A FirstEigen White Paper



Bad Data Should Be Stopped Inside the Pipeline, Not Found After the Fact

Finding the multi-row and business-context rules that profiling alone won't, then drafting contracts from your catalog and enforcing the approved ones inside the pipeline.

© FirstEigen LLC. All rights reserved.

Executive Summary

A pipeline can finish green and still deliver wrong data. A transaction can be valid in isolation and still breach a control that only shows up across several of them, and two well-formed timestamps can contradict the order in which a manufacturer ships and receives parts. A mortgage record can have every date filled in and still miss the payment window the business cares about. Column types and row counts won't tell you any of this, and the usual statistical profiles won't either.

So you need two gates: one after ingest and before any transformation, and a second after the transformations and before you write to a trusted destination. The frameworks that run those checks already exist. What's hard is deciding which checks belong there. Getting them approved, and keeping them current across hundreds or thousands of assets, is harder still.

DataBuck SDK goes after the discovery part. It combines relationships it observes in the data with what the organization already knows: catalog metadata, business policies, existing contracts and reference data, plus query history where it's available. Engineering and business owners review what it proposes, and the approved, versioned expectations are what run in production. Where we think this pays off most is across the whole catalog. The idea is to walk the supported assets and draft Open Data Contract Standard (ODCS) contracts and expectations for each one, instead of writing them by hand, table by table.

© FirstEigen LLC. All rights reserved.

Put the gates inside the pipeline

Gate 1: before you transform

Check incoming DataFrames before any join or aggregation, and before you derive features or compute financial results. Invalid identifiers and broken reference values are best caught as close to the source as possible, and so are prohibited relationships and temporal violations. Critical failures can be quarantined, or the job can stop, under an explicit completeness policy.

Gate 2: before you write

Correct inputs can still produce incorrect outputs. A join might duplicate entities, and an enrichment step might erase a mandatory attribute. Even a plain transformation can throw off totals or ordering. So run an output gate before data lands in analytics tables or operational APIs, and before it feeds financial reporting or AI workloads.

A quick clarification on "in-pipeline." It means the checks run natively inside the processing workflow, not that the data has to fit on one machine. A Spark DataFrame is a logical, distributed dataset, and the rule predicates should normally run on that same distributed engine instead of shipping rows off to an external inspection service.

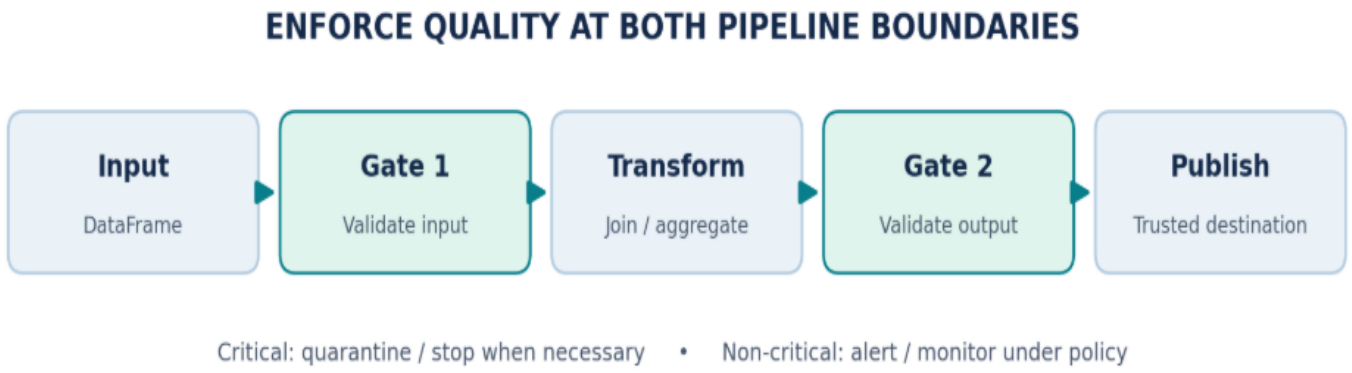

Figure 1. Two gates: one ahead of the transformation, one ahead of the destination write.

© FirstEigen LLC. All rights reserved.

Knowing which check to run is the hard part

Great Expectations, PyDeequ, Soda and Databricks DQX all support programmatic validation, and some offer profiling or rule recommendations. They're good at what they do. But profiling a column can't recover what an organization's policy says about grouped transactions. It can't recover event sequencing or business deadlines either. Those rules depend on context and on relationships between records, and someone has to know which exceptions are allowed.

There's a big gap between "amount is non-null" and "combined activity by an account owner across branches exceeds a control threshold," and another between "date is valid" and "parts receipt occurs after shipment." Anyone can write the first kind. The second kind is where the real risk sits, and recommending it is DataBuck's job. It also records where each rule came from, so a person can review it and decide whether to enforce it.

© FirstEigen LLC. All rights reserved.

Three examples of context-aware discovery: automotive and CCAR

Automotive manufacturing: shipment to receipt

For a large automotive manufacturer, DataBuck identified that a part's received date must fall after its shipped date. Both fields can be valid dates, but the relationship between them is inconsistent if a part appears to arrive before it was shipped. Depending on the source model, enforcing this may mean joining shipments to receipts on a part, lot or shipment identifier.

Illustrative expectation:

$$parts_received_date > parts_shipped_date$$

subject to approved exceptions for same-day events and time zones, and for later corrections.

CCAR mortgage data: timing of the first payment

In a CCAR-related mortgage dataset (CCAR is the Comprehensive Capital Analysis and Review), DataBuck identified a business rule requiring the first payment date to fall within 90 days of the loan approval date. A format check can't catch a violation of that rule, because it compares two business events across a specific interval.

Illustrative expectation:

$$0 \leq \text{DAYS_BETWEEN}(loan_approval_date, first_payment_date) \leq 90$$

This is a requirement for one use case, not a universal CCAR mandate. The approved product or policy definition decides the real constraint.

AML: aggregating transactions across rows and branches

In a dataset tied to anti-money-laundering controls, DataBuck identified a requirement to add up the transactions initiated by the same account owner, across one branch or several. The question is whether the combined amount crosses a defined threshold within the relevant time window. Every individual amount can be perfectly valid. The rule only exists once you group many rows by owner identity and apply the policy threshold to the combined activity.

Illustrative expectation:

$$\text{SUM}(qualifying_transaction_amount)$$

for a given account_owner_id and policy-defined time window, across all relevant branches is evaluated against a threshold T. Exceeding T should raise whatever flag or control outcome the policy defines. It should not automatically drop records, and it is not a finding of unlawful activity. Everything specific comes from the actual policy: the threshold, the period, which transaction types qualify and how ownership is resolved.

© FirstEigen LLC. All rights reserved.

Why a rules engine alone doesn't solve discovery

Use case	Rule shape	Context required	Execution implication
AML / banking	Multi-row group, sum and threshold	Owner identity, branches, qualifying activity, window, policy threshold	Distributed group-by or window, plus a policy alert
Automotive	Cross-event temporal comparison	Shipment/receipt semantics, joining keys, timezone	Row comparison or shipment-receipt join
CCAR / mortgage	Two-date interval constraint	Loan lifecycle definition and approved 90-day window	Date interval predicate and exception policy

These examples show why a rules engine alone doesn't solve discovery. DataBuck can propose a rule from the context it has, but the owners still have to confirm exactly what the rule means, which exceptions apply and which records it covers. They also have to settle the thresholds and the enforcement action before anything is switched on.

© FirstEigen LLC. All rights reserved.

Where the context comes from

DataBuck looks at the shape and behavior of each dataset alongside enterprise evidence, and that evidence isn't all equally authoritative. An approved contract is a requirement, and a policy document expresses business intent. Catalog descriptions tell you what things mean. Joins that keep repeating in query history suggest likely relationships, and statistical patterns are hypotheses, not policy.

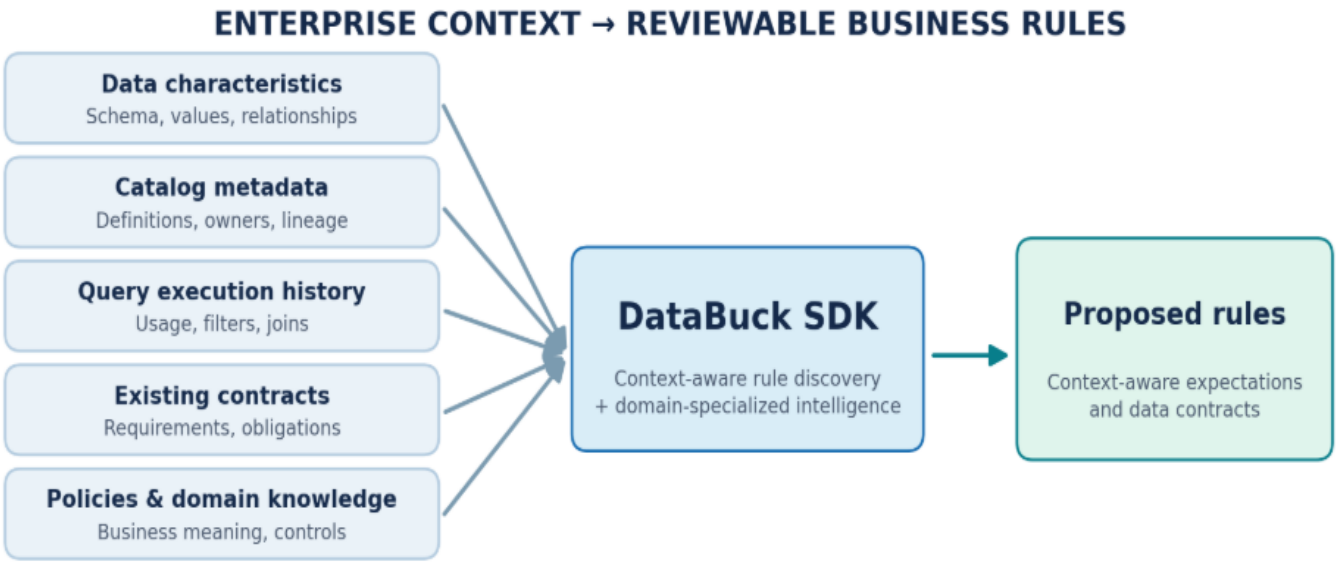

Figure 2. Data characteristics, catalogs, query history, contracts and policies all feed reviewable candidate rules.

Evidence	Candidate rule it may support
DataFrame patterns and dependencies	Cross-column relationships and candidate completeness constraints
Data catalog and glossary	Business identifiers, event meanings, domain definitions and ownership
Query history / lineage	Cross-asset joins, entity keys and frequently used relationships
Policies and existing contracts	Thresholds, timing rules, exceptions and mandatory controls
Reference datasets	Permitted value sets and referential validations

© FirstEigen LLC. All rights reserved.

Data contracts: the bottleneck is the work, not the standard

Data contract standards exist already. The Open Data Contract Standard gives you a machine-readable structure for a dataset's schema and quality expectations, along with service levels, ownership and other contract metadata. What many programs lack isn't another YAML or JSON spec. It's the effort of producing accurate, governed contracts that say something meaningful for every asset that needs one.

Picture a catalog with thousands of tables, each with dozens or hundreds of fields and a web of business relationships. Interviewing SMEs to pin down quality requirements and turning them into ODCS expectations is slow work, and keeping every contract current through schema changes makes it a large implementation program. A well-designed standard doesn't fill itself in.

Catalog-wide generation: how DataBuck approaches it

The intended DataBuck SDK workflow iterates over the assets registered in a connected, supported catalog. For each asset it gathers the schema and metadata it's allowed to see, and analyzes the data and its relationships. Where supported, it also brings in approved policies, contracts and query history, and then proposes context-aware expectations. Those get assembled into an ODCS-compatible contract for the asset, which goes to the owner for approval and is published to an agreed repository or catalog integration.

None of this means every business rule is known automatically. Which catalog connectors and contract publishers you get depends on your SDK edition. Assets with thin metadata or little policy evidence will still need a person, and so will any you can't access.

© FirstEigen LLC. All rights reserved.

From catalog to contract, step by step

Stage	What happens	Governance checkpoint
1. Enumerate	List supported domains, schemas, tables and views from the catalog	Scope, access and asset selection
2. Enrich	Retrieve schema, business terms, lineage, policies and known contracts	Provenance and access controls
3. Discover	Analyze fields, related assets, rows and business context for candidate quality rules	Confidence, false positives, owner review
4. Generate	Create an ODCS-compatible contract per asset with schema and quality expectations	Schema linting; no unsupported assertions
5. Approve	Assign severity, owner, exceptions, effective date and enforcement behavior	Business and engineering sign-off
6. Publish	Version in Git or governed repository and optionally synchronize supported catalog targets	Change approval and release traceability
7. Enforce	Translate or export approved quality rules to supported runtime engines	Validation results, drift monitoring and replay

© FirstEigen LLC. All rights reserved.

What goes into each generated contract

An ODCS-compatible contract starts with an identifier and a version, and points to the catalog asset and its logical schema. Under that sit the column types and whether each column is required, plus quality assertions at the field or dataset level. It can also hold business descriptions and ownership, plus any service-level requirements. Give every expectation a rule ID and a description, and point to its evidence, either in the supported fields or in extension metadata. If you can't explain why a rule exists, nobody will trust it.

For our three use cases the contracts would look quite different. The AML one carries a policy-linked aggregation across rows. The automotive one is about event order between shipment and receipt, and the mortgage one is a first-payment interval. Not every complex check reduces to a simple column predicate, and not every framework executes them the same way. Keep the rule definition engine-neutral where you can, and generate engine-specific expressions or jobs only where you must. Validate the generated contract against the ODCS version you've chosen before you publish it.

© FirstEigen LLC. All rights reserved.

Worked example: CCAR mortgage data contract

The following illustrative ODCS v3.2.0 YAML contract shows how DataBuck-discovered candidate rules can be represented for one mortgage asset. The field names are assumed examples rather than extracted customer schema, and business owners must approve all thresholds and exceptions before activation.

Five controls are represented: positive approved loan amount; valid typed approval and first-payment dates; interest rate greater than 0.25% for jumbo loans or greater than 1% for regular loans; first payment within 0–90 days after approval; and appraised value strictly greater than the approved loan amount.

This is a proposed contract artifact illustrating ODCS fields and SQL quality assertions, not a captured DataBuck SDK export or a claim that ODCS itself executes the checks. Validate the final YAML with an ODCS v3.2-compatible linter and test the generated SQL in the deployment engine.

Implementation notes

Interest rates are assumed to be stored as decimal fractions (0.0025 = 0.25%; 0.01 = 1%). ISO-formatted dates must be parsed and validated at ingestion into the declared date type; if source values are raw strings, add explicit parse-failure checks. The datediff expression uses Databricks/Spark SQL syntax and must be adapted for other engines. Null checks, accepted loan-type values, currency handling, and approved product exceptions should be enforced before production use.

© FirstEigen LLC. All rights reserved.

Example contract: schema (illustrative)

```yaml
apiVersion: v3.2.0
kind: DataContract
id: urn:datacontract:risk:mortgage-loans
name: mortgage_loans_ccar
version: 1.0.0
status: proposed
description:
  purpose: CCAR mortgage quality requirements; subject to owner approval.
schema:
  - name: mortgage_loans
    physicalName: mortgage_loans
    properties:
      - name: loan_id
        logicalType: string
        required: true
      - name: loan_type
        logicalType: string
        required: true
      - name: approved_loan_amount
        logicalType: number
        required: true
      - name: appraised_value
        logicalType: number
        required: true
      - name: interest_rate
        logicalType: number
        required: true
      - name: loan_approval_date
        logicalType: date
        required: true
      - name: first_payment_date
        logicalType: date
        required: true
```


© FirstEigen LLC. All rights reserved.

Example contract: quality rules (illustrative)

```
quality:
  - id: ccar_loan_amount_positive
    type: sql
    query: |
      SELECT COUNT(*) FROM mortgage_loans
      WHERE approved_loan_amount <= 0
    mustBe: 0
  - id: ccar_interest_by_loan_type
    type: sql
    query: |
      SELECT COUNT(*) FROM mortgage_loans
      WHERE loan_type NOT IN ('JUMBO', 'REGULAR')
      OR (loan_type = 'JUMBO' AND interest_rate <= 0.0025)
      OR (loan_type = 'REGULAR' AND interest_rate <= 0.01)
    mustBe: 0
  - id: ccar_first_payment_within_90_days
    type: sql
    query: |
      SELECT COUNT(*) FROM mortgage_loans
      WHERE datediff(first_payment_date, loan_approval_date)
      NOT BETWEEN 0 AND 90
    mustBe: 0
  - id: ccar_appraisal_exceeds_loan
    type: sql
    query: |
      SELECT COUNT(*) FROM mortgage_loans
      WHERE appraised_value <= approved_loan_amount
    mustBe: 0
```


© FirstEigen LLC. All rights reserved.

Where the effort goes

Instead of asking a data engineer to start every table with a blank rules file, the catalog-wide workflow hands them candidate contracts. Human time goes where business meaning or enforcement risk is high. SMEs review the highest-value rules, and engineers spend their time on exception handling and controls, and on integrating with the runtime, instead of rewriting common expectations. You can measure coverage per asset and per domain, and across the whole enterprise.

Discover once, approve explicitly, enforce every run

Discovery belongs in asset onboarding and in scheduled quality reviews, and whenever policies or schemas change. At runtime, load immutable, approved versions. Specialized-model inference shouldn't sit in the path of every row or micro-batch. Keeping the two apart makes the pipeline auditable and keeps its behavior predictable.

© FirstEigen LLC. All rights reserved.

Catalog-wide generation in pseudocode

The pseudocode below is illustrative. It is not a tested catalog iteration or ODCS export API, so check the methods and integrations against the SDK release you're

```
# Conceptual orchestration. Illustrative only, not a released DataBuck API
for asset in supported_catalog.list_assets():
    context = gather_authorized_metadata_and_policies(asset)
    candidates = databuck.discover(asset.dataframe, context=context)
    reviewed = request_owner_approval(candidates, asset)
    contract = render_odcs_compatible_contract(asset, reviewed)
    validate_odcs_schema(contract)
    publish_versioned_contract(contract)

# At pipeline runtime, load an approved contract version
# and compile its rules to the supported execution engine.
```

The loop shows the shape of the work. `list_assets`, `render_odcs_compatible_contract` and the other helpers are placeholders, not public SDK functions.

© FirstEigen LLC. All rights reserved.

Enforcement policies for multi-row and temporal rules

Severity / result	Recommended behavior	Examples
Critical data integrity	Quarantine invalid records or stop job when partial output is unsafe	Receipt before shipment; required lifecycle date missing
Policy control / investigation	Preserve transactions and emit control alert with supporting grouped evidence	AML multi-transaction threshold exceeded
Warning / information	Continue only under approved policy; record exceptions and notify	Unexpected time-window drift or emerging relationships

Never silently drop transactions just because an AML threshold was exceeded. A threshold usually means "review this," not "discard this." For critical quality violations, keep the rejected records, or secure references to them, with the rule identifiers that rejected them and the data asset they came from. Record the event time and decision status too, and keep a way to replay. Set materiality thresholds past which the job fails instead of publishing incomplete results.

Simple temporal comparisons may compile to native Spark predicates. Multi-row rules may need aggregations or windowing, and sometimes entity resolution or joins across datasets, so plan for them. Watch how incremental processing handles late-arriving records and correction events, or you'll generate false alerts. And validate the output again before it reaches a production destination.

© FirstEigen LLC. All rights reserved.

Rolling it out and measuring it

Start with five to ten critical assets in a domain where the business meaning is documented. Connect whatever metadata you have and run candidate discovery, then review real examples with the data owners. Next, generate ODCS-compatible artifacts for each supported asset. Compare the suggested checks with your existing rule inventory to quantify the business conditions you weren't covering.

Start with coverage: how many catalog assets are covered, and what share of candidate rules get approved. Count the newly identified business rules too. Then look at effort and quality: engineer hours per approved contract and the false-positive rate, along with how complete the provenance is and how many contract changes were reviewed. Last, watch cost and payoff. Validation overhead in the pipeline is the cost. Defect escapes and downstream rework avoided are the payoff. Benchmark performance in your own runtime. Be skeptical of anyone, us included, who quotes millisecond latency or universal savings without one.

© FirstEigen LLC. All rights reserved.

Conclusion: the standard is ready, implementation is the hard part

Open standards make data contracts portable and easy to understand. They don't find the rules or the owners, and they don't do the governance for you. DataBuck SDK is aimed at that gap. It walks the cataloged assets to discover business-relevant expectations and drafts ODCS-compatible contracts from them. The approved rules are then available to enforce inside the pipeline.

The AML, automotive and CCAR examples point at where data quality has to go. It has to understand transactions across rows and the sequence of business events, and it has to respect the timing constraints that real policies impose. Find those relationships across the catalog and turn them into approved contracts. Then enforce them before bad data becomes somebody else's incident.

References:

FirstEigen, DataBuck SDK (public preview and edition overview)

Open Data Contract Standard specification

Data Contract CLI, Open Data Contract Standard guidance

About FirstEigen & DataBuck

FirstEigen is an enterprise AI and data quality company. Its flagship platform, DataBuck, automates data quality rule discovery, validation, monitoring, reconciliation, and remediation in one platform. DataBuck uses a domain-specific small Language model (SLM) to generate context-aware rules based on business policies and regulations. The DataBuck SDK integrates with Python, Spark, Databricks, and cloud data pipelines to support reliable enterprise data and AI workflows.
Learn more: **www.firsteigen.com/databuck**

© FirstEigen LLC. All rights reserved.