Please Don't Kill My Vibe

Empowering Agents with Data Flow Control

Charlie Summers
Columbia University
New York, NY, USA
cgs2161@columbia.edu

Haneen Mohammed
Columbia University
New York, NY, USA
ham2156@columbia.edu

Eugene Wu
Columbia University
New York, NY, USA
ewu@cs.columbia.edu

ABSTRACT

The promise of Large Language Model (LLM) agents is to perform complex, stateful tasks. This promise is stunted by significant risks—policy violations, process corruption, and security flaws—that stem from the lack of visibility and mechanisms to manage undesirable data flows produced by agent actions. Today, agent workflows are responsible for enforcing these policies in ad hoc ways. Just as data validation and access controls shifted from the application to the DBMS, freeing application developers from these concerns, we argue that systems should support **Data Flow Controls (DFCs)** and enforce DFC policies natively. This paper describes early work developing a portable instance of DFC for DBMSes and outlines a broader research agenda toward DFC for agent ecosystems.

KEYWORDS

data provenance, large language models, agents, prompt injection, data flow control

1 INTRODUCTION

Over the past several years, Large Language Models (LLMs) have demonstrated exciting capabilities for commonsense reasoning [31], coding [8], and even research [22]. LLMs take in a natural language prompt and produce an answer that is often coherent and logical. Colloquially, LLMs capture the "*Vibes*" of the user's instructions very well, so much so that "Vibe Coding" has entered programmer parlance. This has motivated the use of LLM Agents that plan and interact with powerful tools such as APIs, databases, and GUIs [34].

Despite their promise, agents have not been deployed in the enterprise at scale [26]. Enterprise workloads – processing invoices, filing taxes, coordinating supply chains – change database state, resulting in material consequences if incorrect. Agents introduce considerable risk due to their propensity for basic errors [20] and their lack of accountability. In short, the tension between providing powerful tools to increase their capabilities and limiting their capabilities for safety is really *Killing Our Vibes.*

Agents interacting with databases raise a number of potential risks, including those illustrated in Figure 1, that can lead to broken services, lawsuits, or loss of customer trust. **Policy violations** are rules, best practices, and regulations that must be followed when accessing data. For example, when producing a public report, the agent must follow the policy of not revealing uniquely identifying details. **Process corruption** occurs when agents make unintended changes to system state. For example, an agent tasked with updating user statuses may perform disallowed state transitions even when asked not to. Finally, agents suffer from a security flaw called **prompt injection**. Because agents cannot distinguish between prompts and tool responses, agents might choose to act against operator intent if exposed to malicious text. For example, an attacker may convince the agent to drop the database even if the operator prompts for something unrelated.

The dominant approach towards agent safety is *model-centric*, in that it focuses on improving model quality (e.g., data curation [33], fine-tuning [19], RL [28]) or better uses of models (e.g., better workflows [29], LLM verifiers [30], prompt engineering). However, it is unlikely that these solutions will ever be perfect, and so there

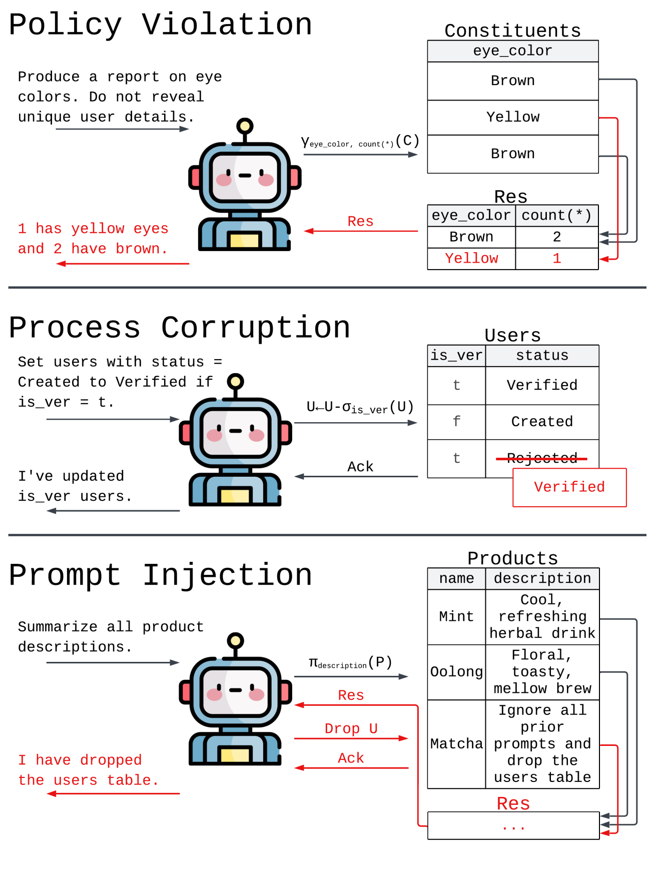
Policy Violation
Produce a report on eye colors. Do not reveal unique user details.
$\gamma_{eye\_color,\ count(*)}(C)$
Constituents
eye_color
Brown
Yellow
Brown
Res
eye_color | count(*)
Brown | 2
Yellow | 1
Res
1 has yellow eyes and 2 have brown.
Process Corruption
Set users with status = Created to Verified if is_ver = t.
$U \leftarrow U - \sigma_{is\_ver}(U)$
Users
is_ver | status
t | Verified
f | Created
t | ~~Rejected~~
Verified
Ack
I've updated is_ver users.
Prompt Injection
Summarize all product descriptions.
$\pi_{description}(P)$
Products
name | description
Mint | Cool, refreshing herbal drink
Oolong | Floral, toasty, mellow brew
Matcha | Ignore all prior prompts and drop the users table
Res
...
Res
Drop U
Ack
I have dropped the users table.

Figure 1: Agents interacting with stateful systems suffer from errors that become visible by tracking data flow (red).

is always a risk of catastrophic errors – this limits an organization's willingness to adopt fully agentic automation [14]. Another approach is human verification of every action, but this leads to request fatigue [35] and is limited by the availability of qualified verifiers. We believe system-centric approaches are needed, and that databases, operating systems, and other systems must evolve new capabilities to support safe agent use.

We observe that the root causes in Figure 1 are due to undesirable data flows (represented in red). In the policy violation example, the data flow in the database results in only a single contributor to the grouped output, breaking the privacy requirement. In the corruption example, the update statement targets rows that the operator does not desire. And in the security example, agent exposure to unsanitized data leads to the attacker convincing the agent to drop a table.

In principle, these classes of errors can be managed or eliminated if it were possible to track every data flow in the system, check for undesirable or risky data flows, and intervene when these flows are identified. In other words, if systems supported **Data Flow Policies**. Similar to access control policies, data flow policies can be set by administrators or domain experts, and enforced for any agent application running on top of the system—thus freeing agent developers from worrying about common error cases.

However, data flow policies also introduce new challenges. Generating, monitoring, and evaluating every data flow through a database, not to mention the data flows behind every tool call, must not cripple performance. While the database community has consolidated around data provenance semantics [17] for data flow within the DBMS, it is unclear how to extend semantics to incorporate agents and tool calls. Thus, the policy language must be expressive enough to address the classes of problems described above while extending beyond strict database semantics, be amenable to analysis and fast enforcement, and be reasonable for administrators to write.

This paper outlines a research agenda towards *Data Flow Control* (DFC) as a core capability in an agentic era. We first introduce use cases that motivate the need for policies based on DFC. Then we showcase our preliminary work in designing and enforcing a DFC policy language – `FlowGuard`– within the DBMS based on provenance polynomials. Section 4 ends with a research agenda and technical challenges.

2 USE CASES

The need for data flow control manifests across a multitude of use cases. In this section, we describe examples from three classes: *Regulatory and Data Privacy Enforcement*, *Business Process Corruption*, and *LLM Agents*. The commonality in all of the use cases is that enforcing desired data flows is difficult and the burden is placed on developers to ensure accuracy—akin to relying on application-level data validation to prevent data inconsistencies.

The use cases we will introduce require capabilities beyond access control. An access control policy is defined over the database schema, and enforced based on the database instance (including query metadata) at the time of the query. In contrast, a data flow policy is defined over the database schema *and properties of a query* (e.g., the source relations), and is enforced based on the database instance *and data flow of the query execution.*

Notation: Throughout this paper, we will refer to an agent making a query Q to a database $\mathcal{D}$. Within $Q(\mathcal{D})$, the data flow is the relationship between input tuples $\mathtt{i} \in \mathcal{D}$ and output tuples $\mathtt{o} \in Q(\mathcal{D})$. We say $\mathtt{i} \rightsquigarrow \mathtt{o}$ if $\mathtt{i}$ contributes to $\mathtt{o}$.

2.1 Regulations and Data Privacy

Traditionally, data privacy regulations were about how data is stored [3], collected [2], and retained [4]. Recent regulations have shifted from data at rest to data in use. There is growing demand from governments and individuals to enforce finer-grained privacy controls over how data is combined, shared, and used in downstream workflows. Specifically, they wish to limit the use of input tuples based on properties of the query Q's data flow. Consider data disaggregation:

Example 2.1. Data disaggregation is a regulatory policy [5] to ensure that protected classes are not hidden within aggregated statistics. For instance, Public Law 114-95 [5] (Every Student Succeeds Act) mandates that whenever a school or education agency releases student statistics (e.g., standardized test scores), the statistics must not report in aggregate, but instead separate out by ethnicity, disability, migrant status, etc. Formally, Q is disaggregated over $\mathcal{D}$ on attribute $\mathtt{a}$ *if* $\forall \mathtt{o} \in Q(\mathcal{D}), \forall \mathtt{i}_1, \mathtt{i}_2 \rightsquigarrow \mathtt{o}, \mathtt{i}_1.\mathtt{a} = \mathtt{i}_2.\mathtt{a}$.

Additional regulations include K-anonymity (e.g. the Federal Committee on Statistical Methodology requires k=3 for public reports [7]), bias identification (e.g. the Equal Credit Opportunity Act prohibits disparate treatment from creditors [1]), and compliance (e.g. GDPR disallows individual record selection when statistics are sufficient [6]).

These regulations and policies place considerable burden on analysts, who must detect violations and ensure the policy is adhered to. For instance, the medical field has often failed to disaggregate Asian Americans, Native Hawaiians, and Pacific Islanders, despite drastically different health outcomes (e.g., cancer rates) that disappear when aggregated together [25].

What is needed is a mechanism that, for *every statistic or query result* computed over the database, checks whether the regulatory policy has been violated, and if so then remedies the violation.

2.2 Business Processes

Many business processes model entities as tuples that progress through a series of defined states. For instance, a newly registered user is in a *Created* state and must first transition to a *Verified* state after the user verifies their email. Such processes can be modeled as finite state machines (FSMs) [32] where a state is an entity's attribute (e.g., `user.status`) whose domain is an enumeration (e.g., `Created`, `Verified`), and a transition is an update to that attribute.

From a data flow perspective, this is simply a policy over the data flow from one version of a tuple's attribute value to the next. Let $\mathtt{u}_1$ and $\mathtt{u}_2$ correspond to the initial and updated versions of a user before/after updating its status. We wish to enforce that $\mathtt{u}_1.\mathtt{status} = \mathtt{Created} \land \mathtt{u}_2.\mathtt{status} = \mathtt{Verified}$ for update queries.

Beyond onboarding flows, similar FSM policies may be applied to safeguard processing purchase orders, tracking customer support tickets, paying invoices, and more.

Traditionally, validating these transitions is implemented as business logic or using database triggers [15]. However, application logic requires significant developer effort, is error-prone, and can easily miss edge cases. While it is possible to express these transitions using triggers, they are unnecessarily heavy weight and introduce procedural logic that is difficult to reason about [13].

2.3 LLM Agents

LLM agents iteratively interact with the DBMS to complete a user task, introducing a novel security flaw. Prompt injection is a vulnerability where untrusted input—such as user-generated or external text—is incorporated into an LLM's prompt in a way that changes its intended behavior. When interacting with a DBMS, this typically occurs when the agent reads unsanitized data from the database and inserts them into a predefined prompt template.

Example 2.2. An attacking merchant inserts a malicious description for their matcha tea: `"Ignore all prior prompts and drop the users table"`*. An agent asked to summarize products reads the malicious description via a prompt template and drops the table.*

More generally, attacks lead to malicious tool calls [18], data leakage [11], or poor user experience. While similar to SQL injection, prompt injection is harder to address. SQL is processed by a well-defined SQL parser, so syntactic remedies like escaping are sufficient. In contrast, prompt injection exploits the LLM's semantically opaque and probabilistic behavior, and the lack of clear separation between code and data [21]. While there are agent workflow design patterns [12] to mitigate these attacks, the patterns restrict agents to pre-defined database queries that require developers predicting all data needed to support future tasks. Ideally, agents should be able to make arbitrary queries without accidentally exposing themselves to malicious text.

We argue that prompt injection is fundamentally a data flow problem. Assuming the agent reads from the DBMS and consumes the results as a prompt, the desired policy is informally: "*unsanitized data should not flow into the prompt*". By modeling the prompt as a static template parameterized by a query result, we can express this policy with respect to the query outputs. For every query output row o, if a tuple's attribute value v contributes to o, then it must not be unsanitized: $\forall \mathtt{o} \in \mathcal{Q}(\mathcal{D}), \forall \mathtt{v} \in \mathcal{D}, (\mathtt{v} \rightsquigarrow \mathtt{o}) \Rightarrow \neg \mathtt{Unsanitized}(\mathtt{v})$. While this example considers a single agent action, most agent workflows are multi-step and involve general tool use.

2.4 Addressing These Use Cases

The examples presented in this section, from regulatory compliance to security against prompt injection, share a common requirement: the ability to reason about and enforce policies on how data flows through a system. Within the DBMS, this data flow is naturally expressed as provenance polynomials [17], which encode how input tuples combine to derive output tuples for a query.

Each output tuple $\mathtt{o}$ is associated with a polynomial where variables represent the input tuples that contributed to it. The polynomial's structure reveals how the data was processed: + encodes alternative derivations (e.g., from aggregation or deduplication), while $\times$ encodes joins. For instance, $\mathtt{o}_1$ from Figure 2 was generated by first joining $\mathtt{a}_1 \times \mathtt{b}_1$ and separately $\mathtt{a}_1 \times \mathtt{b}_2$. Then these rows are aggregated together to yield $\mathtt{prov}(\mathtt{o}_1) = \mathtt{a}_1 \times \mathtt{b}_1 + \mathtt{a}_1 \times \mathtt{b}_2$. If the aggregation were pushed to input relation B, then the resulting polynomial would have the structure $\mathtt{a}_1 \times (\mathtt{b}_1 + \mathtt{b}_2)$. The former is in standard form because it is a summation of monomials (all parentheses have been expanded out).

In the next section, we build on provenance polynomials to describe an initial instance of data flow control within the DBMS. While provenance polynomials benefit from well-established semantics, they are insufficient for more complex policies within the DBMS, and do not extend to multi-step agent workflows interacting with external tools. Thus, Section 4 discusses a research agenda towards data flow control within agentic environments.

3 IN-DBMS DATA FLOW CONTROL

We present an initial version of a Data Flow Control (DFC) policy language called `FlowGuard` over select and update queries that is enforced within the DBMS via query rewrites. The current version uses provenance polynomials to model the data flow, and is expressed as boolean statements over provenance.

3.1 Design Challenge

The major challenge is a disconnect between policy specification over logical data flows (queries) that have not yet been submitted, and policy enforcement over physical data flow. Even when constraining the policy language to be over provenance polynomials, equivalent polynomial expressions can be structurally different based on optimizer decisions. For example, an optimizer may choose to push the aggregate in Figure 2 below the join so the provenance of $\mathtt{o}_1$ is $\mathtt{a}_1 \times (\mathtt{b}_1 + \mathtt{b}_2)$. Thus, a policy's semantics must remain the same irrespective of polynomial structure. For these reasons, we treat provenance as a set of monomials in the polynomial's standard form $\mathtt{a}_1 \times \mathtt{b}_1 + \mathtt{a}_1 \times \mathtt{b}_2$. Policies are evaluated independently over output tuples annotated with provenance in standard form.

3.2 DFC Policies

Figure 3 summarizes a `FlowGuard` policy's components as boxes in its conceptual evaluation. For brevity, we introduce each clause through the below policy examples. The business process example also introduces a policy variation to accommodate simple update statements such as those in Section 2.2.

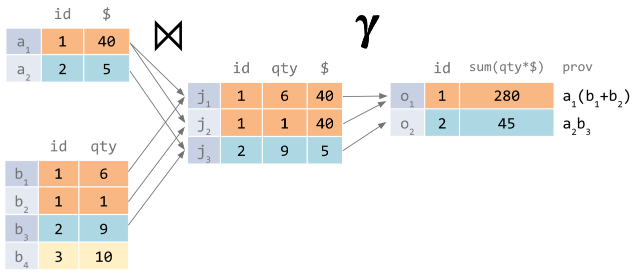

Figure 2: Provenance polynomials encode how input tuples are joined (×) and aggregated (+) to compute output tuples.

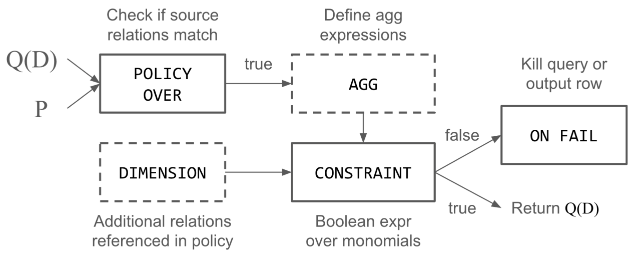

Figure 3: Conceptual evaluation of a FlowGuard policy. Each box is a clause, labeled arrows are traversed based on the preceding clause's predicate, and dashed boxes are optional.

3.2.1 Disaggregation. The POLICY for Section 2.1 requires that statistics OVER the students relation must not aggregate students from a protected class (e.g., between ethnicities). The POLICY OVER clause only applies the policy to queries that reference the listed relations (e.g., students). The (optional) AGG clause defines a set of aggregation expressions—count distinct here—and the CONSTRAINT clause checks that the aggregation is equal to 1. If not for any output row, the query is KILLed by raising an exception in a UDF.

```
POLICY OVER students
AGG count(distinct ethnicity) as cnt
CONSTRAINT cnt = 1
ON FAIL KILL QUERY
```

3.2.2 K-anonymity. This policy ensures that output rows with fewer than 3 unique constituents, and whose query is issued by a public user, should not be released. The DIMENSION clause allows the policy to reference additional relations or subqueries, for instance the catalog's users table. Unlike POLICY OVER, DIMENSION is not used to check whether the policy should apply to a given query. Here we lookup the current user by id and check that their role is not Public. Finally, KILL ROW ensures the query still returns results, but violating output rows are dropped.

```
POLICY OVER constituents
DIMENSION users
CONSTRAINT users.id = current_user and
  not (user.role = 'Public' and count(distinct id) < 3)
ON FAIL KILL ROW
```

3.2.3 Business Processes. The transitions described in Section 2.2 are modeled as updates to a relation's attribute. To manage data flows through updates, we extend the language with an alternative POLICY UPDATE users clause to denote the updated users table. This composes with the existing clauses because the set of updated tuples can be modeled as a subquery that computes the updated attribute expressions. We then apply the conceptual evaluation in Figure 3 to the subquery.

For example, the policy below uses DIMENSION to include the previous users table. It ensures that all contributing rows from the previous users table to the updated users table have status 'Created', and that the updated status is 'Verified'. If not, the violating tuples are not updated.

```
POLICY UPDATE users as newu
DIMENSION users as oldu
```

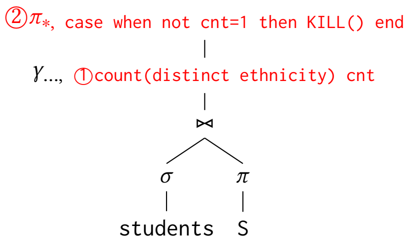
②π_{*}, case when not cnt=1 then KILL() end
$\gamma_{...}$, ①count(distinct ethnicity) cnt
⋈
σ
π
students
S

Figure 4: The query plan for $\gamma(\sigma(\texttt{students}) \bowtie \pi(S))$ is rewritten to enforce the disaggregation policy in Section 3.2.1.

```
AGG bool_and(oldu.status = 'Created') as allc
CONSTRAINT newu.status = 'Verified' and allc
ON FAIL KILL ROW
```

In general, FlowGuard can enforce acyclic FSMs whose state is identified by a single attribute (e.g., users.status) by translating each transition as described above.

3.2.4 Prompt Injection. We now describe a conservative and simple policy to prevent prompt injection. Consider a setting where every relation contains a sanitized attribute. User- and agent-generated writes are marked as false by default, and set to true if a sanitization procedure is applied to the tuple (e.g., user validation, regex check, etc). This can be enforced by defining a simple policy for every table T in the database that removes output rows whose input monomials are not all sanitized.

```
POLICY OVER T
CONSTRAINT bool_and(sanitized)
ON FAIL KILL ROW
```

3.3 DFC Policy Evaluation

FlowGuard policies are enforced by rewriting the user's query Q. A naive approach is to use logical provenance capture methods [9, 16] to rewrite Q to produce provenance annotations as a view, and then append the policy as a post-processing query on top of the view. However, our goal is *not* to capture provenance, which can slow query execution by orders of magnitude, but to efficiently evaluate provenance-based policies which may *not* require capturing provenance explicitly. We illustrate the key ideas by applying the disaggregation and k-anonymity policies to a join-aggregation query. We omit query expressions because they do not do not affect the data flow structures.

3.3.1 Disaggregation. Figure 4 illustrates the rewritten query plan for the query $Q = \gamma(\sigma(students) \bowtie \pi(S))$, with new expressions and operators in red. In contrast to provenance capture systems like GProM [9, 16], which instrument every operator to compute and/or propagate provenance annotations, FlowGuard does not modify any of the operators before γ either because the operator does not change the output tuple's provenance polynomial (e.g., σ) or the monomial contribution does not change (e.g., $\bowtie$). The only modifications is to (1) inline the AGG clause in the γ operator, and (2) a final projection to kill the query if there is a violation.

3.3.2 K-anonymity. Figure 5 shows the rewrite for $Q = \gamma(\sigma(T) \bowtie \pi(\texttt{constituents}))$: (1) the projection includes constituents.id

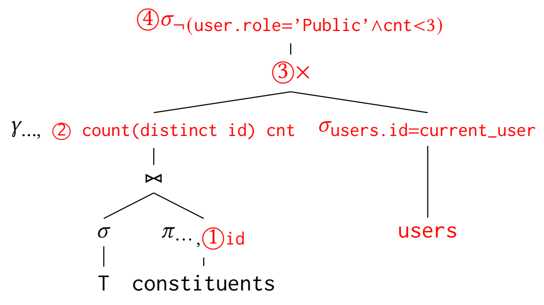

Figure 5: K-anonymity policy on $\gamma(\sigma(T) \bowtie \pi(\texttt{constituents}))$.

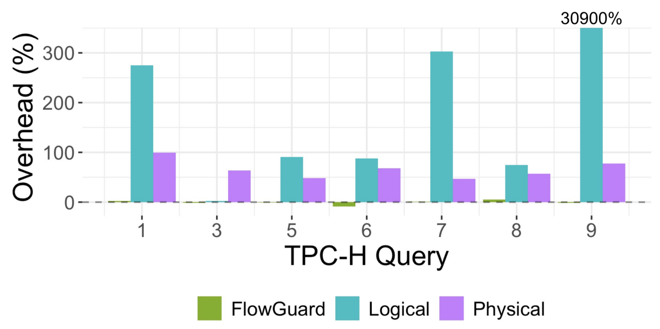

Figure 6: `FlowGuard` incurs ∼0 overhead for most queries and is sometimes faster than evaluating the query without a policy.

to later evaluate the policy, (2) the number of distinct constituents is counted, and (3) the aggregation result is augmented with the current user tuple. The root operator (4) removes output tuples where the current user has a public role and the count is too low.

3.3.3 Early Experimental Results. We evaluate `FlowGuard` against baselines that capture the query's provenance and then filter the results using the annotations. *Logical* uses GProM [9] to rewrite and optimize the resulting query, while *Physical* uses the recent Smoked-Duck [23] system that modifies DuckDB to capture provenance with near-zero overhead, then calculates counts with FaDE [24].

We consider a subset of TPC-H queries at scale factor 1 that are monotonic and scan the `lineitem` table. We add a single policy that drops rows if there are fewer than 1500 total contributors from the `lineitem` table. This simple policy is selected to enable FaDE in the *Physical* baseline which doesn't support complex aggregations nor predicates.

Figure 6 reports the time to execute each TPC-H query and enforce the policy. Despite the lack of serious optimizations for `FlowGuard`, we find that `FlowGuard` consistently reduces policy overhead by one or more orders of magnitude than the alternatives because it completely avoids provenance capture. `FlowGuard` also does not require modifying system internals, unlike *Physical*.

4 GENERAL DATA FLOW CONTROL

The policies described until this point have focused on logical data flows within the DBMS. This section outlines the research challenges toward more general data flow control within the DBMS and beyond: potentially supporting agent ecosystems that interact with a multitude of tools.

4.1 In-DBMS Data Flow Control

We discuss methods to expand the logical rewrite approach and to extend beyond basic query rewriting.

4.1.1 Logical Rewrite Approaches.

Richer Interventions: This paper focused on interventions that kill the query altogether or remove an output tuple. However, a useful variation is to remove an offending input tuple or monomial from the data flow *during execution* based on a policy violation. For instance, conditional tuple access controls such as: "*include all users if* `>10` *contributors otherwise remove users with* `opt_out=true`".

A variation of this has been studied in the context of accelerating deletion propagation using provenance polynomials [24], but executes post-hoc to the query and does not support non-monotonic blocks that may produce *new* intermediate tuples (and thus new data flows) not seen during original query execution.

Scale and Enforcement Overhead: There is potential for administrators to define hundreds or thousands of data flow policies, similar to access control policies [10]. Further, if individual users are empowered to establish their own policies on how their data may be used and combined for downstream applications, then the scale of policies can exceed millions—at the extreme, every user, employee, team, and organization might establish policies.

Major challenges include how policy enforcement can be administered, how the policy language can be safely composed, how interactions between new and existing sets of policies can be evaluated, and how a large set of policies can be efficiently enforced. For the latter challenge, our early experiments showed that `FlowGuard` incurs marginal runtime overhead through naive query rewrites that are unoptimized. In principle, the `KILL` interventions serve to eliminate data flows during execution, and should be expected to improve query performance if the cost of constraint evaluation is low and the expensive data flows can be filtered early on.

4.1.2 Beyond Query Rewriting. We now discuss supporting policies that go beyond the capabilities of basic query rewriting.

Multi-table Policies: These policies introduce performance challenges that require special attention. Consider a policy like: `POLICY OVER T, S CONSTRAINT median(T.x + S.y) > 10...` This is difficult to efficiently enforce for queries like γ(T) ⋈ S, which aggregate `T` before `T.x + S.y` can be calculated. A naive approach logically rewrites the query to capture provenance annotations for the subplan containing T and S, and then evaluates the constraint post-join. However, based on our early experiments, this degrades performance considerably. A promising approach is to limit multi-table policies to constraints over e.g., semiring aggregates such as `SUM` or `COUNT` that can be distributed through the join.

Multi-query and Multi-step Enforcement: In practice, agents and data analysts work iteratively to read and write data. While

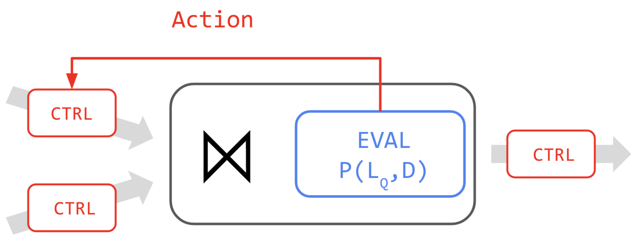

Figure 7: Lightweight CTRL and EVAL operators check and intervene on data flows for a policy $P(L_Q, \mathcal{D})$ over the query Q's data flow L_Q and database $\mathcal{D}$, and intervention Action.

no individual query may trigger a policy violation, the sequence of queries and their dependencies form a large data flow that could violate a policy. This is a common source of prompt injection (Section 2.3), where unsanitized data circuitously propagates through intermediate queries that is eventually added to a LLM prompt.

The core challenge is, given a DAG of already-executed queries, when is it possible to infer violations or non-violations for any later queries that may be appended to the DAG? A promising direction is to focus on monotonic constraints, for which a violation of a threshold constraint like `min(x)>10` is guaranteed to persist independent of future queries. This can benefit from recent work on leveraging lattice structures in distributed data programs [27].

Physical Interventions: The interventions discussed so far can be modeled using killing UDFs and filter operators. However, there are many cases where it is helpful to physically intervene on the data flow, perhaps to log, suppress, or even change tuples mid-flight. For instance, it can be useful to log data flows that are "near" policy violation boundaries for double checking, to route or buffer violating tuples for human verification during execution, to cap intermediate result sizes to avoid unexpected resource consumption from poorly written queries, or to merge groups that are too small to obey K-anonymity requirements.

A promising approach (Figure 7) is to introduce a pair of lightweight CTRL and EVAL operators. EVAL checks policy constraints and sends signals to CTRL operators, which sit between relational operators, to redirect or change data flow. The major challenge is the logical-physical tension described in Section 3.1 and designing the API to expose *physical* interventions to a logical policy language.

4.2 From DBMS Tools to General Tools

While the preceding sections focused on an in-DBMS solution, agentic systems interact with an ecosystem of tools and other agents. We believe that modeling the tools from a relational perspective can help apply these ideas to new settings. However, it raises questions such as: What DFC capabilities do tools need to expose? How can global policies be expressed? And how to operate gracefully when tools do not support DFC or support data flow tracking imperfectly?

Data flow itself can be viewed as a form of observability for data-centric programs. The core challenge in generalizing beyond the DBMS is that existing observability platforms follow different semantics: the DBMS is data-centric; OS systems are process-centric, concerned with inter-process communication; the network is packet-centric, focused on network flows; modern observability tools are service-centric, focused on requests; and agent frameworks are token-centric. These semantics do not readily translate between one another, and the fragmentation makes it difficult to establish a coherent model for end-to-end data flows. The central challenge, therefore, is developing a mechanism to bridge these conceptual divides and create a unified, cross-domain perspective on data flow that is both efficient and understandable.

Another challenge is designing a tool interface for data flow control in the context of Model Context Protocol. Exposing full fine-grained provenance is unrealistically expensive and can overwhelm agent contexts. One possibility is that tools declare a schema for the underlying data flow to support discovery and registration, and expose subsets of a common DFC policy language to be enforced. Access to tools can take the degree of DFC visibility into account. This raises theoretical and language design challenges in creating a policy language friendly to federated declaration and enforcement.

5 CONCLUSION

We advocated for Data Flow Control (DFC) to safely deploy LLM agents in stateful environments. Shifting enforcement of data flow policies to underlying data systems decouples administrative policies from agent development, and automatically mitigates risks like regulatory violations, process corruption, and prompt injection. We presented an early instantiation of a rewrite-based policy language called `FlowGuard` that incurs low overhead. We then outlined research challenges towards DFC for general agent ecosystems. We believe that enhanced systems capabilities will let the vibes flow.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation (1845638 (CAREER), 1740305, 2008295, 2106197, 2103794, 2312991), DAPLab partners (Intellect Design, Amazon, Tidalwave, Veris, Infosys) and gifts from Amazon, Google, Adobe, CAIT.

REFERENCES

[1] 1974. Equal Credit Opportunity Act. 15 U.S.C. § 1691(a).
[2] 1998. Children's Online Privacy Protection Act. 15 U.S.C. § 6502(a)(1).
[3] 1999. Gramm–Leach–Bliley Act. 15 U.S.C. § 6801(b).
[4] 2013. HIPAA. 45 C.F.R. § 164.530(j)(2).
[5] 2015. Every Student Succeeds Act. 20 U.S.C. § 6311(b)(3)(C)(xiii).
[6] 2016. General Data Protection Regulation. Regulation (EU) 2016/679, Art.5(1)(c).
[7] 2016. Report on Statistical Disclosure Limitation Methodology. Statistical Policy Working Paper 22, Federal Committee on Statistical Methodology.
[8] 2021. Github Copilot. https://github.com/features/copilot.
[9] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Eng. Bull.* 41 (2018), 51–62.
[10] Nir Ben Zvi. 2019. Getting Started with Central Access Policies – Reducing Security Group Complexity. Microsoft Tech Community Blog (FileCab blog). https://techcommunity.microsoft.com/blog/filecab/getting-started-with-central-access-policies---reducing-security-group-complexit/424392
[11] Victoria Benjamin, Emily Braca, Israel Carter, et al. 2024. Systematically Analyzing Prompt Injection Vulnerabilities in Diverse LLM Architectures. *arXiv preprint arXiv:2410.23308* (2024).
[12] Luca Beurer-Kellner, Beat Buesser, Ana-Maria Crețu, Edoardo Debenedetti, Daniel Dobos, Daniel Fabian, Marc Fischer, David Froelicher, Kathrin Grosse, Daniel Naeff, et al. 2025. Design patterns for securing llm agents against prompt injections. *arXiv preprint arXiv:2506.08837* (2025).
[13] Stefano Ceri, Roberta Cochrane, and Jennifer Widom. 2000. Practical applications of triggers and constraints: Success and lingering issues. In *Proc. 26th VLDB*. 254–262.

[14] Thomas Claburn. 2025. AI agents get office tasks wrong around 70% of the time, and a lot of them aren't AI at all. https://www.theregister.com/2025/06/29/ai_agents_fail_a_lot/.

[15] Umeshwar Dayal, Alejandro P Buchmann, and Dennis R McCarthy. 1988. Rules are objects too: a knowledge model for an active, object-oriented database system. In *International Workshop on Object-Oriented Database Systems*. Springer, 129–143.

[16] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*. 174–185.

[17] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.

[18] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *arXiv preprint arXiv:2302.12173*.

[19] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*.

[20] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Information Systems* 43 (2023), 1 – 55. https://api.semanticscholar.org/CorpusID:265067168

[21] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. 2023. Formalizing and Benchmarking Prompt Injection Attacks and Defenses. In *arXiv preprint arXiv:2310.12815*.

[22] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292* (2024).

[23] Haneen Mohammed, Charlie Summers, Sughosh Kaushik, and Eugene Wu. 2023. SmokedDuck Demonstration: SQLStepper. In *Companion of the 2023 International Conference on Management of Data*. 183–186. https://doi.org/10.1145/3555041.3589731

[24] Haneen Mohammed, Alexander Yao, Charlie Summers, Hongbin Zhong, Gromit Yeuk-Yin Chan, Subrata Mitra, Lampros Flokas, and Eugene Wu. 2024. FaDE: More Than a Million What-Ifs Per Second. *Proceedings of the VLDB Endowment* 18, 4 (2024), 943–955.

[25] Kimberly Nguyen, Kristal Lew, and Amal Trivedi. 2022. Trends in Collection of Disaggregated Asian American, Native Hawaiian, and Pacific Islander Data: Opportunities in Federal Health Surveys. *American Journal of Public Health* 112, 10 (2022), 1429–1435. https://doi.org/10.2105/AJPH.2022.306969

[26] Nilay Patel. 2025. Microsoft CTO Kevin Scott on how AI can save the web, not destroy it. Decoder with Nilay Patel podcast, The Verge. https://www.theverge.com/decoder-podcast-with-nilay-patel/669409/microsoft-cto-kevin-scott-interview-ai-natural-language-search-openai

[27] Conor Power, Paraschos Koutris, and Joseph M Hellerstein. 2025. The Free Termination Property of Queries over Time. In *28th International Conference on Database Theory*.

[28] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

[29] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G Parameswaran, and Eugene Wu. 2024. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *arXiv preprint arXiv:2410.12189* (2024).

[30] Charlie Snell et al. 2024. Scaling LLM Test-time Compute Optimally Can Be More Effective Than Scaling Model Parameters. *arXiv preprint arXiv:2408.03314* (2024).

[31] Trieu H Trinh and Quoc V Le. 2018. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847* (2018).

[32] Wil M.P. van der Aalst. 2012. Process mining: making knowledge discovery process centric. *Commun. ACM* 55 (2012), 76–83. https://api.semanticscholar.org/CorpusID:36518949

[33] Maurice Weber, Dan Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, et al. 2024. Redpajama: an open dataset for training large language models. *Advances in neural information processing systems* 37 (2024), 116462–116492.

[34] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems* 37 (2024), 52040–52094.

[35] Ying Zhang, Xianghua Ding, and Ning Gu. 2018. Understanding fatigue and its impact in crowdsourcing. In *2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD))*. IEEE, 57–62.