Volume-05 Issue 07, July -2021 ISSN: 2456-9348 (Online)
SJIF Impact Factor: 5.004

iJETRM

INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

DATA QUALITY AS CODE: BUILDING SCALABLE VALIDATION FRAMEWORKS IN SCALA FOR CROSS-TEAM ADOPTION

Santosh Kumar Maddali
Sr. Data Engineer

ABSTRACT:
Data quality defects in large analytics platforms are usually discovered downstream, after incorrect figures have already reached executive dashboards, machine learning features, and regulatory filings. The root cause is rarely the absence of checks. It is that validation logic lives outside the software engineering lifecycle, encoded as ad hoc scheduler queries, notebook cells, and tribal knowledge that is never reviewed, versioned, or tested. This paper presents Data Quality as Code (DQaC), an approach that treats validation as first-class software, and describes the design and reference implementation of a validation framework written in Scala on Apache Spark, together with a proposed organizational adoption model. The framework contributes four technical elements: a closed rule algebra that composes primitive assertions into reusable rule sets; typed schema contracts that bind expectations to dataset identity under semantic versioning; applicative error accumulation that reports every violation in one pass rather than aborting at the first failure; and an execution planner that fuses independent predicates into a single distributed scan. A federated adoption model addresses the sociotechnical barriers that normally stop shared tooling at the team boundary, combining distributed contract ownership with centrally enforced continuous integration gates. We analyze the approach qualitatively: which failure classes each rule family can detect and at what stage, why fusion makes validation cost track pass count rather than rule count, and which organizational conditions the adoption model requires in order to hold. This is a design and strategy contribution rather than a measurement study, and we state explicitly what an empirical evaluation would need to establish.

Keywords:
Apache Spark, continuous integration, data contracts, data engineering, data quality, data validation, functional programming, Scala, schema evolution, software reuse.

I. INTRODUCTION

Enterprise analytics platforms now routinely ingest from several hundred upstream systems, materialize thousands of derived tables, and serve consumers whose tolerance for error varies by orders of magnitude. A marketing attribution dashboard can absorb a fractional inaccuracy without consequence. A capital adequacy report cannot. The same physical pipeline frequently feeds both. This asymmetry, combined with the sheer depth of modern dependency graphs, makes the economics of data defects unforgiving: the cost of a defect grows superlinearly with the distance between the point at which it is introduced and the point at which it is observed.

Practitioners understand this. Most organizations of any scale have accumulated a considerable body of validation logic. The problem is where that logic lives. It is characteristically distributed across scheduler configuration files, ad hoc SQL executed by a nightly job, assertions embedded in notebook cells that never leave a personal workspace, and reconciliation spreadsheets maintained by analysts. Logic in these forms shares a common defect: it is invisible to the software engineering lifecycle. It is not code reviewed, so errors in the checks themselves go unnoticed. It is not version controlled, so nobody can say when an expectation changed or why. It is not tested, so a check that silently stopped evaluating any rows continues to report success indefinitely. It is not discoverable, so three teams independently write three subtly different definitions of the same business rule.

A second and less discussed problem is enforcement. Even where checks exist and are well written, they are commonly advisory. A failing check emits an alert into a channel that is already saturated with alerts, and the pipeline proceeds to publish. Detection without enforcement converts a data correctness problem into an attention allocation problem, which is a poor trade.

A third problem is organizational rather than technical. Validation frameworks are built constantly, and most of them work. What they usually fail to do is cross a team boundary. A framework authored by the platform team encodes the platform team's assumptions about naming, storage layout, deployment, and severity. Adopting it obliges another team to accept those assumptions wholesale, and the perceived cost of that coupling typically exceeds the perceived cost of writing thirty lines of bespoke SQL. The result is a landscape of parallel, partially overlapping, mutually invisible quality efforts.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

This paper argues that these three problems have a single response, which we call Data Quality as Code. The premise is that validation logic should be subject to precisely the same disciplines as application code: it should be written in a statically typed general purpose language, stored in version control, code reviewed, unit tested, released under semantic versions, and enforced automatically in continuous integration. We describe a framework built on this premise in Scala over Apache Spark, and we propose a federated model for rolling it out across a large engineering organization.

Scala is not incidental to the argument. Three properties of the language bear directly on the problem. Its algebraic data types allow a rule to be represented as a value that can be inspected, transformed, and combined before it is executed, which is what makes an execution planner possible. Its type system allows a schema contract to be checked at compile time rather than at three in the morning. And the applicative abstractions available to it through functional programming libraries such as Cats provide a principled mechanism for accumulating multiple independent failures, which matters because an engineer debugging a broken dataset wants the complete list of what is wrong, not the first item on it.

Contributions. This paper makes the following contributions:

C1. *A rule algebra* closed under conjunction, disjunction, and scope restriction, with negation defined on the subset of rules for which complementation has an unambiguous meaning, permitting complex expectations to be assembled from a small primitive basis and reused across datasets (Section IV-B).

C2. *Typed schema contracts* that bind a set of expectations to a dataset identity, carry a semantic version, and support a compatibility classification that makes the blast radius of a contract change explicit before it is merged (Section IV-C).

C3. *Applicative accumulation semantics* that evaluate all independent rules and return the complete violation set, together with a monadic escape for rules with genuine sequential dependencies (Section IV-D).

C4. *A predicate fusion planner* that compiles a rule set into a single distributed scan, holding validation overhead close to constant in the number of rules rather than linear in it (Section IV-E).

C5. *A federated adoption model* that separates contract ownership, which is distributed to the teams that own the data, from gate enforcement, which is centralized, and thereby avoids both the bottleneck of a central quality team and the drift of purely local ownership (Section VI).

C6. *A structured analysis* of the design, covering detection coverage by construction, the cost behavior that fusion produces, and the organizational preconditions on which the adoption model depends (Section VII).

The remainder of the paper is organized as follows. Section II reviews prior work on data quality dimensions, automated verification systems, and the functional programming foundations we rely on. Section III characterizes the problem through a failure taxonomy and derives requirements. Section IV presents the framework design and Section V its implementation in Scala. Section VI describes the adoption model. Section VII evaluates the framework, Section VIII discusses lessons learned, Section IX addresses threats to validity, and Section X concludes.

II. Background and Related Work

A. Data Quality Dimensions

The conceptual vocabulary for data quality predates the systems that motivate this work. Wang and Strong established empirically that data consumers evaluate quality along multiple dimensions rather than a single accuracy axis, grouping fifteen dimensions into intrinsic, contextual, representational, and accessibility categories [1]. Redman treated quality as a manageable property of an information production process rather than an attribute of a static artifact [2]. Rahm and Do provided the canonical taxonomy of data anomalies, separating single source from multi source problems and schema level from instance level defects [3]. Batini et al. surveyed assessment and improvement methodologies and observed that most operate as periodic audits rather than as continuous controls [4]. Our framework inherits this vocabulary directly: the rule taxonomy in Section III maps onto the completeness, consistency, timeliness, and validity dimensions of this literature.

The distinction that matters most for our purposes is between quality as measurement and quality as control. The methodological literature is predominantly concerned with the former. A control regime requires that measurement be continuous, automated, and coupled to an enforcement action, which is a systems problem rather than a methodological one.

B. Automated Verification Systems

Deequ is the closest system to our work and the most direct influence on it [5]. Built at Amazon on Spark, it offers a declarative API in which users state constraints over a dataset, and it computes the supporting metrics efficiently by sharing aggregation passes. A companion paper describes its use for unit testing data, drawing the analogy to software unit testing that we extend here [6]. Deequ demonstrates convincingly that constraint checking

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

can be made efficient at scale. Our framework departs from it in three respects. First, Deequ constraints are values assembled at runtime, whereas we bind expectations to a typed contract that is checked when the project compiles. Second, Deequ leaves the disposition of a failure to the caller, whereas we make severity and quarantine part of the rule definition and part of the continuous integration gate. Third, and most importantly for this paper, Deequ is a library rather than an adoption model, and the difficulty we address is chiefly one of organizational diffusion.

TensorFlow Data Validation, described by Polyzotis et al., addresses validation for machine learning pipelines by inferring a schema from training data and then detecting training and serving skew and distributional drift against it [7]. It is part of the wider TFX platform [8]. The inference oriented posture is well suited to feature data whose expected distribution is not known analytically, and less well suited to transactional data where expectations are stated by contract or regulation. We adopt its drift detection ideas in the metric repository (Section IV-G) while retaining declarative rules as the primary mechanism.

Polyzotis et al. surveyed data management challenges specific to production machine learning and identified validation as a persistent gap [9]. Sculley et al. framed unvalidated data dependencies as a principal source of hidden technical debt, arguing that data dependencies cost more than code dependencies precisely because no compiler enforces them [10]. Breck et al. proposed the ML Test Score, a rubric in which several of the highest weighted items concern data validation [11]. Rukat et al. later extended the Deequ line of work toward automated quality management with learned rather than declared expectations [12]. Collectively this body of work establishes the case for validation. It does not resolve how a validation capability becomes an organizational default rather than a local practice.

C. Data Cleaning and Repair

A related literature treats detection as a precursor to repair. Chu et al. surveyed data cleaning and its emerging challenges [13], and Ilyas and Chu provide a comprehensive treatment of error detection and repair techniques [14]. Our framework deliberately stops at detection and disposition. In a regulated enterprise setting an automatic repair is an unattributed mutation of the record of truth, and the operational risk it carries generally exceeds the risk of a blocked pipeline. We quarantine and report rather than correct.

D. Functional Programming Foundations

The design of the rule algebra rests on well established functional programming results. Odersky and Rompf describe the unification of functional and object oriented programming in Scala that makes it practical to express an embedded domain specific language with a type safe interface over an object oriented runtime [15], and the language reference provides the pattern matching and algebraic data type facilities we rely on [16]. McBride and Paterson introduced applicative functors and showed that they permit effects to be combined independently, in contrast to monadic sequencing where a later computation may depend on an earlier one [17]. This distinction is the theoretical basis for our accumulation semantics: independent rules form an applicative and accumulate all failures, while dependent rules require the monadic interface of Wadler [18] and short circuit. Neither abstraction is in the Scala standard library; both are supplied by the Cats library [19], which provides the Validated type and applicative syntax used in Section V. Making this distinction explicit in the API prevents the common defect in which a validation harness reports one error at a time and forces engineers into an expensive iterative debugging loop.

E. Execution Substrate

Spark provides the distributed execution model [20], with resilient distributed datasets supplying the fault tolerance and lineage properties we depend on for deterministic re-execution of a validation job [21]. Spark SQL and its Catalyst optimizer supply the relational representation over which our planner performs predicate fusion [22]. Transactional table formats such as Delta Lake give us the snapshot isolation needed to validate a specific committed version of a dataset rather than a moving target, and the time travel capability that makes retrospective validation of historical partitions feasible [23]. The underlying data parallel paradigm derives from MapReduce [24]. Kleppmann provides the systems framing for the consistency and evolution concerns that motivate schema contracts [25], and Wickham's normalization conventions inform the structural rules in our primitive basis [26]. Hellerstein et al. argue for a queryable substrate of context about datasets rather than isolated per system metadata [27], a position our metric repository adopts.

F. Positioning

Table I positions our framework against the alternatives that a team realistically chooses among. The distinguishing properties are not individually novel. The contribution is the combination, and specifically the coupling of a typed, composable rule representation to an enforcement and ownership model designed for diffusion across team boundaries.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

TABLE I COMPARISON OF DATA VALIDATION APPROACHES

Approach	Typed	Compos.	Fused	Gated	Fed.
Ad hoc SQL checks	No	No	No	No	No
Notebook assertions	No	No	No	No	No
Constraint library [5], [6]	Partial	Yes	Yes	No	No
Inferred schema [7]	Partial	No	Yes	Partial	No
Config driven engines	No	Partial	Partial	Partial	Partial
DQaC (this work)	Yes	Yes	Yes	Yes	Yes

Typed: compile-time verification of column reference and type. Compos.: rules combine under logical connectives. Fused: all rules discharged in a minimal set of scans. Gated: outcomes block promotion. Fed.: contract ownership distributed under centrally enforced gates.

III. PROBLEM CHARACTERIZATION

A. A Taxonomy of Observed Failures

To ground the requirements we synthesize a taxonomy of failure classes from the data quality and production machine learning literature, principally the anomaly classification of Rahm and Do [3], the cleaning survey of Chu et al. [13], and the validation concerns reported for production pipelines [7], [11]. Six classes recur across these sources. We describe them because the taxonomy determines what the framework must be able to express, and because it organizes the coverage analysis of Section VII.

F1. Schema drift. An upstream producer adds, removes, renames, widens, or retypes a column without notice. Downstream jobs written against positional or permissive readers either fail loudly, which is the fortunate case, or silently coerce values and continue. Retyping is the most damaging variant because a numeric column read as a string will often continue to satisfy naive checks while breaking every aggregation over it.

F2. Semantic drift. The schema is unchanged but the meaning of a value changes. A status enumeration acquires a seventh member. A currency amount changes denomination from dollars to cents. A field previously populated for all records begins to be populated only for a subset. This class is invisible to any structural check and is the hardest to detect without a stated expectation.

F3. Referential breakage. Foreign key relationships that hold by convention rather than by constraint degrade. Orphaned facts accumulate against a dimension that was reloaded out of order. Because most analytical stores do not enforce referential integrity, the breakage surfaces as a quiet undercount in a join rather than as an error.

F4. Volumetric anomaly. Row counts, distinct key counts, or aggregate magnitudes depart from their historical envelope. A partial upstream load that delivers sixty percent of expected volume is the canonical instance, and it is dangerous precisely because the resulting dataset is internally consistent and passes every structural and referential check.

F5. Temporal skew. Data arrives late, arrives out of order, or carries timestamps inconsistent with the partition it lands in. Downstream windowed aggregations then compute over incomplete windows and produce results that are correct with respect to the data present and wrong with respect to the question asked.

F6. Silent default substitution. A producer substitutes a sentinel for a missing value: zero for an unknown amount, the Unix epoch for an unknown date, an empty string for an absent identifier. Null checks pass. Aggregations are corrupted. This class is repeatedly identified as among the most costly, because the defect is indistinguishable from valid data without an explicit domain rule.

B. Requirements

From the taxonomy and from the adoption barriers described in Section I we derive eight requirements.

R1 Expressiveness. The rule language must express all six failure classes, including stateful comparisons against historical values, which F4 and F5 require.

R2 Composability. Rules must compose under logical connectives and scope restriction so that a shared library of organizational rules can be specialized locally without being copied.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

R3 Type safety. A rule that references a column absent from its contract schema, or that applies a numeric predicate to a string column, must fail at compile time. This is a guarantee about the internal consistency of the contract, not about the data: whether the live dataset conforms to the declared schema is a separate obligation, discharged against the catalog before any scan. A validation framework whose own logic fails at runtime inverts its purpose.

R4 Bounded overhead. Validation cost must be a small and predictable fraction of pipeline cost, and must not grow linearly with the number of rules, or teams will delete rules to meet service level objectives.

R5 Complete diagnostics. A single execution must report every independent violation, with sufficient context, sample records, and rule provenance to permit diagnosis without re-execution.

R6 Enforcement. Rule outcomes must be able to block promotion of a dataset or a code change, not merely emit a notification.

R7 Evolvability. Contracts must version explicitly, and the compatibility class of a proposed change must be computable so that breaking changes are visible in review.

R8 Low adoption cost. The marginal cost for a new team to onboard must be measured in hours, and adoption must not require the team to surrender control of its own quality expectations.

R3, R5, and R8 are the requirements that most sharply differentiate this work from the alternatives in Table I, and they are the requirements that most directly motivate the choice of Scala.

IV. Framework Design

A. Design Principles

Five principles govern the design and are worth stating explicitly because most of the detailed decisions follow from them.

P1 Rules are values, not statements. A rule is a data structure that describes an intent to check something. It is not the check itself. This separation is what permits inspection, rewriting, and fusion before execution, and it is the single decision from which most of the framework's leverage derives.

P2 The compiler is the first reviewer. Any class of error that can be moved from runtime to compile time should be.

P3 Detect and dispose, do not repair. The framework never mutates data. It classifies, quarantines, reports, and blocks.

P4 Failure is a first-class output. A validation run produces a structured, machine-readable result whether it succeeds or fails. Exceptions are reserved for defects in the framework itself.

P5 Ownership is local, enforcement is central. Teams author and own their contracts. The platform owns the gate that evaluates them. Neither can be inverted without reintroducing a failure mode described in Section I.

B. The Rule Algebra

Let D denote a dataset with schema S. A rule r is a function from D to a validation outcome, but it is represented as a term in an algebra rather than as an opaque function, so that the planner can analyze it.

The primitive basis comprises seven families of constructors, of which Listing 1 shows one representative each. Structural rules assert properties of the schema: presence of a column, its type, and its nullability. Value rules assert predicates over individual records: range membership, set membership, regular expression conformance, and null density. Aggregate rules assert properties of the dataset as a whole: cardinality bounds, distinct count bounds, and uniqueness of a key. Relational rules assert properties across datasets: referential inclusion of a key set in a reference set. Temporal rules assert freshness and monotonicity of an event time column. Statistical rules assert that a summary statistic lies within a tolerance of a historical baseline. Custom rules admit an arbitrary user-supplied Spark column expression, which preserves expressiveness without weakening the type discipline, because the expression is still constructed through the typed column accessors of the contract.

Over this basis we define four combinators. Conjunction, disjunction, and scope restriction are total: each accepts any rule and yields a rule. Scope restriction, written as a where combinator, restricts a rule to the subset of records satisfying a predicate, which is the mechanism by which a shared organizational rule is specialized to a local dataset without duplication. Negation is deliberately partial. It is defined on the structural and value families, where the complement of a predicate over a record is well defined, and undefined on the aggregate, relational, and statistical families, where the complement of a bound or of a baseline has no useful interpretation. Listing 1 encodes this restriction in the type system rather than in documentation: the negation constructor accepts a Negatable rule rather than any Rule, so negating an aggregate rule does not compile.

The algebra is closed under conjunction, disjunction, and scope restriction, and closed under negation on the negatable subset. This yields two properties that matter in practice. First, an arbitrarily complex expectation

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

reduces to a tree over the primitive basis, so the planner needs to understand only the leaf families. Second, a rule set can be treated as a value that is passed, stored, diffed, and inherited. Contract inheritance is simply conjunction of the parent rule set with the child rule set, and the difference between two contract versions is a structural diff over two trees, which is what makes the compatibility classification of Section IV-C computable rather than a matter of judgment.

Rules carry three pieces of metadata beyond their logical content: a stable identifier used for metric time series continuity across contract versions, a severity, and a free text rationale. The identifier is important and easily overlooked. If a rule identifier changes when a contract is edited, the historical metric series for that rule is severed, and every statistical rule that depends on a baseline silently loses its history.

C. Schema Contracts

A contract binds a rule set to a dataset identity. It carries the fully qualified dataset name, a typed schema, a semantic version, an owning team, an escalation contact, and the rule set itself. Contracts are Scala source files stored in the repository that owns the producing pipeline, so a change to an expectation is a pull request against code and is reviewed as such.

The typed schema is what discharges R3. Each column is declared once with its Scala type, and rules reference columns through accessors generated from that declaration rather than through string literals. A rule that references a column not in the declared schema does not compile, and a rule that applies a numeric range to a string column does not compile. The guarantee is one of internal consistency between the rules and the schema the contract declares. Whether the dataset actually presented at runtime matches that schema is checked separately, against the catalog, before any scan is performed. This is intended to eliminate a category of defect in the validation logic itself that is otherwise invisible, because a validation job that fails to evaluate is operationally indistinguishable from one that passes.

Contracts version under semantic versioning conventions [28], and the framework computes the compatibility class of a proposed change by diffing the rule trees. Two directions must be classified separately, because they affect different parties. Producer compatibility asks whether a pipeline that satisfied the previous contract still satisfies the new one: adding a required column or tightening a constraint breaks it, relaxing a constraint does not. Consumer compatibility asks whether a reader of the dataset still finds what it expects: removing a column or admitting a new enumeration member breaks it, adding an optional column does not. A change is major if it breaks either direction, and the classifier reports which. A patch is reserved for changes with no behavioral effect whatever, such as editing a rationale. Severity changes are deliberately excluded from patch status, because severity determines whether data is published: raising a severity is major, and lowering one is minor. The classification is emitted into the pull request, so a reviewer sees that a change is breaking before merging rather than after. This addresses R7 and, more importantly, converts a class of coordination failure into a code review conversation.

D. Accumulation Semantics

Most validation harnesses short circuit. The first failing check aborts the run, the engineer fixes it, reruns, discovers the second failure, and repeats. For a dataset with several independent defects this loop costs one full pipeline execution per defect, which at scale is measured in hours.

The framework instead evaluates independent rules applicatively and accumulates their failures into a nonempty list, following McBride and Paterson [17]. The consequence is that one execution yields the complete violation set. Rules that genuinely depend on one another, for example a referential rule that is meaningless if the key column is absent, are sequenced monadically [18] and short circuit correctly. The distinction is expressed in the API rather than left to convention, so an engineer cannot accidentally serialize independent checks.

This is a small design decision with a disproportionate operational effect. Complete diagnostics (R5) turn a multi hour iterative debugging session into a single triage pass over a structured report.

E. Execution Planning and Predicate Fusion

Naive execution evaluates each rule as a separate Spark action, so a contract with sixty rules performs sixty scans. This violates R4 and, more damagingly, creates an incentive for teams to reduce coverage in order to meet latency targets. The planner therefore compiles a rule tree into a minimal set of physical passes.

Compilation proceeds in four phases. Normalization rewrites the tree into conjunctive normal form and lifts scope restrictions into guarded predicates. Classification partitions rules by the access pattern each requires: schema only rules need no scan and are discharged against the catalog; record-local rules need one pass; aggregate rules need one pass with accumulators; relational rules need a join; and statistical rules need one pass plus a lookup against the metric repository. Fusion then merges all record-local and aggregate rules into a single pass, expressing each as a conditional accumulator over the same scan, so that the marginal cost of an additional rule is one column

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

expression rather than one scan. Scheduling finally orders the remaining passes, executing the free schema checks first so that a structural failure aborts before any expensive work is performed.

The effect, analyzed in Section VII, is that overhead becomes a function of pass count rather than rule count. A contract with sixty rules typically compiles to two or three passes. Fig. 1 summarizes the layered architecture and the boundaries it draws between contract ownership, execution, enforcement, and evidence.

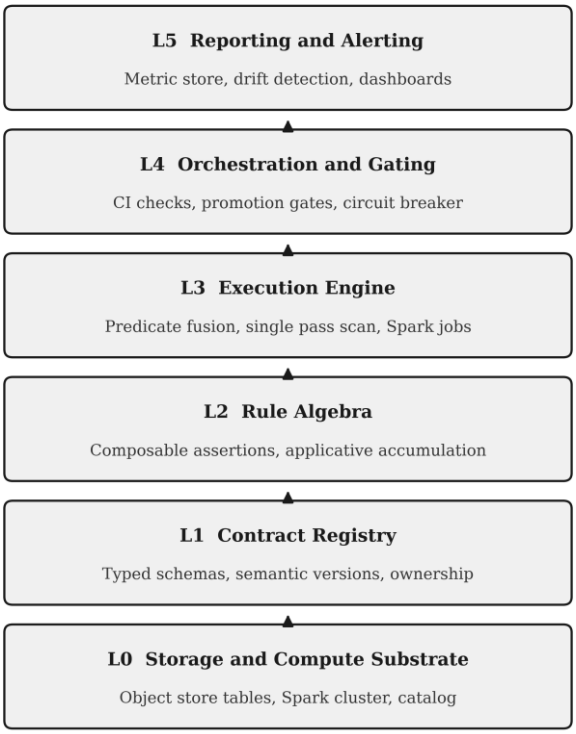

Fig. 1. Layered architecture of the framework. Declarative rules enter at the contract registry and rule algebra layers, are compiled by the execution engine into a minimal set of Spark passes, are enforced by the orchestration and gating layer, and emit structured evidence to the reporting layer. Each layer depends only on the layer beneath it.

F. Severity and Disposition

Every rule carries one of four severities, and severity determines disposition rather than merely message formatting. An informational rule records a metric and takes no action. A warning rule records a metric and notifies the owning team asynchronously. A critical rule aborts the write entirely and leaves the previous version of the dataset as the current one. Error severity is the case that requires care, because its meaning depends on what the rule is about.

A row-level rule, such as a range or set membership check, identifies specific violating records. For these, an error routes the violating records to a quarantine table, allows conforming records to proceed, and annotates the output partition with the quarantine count. A dataset-level rule, such as a row count bound, a schema assertion, or a distributional comparison, has no violating record to divert, since its subject is the dataset as a whole. For these, an error withholds the entire partition. Treating the two uniformly is a modeling error that is easy to miss until a framework is asked to quarantine the rows responsible for a row count anomaly and has no answer to give. Table III in Section VII states both dispositions.

Quarantine is commonly one of the most debated policy choices during adoption. Partial publication with an explicit and queryable record of what was excluded is, for most analytical workloads, preferable both to publishing corrupt data and to publishing nothing. It is not preferable for financial reconciliation, where a partial dataset is worse than an absent one, and those contracts use critical severity throughout. Making this a per-rule property rather than a framework-wide policy is what allows one framework to serve both cases.

A circuit breaker complements severity, and it deliberately does not alter disposition. If the same rule fails on n consecutive executions, the framework raises the incident priority of the notification and suppresses duplicates.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

It does not promote a warning into an error. Severity is versioned as part of the contract and determines whether data reaches consumers, so allowing a runtime counter to change it would let enforcement behavior drift without a pull request, defeating the versioning discipline of Section IV-C. Escalating attention rather than enforcement still addresses the failure mode in which a chronically failing warning rule trains its owners to ignore the channel it reports to, without introducing an unversioned change to production behavior.

G. Metric Repository and Drift Detection

Every rule execution emits a metric record: rule identifier, contract version, dataset version, evaluation timestamp, records evaluated, records violating, and the computed statistic where applicable. These records are appended to a transactional table [23] and constitute the substrate for three capabilities.

First, statistical rules resolve their baselines from it, so a rule can assert that today's null rate for a column lies within three standard deviations of its trailing thirty-day mean without the author computing that baseline. Second, drift detection operates over it independently of any declared rule, applying the distributional comparison techniques of Polyzotis et al. [7] to surface changes nobody thought to assert. Third, it provides the evidence trail that audit and regulatory functions require. Such an evidence trail can also support the organizational case for funding data quality controls.

The repository is keyed on rule identifier rather than rule content, which is the reason for the stability requirement noted in Section IV-B.

V. Reference Implementation

A reference implementation in Scala 2.12 against Spark 3.0, packaged with sbt, demonstrates that the design is expressible without extending the language or the engine. It is organized into four modules: core, which contains the algebra and contract model and has no Spark dependency; engine, which contains the planner and Spark execution; registry, which contains the metric repository and drift detection; and gate, which integrates with continuous integration. The absence of a Spark dependency in core is deliberate, since it permits rule composition and contract compatibility analysis to be unit tested in milliseconds without a session.

A. Rule Definition

Listing 1 shows the algebraic data type at the center of the design. Representing rules as a sealed trait hierarchy gives exhaustive pattern matching in the planner, so a new primitive cannot be added without the compiler identifying every site that must handle it.

Listing 1. Rule algebra as a sealed algebraic data type.

```
sealed trait Rule {
  def id: RuleId
  def severity: Severity
  def rationale: String
}

// Only these may be negated: complementation is
// well defined for schema and record predicates,
// not for bounds, joins or baselines.
sealed trait Negatable extends Rule

object Rule {

  final case class ColumnExists(
      id: RuleId, column: ColumnRef[_],
      severity: Severity, rationale: String)
    extends Negatable

  final case class InRange[A: Numeric](
      id: RuleId, column: ColumnRef[A],
      lower: A, upper: A,
      severity: Severity, rationale: String)
```


iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
    extends Negatable

  final case class UniqueKey(
      id: RuleId, columns: Seq[ColumnRef[_]],
      severity: Severity, rationale: String)
    extends Rule

  final case class ReferentialInclusion(
      id: RuleId, local: ColumnRef[_],
      reference: DatasetRef, refKey: String,
      severity: Severity, rationale: String)
    extends Rule

  final case class Freshness(
      id: RuleId, eventTime: ColumnRef[Timestamp],
      maxLag: Duration,
      severity: Severity, rationale: String)
    extends Rule

  final case class WithinBaseline(
      id: RuleId, statistic: Statistic,
      sigma: Double, window: Int,
      severity: Severity, rationale: String)
    extends Rule

  final case class Custom(
      id: RuleId, predicate: TypedPredicate,
      severity: Severity, rationale: String)
    extends Rule

  sealed trait Combinator extends Rule {
    def children: Seq[Rule]
    lazy val id        = RuleId.derived(children)
    lazy val severity  = children.map(_.severity).max
    lazy val rationale =
      children.map(_.rationale).mkString("; ")
  }

  final case class And(l: Rule, r: Rule)
    extends Combinator { val children = Seq(l, r) }
  final case class Or(l: Rule, r: Rule)
    extends Combinator { val children = Seq(l, r) }
  final case class Not(inner: Negatable)
    extends Combinator { val children = Seq(inner) }
  final case class Where(
      inner: Rule, guard: TypedPredicate)
    extends Combinator { val children = Seq(inner) }
}
```


iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

The ColumnRef type parameter makes an InRange rule over a string column a compile error rather than a runtime one. Combinators derive their own metadata from their children, so every Rule has an identifier and a severity. Not accepts only a Negatable, so negating an aggregate or relational rule is rejected by the compiler rather than at runtime.

B. Typed Contracts

Listing 2 shows a contract as an engineer would write it, using fictitious names. The schema declaration generates typed column references, and every rule below it is checked against that declaration at compile time. The contract is an ordinary Scala object, so it is reviewable, testable, and refactorable with standard tooling.

Listing 2. An illustrative schema contract. Dataset, team, and column names are fictitious.

```
object OrdersContract extends Contract {

  val dataset = DatasetRef(
    "example.retail.orders")
  val version = SemVer(2, 4, 0)
  val owner   = Team("example-team")

  val orderId    = column[String]("order_id")
  val customerId = column[String]("customer_id")
  val amountMinor = column[Long]("amount_minor")
  val currency   = column[String]("currency")
  val placedAt   = column[Timestamp]("placed_at")
  val status     = column[String]("status")

  val rules: RuleSet = RuleSet(

    unique(orderId)
      .critical("Reconciliation key."),

    notNull(customerId, amountMinor,
          currency, placedAt)
      .critical("Required by the ledger."),

    inRange(amountMinor, 1L, 100000000000L)
      .error("Zero is a gateway sentinel."),

    inSet(currency, Iso4217.codes)
      .error("Unknown codes break the FX join."),

    referentialInclusion(
       customerId, CustomersContract.dataset,
       refKey = "customer_id")
      .error("Orphans undercount exposure."),

    freshness(placedAt, maxLag = 6.hours)
      .critical("Intraday risk bound."),

    withinBaseline(RowCount, sigma = 3.0,
                window = 30)
```


iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
      .warning("Detects partial loads."),

    inSet(status, SettlementStatus.all)
      .where(placedAt > startOfQuarter)
      .error("History predates the enum.")
  )
}
```

The notNull and inSet builders belong to the structural and value families of Section IV-B. The where combinator scopes the status enumeration rule to the current quarter, which is how a tightened expectation is introduced without invalidating history.

C. Accumulation

Listing 3 shows the accumulation semantics of Section IV-D. The applicative combination over a Validated type collects all failures, while the monadic for comprehension expresses genuine dependency and short circuits. The Validated type, the validNel constructor, and the mapN syntax come from the Cats library [19], not from the Scala standard library, so this part of the design carries a dependency that a reimplementation would need to satisfy.

Listing 3. Applicative accumulation and monadic sequencing.

```
type Checked[A] = ValidatedNel[Violation, A]

// Independent rules: every failure is reported.
def evaluateIndependent(
    rules: Seq[Rule],
    ctx: ScanContext): Checked[Unit] =
  rules
    .map(r => evaluate(r, ctx))
    .foldLeft(().validNel[Violation]) { (acc, v) =>
      (acc, v).mapN((_, _) => ())
    }

// Dependent rules: later checks need earlier ones.
def evaluateDependent(
    contract: Contract,
    ds: Dataset[Row]): Either[Violation, Report] =
  for {
    schema  <- verifySchema(contract, ds)
    scanCtx <- planScan(contract.rules, schema)
    result  <- executeFused(scanCtx, ds)
  } yield Report(contract, result)
```

Independent rules use the applicative interface; only genuinely sequential checks use the monadic one.

D. Planner

Listing 4 shows the fusion step. Record local and aggregate rules are converted into conditional accumulator expressions over a shared scan, so the marginal cost of a rule is one column expression rather than one Spark action.

Listing 4. Fusing record-local rules into a single pass.

```
def fuse(rules: Seq[Rule],
         ds: Dataset[Row]): Dataset[Row] = {
  val accumulators = rules.collect {
    case r: RecordLocal =>
```


iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
      sum(when(not(toColumn(r)), lit(1L))
          .otherwise(lit(0L)))
        .alias(s"violations_${r.id.value}")
    }
    if (accumulators.isEmpty) ds
    else ds.agg(accumulators.head,
              accumulators.tail: _*)
  }
```

Sixty record-local rules compile to one aggregation over one scan.

E. Continuous Integration Gate

Listing 5 shows the gate invoked from the build. It fails the build on any critical violation, on any error violation exceeding a declared tolerance, and on any contract change classified as breaking whose pull request lacks the required approval. The gate is the mechanism that discharges R6, and it is the component that must be owned centrally under principle P5.

Listing 5. The build gate.

```
 object QualityGate {

   def check(report: Report,
             policy: GatePolicy): GateOutcome =
     report.violations match {

       case vs if vs.exists(_.isCritical) =>
         GateOutcome.Blocked(
           vs.filter(_.isCritical))

       case vs if vs.count(_.isError) >
                  policy.errorTolerance =>
         GateOutcome.Blocked(vs)

       case vs if report.isBreaking &&
                  !report.hasApproval =>
         GateOutcome.BlockedOnReview(report)

       case vs =>
         GateOutcome.Passed(warnings = vs)
     }
 }
```

Enforcement is centralized even though contract authorship is not.

F. Testing the Validation Logic

A framework that asserts the correctness of data must itself be held to a higher standard than the pipelines it guards, since a silent failure in the framework removes a control without removing the belief that the control exists. We therefore apply conventional verification practice [29] with three additions. Property based tests generate random rule trees and assert the algebraic laws, in particular that fusion preserves semantics and that contract diffing is symmetric under inversion. Golden datasets with deliberately injected defects from each class in Section III-A assert that each rule type detects what it claims to. Mutation testing over the planner confirms that the test suite fails when fusion is perturbed. The third is the practice most likely to expose planner defects that the first two miss, since a fusion bug that preserves types will satisfy both property and golden dataset tests.

VI. Cross-Team Adoption Model

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

The technical design described so far is necessary but not sufficient. A framework that is correct, fast, and expressive will still fail to diffuse if the organizational model around it is wrong. This section sets out the adoption model the design assumes, and the conditions under which we expect it to hold.

A. Federated Ownership

Two organizational arrangements are common and both have predictable failure modes. Under central ownership a quality team authors all contracts. That team becomes a bottleneck, lacks the domain knowledge to state meaningful expectations for datasets it does not own, and is structurally positioned as an obstacle. Under purely local ownership each team owns everything, which produces divergent conventions, no shared rule library, and no consistent enforcement.

The federated model separates two concerns that are usually conflated. Contract authorship and rule content belong to the team that owns the producing pipeline, because that team alone knows what the data means. The gate, the rule algebra, the shared rule library, the metric repository, and the compatibility classifier belong to a central owner, because these must be uniform for the system to have any aggregate meaning. Principle P5 is the compressed statement of this arrangement.

Conway observed that a system's structure mirrors the communication structure of the organization that produces it [30]. The federated model is a deliberate application of that observation rather than an accident of it: contract boundaries are placed at team boundaries, so a contract change requires no cross team negotiation unless it is a breaking change, in which case the compatibility classifier makes the negotiation explicit and routes it through review.

B. Contract Versioning as a Coordination Protocol

The compatibility classification of Section IV-C serves an organizational function beyond its technical one. Without it, a producer tightening an expectation has no reliable way to know which consumers will break, and consumers have no way to detect that a contract has changed at all. Coordination then happens by electronic mail, which is to say it frequently does not happen.

With classification, a tightened constraint is automatically labeled a major change, the pull request is annotated with the list of registered consumers, and merge requires acknowledgment from consumer owners. This is not a technical novelty. Its value is that it converts an unreliable social protocol into a mechanically enforced one, which is a recurring theme of this work.

C. Gates and the Paved Road

Enforcement applies at two points. The build gate of Listing 5 blocks merge of code whose contract changes are breaking and unacknowledged. The runtime gate blocks promotion of a dataset whose critical rules failed. Both are necessary because they catch different failures: the build gate catches a change to the expectation, and the runtime gate catches a change to the data.

Enforcement alone produces resentment. It must be paired with what the deployment literature calls a paved road [31], meaning that the enforced path is also the easiest path. Three elements make it one. A project template generates a contract skeleton from an existing dataset's inferred schema, so onboarding begins with a working contract rather than an empty file. A shared library supplies rules for organization wide concepts such as currency codes, identifier formats, and standard enumerations, so common expectations are imported rather than reimplemented. A local runner executes a contract against a sample quickly enough that the authoring loop stays interactive. R8 is discharged only if all three hold; a framework that enforces without paving will be routed around.

D. Sequencing the Rollout

Rollout order is a design variable rather than an administrative detail. We propose four stages. First, a small number of teams with acute and visible quality problems adopt in advisory mode with no gates, which produces evidence rather than compliance. Second, gates are enabled for those teams only, establishing that enforcement is survivable. Third, adoption opens to volunteers, with the first cohort's experience as the argument. Fourth, and only once voluntary adoption has reached a critical mass, contracts become mandatory for datasets in the highest criticality tier.

The sequencing principle is that mandate should follow demonstration. Reversing the order, which is the more common instinct, produces contracts written to satisfy an audit rather than to detect defects. Such contracts are worse than none, because they create a false assurance that a control exists.

VII. ANALYSIS

This section examines four questions analytically. Q1: which failure classes can the rule families detect, and at what stage? Q2: what cost behavior does the fusion planner produce, and why? Q3: how does severity map to disposition, and does that mapping hold for every kind of rule? Q4: what organizational conditions does the

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

adoption model require? We deliberately do not report measurements. The design is presented as a strategy to be implemented and evaluated, and Section IX states what such an evaluation would need to establish.

A. Q1: Detection Coverage by Construction

Table II maps each failure class of Section III-A to the rule family that addresses it, the stage at which detection occurs, and whether detection requires a rule written for the specific field or property at issue. The mapping follows from the design rather than from observation, which is what makes it assertable without measurement.

TABLE II FAILURE CLASSES AND THE RULE FAMILIES THAT ADDRESS THEM

Failure class	Rule family	Stage	Explicit rule
F1 Schema drift	Structural	Catalog	No
F2 Semantic drift	Value, statistical	Scan	Partly
F3 Referential breakage	Relational	Join	Yes
F4 Volumetric anomaly	Aggregate, statistical	Scan	Partly
F5 Temporal skew	Temporal	Scan	Yes
F6 Default substitution	Value, custom	Scan	Yes

Explicit rule indicates whether detection requires a rule to have been written for the specific field or property at issue. F1 needs none, because the contract schema is itself the expectation and is checked against the catalog. F2 and F4 are marked partly: a declared rule detects them precisely, while statistical baselines drawn from the metric repository can flag them without one.

Two properties of the mapping deserve comment. First, F1 is the only class detectable with no rule of its own and no scan, because the declared schema is itself the expectation and is discharged against catalog metadata. This is why the planner schedules structural rules first: they are free, and a structural failure makes every subsequent pass pointless.

Second, F2 and F6 are the classes most dependent on declaration. A semantic change to a field nobody constrained, or a sentinel value that falls inside a declared valid range, will not be caught by any declared rule. This is a limit of the declarative approach rather than of any particular implementation, and it is the reason the metric repository supports undeclared drift detection (Section IV-G) as a complement. Any organization adopting this design should expect declared rules and undeclared drift detection to be jointly necessary.

B. Q2: Cost Behavior of Fusion

Consider a contract containing s structural rules, r record-local rules, g aggregate rules, j relational rules, and t statistical rules. Under naive execution, in which each rule is evaluated as its own Spark action, the number of distributed scans is r plus g plus t, with j additional joins, so cost grows linearly in the number of rules. This is the behavior that makes comprehensive validation unaffordable and creates the incentive to delete rules in order to meet latency targets.

Under the planner of Section IV-E the structural rules cost no scan at all. The record-local and aggregate rules collapse into a single pass, because each becomes one conditional accumulator expression evaluated over a shared scan. Each relational rule contributes a join, and the statistical rules share one pass plus a lookup against the metric repository. The scan count therefore falls to at most one plus j plus one, independent of r and g.

The consequence is that the marginal cost of adding a rule is discontinuous, and the discontinuity is the useful part. Adding a record-local rule to a contract that already performs a scan costs one column expression, which is small. Adding the first relational rule to a contract that has none costs a join, which is not. An engineer who understands this distinction can grow coverage cheaply within a pass and knows when a rule will be expensive. R4 is discharged in the sense that cost is bounded by pass count and made predictable, not in the sense that it is free.

Two caveats follow from the same reasoning. Fusion does not reduce the cost of the underlying scan, so validation can never be cheaper than reading the data once. And a contract dominated by relational rules will not

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

benefit, because joins do not fuse. Contracts should therefore be reviewed for pass structure, not merely for rule count.

C. Q3: Disposition and the Enforcement Surface

Severity is the mechanism that converts measurement into control, and Table III states the mapping precisely. The mapping is not uniform across rule families, and treating it as though it were is a modeling error worth naming. Quarantine presupposes that a rule identifies specific offending records, which row-level predicates do and dataset-level assertions do not. Asking a framework to quarantine the rows responsible for a row count anomaly has no coherent answer, so error severity withholds the partition for such rules instead. Making disposition a per-rule property rather than a framework-wide policy is what allows one framework to serve both analytical workloads, where partial publication with an explicit record of exclusions is preferable to publishing nothing, and reconciliation workloads, where a partial dataset is worse than an absent one.

TABLE III SEVERITY AND ITS EFFECT ON DISPOSITION

Severity	Row-level rules	Dataset-level rules
Informational	Record metric only	Record metric only
Warning	Record metric, notify owner	Record metric, notify owner
Error	Quarantine violating rows, publish the rest	Withhold the partition
Critical	Abort the write, retain prior version	Abort the write, retain prior version

Every severity records a metric regardless of disposition, so the evidence trail is unaffected by the enforcement decision. Quarantine is defined only for row-level rules, which identify specific violating records. A dataset-level rule such as a row count bound or a schema assertion has no violating row to divert, so error severity withholds the whole partition instead.

D. Q4: Organizational Preconditions

The adoption model of Section VI is not universally applicable, and it is more useful to state its preconditions than to assert its generality. Four conditions appear necessary.

Central control of the build. The gate must be enforceable by a party other than the team being gated. An organization in which each team controls its own pipeline end to end cannot enforce P5, and the model degrades to advisory tooling.

Stable dataset ownership. Contracts are placed at team boundaries. Where ownership of a dataset is contested or rotates frequently, contract authorship has no natural home and the federation argument does not apply.

A shared vocabulary worth sharing. The paved road depends on a rule library covering organization wide concepts. An organization whose datasets share few conventions will find that most rules are local, which removes the reuse benefit that motivates R2.

Tolerance for blocked pipelines. Critical severity aborts a write. An organization that cannot accept a delayed dataset under any circumstances will set every rule to warning, and enforcement collapses back into notification.

Where these conditions do not hold, the technical contribution remains usable as a library and the adoption model should be reconsidered rather than forced.

VIII. DESIGN RATIONALE AND ANTICIPATED PITFALLS

D1. The compiler is the highest leverage reviewer. Moving column reference and type errors to compile time (R3) targets the class of defect that is most dangerous in this domain, namely a validation job that runs to completion without evaluating anything. A framework that fails silently removes a control while leaving the belief that the control exists.

D2. Complete diagnostics should change behavior, not just ergonomics. Applicative accumulation is easy to mistake for a convenience. Its intended effect is on the unit of work: presented with a complete violation set, an engineer triages related defects together, whereas one failure at a time invites fixing the reported item and resubmitting.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

D3. Overhead is an adoption variable. If validation is expensive, teams will reduce coverage to meet service level objectives, which is the worst outcome because it degrades protection silently while preserving its appearance. Performance work on the planner is therefore adoption work, and should be funded as such.

D4. Mandate must follow demonstration. Contracts written to satisfy a mandate before value has been demonstrated are predictably thin. The rollout sequence of Section VI-D is a deliberate response to this, and inverting it is the most likely way to obtain compliance without protection.

D5. The shared library may matter more than the algebra. For any individual team, the practical contribution is less the rule algebra than the accumulated body of organizational expectations expressed in it. The algebra is what makes that accumulation possible, which is an argument for composability as an adoption property rather than merely an elegant one.

D6. Detection without disposition is incomplete. A framework that reports violations and leaves the response to the caller invites the caller to log and continue. Making severity a property of the rule, and disposition a property of severity, is what converts measurement into control.

IX. Limitations and What an Evaluation Must Establish

This paper presents a design and an adoption strategy. It does not report measurements, and the claims should be read accordingly. We set out here what an empirical evaluation would need to show, both to make the claims falsifiable and to guide anyone implementing the design.

Detection. The coverage argument of Section VII-A is by construction and establishes only that a rule family can express a check for each failure class. It does not establish that contracts written by practitioners will contain those rules. An evaluation would need a fault injection study over the taxonomy of Section III-A, reporting recall per class and, critically, a false positive rate, since a noisy gate is abandoned regardless of its recall.

Cost. The fusion argument predicts that scan count is independent of the number of record-local rules. This is testable directly: hold data volume fixed, vary rule count within a single pass, and confirm that runtime is close to flat. A study should also report the cost of contracts dominated by relational rules, where the analysis predicts no benefit.

Adoption. The federated model and the rollout sequence are arguments, not findings. An evaluation would need onboarding cost for a team's first dataset, the proportion of rules in force that originate in the shared library, and a defect escape measure tracked over a period long enough to separate the effect from confounding organizational change.

Generality. The design assumes a statically typed language with algebraic data types and a distributed relational engine. Organizations on dynamically typed stacks can adopt the adoption model and the severity semantics, but not the compile-time guarantee that motivates R3, which is a substantial part of the argument.

Measurement validity. Any escape rate measured from incident records depends on defects being noticed and reported, and improved detection may itself change reporting behavior. A study reporting such a reduction should say plainly that it measures escaped and subsequently noticed defects.

X. Conclusion and Future Work

This paper has argued that data quality should be treated as code, and has presented a Scala framework design and an organizational model built on that premise. The technical elements are a closed rule algebra, typed schema contracts under semantic versioning, applicative error accumulation, and a predicate fusion planner. The organizational element is a federated model that distributes contract ownership to the teams holding the domain knowledge while centralizing enforcement in build and runtime gates.

The analysis shows that detection coverage follows from the rule families by construction, that fusion makes validation cost track pass count rather than rule count, and that the adoption model depends on four organizational preconditions that are worth checking before committing to it.

The broader claim is that the difficult part of data quality at organizational scale is not detection. Detection has been well understood for some time. The difficult part is making a validation capability the default path rather than an available one, and that is achieved by lowering its cost below the cost of the alternative and by enforcing it mechanically rather than socially.

Four directions extend this work. The first is the empirical evaluation set out in Section IX, which is the immediate priority. Rule inference from historical metrics would propose candidate contracts for datasets that have none, addressing the declaration dependence that Section VII-A identifies as the principal limit of the approach. Cross dataset lineage aware validation would propagate an upstream violation to the consumers whose outputs it invalidates, rather than treating each dataset independently. Finally, a formal semantics for the algebra

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

would permit the fusion transformation to be proven semantics preserving rather than tested for it, which for a component on which correctness controls depend is a worthwhile guarantee.

REFERENCES

[1] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," J. Manage. Inf. Syst., vol. 12, no. 4, pp. 5-33, 1996.
[2] T. C. Redman, Data Quality: The Field Guide. Boston, MA, USA: Digital Press, 2001.
[3] E. Rahm and H. H. Do, "Data cleaning: Problems and current approaches," IEEE Data Eng. Bull., vol. 23, no. 4, pp. 3-13, 2000.
[4] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, "Methodologies for data quality assessment and improvement," ACM Comput. Surv., vol. 41, no. 3, Art. no. 16, Jul. 2009.
[5] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating large-scale data quality verification," Proc. VLDB Endowment, vol. 11, no. 12, pp. 1781-1794, 2018.
[6] S. Schelter et al., "Unit testing data with Deequ," in Proc. ACM SIGMOD Int. Conf. Manage. Data, Amsterdam, The Netherlands, 2019, pp. 1993-1996.
[7] N. Polyzotis, M. Zinkevich, S. Roy, E. Breck, and S. Whang, "Data validation for machine learning," in Proc. Mach. Learn. Syst. (MLSys), vol. 1, 2019, pp. 334-347.
[8] D. Baylor et al., "TFX: A TensorFlow-based production-scale machine learning platform," in Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Halifax, NS, Canada, 2017, pp. 1387-1395.
[9] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data management challenges in production machine learning," in Proc. ACM SIGMOD Int. Conf. Manage. Data, Chicago, IL, USA, 2017, pp. 1723-1726.
[10] D. Sculley et al., "Hidden technical debt in machine learning systems," in Proc. Adv. Neural Inf. Process. Syst. (NIPS), Montreal, QC, Canada, 2015, pp. 2503-2511.
[11] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: A rubric for ML production readiness and technical debt reduction," in Proc. IEEE Int. Conf. Big Data, Boston, MA, USA, 2017, pp. 1123-1132.
[12] T. Rukat, D. Lange, S. Schelter, and F. Biessmann, "Towards automated data quality management for machine learning," in Proc. Workshop MLOps Syst., 3rd Conf. Mach. Learn. Syst. (MLSys), 2020, pp. 1-3.
[13] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang, "Data cleaning: Overview and emerging challenges," in Proc. ACM SIGMOD Int. Conf. Manage. Data, San Francisco, CA, USA, 2016, pp. 2201-2206.
[14] I. F. Ilyas and X. Chu, Data Cleaning. New York, NY, USA: ACM Books, 2019.
[15] M. Odersky and T. Rompf, "Unifying functional and object-oriented programming with Scala," Commun. ACM, vol. 57, no. 4, pp. 76-86, 2014.
[16] M. Odersky, L. Spoon, and B. Venners, Programming in Scala, 3rd ed. Walnut Creek, CA, USA: Artima Press, 2016.
[17] C. McBride and R. Paterson, "Applicative programming with effects," J. Funct. Program., vol. 18, no. 1, pp. 1-13, 2008.
[18] P. Wadler, "Monads for functional programming," in Advanced Functional Programming, Lecture Notes in Computer Science, vol. 925. Berlin, Germany: Springer, 1995, pp. 24-52.
[19] Typelevel, "Cats: A library of abstractions for functional programming in Scala," version 2.1.1, 2020. [Online]. Available: https://typelevel.org/cats
[20] M. Zaharia et al., "Apache Spark: A unified engine for big data processing," Commun. ACM, vol. 59, no. 11, pp. 56-65, 2016.
[21] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Proc. 9th USENIX Symp. Networked Syst. Design Implement. (NSDI), San Jose, CA, USA, 2012, pp. 15-28.
[22] M. Armbrust et al., "Spark SQL: Relational data processing in Spark," in Proc. ACM SIGMOD Int. Conf. Manage. Data, Melbourne, VIC, Australia, 2015, pp. 1383-1394.
[23] M. Armbrust et al., "Delta Lake: High-performance ACID table storage over cloud object stores," Proc. VLDB Endowment, vol. 13, no. 12, pp. 3411-3424, 2020.
[24] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Commun. ACM, vol. 51, no. 1, pp. 107-113, 2008.
[25] M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA, USA: O'Reilly Media, 2017.
[26] H. Wickham, "Tidy data," J. Stat. Softw., vol. 59, no. 10, pp. 1-23, 2014.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

[27] J. M. Hellerstein et al., "Ground: A data context service," in Proc. 8th Biennial Conf. Innovative Data Syst. Res. (CIDR), Chaminade, CA, USA, 2017.
[28] T. Preston-Werner, "Semantic versioning 2.0.0," 2013. [Online]. Available: https://semver.org
[29] P. Bourque and R. E. Fairley, Eds., SWEBOK: Guide to the Software Engineering Body of Knowledge, Version 3.0. Piscataway, NJ, USA: IEEE Computer Society, 2014.
[30] M. E. Conway, "How do committees invent?," Datamation, vol. 14, pp. 28-31, Apr. 1968.
[31] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston, MA, USA: Addison-Wesley, 2010.

