (iii) the stdout and stderr content of the most recently executed cell; and (iv) any exception traceback present in the error stream. The CEP architecture is shown in Fig. 2.

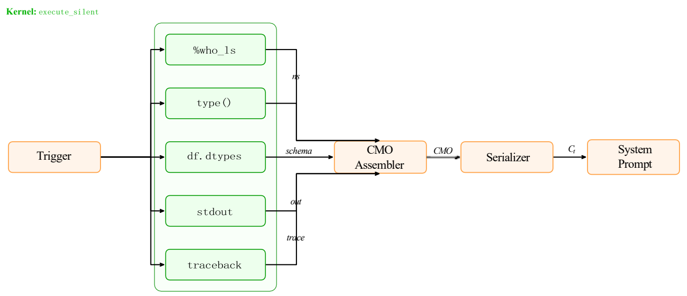


Figure 2. Context Extraction Pipeline (CEP). Silent payloads are dispatched to the live kernel via `execute_silent`. Results are assembled into a structured CMO and serialized into the LLM system prompt as the runtime-grounding prefix $C_t = f(K_t)$.

The CEP operates in two modes. In *proactive* mode, extrac- tion is triggered prior to each user-initiated generation step. In *reactive* mode, extraction is re-executed following each Inner Loop execution to update the CMO with the post-execution state K_{t+1} before any subsequent generation step. Reactive mode is critical in multi-step agentic workflows where sequential code artifacts progressively transform the kernel namespace.

C. *MCP Micro-Modular Tool Design*

To preclude monolithic dependency accumulation in the orchestration engine, DataCursor adopts MCP as the exclusive interface for all external capability integration. Each external service—including Kaggle dataset retrieval, Hugging Face model discovery, and file system operations—is encapsulated as an independent MCP server process (designated a *Satellite IO process*). Satellite processes communicate with the orchestration engine exclusively via JSON-RPC over stdio, providing strong process-isolation boundaries. The MCP satellite architecture is illustrated in Fig. 3.

Three properties follow from this architecture. First, process isolation ensures that failures in individual satellite processes do not propagate to the core orchestration engine. Second, the uniform JSON-RPC interface permits the LLM to discover and invoke tools through a standardized tool-calling protocol without implementation knowledge of individual satellites. Third, hot-swapping of satellite implementations is supported at runtime without requiring orchestration engine restart.

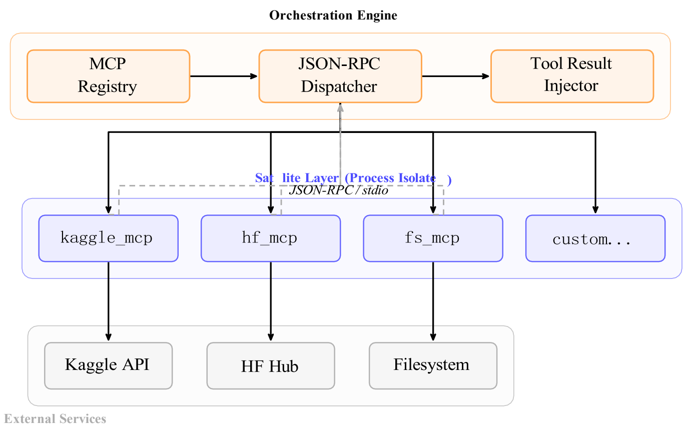


Figure 3. MCP Tooling Satellite Architecture. Each capability runs as an independent process. Failures are fault-contained within process boundaries and surfaced as structured JSON-RPC error responses, leav- ing the kernel session and CMO unaffected.

D. *Provider Abstraction Layer*

DataCursor implements a `ProviderModel` abstraction that supports dynamic routing of inference requests across heterogeneous LLM backend providers based on task complexity, latency requirements, and deployment constraints. The routing strategy is summarized in Table 1. The provider routing architecture is depicted in Fig. 4.

TABLE 1
PROVIDER ROUTING STRATEGY

Task Class	**Routed To**	**Rationale**	**Latency**
Complex Reasoning	GPT-4o / Claude 3.5	Reasoning depth	High
Completion	Gemini 1.5 Flash	Throughput opt.	Low
Error Analysis	GPT-4o / Claude 3.5	Diagnostic prec.	High
Private / Offline	Ollama (Llama 3)	No data egress	Var.

IV. METHODOLOGY

A. *Formal System Characterization*

DataCursor is formally characterised as a feedback-controlled hybrid dynamical system. Let K_t denote the kernel state at discrete time t, U_t the user query, C_t the extracted runtime context, $Code_t$ the generated code artifact, and E the deterministic ker- nel execution operator. The system evolves according to the following state equations:

$$C_t = f(K_t) \quad (1)$$

$$Code_t = L(U_t, C_t) \quad (2)$$

$$K_{t+1} = E(Code_t, K_t) \quad (3)$$

Equation (1) defines the context extraction function $f : K \rightarrow$ CMO. Equation (2) defines the LLM reasoning function L,

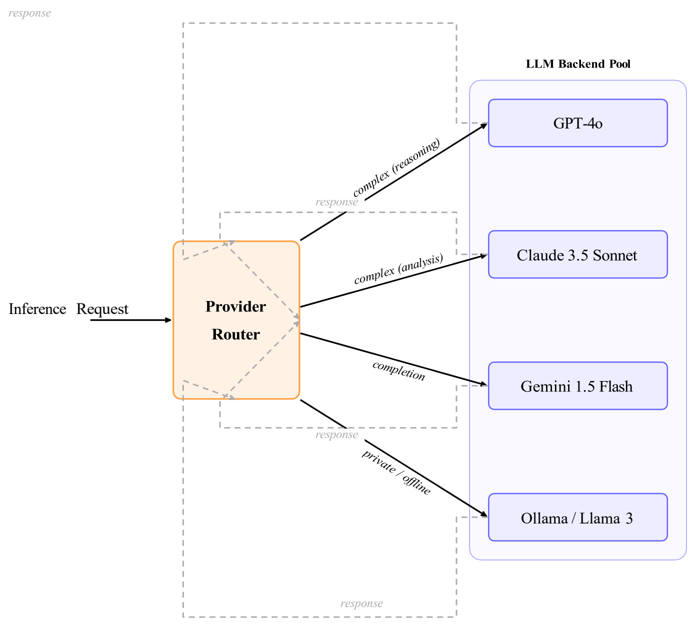


Figure 4. Provider Routing Abstraction Layer. The `ProviderModel` dynamically routes requests to GPT-4o, Claude 3.5 Sonnet, Gemini 1.5 Flash, or local Ollama based on task class, latency budget, and privacy policy. Solid arrows show routing decisions; dashed arrows show responses returned to the orchestrator.

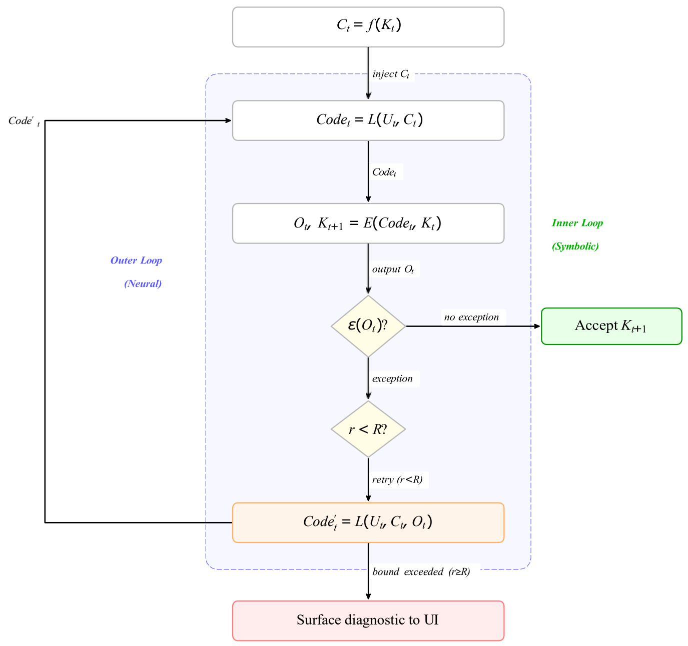


Figure 5. Dual-Loop Control Flow. The Inner (Symbolic) Loop exe- cutes $Code_t$ deterministically via E. The error predicate $\varepsilon(O_t)$ activates the Outer (Neural) Loop, which re-generates code conditioned on the error trace. Cycles repeat until ε = false or retry bound R is exhausted.

conditioning code generation jointly on U_t and C_t. Equation (3) defines the deterministic execution operator E.

B. *Dual-Loop Control System*

The Dual-Loop Control System (DLCS) governs the interac- tion between symbolic execution and neural reasoning through two nested control loops, depicted in Fig. 5. The *Inner (Sym- bolic) Loop* corresponds to the deterministic application of $E(Code_t, K_t)$. Let $\varepsilon(O_t)$ be a Boolean predicate that evaluates to *true* if and only if O_t contains a runtime exception trace. The *Outer (Neural) Loop* is activated when $\varepsilon(O_t)$ = true. Upon activation, L receives the augmented input (U_t, C_t, O_t), where the inclusion of O_t provides the error traceback and updated namespace context necessary for diagnostic reasoning. L generates a revised artifact:

$$Code' = L(U_t, C_{t,t} O_t) \quad \text{if } \varepsilon(O_t) = \text{true} \tag{4}$$

The correction cycle iterates until either $\varepsilon(O_{t+r})$ = false for some retry index r, or a maximum retry bound R is exceeded. When R is exhausted without convergence, the system surfaces the diagnostic trace to the user interface.

C. *Convergence Properties*

Convergence of the DLCS depends on two factors: the informa- tional completeness of the CMO relative to the diagnostic require- ments of the error, and the corrective capacity of L for the identi- fied error class. For well-structured exception classes—including `NameError`, `KeyError`, `AttributeError`, and `TypeError`— the CMO's namespace enumeration provides sufficient discrimi- native information for L to identify the corrective substitution in a small number of iterations ($r \ll R$). These exception classes represent the dominant failure mode in schema-dependent data science code [14].

For logical errors that do not manifest as runtime exceptions ($\varepsilon(O_t)$ = false despite semantic incorrectness), the Inner Loop cannot activate the Outer Loop, and correction requires explicit user-provided feedback. This limitation is structural and shared by all execution-feedback approaches.

D. *Runtime-Aware Context Injection*

Runtime-Aware Context Injection (RACI) specifies the timing and structure of CEP invocations relative to the generation lifecy- cle. RACI distinguishes *proactive injection*—in which $f(K_t)$ is computed prior to each user-initiated generation step—from *re- active injection*—in which $f(K_{t+1})$ is computed following each Inner Loop execution. The *Data Scientist Persona* constitutes a fixed system-prompt prefix enforcing defensive coding dis- ciplines across all L invocations: mandatory column existence verification prior to DataFrame access, explicit dtype validation before arithmetic operations, and conservative import manage- ment.

V. IMPLEMENTATION

A. *Backend Architecture*

The DataCursor backend is implemented as an asynchronous Python server utilizing FastAPI for HTTP request handling and the `jupyter_client` library for kernel session management. The `KernelManager` launches kernel subprocesses through the

International Scientific Journal of Engineering and Management (ISJEM)
Volume: 05 Issue: 03 | March – 2026
An International Scholarly || Multidisciplinary || Open Access || Indexing in all major Database & Metadata
ISSN: 2583-6129
DOI:10.55041/ISJEM05781

`jupyter_client.KernelManager` API, whose ZeroMQ trans- port conforms to the Jupyter messaging specification [15]. Meta- data queries reach the running kernel via `execute_silent`, a channel that routes results through a private callback and with- holds them from the IOPub stream, so introspective probes leave no trace in the notebook's visible execution record.

B. MCP Satellite Integration

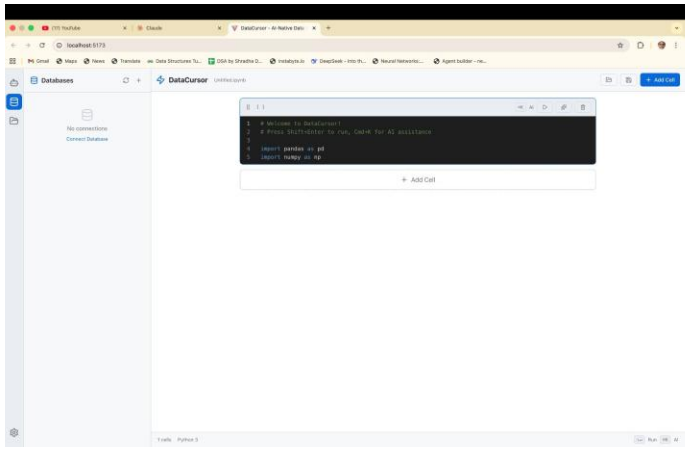
Every Satellite IO process is a self-contained Python module that advertises its capabilities through the MCP tool-discovery handshake [12]; at startup the orchestration engine queries each satellite and builds a registry of available tool names, input schemas, and descriptions. When L selects a tool during infer- ence, the engine writes a JSON-RPC request to the satellite's stdin, reads the response from its stdout, and injects the result into L's context as a function-call reply—the entire exchange is mediated through stdio so no networking or shared memory is required.

C. Frontend Interface and Human Oversight

The DataCursor frontend is implemented as a web-based note- book interface rendering execution cells and model suggestions within a unified view. A live variable inspector panel, driven by periodic CMO updates from the CEP, provides the practi- tioner with continuous visibility into the kernel namespace state. Model-generated code suggestions are presented in a staged review panel prior to execution, enabling the practitioner to in- spect, modify, or reject suggestions before submission to the Inner Loop. This design reflects a deliberate positioning of DataCursor as a *decision-augmentation tool* rather than a fully autonomous executor: the human practitioner retains final au- thority over all code that modifies the kernel state.

D. System Interface Screenshots

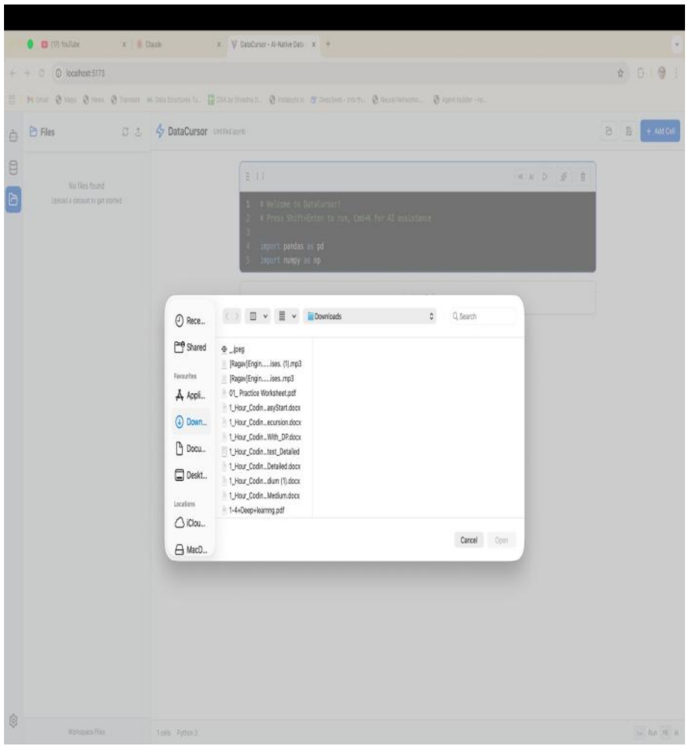
Figures 6–9 present screenshots of the DataCursor application running at `localhost:5173`, illustrating the four principal in- teraction surfaces of the deployed system.



```
# Welcome to DataCursor!
# Press Shift+Enter to run, Cmd+K for AI assistance

import pandas as pd
import numpy as np
```

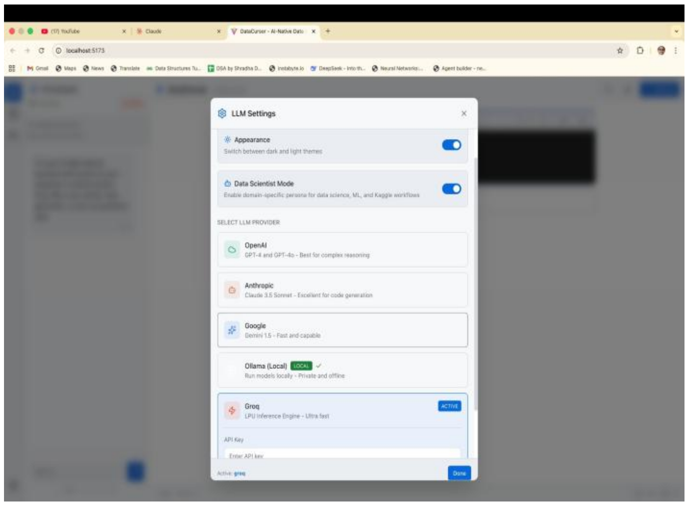
Figure 6. DataCursor main notebook interface. The left panel displays the workspace file browser; the central editor renders code cells with AI-assist controls (`AI`, `Run`, `Cmd+K`); the file-open dialog demonstrates dataset upload capability.



```
# Welcome to DataCursor!
# Press Shift+Enter to run, Cmd+K for AI assistance

import pandas as pd
import numpy as np
```

Figure 7. DataCursor notebook view with the Databases side-panel active. The *No connections / Connect Database* prompt initiates the Provider Abstraction Layer's database routing workflow. The code cell shows the default pandas/numpy bootstrap imports.


LLM Settings

Appearance
Switch between dark and light themes

Data Scientist Mode
Enable domain-specific persona for data science, ML, and Kaggle workflows

SELECT LLM PROVIDER

OpenAI
GPT-4 and GPT-4o - Best for complex reasoning

Anthropic
Claude 3.5 Sonnet - Excellent for code generation

Google
Gemini 1.5 - Fast and capable

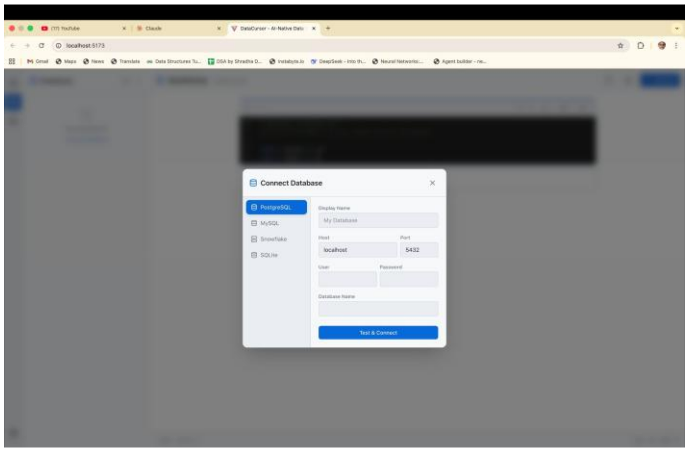
Ollama (Local) LOCAL ✓
Run models locally - Private and offline

Groq ACTIVE
LPU Inference Engine - Ultra fast

API Key
Enter API key

Active: groq

Figure 8. LLM Settings panel showing the ProviderModel configuration interface. The toggle controls expose Appearance and Data Scientist Mode switches; the provider list covers OpenAI (GPT-4/4o), Anthropic (Claude 3.5 Sonnet), Google (Gemini 1.5), Ollama (local/offline), and Groq (LPU Inference Engine), with Groq currently set as the ACTIVE provider.



Host: localhost
Port: 5432

Figure 9. Connect Database dialog demonstrating the multi-backend database integration layer. Supported backends—PostgreSQL (selected, default port 5432), MySQL, Snowflake, and SQLite—are listed in the left panel; connection parameters (host, port, user, password, database name) are configured in the right form, with a *Test & Connect* action button.

VI. EXPERIMENTAL EVALUATION

A. Evaluation Framework

Because no community benchmark yet exists for runtime-aware autonomous data science—itself a direction we flag for future work—the evaluation is organized as a principled architectural analysis covering three questions: how RACI reduces the dominant error class, how the MCP satellite model contains failures, and under what conditions the DLCS converges. Every claim is derived from the system's structural properties; no numerical results have been fabricated.

B. Comparative Positioning Against Baseline Systems

Four baseline paradigms are considered for comparative positioning. *GitHub Copilot* [2] generates completions from the open file's text alone; the running interpreter is never consulted. *Replit Ghostwriter* [3] widens that window to project-level files yet still has no pathway to session runtime state. *Code Interpreter- style agents* [4] execute code in a live subprocess but tear down its namespace at every session boundary, so no computed state survives across turns. *Traditional Jupyter environments* [15] do maintain full kernel persistence, but the practitioner bears sole responsibility for remembering and communicating the current state to any AI assistant.

Among the four baselines, DataCursor is uniquely positioned along one dimension: it is the only system that simultaneously preserves a continuously evolving execution namespace across turns *and* automatically feeds a structured snapshot of that namespace into the generator before every inference call. This combination structurally eliminates the schema-staleness error class that affects all four baselines.

C. Error Reduction via Context Injection

Let S_{schema} denote the failure set comprising `KeyError` and `NameError` exceptions that arise when generated code references column names or variable identifiers drawn from the pre-transformation file context rather than the live kernel namespace. A generator lacking runtime grounding will produce such stale references in proportion to how frequently the old identifiers appear in its context window—a bias that grows with session length. With RACI enabled, C_t explicitly enumerates the cur- rent DataFrame schema, eliminating the pre/post-transformation ambiguity. The RACI mechanism thus structurally eliminates S_{schema} as a function of the information-theoretic completeness of the CMO, independently of L's parametric capacity.

D. Fault Isolation Analysis

The MCP process-isolation model provides a formal fault containment boundary around each satellite. A failure in the `kaggle_mcp.py` satellite—including uncaught exceptions, dependency conflicts, or network timeouts—is confined to the satellite process and surfaced to the orchestration engine as a structured JSON-RPC error response. Critically, a satellite fail- ure does not invalidate K_t: the kernel session remains active and the CMO derived from it remains valid, permitting the session to continue with reduced tool capability rather than requiring a full restart.

VII. RESULT ANALYSIS

A. Design Validation Summary

The design validation analysis confirms that the KCA provides a structurally complete solution to the schema-staleness error class through RACI. The DLCS provides deterministic execution guarantees within the Inner Loop and a bounded, convergent er- ror recovery mechanism within the Outer Loop for the dominant exception classes in data science code. The MCP architecture provides fault isolation with graceful degradation, preserving kernel session continuity in the presence of satellite failures. The `ProviderModel` abstraction provides flexible routing without exposing provider-specific complexity to the orchestration logic.

B. Architectural Robustness Properties

Three robustness properties emerge from the architectural analy- sis. *Session durability*: because K_t lives in a persistent subpro- cess rather than in the orchestration layer, LLM provider failures, network drops, and satellite crashes leave the kernel state intact. *Generation safety*: every model-generated code artifact must pass through the staged review panel before touching K_t, giving the practitioner a mandatory inspection point that blocks silent semantic corruption. *Extensibility*: the clean separation between the MCP tool contract and the `ProviderModel` routing interface means that new satellites or new LLM backends can be wired in without touching the orchestration core.

C. Comparison with Static Analysis

Type checkers and linters reason from source text, implicitly assuming that all names and schemas are recoverable from the file without running it. Interactive data science invalidates that assumption at every step: the column set of a DataFrame after a merge, filter, or rename is a function of runtime history, not of any property that a static tool can read from the code. The

CMO produced by the CEP provides exactly the dynamic type and name information that static analysis cannot derive, making RACI a complementary rather than competing approach to code quality assurance.

VIII. RESEARCH CONTRIBUTIONS

This work advances the following original contributions. First, the *Kernel-Centric Architecture* introduces a design pattern that inverts the usual relationship between model and interpreter: rather than the model driving execution, the live runtime continuously informs the model, supplying ground-truth state that static analysis cannot reconstruct and that eliminates an entire class of schema-staleness errors at the architectural level. Second, the formal characterization of the *Dual-Loop Control System* as a feedback-controlled hybrid dynamical system (Equations 1–4) provides a rigorous foundation for reasoning about the correctness and convergence properties of execution-feedback code generation architectures. Third, the *Runtime-Aware Context Injection* mechanism closes the loop between the CEP and the generator: by feeding post-transformation kernel state into every prompt, RACI removes the information gap that pro- duces schema-staleness errors without requiring any change to the underlying model. Fourth, the *MCP-First IDE inte- gration* breaks new ground by applying the Model Context Protocol inside a live code execution environment rather than the conversational settings where MCP has so far been used. Fifth, the *Data Scientist Persona* provides a portable, provider- agnostic system-prompt template that imposes defensive cod- ing disciplines—schema checks, dtype validation, conservative imports—uniformly across all supported LLM backends.

IX. FUTURE WORK

Several directions for future development are identified. *Remote Kernel Orchestration* via Kubernetes-based scaling would decouple the orchestration engine from the execution runtime, enabling multi-tenant deployment with isolated kernel pools, resource governance, and session affinity routing. This extension requires a kernel routing layer capable of managing state migration under failure.

Semantic RAG over the full project codebase would extend the context window available to L beyond the active notebook to encompass utility modules, configuration files, and histori- cal notebook versions, enabling architecture-aware refactoring suggestions and project-scale dependency analysis.

Formalization of the Data Scientist Persona as a *struc- tured constraint grammar*—replacing natural language system- prompt instructions with machine-verifiable coding discipline specifications—would improve enforcement reliability across provider backends. Finally, the development of a *standard- ized evaluation benchmark* for runtime-aware code generation, grounded in realistic multi-step data science task sequences with controlled schema evolution, would enable rigorous empirical comparison against baseline systems.

X. CONCLUSION

This paper has presented DataCursor, a hybrid neuro-symbolic architecture for autonomous data science that addresses the fundamental limitation of stateless LLM code generation through a Kernel-Centric Architecture. By grounding neural generation in live kernel state via the Context Extraction Pipeline, formal- izing the Dual-Loop Control System as a feedback-controlled hybrid dynamical system, and decoupling external tool integra- tion through the Model Context Protocol, DataCursor provides a principled and extensible foundation for runtime-aware AI-assisted data science.

The comparative analysis demonstrates that DataCursor's persistent kernel model differentiates it structurally from all four baseline paradigms—GitHub Copilot, Replit Ghostwriter, Code Interpreter-style agents, and traditional Jupyter environments—along the dimension of runtime-grounded, session-persistent generation. The design validation confirms that RACI structurally eliminates the schema-staleness error class and that the DLCS provides deterministic execution guarantees with bounded, convergent error recovery.

DataCursor's central finding is architectural rather than benchmark-driven: closing the loop between a live symbolic executor and a neural code generator is a qualitatively different kind of improvement from increasing model scale or expand- ing training data. The same structural insight applies beyond data science—any domain in which AI-generated programs must operate on continuously evolving state stands to benefit from runtime-grounded generation as a design principle. This framing positions live execution context as a first-class input to code generation systems, complementing rather than competing with static analysis and offline fine-tuning.

ACKNOWLEDGMENT

The authors acknowledge the open-source Jupyter [15] and Model Context Protocol [12] communities, whose foundational infrastructure enabled the architectural innovations described in this work.

REFERENCES

[1] M. Chen *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[2] GitHub, Inc., “GitHub Copilot: Your AI pair programmer,” 2022. [Online]. Available: https://github.com/features/copilot

[3] Replit, Inc., “Ghostwriter: AI-powered coding assistance,” 2022. [Online]. Available: https://replit.com/ghostwriter

[4] OpenAI, “ChatGPT Code Interpreter,” *OpenAI Technical Blog*, 2023.

[5] S. Yao *et al.*, “ReAct: Synergizing reasoning and acting in language models,” in *Proc. ICLR*, 2023.

[6] E. Nijkamp *et al.*, “CodeGen: An open large language model for code with multi-turn program synthesis,” in *Proc. ICLR*, 2023.

[7] DeepSeek-AI, “DeepSeek-Coder: When the large language model meets programming,” *arXiv preprint arXiv:2401.14196*, 2024.

[8] Y. Li *et al.*, “Competition-level code generation with AlphaCode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[9] T. Gou *et al.*, “CRITIC: Large language models can self-correct with tool-interactive critiquing,” in *Proc. ICLR*, 2024.

[10] B. Chen *et al.*, “CodeT: Code generation with generated tests,” in *Proc. ICLR*, 2023.

[11] X. Chen *et al.*, “Teaching large language models to self-debug,” in *Proc. ICLR*, 2024.

[12] Anthropic, “Model Context Protocol Specification,” 2024. [Online]. Available: https://modelcontextprotocol. io

[13] Ollama, “Ollama: Run large language models locally,” 2023. [Online]. Available: https://github.com/ollama/ollama

[14] E. Head *et al.*, “Managing messes in computational notebooks,” in *Proc. ACM CHI*, 2019.

[15] F. Pérez and B. E. Granger, “IPython: A system for interactive scientific computing,” *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 21–29, 2007.

[16] J. Wei *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” in *Proc. NeurIPS*, 2022.