let’s go deeper into **Models vs Contracts**, since that’s where most orgs get tangled.

**🔹 1. Models**

Think of **models** as **internal intent + design metadata**.  
They describe what *you* (the producer/platform team) believe about your data.

**Purpose:**

* Define structure, quality assumptions, and operational SLAs **before any sharing**.
* Allow early testing/validation even if no consumer exists yet.

**What they contain:**

* **Schema shape**: field names, types, nullability, keys.
* **Volume expectations**: avg, stdev, tolerances (5M rows/day ±10%).
* **Timeliness**: expected arrival times, refresh frequencies.
* **Lineage assumptions**: “Dataset B depends on Dataset A daily.”
* **Data quality rules**: “amount > 0 for all rows.”

**Characteristics:**

* Stored in **catalogs, metadata repos, or modeling tools**.
* Owned by **producer/platform teams**.
* May never leave your boundary — consumers might never see your internal design notes.

**🔹 2. Contracts**

Contracts are **shared, enforceable promises**.  
They’re what the outside world (consumers, governance, downstream platforms) can **rely on**.

**Purpose:**

* Expose expectations that matter at the **exchange boundary**.
* Allow consumers to build integrations, checks, and trust.

**What they contain:**

* Subset of **model expectations**, but only those relevant across boundaries.
* **Schema guarantees**: names, types, versioning rules (backward/forward compatible).
* **Operational SLAs**: delivery by 9am UTC, completeness ≥ 99%.
* **Contractual obligations**: “Breaking schema without notice = violation.”

**Characteristics:**

* Stored in **contract registries**, not just catalogs.
* Owned **jointly** (producer = responsible, consumer = dependent).
* Enforced at runtime by the **execution engine**.

**🔹 3. Models → Contracts → Execution**

Here’s how they link:

| **Layer** | **Who Authors It** | **Who Consumes It** | **Example** |
| --- | --- | --- | --- |
| **Model** | Producer team | Internal platform, QA | “Dataset X should have ~5M rows/day.” |
| **Contract** | Producer + Consumer | Both sides | “Dataset X will publish ≥ 4.5M rows/day by 9am UTC.” |
| **Execution** | Platform/Engine | Ops + Governance | Run checks → PASS/FAIL (volume check triggered alert today). |

**🔹 4. Why This Matters**

* If you only have **models**:  
  → Great for design-time, but consumers can’t rely on them.
* If you only have **contracts**:  
  → Consumers get promises, but producers may not have internal metadata discipline to keep them accurate.
* If you unify:  
  → **Author once in the model → publish subset as a contract → enforce via execution.**

**🔹 5. Analogy**

* **Model = Blueprint** (your house plan: number of rooms, plumbing layout).
* **Contract = Lease agreement** (what you promise the tenant: 2 bedrooms, working plumbing, fixed rent).
* **Execution = Inspections** (is the plumbing working today? did both bedrooms get delivered?).

✅ So in your situation:

* **Persist expectations in models first** → they’re the “single source of truth.”
* **Generate contracts from models** → exposing only what’s relevant at boundaries.
* **Enforce via execution layer** → checks run automatically, results scored & surfaced.

Deep dive and put all the things together, no need to generate diagrams

Great — below is a **single, comprehensive deep-dive blueprint** that pulls *models, contracts, execution, telemetry, UI, governance, APIs, rules, and migration* together into a practical implementation plan. I left out diagrams per your request and focused on clear, actionable detail you can hand to platform, product, and governance teams.

**1) Core Concepts & Definitions (short)**

* **Model** — internal design metadata: full schema, baselines, tolerances, lineage, rule library. Owned by producer/platform team.
* **Contract** — shareable, enforceable subset of model exposed at an exchange boundary (dataset, distribution, subscription). Jointly owned (producer + consumers). Versioned (SemVer) and lifecycle-managed.
* **Execution Engine** — runs checks (policy-as-code), emits pass/fail/omit events, writes detailed run logs for audit, and creates aggregated rollups for UI/search.
* **SDK / Agent** — policy runner used by producers/consumers to evaluate checks close to the source (push) or during ingestion (pull).
* **Scorecard** — persona-specific rollup of checks into a hygiene score (weights configurable by product/role).
* **Exposure boundary** — any point data leaves an application boundary (push/pull, share, grant access, API, dataset publish) → that triggers contract obligations.

**2) Authoring: What lives in the Model vs Contract**

**Model (full fidelity)**

* Schema: fields, types, keys, nullability
* Business rules: e.g., amount > 0
* Baselines: rolling 28d avg, sd, percentiles
* Operational expectations: cadence, refresh frequency, readiness times
* Tolerances: ±% bands, absolute floors
* Lineage & dependencies
* Anomaly metadata: holiday calendars, deploy windows
* Example storage: metadata catalog (YAML/JSON), with linkage to repo/PR history.

**Contract (subset, enforceable)**

* Dataset ID, owner, producers, consumers
* Contract version (SemVer) and compatibility policy
* Schema guarantees (required public fields, compatibility rules)
* SLAs: ready\_by, completeness\_min, volume\_floor, deviation\_pct
* Obligations: break policy, incident notify targets, retention
* Lifecycle state: Draft / Proposed / Approved / Active / Deprecated / Retired

**Principle:** Contract fields must be well-scoped — publish only what the consumer needs to rely on.

**3) Contract Lifecycle & Versioning (practical rules)**

* **Draft → Proposed** (producer authors; automated tests against model)
* **Proposed → Approved** (governance + consumer sign-off)
* **Approved → Active** (publish and bind; platform enforces)
* **Active → Deprecated** (announce breaking change; begin sunset)
* **Deprecated → Retired** (sunset elapsed; contract removed)

**Versioning**

* PATCH (x.y.z): docs, metadata-only changes — no consumer action
* MINOR (x.Y.z): additive, backward-compatible additions (optional fields)
* MAJOR (X.y.z): breaking changes — must deprecate old major, announce 30–90d, migration plan required

**4) Execution Semantics — Checks & Categories**

**Categories of checks**

* **Schema validation** (type, nullability, required fields)
* **Completeness** (header vs payload count, PK presence)
* **Volume** (today vs baseline ± tolerance; absolute floors)
* **Timeliness** (arrival vs ready\_by SLA; lag p50/p95)
* **Business rules / Domain checks** (predicates like amount > 0)
* **Anomaly assist** (seasonality, holidays, deploy cooloff)
* **Contract conformance** (contract version used, obligations honored)

**Execution outputs**

A run emits a structured results object with:

* execution\_id, dataset, contract\_version, producer, consumer
* checks array {name, status, observed, expected, details}
* score object {overall, components, weights}
* timestamps {started, completed}
* evidence\_links (contract snapshot id, run logs, failed examples)

Statuses include: PASS, FAIL, OMIT (rule not applicable), NOT\_RECEIVED.

**7) APIs (recommendations & examples)**

**Contract Registry**

* POST /contracts — create draft
* PUT /contracts/{id} — update draft
  + POST /contracts/{id}/propose
  + POST /contracts/{id}/approve
  + POST /contracts/{id}/activate
  + POST /contracts/{id}/deprecate {sunset\_days}
* GET /contracts/{id} — returns contract & version history

**Subscription / Exposure**

* POST /subscriptions — consumer subscribes to dataset + contract\_version + purpose
* GET /subscriptions?consumer=...

**8) Alerts, SLIs & SLOs**

* **SLIs**: p95 ingestion lag, on-time delivery %, volume-deviation %
* **SLOs**: on-time ≥ 99% monthly; schema violations ≤ 3/day (7d rolling)
* **Alert severities**:
  + P1: SLA miss + volume below floor (auto-pager)
  + P2: Large deviation but arrived on time (email + ticket)
  + P3: Minor schema additions (digest)
* **Routing**: configured per contract (contract.obligations.incident\_notify)
* **Suppressions**: use holiday calendars & deploy cooloff windows to avoid noisy alerts

**10) Subscription & Transformations**

* **Subscription record**: consumer\_id, dataset\_id, contract\_version, purpose, retention, transformation\_flag
* If a consumer materially transforms data into a new dataset (changes semantics), that becomes a **new dataset** with its **own contract** — the consumer becomes producer for that new dataset.
* For streaming transforms, enforce lineage metadata and attach original contract snapshot to derived dataset metadata.

**11) Audit, Evidence & Compliance**

* **Immutable run logs** stored with a run hash and contract snapshot id (which contract version applied).
* **Chain-of-custody**: store producer timestamp, ingestion timestamp, consumer pull timestamp.
* **Contract snapshot**: store exact contract content used during a run to prove which obligations were enforced.
* **Retention**: align with org policy (e.g., 1 year hot, 3–7 years cold).
* **Access control**: RBAC on who can create/approve/deprecate contracts; audit trails on contract changes.

**16) Governance & RACI (who does what)**

* **Produces**: owns model, proposes contracts, ensures producer checks, provides run evidence.
* **Consumes**: subscribes to contract, validates integration, raises concerns on breaking changes.
* **Governance (CDO)**: approves contracts, sets sunset windows, enforces policy.

**17) Quick checklist to get started (actionable)**

1. Choose 3 exemplar datasets (high-impact) as pilots.
2. Create model metadata for each (schema, baseline, SLA).
3. Draft contracts (limited fields) and run them through governance for approval.
4. Stand up ingestion endpoint and basic SDK to emit schema + header checks.
5. Surface runs in a simple Ops UI and create first alert for P1 condition (SLA miss + floor).
6. Iterate contract fields & scoring based on pilot feedback and broaden rollout.

**18) Next deliverables I can produce for you (pick any)**

* Contract YAML template + validation schema (JSON Schema).
* SDK skeleton (Python) with sample rule DSL runner.
* Concrete API OpenAPI spec for Contract Registry & Ingest endpoints.
* Example product scorecard config + UI wireframe text spec.
* Migration plan to move current V1 scoring into versioned scoring (deltas, rescore).

**1. Execution ↔ Contract Link**

Every **execution run** must reference the **exact contract version** it was validated against. That creates a **foreign key–style relationship**:

* **Contract table/registry**
  + contract\_id (e.g., payments.transactions)
  + version (SemVer: 2.1.0)
  + snapshot\_hash (hash of the contract YAML at approval/activation)
  + lifecycle state, metadata, etc.
* **Execution table**
  + execution\_id (UUID/timestamp-based)
  + dataset\_id (should match contract\_id)
  + contract\_version (FK → Contract.version)
  + contract\_snapshot\_hash (FK → Contract.snapshot\_hash, immutable)
  + producer\_id
  + consumer\_id
  + check results, score, timestamps, etc.

So, in relational terms:

ALTER TABLE executions

ADD CONSTRAINT fk\_contract

FOREIGN KEY (dataset\_id, contract\_version, contract\_snapshot\_hash)

REFERENCES contracts (id, version, snapshot\_hash);

This guarantees that when you inspect any past execution, you know *exactly which contract snapshot* governed it — even if the contract was later updated.

**2. Why include snapshot hash?**

* A contract may be modified **within the same version** (e.g., fixing a description).
* By storing the **hash of the actual YAML content**, you guarantee immutability of the evidence.
* This is your **chain of custody**: no “moving target” contracts.

**3. Practical workflow**

* At **execution start**, the SDK/engine queries the contract registry for the **active contract snapshot** for dataset payments.transactions@2.1.0.
* It stamps that snapshot hash into the run metadata.
* When persisting execution results → it writes (contract\_id, version, snapshot\_hash) alongside.
* Later: when viewing results in Ops UI or scorecards, you can fetch the **same snapshot** for audit/compliance.

**4. UI / Audit implications**

* **Ops UI** → “This run failed under contract payments.transactions@2.1.0 (snapshot 9ac7f1…).”
* **Audit evidence** → “Execution 12345 validated against contract YAML hash 9ac7f1… (content attached).”
* **Governance** → If a consumer disputes a failure, you can reproduce the evaluation using the frozen snapshot.

Would you like me to sketch a **minimal relational schema (Postgres style)** for both contracts and executions tables with these FKs?

**You said:**

What is contract registry here, is it a table

Great question 👍 — the **“contract registry”** is the **system of record** for all contracts.

At minimum it **is a set of tables in a database**. Depending on maturity, it could grow into a microservice with APIs on top, but the storage backbone is a relational model.

**1. What the Contract Registry Does**

* Stores all **contracts** and their **snapshots** (versions of the YAML).
* Enforces **lifecycle state transitions** (Draft → Active, etc.).
* Provides **lookup** so producers/consumers/SDKs can fetch the correct contract version.
* Ensures **immutability** — once a snapshot is activated, it’s frozen (only deprecate/retire allowed).
* Acts as the **foreign key target** for **execution runs**.

**4. Contract Registry in Practice**

* At MVP: ✅ It’s **just tables** in a metadata DB.
* With APIs: expose CRUD endpoints (POST /contracts, POST /contracts/{id}/propose, etc.).
* With governance: add workflow + approvals before moving state forward.
* With audit: attach evidence (who approved, when, commit hash of YAML in git).

**1. What’s in the contract itself (conceptually)**

A **contract** describes:

* Who **owns** it (accountable team)
* Who the **producers** are (who can publish under it)
* Who the **consumers** are (who is subscribed / bound to it)

So yes — conceptually, producer & consumer info is **part of the contract definition**.  
But in the registry schema, you usually want to **normalize it**.

**1. Definition of a Contract**

* **Producer Contract** = one per data asset (default).
  + Defines schema, allowed values, SLAs, DQ rules (validations, thresholds).
* **Consumer Sub-Contract (optional)** = only when consumer needs materially different requirements.
  + Think of it as a *derived contract version* referencing the producer contract.

**2. Enforcement of Contracts**

* **Step 1: Write/Publish Contract**  
  Producer publishes contract (API/UI → stored in metamodel tables).
* **Step 2: Enforce Contract**  
  Enforcement engine (TDQ SDK / BTQ SDK) validates **data in motion / data at rest** against the rules from the contract.
* **Step 3: Monitor Compliance**  
  Store pass/fail results (violations).
  + If violation → look up consumer contract thresholds → decide severity.
  + Trigger alerts (email, Jira ticket, etc.) if configured.

**3. Producer vs Consumer Contracts**

* **Producer Contract (preferred)**
  + One source of truth per data asset.
  + Lower maintenance overhead.
* **Consumer-Specific Needs**
  + Instead of full consumer-oriented contracts, define **sub-contracts / overlays**.
  + Example: Producer defines dq\_threshold = 95%.
    - Consumer A → override: ≥ 99%.
    - Consumer B → override: ≥ 90%.
  + Violations = checked **relative to the consumer’s sub-contract**.

**Contract Schema**

**Dataset-Level**

1. **Producer publishes contract** → contracts, contract\_versions, contract\_rules.
2. **Consumer subscribes** → contract\_subscriptions.
   * Optional stricter requirements → subscription\_overrides.
3. **Execution runs** (batch/stream/API call) → executions.
4. **Validation engine**:
   * Load base rules from contract\_rules.
   * Apply overrides from subscription\_overrides.
   * Check data delivery.
5. **Violations logged** → contract\_violations, tied to both execution + subscription.

**Dataset-level contract model** with richer governance info:

* **Publisher (application/service)** = who *produces* data.
* **Consumer (application/service)** = who *uses* data.
* **Access & Use** = how data may be accessed (e.g., API, batch, streaming) and allowed usage (analytics, ML, compliance).
* **Recovery Objectives** = RPO (Recovery Point Objective) and RTO (Recovery Time Objective) for resilience.

Let’s redesign the schema with those included.

**🔑 How This Model Captures Your Needs**

* **Publisher application**
  + In contract\_versions.publishers (list of valid apps).
  + In executions.publisher\_app (actual app that published/validated).
* **Consumer application**
  + In contract\_subscriptions.consumer\_app.
  + Linked to violations via subscription\_id.
* **Access & Use**
  + In contract\_versions.access\_modes (API, Batch, Streaming).
  + In contract\_versions.allowed\_uses (Analytics, ML, Compliance).
* **Recovery Objectives**
  + In contract\_versions.rpo\_minutes and contract\_versions.rto\_minutes.

**Contract: payments.transactions@1.0**

* **Technical Rules**
  + txn\_id must be non-null.
  + event\_time within 5 min of arrival.
* **Business Rules**
  + amount >= 0.
  + currency ∈ {USD, EUR, GBP}.
* **RPO/RTO**
  + RPO = 15 min, RTO = 60 min.

**Execution Result**

* TDQ pass rate = 98%
* BDQ pass rate = 92%
* Violations logged for negative amount values (BDQ).

**Contract Schema**

**🔑 Key Benefits of Separation**

* **TDQ vs BDQ clearly scoped** → no mixing system-level checks with semantic rules.
* **Different override semantics** → TDQ overrides = thresholds, BDQ overrides = business expressions.
* **Violations stored separately** → makes reporting easy: “system quality vs business quality.”
* **Executions** can summarize results from both in one place.

**🔄 Lifecycle Walkthrough**

1. **Contract Version** created (payments.transactions@1.0)
   * RPO = 15 min, RTO = 60 min
   * TDQ rules (null %, timeliness)
   * BDQ rules (amount ≥ 0, currency in {USD, EUR})
2. **Publisher app** (payments-service) publishes data → validated at runtime.
3. **Consumers** (risk-engine, reporting) subscribe → overrides applied.
4. **Execution run** (batch or stream micro-batch) checks data against TDQ & BDQ.
5. **Violations** recorded (per rule, per consumer).
6. **Alerts** triggered:
   * Rule = BDQ “amount ≥ 0” failed, Critical severity.
   * Alert sent via Slack to payments team + PagerDuty for ops.
   * Alert tracked until acknowledged.

**✅ Benefits of This Model**

* **Clear separation** of TDQ vs BDQ.
* **Consumer-specific strictness** via overrides.
* **Full lineage**: Contract → Rule → Execution → Violation → Alert.
* **Ops integration**: Alerts feed into enterprise monitoring (Slack, PagerDuty, SIEM).
* **Recovery objectives (RPO/RTO)** modeled for SLA compliance.

Defines:

* Schema + rules (TDQ & BDQ)
* SLAs (RPO, RTO)
* Access methods (API, Batch, Streaming)
* Allowed usage (Analytics, ML, Compliance, etc.)

**🔹 Meta-Model Summary (Tables / Entities)**

* **producer\_applications**
* **producer\_contracts** (with versions, schema, access, usage, SLA)
* **producer\_tdq\_rules**
* **producer\_bdq\_rules**
* **consumer\_applications**
* **consumer\_contracts** (link to producer, declare access/use)
* **consumer\_tdq\_overrides**
* **consumer\_bdq\_overrides**
* **executions** (actual runs against data)
* **violations** (which rules failed, severity)
* **alerts** (notifications triggered from violations)

**🔹 Example Flow**

1. **Producer** payments-service publishes contract v1.0.
   * Timeliness ≤ 5 min, Null ≤ 5%.
   * Allowed use: {Analytics, ML}.
   * Access: {Batch, Streaming}.
2. **Consumer** risk-engine subscribes.
   * Access = Streaming.
   * Intended use = Compliance.
   * Overrides: timeliness ≤ 1 min, Null ≤ 1%.
3. **Execution** detects timeliness = 3 min.
   * Passes producer SLA (≤5).
   * Fails consumer override (≤1).
4. **Violation recorded**: consumer-specific.
5. **Alert sent**: only to risk-engine team.

✅ This meta-model gives you:

* Governance anchored at producer.
* Flexibility for consumers to declare stricter expectations.
* Clear separation of **technical vs business quality**.
* Traceability from **contract → execution → violation → alert**.

**🏗️ Unified Contract Model**

**🔹 Core Idea**

* **One contracts table**
  + A **producer contract** = a root contract (no parent).
  + A **consumer contract** = a child referencing a producer contract (or another consumer if chaining is allowed).
* Rules (TDQ/BDQ) always attach to a contract.
* Consumers may **override** producer rules by attaching stricter versions to their child contract.
* Access & usage policies are part of the contract definition.

**🏗️ Meta-Model**

**🔹 Contracts (entity identity)**

This is the **logical contract** (producer or consumer).  
Each contract can have multiple versions.

**🔹 Contract Versions**

Each version is immutable and contains the actual definition.  
This is where we store YAML, SLA, RPO/RTO, etc.

Only place where version\_id exists.  
Everything else hangs off this table (directly or indirectly).

**🔹 Data Access & Usage**

Point to the **contract**, not the version.  
When you want access/usage for a specific version → you join through contract\_versions.

**🔹 Runtime: Executions, Violations, Alerts**

Executions reference a **version\_id**, since they must run against an immutable definition.  
Violations and alerts flow from executions (so they don’t need version\_id themselves).

**🔄 How This Works**

* **Producer defines contract** → row in contracts.
* **Publishes version 1.0** → row in contract\_versions.
* **Attaches rules, access, usage** → tied to contract\_id.
* **Consumer contract** → contracts row with parent\_contract\_id=producer\_contract\_id.
* **Consumer overrides** → just new TDQ/BDQ rows tied to consumer’s contract\_id.
* **Execution** → must point to a version\_id.
* **Violation → Alert** → flow through execution to reach the right version context.

**Contract Tables**

**contracts**

Holds **main producer contracts** and optional **consumer override contracts**.

* Consumer override contracts reference the parent producer contract.

This is the **logical contract** (producer or consumer).  
Each contract can have multiple versions.

**🔹 Contract Versions**

Each version is immutable and contains the actual definition.  
This is where we store YAML, SLA, RPO/RTO, etc.

Only place where version\_id exists.  
Everything else hangs off this table (directly or indirectly).

**contract\_versions**

Stores **versions** of contracts. Only this table has the versioning info.

**2️⃣ Subscription Table**

Each consumer subscribes to a contract version. Optional consumer contract is just a separate contract referenced here.

**3️⃣ Technical & Business Data Quality Tables**

**technical\_data\_quality\_rules**

TDQ rules associated with a contract version.

**4️⃣ Alerts Table**

TDQ/BDQ violations can trigger alerts.

**5️⃣ Data Access & Usage Tables**

Tracks **who accessed the data**, purpose, and retention requirements.

**6️⃣ Applications Table**

Simple table to store publisher and consumer applications.

**🔹 Key Notes**

1. **Versioning:** Only contract\_versions has version info. All references from subscriptions, TDQ, BDQ, alerts, and access/usage are to a specific version.
2. **Consumer overrides:** Optional — they just create a new contract with parent\_contract\_id pointing to the producer contract.
3. **Subscriptions:** Only link **consumer + contract version**. No need to store producer info separately here.
4. **Alerts & violations:** Linked to contract\_versions to maintain historical context.
5. **Data access/usage:** Flexible tables allow tracking both technical and business usage.

**1. Contracts**

**-- =========================================**

**-- 1. Applications Table**

**-- =========================================**

**CREATE TABLE applications (**

**application\_id UUID PRIMARY KEY,**

**application\_name TEXT NOT NULL,**

**owner TEXT,**

**created\_at TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 2. Contracts Table**

**-- =========================================**

**CREATE TABLE contracts (**

**contract\_id UUID PRIMARY KEY,**

**parent\_contract\_id UUID REFERENCES contracts(contract\_id), -- NULL for producer contract**

**contract\_name TEXT NOT NULL,**

**description TEXT,**

**publisher\_id UUID NOT NULL REFERENCES applications(application\_id), -- producer application**

**created\_at TIMESTAMP DEFAULT now(),**

**updated\_at TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 3. Contract Versions Table**

**-- =========================================**

**CREATE TABLE contract\_versions (**

**contract\_version\_id UUID PRIMARY KEY,**

**contract\_id UUID NOT NULL REFERENCES contracts(contract\_id),**

**version\_number INT NOT NULL,**

**effective\_date TIMESTAMP NOT NULL DEFAULT now(),**

**expiry\_date TIMESTAMP,**

**sla\_details JSONB,**

**recovery\_objectives JSONB,**

**created\_at TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 4. Subscriptions Table**

**-- =========================================**

**CREATE TABLE subscriptions (**

**subscription\_id UUID PRIMARY KEY,**

**contract\_version\_id UUID NOT NULL REFERENCES contract\_versions(contract\_version\_id),**

**consumer\_id UUID NOT NULL REFERENCES applications(application\_id),**

**status TEXT NOT NULL CHECK (status IN ('Active','Suspended','Revoked')),**

**start\_date TIMESTAMP NOT NULL DEFAULT now(),**

**end\_date TIMESTAMP,**

**created\_at TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 5. Technical Data Quality Rules Table**

**-- =========================================**

**CREATE TABLE technical\_data\_quality\_rules (**

**tdq\_rule\_id UUID PRIMARY KEY,**

**contract\_version\_id UUID NOT NULL REFERENCES contract\_versions(contract\_version\_id),**

**data\_element TEXT NOT NULL,**

**rule\_expression TEXT NOT NULL,**

**threshold NUMERIC,**

**created\_at TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 6. Business Data Quality Rules Table**

**-- =========================================**

**CREATE TABLE business\_data\_quality\_rules (**

**bdq\_rule\_id UUID PRIMARY KEY,**

**contract\_version\_id UUID NOT NULL REFERENCES contract\_versions(contract\_version\_id),**

**data\_element TEXT NOT NULL,**

**rule\_description TEXT NOT NULL,**

**acceptable\_values JSONB,**

**created\_at TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 7. Contract Alerts Table**

**-- =========================================**

**CREATE TABLE contract\_alerts (**

**alert\_id UUID PRIMARY KEY,**

**contract\_version\_id UUID NOT NULL REFERENCES contract\_versions(contract\_version\_id),**

**alert\_type TEXT NOT NULL CHECK (alert\_type IN ('Email','Ticket','Webhook')),**

**severity TEXT NOT NULL CHECK (severity IN ('Low','Medium','High','Critical')),**

**recipients JSONB,**

**created\_at TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 8. Data Access Table**

**-- =========================================**

**CREATE TABLE data\_access (**

**access\_id UUID PRIMARY KEY,**

**contract\_version\_id UUID NOT NULL REFERENCES contract\_versions(contract\_version\_id),**

**accessor\_id UUID NOT NULL REFERENCES applications(application\_id),**

**access\_type TEXT NOT NULL CHECK (access\_type IN ('Read','Write','Export','ETL')),**

**purpose TEXT,**

**timestamp TIMESTAMP DEFAULT now()**

**);**

**-- =========================================**

**-- 9. Data Usage Table**

**-- =========================================**

**CREATE TABLE data\_usage (**

**usage\_id UUID PRIMARY KEY,**

**contract\_version\_id UUID NOT NULL REFERENCES contract\_versions(contract\_version\_id),**

**accessor\_id UUID NOT NULL REFERENCES applications(application\_id),**

**usage\_description TEXT,**

**usage\_start TIMESTAMP,**

**usage\_end TIMESTAMP**

**);**

**-- =========================================**

**-- Indexes for Performance**

**-- =========================================**

**CREATE INDEX idx\_contract\_versions\_contract ON contract\_versions(contract\_id);**

**CREATE INDEX idx\_subscriptions\_contract\_version ON subscriptions(contract\_version\_id);**

**CREATE INDEX idx\_tdq\_contract\_version ON technical\_data\_quality\_rules(contract\_version\_id);**

**CREATE INDEX idx\_bdq\_contract\_version ON business\_data\_quality\_rules(contract\_version\_id);**

**CREATE INDEX idx\_alerts\_contract\_version ON contract\_alerts(contract\_version\_id);**

**CREATE INDEX idx\_data\_access\_contract\_version ON data\_access(contract\_version\_id);**

**CREATE INDEX idx\_data\_usage\_contract\_version ON data\_usage(contract\_version\_id);**

**🔄 Flow Recap**

1. **Producer defines contract** → contracts row (role='Producer').
2. **Publishes version** → contract\_versions.
3. **Consumers subscribe** → subscriptions link producer ↔ consumer.
   * Can pin to version or always follow latest.
4. **Rules, access, usage** → tied to contract\_id.
5. **Execution runs** → logs into executions with version\_id.
6. **Violations + alerts** → flow from executions to downstream notification.

-- =========================================

-- 1. Applications (Producer and Consumers)

-- =========================================

INSERT INTO applications (application\_id, application\_name, owner) VALUES

('11111111-1111-1111-1111-111111111111', 'Sales\_Data\_Producer', 'Alice'),

('22222222-2222-2222-2222-222222222222', 'Marketing\_Consumer', 'Bob'),

('33333333-3333-3333-3333-333333333333', 'Finance\_Consumer', 'Charlie');

-- =========================================

-- 2. Producer Contract

-- =========================================

INSERT INTO contracts (contract\_id, parent\_contract\_id, contract\_name, description, publisher\_id)

VALUES

('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa', NULL, 'Sales Data Contract', 'Contract for Sales Data', '11111111-1111-1111-1111-111111111111');

-- =========================================

-- 3. Contract Versions

-- =========================================

INSERT INTO contract\_versions (contract\_version\_id, contract\_id, version\_number, effective\_date, sla\_details, recovery\_objectives)

VALUES

('bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa', 1, NOW(),

'{"availability":"99.9%","latency":"5s"}', '{"RTO":"2h","RPO":"15m"}'),

('cccccccc-cccc-cccc-cccc-cccccccccccc', 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa', 2, NOW(),

'{"availability":"99.95%","latency":"2s"}', '{"RTO":"1h","RPO":"5m"}');

-- =========================================

-- 4. Consumer Override Contract (Optional)

-- =========================================

INSERT INTO contracts (contract\_id, parent\_contract\_id, contract\_name, description, publisher\_id)

VALUES

('dddddddd-dddd-dddd-dddd-dddddddddddd', 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa', 'Sales Data Consumer Override', 'Override for Finance Consumer', '11111111-1111-1111-1111-111111111111');

-- =========================================

-- 5. Consumer Subscription

-- =========================================

INSERT INTO subscriptions (subscription\_id, contract\_version\_id, consumer\_id, status)

VALUES

('eeeeeeee-eeee-eeee-eeee-eeeeeeeeeeee', 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', '22222222-2222-2222-2222-222222222222', 'Active'),

('ffffffff-ffff-ffff-ffff-ffffffffffff', 'cccccccc-cccc-cccc-cccc-cccccccccccc', '33333333-3333-3333-3333-333333333333', 'Active');

-- =========================================

-- 6. Technical Data Quality Rules

-- =========================================

INSERT INTO technical\_data\_quality\_rules (tdq\_rule\_id, contract\_version\_id, data\_element, rule\_expression, threshold)

VALUES

('1111aaaa-1111-aaaa-1111-aaaaaaaa1111', 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', 'sales\_amount', 'sales\_amount >= 0', 0),

('2222bbbb-2222-bbbb-2222-bbbbbbbb2222', 'cccccccc-cccc-cccc-cccc-cccccccccccc', 'sales\_date', 'sales\_date IS NOT NULL', NULL);

-- =========================================

-- 7. Business Data Quality Rules

-- =========================================

INSERT INTO business\_data\_quality\_rules (bdq\_rule\_id, contract\_version\_id, data\_element, rule\_description, acceptable\_values)

VALUES

('aaaa1111-aaaa-1111-aaaa-111111aaaa11', 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', 'region', 'Region must be in allowed set', '["US","EU","APAC"]'),

('bbbb2222-bbbb-2222-bbbb-222222bbbb22', 'cccccccc-cccc-cccc-cccc-cccccccccccc', 'product\_category', 'Must be valid category', '["Electronics","Furniture","Clothing"]');

-- =========================================

-- 8. Contract Alerts

-- =========================================

INSERT INTO contract\_alerts (alert\_id, contract\_version\_id, alert\_type, severity, recipients)

VALUES

('aaaabbbb-aaaa-bbbb-aaaa-aaaabbbb1111', 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', 'Email', 'High', '["admin@sales.com"]'),

('ccccdddd-cccc-dddd-cccc-ccccdddd2222', 'cccccccc-cccc-cccc-cccc-cccccccccccc', 'Ticket', 'Medium', '["ops@sales.com"]');

-- =========================================

-- 9. Data Access

-- =========================================

INSERT INTO data\_access (access\_id, contract\_version\_id, accessor\_id, access\_type, purpose)

VALUES

('1111cccc-1111-cccc-1111-cccc1111cccc', 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', '22222222-2222-2222-2222-222222222222', 'Read', 'Marketing analysis'),

('2222dddd-2222-dddd-2222-dddd2222dddd', 'cccccccc-cccc-cccc-cccc-cccccccccccc', '33333333-3333-3333-3333-333333333333', 'Export', 'Finance reporting');

-- =========================================

-- 10. Data Usage

-- =========================================

INSERT INTO data\_usage (usage\_id, contract\_version\_id, accessor\_id, usage\_description, usage\_start, usage\_end)

VALUES

('aaaacccc-aaaa-cccc-aaaa-aaaacccc1111', 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', '22222222-2222-2222-2222-222222222222', 'Monthly marketing campaign', NOW() - INTERVAL '1 month', NOW()),

('bbbbdddd-bbbb-dddd-bbbb-bbbbdddd2222', 'cccccccc-cccc-cccc-cccc-cccccccccccc', '33333333-3333-3333-3333-333333333333', 'Quarterly financial report', NOW() - INTERVAL '3 months', NOW());

**1️⃣ Contract & Version Management**

**1.1 Get the latest version of all contracts**

SELECT c.contract\_id, c.contract\_name, cv.contract\_version\_id, cv.version\_number, cv.effective\_date

FROM contracts c

JOIN contract\_versions cv

ON c.contract\_id = cv.contract\_id

WHERE cv.version\_number = (

SELECT MAX(version\_number)

FROM contract\_versions

WHERE contract\_id = c.contract\_id

);

**1.2 Get all versions of a contract**

SELECT cv.contract\_version\_id, cv.version\_number, cv.effective\_date, cv.status

FROM contract\_versions cv

WHERE cv.contract\_id = 'CONTRACT\_ID\_HERE'

ORDER BY cv.version\_number DESC;

**1.3 Get parent contract info (for consumer-specific overrides)**

SELECT c.contract\_id, c.contract\_name, c.parent\_contract\_id

FROM contracts c

WHERE c.parent\_contract\_id IS NOT NULL;

**2️⃣ Subscriptions**

**2.1 List all consumers subscribed to a contract**

SELECT s.subscription\_id, a.application\_name AS consumer\_name, cv.version\_number

FROM subscriptions s

JOIN applications a ON s.consumer\_id = a.application\_id

JOIN contract\_versions cv ON s.contract\_version\_id = cv.contract\_version\_id

WHERE cv.contract\_id = 'CONTRACT\_ID\_HERE';

**2.2 List all subscriptions with contract details**

SELECT s.subscription\_id, c.contract\_name, cv.version\_number, a.application\_name AS consumer\_name, s.status

FROM subscriptions s

JOIN contract\_versions cv ON s.contract\_version\_id = cv.contract\_version\_id

JOIN contracts c ON cv.contract\_id = c.contract\_id

JOIN applications a ON s.consumer\_id = a.application\_id;

**2.3 Find contracts with no active subscriptions**

SELECT c.contract\_id, c.contract\_name

FROM contracts c

LEFT JOIN contract\_versions cv ON c.contract\_id = cv.contract\_id

LEFT JOIN subscriptions s ON cv.contract\_version\_id = s.contract\_version\_id

WHERE s.subscription\_id IS NULL;

**3️⃣ Technical Data Quality (TDQ)**

**3.1 List TDQ rules for a contract version**

SELECT tdq.data\_element, tdq.rule\_expression, tdq.threshold

FROM technical\_data\_quality\_rules tdq

WHERE tdq.contract\_version\_id = 'CONTRACT\_VERSION\_ID\_HERE';

**3.2 Find contracts missing TDQ rules**

SELECT c.contract\_id, c.contract\_name

FROM contracts c

JOIN contract\_versions cv ON c.contract\_id = cv.contract\_id

LEFT JOIN technical\_data\_quality\_rules tdq ON cv.contract\_version\_id = tdq.contract\_version\_id

WHERE tdq.rule\_id IS NULL;

**4️⃣ Business Data Quality (BDQ)**

**4.1 List BDQ rules for a contract version**

SELECT bdq.data\_element, bdq.rule\_description, bdq.acceptable\_values

FROM business\_data\_quality\_rules bdq

WHERE bdq.contract\_version\_id = 'CONTRACT\_VERSION\_ID\_HERE';

**4.2 Find contracts missing BDQ rules**

SELECT c.contract\_id, c.contract\_name

FROM contracts c

JOIN contract\_versions cv ON c.contract\_id = cv.contract\_id

LEFT JOIN business\_data\_quality\_rules bdq ON cv.contract\_version\_id = bdq.contract\_version\_id

WHERE bdq.rule\_id IS NULL;

**5️⃣ Alerts**

**5.1 Get alerts for a contract version**

SELECT alert\_type, severity, recipients

FROM contract\_alerts

WHERE contract\_version\_id = 'CONTRACT\_VERSION\_ID\_HERE';

**5.2 List contracts with high severity alerts**

SELECT c.contract\_name, cv.version\_number, ca.alert\_type, ca.severity

FROM contracts c

JOIN contract\_versions cv ON c.contract\_id = cv.contract\_id

JOIN contract\_alerts ca ON cv.contract\_version\_id = ca.contract\_version\_id

WHERE ca.severity = 'HIGH';

**6️⃣ Data Access & Usage**

**6.1 List all data access for a contract version**

SELECT da.access\_type, da.purpose, a.application\_name AS accessor\_name

FROM data\_access da

JOIN applications a ON da.accessor\_id = a.application\_id

WHERE da.contract\_version\_id = 'CONTRACT\_VERSION\_ID\_HERE';

**6.2 List all data usage for a contract version**

SELECT du.usage\_description, a.application\_name AS accessor\_name, du.usage\_start, du.usage\_end

FROM data\_usage du

JOIN applications a ON du.accessor\_id = a.application\_id

WHERE du.contract\_version\_id = 'CONTRACT\_VERSION\_ID\_HERE';

**6.3 Find usage by a specific consumer**

SELECT du.usage\_description, c.contract\_name, cv.version\_number, du.usage\_start, du.usage\_end

FROM data\_usage du

JOIN contract\_versions cv ON du.contract\_version\_id = cv.contract\_version\_id

JOIN contracts c ON cv.contract\_id = c.contract\_id

WHERE du.accessor\_id = 'CONSUMER\_APPLICATION\_ID\_HERE';

**7️⃣ Analytics / Summary Queries**

**7.1 Count of consumers per contract version**

SELECT cv.contract\_id, cv.contract\_version\_id, COUNT(s.subscription\_id) AS consumer\_count

FROM contract\_versions cv

LEFT JOIN subscriptions s ON cv.contract\_version\_id = s.contract\_version\_id

GROUP BY cv.contract\_id, cv.contract\_version\_id;

**7.2 Contracts with no TDQ or BDQ rules**

SELECT c.contract\_id, c.contract\_name

FROM contracts c

JOIN contract\_versions cv ON c.contract\_id = cv.contract\_id

LEFT JOIN technical\_data\_quality\_rules tdq ON cv.contract\_version\_id = tdq.contract\_version\_id

LEFT JOIN business\_data\_quality\_rules bdq ON cv.contract\_version\_id = bdq.contract\_version\_id

WHERE tdq.rule\_id IS NULL OR bdq.rule\_id IS NULL;

**7.3 Subscriptions per consumer**

SELECT a.application\_name AS consumer\_name, COUNT(s.subscription\_id) AS active\_subscriptions

FROM subscriptions s

JOIN applications a ON s.consumer\_id = a.application\_id

GROUP BY a.application\_name;

✅ These SQL statements cover:

* **Contract management** (versions, parent/child relationships)
* **Subscriptions** (consumer lists, missing subscriptions)
* **Technical & Business Data Quality rules**
* **Alerts**
* **Data Access & Usage**
* **Analytics / summary metrics**

Contracts:

type: object

properties:

contract\_id:

type: string

description: Unique identifier for the contract

contract\_name:

type: string

description: Name of the contract

parent\_contract\_id:

type: string

nullable: true

description: Optional reference to a parent contract (for consumer overrides)

required:

- contract\_id

- contract\_name

Contract\_Versions:

type: object

properties:

contract\_version\_id:

type: string

description: Unique identifier for this contract version

contract\_id:

type: string

description: Reference to parent contract

version\_number:

type: string

description: Version number (e.g., 1.0, 1.1)

effective\_date:

type: string

format: date

description: Start date for this version

status:

type: string

enum: [DRAFT, ACTIVE, RETIRED]

required:

- contract\_version\_id

- contract\_id

- version\_number

- effective\_date

- status

Subscriptions:

type: object

properties:

subscription\_id:

type: string

description: Unique identifier for subscription

contract\_version\_id:

type: string

description: Reference to contract version

consumer\_id:

type: string

description: Identifier of the consumer application

subscription\_date:

type: string

format: date

description: Date the subscription was created

required:

- subscription\_id

- contract\_version\_id

- consumer\_id

- subscription\_date