Beyond the Shell: Extending Agents with Reactive Python Notebooks

Trevor Manz
trevor@marimo.io
marimo

Myles Scolnick
myles@marimo.io
marimo

Akshay Agrawal
akshay@marimo.io
marimo

ABSTRACT

For iterative computational work in research, data analysis, and ML, humans have widely adopted stateful programming environments. A live runtime keeps state in memory, supporting inspection and iteration as work takes shape. We introduce a design pattern that lets a coding agent work inside the environment directly. The agent acts on the runtime by writing code in its host language. The runtime's structure holds the agent's working state, and its rules constrain what the agent produces. These three properties match three structural needs of such work, namely a long tail of operations, persistent in-memory values, and coherent iteration over partial state. We instantiate the pattern as **marimo pair**[1], which lets a coding agent drive a live marimo Python kernel through a single code action. marimo's rules (reactive recompute, scrubbed-on-delete, transactional mutations) accumulate the agent's exploration into a runnable, reproducible Python program. An agent can pair with a human in the same live notebook or drive an analysis end-to-end on its own.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; • **Computing methodologies** → *Artificial intelligence*; • **Human-centered computing** → Human computer interaction (HCI).

KEYWORDS

agentic coding systems, reactive notebooks, code mode, computational notebooks, agent environments, marimo

1 INTRODUCTION

A language model's effectiveness depends on its scaffold [22], comprising the available tools, their environment, and the control loop that composes them. The scaffold shapes both how well the model works and what it produces.

Today's agentic coding systems (Claude Code, Codex, Cursor) externalize their working context into a shell, with files holding intermediate state between invocations. Recursive language models [22] (RLMs) make the pattern explicit using a Python REPL where context lives as in-memory variables. The shell is itself a thin REPL with the same offloading property. Each invocation starts fresh, and intermediate state must be marshaled to disk between steps.

For iterative computational work (research, exploratory data analysis, algorithm prototyping, ML), humans have long worked in stateful environments such as Excel for spreadsheets, RStudio for R, and Jupyter for Python [7, 11]. Notebooks support thinking and storytelling with code and data [4]. They provide persistent in-memory state across steps, selective re-execution rather than restart, and a literate artifact combining code, output, and prose [13]. Agentic coding systems on a default file-and-shell scaffold miss these affordances.

We introduce a design pattern in which a coding agent works inside a stateful programming environment. The agent operates a live runtime through a single host-language code action. The runtime's structure is the agent's offloaded context store, and its semantics constrain the artifact. The pattern extends rather than replaces the agent's environment. An agent can pair with a human in a live notebook or drive an analysis end-to-end on its own. Instantiated as marimo pair[2], it lets a coding agent drive a live marimo[3] Python kernel, where marimo's rules (reactive recompute, scrubbed-on-delete, transactional mutations) accumulate the agent's exploration into a runnable, reproducible Python program. Liu et al. [9] argue data systems must be redesigned to support *agentic speculation*, alongside related proposals at CIDR 2026 [14, 15]; our work is the parallel move at the runtime layer.

2 BACKGROUND

An agent's action substrate shapes what it can do. ReAct [19] structures tool-mediated agency through an interleaved Thought–Action–Observation loop. Code-as-action, in which the language model emits executable code rather than structured tool calls, broadens this substrate. Liang et al.'s Code-as-Policies [8] drives robots from generated code, Gao et al.'s PAL [3] routes math reasoning through a Python interpreter, Wang et al.'s CodeAct [18] demonstrates Python actions outperform JSON tool-calling on agent benchmarks, and Nguyen et al.'s DynaSaur [10] extends the substrate to dynamic action sets. Cloudflare's code mode [1] applies the same shape to wrapped APIs. Our work sits in this lineage at a different layer. The wrapped surface is a stateful programming runtime, and the agent's Python runs in a kernel that exposes mutations through a small Python module (§3).

Recursive language models (RLMs) [22] are the closest academic articulation of our setting. The language model sees only the query while a Python REPL holds the context as an in-memory variable; the action space is `execute_code` plus `FINAL(...)`; the language model writes code that greps, slices, and summarizes the context, optionally invoking other language model calls recursively. marimo pair shares this substrate but operates in a different setting. The REPL is the user's live notebook rather than a per-task ephemeral store, state persists across sessions and is shared with a human, there are no recursive sub-calls, and the byproduct is the notebook itself.

[1] https://github.com/marimo-team/marimo-pair

[2] Named for pair programming, though the agent can also work on its own.

[3] https://github.com/marimo-team/marimo

marimo notebooks are reactive Python. Cell dependencies are extracted statically, and the runtime treats the notebook as an authoritative dataflow graph. Editing a cell re-runs its downstream automatically; deleting a cell scrubs the variables it defined; the notebook's `.py` file is regenerated from the kernel's state. The runtime guarantees no hidden cross-cell state and determines execution from the dataflow graph rather than the agent's run order (in contrast to Jupyter, where state depends on which cells were run when). §3.3 leans on these properties.

3 THE PATTERN: OPENING UP THE RUNTIME

Our aim is to connect marimo, a stateful Python runtime, to agentic coding systems for iterative computational work. The statefulness that makes such an environment useful for humans is what makes it hard to expose to an agent acting from outside it. MCP is a standard for connecting AI applications to external systems. We built a marimo MCP server to expose the live notebook state through read-only tools like `get_lightweight_cell_map`, `get_tables_and_variables`, and `get_cell_dependency_graph`. These tools surfaced structure invisible from the `.py` file, including which cells existed, which variables were in scope, and how the dataflow graph wired them together. Writes went through the agent's existing file-editing tools. Marimo observed the resulting filesystem edits and applied them to the runtime.

However, both halves had limits. The read tools only exposed what we had anticipated. The agent often wanted to ask the next question, or to inspect an object the tools did not cover. Edits had the opposite problem. The agent could change anything in the file, but errors stayed in the notebook, so it could not tell whether its edit had worked. The split between curated reads and filesystem writes kept the agent outside the runtime. Cloudflare [1] reports a related dynamic on wrapped APIs.

The alternative is a single code action, in which the agent writes Python directly *against* the runtime with full visibility into its state. The runtime is the control plane, intercepting actions to apply its semantics and giving fast feedback for self-correction. The pattern has three properties (§3.1–§3.3). P2 and P3 are orthogonal. Jupyter exposes rich structure without runtime invariants, while a capability-restricted sandbox imposes invariants without introspectable structure. marimo pair instantiates the pattern by letting a coding agent drive a live marimo Python kernel, with a small Python module (`marimo._code_mode`) as its interface to the notebook (Figure 1).

3.1 A single code action

The host language is the action surface. The agent's payload is unmodified Python, evaluated in the kernel with read access to all cell variables. New bindings live in a scratchpad and disappear when the call ends. To make persistent notebook changes, the agent uses `marimo._code_mode` (§3.2).[4] Anything Python can do, the agent can do (`df.head()`, fitting a model, plotting) without an intervening discrete-operation surface. P1 alone is not novel. Zhang et al.'s

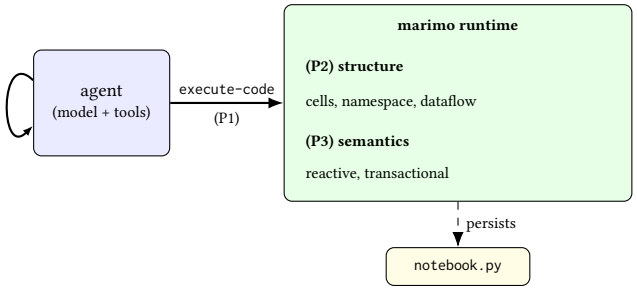

Figure 1: Architecture. The agent (left) operates in a model-and-tools loop and reaches the runtime (right) through a single code action (P1). The runtime is the medium for the agent's work: its structure (cells, namespace, dataflow graph) is the offloaded context store the agent inspects and mutates (P2); its semantics (reactive recompute, scrubbed-on-delete, transactional mutations) shape what the agent's actions produce (P3). The notebook persists as a `.py` file across sessions.

L0 [23] and You et al.'s *DatawiseAgent* [20] run agents in Jupyter kernels via code-execution loops, and ChatGPT Code Interpreter[5], Open Interpreter[6], Jupyter AI[7], agentnb[8], and mcp-repl[9] also realize P1. The pattern adds two more properties (P2, P3), reachable *through the same single action surface.*

3.2 The runtime as offloaded context store

The kernel's state (cells, namespace, reactive dataflow graph) is the agent's offloaded context store, reachable from inside the code action through a small Python module the kernel itself exposes:

```
import marimo._code_mode as cm

# API is discoverable via help(cm), etc.

async with cm.get_context() as ctx:
    # cells are Python objects: ctx.cell[cid].code,
        output, ...
    cid = ctx.create_cell("df.head()")
    ctx.run_cell(cid)
```

Unlike a JSON tool surface, `cm` does not mediate the agent's access at all. The agent has one action surface, the code action, and `cm` is reached *inside* that surface, not alongside it. The code action is a tool, but its payload is unstructured host-language code, and the agent's expressivity is Python's, not a fixed schema's. A mediated tool surface adds a second control plane the agent's code has to coordinate with. Every step alternates between writing Python and emitting a tool call. Here, `cm` is part of the same code the agent already writes. Calls compose with arbitrary host-language constructs (loops, comprehensions, locally defined helpers) and remain discoverable through `help(cm)` at runtime. An unbounded set of operations is reachable through a fixed set of API entry points.

[4] Currently exposed as a shell command (`execute-code.sh`) over HTTP. The transport is incidental: an MCP server, a host-SDK function call, or any channel that delivers a Python string to the kernel realizes the same surface.

[5] https://openai.com/blog/chatgpt-plugins
[6] https://github.com/OpenInterpreter/open-interpreter
[7] https://github.com/jupyterlab/jupyter-ai
[8] https://github.com/oegedijk/agentnb
[9] https://github.com/posit-dev/mcp-repl

3.3 Runtime semantics as constraints

The runtime applies invariants at specific execution points, not by restricting the action surface. Three constraints carry most of the weight in marimo:

(a) *reactive recompute*: editing a cell re-runs its downstream automatically, freeing the agent from tracking the dependency structure;
(b) *scrubbed-on-delete*: deleting a cell removes its defined variables from kernel memory, so stale state cannot accumulate;
(c) *transactional mutations*: `cm` operations queue inside an async context and flush on exit, so the reactive graph sees a coherent batch rather than partial state:

```
async with cm.get_context() as ctx:
    cid = ctx.create_cell("x = expensive_load()")
    ctx.run_cell(cid)
# operations queue; on exit, flushed as one batch
```

These constraints are how marimo already works. At commit, marimo dry-runs the batch through three static checks for syntax, multiply-defined names, and cycles in the dependency graph. These are the same checks marimo runs when a human edits the notebook. If any check fails, the transaction is rejected and no notebook state changes. Direct writes to the `.py` file are silently overwritten by the kernel, so changes only persist through `cm`. P3's design move is to put constraints at intervention points rather than restricting the action surface. In practice, the agent inspects state through the scratchpad and commits batches through `cm`. The runtime either rejects a failing batch before any cells change, or accepts a passing one into the dataflow graph and writes it back to the `.py` file. Reactive recompute, scrubbed-on-delete, and transactional mutations are marimo's vocabulary for the move; other runtimes might realize P3 with MVCC-style isolation, capability-restricted contexts, or branch-and-merge primitives. We return to this generality in §5.

3.4 Implementation

marimo pair packages the pattern as an agent skill paired with a single shell tool. The skill is intentionally declarative. It points the agent at where the runtime exposes its surface (`marimo._code_mode`) and how to discover it via `help(cm)`, rather than enumerating operations or sequencing work. `marimo._code_mode` is a semi-private interface, not a versioned API. Its consumer is a model that reads docs and reasons about what it finds, not a program calling stable signatures, so the skill is loosely coupled to the runtime version.

4 STRUCTURAL FIT TO DATA WORK

Iterative computational work has three structural features that the file-system substrate of an agentic coding system does not carry well:

D1. Persistent in-memory values. Notebooks hold fitted models, dataframes, intermediate plots, and partial pipelines. These values don't fit in a prompt and are often expensive or non-deterministic to re-derive from source [5, 12].

D2. A long tail of operations. What an analyst wants depends on the data's concrete state, including its columns, dtypes, value distributions, and edge cases. A static tool surface enumerates a fixed set of operations, but the long tail of useful queries outpaces any such enumeration.

D3. Coherent iteration over partial state. Exploration produces dead ends like wrong cells edited, variables that should be gone, or packages that break the environment. Without runtime help, partial state accumulates across actions until errors compound [5].

We address each pair in turn.

4.1 P1 ↔ D2: Long tail of operations

P1 is the host language as the action surface. D2's operations depend on values the agent cannot see at planning time, and P1 lets the agent see those values by running code at planning time. We saw this directly during the MCP detour (§3). Every new tool we added was a thin Python wrapper over marimo's state, until we removed the wrappers and let the agent write Python.

DSBench [6] reports best-agent accuracy at 34% on notebook-style analysis tasks, where success often depends on inspecting values the agent cannot see in the prompt.

4.2 P2 ↔ D1: Persistent in-memory values

P2 is the runtime's structure (cells, namespace, dataflow graph) as the agent's offloaded context store. D1's values don't fit in a prompt and are expensive to re-derive, while P2 already holds them in memory by virtue of being the kernel where they were computed. The agent inherits the index (names, cells, dependencies) without maintaining a parallel representation. Without P2, the agent would have to either reload values from source on every action (often impossible for partial pipelines) or summarize them into its context window (often inadequate for high-cardinality state). P2 makes the runtime do the indexing for the agent.

P2 corresponds to what Liu et al. [9] call an *agentic memory store*, but at the runtime layer rather than the data-system layer. Their store is queryable across tasks and indexes table-level metadata as a pseudo-index for future probes. marimo's runtime indexes live values within one session and does not today reach across sessions. We do not claim marimo matches every feature of their proposal, only that the runtime is one valid home for an agent's working memory.

4.3 P3 ↔ D3: Coherent iteration

P3 is the runtime's semantics as constraints. Three marimo constraints cover three failure modes that arise during iterative exploration:

(a) *Reactive recompute* prevents **stale dependents**: a downstream cell reading a value its upstream has already updated.
(b) *Scrubbed-on-delete* prevents **zombie variables**: names that survive their defining cell's removal.
(c) *Transactional mutations* prevent **partial-batch state**: an exception mid-sequence that leaves the dataflow graph half-updated.

D3 is the accumulation of inconsistent partial state across exploratory actions, and each constraint eliminates one source of inconsistency. Without these constraints, each failure mode would compound across actions, with stale dependents leaking into downstream operations, zombie variables shadowing new bindings, and partial-batch state leaving the dataflow graph in

mixed conditions. The agent would have to inventory and reason about each. P3 collapses them into runtime invariants the agent does not have to track. Liu et al. [9] use *steerability* for runtime feedback the agent acts on. Our constraints apply automatically, whether the agent asks or not.

5 DISCUSSION

marimo pair makes three design choices. First, marimo's rules operate within a single notebook session. Multi-agent coordination on one session, and coordination across sessions, would need different rules and are left for future work. Second, the runtime checks each change as it happens, not afterward (cf. Tagliabue et al. [17]). Third, the agent's interface is Python, not a fixed list of operations. The same interface shape admits other rule vocabularies, like MVCC-style isolation for concurrent agents [16] or branch-and-merge for review-gated workflows [2]. These pursue different priorities than ours.

Artifact. The byproduct of an agent's session is a `.py` notebook that persists, can be reopened by a human, and re-runs in declared-dependency order. The agent leaves the same artifact a human would.

Affordances. The same action surface handles more than cell mutation; marimo's package installation runs through it as well. The transactional flush in P3 also admits pre-execution hooks (linting, type-checking, or other static checks) that could enforce invariants before code lands.

Sandboxing and safety. Hardening marimo pair against untrusted code is future work, but the design has one structural property that simplifies the problem. Every action the agent takes runs through Python in the kernel (P1), which makes the kernel the natural place to enforce policy. Where the kernel runs (local or remote) and how it is isolated (sandboxed or not) are deployment choices, separate from the agent interface. A policy layer at the kernel can inspect each proposed action before it executes and admit or reject it, extending P3 from correctness checks to authorization.

Limitations and future work. A proper evaluation of marimo pair is future work. The natural baselines are shell-only coding agents, Jupyter-based agent systems [20, 23], and curated notebook-AI tools. We expect the win to be largest on tasks where there are many directions to search and restarting is expensive. Liu et al. [9] make the same kind of move at the data-systems layer (cf. Zeighami et al. [21] on proactive data systems). Their *agentic memory store* sits across sessions; ours sits within a session. The two layers compose.

REFERENCES

[1] Cloudflare. 2025. Code Mode: the better way to use MCP. https://blog.cloudflare.com/code-mode/.

[2] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach to Weak Consistency. In *SIGMOD*. 1615–1628.

[3] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-Aided Language Models. In *ICML*. https://arxiv.org/abs/2211.10435.

[4] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering* 23, 2 (2021), 7–14. https://doi.org/10.1109/MCSE.2021.3059263.

[5] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *CHI*. 270:1–270:12. https://doi.org/10.1145/3290605.3300500.

[6] Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. DSBench: How Far Are Data Science Agents from Becoming Data Science Experts? arXiv:2409.07703; https://arxiv.org/abs/2409.07703.

[7] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks — a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87.

[8] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as Policies: Language Model Programs for Embodied Control. In *ICRA*. https://arxiv.org/abs/2209.07753.

[9] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2026. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. In *CIDR*. https://arxiv.org/abs/2509.00997.

[10] Dang Nguyen, Viet Dac Lai, Seunghyun Yoon, Ryan A. Rossi, Handong Zhao, Ruiyi Zhang, Puneet Mathur, Nedim Lipka, Yu Wang, Trung Bui, Franck Dernoncourt, and Tianyi Zhou. 2025. DynaSaur: Large Language Agents Beyond Predefined Actions. In *COLM*. https://arxiv.org/abs/2411.01747.

[11] Fernando Pérez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007), 21–29. https://doi.org/10.1109/MCSE.2007.53.

[12] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *MSR*.

[13] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *CHI*. 32:1–32:12. https://doi.org/10.1145/3173574.3173606.

[14] Matthew Russo and Tim Kraska. 2026. Deep Research is the New Analytics System: Towards Building the Runtime for AI-Driven Analytics. In *CIDR*. https://arxiv.org/abs/2509.02751.

[15] Charlie Summers, Haneen Mohammed, and Eugene Wu. 2026. Please Don't Kill My Vibe: Empowering Agents with Data Flow Control. In *CIDR*. https://arxiv.org/abs/2512.05374.

[16] Jacopo Tagliabue, Federico Bianchi, and Ciro Greco. 2025. Trustworthy AI in the Agentic Lakehouse: from Concurrency to Governance. arXiv:2511.16402; https://arxiv.org/abs/2511.16402.

[17] Jacopo Tagliabue and Ciro Greco. 2025. Safe, Untrusted, "Proof-Carrying" AI Agents: toward the agentic lakehouse. In *IEEE Big Data Workshop on Secure and Safe AI Agents for Big Data Infrastructures*. https://arxiv.org/abs/2510.09567.

[18] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *ICML*. https://arxiv.org/abs/2402.01030.

[19] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*. https://arxiv.org/abs/2210.03629.

[20] Ziming You, Yumiao Zhang, Dexuan Xu, Yiwei Lou, Yandong Yan, Wei Wang, Huaming Zhang, and Yu Huang. 2025. DatawiseAgent: A Notebook-Centric LLM Agent Framework for Adaptive and Robust Data Science Automation. In *EMNLP*. https://arxiv.org/abs/2503.07044.

[21] Sepanta Zeighami, Yiming Lin, Shreya Shankar, and Aditya G. Parameswaran. 2025. LLM-Powered Proactive Data Systems. *IEEE Data Engineering Bulletin* 49, 1 (2025). https://arxiv.org/abs/2502.13016.

[22] Alex L. Zhang, Tim Kraska, and Omar Khattab. 2025. Recursive Language Models. *arXiv preprint arXiv:2512.24601* (2025).

[23] Junjie Zhang, Jingyi Xi, Zhuoyang Song, Junyu Lu, Yuhua Ke, Ting Sun, Yukun Yang, Jiaxing Zhang, Songxin Zhang, and Zejian Xie. 2025. L0: Reinforcement Learning to Become General Agents. arXiv:2506.23667; https://arxiv.org/abs/2506.23667.