**Key principle**

**Rules must be implemented once and reused everywhere** (or enforced centrally at the canonical sink). Aim for a single source of truth for validation and rejection logic, with defensive DB-side enforcement as the final guardrail.

**Patterns you can use (ordered by recommended practice)**

1. **Shared validation library (recommended)**
   * Implement rules as a language-agnostic specification (JSON/YAML) or as code in a shared library/package that streaming and batch jobs import.
   * Example: trade\_rules.validate(trade) library published to your artifact repo (PyPI/NPM/Maven). Both Flink, Spark, and batch workers use it.
   * Pros: Single implementation, unit-testable, fast. Cons: You must update & redeploy processors when rules change.
2. **Central validation microservice / API**
   * Expose validation as a stateless service (POST /validate) that returns OK or REJECT + reasons. Streaming and batch processors call it.
   * Pros: One deploy point, no redeploy of processors when rules change. Cons: Introduces network latency and a runtime dependency; needs scaling and high availability.
3. **Policy-as-data + rule engine**
   * Store rules in a policy engine (Open Policy Agent, Drools) or policy store (feature flags for rules). Processors load rules at startup or fetch them periodically.
   * Pros: Human-readable rules, hot reload in some engines, consistent enforcement. Cons: learning curve, operational overhead.
4. **DB-side enforcement (final guardrail)**
   * Enforce invariants with conditional MERGE/UPDATE WHERE clauses, CHECK constraints, or stored procedures.
   * Example: prevent updates when incoming maturity\_date < stored maturity\_date in the MERGE statement.
   * Pros: Ensures correctness even if upstream processors misbehave. Cons: DB may reject many records; careful error handling needed.
5. **Outbox + CDC with rules in DB (DB-as-truth)**
   * Persist incoming canonical event to a staging table, run DB-side validation (via stored proc or trigger) which moves accepted rows to trade\_store and rejected rows to rejections. Then CDC publishes accepted events.
   * Pros: Atomic, auditable, single place for acceptance/rejection. Cons: More DB work and potentially more load.

**Practical hybrid architecture (best of both worlds)**

* **Primary enforcement**: Shared validation library used by both streaming jobs (Flink/Kafka consumers) and batch jobs (Spark/SparkSQL). This keeps low-latency processing fast and consistent.
* **Secondary enforcement (DB guardrail)**: All processors perform idempotent MERGE into the trade\_store that includes conditional clauses to reject/ignore invalid updates (e.g., maturity-date rule). This prevents any misbehaving client from corrupting the canonical store.
* **Observability**: Every rejection must be logged with reason, source, and full payload to a rejection topic or S3 for reconciliation and manual review.

**Handling rejections: operational patterns**

* **Reject and notify**: write rejected record to a rejections topic (Kafka) or S3 bucket with metadata and reason; alert middle office if high severity.
* **Dead-letter queue (DLQ)**: For streaming, push to DLQ topic with reasons. For batch, produce a rejection file (parquet/csv) and alert.
* **Compensation**: Optionally include automatic remediation actions or manual workflows for fixes (human-in-the-loop).