

Chapter 2

A Novel Architecture for T0-Certified Feature Stores (The Governance Layer)

Revision date: 2026-01-14

Abstract. Modern marketplace ML systems are rarely limited by model class; they fail in the data plane - through silent schema drift, unstable joins, and, most critically, *time leakage* that allows features to encode information unavailable at the decision time t_0 . This chapter presents a governance architecture for *T0-certified feature stores*: a family of SQL entrypoints whose outputs are provably constrained to be functions of the information available at t_0 , monitored for drift, and blocked from consumption when stale or non-compliant. We formalize the T0 requirement in filtration language, specify a contract model that binds feature schemas to cryptographic hashes, and describe a certification program that combines static view-closure analysis, dynamic invariants, and drift-tolerant baselining. The result is an operationally enforceable guarantee: model training and scoring cannot proceed unless every upstream feature block is both *fresh* and *leak-safe* under the declared policy.

2.1 Why governance is the bottleneck in high-frequency marketplace ML

Marketplace pricing and liquidity models operate in an adversarial environment: the market itself evolves, data is partially observed, and the platform continuously re-labels historical outcomes as new events arrive. In this setting, the primary threat to scientific validity is not overfitting in the learner; it is *time inconsistency* in the feature pipeline. A seemingly innocuous join, materialized view refresh, or aggregation window can allow post-decision information to enter the training matrix. The resulting offline metrics are inflated, hyperparameter tuning selects fragile configurations, and production performance degrades precisely where the tail risk is highest.

The governance layer presented in this chapter treats leakage and drift as first-class failure modes. Instead of trusting each feature author to remember every temporal edge case, the system enforces a small set of non-negotiable invariants at the boundary between data production and model consumption.

Design goals

- **T0 safety:** every feature must be computable from information available at decision time t_0 .
- **Determinism:** rerunning the same feature query over the same upstream data must produce identical results.
- **Contractual stability:** changes to schemas and feature sets must be explicit, versioned, and hash-addressed.
- **Operational enforceability:** training and scoring must be blocked when a feature block is stale or non-compliant.
- **Drift observability:** slow, expected drift should be measurable; unexpected drift should trigger alarms.
- **Composable stores:** multiple specialized feature stores must integrate without re-introducing leakage.

These goals imply a governance mechanism that is not merely documentation. It must be executable, automatable, and capable of failing closed.

2.2 Formalizing T0-compliance as measurability

Let t_0 denote the decision time at which the model is evaluated (e.g., listing edited time). Let D be the underlying event data: listings, edits, image arrivals, derived signals, and reference dimensions. Define a filtration $\{F_t\}$ representing the information available up to time t .

$$x_i(t_0) = g_i(D_{\leq t_0}), \quad x_i(t_0) \in \mathcal{F}_{t_0}$$

Equation 2.1 - T0 compliance as measurability with respect to the information filtration.

Leakage occurs when a feature depends on D after t_0 , even indirectly. For example, a daily median computed over a calendar day that extends past t_0 is not F_{t_0} -measurable. Similarly, a geo enrichment that uses the latest mapping table without as-of semantics can silently rewrite historical rows.

Definition 2.1 (T0-certified entrypoint). A feature store entrypoint view V is *T0-certified* if, for every row keyed by $(\text{generation}, \text{finn_id}, t_0)$, the emitted feature vector is a deterministic function of only the upstream data available at or before t_0 , and the view is subject to active enforcement and drift monitoring under a declared certification policy.

2.3 Architecture: certified entrypoints as the only interface to learning

The governance layer is designed around a single architectural constraint: **models never query raw tables or intermediate materialized views directly**. Instead, every feature block exposes a *certified entrypoint* view that is treated as the only legal boundary for consumption. All training and scoring jobs use these entrypoints, and every entrypoint is subject to certification checks and access control.

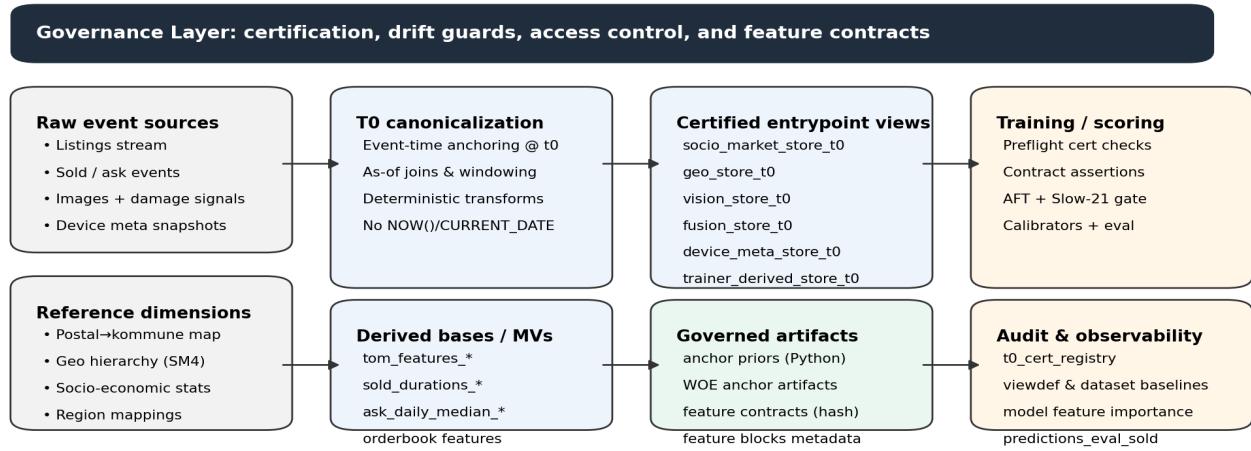


Figure 2.1 - End-to-end feature architecture with an explicit governance layer.

This design decouples *feature computation* from *feature eligibility*. Engineers may iterate on derived materialized views and intermediate transformations, but the system only exposes an entrypoint once it can be certified. Conversely, certification can be enforced without re-running expensive refresh pipelines: the check is lightweight, consulting only the certification registry and drift baselines.

2.4 Data contracts: binding feature schemas to immutable hashes

Certification prevents temporal leakage, but it does not prevent *silent interface drift*: a store may remain leak-safe while adding, removing, renaming, or re-encoding columns. In a multi-store system, these changes can invalidate trained models without an obvious operational fault. To address this, the governance layer introduces a **feature contract** for every feature block.

$$c = \text{SHA256}(\text{block} \parallel \text{feature_set} \parallel \text{schema})$$

Equation 2.2 - Contract hashes bind the declared feature interface to an immutable digest.

A contract is an explicit statement of the feature block interface: the block name, feature set identifier, and the ordered schema (including names, dtypes, and any categorical level dictionaries when applicable). The contract is fingerprinted into a cryptographic hash and stored in the database. At training time, the pipeline asserts that each consumed block matches the expected contract hash. If any block is inconsistent, training fails closed.

Contract enforcement in the training pipeline

In the training entrypoint, contracts are asserted before the dataset is loaded. The model configuration therefore becomes reproducible: the run is a function of (a) the model code revision and (b) the set of contract hashes.

```
def assert_feature_contract(conn, *, block_name, feature_set, expected_hash):
    sql = "SELECT ml.assert_feature_contract(%s, %s, %s)"
    with conn.cursor() as cur:
        cur.execute(sql, (block_name, feature_set, expected_hash))
        ok = bool(cur.fetchone()[0])
    if not ok:
        raise RuntimeError(f"Feature contract mismatch: {block_name} ")
```

Listing 2.1 - Contract assertion pattern in the training pipeline.

This pattern turns interface stability into a runtime invariant rather than an informal promise. It also enables clean rollback: reverting a feature store to a previous certified contract restores compatibility without modifying model code.

2.5 Certification primitives: registry, baselines, and guardrails

Contracts specify *what* is consumed; certification specifies *when* and *under what temporal guarantees* it may be consumed. The certification system is implemented as three persistent tables plus store-specific certification procedures. Together, they define a minimal governance kernel that any feature store can opt into.

Registry and baseline tables

The governance kernel maintains: (i) a registry that records certification status and freshness, (ii) a view-definition baseline that detects DDL drift, and (iii) a dataset hash baseline that detects semantic drift in outputs. These tables are owned by the audit schema and are append-only or controlled by stored procedures.

Object	Purpose	Key fields (illustrative)
audit.t0_cert_registry	Single source of truth for whether an entrypoint is certified and how fresh it is.	entrypoint, status, certified_at, max_age_hours, viewdef_objects, dataset_days, notes
audit.t0_viewdef_baseline	DDL drift detection: store the exact view definition (or digest) and dependency count.	entrypoint, captured_at, viewdef_digest, viewdef_objects
audit.t0_dataset_hash_baseline	Semantic drift detection: store daily dataset hashes computed from sampled outputs.	entrypoint, day, captured_at, dataset_sha256, sample_limit

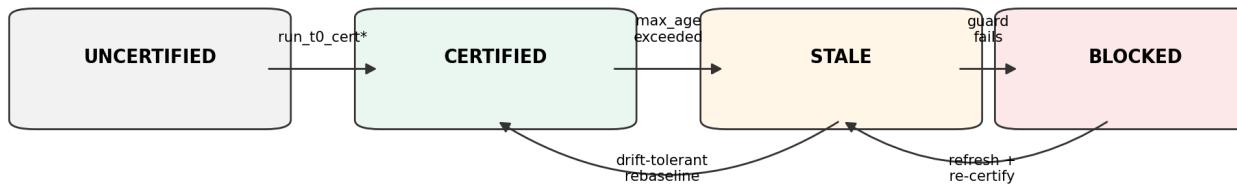
Table 2.1 - Governance kernel tables for certification and drift baselining.

Static guardrails: view-closure analysis

The first layer of certification is a static scan over the entrypoint's *view closure* - the set of dependent views and functions referenced by the entrypoint. The scan enforces a denial-list of forbidden time-of-query tokens (e.g., NOW(), CURRENT_DATE) and a deny-list of forbidden dependencies (e.g., post-t0 labels). Because the scan is performed over the closure, it detects leakage introduced indirectly through nested views.

Dynamic guardrails: invariants and future-information tests

Static analysis alone cannot catch all temporal failure modes. The certification program therefore includes dynamic invariants that are evaluated as SQL queries. Examples include: (i) absence of post-t0 labels in feature tables, (ii) uniqueness of keys per (generation, finn_id, t0), (iii) monotonicity constraints for as-of mappings, and (iv) targeted perturbation tests that intentionally shift t0 to validate that 'future' information does not leak through aggregates.



Enforcement points:

- query-time cert_guard views (CROSS JOIN audit.require_certified_strict)
- training preflight checks (max_age_hours + contract hashes)
- access control: deny base tables; allow only certified entrypoints

Figure 2.2 - Certification lifecycle and enforcement points.

2.6 Enforcement model: fail-closed access at query time and at train time

A governance system is only as strong as its enforcement boundary. The architecture uses two complementary enforcement mechanisms: **query-time blocking** (through cert_guard views) and **train-time preflight** (through explicit checks in the training program). Either mechanism is sufficient to prevent accidental consumption; together they provide defense in depth.

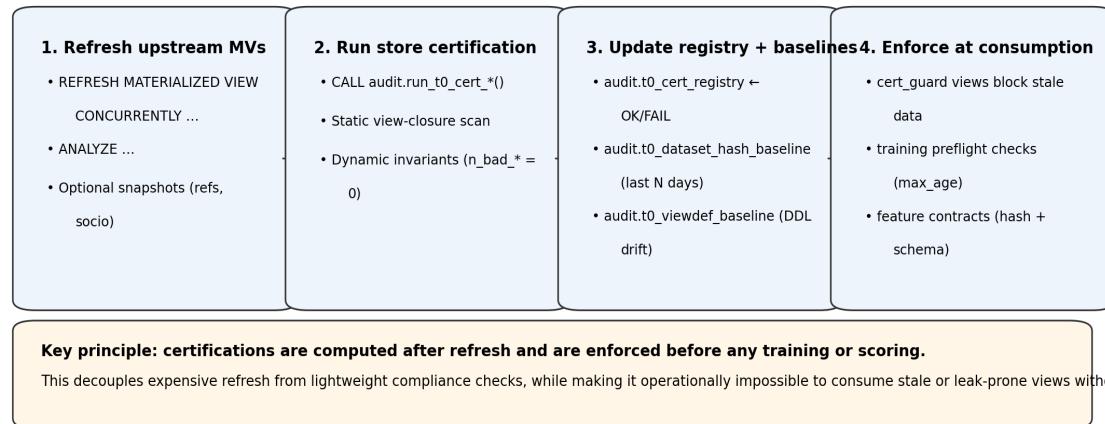


Figure 2.3 - Certification pipeline from refresh to enforced consumption.

Query-time enforcement via cert_guard views

The most reliable place to enforce certification is at the SQL boundary. A *cert_guard view* is a thin wrapper around an entrypoint that invokes a guard function as part of query execution. If the entrypoint is stale or uncertified, the guard raises an exception and the query fails. Because the guard executes in the database, downstream consumers cannot 'forget' to check certification.

```
-- Pattern: certified entrypoint wrapped by a guard
CREATE OR REPLACE VIEW ml.socio_market_feature_store_t0_cert_guard_v AS
SELECT s.*
FROM   ml.socio_market_feature_store_t0_v1_v s
CROSS JOIN LATERAL audit.require_certified_strict(
    'ml.socio_market_feature_store_t0_v1_v',
    max_age_hours => 24
);
```

Listing 2.2 - Query-time enforcement via cert_guard wrapper views.

Guarded training views: enforce once, read fast MVs

Some stores expose large, performance-critical materialized views (MVs) that should be read directly during training (for example, trainer-derived features merged in chunks). The governance layer supports a pattern that enforces certification exactly once per query using a MATERIALIZED guard CTE, while still selecting from the MV in the hot path.

```
CREATE OR REPLACE VIEW ml.trainer_derived_features_train_v AS
WITH guard AS MATERIALIZED (
    SELECT audit.require_certified_strict('ml.trainer_derived_feature_store_t0_v1_v') AS ok
)
SELECT mv.*
FROM   ml.trainer_derived_features_v1_mv mv
CROSS JOIN guard;
```

Listing 2.3 - Guard-once training view: MATERIALIZED certification check + MV hot path.

Train-time enforcement via preflight checks

Train-time enforcement is implemented in the training program as a preflight stage. The program explicitly calls `audit.require_certified_strict` for each required entrypoint and asserts feature contracts for external blocks. This produces human-readable logs ('[cert] OK ... max_age=24 hours') and fails fast before any heavy training compute.

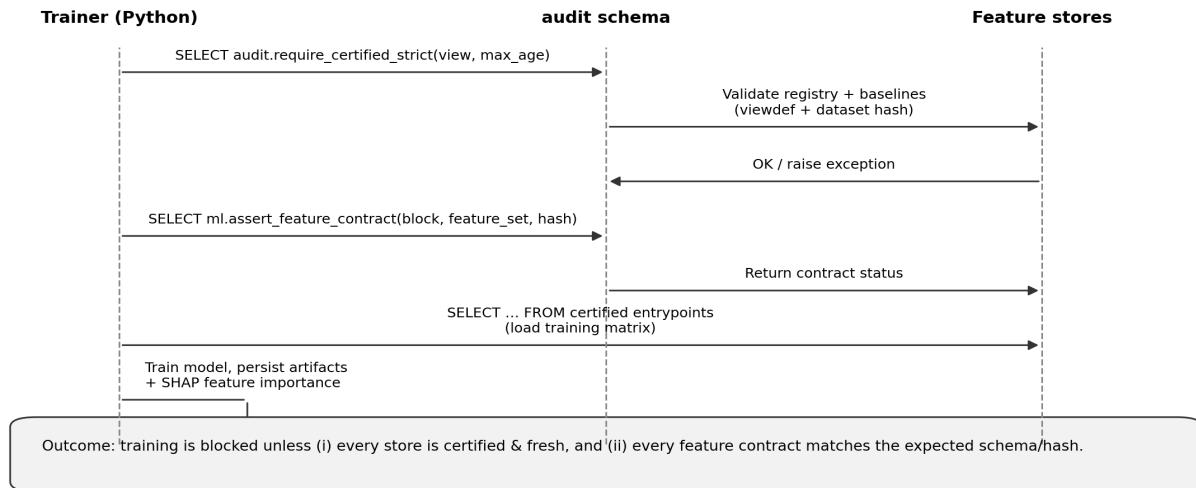


Figure 2.4 - Training preflight sequence: certification and contract checks before data load.

2.7 Drift-tolerant certification: controlled evolution without silent regressions

Real feature stores evolve. A strict 'bit-for-bit identical forever' rule is operationally infeasible for some blocks, especially those derived from upstream scrapers, external ML signals, or long-tail categorical statistics. The governance layer therefore distinguishes **strict** certification from **drift-tolerant** certification.

Dataset hash baselines

Drift is measured by hashing samples of the entrypoint output for a sliding window of days. A canonical serialization ensures stability across execution plans and minor formatting differences. The baseline table stores the expected hash per day.

$$h(d) = \text{SHA256}(\text{canonJSON}(\text{sample}(V_{t_0-d})))$$

Equation 2.3 - Dataset hash baseline for drift detection (per-day hashed samples).

When an entrypoint is certified, the system writes the baseline hashes for the last N days. On subsequent certifications, newly computed hashes are compared to the baseline. Strict certification requires equality; drift-tolerant certification allows deviation within an explicitly declared policy (for example, allowing drift in rare categorical histograms while still enforcing key invariants and DDL stability).

Rebaselining as a privileged operation

Drift tolerance is not 'ignore drift.' It is 'measure drift and permit rebaselining only through an explicit, auditable procedure.' Rebaselining updates the baseline hashes for a declared date window and records the operation timestamp. This makes drift a managed change, not an accidental one.

```
SELECT audit.rebaseline_last_n_days('ml.device_meta_store_t0_v1_v', 10);
```

Listing 2.4 - Rebaselining drift baselines over a bounded lookback window.

Case study: drift-allowed certification for device meta statistics

Certain device-meta aggregates (e.g., rarity scores, vote shares, and high-cardinality category entropy) are expected to drift as the market composition shifts. The system can certify such a store by enforcing the invariants that matter for leakage safety (no future labels, key uniqueness, stable view definition) while allowing controlled drift in the distributional statistics.

```
-- Excerpt: drift-allowed certification pattern (simplified)
CREATE OR REPLACE PROCEDURE audit.certify_device_meta_store_drift_allowed_v1(
    p_max_age_hours int DEFAULT 24, p_days int DEFAULT 10, p_sample_limit int DEFAULT 50000,
    p_max_bad_days int DEFAULT 2, p_notes text DEFAULT NULL)
LANGUAGE plpgsql AS $$

BEGIN
    -- 1) Rebaseline last N days
    CALL audit.rebaseline_last_n_days('ml.device_meta_store_t0_v1_v', p_days, p_sample_limit);
    -- 2) Enforce viewdef stability (digest compare)
    -- 3) Run invariants + drift diagnostics
    -- 4) Upsert audit.t0_cert_registry as CERTIFIED
END; $$;
```

Listing 2.5 - Drift-allowed certification skeleton for a statistically evolving store.

2.8 Access control: least privilege as a leakage prevention mechanism

Temporal certification is undermined if downstream users can bypass it by querying raw tables. The governance layer therefore adopts **least privilege** as a leakage control: training and scoring roles are granted SELECT on certified entrypoints (and their cert_guard wrappers) but are denied access to raw event tables and intermediate views.

This enforcement pattern changes the system's security posture. Instead of trusting every consumer to 'do the right thing,' the database becomes a policy engine: the only exposed interfaces are those that have passed certification and are tracked in the registry.

Certified entrypoints as security boundaries

- Raw tables (e.g., listings event streams) remain accessible only to ETL and feature-engineering roles.
- Intermediate materialized views remain internal implementation details and may change without consumer breakage.
- Entrypoint views are treated as stable APIs; their schemas are governed by feature contracts.
- cert_guard wrappers enforce freshness at query time and can be granted broadly without exposing internals.

The combined effect is a strong separation of concerns: feature engineering can evolve rapidly, while model consumers observe a stable, certified interface.

2.9 Operations: refresh, certify, confirm

A governance layer must be operable by humans under time pressure. The system therefore provides a runbook that makes the refresh -> certify -> confirm sequence explicit and repeatable. The core operational discipline is: **refresh pipelines do not implicitly certify**. Certification is a separate, logged step.

Standard daily workflow

- **Refresh** upstream MVs and reference snapshots (concurrent where possible).
- **Certify** each store by calling its *run_t0_cert_** procedure (or a consolidated 'certify all' procedure).
- **Confirm** registry status and max-age compliance before starting training jobs.
- **Train** (preflight checks re-validate certifications at runtime).

Store certification entrypoints

Feature block	Certified endpoint view	Certification procedure (illustrative)
Socio / market	ml.socio_market_feature_store_t0_v1_v	audit.run_t0_cert_socio_market_store_v1
Geo	ml.geo_feature_store_t0_v1_v	audit.run_t0_cert_geo_store_v1
Vision	ml.vision_feature_store_t0_v1_v	audit.run_t0_cert_vision_store_v1
Fusion	ml.fusion_feature_store_t0_v1_v	audit.run_t0_cert_fusion_store_v1
Device meta	ml.device_meta_store_t0_v1_v	audit.run_t0_cert_device_meta_store_v1
Trainer-derived	ml.trainer_derived_feature_store_t0_v1_v	audit.run_t0_cert_trainer_derived_store_v1
WOE anchor	ml.woe_anchor_feature_store_t0_v1_v	audit.run_t0_cert_woe_anchor_store_v1

Table 2.2 - Examples of certified entrypoints and corresponding certification procedures.

Session safety defaults

Operational governance includes 'fail fast' database session settings that prevent multi-minute stalls and reduce the risk of training jobs blocking each other. These settings are typically applied at connection initialization.

```
SET lock_timeout = '5s';
SET statement_timeout = '20min';
SET jit = off;
```

Listing 2.6 - Recommended Postgres session safety settings for training workloads.

Confirmation queries

```
-- Confirm that all required entrypoints are certified and fresh
SELECT entrypoint, status, certified_at, max_age_hours
FROM   audit.t0_cert_registry
WHERE  status <> 'CERTIFIED'
      OR certified_at < NOW() - (max_age_hours || ' hours')::interval
ORDER BY certified_at ASC;
```

Listing 2.7 - Typical confirmation query for stale or uncertified entrypoints.

2.10 Observability: feature block attribution and audit trails

Governance is incomplete without observability. The system therefore attaches metadata to every feature column indicating its originating *feature block* (e.g., socio-economic store, vision store, trainer-derived store). During evaluation, feature importance values (gain, SHAP mean, SHAP mean absolute) are persisted with this block metadata, enabling block-level accountability and regression detection.

Feature block tagging

The training pipeline includes a deterministic mapping from feature names to blocks. This mapping is used both in logging and when persisting feature importance records.

```
def infer_feature_block(feature_name: str) -> str:
    if feature_name.startswith(('ptv_', 'anchor_')):
        return 'anchor_priors_store'
    if feature_name.startswith(('dev_', 'gmc_')):
        return 'device_meta_store'
    if feature_name.startswith(('fusion_',)):
        return 'fusion_store'
    if feature_name.startswith(('image_', 'dmg_', 'caption_', 'stock_')):
        return 'vision_store'
    if feature_name.startswith(('allgen_', 'gen_', 'speed_', 'battery_', 'delta_')):
        return 'trainer_derived_store'
    return 'main_feature_store'
```

Listing 2.8 - Feature block inference used for audit and SHAP attribution.

Block-level SHAP sums act as a governance signal: they quantify how much predictive power is sourced from each store. This helps prioritize certification hardening and guides investment in store quality.

2.11 A formal guarantee: certified stores imply T0-safe training matrices

The governance layer's purpose is to transform a social process ('be careful about leakage') into a technical guarantee. While the full proof depends on the specifics of each store, the kernel supports a general theorem of the following form.

Theorem 2.1 (Fail-closed T0 safety under certification). Assume an entrypoint view V satisfies: (i) its view closure contains no forbidden time-of-query tokens and no forbidden dependencies; (ii) its certification procedure validates key invariants (uniqueness and absence of future labels) on a representative window; (iii) its view definition digest matches the recorded baseline; and (iv) its dataset hashes match the baseline under the declared drift policy. Then any consumer that accesses V exclusively through cert_guard enforcement (or through preflight checks that call `audit.require_certified_strict`) obtains feature vectors that are deterministic functions of information available at t_0 for each row key (generation, finn_id, t_0).

Proof sketch

Conditions (i) and (iii) rule out explicit and structural sources of time travel (e.g., `NOW()` usage and untracked DDL changes), ensuring that the transformation logic is stable and does not depend on query time. Condition (ii) excludes the most common semantic leakage channels: joins to post-t0 labels and duplicates that allow non-deterministic row selection. Condition (iv) controls latent semantic drift by requiring output agreement with recorded baselines (strict) or with an explicitly permitted drift policy (tolerant). Finally, cert_guard enforcement ensures the conditions are checked at the moment of consumption; the system therefore fails closed rather than relying on external discipline.

In practice, each store's certification procedure is a bundle of invariants tailored to its data semantics. The governance kernel standardizes their outputs (registry status and baselines) so that training and scoring pipelines can treat all stores uniformly.

2.12 Limitations and future work

No governance system eliminates all risk. The architecture is strongest against accidental leakage and silent drift; it is not a substitute for careful statistical validation. Key limitations include:

- **Static analysis is conservative:** view-closure token scans can miss leakage introduced through opaque UDFs or external services unless those dependencies are explicitly modeled.
- **Hash baselines are sampled:** dataset hashes are computed on bounded samples for practicality; rare-event drift can evade detection without targeted invariants.
- **Reference dimensions require as-of discipline:** if a dimension table is updated in place without historical versioning, certification must rely on periodic snapshotting.
- **Complex temporal joins remain subtle:** window functions and late-arriving events demand store-specific invariants beyond the kernel.

Future work naturally divides into two tracks. First, richer formal tooling: automated lineage extraction and proofs that relate SQL semantics to the filtration model. Second, deeper operational automation: continuous certification, anomaly detection on drift metrics, and automated rollback to last-known-good certified versions.

Summary. This chapter introduced the governance layer that makes T0-safe, reproducible feature engineering operational. By combining certified entrypoints, contract hashes, drift baselines, and fail-closed enforcement, the system converts temporal correctness from a best-effort guideline into an enforceable interface for survival modeling and tail-risk prediction.