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.

**5) Scoring — algorithm & weights**

* Score = weighted average of check outcomes (PASS=1, FAIL=0, OMIT excluded)
* Example weights: schema 30%, volume 40%, timeliness 30% (product-configurable)
* overall\_score = sum(weight\_i \* score\_i) / sum(weights of applicable checks)
* Add state mapping:
  + = 0.95 → Green
  + 0.75–0.95 → Amber
  + < 0.75 → Red
  + NOT\_RECEIVED → distinct state (no score, show as “no data”)
* Support multi-version scoring: store versioned scoring algorithm identifiers and keep historical scores based on algorithm version.

**6) Data Plane & Observability**

* **Raw run logs (Aurora/Postgres):** immutable per-run results, line-level evidence, contract snapshot reference, full check diagnostics.
* **Search & rollups (OpenSearch / Elastic):** aggregated hygiene, trend, domain rollup, time-series for dashboards.
* **Materialized views:** historical latest vs full history (be careful with refresh windows / eventual consistency).
* **Timestamps & freshness:** every UI view must show last-refresh timestamp and contract version applied to each run.

**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

**Execution / Ingest**

* POST /checks/run — submit run results (bulk)
* GET /checks/results?dataset=...&from=...&to=...
* POST /search/reindex — trigger reindex to search (admin)
* POST /score/rescore?v=2 — backfill new scoring algorithm

**Subscription / Exposure**

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

**8) SDK / Policy-as-Code**

* Provide lightweight libraries in Java / Python / Scala that:
  + Evaluate rules (DSL or WASM), emit structured check events
  + Validate header counts, schema checks, business rules
  + Sign results (hash) and push to ingestion API
  + Support offline (batch) mode and streaming mode
  + Allow configuration: contract\_id, contract\_version, baseline window, holidays

**Policy DSL** (example)

rules:

- id: amount\_positive

when: dataset == "payments.transactions"

eval: "amount > 0"

severity: high

- id: daily\_volume

type: aggregate

field: row\_count

method: pct\_dev

baseline\_window: 28

tolerance\_pct: 10

**9) 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.

**12) Implementation Roadmap (pragmatic phases)**

**Phase 0 — Foundations (2–4 weeks)**

* Inventory top N datasets (by consumer count / criticality)
* Define model template and contract schema (YAML)
* Agree lifecycle + governance process and owners
* Build minimal POST /contracts, GET /contracts/{id} APIs
* Start capturing model metadata in catalog

**Phase 1 — Execution MVP (4–8 weeks)**

* Build execution ingestion endpoint (POST /checks/run)
* Provide a simple SDK that emits schema + header checks
* Persist run logs in DB and index basic rollups in search
* Implement basic scorecard (schema+timeliness+volume weights)
* Hook on-call for top 3 datasets to test alerts

**Phase 2 — Contract enforcement & lifecycle (6–10 weeks)**

* Enforce contract binding on ingestion (reject runs that violate hard floors if configured)
* Implement lifecycle endpoints (propose, approve, activate, deprecate)
* Add contract snapshot storage and link to runs
* Add subscription API

**Phase 3 — Maturity (ongoing)**

* Add rule DSL, advanced anomaly detection, holiday calendars
* Improve UI (Ops UI + Persona Scorecards), drilldowns
* Implement versioned scoring and rescore tooling
* Add audit + retention automation

**13) Acceptance Criteria (what “done” looks like)**

* Authoring: model metadata stored in catalog for top 50 datasets
* Contracts: can be created, proposed, approved, and activated; contract snapshots saved with runs
* Execution: SDK runs checks and posts run results that appear in DB + search
* UI: Scorecard shows latest hygiene and Ops UI shows failing runs with timestamps
* Alerts: P1 for SLA miss + floor breach routes to on-call and creates ticket
* Audit: each run has linked contract snapshot and immutable log entry

**14) Common Pitfalls & How to Mitigate**

* **Pitfall:** Too many contract fields → becomes a maze.  
  **Fix:** Publish minimal, high-value SLAs + schema; keep internal model rich.
* **Pitfall:** Skipping governance → many incompatible contracts.  
  **Fix:** Lightweight automated gating (automated tests + governance review).
* **Pitfall:** Score semantics disagree across teams (UI vs product).  
  **Fix:** Versioned scoring config, show underlying checks, and keep weights auditable.
* **Pitfall:** Alert fatigue from naive thresholds.  
  **Fix:** Adaptive tolerances, holiday calendars, deploy cooloff windows, and anomaly assist.
* **Pitfall:** “Consumer-only failures” because producers aren’t instrumented.  
  **Fix:** SDK adoption plan + platform policy for producer instrumentation for critical datasets.

**15) Sample artifacts (concise examples)**

**Model expectation (YAML snippet)**

volumes:

cadence: daily

baseline\_rows: 5000000

method: rolling\_28d

tolerance:

absolute\_pct: 10

min\_rows: 4500000

timeliness:

ready\_by: "09:00Z"

lag\_sli:

p95\_minutes: 60

**Contract (YAML snippet)**

contract:

id: payments.transactions

version: "2.1.0"

owner: seal:payments

producers: [svc:txn-writer]

consumers: [svc:recon]

slas:

ready\_by: "09:00Z"

completeness\_min: 0.99

volume\_floor: 4500000

deviation\_pct: 10

obligations:

incident\_notify: "on\_fail:p1@payments.oncall"

**Execution result (JSON snippet)**

{

"execution\_id": "exec\_2025-09-09T08:40:12Z\_abc123",

"dataset": "payments.transactions",

"contract\_version": "2.1.0",

"checks": [

{"name":"schema\_validation","status":"PASS"},

{"name":"volume\_deviation","status":"FAIL","observed":4300000,"expected":5000000,"tolerance\_pct":10}

],

"score":{"overall":0.88,"weights":{"schema":0.3,"volume":0.4}}

}

**SQL to compute pct deviation (example)**

WITH baseline AS (

SELECT avg(cnt) AS avg\_cnt

FROM daily\_counts

WHERE day >= current\_date - interval '28 days'

),

today AS (

SELECT cnt FROM daily\_counts WHERE day = current\_date

)

SELECT

t.cnt,

b.avg\_cnt,

(t.cnt - b.avg\_cnt) / NULLIF(b.avg\_cnt,0) \* 100 AS pct\_dev

FROM today t CROSS JOIN baseline b;

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

* **Producer team**: owns model, proposes contracts, ensures producer checks, provides run evidence.
* **Consumer team**: subscribes to contract, validates integration, raises concerns on breaking changes.
* **Governance (CDO)**: approves contracts, sets sunset windows, enforces policy.
* **Platform**: provides SDK, execution engine, registry, search/rollups, APIs, and auditing.
* **Product**: defines scorecard personas and weights, defines business SLAs.

**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).

If you want, I’ll **generate one of those deliverables now** (e.g., Contract YAML + JSON Schema or a sample SDK runner in Python). Which would be most useful next?

**You said:**

Contract yaml + json schema

**ChatGPT said:**

Perfect — let’s build both:

1. **Contract YAML Template (with placeholders & annotations)**
2. **JSON Schema** that validates such contracts.

**1) Contract YAML Template**

contract:

id: payments.transactions # unique dataset identifier (namespace + name)

version: "1.0.0" # SemVer; PATCH=metadata, MINOR=additive, MAJOR=breaking

owner: team:payments # accountable team (org/team/service)

producers: # services that produce this dataset

- svc:txn-writer

consumers: # services consuming under this contract

- svc:recon

- svc:ledger

schema:

compatibility: backward # backward | forward | none

fields:

- name: transaction\_id

type: string

required: true

- name: amount

type: decimal(18,2)

required: true

- name: event\_time

type: timestamp

required: true

slas:

ready\_by: "09:00Z" # dataset arrival deadline

completeness\_min: 0.99 # % of expected rows/keys present

volume\_floor: 4500000 # absolute minimum rows

deviation\_pct: 10 # allowed deviation from baseline (%)

obligations:

break\_policy: "announce 30d, deprecate 90d"

incident\_notify: "on\_fail:p1@payments.oncall"

lifecycle:

state: Draft # Draft | Proposed | Approved | Active | Deprecated | Retired

since: 2025-09-09 # ISO8601 date

metadata: # optional freeform annotations

description: "Transactions dataset contract for reconciliation"

tags: ["finance", "daily", "sensitive"]

**2) JSON Schema for Contracts**

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "Data Contract Schema",

"type": "object",

"required": ["contract"],

"properties": {

"contract": {

"type": "object",

"required": ["id", "version", "owner", "schema", "slas", "obligations", "lifecycle"],

"properties": {

"id": { "type": "string", "pattern": "^[a-zA-Z0-9\_.-]+$" },

"version": { "type": "string", "pattern": "^[0-9]+\\.[0-9]+\\.[0-9]+$" },

"owner": { "type": "string" },

"producers": {

"type": "array",

"items": { "type": "string" }

},

"consumers": {

"type": "array",

"items": { "type": "string" }

},

"schema": {

"type": "object",

"required": ["compatibility", "fields"],

"properties": {

"compatibility": { "enum": ["backward", "forward", "none"] },

"fields": {

"type": "array",

"items": {

"type": "object",

"required": ["name", "type"],

"properties": {

"name": { "type": "string" },

"type": { "type": "string" },

"required": { "type": "boolean" }

}

}

}

}

},

"slas": {

"type": "object",

"required": ["ready\_by", "completeness\_min"],

"properties": {

"ready\_by": { "type": "string", "pattern": "^[0-9]{2}:[0-9]{2}Z$" },

"completeness\_min": { "type": "number", "minimum": 0, "maximum": 1 },

"volume\_floor": { "type": "integer", "minimum": 0 },

"deviation\_pct": { "type": "integer", "minimum": 0, "maximum": 100 }

}

},

"obligations": {

"type": "object",

"properties": {

"break\_policy": { "type": "string" },

"incident\_notify": { "type": "string" }

}

},

"lifecycle": {

"type": "object",

"required": ["state", "since"],

"properties": {

"state": {

"enum": ["Draft", "Proposed", "Approved", "Active", "Deprecated", "Retired"]

},

"since": { "type": "string", "format": "date" }

}

},

"metadata": {

"type": "object",

"properties": {

"description": { "type": "string" },

"tags": {

"type": "array",

"items": { "type": "string" }

}

}

}

}

}

}

}

✅ With this setup:

* Teams write contracts in YAML following the template.
* You validate them against the JSON Schema (with ajv, jsonschema in Python, etc.).
* You enforce SemVer, lifecycle state transitions, and SLA/obligation consistency on top of schema validation.

Do you want me to also generate a **ready-to-use Python validator** that loads a YAML contract and validates it against this JSON Schema?

**You said:**

Does the contract have a foreign key reference in executions

**ChatGPT said:**

Yes — and that’s actually **a key design principle** for traceability.  
Let me unpack it cleanly:

**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.

✅ So yes:  
Executions should have a **foreign key reference** to the **contract snapshot** (id + version + hash).  
That’s what makes the whole thing auditable, non-repudiable, and regulator-friendly.

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

**ChatGPT said:**

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**.

**2. Minimal Relational Schema (Postgres style)**

**contracts**

Represents the “logical” dataset contract across versions.

CREATE TABLE contracts (

id TEXT NOT NULL, -- dataset identifier (e.g., payments.transactions)

owner TEXT NOT NULL,

created\_at TIMESTAMP NOT NULL DEFAULT now(),

PRIMARY KEY (id)

);

**contract\_versions**

Each version of a contract (semver + frozen YAML snapshot).

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL, -- e.g. "2.1.0"

snapshot\_hash TEXT NOT NULL, -- SHA256 of YAML content

yaml\_content JSONB NOT NULL, -- full contract YAML frozen here

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

**executions**

Execution runs that reference a contract snapshot.

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

contract\_version TEXT NOT NULL,

contract\_snapshot\_hash TEXT NOT NULL,

producer\_id TEXT,

consumer\_id TEXT,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

results JSONB, -- checks, score, diagnostics

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

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

**3. Why split contracts and contract\_versions?**

* contracts = logical dataset identity (payments.transactions).
* contract\_versions = immutable snapshots tied to semver.
* This makes it easy to query “all versions of this dataset contract” or “latest active version.”

**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).

👉 So yes — the registry **is a table-backed system** (or set of tables) that acts as the **source of truth** for contracts and snapshots.

Would you like me to also show how this registry **integrates with Git** (i.e. contract YAMLs stored in git repo but indexed in DB for lookup)? That’s a common hybrid approach.

**You said:**

Does the contract table need to have producer and consumer info

**ChatGPT said:**

Excellent point. Let’s break it down carefully, because this is where people either **overload the contract table** or keep it clean with **relational separation**.

**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**.

**2. Schema Options**

**Option A — Inline (simple, quick)**

Store producers and consumers as arrays (JSONB/text array) in the contract\_versions table:

ALTER TABLE contract\_versions

ADD COLUMN producers TEXT[] NOT NULL DEFAULT '{}',

ADD COLUMN consumers TEXT[] NOT NULL DEFAULT '{}';

Pros:

* Simple, single-row lookup for a contract.
* Works fine for early MVP.

Cons:

* Harder to track **history of who subscribed when**.
* No room for consumer-specific metadata (e.g., purpose, retention, SLA override).

**Option B — Normalized (scalable, auditable)**

Create separate tables for producers and consumers:

CREATE TABLE contract\_producers (

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

producer\_id TEXT NOT NULL,

PRIMARY KEY (contract\_id, version, snapshot\_hash, producer\_id),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

CREATE TABLE contract\_consumers (

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_id TEXT NOT NULL,

purpose TEXT,

retention\_days INT,

subscribed\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash, consumer\_id),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

Pros:

* Tracks **multiple consumers** per contract cleanly.
* Allows consumer metadata (purpose, retention, notified contacts, etc.).
* Easy to audit subscriptions over time.

Cons:

* More joins.
* Slightly more complex queries.

**3. Recommendation**

* For **MVP**: store producers inline in contract\_versions (array).
* For **consumers**: use a **separate contract\_consumers table** because subscriptions evolve independently of contract lifecycles.

That gives you:

* Producers bound at contract definition time (who can publish).
* Consumers as **registrations/subscriptions** (who depends on it, and under what terms).

**4. Execution linkage**

In executions, you’ll record:

* producer\_id (must match contract\_producers entry)
* consumer\_id (must match contract\_consumers entry)

That way you can prove:  
“This execution was producer svc:txn-writer delivering to consumer svc:ledger under contract payments.transactions@2.1.0.”

✅ So:

* **Producer info** → can live *inside contract\_versions* (they don’t change often).
* **Consumer info** → better as a **separate table** (subscriptions are dynamic and auditable).

Would you like me to draw up a **normalized full registry schema** (contracts + versions + producers + consumers + executions) in SQL so you can see it all stitched together?

**You said:**

Full table definition with option B

**ChatGPT said:**

Perfect 👍 — let’s build the **full normalized schema** with **Option B** (producers inline in contract versions, consumers normalized in their own table).

**📑 Contract Registry & Execution Schema (Option B)**

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

-- 1. Contracts (logical dataset identity)

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

CREATE TABLE contracts (

id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL, -- accountable team

created\_at TIMESTAMP NOT NULL DEFAULT now()

);

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

-- 2. Contract Versions (immutable snapshots)

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

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL, -- SemVer, e.g. "2.1.0"

snapshot\_hash TEXT NOT NULL, -- SHA256 of YAML/JSON snapshot

yaml\_content JSONB NOT NULL, -- full frozen contract definition

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

producers TEXT[] NOT NULL DEFAULT '{}', -- who can publish under this contract

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP NOT NULL DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

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

-- 3. Contract Consumers (registrations / subscriptions)

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

CREATE TABLE contract\_consumers (

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_id TEXT NOT NULL, -- e.g. "svc:ledger"

purpose TEXT, -- why consumer needs data

retention\_days INT, -- optional consumer-specific retention

subscribed\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash, consumer\_id),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 4. Executions (runtime validations / deliveries)

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

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

contract\_version TEXT NOT NULL,

contract\_snapshot\_hash TEXT NOT NULL,

producer\_id TEXT NOT NULL, -- must be in contract\_versions.producers

consumer\_id TEXT, -- optional: direct consumer target

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

results JSONB, -- full check results, metrics, diagnostics

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

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

**🔑 Key Design Notes**

1. **contracts**
   * Represents a logical dataset (stable identifier).
   * Owned by one accountable team.
2. **contract\_versions**
   * Each version = immutable snapshot of the contract YAML/JSON.
   * Holds producer list (TEXT[]).
   * Lifecycle managed here.
3. **contract\_consumers**
   * Normalized table.
   * Tracks who subscribed to which version + snapshot.
   * Can hold consumer metadata (purpose, retention, SLA overrides, etc.).
4. **executions**
   * Every run references a **contract snapshot** via FK.
   * Stores producer\_id, and optionally consumer\_id (for targeted deliveries).
   * results JSONB stores check outcomes.

**✅ Benefits**

* Immutable history → every execution points to **exact contract snapshot**.
* Producers → tightly coupled with contract versions.
* Consumers → loosely coupled & auditable via their own table.
* Flexible enough to support governance workflows (approvals, state transitions).

Would you like me to also add a **query example** (e.g., “show all consumers of payments.transactions@2.1.0 and their last execution status”)?

**You said:**

little bit different. second thing. The definition of contract violation, right? So end end of the day, the first step is to writing a contract or getting into a contract, then enforcing the contract, right? To ensure that whatever contract parameters has been agreed upon is being met on day-to-uty basis while data exchange. Obviously, that will be done by TBQ, right? Or a BTQ SJK both in that case. Two, it will check against the data exchange with what is defined in the contract, right? That's mind, right? When you say consumer contract, we are also talking about the subscription here, right? Sorry, subscription is coming up in the producer-riented contract. In this, what I amm saying, I'm talking about individual contracts for each and every consumer. Mm hmm. Okay. So. So you are saying like there is there is no relation to the producer contact at that point when you talk about the consumer contact is that what we need to think? yes, exactly. So when I'm just I'm just explaining you what this producer and what is consumer contract. End of the thing we are moving towards just the producer contract. That means one contract per data asset, it. So there are 2000, you, data assets defined, 2,000 producer contracts only. Only. Okay. Not more than that. Okay. Yes. It can be more because there are some cases that we can think of. Like as given an example, right? In a consumer. So a 2018 assets that you have. Now, say, for example, every data assets is publishing data to three different consumers in that case. Right? So you end up having 6,000 data contracts for 2,000 data assets, right? if you take a consumer-oriented approach, right? why we wanted that approach earlier was that? Because every consumer data requirement might be unique. or usage might be unique, right? Critibility of data won't be unique. The data called the violations might be unique. So, for example, right? for one consumer, right? If data quality P tolerance level, right, for a particular data element is 10%, right? For someone else, would be 50%. I'm good example, right? for someone would be less than 1%, right? So what I'm trying to say is that they cannot be a single data quality rule applicable for every country.. Right? Because every consumer might be having its own and even the SLA timeiness of freshness. say, for example, a consumer taking data every day or every fres. consumer may be taking one a month. They're putting applications. Once a quarter in that case, right? Think about a wind-based you detection that happens, right? Some of them might be some of you might be taking it on a real-time basis, right? Some might be taking on the end of the day consuming all the events together. Right? So how you define the SLLA being one for all, right? For one application it would be real time. For one, it will be end of the day, right? So in such cases, just having one producer contract might not suffice, right? Because every dataetection type is different, as there is different quality rules are different. may be different, right? Threshold and all are different, right? And sport and the third part is that if it defined one, unique producer contract only, right, how the individual contract violation will be identified. right? As I said earlier, right, for consumer A, right? D quality is 99%. That is an acceptable acceptable range. And if it is more than 99%, it is not. But for consumer it B, data quality parameter within 90% is extual range, right? Beyond 900. So if less than 80% less than 1% of contract violation from consumer A can be also contract violation, not so you will not do contact violation consumer B in that case because for them, special is not met, right? So in this case, will that be termed as a contract violation for both or only one? Right? So, again, I think these things, but that for one key challenge, you're having a consumer oriented unique approach is that the more customizationally bring in, right, the more complex it will become, right, as you will not, that will not aid into adoptions unless the consumers might get on board to the entire concept of data contract, right? Because they'll have to create every data exchange. They are like, I know they are thousands of distributions, right, for you in CCB, right? And if a permission combination of two is 2 is there right, you might end up getting so many hundreds and thousands of f contracts, right? And then maintaining these data contracts and all those things comes along with that, right? So it's rather than that right up. So again, whatever I'm speaking now, right, same points were discussed with and deep analance and deep agreed that yes, consumer looks fine, but then the complexity of that, right? The amount of effort are you also put in there, it' not eight in adoptions, less consumable less people will get on boarded because you'll end up getting every paper of contract, right? So then the agreement, what was agreed in that call was that. We'll take a producer, oriented approach only, produceer data contract, right? However, for unique edge cases of a consumer, which is different, we can create a separate subcontract in that case. We can find out the way how to deal. You know, with that particular feature in that case later on. As of family take the approaching the wherever we see that a unique or according to me, if the demand is materially different from a consumer B, based on a major primary contract in that case, right, then we can get with more version of a contract, that version, and most sub contract. Got it. Makes sense. All right. And that's good. That's good. that's sort of approved will be going to take a oriented approach only. And so all those details of a contract, it will mostly around all these data elements that we have data quality rules, data valations, rules. SLs and all those things that is already there has to be taken up. So And so I want, you know, if you can ping me that, you know, the schema that you' showing me earlier that I'd prepared, right? Because that gives gives me the view about how many high level, what. I'm going to ping you this. There is I'm showing you the tool, right? Also, a request for the tool from the software and like it goes to software, you can set it up and you know, get it on your machine. But for now you need we still need to get the license thing. We'll find it out how we can get the licensing part, right? But for now, just try to get the get the software and then install it in your machine. That's one thing you can try. Yeah, let me ping this to you. I'm just I'm to see. But even Jason Jason definitely can help us, but know. The thing is, like, Jason who? Okay, for example, if you can open this guy, oops, and finger the same thing. Okay, just try to open that link and it will take you to Jason sunfile, right? It' asking me to continue with that. You can do that. That's a login. Would you able to login? No. What? It's the same one second.. What is this happening now?. This is the peach.. don't have or what? I don't I't know what's going on, but what I can do is I can. I just might be the rule, I think so. I' only for the architects, this entire thing. Okay, you need to get it, I guess.. I don't want to as long. No, I want to send you this, Jason file, right? You can get it, not a big deal. Makes sense. or the checks or the schema and all those things, what Ed and all is thinking and whether we need to revisit our contact component specifications that the r should you last week, right? Whether we need to change anything there or not in that. I'm just seeing you this, I downloaded the file for you. Okay. There you. Just download this, you should have it. think should be good. Okay. Um, that is that. So, oral, right, Nitin, I think, you know,. You can take us five story for next print for design and all those things, right? So my point is, I like, you know, we can start doing it. What the information in your brain, right, based on your interactions and all that, if you put it into your document, right? I know, like you're charting and all, but if somebody comes in and like and my air, right, or somebody comes into the picture, you know, instead of explaining the things like kind of you have a conference page, or any document, I remember we sent you sent one document is that document or do you think like and we can put into a different confluence and know capture all these? I will put it there. I mean I will put it there so as of now, I have all those details in I show you you? Yeah, yeah points, what do you all columns and all that. What are the theory we are talking about it in general, with the architects or no PO as a PO, you are talking with the registry and everybody, right? Those bullet points, if you wanted to put it somewhere I mean, like I'm more of a requirement document, if you think about it, right? That's what I'm trying to say. what I'll do next is that. I mean, all those requirements that we were discussing were cause, right? I'll update that all those things in those respective epics and. Even that's also. Yeah. We don't manage clean folder epics for almost everyone, like Nal those features. Firstly the metamal design, the API and catalui. They' creating their contracts So UI. So I'm saying create a contracts to UIR an API, right, then the next question would be, are the detailments that you want, right in that case? So that determents would defined there, then the next question would be like, okay, what will the possible values that can you take for defined for the data element in that gazette, acceptable values. So these information also will get it, but in that case, we'll also have a very closely leaased that the catal team is available right? What are those? Because most of the information is are already defined in the catalog, like the dataacid details and element details, the rule of the dataital elements and data quality rules, the it is not there, we can define those, you know, are theoof I users to define them. Makes sense. M sense. Yeah. Start putting there like, you know, we'll also jump in the next expand and let's see like, you know, by then in the next couple of days, we'll try to conclude whatever we are doing or else it will take one more day extra, maybe in the next, one or two days's max, right? But we'll start doing something in the next sprint definitely. Yeah. That's for sure. I was talking to him because, you know, this is one other thing we need to do anyway. So we need to do it at some point, yeah. It was just reading this. Looks like his defined collections and every collections are different sub names, and every distribution you are in, look, those are distutions. If every distribution is defining data quality versions, schemas, A, if you could document, if you see, there's a lot of details, there are almost like 1,500 line, 15,000 lines, I mean it's just not the contact they defined everything there, right? So definitely like an apparent relation, whatnot, there are so many things going on there. but yep, definitely. I thought like, you know, we can see this info in properly and what do you call. in that. I can, yes, this is very very, very technical, I think. It's not easy to I you're going to join the dot here. of all the appers of components, the Mities, the element level also, you know, lineage, you cannot see from. That you can see also this issue. you can look at the entity and their properties, I would say, not the relations between them, right? Hope that makes sense. So that's yep, fine. I think is define consum contract features tack to defend objects, not in datata contract, but in publisher, in catalogue. Okay, it's in business data catalogue. So we are talking about the subscriptions also, right? Think about the subscription or we's just going to work just in case, right? I think, you know, they mentioned the subscription party even in this document. Take a look at it, like, you know, how it is, it says like in a subscription, consumer and all that stuff, right that I can't that's right. You see that, right? So yeah, yeah to get an idea, like, you know, what they are thinking and what we are thinking, something like that. I think we'll need this what we call this one that view will give us because I can see different. I will try get this hopefully I think this is I think I think because if he is proposing to make some features defined in the catalogue, like the subscription thing, right? I can see he's there, both in the catalog and in contract as well. So is the, you know, data quality features also is defined in catalog and contract as well. So maybe he's, again, I don't know. I'm just guessing that maybe he's proposing those should be defined in the catalogue, but I defined the catalog and use in the contract as a reference, right? Not defined in the contract level, but define a data asset level in a catalogue. Correct. So, yeah, you know, let's I will try to get that to license if I can. I don't know. We'll try to reach out some here. Because, you know, Ser is said like he's going to talk to Ed, if possible, then he will get the license. We'll see. Okay? If we get the license, we can capture those exact picture and I will send it to you if I can by the end out of the tray or else will next week, definitely. Because, you know, Eddie is going to have some discussions next week with this all the EDs and looks like. They have the discussion, they will get some idea like, you know, what exactly they're going to do. And after that it comes down to us, I guess. I don't know whether I will be in that meeting and all, but, you know, at some point we'll be having some meeting with Ed. has defined a lot of details, right? I can see that he has the defined of primary and foreign across all entities also defined Exactly. So the point is, right, if you think about it, this is only the data quality Jason that I have sent to you, but there are other parts, like no d Okay, this is just data quality, Jason quality. That's it. This is just quality. It's not everything. We want me to send everything, I can send it, like it's almost like four. No, I just need quality. There are so many. For Cathal itself, they have one more model and the model is like, I don't know how many lines it is. It's again, a big Jason, more than the quality. It's 19,000 lines. So he has his own catalog model. This is for quality that I has signed you., I mean, no, when you put into the document, this is. Had you written these things? No, no, no. He will not return. What will happen? The tool will generate it, right? All you. I'm saying even in this basic thing, right? What you see here, this,000 lines is has there been written by him or some tool that provides these? Two. What do is you go and drag a picture and you annotate the picture with the different things that you want. Okay, so okay, I got it. It's kind of defining the urban model and then you create some constraints based on what' defining the. And the behind the scenes is backed by adj song like Dave, and this Jason will be used to render the object again at some point. You cannot go and write like this, right? If you see those IDs here and there, those are right kind of random IDs. You cannot get anything out of it. Yeah. Okay, all right. I think I'm going I'm waiting for that. You will be more helpful rather than browsing this thing. Yeah. I mean, I mean, like, you know, we have a good idea, at least, right? What are the details you are telling so far, like, you know. But object by object you know, how we need to write down the morals. I mean, right? APA contracts. If you think about the APA contracts, don't worry about the contract details, right? High-level APs that you are thinking, right? So at least like in a bullet point them here and there in your car somewhere. So that's like, whatever you are envisioning, like, okay, creating a contract, you know, subscribing what, how the subscription different, well, sorry, sorry. What are the APSs for the subscri subscription? If you are a consumer, what are the how you are going to consume these contact? Those kind of details, if you put in the, you know, formal Yeah, I think I think this first first firstly with this concentrate on the. the hunt. Yeah, yeahending the contract and that two producer contract. And then I'm saying producer contract. So I'm talking about So you first, you define the contract first as a producer and you say, like, you know, this is my details and all that sort of cell and everything. And even the detailments everything. Now you are done with your part, right, as a publisher. That's So again, one part is to ability to create a contract for a new data asset, right? Second is modification of a data contract as well, right? So, so for example, if a data contract a data asset is registered in catalogue, right, with those data elements, right? So our API should, gather that pull that information and create the minimum depact contact mandory that we have defined based on those data elements to be created first. So next thing is that we should be able to capture the versions automatically based on this API call, wheneverever there's a change in the data as in the catalogs. say, for example, today, it data set tomorrow, change the SLA. Or I added a new data element and I have not broken the account track, but I have updated that in the data cap log. So we should be able to quantify the existing data contract or change the version, as we call it, an existing data contract with a new data element that has been added there. So that ideally, when we enforce, we have to ensure that the always the latest version of the contractor should be enforced in that case. Let's say, for example, someone, if we don't change that SLA in their contract, every day there will be a number of violations that we identified false violations, right? The data was not at a long time, but then we have to make those changes. So So basically, ability to create a contract through API by reference to CCB. built to modify that, right? From those, right? Third, But before that, before these to the first were defining the metamodal design tables and all those things, build the storage for data contract, right? That is the first stop. And these two will be the second and third step in that side. And then comes the you know, there are other things as like, you know defining data quality rules are built to define the data quality rules in that case. Or, you know, make some changes in the data contract from a UI use the interfaces. Right? So we want to go with the user interface first, we just want to ensure at case we established a connectivity, have some data contracts with us in our data ways and then we can see how we displayed because the next feature for UA perspective would be search and discourbability of data assets to create a data contract from UI. And when I'm saying further from UI, I should be able to search a data asset. I'm searching the data asset also, then the willy clue, you know, look up to the catalog and see that real data assets, right? And then by biical a button, you can create a contract. And when you click on to create a button of a contract and a new, you could see a former wizard in front of you, right? Which is repopulate all this information, which is already there for the data asset, right? And you can, you know, wherever you think the as a user, I want to change this data element, this data quality rule. This SLA and all those things, right? You can make those changes there and submit a contract from UI. So that's the prefection of creato contracts from UI. Even then the modification of contracts are UI, right? And then the ability to view all the existing contracts. Okay. From a desult perspective, right? Okay, for this data, as, these are a contracts find, what are the violations there and all the things also we can identify how violations is a thing? Because the data quality rules that we defined when we create a contract, right? So those data quality rules will be referenced by the K USDK. So that one more AP also needs to build with TDQSDK and data contract with which KSDK will retrieve all the information of data quality rules defined in the contract, and they will learn the data quality check. Right? What data quality check? They run There is the pass or field, then at least we should get that, you know, a result with us to store, whether it's a fail or a pass violation. If it's a violation, then the next action would be that is there an alerting feature configured by a consumer or not, right? For a particular data contract? Alerting feature can be, you know, raising an email or raising a new ticket or GR in that case, depending upon the severity of an alert that has been raised, right? We can give that optional while defining the contract. It's itself that if they want to alert it if this data is a violation, how they want to alert it to who they want to be alerted, right? All those things also will be above that data contact and we create first, right? So it will take a thing. So now we can see that there's one interact in the data because DK, one interaction catalogue as well, and there might be a potential interaction with Jera or you know, tools as that in that case, right? So, again, this alerting alting, we are not going to alert. I think that will be somewhere part of DSI. So we have to check that feature of what you it is defined here. But then we we should be able to enable users to define the electricic mechanism, not actually one those alerts in that case, right in a contract. So those things, so, I think that are there are a lot of, I mean, there are a lot of things, right, as you can. Yeah, so before talking about.. Yeah, first question is that can you the first course look, let's let's give this ability to create a contract with a API.UIAPI. Yeah, all that. Okay. For me, right? For my understanding of the business overall, like in you create a contact as a publisher, if I come in, you know who's going to consume like what is the model like, you know, the subscriptions are going to be in between after that as a hooked into the subscription as a as a consumer, something like that. Is that what the idea is? I don't I don't know understand because when I saw other data contracts products at we have here, right, in this and like we have some we have something like decas our data contract with a service platform. I'm sure somebody might have told you what they are taking consum or interprove. They don't have, you know, subscription-based approach, right? And even the other form by came that they are doing so they also don't have subscription-based no approach. Yeah, we will talk for once we get there, right. I will look like how to look like an implementation level is something we are still brainsting is what I would say. makes sense. Absolutely. Okay. That gives me some idea. Okay, makes sense. Good, good. All right, I think we have that PB at 11 that day. You can bring it app to start with right? And you, ask the team like, you know, we need to we need to start adding some of the giras for the next sprint so that you know, we can at least put some, I mean, we can gather the has some epic, I will replace it here for your reference, right? And you can create what you call this, you know, or respective, you know, stories, spike or, you know, let me show my screen here, right way so that is easier. I've already have this in front of me. One second. I' last the g to join. He just pinged me. I to ask him to join. And to be that's right.,in. And, yeah, I got it. By the way, like an obiliated happy birthday wishes, man. When is your birthday yesterday? Yesterday. It was. Awesome. So what did you do? Anything interesting? No, nothing much. I worked half day and the rest of the half day, just one leave dinner that's. Oh, okay. I' take leave and I don't do that much. Yeah, yeah. I need to take break in board days, right? Take your break your respective, like you no need to work, I would say. Birs and what do you call like, you wedding anniversaries, whatever, vacations. Yeah, absolutely. Nice. All I maj is here. We can quickly ya. I like to know Nitin wanted to bring in some of the stories for the next sprint, right? Like, you know, models or whatever, like, you know, for the data can't that we need to start doing, right? One is definitely thinking about the data model, okay? Let's see, like in what are all the things we can do. Second part is going to be design document that we can prepare, right? Out of the data model and the minimal APS that we can think about it at all of that, like, you know, he captured, I think, the details in the API, whatever the missing stuff he's going to capture anymore.. Okay, go ahead. I mean, I mean, these very very high level, you know like this epic is more from discovery phase at what we want, right? So in this, I think, so this, whatever we are in, whatever spike stories that we're trying to create as part of our a discovery phase B technical or product-based disgrreate, we can attach or can create substies for dess research. So I recall, you know, S said that he want one for, you know, mamoral agreement and sign up for data contracts. So there's one story for that. And Jeremy, if you recollect, if you weeks back, right, we thought that what should be the design work that we need to build up, right? So what I'm trying to say is that we can we can use this epic and create stories, we spike or anything, right, which you want for the analysis work, for, you know, Viet metal design or defining framework. So I forget, even if you're not defining tables now or creating physical tables now, but at least how to look like investigating and whatever time you spend to do that pre-work of actually defying the physical dums, we can raise pikes for that and tag it to this epic in that case. So this is the first one, right? And there's one more, so key. So first, basically what we want is that first create the latest storage, the metab design, defining objects and all those things. We are, we're going to store the data on press information, right? This is the first step. Second step would be to, you know, create contracts. So again, first as create an epic for that as well here, right? So when, when we actually do that, right, creating data elements, tables, and objects, the rooms and all those things, right? Then we can create studies for those here, that to this particular epic. Again, I' made a very generric., he says storage, what does it mean exactly? Again, the model part that we are talking about? Yes, I'm talking about different politics that are indicated like politics.able. Maybe you can say model somewhere on the hiic name., you can again. It's just a placehold on. Okay. July, I think yeah, so is this epic is sp missing is touring. I'm saying defining the metam defining tables, objects, elements, constraints, primary keys, fallen keys, indexing, and all those things in the table that we going to do... That's complic up, right? And next, epic that we have is this one, right? So what we want is it, what we want is that? ability to create a contract by retrieving data, as information from the catalogue, right? So basically what we want is that we want an EPI wherein visual will, again, I don't want to go into a how part. What we want is that we should be able to retrieve data as a details from catalogue, okay? To create a contract. So this epic would be for that, I'll give more, you now the next question would be like, what data elements that you want that should be part of the API field? What should be acceptable values and all this? And so for this case, I think we'll have to have a connect with the catalog thing, right? That if these are intel that we want from data assets, whether they have an API already created, which we can call to pull the information or not, I recall there is a CCM API, which is already in place and they are lifting and shifting that in the data hub as well in that case, right? So, yeah, so basically, whether that will be sufficient for us to have all this information that we need from contract perspective, that I can tell you what data means that we want, right? And if we need any enhancement from the catalog site, we can tell them up front. Look, this is what we want and we have to make those enhancements. So this epic can be used, you know, for that.ving, you know, details of a data asset and, you know, create a contract. Now, if you think that this epic is to big enough, right, when we need to in, again, this one just for not just creation, but if NS, you right, creating modification, all those things will be, you will be, if not part of this epic, I can create other epics as well in that case, but the idea is that, I will integrate with catalog so that we can create contracts as if in a new data is registered in the catalogue, or as in when any data asset goes to any material change adding new data elements changing theid is all those things. So there has to be a mod there has to be different API who that you know rece that information and the changes that happens and st again back to our database with those different versings and all those things.. So this epic is for that, I think, primary write-up, primarily, I think, in next court rights, we might be concentrating in this epic and the metamodel one, I think, this one. totally one. Okay. Makes sense. So of course, like, you know, the ABAA part is going to be a little, I know, it's going to be more and, what do you call it? Like we can add more stories to that. You can keep one AP problem to start with, even though we are trying to connect to the CCB catalogue, that's internal to the the EPS that we are going to build, right? But wherever is possible, you know what we can do is we can create a separate story for that altogether, right? For the mapping exercise or whatever. separate stories like, you know, for all. And we can take it from there. EP wise, you know, I don't know how how usually the team is doing here, but, you know, usually, right, for all the work we use to create one AP and all, but for APs work, we can AP. So please for all the APs work, we can do one one AP. for all the modeling work, we can do, you know, one epic, something like that. In that way, we can we will not miss out basically. What EPS we have been so far and whenever you wanted to do any kind of analys quick right? can go to one one EP communic can check that. So like in this case, right, for this API, this epic for API with catalog Integration, I've created two stories. Again, sometimes back. When's retrieve the data distcription details from system catalogue for data contract, right? So when I'm seeing distant details, the data asset details that we want, like you know data set name, application name, distribution name, data elements, the rule and all the similar been defined. So for this, there is already an AI there, which is the NCC, API, that is sufficient or not, is something that we need to invest phone confirm. So that is one, right? second part is probably the cashing, the details that you want. So when is it earlier that, you know, we need to, you know, identify if there's any modification happening on an existing data asset, right? So how to do that? I recall, you know, some time back Lucas told me that we can do that by cashing functionality when can store it to datater that has been retrieved. And from that cash, we can check whether there has been any change or not. I there is one more point, right? We were thinking about integrating with the catalogues by MSK or on a cock events. and getting the latest information. I don't know whether that party is already done. That's one way of getting the latest and updating the details onto our cash or on our storage, right? Yeah, suicide. But the say that each time if you wanted to go get that data and compare, that's that that's as if like you invoking the CMS APIs, all the time you what I mean? Yeah, exactly. Yeah. So again, we't, I'm not saying that we should do it every time what we want to do here is that, you know, at least, again, I might meet on 100% clear here, but what I was thinking is that at least once a day, right? That the call Shabut once a day and to ensure that, you know, and the change is supposed to be resetting and that they show that and next day when they' that kind ofaka is not enabled that we can do that part. Sure. Yeah. And, you know, make it more sophistic area in real time, if we need to do it, I don't think so. We might need to do that because how frequently someone changes, right? Not every day, twice, twice a day, right, if someone changes in existing data asset in that case, right? So it happens like once a month, right, or once a quarter, whenever a release happens, right? Then only you might change the data element in that case, provided. In that case, there's only change. So, yeah, I mean, so what does I show you is that similar to this, right? Whatever I think I will create those, you know, user stories, right? And wherever you think that we need this technical stories for this particular thing, right? We can create those. Yeah, yeah. It will brainstorm between myself, the Jeremiah and some era, definitely. will come up with something. I mean, but, you know, you're right. Like, you know, these are the epics, right? Under these epics, if you want to, I don't know, whether you want us to create some a story solad, I think there are some story solid ready you' showing me. Sam's created something and or something like that. Yeah, I can read it in a while. Oh, I haven't used it. I end my policeholders story. That's okay. Those are the things. Those are the stories we wanted to pull in the next spring. That's what I'm thinking about, right? Yes. So how I design an agreement, right? Exactly. We just need the placeholders there and pull into an sprint and do some trip work, something like that, even though we just did some kind of outcome of it, that would be great, right?. So one more thing to Unan, right? Do we have a epic, we are creating epics, right? Do we have a parent epic that we are talking about all these things that can't simply, Data quality? I. So we have initiatives for data contract. So under that initiative, we are getting the epic for data contract. But all these five episodes, there are five epics only there has been created. I'll create to. This is the parent epic. The one epic link. If you scroll up on the same thing, Okay. This is epic. What is the epic name there in this? It's. Go to one of the epics. I will tell you what I mean. Any epic that you use. This is epic. Okay, if you see the epic name there, in this screen, if you see that epic name, label under the label, you see the epic name. Yeah. No, no, not that that. On the description point, right? If you see type type equal epic, after that could look critical, then the leg equal to there is epic name there. Okay. So I was talking about that Data product. Can you click on that epic? Go there? I I think this is because it is like that that these epic names is okay. So first, the incurating epic okay, the first epic is created a line, right? And then from there, it gets all. So do align. The one lady showing up is the airline there. doesn't showing up there is a line, I can that's. The point is, like, you start from the higher up, from there, you have a list of epics to organize it basically, right? If somebody want, like, you know, what are the epics you're working on, like and we can say, like and go to here and that that. Yes, yes, we can have it. Yeah. Yeah. organ nice. No worries. All right. Sounds good. We will pick up something Nitin indefinitely. We have PBR in 30 minutes. We can bring this up. I don't know how many PBS we are going to have. Today is one and tomorrow.. This epic probably we might not be able to discuss too much details on this epic because we don't have that much information, we are we're just going to say that we are going to work on some spike stories for met terminal design agreement in that case. So mean, how much amount of time would be estimation all we cannot do it now. But I think, I guess it's up to you guys, right? So basically, two epics, one is for model, one is for design work, right? Design document that we wanted to produce something, you know, irrespective. So we'll create those two epics, sorry, spikes, I mean, spikes and know, we' pull in between myself and Jeremiah will take care of those things, like, you know, of course, we'll close, I don't know, will try to put in the details there. Yeah. Yeah, and this is a pick I' helping you so that you are. This is 2168, yeah. Oh. Okay. Awesome. Sounds good. Yeah. Looking forward to Vart on these contracts, man. Iron Man, like in, we thought of starting last week, but you know, with we are kind of waiting for that AID stuff because we need to respect whatever they're doing and, you know, ensure like, you know, we are airligned with their modeling, right? So it's of deviating and, I think. Yep. Okay. All right, cool in. Yeah, okay. Thank you. I think probably, if not immediately, we'll have to engage with a catalogue team as well. You, not now immediately, but at least the next week or next, next week, so that we know that what and all, you know, how they are strolling that information, what AP is available to review the information, from APS stuff, right? So that if at all we think that they have to make some changes that they are in, so that we can pull their information from cat clock, right? So they can do their priation and all the. No, I think you already map it, right? What is the information available in the catalog more or less?. I have, but I think I still have to just know, given that they are also building it making some changes in the catalogue rightes. I just wanted to ensure that whatever we think like those data elements are very much there and what are the individual data? I don't want to look, when we start working on that API story let's let's see, for example, of the next print, right? So I don't want that there is an open dependency wherein capital optum comes and say that, okay, this feature or this data field is not there with us, right? We need to develop a build that that might require some estimation effort there and is, right. So for man is that identifying any piece of work needs to do so that we can pull all this contact information from the PPI. Okay, Sounds good. Okay. Okay. Cool. Yeah, yeah, All right. Thank you. Appreciate it. I know.

**ChatGPT said:**

Got it 👍 — let me break this down for you in structured way because you’re juggling **producer-oriented contracts**, **consumer customizations**, and **contract violations**.

**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**.

**4. Contract Violation Definition**

* **Violation is relative to contract agreed by that consumer.**
  + Example:
    - DQ = 92%
    - Producer contract = 90%
    - Consumer A requires 99% → Violation for A ✅
    - Consumer B requires 90% → No violation ❌
* Means: One dataset delivery can be **valid for some consumers, invalid for others**.

**5. Metamodel / Storage Tables (Option B extended)**

Here’s how you can model it:

**Producer Contract Tables**

* contract\_producer (id, asset\_id, version, created\_at)
* contract\_rule (rule\_id, producer\_contract\_id, element\_id, rule\_type, threshold, severity)

**Consumer Sub-Contract Tables**

* contract\_consumer (id, producer\_contract\_id, consumer\_id, version, created\_at)
* contract\_consumer\_override (consumer\_contract\_id, rule\_id, override\_threshold, override\_sla, severity\_override)

**Violations Table**

* contract\_violation (id, contract\_id, consumer\_id, asset\_id, rule\_id, violation\_type, observed\_value, timestamp, severity)

**6. Subscription Layer**

* Subscription = consumer expresses interest in producer contract.
* Subscription references:
  + **Producer Contract ID** (mandatory)
  + **Consumer Overrides** (optional)
* This creates the effective contract for enforcement.

**7. Workflow**

1. Producer publishes contract.
2. Consumer subscribes → optional overrides → consumer contract.
3. Enforcement engine validates each data exchange:
   * Check rules → store results → log violations.
   * Apply consumer overrides → determine consumer-specific violation.
4. Alerting (optional) triggered.

👉 Question for you: Do you want me to **expand the table schema** into a full YAML + JSON Schema (like earlier) for this **Producer + Consumer + Violation model**? That way you’ll have the **contract + enforcement + violation storage design all together**.

**You said:**

Yea

**ChatGPT said:**

Perfect 👍 let’s stitch everything together into one coherent model that covers **producers**, **consumers**, and **violations**.

**1. Contract YAML (Producer + Consumer Override)**

Here’s how a producer contract can declare defaults, with consumer overrides layered on top.

contract:

id: payments.transactions

version: "1.0.0"

owner: team:payments

schema:

compatibility: backward

fields:

- name: transaction\_id

type: string

required: true

- name: amount

type: decimal(18,2)

required: true

- name: event\_time

type: timestamp

required: true

dq\_rules:

- id: completeness

metric: completeness

threshold: 0.95

severity: high

- id: timeliness

metric: arrival\_time

max\_delay\_minutes: 30

severity: medium

slas:

ready\_by: "09:00Z"

completeness\_min: 0.95

volume\_floor: 4\_500\_000

deviation\_pct: 10

producers:

- svc:txn-writer

consumers:

- id: svc:ledger

overrides:

- rule\_id: completeness

threshold: 0.99 # stricter

severity: critical

- id: svc:recon

overrides:

- rule\_id: timeliness

max\_delay\_minutes: 15 # stricter

**2. JSON Schema (Validation of YAML Contracts)**

Here’s the schema to validate the above structure.

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "Data Contract Schema with Overrides",

"type": "object",

"required": ["contract"],

"properties": {

"contract": {

"type": "object",

"required": ["id", "version", "owner", "schema", "dq\_rules", "slas", "producers"],

"properties": {

"id": { "type": "string" },

"version": { "type": "string", "pattern": "^[0-9]+\\.[0-9]+\\.[0-9]+$" },

"owner": { "type": "string" },

"schema": {

"type": "object",

"required": ["fields"],

"properties": {

"compatibility": { "enum": ["backward", "forward", "none"] },

"fields": {

"type": "array",

"items": {

"type": "object",

"required": ["name", "type"],

"properties": {

"name": { "type": "string" },

"type": { "type": "string" },

"required": { "type": "boolean" }

}

}

}

}

},

"dq\_rules": {

"type": "array",

"items": {

"type": "object",

"required": ["id", "metric"],

"properties": {

"id": { "type": "string" },

"metric": { "type": "string" },

"threshold": { "type": "number" },

"max\_delay\_minutes": { "type": "integer" },

"severity": { "type": "string" }

}

}

},

"slas": {

"type": "object",

"required": ["ready\_by", "completeness\_min"],

"properties": {

"ready\_by": { "type": "string" },

"completeness\_min": { "type": "number" },

"volume\_floor": { "type": "integer" },

"deviation\_pct": { "type": "integer" }

}

},

"producers": {

"type": "array",

"items": { "type": "string" }

},

"consumers": {

"type": "array",

"items": {

"type": "object",

"required": ["id"],

"properties": {

"id": { "type": "string" },

"overrides": {

"type": "array",

"items": {

"type": "object",

"properties": {

"rule\_id": { "type": "string" },

"threshold": { "type": "number" },

"max\_delay\_minutes": { "type": "integer" },

"severity": { "type": "string" }

}

}

}

}

}

}

}

}

}

}

**3. Relational Tables**

Now the SQL model that implements this.

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

-- Contracts (logical dataset identity)

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

CREATE TABLE contracts (

id TEXT PRIMARY KEY,

owner\_team TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

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

-- Contract Versions (immutable snapshots)

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

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

producers TEXT[] NOT NULL DEFAULT '{}',

created\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

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

-- DQ Rules (defined by producer contract)

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

CREATE TABLE dq\_rules (

rule\_id TEXT NOT NULL,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

metric TEXT NOT NULL,

threshold NUMERIC,

max\_delay\_minutes INT,

severity TEXT,

PRIMARY KEY (contract\_id, version, snapshot\_hash, rule\_id),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- Consumers (subscriptions with overrides)

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

CREATE TABLE contract\_consumers (

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_id TEXT NOT NULL,

subscribed\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash, consumer\_id),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

CREATE TABLE consumer\_overrides (

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_id TEXT NOT NULL,

rule\_id TEXT NOT NULL,

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (contract\_id, version, snapshot\_hash, consumer\_id, rule\_id),

FOREIGN KEY (contract\_id, version, snapshot\_hash, consumer\_id)

REFERENCES contract\_consumers(contract\_id, version, snapshot\_hash, consumer\_id),

FOREIGN KEY (contract\_id, version, snapshot\_hash, rule\_id)

REFERENCES dq\_rules(contract\_id, version, snapshot\_hash, rule\_id)

);

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

-- Executions (runtime validations)

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

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

contract\_version TEXT NOT NULL,

contract\_snapshot\_hash TEXT NOT NULL,

producer\_id TEXT NOT NULL,

consumer\_id TEXT,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

results JSONB,

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

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- Violations (results per rule, per consumer)

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

CREATE TABLE contract\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

rule\_id TEXT NOT NULL,

consumer\_id TEXT, -- null if producer baseline violation

observed\_value NUMERIC,

expected\_value NUMERIC,

severity TEXT,

timestamp TIMESTAMP DEFAULT now(),

FOREIGN KEY (contract\_id, version, snapshot\_hash, rule\_id)

REFERENCES dq\_rules(contract\_id, version, snapshot\_hash, rule\_id)

);

**4. Flow Recap**

* **Producer defines contract** → stored in contracts, contract\_versions, dq\_rules.
* **Consumers subscribe** → recorded in contract\_consumers.
* **Consumers override rules** → stored in consumer\_overrides.
* **Executions** validate data against producer rules + consumer overrides.
* **Violations** recorded per rule per consumer.

✅ This ties **YAML → JSON Schema → Relational tables → Execution + Violations** into one consistent model.

Do you want me to also add an **example SQL query** like:  
“Show all violations for svc:ledger against payments.transactions@1.0.0 in the last 7 days”?

**You said:**

So we need subscription table

**ChatGPT said:**

Exactly 👍 — you need a **subscription table** because that’s what ties a **consumer** to a **producer’s contract version**.

Think of it this way:

* The **producer contract** is the “offer.”
* A **consumer subscription** is the “acceptance.”
* Together, they form the **agreement** you enforce at runtime.

**Why a Subscription Table?**

* **Decoupling:** Consumers subscribe to a producer contract version (or range).
* **Overrides:** Subscription is where you attach consumer-specific overrides (stricter thresholds, timeliness, etc.).
* **Auditability:** You can track *when* a consumer subscribed and *what version* they were bound to.
* **Lifecycle:** If producer updates contract → consumers must explicitly re-subscribe or be auto-migrated.

**Suggested Table Design**

**contract\_subscriptions**

Links consumers to a contract snapshot.

CREATE TABLE contract\_subscriptions (

subscription\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_id TEXT NOT NULL, -- e.g. "svc:ledger"

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

**subscription\_overrides**

Holds consumer-specific rule customizations.

CREATE TABLE subscription\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id TEXT NOT NULL,

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, rule\_id)

);

**Benefits of this Design**

* A consumer can subscribe to **multiple producer contracts**.
* A producer contract can have **many consumers**.
* Overrides live **per subscription**, not globally.
* Execution engine just looks up **active subscriptions** to know which consumer checks apply.

**Flow with Subscriptions**

1. Producer publishes contract payments.transactions@1.0.0.
2. Consumer svc:ledger subscribes → entry in contract\_subscriptions.
3. If svc:ledger needs stricter rules → subscription\_overrides.
4. When executions run:
   * Validate against **producer rules**.
   * Apply **subscription overrides** per consumer.
   * Log violations in contract\_violations.

✅ So yes — the subscription table is **essential** to manage the many-to-many relationship between producers and consumers cleanly.

Would you like me to **redraw the full schema** (producer contracts + subscriptions + overrides + executions + violations) in a compact SQL block, so you can see it all stitched together with subscriptions at the center?

**You said:**

Design whole thing again

**ChatGPT said:**

Perfect — let’s do a **clean slate design** that stitches together everything we’ve discussed: **producer contracts, consumers, subscriptions, overrides, executions, and violations**.

**📑 End-to-End Contract Registry Schema**

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

-- 1. Contracts (logical dataset identity)

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

CREATE TABLE contracts (

id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL, -- accountable team

created\_at TIMESTAMP NOT NULL DEFAULT now()

);

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

-- 2. Contract Versions (immutable snapshots published by producers)

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

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL, -- SemVer, e.g. "1.0.0"

snapshot\_hash TEXT NOT NULL, -- SHA256 of full YAML/JSON snapshot

yaml\_content JSONB NOT NULL, -- frozen contract definition

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

producers TEXT[] NOT NULL DEFAULT '{}', -- allowed producers

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP NOT NULL DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

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

-- 3. Rules (per version snapshot)

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

CREATE TABLE contract\_rules (

rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

element TEXT NOT NULL, -- column/field name

rule\_type TEXT NOT NULL, -- schema | dq | timeliness | pii | ...

threshold NUMERIC, -- e.g. 95.0 for DQ %

max\_delay\_minutes INT, -- for timeliness rules

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 4. Subscriptions (consumers binding to contracts)

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

CREATE TABLE contract\_subscriptions (

subscription\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_id TEXT NOT NULL, -- e.g. "svc:ledger"

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 5. Subscription Overrides (consumer-specific adjustments)

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

CREATE TABLE subscription\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, rule\_id)

);

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

-- 6. Executions (runtime validations / deliveries)

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

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

producer\_id TEXT NOT NULL,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

results JSONB, -- aggregated validation results

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 7. Violations (per rule, per subscription, per execution)

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

CREATE TABLE contract\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

observed\_value NUMERIC,

violation\_type TEXT NOT NULL, -- schema\_mismatch | dq\_failure | sla\_breach ...

severity TEXT NOT NULL,

occurred\_at TIMESTAMP NOT NULL DEFAULT now()

);

**🔑 How It All Fits Together**

1. **Producer publishes contract** → stored in contracts + contract\_versions + contract\_rules.
2. **Consumers subscribe** → contract\_subscriptions.
   * Optionally attach stricter requirements via subscription\_overrides.
3. **Execution runs** (batch, stream, API call) → executions.
4. **Validation engine** checks rules:
   * Base rules from producer contract.
   * Apply overrides from subscription (if any).
5. **Violations** logged in contract\_violations.
   * Each violation is tied to the subscription (so violations are *consumer-specific*).

**✅ Benefits**

* **Immutable history** → all validations tied to exact contract snapshot.
* **Many-to-many** → one producer → many consumers; one consumer → many producer contracts.
* **Consumer-specific requirements** without duplicating contracts.
* **Audit-ready** → you can answer *who subscribed to what, when, under which rules, and what violations occurred*.

Would you like me to also **draft a sample YAML + JSON Schema** for the **subscription layer** (so you can see how a consumer declares overrides on top of a producer contract)?

**You said:**

Creating contracts at product level

**ChatGPT said:**

Got it 👍 — now we’re moving **one layer higher in abstraction**.  
So far, we’ve been designing contracts at the **dataset level** (e.g. payments.transactions).  
But in a real organization, you often want to define contracts at the **product (domain) level**, and then break them down into datasets, events, or APIs.

**🔹 Why Product-Level Contracts?**

* **Ownership** → Products map to business domains (Payments, Lending, KYC).
* **Consistency** → Shared rules (naming, PII policies, SLAs) apply across all assets in a product.
* **Reusability** → Instead of repeating the same rules across 10 datasets, define them once at the product level.
* **Governance** → Easier to enforce compliance when you anchor contracts to products, not just datasets.

**🔹 Layered Contract Model**

1. **Product Contract**
   * High-level definition of business domain.
   * Applies **global rules** (data policies, SLA policies, DQ baselines).
   * Example: Product = Payments.
2. **Dataset/Event Contracts (children)**
   * Specific datasets, events, or APIs within the product.
   * Inherit product-level rules.
   * Can add **dataset-specific rules**.
   * Example: payments.transactions, payments.refunds.
3. **Consumer Subscriptions**
   * Consumers subscribe to **dataset/event-level contracts**.
   * Still respect **product-level rules** as baseline.

**🔹 Extended Schema (Product Level)**

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

-- Products (business domains)

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

CREATE TABLE products (

id TEXT PRIMARY KEY, -- e.g. "payments"

name TEXT NOT NULL,

owner\_team TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

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

-- Product Contracts (global rules for the product)

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

CREATE TABLE product\_contracts (

product\_id TEXT NOT NULL REFERENCES products(id),

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL, -- full definition (rules, policies)

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

created\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (product\_id, version, snapshot\_hash)

);

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

-- Product Rules

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

CREATE TABLE product\_rules (

rule\_id UUID PRIMARY KEY,

product\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

rule\_type TEXT NOT NULL, -- dq\_baseline | pii\_policy | naming | sla\_default ...

rule\_expression JSONB, -- flexible definition (regex, threshold, policy ref)

severity TEXT NOT NULL,

FOREIGN KEY (product\_id, version, snapshot\_hash)

REFERENCES product\_contracts(product\_id, version, snapshot\_hash)

);

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

-- Link product → dataset contracts (inheritance)

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

ALTER TABLE contracts

ADD COLUMN product\_id TEXT REFERENCES products(id);

**🔹 How It Works in Practice**

* **Step 1: Product owner** publishes Payments Product Contract v1.0.
  + Rules:
    - All datasets must use prefix pay\_.
    - All PII fields must be masked at rest.
    - DQ threshold baseline = 95%.
* **Step 2: Dataset team** creates payments.transactions@1.0.
  + Inherits product rules.
  + Adds specific rules (e.g., transaction\_amount >= 0, currency in [USD, EUR]).
* **Step 3: Consumer team** subscribes to payments.transactions.
  + Inherits product rules + dataset rules.
  + Adds overrides if needed (e.g., DQ >= 99%).
* **Step 4: Execution** validates dataset delivery:
  + Check product rules.
  + Check dataset rules.
  + Apply consumer overrides.

**✅ Benefits**

* Governance is anchored at **product/domain level**.
* Shared policies (PII, naming, baseline DQ) are **centralized**.
* Dataset contracts remain **lightweight**, focusing only on specifics.
* Enforcement engine naturally layers rules:  
  **Product → Dataset → Consumer**.

Would you like me to **re-draw the SQL schema** with **products, product contracts, dataset contracts, and subscriptions all stitched together** — so you can see the final hierarchy (Product → Dataset → Consumer)?

**You said:**

Go without product level contracts

**ChatGPT said:**

Got it ✅ — let’s simplify back to **dataset-level contracts only**, no product-level abstraction.  
This means:

* **Contract = Dataset-level definition** (schema, SLAs, DQ rules).
* **Consumers** subscribe directly to dataset contracts.
* **Overrides** (consumer-specific stricter needs) sit at subscription level.
* **Executions** validate deliveries against these rules.
* **Violations** log failures per consumer per execution.

**📑 Final Schema (Dataset-Level Only)**

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

-- 1. Contracts (logical dataset identity)

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

CREATE TABLE contracts (

id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL, -- accountable team

created\_at TIMESTAMP NOT NULL DEFAULT now()

);

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

-- 2. Contract Versions (immutable snapshots published by producers)

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

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL, -- SemVer, e.g. "1.0.0"

snapshot\_hash TEXT NOT NULL, -- SHA256 of full YAML/JSON snapshot

yaml\_content JSONB NOT NULL, -- frozen contract definition

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

producers TEXT[] NOT NULL DEFAULT '{}', -- allowed producers

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP NOT NULL DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

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

-- 3. Rules (per version snapshot)

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

CREATE TABLE contract\_rules (

rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

element TEXT NOT NULL, -- column/field name

rule\_type TEXT NOT NULL, -- schema | dq | timeliness | pii | ...

threshold NUMERIC, -- e.g. 95.0 for DQ %

max\_delay\_minutes INT, -- for timeliness rules

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 4. Subscriptions (consumers binding to contracts)

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

CREATE TABLE contract\_subscriptions (

subscription\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_id TEXT NOT NULL, -- e.g. "svc:ledger"

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 5. Subscription Overrides (consumer-specific adjustments)

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

CREATE TABLE subscription\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, rule\_id)

);

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

-- 6. Executions (runtime validations / deliveries)

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

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

producer\_id TEXT NOT NULL,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

results JSONB, -- aggregated validation results

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 7. Violations (per rule, per subscription, per execution)

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

CREATE TABLE contract\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

observed\_value NUMERIC,

violation\_type TEXT NOT NULL, -- schema\_mismatch | dq\_failure | sla\_breach ...

severity TEXT NOT NULL,

occurred\_at TIMESTAMP NOT NULL DEFAULT now()

);

**🔑 Flow Recap (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.

👉 This is the **leanest, dataset-only model**.  
No product/domain layer — everything is anchored at dataset contract level.

Do you want me to also show you an **example YAML + JSON Schema** for a dataset-level contract (with rules + overrides), so you can see how it maps into these tables?

**You said:**

Include publisher(application) , consumer (application), access and use, Recovery objectives

**ChatGPT said:**

Perfect — you want to extend the **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.

**📑 Extended Dataset-Level Contract Schema**

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

-- 1. Contracts (dataset identity)

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

CREATE TABLE contracts (

id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL, -- accountable team

created\_at TIMESTAMP NOT NULL DEFAULT now()

);

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

-- 2. Contract Versions (immutable snapshots)

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

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL, -- SemVer

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL, -- full frozen definition

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

publishers TEXT[] NOT NULL DEFAULT '{}', -- publishing applications/services

access\_modes TEXT[] NOT NULL DEFAULT '{}', -- e.g. ['API','Batch','Streaming']

allowed\_uses TEXT[] NOT NULL DEFAULT '{}', -- e.g. ['Analytics','ML','Compliance']

rpo\_minutes INT, -- Recovery Point Objective

rto\_minutes INT, -- Recovery Time Objective

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP NOT NULL DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

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

-- 3. Rules (validation + SLA)

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

CREATE TABLE contract\_rules (

rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

element TEXT NOT NULL, -- field name

rule\_type TEXT NOT NULL, -- schema | dq | timeliness | pii | ...

threshold NUMERIC,

max\_delay\_minutes INT,

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 4. Subscriptions (consumers binding to contract versions)

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

CREATE TABLE contract\_subscriptions (

subscription\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_app TEXT NOT NULL, -- consuming application/service

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 5. Subscription Overrides (consumer-specific stricter needs)

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

CREATE TABLE subscription\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, rule\_id)

);

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

-- 6. Executions (runtime validations)

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

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

publisher\_app TEXT NOT NULL, -- who ran this execution (producer)

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

results JSONB,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 7. Violations (consumer-specific failures per execution)

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

CREATE TABLE contract\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

observed\_value NUMERIC,

violation\_type TEXT NOT NULL,

severity TEXT NOT NULL,

occurred\_at TIMESTAMP NOT NULL DEFAULT now()

);

**🔑 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.

**✅ Example**

* Contract: payments.transactions@1.0
  + Publisher apps: ["svc:payments-core"]
  + Access modes: ["API","Streaming"]
  + Allowed uses: ["Analytics","Compliance"]
  + RPO = 15 min, RTO = 60 min
* Consumer svc:ledger subscribes → wants DQ >= 99% (override).
* Execution 2025-09-13T12:00:
  + Publisher = svc:payments-core
  + Checks pass baseline but fail ledger’s stricter override.
  + Violation logged against svc:ledger subscription.

Would you like me to also **draft a sample contract YAML** (showing publisher, consumer, access/usage, RPO/RTO, and rules) that would map into this schema?

**You said:**

Include technical data quality and business data quality

**ChatGPT said:**

Nice 👌 — you want to distinguish **Technical Data Quality (TDQ)** vs **Business Data Quality (BDQ)** in the contract model.  
That’s an important separation because:

* **Technical DQ (system-level checks)** → schema validity, null % thresholds, timeliness, duplicates, type mismatches.
* **Business DQ (semantic checks)** → domain rules, reference integrity, business constraints (e.g., “amount >= 0”, “status ∈ {open, closed}”).

Let’s extend the schema to capture both.

**📑 Extended Dataset-Level Contract Schema (with TDQ + BDQ)**

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

-- 1. Contracts (dataset identity)

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

CREATE TABLE contracts (

id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL,

created\_at TIMESTAMP NOT NULL DEFAULT now()

);

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

-- 2. Contract Versions (immutable snapshots)

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

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

publishers TEXT[] NOT NULL DEFAULT '{}', -- publishing apps

access\_modes TEXT[] NOT NULL DEFAULT '{}', -- API | Batch | Streaming

allowed\_uses TEXT[] NOT NULL DEFAULT '{}', -- Analytics | ML | Compliance

rpo\_minutes INT, -- Recovery Point Objective

rto\_minutes INT, -- Recovery Time Objective

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP NOT NULL DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

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

-- 3. Rules (now tagged as TDQ vs BDQ)

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

CREATE TABLE contract\_rules (

rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

element TEXT NOT NULL, -- column/field name

rule\_category TEXT NOT NULL CHECK (rule\_category IN ('Technical','Business')),

rule\_type TEXT NOT NULL, -- schema | null\_check | timeliness | value\_domain | business\_constraint

threshold NUMERIC,

max\_delay\_minutes INT,

rule\_expression JSONB, -- for complex BDQ checks (e.g. SQL expression)

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 4. Subscriptions (consumer applications)

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

CREATE TABLE contract\_subscriptions (

subscription\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_app TEXT NOT NULL,

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 5. Subscription Overrides (stricter consumer-specific rules)

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

CREATE TABLE subscription\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, rule\_id)

);

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

-- 6. Executions (runtime validations)

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

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

publisher\_app TEXT NOT NULL,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

results JSONB, -- aggregated validation results (TDQ + BDQ)

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 7. Violations (per rule, per subscription, per execution)

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

CREATE TABLE contract\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

rule\_id UUID NOT NULL REFERENCES contract\_rules(rule\_id),

observed\_value NUMERIC,

violation\_type TEXT NOT NULL, -- schema\_mismatch | dq\_failure | business\_logic ...

rule\_category TEXT NOT NULL, -- 'Technical' or 'Business'

severity TEXT NOT NULL,

occurred\_at TIMESTAMP NOT NULL DEFAULT now()

);

**🔑 Key Additions**

* contract\_rules.rule\_category → distinguishes TDQ vs BDQ.
* rule\_expression → JSON for richer BDQ logic (could embed SQL, regex, or DSL).
* contract\_violations.rule\_category → so you know if a violation is **technical** or **business**.
* executions.results → can hold a breakdown (tdq\_pass\_rate, bdq\_pass\_rate).

**✅ Example**

**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).

👉 Do you want me to also **draft a sample YAML contract** showing how TDQ and BDQ rules look in practice (with publishers, consumers, access, RPO/RTO) so it directly maps into this schema?

**You said:**

Separate tables for tdq and Bdq

**ChatGPT said:**

Perfect 👍 — splitting **Technical Data Quality (TDQ)** and **Business Data Quality (BDQ)** rules into separate tables makes governance cleaner, queries simpler, and avoids overloading one rules table.

Here’s the redesigned schema with that separation:

**📑 Dataset Contract Schema with TDQ + BDQ Tables**

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

-- 1. Contracts (dataset identity)

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

CREATE TABLE contracts (

id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL,

created\_at TIMESTAMP NOT NULL DEFAULT now()

);

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

-- 2. Contract Versions

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

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

publishers TEXT[] NOT NULL DEFAULT '{}', -- publishing apps

access\_modes TEXT[] NOT NULL DEFAULT '{}', -- API | Batch | Streaming

allowed\_uses TEXT[] NOT NULL DEFAULT '{}', -- Analytics | ML | Compliance

rpo\_minutes INT, -- Recovery Point Objective

rto\_minutes INT, -- Recovery Time Objective

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP NOT NULL DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

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

-- 3A. Technical Data Quality (TDQ) Rules

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

CREATE TABLE contract\_tdq\_rules (

tdq\_rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

element TEXT NOT NULL, -- column/field name

rule\_type TEXT NOT NULL, -- schema | null\_check | duplicates | timeliness

threshold NUMERIC, -- % nulls allowed, duplicate ratio, etc.

max\_delay\_minutes INT, -- for timeliness checks

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 3B. Business Data Quality (BDQ) Rules

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

CREATE TABLE contract\_bdq\_rules (

bdq\_rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

business\_name TEXT NOT NULL, -- business concept (e.g. "amount rule")

rule\_type TEXT NOT NULL, -- value\_domain | reference\_integrity | business\_constraint

rule\_expression JSONB NOT NULL, -- flexible definition: SQL, regex, DSL

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 4. Subscriptions (consumer applications)

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

CREATE TABLE contract\_subscriptions (

subscription\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_app TEXT NOT NULL,

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 5A. TDQ Subscription Overrides

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

CREATE TABLE subscription\_tdq\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

tdq\_rule\_id UUID NOT NULL REFERENCES contract\_tdq\_rules(tdq\_rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, tdq\_rule\_id)

);

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

-- 5B. BDQ Subscription Overrides

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

CREATE TABLE subscription\_bdq\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

bdq\_rule\_id UUID NOT NULL REFERENCES contract\_bdq\_rules(bdq\_rule\_id),

override\_expression JSONB, -- allow stricter consumer-specific business rule

override\_severity TEXT,

PRIMARY KEY (subscription\_id, bdq\_rule\_id)

);

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

-- 6. Executions (runtime validations)

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

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

publisher\_app TEXT NOT NULL,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

tdq\_results JSONB, -- summary: pass %, failed checks

bdq\_results JSONB, -- summary: pass %, failed checks

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

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

-- 7A. TDQ Violations

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

CREATE TABLE tdq\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

tdq\_rule\_id UUID NOT NULL REFERENCES contract\_tdq\_rules(tdq\_rule\_id),

observed\_value NUMERIC,

violation\_type TEXT NOT NULL, -- schema\_mismatch | dq\_failure | timeliness

severity TEXT NOT NULL,

occurred\_at TIMESTAMP NOT NULL DEFAULT now()

);

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

-- 7B. BDQ Violations

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

CREATE TABLE bdq\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

bdq\_rule\_id UUID NOT NULL REFERENCES contract\_bdq\_rules(bdq\_rule\_id),

observed\_value JSONB,

violation\_type TEXT NOT NULL, -- business\_logic | value\_domain | reference\_integrity

severity TEXT NOT NULL,

occurred\_at TIMESTAMP NOT NULL DEFAULT now()

);

**🔑 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.

👉 Do you want me to also **draft a YAML contract example** where TDQ and BDQ rules are explicitly separated — so you can see how they map into these tables?

**You said:**

Include alert and re model everything

**ChatGPT said:**

Got it ✅ — let’s **remodel everything cleanly from scratch**, now including **alerts** in the design.

We’ll separate **TDQ** and **BDQ**, keep **producers/consumers**, add **subscriptions and overrides**, track **executions and violations**, and introduce **alerts** so stakeholders (publisher, consumer, ops) can react in real time.

**🏗️ End-to-End Data Contract Model (with Alerts)**

**1. Contracts & Versions**

-- Datasets that are governed by contracts

CREATE TABLE contracts (

id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

-- Immutable contract snapshots

CREATE TABLE contract\_versions (

contract\_id TEXT NOT NULL REFERENCES contracts(id),

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

publishers TEXT[] NOT NULL DEFAULT '{}', -- publishing apps

access\_modes TEXT[] NOT NULL DEFAULT '{}', -- API | Batch | Streaming

allowed\_uses TEXT[] NOT NULL DEFAULT '{}', -- Analytics | ML | Compliance

rpo\_minutes INT, -- Recovery Point Objective

rto\_minutes INT, -- Recovery Time Objective

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

**2. Rules (TDQ vs BDQ)**

-- Technical Data Quality Rules

CREATE TABLE contract\_tdq\_rules (

tdq\_rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

element TEXT NOT NULL, -- column/field name

rule\_type TEXT NOT NULL, -- schema | null\_check | duplicates | timeliness

threshold NUMERIC, -- % allowed (e.g. null ≤ 1%)

max\_delay\_minutes INT, -- timeliness threshold

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

-- Business Data Quality Rules

CREATE TABLE contract\_bdq\_rules (

bdq\_rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

business\_name TEXT NOT NULL, -- "amount rule"

rule\_type TEXT NOT NULL, -- value\_domain | reference\_integrity | business\_constraint

rule\_expression JSONB NOT NULL, -- DSL, SQL, regex, lookup reference

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

**3. Subscriptions & Overrides**

-- Consumer subscriptions

CREATE TABLE contract\_subscriptions (

subscription\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

consumer\_app TEXT NOT NULL,

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

-- TDQ overrides (consumer stricter checks)

CREATE TABLE subscription\_tdq\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

tdq\_rule\_id UUID NOT NULL REFERENCES contract\_tdq\_rules(tdq\_rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, tdq\_rule\_id)

);

-- BDQ overrides (consumer stricter rules)

CREATE TABLE subscription\_bdq\_overrides (

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

bdq\_rule\_id UUID NOT NULL REFERENCES contract\_bdq\_rules(bdq\_rule\_id),

override\_expression JSONB,

override\_severity TEXT,

PRIMARY KEY (subscription\_id, bdq\_rule\_id)

);

**4. Executions & Violations**

-- Execution runs (batch or streaming validation cycles)

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

publisher\_app TEXT NOT NULL,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

tdq\_results JSONB, -- summary: pass %, failed checks

bdq\_results JSONB, -- summary: pass %, failed checks

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES contract\_versions(contract\_id, version, snapshot\_hash)

);

-- TDQ Violations

CREATE TABLE tdq\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

tdq\_rule\_id UUID NOT NULL REFERENCES contract\_tdq\_rules(tdq\_rule\_id),

observed\_value NUMERIC,

violation\_type TEXT NOT NULL, -- schema\_mismatch | dq\_failure | timeliness

severity TEXT NOT NULL,

occurred\_at TIMESTAMP DEFAULT now()

);

-- BDQ Violations

CREATE TABLE bdq\_violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

subscription\_id UUID NOT NULL REFERENCES contract\_subscriptions(subscription\_id),

bdq\_rule\_id UUID NOT NULL REFERENCES contract\_bdq\_rules(bdq\_rule\_id),

observed\_value JSONB,

violation\_type TEXT NOT NULL, -- business\_logic | value\_domain | reference\_integrity

severity TEXT NOT NULL,

occurred\_at TIMESTAMP DEFAULT now()

);

**5. Alerts (NEW 🔔)**

-- Alerts triggered by violations

CREATE TABLE alerts (

alert\_id UUID PRIMARY KEY,

violation\_id UUID, -- either TDQ or BDQ violation

contract\_id TEXT NOT NULL,

subscription\_id UUID,

execution\_id UUID,

severity TEXT NOT NULL CHECK (severity IN ('Info','Warn','Error','Critical')),

alert\_channel TEXT NOT NULL, -- email | slack | pagerduty | webhook

alert\_message TEXT NOT NULL,

acknowledged BOOLEAN DEFAULT false,

acknowledged\_by TEXT,

acknowledged\_at TIMESTAMP,

created\_at TIMESTAMP DEFAULT now()

);

**🔄 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.

👉 Do you also want me to **draft a sample YAML contract** (with TDQ + BDQ rules, publishers, consumers, access, RPO/RTO, and alert policies) so you can see how it flows into this schema?

**You said:**

Can we do main contract for producer but option to define consumer contract where they can override

**ChatGPT said:**

Exactly 👍 — that’s the right way to think about **producer vs consumer contracts**.  
It avoids every consumer trying to redefine the world, while still letting them tighten requirements.

Let’s reframe the model:

**🏗️ Producer vs Consumer Contracts**

**🔹 Producer Contract (Main Contract)**

* Owned by the **publisher application/team**.
* Defines the **base schema, TDQ rules, BDQ rules, RPO/RTO, allowed uses, access modes**.
* Immutable once versioned (v1.0, v1.1, …).
* Every consumer inherits this contract.

**🔹 Consumer Contracts (Overrides)**

* Consumer apps create a **subscription** to a producer contract.
* They may add **stricter overrides**, but cannot weaken producer rules.
* Overrides apply only to their validation scope.
* Examples:
  + Producer rule: *null\_rate ≤ 5%*
  + Consumer override: *null\_rate ≤ 1%*
  + Producer rule: *currency ∈ {USD, EUR}*
  + Consumer override: *currency = USD only*

**📑 Schema (Remodeled)**

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

-- Producer Contracts

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

CREATE TABLE producer\_contracts (

contract\_id TEXT PRIMARY KEY, -- "payments.transactions"

owner\_team TEXT NOT NULL,

publisher\_app TEXT NOT NULL, -- owning app

created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE producer\_contract\_versions (

contract\_id TEXT NOT NULL REFERENCES producer\_contracts(contract\_id),

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

access\_modes TEXT[] NOT NULL DEFAULT '{}', -- API | Batch | Streaming

allowed\_uses TEXT[] NOT NULL DEFAULT '{}', -- Analytics | ML | Compliance

rpo\_minutes INT,

rto\_minutes INT,

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

**🔹 Rules**

-- Technical rules for producer

CREATE TABLE producer\_tdq\_rules (

tdq\_rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

element TEXT NOT NULL,

rule\_type TEXT NOT NULL,

threshold NUMERIC,

max\_delay\_minutes INT,

severity TEXT NOT NULL,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES producer\_contract\_versions(contract\_id, version, snapshot\_hash)

);

-- Business rules for producer

CREATE TABLE producer\_bdq\_rules (

bdq\_rule\_id UUID PRIMARY KEY,

contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

business\_name TEXT NOT NULL,

rule\_type TEXT NOT NULL,

rule\_expression JSONB NOT NULL,

severity TEXT NOT NULL,

FOREIGN KEY (contract\_id, version, snapshot\_hash)

REFERENCES producer\_contract\_versions(contract\_id, version, snapshot\_hash)

);

**🔹 Consumer Contracts (Subscriptions with Overrides)**

-- Consumers bind to a producer contract

CREATE TABLE consumer\_contracts (

consumer\_contract\_id UUID PRIMARY KEY,

producer\_contract\_id TEXT NOT NULL REFERENCES producer\_contracts(contract\_id),

version TEXT NOT NULL, -- must align with producer version

snapshot\_hash TEXT NOT NULL,

consumer\_app TEXT NOT NULL,

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true

);

-- TDQ overrides

CREATE TABLE consumer\_tdq\_overrides (

consumer\_contract\_id UUID NOT NULL REFERENCES consumer\_contracts(consumer\_contract\_id),

tdq\_rule\_id UUID NOT NULL REFERENCES producer\_tdq\_rules(tdq\_rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (consumer\_contract\_id, tdq\_rule\_id)

);

-- BDQ overrides

CREATE TABLE consumer\_bdq\_overrides (

consumer\_contract\_id UUID NOT NULL REFERENCES consumer\_contracts(consumer\_contract\_id),

bdq\_rule\_id UUID NOT NULL REFERENCES producer\_bdq\_rules(bdq\_rule\_id),

override\_expression JSONB,

override\_severity TEXT,

PRIMARY KEY (consumer\_contract\_id, bdq\_rule\_id)

);

**🔹 Execution & Alerts**

-- Validations (executions) run per producer contract

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

producer\_contract\_id TEXT NOT NULL,

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

publisher\_app TEXT NOT NULL,

started\_at TIMESTAMP NOT NULL,

completed\_at TIMESTAMP,

status TEXT NOT NULL CHECK (status IN ('Running','Success','Failed')),

tdq\_results JSONB,

bdq\_results JSONB,

FOREIGN KEY (producer\_contract\_id, version, snapshot\_hash)

REFERENCES producer\_contract\_versions(contract\_id, version, snapshot\_hash)

);

-- Violations (per consumer, based on their overrides)

CREATE TABLE violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

consumer\_contract\_id UUID NOT NULL REFERENCES consumer\_contracts(consumer\_contract\_id),

rule\_type TEXT NOT NULL, -- TDQ | BDQ

rule\_ref UUID NOT NULL, -- points to producer rule

observed\_value JSONB,

violation\_type TEXT NOT NULL,

severity TEXT NOT NULL,

occurred\_at TIMESTAMP DEFAULT now()

);

-- Alerts

CREATE TABLE alerts (

alert\_id UUID PRIMARY KEY,

violation\_id UUID NOT NULL REFERENCES violations(violation\_id),

producer\_contract\_id TEXT NOT NULL,

consumer\_contract\_id UUID,

severity TEXT NOT NULL,

alert\_channel TEXT NOT NULL, -- email | slack | pagerduty | webhook

alert\_message TEXT NOT NULL,

acknowledged BOOLEAN DEFAULT false,

acknowledged\_by TEXT,

acknowledged\_at TIMESTAMP,

created\_at TIMESTAMP DEFAULT now()

);

**🔄 Flow**

1. **Producer team** defines contract (payments.transactions@1.0).
2. **Rules** (TDQ/BDQ) apply globally.
3. **Consumers** subscribe and create **consumer contracts** with overrides.
   * Producer null\_rate ≤ 5% → Consumer null\_rate ≤ 1%
   * Producer currency ∈ {USD, EUR} → Consumer currency = USD only
4. **Executions** validate incoming data against producer rules.
5. **Violations** logged per consumer contract (overrides included).
6. **Alerts** routed to relevant producers/consumers.

✅ This way:

* Producers define the **minimum guarantees**.
* Consumers can demand **higher quality** without breaking others.
* Alerts can be scoped (producer-wide issue vs consumer-specific stricter failure).

Would you like me to also **sketch a YAML structure** showing how a producer defines rules and a consumer contract references + overrides them? That way you’ll see the JSON/YAML side that maps to these tables.

**You said:**

Include data access and usage

**ChatGPT said:**

Perfect — let’s extend the **producer/consumer contract model** with **data access** (how consumers get the data) and **usage** (what they are allowed to do with it).

**🏗️ Producer + Consumer Contracts with Access & Usage**

**1. Producer Contract**

The **source of truth** — owned by the publisher.  
Defines:

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

CREATE TABLE producer\_contracts (

contract\_id TEXT PRIMARY KEY, -- e.g. "payments.transactions"

owner\_team TEXT NOT NULL,

publisher\_app TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE producer\_contract\_versions (

contract\_id TEXT NOT NULL REFERENCES producer\_contracts(contract\_id),

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

yaml\_content JSONB NOT NULL,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

access\_modes TEXT[] NOT NULL DEFAULT '{}', -- API | Batch | Streaming

allowed\_uses TEXT[] NOT NULL DEFAULT '{}', -- Analytics | ML | Compliance

rpo\_minutes INT,

rto\_minutes INT,

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

created\_at TIMESTAMP DEFAULT now(),

PRIMARY KEY (contract\_id, version, snapshot\_hash)

);

**2. Rules (Producer-Level)**

Split into **TDQ** and **BDQ**, as before.  
(unchanged from earlier, so I won’t repeat here — but both reference producer\_contract\_versions.)

**3. Consumer Contracts (Subscriptions with Overrides)**

Consumers **bind to a producer contract version**, inherit all rules, and may:

* Override thresholds (stricter only).
* Specify their **data access method** (how they receive data).
* Declare **intended use** (why they use the data).

-- Consumers create their own contracts linked to producer contract

CREATE TABLE consumer\_contracts (

consumer\_contract\_id UUID PRIMARY KEY,

producer\_contract\_id TEXT NOT NULL REFERENCES producer\_contracts(contract\_id),

version TEXT NOT NULL, -- aligns with producer version

snapshot\_hash TEXT NOT NULL,

consumer\_app TEXT NOT NULL,

data\_access\_mode TEXT NOT NULL CHECK (data\_access\_mode IN ('API','Batch','Streaming')),

intended\_uses TEXT[] NOT NULL, -- Analytics | ML | Compliance | Product

subscribed\_at TIMESTAMP DEFAULT now(),

active BOOLEAN DEFAULT true

);

-- TDQ overrides (stricter checks)

CREATE TABLE consumer\_tdq\_overrides (

consumer\_contract\_id UUID NOT NULL REFERENCES consumer\_contracts(consumer\_contract\_id),

tdq\_rule\_id UUID NOT NULL REFERENCES producer\_tdq\_rules(tdq\_rule\_id),

override\_threshold NUMERIC,

override\_max\_delay\_minutes INT,

override\_severity TEXT,

PRIMARY KEY (consumer\_contract\_id, tdq\_rule\_id)

);

-- BDQ overrides (stricter business constraints)

CREATE TABLE consumer\_bdq\_overrides (

consumer\_contract\_id UUID NOT NULL REFERENCES consumer\_contracts(consumer\_contract\_id),

bdq\_rule\_id UUID NOT NULL REFERENCES producer\_bdq\_rules(bdq\_rule\_id),

override\_expression JSONB,

override\_severity TEXT,

PRIMARY KEY (consumer\_contract\_id, bdq\_rule\_id)

);

**4. Execution & Violations**

(Same as before — execution runs validate producer contract rules + consumer overrides.)

**5. Alerts**

(Same as before — alerts triggered on violations, routed to producer or consumer stakeholders.)

**🔄 Example Lifecycle with Access & Usage**

1. **Producer contract** (payments.transactions@1.0)
   * Publisher app = payments-service
   * Access modes = {Streaming, Batch}
   * Allowed uses = {Analytics, ML, Compliance}
   * RPO = 15 min, RTO = 60 min
   * TDQ rules: schema, null ≤ 5%, timeliness ≤ 5 min
   * BDQ rules: amount ≥ 0, currency in {USD, EUR}
2. **Consumer contract** (risk-engine)
   * Inherits from producer contract
   * Data access mode = Streaming
   * Intended uses = Compliance
   * Overrides: null ≤ 1%, timeliness ≤ 1 min
3. **Execution** runs → validates producer rules, applies consumer overrides.
4. **Violations**: risk-engine gets flagged if stricter thresholds fail.
5. **Alerts**:
   * Producer violation (schema mismatch) → notify publisher + all consumers.
   * Consumer-specific stricter violation (null >1% but ≤5%) → alert only risk-engine.

✅ With this model:

* Producer sets **baseline guarantees** (contract).
* Consumers bind with their own **access + use intent**.
* Overrides make consumer expectations explicit.
* Violations + alerts respect both sides.

Would you like me to **draft a YAML contract pair** (producer + consumer) to illustrate how access, usage, and overrides look in practice?

**You said:**

Meta model with all

**ChatGPT said:**

Got it 👍 — you want the **meta-model** that ties everything together:  
producers, contracts, rules (TDQ/BDQ), consumers with overrides, access, usage, executions, and alerts.  
Let’s design this as a **conceptual meta-model** (entities + relationships), not just SQL — so it’s clear how the whole system works.

**📐 Meta-Model for Data Contracts**

**🔹 Core Entities**

1. **Producer Application**
   * Publishes data.
   * Owns one or more **Producer Contracts**.
2. **Producer Contract**
   * Anchored to a dataset/event/API.
   * Versioned.
   * Defines:
     + Schema (fields, types, PII flags)
     + **TDQ rules** (technical quality)
     + **BDQ rules** (business quality)
     + SLAs (RPO, RTO)
     + Access modes (API, Batch, Streaming)
     + Allowed uses (Analytics, ML, Compliance, etc.)
3. **TDQ Rules**
   * Technical constraints (null %, timeliness, record count).
   * Linked to producer contract version.
4. **BDQ Rules**
   * Business/domain constraints (valid values, referential integrity).
   * Linked to producer contract version.
5. **Consumer Application**
   * Subscribes to producer contract.
   * May create **Consumer Contract**.
6. **Consumer Contract**
   * References producer contract version.
   * Declares:
     + Data access mode (how it consumes).
     + Intended use (why it consumes).
   * May override producer rules with **stricter thresholds only**.
7. **Consumer Overrides**
   * Separate objects pointing back to producer TDQ/BDQ rules.
   * Store stricter thresholds, custom expressions, or severity levels.
8. **Executions**
   * Run periodically (or on data arrival).
   * Apply producer rules + consumer overrides.
   * Record violations.
9. **Violations**
   * Captured from executions.
   * Reference which rule failed (TDQ/BDQ).
   * Tagged as **producer violation** (baseline broken) or **consumer violation** (override broken).
10. **Alerts**
    * Generated from violations.
    * Routed differently:
      + Producer violation → notify producer + all consumers.
      + Consumer violation → notify that consumer only.
    * Can carry severity, notification channel, recovery workflow link.

**🔹 Relationships (high level)**

Producer App ───< Producer Contract ───< TDQ Rules

│

└──< BDQ Rules

│

└──< Consumer Contracts ───< Overrides (TDQ/BDQ)

│

Consumer App ──────────────────────────────────────────────┘

Executions ───< Violations ───< Alerts

**🔹 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**.

Do you want me to **draw this meta-model as an ER diagram (boxes + relations)** so you can visualize the entities, or keep it textual only?

**You said:**

Can we keep just create one contract table and use it for both producer and consumer, essentially consumer contract is just a child reference to parent

**ChatGPT said:**

Yes 💯 — that’s a really elegant simplification.  
Instead of splitting into *producer\_contracts* and *consumer\_contracts*, you can model **all contracts in one table**, and then rely on **self-references** to distinguish producer vs. consumer.

**🏗️ 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.

**🔹 Schema (Unified)**

CREATE TABLE contracts (

contract\_id UUID PRIMARY KEY,

parent\_contract\_id UUID NULL REFERENCES contracts(contract\_id),

dataset\_name TEXT NOT NULL, -- e.g. "payments.transactions"

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

owner\_app TEXT NOT NULL, -- producer or consumer application

role TEXT NOT NULL CHECK (role IN ('Producer','Consumer')),

access\_modes TEXT[] NOT NULL DEFAULT '{}', -- API | Batch | Streaming

intended\_uses TEXT[] NOT NULL DEFAULT '{}', -- Analytics | ML | Compliance | Product

rpo\_minutes INT,

rto\_minutes INT,

yaml\_content JSONB NOT NULL,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

created\_at TIMESTAMP DEFAULT now()

);

-- TDQ Rules (per contract, producer or consumer)

CREATE TABLE tdq\_rules (

rule\_id UUID PRIMARY KEY,

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

rule\_name TEXT NOT NULL,

rule\_expression JSONB NOT NULL,

severity TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

-- BDQ Rules (per contract, producer or consumer)

CREATE TABLE bdq\_rules (

rule\_id UUID PRIMARY KEY,

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

rule\_name TEXT NOT NULL,

rule\_expression JSONB NOT NULL,

severity TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

-- Executions (runs validating rules)

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

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

run\_time TIMESTAMP NOT NULL,

status TEXT NOT NULL CHECK (status IN ('Pass','Fail','Partial')),

record\_count BIGINT,

processed\_at TIMESTAMP DEFAULT now()

);

-- Violations

CREATE TABLE violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

rule\_id UUID NOT NULL, -- can be TDQ or BDQ rule

severity TEXT NOT NULL,

details JSONB,

created\_at TIMESTAMP DEFAULT now()

);

-- Alerts

CREATE TABLE alerts (

alert\_id UUID PRIMARY KEY,

violation\_id UUID NOT NULL REFERENCES violations(violation\_id),

notified\_apps TEXT[] NOT NULL, -- producer app, consumer app(s)

channel TEXT NOT NULL, -- Slack, Email, PagerDuty

created\_at TIMESTAMP DEFAULT now(),

resolved\_at TIMESTAMP

);

**🔹 How It Works**

1. **Producer contract**
   * role = 'Producer', parent\_contract\_id = NULL.
   * Defines schema + baseline TDQ/BDQ rules + SLAs + access modes + allowed uses.
2. **Consumer contract**
   * role = 'Consumer', parent\_contract\_id = <producer\_contract\_id>.
   * Inherits producer rules.
   * May attach its own TDQ/BDQ rules (only stricter).
   * Declares access method + intended use.
3. **Execution**
   * Runs against producer contract data.
   * Evaluates both producer and consumer rules.
4. **Violations & Alerts**
   * If producer baseline fails → alert producer + all consumers.
   * If only consumer override fails → alert that consumer only.

**🔹 Example**

* Contract #1 (Producer):
  + Dataset: payments.transactions
  + App: payments-service
  + Null ≤ 5%, Timeliness ≤ 5 min.
  + Access = {Batch, Streaming}, Uses = {Analytics, ML}.
* Contract #2 (Consumer):
  + Parent = #1
  + App: risk-engine
  + Overrides: Null ≤ 1%, Timeliness ≤ 1 min.
  + Access = Streaming, Use = Compliance.

✅ Advantages:

* **Unified design** → no duplication of producer/consumer schema.
* **Tree structure** → one producer, many consumers.
* **Simple rule handling** → attach rules directly to any contract.
* **Clear inheritance** → consumer overrides just become stricter child rules.

Would you like me to also show how **inheritance resolution works** (e.g., how to compute the *effective contract* a consumer sees by combining producer baseline + consumer overrides)?

**You said:**

Include data access and usage tables

**ChatGPT said:**

Perfect 👌 — if we keep the **unified contracts table**, we can normalize **Data Access** and **Usage** into separate child tables. That way we don’t overload the contract row, and we can track multiple access modes (API + Batch) or multiple usage intents (Analytics + Compliance) cleanly.

Here’s the re-modeled version:

**🏗️ Unified Contract Meta-Model (with Access & Usage)**

**🔹 Core Contract Table**

CREATE TABLE contracts (

contract\_id UUID PRIMARY KEY,

parent\_contract\_id UUID NULL REFERENCES contracts(contract\_id), -- producer → consumer link

dataset\_name TEXT NOT NULL, -- e.g. "payments.transactions"

version TEXT NOT NULL,

snapshot\_hash TEXT NOT NULL,

owner\_app TEXT NOT NULL, -- producer or consumer application

role TEXT NOT NULL CHECK (role IN ('Producer','Consumer')),

rpo\_minutes INT, -- Recovery Point Objective

rto\_minutes INT, -- Recovery Time Objective

yaml\_content JSONB NOT NULL, -- full contract definition

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

created\_at TIMESTAMP DEFAULT now()

);

**🔹 Data Access Table**

Represents **how** a producer exposes or a consumer subscribes.

CREATE TABLE contract\_access (

access\_id UUID PRIMARY KEY,

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

mode TEXT NOT NULL CHECK (mode IN ('API','Batch','Streaming','FileDrop')),

endpoint TEXT, -- e.g. API URL, Kafka topic, S3 bucket path

format TEXT, -- e.g. JSON, Avro, Parquet

frequency TEXT, -- e.g. "real-time", "hourly", "daily"

created\_at TIMESTAMP DEFAULT now()

);

**🔹 Data Usage Table**

Represents **why** the data is used.

CREATE TABLE contract\_usage (

usage\_id UUID PRIMARY KEY,

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

purpose TEXT NOT NULL CHECK (purpose IN

('Analytics','ML','Compliance','Reporting','ProductFeature')),

description TEXT,

restrictions JSONB, -- e.g. {"pii": "mask", "geo": "US-only"}

created\_at TIMESTAMP DEFAULT now()

);

**🔹 Rules Tables**

-- Technical DQ Rules

CREATE TABLE tdq\_rules (

rule\_id UUID PRIMARY KEY,

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

rule\_name TEXT NOT NULL,

rule\_expression JSONB NOT NULL, -- threshold, regex, percentile, etc.

severity TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

-- Business DQ Rules

CREATE TABLE bdq\_rules (

rule\_id UUID PRIMARY KEY,

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

rule\_name TEXT NOT NULL,

rule\_expression JSONB NOT NULL, -- enum, lookup, foreign key ref, etc.

severity TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

**🔹 Execution, Violations, Alerts**

-- Contract execution results

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

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

run\_time TIMESTAMP NOT NULL,

status TEXT NOT NULL CHECK (status IN ('Pass','Fail','Partial')),

record\_count BIGINT,

processed\_at TIMESTAMP DEFAULT now()

);

-- Violations

CREATE TABLE violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

rule\_id UUID NOT NULL, -- FK to tdq\_rules or bdq\_rules

severity TEXT NOT NULL,

details JSONB,

created\_at TIMESTAMP DEFAULT now()

);

-- Alerts triggered from violations

CREATE TABLE alerts (

alert\_id UUID PRIMARY KEY,

violation\_id UUID NOT NULL REFERENCES violations(violation\_id),

notified\_apps TEXT[] NOT NULL, -- producer app, consumer app(s)

channel TEXT NOT NULL, -- Slack, Email, PagerDuty

created\_at TIMESTAMP DEFAULT now(),

resolved\_at TIMESTAMP

);

**🔄 Flow with Access & Usage**

1. **Producer contract (root)**
   * Contract row: role=Producer.
   * contract\_access: {Batch → S3, Streaming → Kafka}.
   * contract\_usage: {Analytics, ML}.
   * TDQ/BDQ rules baseline.
2. **Consumer contract (child)**
   * Contract row: role=Consumer, parent\_contract\_id=Producer.
   * contract\_access: {Streaming → Kafka}.
   * contract\_usage: {Compliance}.
   * TDQ override: null ≤ 1%.
3. **Execution** validates rules.
4. **Violations** recorded (producer baseline or consumer stricter).
5. **Alerts** routed based on violation type.

✅ This way:

* **contracts** = hierarchy (producer/consumer).
* **contract\_access** = technical how.
* **contract\_usage** = business why.
* Rules, executions, violations, alerts stay the same.

Do you want me to also show an **example YAML/JSON contract** that reflects this design (with producer + consumer, including access & usage sections)?

**You said:**

Include versions with above structure

**ChatGPT said:**

Great point 👍 — versioning needs to be explicit in the unified design so we can track changes and consumers can “lock” to a producer contract version.

Here’s how we can extend the unified model to include **versions** while still keeping access, usage, TDQ, BDQ, executions, violations, and alerts.

**🏗️ Unified Contract Meta-Model with Versions**

**🔹 Contracts (entity identity)**

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

CREATE TABLE contracts (

contract\_id UUID PRIMARY KEY,

parent\_contract\_id UUID NULL REFERENCES contracts(contract\_id),

dataset\_name TEXT NOT NULL, -- e.g. "payments.transactions"

owner\_app TEXT NOT NULL, -- producer or consumer application

role TEXT NOT NULL CHECK (role IN ('Producer','Consumer')),

created\_at TIMESTAMP DEFAULT now()

);

**🔹 Contract Versions**

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

CREATE TABLE contract\_versions (

version\_id UUID PRIMARY KEY,

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

version TEXT NOT NULL, -- e.g. "1.0", "1.1"

snapshot\_hash TEXT NOT NULL, -- content hash for immutability

yaml\_content JSONB NOT NULL, -- full contract definition

rpo\_minutes INT,

rto\_minutes INT,

state TEXT NOT NULL CHECK (state IN

('Draft','Proposed','Approved','Active','Deprecated','Retired')),

created\_at TIMESTAMP DEFAULT now(),

activated\_at TIMESTAMP,

deprecated\_at TIMESTAMP,

retired\_at TIMESTAMP,

UNIQUE (contract\_id, version, snapshot\_hash)

);

**🔹 Access & Usage (Versioned)**

CREATE TABLE contract\_access (

access\_id UUID PRIMARY KEY,

version\_id UUID NOT NULL REFERENCES contract\_versions(version\_id),

mode TEXT NOT NULL CHECK (mode IN ('API','Batch','Streaming','FileDrop')),

endpoint TEXT,

format TEXT, -- e.g. JSON, Avro, Parquet

frequency TEXT, -- e.g. "real-time", "hourly"

created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE contract\_usage (

usage\_id UUID PRIMARY KEY,

version\_id UUID NOT NULL REFERENCES contract\_versions(version\_id),

purpose TEXT NOT NULL CHECK (purpose IN

('Analytics','ML','Compliance','Reporting','ProductFeature')),

description TEXT,

restrictions JSONB, -- e.g. {"pii":"mask","geo":"US-only"}

created\_at TIMESTAMP DEFAULT now()

);

**🔹 Rules (Versioned)**

CREATE TABLE tdq\_rules (

rule\_id UUID PRIMARY KEY,

version\_id UUID NOT NULL REFERENCES contract\_versions(version\_id),

rule\_name TEXT NOT NULL,

rule\_expression JSONB NOT NULL,

severity TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE bdq\_rules (

rule\_id UUID PRIMARY KEY,

version\_id UUID NOT NULL REFERENCES contract\_versions(version\_id),

rule\_name TEXT NOT NULL,

rule\_expression JSONB NOT NULL,

severity TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

**🔹 Executions, Violations, Alerts (Versioned)**

CREATE TABLE executions (

execution\_id UUID PRIMARY KEY,

version\_id UUID NOT NULL REFERENCES contract\_versions(version\_id),

run\_time TIMESTAMP NOT NULL,

status TEXT NOT NULL CHECK (status IN ('Pass','Fail','Partial')),

record\_count BIGINT,

processed\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE violations (

violation\_id UUID PRIMARY KEY,

execution\_id UUID NOT NULL REFERENCES executions(execution\_id),

rule\_id UUID NOT NULL, -- FK to tdq\_rules or bdq\_rules

severity TEXT NOT NULL,

details JSONB,

created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE alerts (

alert\_id UUID PRIMARY KEY,

violation\_id UUID NOT NULL REFERENCES violations(violation\_id),

notified\_apps TEXT[] NOT NULL,

channel TEXT NOT NULL,

created\_at TIMESTAMP DEFAULT now(),

resolved\_at TIMESTAMP

);

**🔄 How It Plays Together**

1. **Producer Contract**
   * contracts row with role=Producer.
   * contract\_versions: v1.0 with schema, SLA, baseline TDQ/BDQ, access (Batch + Streaming), usage (Analytics + ML).
2. **Consumer Contract**
   * contracts row with role=Consumer, parent\_contract\_id = producer’s contract\_id.
   * contract\_versions: v1.0 with stricter overrides, own access (Streaming), usage (Compliance).
3. **Execution**
   * Runs against a **version\_id**.
   * Applies rules from that version (including overrides if consumer).
4. **Violation & Alerts**
   * If producer baseline fails → alert producer + all consumers bound to that producer contract version.
   * If only consumer stricter rule fails → alert that consumer only.

✅ Benefits of this structure:

* **Single unified table** for contracts (hierarchy via parent).
* **Versions are first-class citizens** (immutability, lifecycle management).
* **Access & Usage** tied to version (since they can evolve over time).
* **Rules + executions** also versioned → ensures historical traceability.

Would you like me to **walk through an end-to-end sample** (producer + consumer YAML contract definitions, both stored in this versioned model)?

**You said:**

Do we need version in all tables?

**ChatGPT said:**

Great question 👌 — short answer: **no, you don’t need an explicit version column in every table** if you model correctly.  
The **contract\_versions** table should be the central “anchor” for versioning, and everything else (access, usage, rules, executions, violations, alerts) just points to a specific version\_id.

**🔹 Why not put version everywhere?**

* **Normalization**: If you keep version in every child table (tdq\_rules.version, bdq\_rules.version, etc.), you risk **redundancy** and **inconsistency** (e.g., mismatched version numbers).
* **Central control**: contract\_versions is the **only place** that knows about version strings (1.0, 1.1, etc.).
* **Traceability**: All other tables only need a version\_id (FK). That gives you lineage back to the producer/consumer and its version metadata.

**🔹 Where Version is Needed**

✅ **Keep version only in:**

* contract\_versions (so you can human-read versions like 1.0, 1.1).
* Possibly in contracts (only if you want to store the current active version reference).

❌ **Don’t duplicate version in:**

* contract\_access
* contract\_usage
* tdq\_rules
* bdq\_rules
* executions
* violations
* alerts

These should all reference version\_id, which ties back to a specific immutable contract version.

**🔹 Example Flow**

1. **Producer contract v1.0** → stored in contract\_versions (id=101, version='1.0').
   * tdq\_rules.version\_id=101
   * contract\_access.version\_id=101
2. **Producer publishes v1.1** → new row in contract\_versions (id=102, version='1.1').
   * New tdq\_rules, bdq\_rules, contract\_access linked to 102.
   * Old rules remain tied to 101.
3. **Execution run** references version\_id=102.
   * Violations and alerts flow from that context.
4. If a consumer is still on v1.0 → they stay tied to 101 until they explicitly subscribe to 102.

✅ This keeps the model **clean**: only contract\_versions has the explicit version string, everything else just “inherits” via FK.

Do you want me to show you a **minimal meta-model diagram** (in text form, like ERD style) with version\_id relationships so you can visualize how all tables connect?

**You said:**

Do we need to put version id in all tables

**ChatGPT said:**

You’ve zeroed in on the key modeling trade-off 👍.

The answer depends on what each table **represents**:

**🔹 General Rule**

* If a table’s data is **immutable and tied to a specific contract definition**, then yes → it needs a version\_id FK.
* If a table’s data is **runtime/execution data**, then it should point to version\_id **indirectly** (through execution or violation) instead of carrying it everywhere.

**🔹 Where version\_id is required**

These represent **contract definition state** and must be tied directly to a version:

* **contract\_access** → because access endpoints can change per version.
* **contract\_usage** → usage policies may evolve across versions.
* **tdq\_rules** → rule sets evolve per version.
* **bdq\_rules** → same as above.

So: ✅ keep version\_id in these.

**🔹 Where version\_id is optional/indirect**

* **executions** → should reference a version\_id directly (because execution must be evaluated against the exact contract version).
* **violations** → should reference execution\_id, which already points to a version\_id. No need to duplicate.
* **alerts** → should reference violation\_id (and get to version through violation → execution).

So: ❌ don’t put version\_id directly into violations or alerts unless you want denormalization for performance.

**🔹 Meta-Model Summary**

* **contracts** (identity)
* **contract\_versions** (version metadata, RPO/RTO, YAML, lifecycle)
* **contract\_access** (FK → version\_id)
* **contract\_usage** (FK → version\_id)
* **tdq\_rules** (FK → version\_id)
* **bdq\_rules** (FK → version\_id)
* **executions** (FK → version\_id)
* **violations** (FK → execution\_id → version\_id implicit)
* **alerts** (FK → violation\_id → execution\_id → version\_id implicit)

✅ This way:

* All **definition-time artifacts** (access, usage, rules) are versioned explicitly.
* All **runtime artifacts** (executions, violations, alerts) only tie into version indirectly.

Would you like me to sketch a **step-by-step example query** (like: “show me all alerts for contract X, version 1.0”), to illustrate how these relationships play out in practice?