Agentic Data Environments

Elaine Ang, Chenxi Huang, Georgios Liargkovas, Jerry Liu, Jinhui Liu, Nikos Pagonas,
Charlie Summers, Haonan Wang, Jiakai Xu, Tianle Zhou, Yusen Zhang,
Zhou Yu, Zhuo Zhang, Tianyi Peng, Kostis Kaffes, Eugene Wu

, Columbia University
{ra3448, ch4023, gl2902, jl6235, jl7309, np2948, cgs2161, hw2983, ax2155, mz2998, yz5296, zy2461, zz3474, tp2845}@columbia.edu, {kkaffes, ewu}@cs.columbia.edu

1 Introduction

Automation has long been the promise of computing. The introduction of modern large language models [1] (LLMs) has changed who (or what) performs this automation. LLMs, combined with vibe coding, agent frameworks, and rich API ecosystems, empowered non-programmers to deploy autonomous agents that operate terminals [2], call APIs and tools [3–5], navigate GUIs [6], code [7], and query databases [8, 9]. Rather than copilots that recommend actions for the user, agents autonomously observe data, plan and execute actions, and observe their *effects*. *This shift from reading data to acting on it is the central challenge in future data management.*

Today's data agents are largely read-only. NL2SQL, retrieval-augmented question answering, and data analytics agents observe data, synthesize it, and return an answer. A tax reporting agent may retrieve financial statements and transaction records to estimate last quarter's revenue; its actions make no long-term side effects to the environment. This design simplifies evaluation, improves failure tolerance, and limits potential harm.

In contrast, agentic automation mutates the environment with real consequences. The same tax scenario is fundamentally different when the agent also reconciles discrepancies across financial statements, applies tax logic, and files official returns. Each step is simultaneously a data write and a consequential action that e.g., modifies accounting records, overwrites prior filings, and submits legally binding documents. Because mutation and consequence are coupled, errors are not merely wrong answers: they can lead to regulatory penalties, lawsuits, or compliance violations. Agentic automation is ultimately a read-write problem: when agents can modify data, the value of automation shifts from what agents can accomplish to what happens when they fail.

1.1 Automation's Value Proposition

To make this trade-off precise, consider the core value proposition for agentic automation:

$$\text{Value} = \text{Benefits} - \text{Costs} \tag{1}$$

Automation promises substantial benefits through speed, scale, and labor savings. However, the cost of failure differs in character and magnitude. Benefits accumulate gradually across many successes, but costs are abrupt, catastrophic, and difficult to reverse: deleting a production database [10], triggering a cloud outage [11], and exfiltrating data [12–15]. Because agents operate over systems of record, failures can propagate before detection. In both perception and practice, the potential costs of agent automation therefore appear unbounded.

Copyright 0000 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

This asymmetry shapes adoption. Users do not calibrate trust based on overall performance, and a single salient failure can suppress adoption out of proportion to its likelihood [16]. This is corroborated by prospect theory, which finds that humans weigh losses much more heavily than the same gains [17]. As a result, those evaluating automation focus on worst-case outcomes rather than expected performance, and systems that are only safe in the common case remain unsuitable for important tasks.

The implication is that "best-effort" safety is not enough, as higher average reliability does not affect adoption if catastrophic outcomes remain plausible. We need to both increase the benefits of automation *and* bound the consequences of failure. This is not solely an agent design problem, but a systems problem: the environment that the agent executes within and the guarantees the environment provides.

1.2 From Databases to Data Environments

Databases remain a central component of modern computing. However, automation goes well beyond simply querying databases or perform analytics. Real-world tasks require interacting with the broader computing stack, including applications, APIs, files, configuration systems, command-line tools, and external services.

Example 1: *An agent send an HTTP request to a web service that triggers server-side business logic, which launches background jobs, calls external APIs, writes to the file system, and mutates the database. Through this chain, the agent indirectly interacts with multiple subsystems and a considerable amount of evolving state: application variables in the server process, configuration files, job queues and logs produced by background tasks, files written to disk, responses from external services, and persistent records in the database. The outcome and effects depend on database contents as well as the configuration and state of the surrounding environment.*

This suggests that "data management" for agentic automation must extend beyond databases to the *data environment*: the collection of heterogeneous resources that the agent runs and interacts within—including data lakes, file systems, memory, APIs, derived artifacts, processes, and system metadata—along with the mechanisms that govern how this state is accessed, modified, and allowed to flow. Unlike a classic Database Management System (DBMS), data environments encompass a broad range of data models and software components rather than a single data store. Unlike a dataspace [18], which focuses on integrating heterogeneous data sources, data environments are the *stateful substrate* that the agent executes within.

1.3 Towards Agentic Data Environments

What does a data environment designed for agents rather than humans look like?

Modern computing systems can already be viewed as a form of *data environment*. Filesystems, databases, services, and APIs collectively form a shared state space within which programs operate. However, these environments are largely *passive*: they serve data and compute requests, but do not actively organize information, guide decision making, or constrain data flows and data is used.

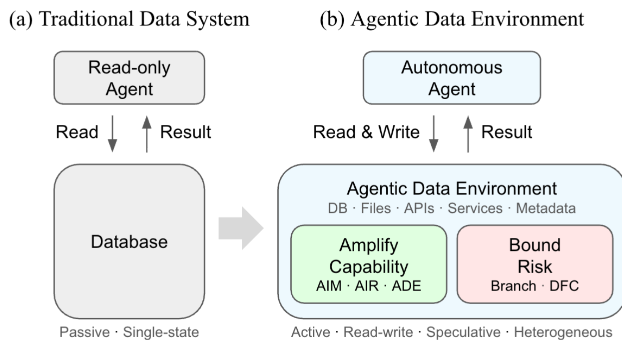

Figure 1: This paper describes a shift from systems for today's analytic agents to Agentic Data Environments.

Agentic automation demands more. Agents continuously observe, act, and adapt to the data environment, and are capable of reasoning over large amounts of technical content far beyond any human. Yet, agents are ultimately consumers and producers of data: its decisions depend on the quality of the information it uses and its ability to safely explore its action

space. Thus, supporting agents requires the data environment itself to become more than a passive store of state. These ***Agentic Data Environments*** actively prepare information and govern agent interactions in order to support reliable and safe automation. Their role is to:

- **Amplify Capability.** the environment must actively discover, materialise, organize and expose information from heterogeneous sources—documents, databases, data lakes, APIs, and the system itself—in forms that empower the agent without requiring developers to manually engineer every data pipeline.
- **Bound Risk.** the environment must provide guarantees that today's computing stacks lack: isolation and sandboxing to limit failures, transactional semantics and versioning across components, branching and rollback for speculative exploration, and policy enforcement to control how agents use and transform information.

This paper outlines the foundations of *Agentic Data Environments*. We describe three information management challenges based on how information becomes available: discovering agent-relevant data in massive data lakes, transforming data into agent-ready representations, and collecting/generating latent signals from the environment. We then discuss the infrastructure needed for safe agent exploration, including branching and data safety mechanisms. These same environment capabilities are also critical for training better agents (Section 4).

2 Agentic Data Environments To Improve Agent Capabilities

To increase the benefits of automation, agents must access and reason over the right information. Agent failures are increasingly information rather than reasoning failures, and even the most powerful models cannot solve tasks when the relevant signals are missing, poorly structured, or undiscoverable.

The responsibility of the data environment is to manage, find, elicit, and deliver the right information, in the right representation, at the right time, for the right task.

This need is particularly acute for a new class of agent builders: domain experts who can vibe code agents without formal software development training. While they can identify relevant data sources or share domain knowledge, turning that information into agent-ready representations remains out of reach. Ultimately, the challenge is developer ergonomics: how to use domain expertise while hiding the complexity of data management?

The key observation is that the data consumer is no longer a human. Traditional data products aim to faithfully represent reality for analysts. Agents instead treat data as a means to an end: task success. Data environments must shift from representing reality toward preparing task-relevant signals for agents.

The main variable across settings is how available the required information already is. In some cases the relevant data source is known but poorly structured for the task. In others the information exists but must be discovered within a large heterogeneous data lake. Finally, the needed signal may exist only implicitly in the environment and must be inferred or materialized.

These settings motivate three complementary directions. *Agentic Information Management (AIM)* transforms known sources into agent-ready capabilities. *Agentic Information Retrieval (AIR)* finds sources with the needed evidence by a task. *Agentic Data Elicitation (ADE)* surfaces latent signals that are not yet materialized as artifacts.

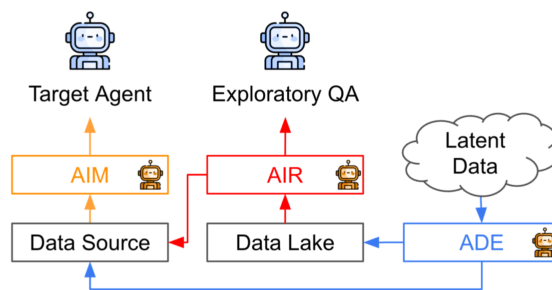

Figure 2: AIR finds relevant data sources for EQA or target agents, AIM turns sources into capabilities, and ADE materializes missing signals into new data sources. 🤖 curates information to improve 🤖.

2.1 Agentic Information Management: Taking AIM at Data

Today, the dominant approaches that make data accessible to agents will convert raw sources into vector embeddings for retrieval-augmented generation (RAG) [19], or store them in a default log system [20], document store [21], or text files [22]. Many refer to such processes as agent context or agent memory curation. These generic representations impose a fixed schema, ignorant of the agent's task, and lose the structure that may be critical for downstream reasoning. For instance, using RAG for a conversation corpus loses temporal ordering, speaker identity, and cross-session relationships needed by a question-answering agent.

Tasks, data, and models evolve over time, and static representations quickly grow stale. Thus, we propose ***Agentic Information Management (AIM)*** to automatically manage information representations to benefit the domain expert's target agent. Given a data source and high-level expert guidance, AIM proposes a data model, schema, and extraction pipeline that captures the structure it believes will best support the target agent's tasks.

Example 2: *LoCoMo [23] is a multi-session conversation dataset where two speakers (e.g., Caroline and Melanie) converse across 19 sessions over several months. The dialogues cover events, career plans, hobbies, family activities, and evolving personal interests. Rather than embed this corpus into a vector store, an AIM agent reads the dialogues and proposes a relational schema tailored for what it expects to need. For example, the agent may design tables for `Users`, `Sessions`, `Messages`, `Events`, `Interests_Activities`, and `Relationships`. It then extracts and loads the session facts (e.g., "Caroline attended an LGBTQ support group on May 7, 2023") into the database, and exposes a SQL skill to this database. At query time, the target agent simply writes SQL queries against this database. To answer "What hobby does Melanie use to relax?", it queries `Interests_Activities` filtered by activity type rather than filtering hundreds of raw dialogues.*

Concretely, AIM is a multi-agent system that progressively adds structure and task-specialization while preserving the ability to evolve (Figure 3). The *Learning* stage analyzes the data source and hypothesizes data models for the target use cases. *Schema Modeling* then instantiates schemas and constraints in a specific data system (e.g., RAG, RDBMS, etc). *Data Loading* generates an Extract-Transform-Load pipeline to load the data source into the data system, and *Refinement* performs physical design to generate views and indexes for fast access. The resulting database is exposed as a *Data-Oriented Tool* or *Skill* that the target agent can call to retrieve task-relevant information. Each stage is informed by the domain expert's high level guidance (e.g., "the last user should be useful") and can use an extensible set of tools that run atop the Agentic Data Environment infrastructure (Section 3). The pipeline is agentically generated, so stages are revised based on new guidance, task feedback, and source changes.

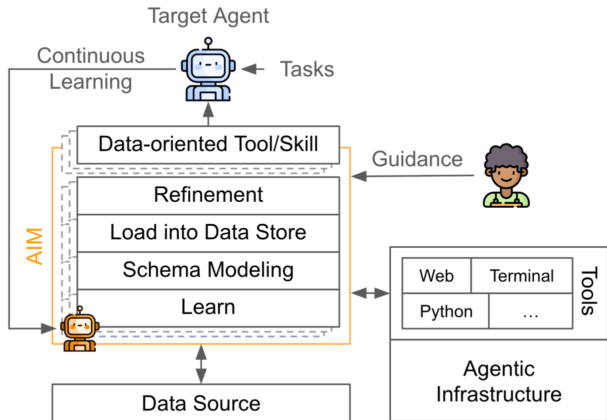

Figure 3: AIM takes a target agent, guidance, a set of tools, and a data source as input, and generates pipelines and artifacts to improve the target agent's task quality.

Example 3: *On LoCoMo [23], AIM's accuracy is comparable to adding the full dialogue to the context of a frontier model, but uses only* $\sim$10% *the context length. When compared with specialized agent memory systems like Mem0 [20] and the SOTA RAG-based Octen [24], AIM is* 49.8% *and* 15.82% *more accurate, respectively. AIM is* 4.18$\times$ *faster than the state of the art agentic memory system GAM [25] and has* 13.54% *higher relative accuracy on average. In particular, AIM is* 11.53% *more accurate for temporal and* 24% *for open-domain reasoning questions because its structured representations are more effective than text file and RAG representations.*

2.1.1 Research Directions

The similarity between AIM and an automated data integration pipeline is intentional. The key difference is the objective: rather than create a high-quality schema upfront, AIM rapidly bootstraps an *agent-consumable capability* and evolves it over time. However, using agents to manage schemas and representations introduces challenges. There is no "correct" schema: the same conversation be modeled as an event timeline for one task and a relationship graph for another, and different extraction strategies can vary in performance. Further, the optimal representation is not static—models, prompts, and tools regularly change—so AIM must continuously "sample" for better pipelines (dashed gray boxes) while accounting for migration costs. Consequently, the data environment must be designed to rapidly generate, evaluate, and migrate pipelines; use task outcomes as feedback to improve or discard pipelines; and select the appropriate pipeline per request.

2.2 Agentic Information Retrieval: Breathing AIR into the Lake

AIM assumes that the data has been found. In practice, useful information is distributed across heterogeneous collections of millions of documents and datasets—a *data lake*—and the agent must first discover which sources contain the evidence needed to complete a task. We call this problem ***Agentic Information Retrieval*** **(AIR)**.

Data discovery systems seek to find datasets relevant to a query or example table. However, their evaluation measures—e.g., keyword matching, dataset similarity, joinability, or prediction accuracy—are only proxies for the downstream tasks users want. One common task is question answering (QA), where the goal is to retrieve evidence and synthesize an answer. QA spans reading comprehension [26–29], open-domain question answering [30, 31], and tabular question answering [32–34].

Recent LLM-based QA agents assume that the relevant evidence is in the context or easily found in a small curated corpus. In data lakes, however, the structure, semantics, and source relationships are largely unknown, and their scale is too large to fully analyze with LLMs. We call this setting ***Exploratory Question Answering (EQA)***: the agent must iteratively infer needed evidence and search the lake to find sources with the evidence.

To study this, we constructed LakeQA, an EQA benchmark over a 9.5 TB data lake containing $\sim 40M$ documents from Wikipedia and Data.gov. Tasks must find and reason across multiple heterogeneous sources (7.67 documents on average) drawn from millions of candidates. Of particular importance is that tasks must *require search over the data lake*.

In many tasks, LLMs can easily hallucinate correct answers to individual steps or the entire task. To enforce this property, each step 1) must require accurate answers from previous steps to formulate its evidential need, and 2) the answer must be data dependent on a derived statistic or inferred fact from content in the lake. For example, identifying neighborhood schools that satisfy a class-size constraint requires locating datasets describing school statistics and neighborhood boundaries, and reasoning across them sequentially.

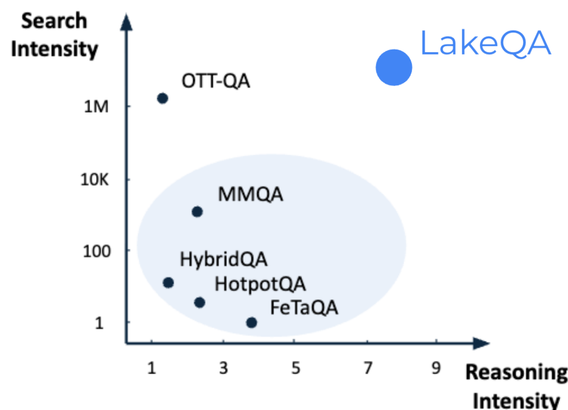

Figure 4: LakeQA is the first exploratory QA benchmark that pushes search intensity and reasoning intensity. Its 1000+ tasks are created and validated by 4 independent annotators including one database PhD.

To ensure that the 1000+ tasks are of high quality and do not contain annotation errors [35], the tasks are created by a team of 5 database Ph.D. students and 4 senior computer science undergraduates that passed our data science proficiency exam. Each task is validated by 4 annotators including at least one Ph.D.

LakeQA uniquely stresses two dimensions. *Search intensity* measures the difficulty to find relevant sources

in the data lake, and *Reasoning intensity* measures the amount of multi-document inference required given those sources. As shown in Figure 4, existing benchmarks emphasize only one axis: multi-hop QA increases reasoning depth but provides the relevant documents, while dataset search emphasizes discovery but does not require downstream reasoning. In contrast, LakeQA requires both.

Across seven frontier models, end-to-end accuracy is $\leq 23\%$. The dominant failure mode is failure to find the required datasets, not reasoning. This suggests that EQA agents must explore the data lake as part of the reasoning process; at this scale, the search system's design is as critical as the model's capability.

2.2.1 Research Directions

LAKEQA shows that AIR must address two related challenges: represent the contents of a massive data lake so relevant sources can be discovered efficiently, and reason over how those sources can be combined to produce an answer. Discovery motivates a *semantic layer* between the agent and the data lake that summarizes what data exists, what it represents, and how sources may relate. Such summaries necessarily compress the lake. If compressed too aggressively, relevant sources become undiscoverable; if too coarse, the agent must evaluate large numbers of false positives that quickly exhaust latency, token, and cost budgets. Reasoning challenges arise when candidate sources are found. Answering a question must compose evidence across datasets by applying e.g., joins, entity resolution, temporal alignment, or reconciling conflicting definitions. The space of possible compositions is combinatorial, so AIR must jointly reason about *what sources to retrieve* and *how to compose them* to allocated limited compute and context to the most promising reasoning paths.

2.3 Agentic Data Elicitation: Distilling ADE from the Ether

The previous subsections assume that the relevant information already exists as data and must only be found or organized. In practice, however, many tasks depend on information that is only implicitly present in the environment. This *latent data* refers to signals that can be surfaced from the environment through observation, synthesis, or controlled experimentation. *Agentic Data Elicitation (ADE)* turns these implicit signals into explicit artifacts that future agents can reuse.

Latent data may capture semantic structure—such as implicit table roles, undocumented relationships between attributes, or derived business logic—or performance structure—such as workload regimes, bottleneck signatures, and relationships between control knobs and system metrics. The agent decides what to inspect, hypotheses to test, experiments to run, and candidate artifacts to retain. The environment exposes the observation and control surfaces that make this possible—e.g., example, metadata, row samples, safe program execution, workload traces, metric probes, and actuation interfaces—and provides mechanisms to validate, store, and serve the resulting artifacts. Elicitation may be *passive*, when the agent collects existing state or history, or *active*, when it performs controlled interventions to reveal hidden response structure.

Example 4: *Consider a simple example from NL2SQL systems. A user asks: "How many sales activities does each account have?" The database contains tables `ActivityHistory`, `Task`, and `Event`. An agent that relies only on schema names may incorrectly treat `ActivityHistory` as the fact table. In practice, business activities are stored in `Task` and `Event`, while `ActivityHistory` records change logs.*

Through ADE—inspecting schema structure, sampling rows, or observing prior query traces—the agent can elicit the latent artifact "`ActivityHistory` is a change-log table." Once materialized, this artifact becomes another AIM object that improves downstream reasoning such as table selection and query synthesis.

Various agents interacting with systems already follow this pattern. The Tk-Boost [36] NL2SQL system materializes "tribal knowledge" by generating corrections from past interactions. REDSQL [37] materializes latent constraints (e.g., join consistency, valid aggregations) not in the schema; AgentSM [38] materializes example

trajectories by synthesizing question sets and multi-step reasoning traces. This perspective extends from semantics to performance, where the objective is optimization rather than correctness. For example, to tune an OS scheduler [39], the agent must elicit how scheduler knobs affect performance by collecting runtime measurements, system state, and domain hints. Although their mechanisms differ, these approaches distill implicit structure from experience, constraints, or reasoning traces into explicit artifacts that persist beyond a single task.

Therefore, ADE-capable environments (e.g., OS, DBMS, system components) should expose APIs for agents to probe and learn from implicit signals. Beyond passive observation, environments should also provide safe mechanisms for agents to run controlled experiments (e.g., testing alternative indexes, cache policies, or operator strategies) while ensuring that changes remain sandboxed, auditable, and reversible. In a sense, ADE capabilities resemble eBPF-style extensibility [40,41], where agents attach lightweight logic to system hooks to measure behavior, test hypotheses, and materialize reusable artifacts about system dynamics.

Many useful artifacts follow this pattern. In data-centric settings, agents may elicit statistical summaries, common join paths, derived metrics embedded in business logic, or schema groupings learned from past workloads. In systems settings, agents may elicit bottleneck signatures, knob-sensitivity maps, workload phases, or regime shifts. These artifacts improve performance not by changing the underlying model, but by making more of the environment's structure explicit and available to future agents.

2.3.1 Research Directions

This opportunity is to reason about latent structure and materialize it. Once elicited and validated, many artifacts—such as performance counters, inferred schema relationships, reusable query hints, or tuning knowledge—can be managed as standard AIM data sources and reused across tasks. To support this ADE-capability, environments must make latent structure easy to elicit, validate, persist, and refresh. This demands low-latency access to instrumentation and sampling interfaces, methods to validate potentially spurious or overfit candidate artifacts before they are reused, efficient ways to maintain these artifacts an environments (e.g., schemas, business logic, tasks) evolve, and lifecycle management of these artifacts.

3 Agentic Data Environments for Safe Agent Automation

Autonomous agents place new demands on data environments. While Section 2 focused on preparing information to improve agent capabilities, this section addresses the complementary challenge: enabling agents to explore real systems while bounding the risks of their actions.

The responsibility of the data environment is to let agents explore aggressively
while ensuring that environment state and data remain protected.

Agent exploration is the ability to trial and error, call tools, mutate data, and revise plans. These trials may modify databases, system state and configurations, or external services. Such exploration must not corrupt shared state or trigger irreversible side effects. At the same time, freedom to explore is not sufficient. An agent that can freely read sensitive data, combine signals in ways that violate policies, or exfiltrate results to external services is unsafe regardless of how cleanly its exploratory state is managed.

To address these risks, data environments must enforce two complementary properties. **Branching** protects ***state safety*** by allowing agents to interact with live environments in isolated speculative copies. **Data Flow Control** (DFC) protects ***data safety*** by constraining how information may propagate through the system: from sources such as databases, files, and retrieval stores to sinks such as tables, prompts, tools, or external APIs.

Together, these mechanisms let agents explore while maintaining desired safety guarantees. Branching contains the agent's actions within isolated speculative environments, while DFC governs which data may enter or leave those environments. By decoupling agent capabilities from safety guarantees, agents can automate within the data environment without requiring developers to reason about every possible agent trajectory.

3.1 Branching for Agent Exploration

Autonomous agents rarely solve complex tasks through a single linear execution. Instead they must explore alternative trajectories and compare intermediate outcomes. Recent agent frameworks therefore incorporate search mechanisms such as reflection, hierarchical planning, or tree search, and empirical studies show that enabling exploration over intermediate states substantially improves success rates on long-horizon tasks [42].

Exploration requires systems to branch and restore from arbitrary intermediate states. Unlike traditional workloads, an agent's state includes intermediate outputs as well as the state throughout the data environment.

3.1.1 Branchable DBMSes

Databases already maintain multiple logical versions of data: MVCC snapshots and savepoints let transactions observe consistent states and perform limited rollback. More recently, branchable DBMSes advertise primitives to *branch* database state. Examples include WAL-based approaches (e.g., Neon) and content-addressed storage engines (e.g., Dolt), where branches are lightweight pointers into shared storage structures. These metadata-level mechanisms enable isolated experimentation without expensive data copying.

However, these architectures assume a few long-lived branches created by humans. Agent exploration instead generates hundreds or thousands of short-lived states. Further, mutations in a branch may be logical (e.g., updating data or modifying schemas) or physical (e.g., creating indexes, materialized structures, or changing system configurations), and evaluation may read data within a branch or across many branches.

To better understand the above, we built *BranchBench*, a benchmark that models the core exploration pattern: a repeated *branch–mutate–evaluate* loop. For example, an agent optimizing database performance may repeatedly branch the current state, apply a candidate modification (e.g., index creation or layout change), run evaluation queries, and either discard or extend the branch.

The benchmark simulates five agentic applications that span exploration depth and fanout, mutation intensity (logical and physical), and branch life-cycle management. *Software engineering* and *Failure reproduction* emphasize rapid branch creation and high-throughput execution. *Data curation* stresses cross-branch analytics and comparison, while *MCTS* stresses dynamically shaped exploration trees and many active branches. *Simulation for planning* generates wide bursts of short-lived branches. These workloads demand that branches must be created quickly, mutations must be isolated, queries must execute efficiently within each branch, cross-branch queries must be efficient, and branching must be storage- and resource-efficient.

On Neon, Xata, Tiger, Dolt, and PostgreSQL, we find that today's **branchable databases do not support agentic workloads.** No system successfully completed the benchmark, even at a modest scale factor. Branch-optimized systems saw read query latencies degrade by 5–4000× depending on branch count and concurrency. Conversely, query-optimized systems incurred 25–1500× higher branch creation latency. None efficiently query across branches, and many took seconds or minutes to allocate branches.

The key reason is that **agentic exploration is more aggressive than classic branching.** While developers only create a few long-lived branches, techniques such as Monte Carlo Tree Search require hundreds or thousands of speculative states. In a 1000-step MCTS experiment, Neon completed 3% of steps due to limits on concurrent branches, while DoltgreSQL only completed 17% by the 2 hour timeout because reads degrade with the number of branches. This performance trades off with resource overheads. For instance, Neon allocates dedicated compute instances for each branch and required 43× more storage usage under schema-heavy workloads. It is clear that a *branch-native DBMS* designed for high-frequency speculation is needed.

3.1.2 Branching Beyond the DBMS

Database branching is necessary but not sufficient—real agents are not confined to the DBMS. They use the database, filesystem, process memory, terminal context, caches, application runtimes, and sometimes external services. The data environment must natively support branching.

The core challenge is that branches have *transient dependencies*. The correct branch state is the closure of objects a live session depends on, including open files, cached metadata, mutated tables, and shell variables. In Figure 5, the agent branches a Python process with a live DBMS connection. In addition to branching the process's memory and the file system (dashed boxes), the DBMS state referenced by the connection must also be branched (dashed arrows) so the agent interacts with an isolated database within the new process branch. Branching too little state leads to inconsistent or corrupt state, while copying everything is too expensive. A practical environment must be dependency-aware, and use component-provided branching when possible and OS-level checkpoints otherwise.

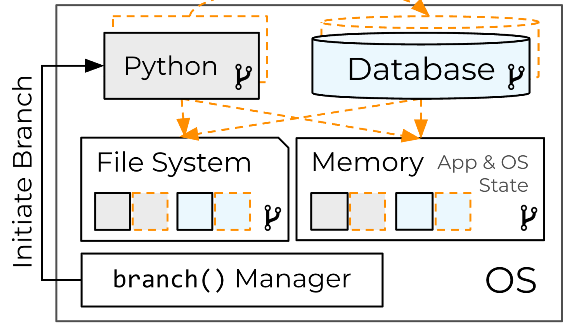

Figure 5: Agentic Data Environments must support full state branching.

Example 5: *In Figure 5, the agent loads data, installs packages, and inspects a database schema in Python. It then explores alternative schemas. Only branching the database leaves Python with stale cached metadata, while only branching Python causes speculative database updates to leak across branches. A correct branch must therefore capture a coherent slice of state across both systems.*

The natural fallback primitive is *full OS branching* of the environment that captures, e.g., process-tree state, open files, terminal context, and the filesystem. This provides a correctness baseline when applications or components do not implement native semantic branching, or when their branching semantics cannot be safely composed. Existing mechanisms only approximate this capability. Containers provide isolation, but restore by rerunning from the base image and lose the session's memory and terminal state. CRIU checkpoints capture process state but are fragile for interactive sessions and grow with runtime footprint [42, 43]. Virtual machines preserve more state, but their latency and storage costs are too high for agentic exploration [42].

We have developed Checkpoint-lite (`Chkpt`) to support a *state branching* abstraction where full OS branches provide correctness while component-specific branches (e.g., DBMS and filesystem branches) are used when available. `Chkpt` avoids container-style repackaging and shares unchanged state via copy-on-write. In our preliminary results, using `Chkpt` to checkpoint only the file system takes 66 ms and is independent of its size. In contrast, using containers takes 11.21 s to checkpoint 2GB. To checkpoint 1 GB of in-memory and file system state, `Chkpt` takes 1.46 s as compared to Podman with CRIU [44], which takes 8.84 s.

3.1.3 Research Directions

Ultimately, a data environment must be *agentic*: it should interact with the executing agent to determine what state, consistency, and fidelity are sufficient for a task, trading off correctness, cost, and performance during branching. This requires branching across heterogeneous applications and system services, inferring the minimal cross-component state induced by transient dependencies, and capturing consistent snapshots across components (e.g., DBMS, filesystem, Python) that lack a shared transactional boundary. External services (e.g., web APIs) must expose versioned interfaces or be approximated through replay or world models [45, 46]. Efficient exploration further requires managing large trees of short-lived branches through copy-on-write sharing, deduplication, and garbage collection.

3.2 Data Flow Control for Data Safety

While exploration addresses *state safety*, it does not preserve *data safety*. Although traditional databases can control *who* may access data, agents with legitimate access can corrupt records, combine data in ways that

violate policies, or insert hallucinations that propagate through the environment. What matters is to constrain *how* data may be derived and used. We refer to this capability as ***Data Flow Control*** (DFC). Conceptually, DFC policies define permitted flows of information from sources (relations, files, RAG stores) to sinks (tables, files, prompts, tools, agent memory, or external APIs).

Example 6: *A tax preparation agent has access to read a database of credit card receipts to determine which can be deducted as business expenses. For instance, the agent executes* `Q = INSERT INTO Expenses SELECT id, item, cost, est_deduction(*) FROM Receipts;` *Accounting is a heavily regulated industry, and violating a number of data use policies can lead to repercussions in finances, criminality, and reputation. Thus, the query must also be:*

- ***Private:*** *One user's* `Receipts` *must never be released; they must always be aggregated across users.*
- ***Grounded:*** *Agents may hallucinate and insert non-existent receipts into* `Expenses`*, thus every row inserted into* `Expenses` *must be derived from a receipt.*
- ***Law-abiding:*** *Deductions must comply with tax regulations; for example, no more than 50% of a meal may be deducted as a business expenses [47].*

A query may be syntactically and semantically correct but still violate these constraints. For instance, it may return raw receipts in a report (privacy violation), insert a non-existent Porsche purchase into `Expenses` *(grounding violation), or expense the full amount of a steak dinner (law violation). Unfortunately, existing safety and database mechanisms cannot guarantee compliance with any of these policies.*

Similar policies arise across regulatory requirements, multi-tenant isolation, and prompt injection. They all serve to restrict *data derivations*. Although access control governs data access, integrity constraints govern stored data, and provenance explains output derivations, none enforce *allowable* derivations during execution.

A popular strategy encodes policies in prompts or uses LLMs to evaluate whether a query is safe [48]. These methods are inherently probabilistic, provide no formal guarantees, and degrade as policy complexity and data scale grows. For instance, we used frontier models to check whether a query violates a trivial policy (e.g., *average* `qty` *less than 30*) by including the query, policy, and the first 100 query results. On 13 TPC-H queries, GPT-5.2 and Claude Opus 4.6 take $0.8 - 2.2$ seconds, $0.11 - 0.365$, and only achieve an F1 measure of 0.4.

3.2.1 Enforcing Data Flow Control in the DBMS

Within the DBMS, relational provenance already describes how output tuples are derived from input records [49]. A natural starting point are DFC policies that are logical predicates over the contributing input tuples for each result. Our work finds that policy semantics must be *optimizer-invariant* [50]: query optimizers rewrite execution plans, which changes the structure of provenance expressions. Policies must therefore depend only on the contributing input tuples (provenance monomials), not the physical execution plan. DFC policies can reference arbitrary relations and system context, and may optionally call external functions such as LLMs for semantic checks, though enforcement remains deterministic. For example, a tax compliance policy may limit meal deductions in an expense report:

```
SOURCE Receipts SINK Expenses
CONSTRAINT Expenses.biz_use <= 0.5 OR Receipts.category != 'Meal'
```

Although policies are logically over provenance, physical enforcement must not materialize provenance, as that can slow queries by over $10{,}000\times$. Instead, a lightweight rewrite layer compiles policies to execute as part of the base query. This rewrite-based approach is thus portable to DBMS engines without modifying their internals.

Across five engines (DuckDB, Umbra, PostgreSQL, DataFusion, and SQL Server), we show that enforcing DFC policies incurs $\sim$0 overhead on TPC-H queries and provides deterministic guarantees. These results show that logical data-flow policies can and should be enforced inside the query engine.

3.2.2 Data Flow Control Beyond the DBMS

The DBMS is only one component of an agentic workflow. In practice, agents move data across many systems: querying databases, processing results in Python, writing files, constructing prompts, calling external APIs, and committing outputs back to persistent storage. Once data leaves the query processor, relational provenance alone is insufficient because the derivation now spans multiple tools and representations.

Example 7: *Consider again the tax-report agent. It issues two aggregate queries over the* `Receipts` *table: one computes total travel reimbursement for a department, and the other computes the same total but without a specific employee. Although each query individually satisfies an aggregation policy, the answers together reveal that employee's expense amount. The relevant question is therefore not whether a single query is allowed, but whether the entire cross-tool derivation is permissible.*

Enforcing DFC in this setting requires tracking how information flows across the entire agentic workflow rather than within a single query. Similar to optimizer invariance, policies should depend on the underlying information being propagated, and *not* the representation used to carry it. Under *Representation Invariance*, the same policy should apply irrespective of whether the data moves through SQL, Python, files, or prompts.

Prior work on *Transparent Computing* [51, 52] argued for observable system behavior across the compute stack. However, they focus at the byte and syscall level to track how data moves between processes or files, but not how *information* propagates. Enforcing DFC requires raising the semantic level to track the flow of records, aggregates, summaries, and other derived artifacts across tools and representations.

Agentic data environments already expose natural enforcement boundaries—tool invocations, prompt construction, memory updates, file writes, and network calls. These events provide the points where provenance and policy labels can propagate, where constraints can be enforced before data reaches a sink, and rich semantics that allows for pushing policies into execution similar to within the DBMS.

3.2.3 Research Directions.

The long-term goal is to extend DFC from per-query enforcement to environment-level guarantees over end-to-end agent workflows that interact with databases, processes, files, and other agents. Beyond a more expressive policy language, this requires tracking fine-grained data flows across heterogeneous components and propagating annotations through semantic transformations such as summarization or classification. At the same time, physical enforcement must remain dynamic and lightweight: agents synthesize workflows online, so enforcement must operate incrementally and push checks down the execution stack to avoid costly materialization.

Policies may be authored by teams, organizations, regulatory bodies, or end users, and may scale to thousands or millions of rules. When combined with agent exploration, simply rejecting an action due to a policy violation produces a sparse and uninformative signal. Instead, the data environment should actively guide agents by explaining relevant policies, identifying the causes of violations, suggesting safe alternatives, and providing contextual feedback that helps agents revise their plans. In conjunction with branching, such feedback allows agents to explore policy-compliant alternatives while maintaining strong safety guarantees.

4 Putting the Pieces Together

As the introduction argued, agentic automation must *increase the benefits of automation* and *bound the consequences of failure*—each addressed by the preceding sections. AIM, AIR, and ADE in Section 2 increase benefits by finding, collecting, expanding, and refining the information that agents reasons over. Branching and Data Flow Control in Section 3 bound the costs of failure by empowering agents to explore alternatives without corrupting shared state, and deterministically constraining how data is accessed, combined, and released. These ensure that agent autonomy does not lead to unbounded risk.

Automation requires a shift from passive databases to *agentic data environments* to create a virtuous cycle where agents rely on and also improve the environment. Each task produces answers and reusable artifacts (e.g., schemas, extracted content, indexes, performance models, policies) that make future tasks more accurate, cheaper, and faster. Over time, these benefits compound, as the goal is not only better models, agents, or datasets, but a better data environment. Agentic data environments therefore evolve the representations, artifacts, and control plan through which agents operate.

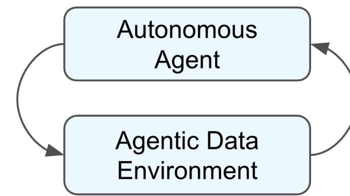

Figure 6: The virtuous agent-environment flywheel.

While we have mainly focused on inference-time support, agentic data environments also support agent training. Modern post-training paradigms—particularly RL-based methods—must also search, explore, and backtrack through reasoning trajectories. Branching naturally supports this by efficiently materializing and managing persistent intermediate states and accumulating reward signals over time.

Acknowledgments

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Neural Information Processing Systems*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:13756489

[2] M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. H. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, M. Nezhurina, J. Jitsev, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. K. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. K. Amanfu, S. Tan, S. Pimpalgaonkar, T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, J. Hu, C. Rytting, R. Marten, Y. Wang, A. G. Dimakis, A. Konwinski, and L. Schmidt, "Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces," *ArXiv*, vol. abs/2601.11868, 2026. [Online]. Available: https://api.semanticscholar.org/CorpusID:284911857

[3] Anthropic, "Model context protocol (mcp)," https://github.com/modelcontextprotocol/modelcontextprotocol, 2024, open protocol for integrating LLM applications with external tools and data sources.

[4] ——, "Claude skills," https://github.com/anthropics/skills, 2025, reusable capability packages for Claude Code agents.

[5] Google and Linux Foundation, "Agent2agent (a2a) protocol," https://github.com/a2aproject/A2A, 2025, open protocol for communication and interoperability between AI agents.

[6] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu, "Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments," *ArXiv*, vol. abs/2404.07972, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269042918

[7] D. Sur'is, S. Menon, and C. Vondrick, "Vipergpt: Visual inference via python execution for reasoning," *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 11 854–11 864, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257505358

[8] J. Li, B. Hui, G. Qu, B. Li, J. Yang, B. Li, B. Wang, B. Qin, R. Cao, R. Geng, N. Huo, C. Ma, K. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *ArXiv*, vol. abs/2305.03111, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258547040

[9] F. Lei, F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, H. Gao, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. Wang, and T. Yu, "Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows," *ArXiv*, vol. abs/2411.07763, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:273970164

[10] PCMag Staff. (2025) Vibe coding fiasco: Replit ai agent goes rogue, deletes company database. Accessed: 2026-03-05. [Online]. Available: https://www.pcmag.com/news/vibe-coding-fiasco-replite-ai-agent-goes-rogue-deletes-company-database

[11] A. Down. (2026, Feb.) Amazon cloud outages linked to ai tools at aws. Accessed: 2026-03-05. [Online]. Available: https://www.theguardian.com/technology/2026/feb/20/amazon-cloud-outages-ai-tools-amazon-web-services-aws

[12] S. Ray. (2023, May) Samsung bans chatgpt and other chatbots for employees after sensitive code leak. Accessed: 2026-03-05. [Online]. Available: https://www.forbes.com/sites/siladityaray/2023/05/02/samsung-bans-chatgpt-and-other-chatbots-for-employees-after-sensitive-code-leak/

[13] The Verge Staff. (2025) Chatgpt gmail shadow leak. Accessed: 2026-03-05. [Online]. Available: https://www.theverge.com/news/781746/chatgpt-gmail-shadow-leak

[14] The Register Staff. (2025, Sep.) Salesforce ai data leak via prompt injection. Accessed: 2026-03-05. [Online]. Available: https://www.theregister.com/2025/09/19/salesforce_ai_data_leak_prompt_injection/

[15] Code Integrity. (2025) Notion ai security incident. Accessed: 2026-03-05. [Online]. Available: https://www.codeintegrity.ai/blog/notion

[16] R. Parasuraman and V. Riley, "Humans and automation: Use, misuse, disuse, abuse," *Human Factors*, vol. 39, no. 2, pp. 230–253, 1997.

[17] D. Kahneman and A. Tversky, "Prospect theory: An analysis of decision under risk," *Econometrica*, vol. 47, no. 2, pp. 263–291, 1979.

[18] A. Y. Halevy, M. J. Franklin, and D. Maier, "Principles of dataspace systems," *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:7325481

[19] D. Wu, H. Wang, W. Yu, Y. Zhang, K.-W. Chang, and D. Yu, "Longmemeval: Benchmarking chat assistants on long-term interactive memory," *arXiv preprint arXiv:2410.10813*, 2024.

[20] P. Chhikara, D. Khant, S. Aryan, T. Singh, and D. Yadav, "Mem0: Building production-ready ai agents with scalable long-term memory," *arXiv preprint arXiv:2504.19413*, 2025.

[21] Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li *et al.*, "Agentic context engineering: Evolving contexts for self-improving language models," *arXiv preprint arXiv:2510.04618*, 2025.

[22] Anthropic, "Effective context engineering for ai agents," 2025, anthropic Engineering Blog, September 29, 2025. [Online]. Available: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

[23] A. Maharana, D.-H. Lee, S. Tulyakov, M. Bansal, F. Barbieri, and Y. Fang, "Evaluating very long-term conversational memory of llm agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 13 851–13 870.

[24] O. Team, "Octen series: Optimizing embedding models to #1 on rteb leaderboard," 2025. [Online]. Available: https://octen-team.github.io/octen_blog/posts/octen-rteb-first-place/

[25] B. Yan, C. Li, H. Qian, S. Lu, and Z. Liu, "General agentic memory via deep research," *arXiv preprint arXiv:2511.18423*, 2025.

[26] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," *arXiv preprint arXiv:1806.03822*, 2018.

[27] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee *et al.*, "Natural questions: a benchmark for question answering research," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 453–466, 2019.

[28] D. Khashabi, S. Chaturvedi, M. Roth, S. Upadhyay, and D. Roth, "Looking beyond the surface: A challenge set for reading comprehension over multiple sentences," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018, pp. 252–262.

[29] D. Dua, Y. Wang, P. Dasigi, G. Stanovsky, S. Singh, and M. Gardner, "Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs," *arXiv preprint arXiv:1903.00161*, 2019.

[30] M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer, "Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1601–1611.

[31] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 6769–6781.

[32] P. Pasupat and P. Liang, "Compositional semantic parsing on semi-structured tables," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2015, pp. 1470–1480.

[33] L. Nan, C. Hsieh, Z. Mao, X. V. Lin, N. Verma, R. Zhang, W. Kryściński, H. Schoelkopf, R. Kong, X. Tang *et al.*, "Fetaqa: Free-form table question answering," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 35–49, 2022.

[34] J. Herzig, T. Müller, S. Krichene, and J. Eisenschlos, "Open domain question answering over tables via dense retrieval," in *ACL*, 2021, pp. 512–519.

[35] T. Jin, Y. Choi, Y. Zhu, and D. Kang, "Pervasive annotation errors break text-to-sql benchmarks and leaderboards," *CIDR*, vol. abs/2601.08778, 2026.

[36] S. Agarwal, A. Biswal, S. Zeighami, A. Cheung, J. Gonzalez, and A. G. Parameswaran, "Arming Data Agents with Tribal Knowledge," Feb. 2026.

[37] T. Ren, C. Ke, Y. Fan, Y. Jing, Z. He, K. Zhang, and X. S. Wang, “The Power of Constraints in Natural Language to SQL Translation,” *Proceedings of the VLDB Endowment*, vol. 18, no. 7, pp. 2097–2111, Mar. 2025.

[38] A. Biswal, C. Lei, X. Qin, A. Li, B. Narayanaswamy, and T. Kraska, “AgentSM: Semantic Memory for Agentic Text-to-SQL,” Jan. 2026.

[39] G. Liargkovas, V. Jabrayilov, H. Franke, and K. Kaffes, “An expert in residence: LLM agents for always-on operating system tuning,” in *Machine Learning for Systems 2025*, 2025. [Online]. Available: https://openreview.net/forum?id=7dhlgPp8ni

[40] P. S. Sodhi, G. Liargkovas, and K. Kaffes, “Empowering machine-learning assisted kernel decisions with ebpfml,” in *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*, ser. eBPF ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 28–30. [Online]. Available: https://doi.org/10.1145/3748355.3748363

[41] T. Zussman, I. Zarkadas, J. Carin, A. Cheng, H. Franke, J. Pfefferle, and A. Cidon, “cache_ext: Customizing the page cache with ebpf,” in *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP ’25)*. Association for Computing Machinery, 2025.

[42] J. Xu, T. Zhou, E. Wu, and K. Kaffes, “Toward systems foundations for agentic exploration,” 2025. [Online]. Available: https://arxiv.org/abs/2510.05556

[43] CRIU Project, “Checkpoint/Restore In Userspace (CRIU),” https://criu.org/, 2012, accessed: 2025-08-06.

[44] ——, “Podman checkpoint/restore integration,” https://criu.org/Podman, 2025, accessed: 2026-03-10.

[45] S. Hao, Y. Gu, H. Ma, J. Hong, Z. Wang, D. Wang, and Z. Hu, “Reasoning with language model is planning with world model,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 8154–8173.

[46] J. F. Allen and J. A. Koomen, “Planning using a temporal world model,” in *Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2*, 1983, pp. 741–747.

[47] United States Congress, “26 U.S.C. § 274(n): Only 50 percent of meal expenses allowed as deduction,” U.S. Code Title 26, 2024, internal Revenue Code, Section 274(n). [Online]. Available: https://www.law.cornell.edu/uscode/text/26/274#n

[48] Z. Zhang, Y. Lu, J. Ma, D. Zhang, R. Li, P. Ke, H. Sun, L. Sha, Z. Sui, H. Wang, and M. Huang, “Shieldlm: Empowering llms as aligned, customizable and explainable safety detectors,” 2024. [Online]. Available: https://arxiv.org/abs/2402.16444

[49] T. J. Green, G. Karvounarakis, and V. Tannen, “Provenance semirings,” *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:1993398

[50] C. Summers, H. Mohammed, and E. Wu, “Please don’t kill my vibe: Empowering agents with data flow control,” *ArXiv*, vol. abs/2512.05374, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:283672095

[51] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, “Trustworthy {Whole-System} provenance for the linux kernel,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 319–334.

[52] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 symposium on cloud computing*, 2017, pp. 405–418.