ChatGPT

Deep Research Report on Systems Similar to a Lightweight “Quality Sentinel” Validation Layer

Executive summary

A “Quality Sentinel” that runs per-module sanity checks on API/module outputs, flags anomalies with confidence notes for the UI, produces an admin audit feed, and supports cross-reference checks against benchmarks is basically a **hybrid of (a) data-quality unit testing, (b) observability/error grouping, and (c) contract/schema validation**—but relocated into the “hot path” of runtime responses. Great Expectations and Deequ are the closest conceptual matches on the *rule-driven* side (they treat checks like “unit tests for data” and produce human-readable reports), while Monte Carlo, Datadog, and Sentry are closest on the *operational* side (anomalies/issues, grouping, and alerting workflows). [1]

The core design tension is simple and non-negotiable: **every millisecond you add to the request path is a tax you pay forever**. So the “lightweight” part isn’t marketing—it has to be an architectural constraint. This is why mature systems split checks into (1) *fast inline guards* (schema, bounds, invariants), and (2) *asynchronous statistical/anomaly checks* (baseline drift, cross-reference comparisons, aggregation) that can lag by seconds/minutes without tanking latency or availability. Monte Carlo explicitly leans on learned historical baselines for anomaly detection, and tools like Evidently compare “current” vs “reference” datasets to detect distribution shift—useful patterns, but usually better off the hot path unless you’re doing tiny samples. [2]

A practical “minimal viable sentinel” that still feels valuable typically includes: (1) a middleware/post-query hook framework, (2) a small rule DSL in JSON/TypeScript with severity + confidence, (3) a persistent audit/event stream, (4) UI surface area for *non-blocking warnings*, and (5) alert routing (with grouping and rate limits). The closest off-the-shelf UX metaphors for audit/grouping are Sentry’s “issues” (events grouped by fingerprint) and Datadog’s “issues” in Error Tracking (aggregated error data + monitors for regressions/new issues). [3]

Unspecified details that materially affect design: where “modules” live (monolith vs microservices), data volumes, whether responses are deterministic, the definition of “benchmark” (golden dataset vs live secondary source), and your tolerance for fail-open vs fail-closed behavior. The recommended architecture below calls these out explicitly.

Comparable tools and how they map to sentinel features

What “similar” really means in practice

Most named tools in this space were born in one of two worlds:

1) **Data quality / data observability (batch-leaning)**
They assume you can run checks "around" a dataset/table/pipeline, generate reports, and alert when metrics drift. Great Expectations formalizes expectation suites and renders validation results into "Data Docs." [4]
Deequ positions itself explicitly as "unit tests for data" at scale on Spark, with research describing efficient constraint verification and incremental computation. [5]
Soda uses a YAML DSL (SodaCL) to define checks, and Soda Core turns those checks into SQL queries over your data sources. [6]
Evidence note: as of **January 27, 2026**, Soda announced a license change of Soda Core from Apache 2.0 to **Elastic License 2.0 (ELv2)**—still free to use internally, but with restrictions (notably around offering it as a managed service). [7]

2) **Observability / error tracking (runtime-leaning)**
These tools live in production traffic: they group events, show timelines, drive alerts, and support workflows. Datadog Error Tracking groups large volumes of errors into an "issue" abstraction and supports monitors on new/regressing/high-impact issues. [8]
Sentry similarly treats an "issue" as grouped events based on a fingerprint (grouping algorithm), with breadcrumbs providing a pre-error event trail. [9]

A Quality Sentinel is basically trying to steal the **best ideas from both**: rule-based validation *and* production-grade workflows.

Visual grounding

Feature-to-tool mapping

The table below maps the requested Sentinel features (left) to concrete, known capabilities in common tools —where there's a strong match, a partial match, or a conceptual pattern you can adapt.

Sentinel capability	Great Expectations	Deequ	Soda	Evidently	Monte Carlo	Datadog	Sentr
Rule-driven per-asset/ module checks	Expectation Suites + Checkpoints validate data and produce results. [10]	Declarative checks as "unit tests for data." [5]	SodaCL YAML checks executed via scans. [6]	Test suites/ metrics for data & ML evals. [11]	Monitors (often baseline-driven) across tables/ pipelines. [12]	Monitors/ metrics for services/ errors/ latency. [13]	Issue event valida SDK instru patte

Sentinel capability	Great Expectations	Deequ	Soda	Evidently	Monte Carlo	Datadog	Sentr
“Anomaly” detection vs hard thresholds	Mostly explicit expectations (threshold-ish), though can be extended. [15]	Has anomaly detection components and examples showing anomaly checks on metrics. [16]	Automated monitoring checks (row count anomalies, schema changes) exist but not in Soda Core OSS per docs. [17]	Drift detection using statistical tests; compares reference vs current. [18]	“Learns normal patterns” and alerts when violated; minimal manual thresholds emphasized. [19]	Anomaly monitors exist for metrics; Error Tracking monitors for regressions. [20]	Autor issue group trend
Confidence / probability-style explanation	Not a first-class UX primitive; you can add metadata. [22]	Can frame anomaly checks statistically (e.g., normal dev thresholds). [23]	Results typically pass/ fail on checks; advanced features vary by tier. [24]	Provides drift detected/ not detected based on statistical tests. [18]	Baseline/ anomaly systems often imply confidence; implementation details proprietary. [19]	Alert conditions + anomaly detection; confidence mostly implicit in monitors. [25]	Group expla availa group [26]
UI-facing warnings (inline badges, etc.)	Data Docs = human-readable reporting; not in-app end-user UI. [27]	Library-level; UI is usually custom. [28]	Soda Cloud provides UI; Soda Core is CLI/library. [24]	Produces reports/ tests; cloud adds dashboards. [29]	Product UI for anomalies + lineage/impact. [30]	Dashboards, monitors, explorers. [31]	Issue bread timeli
Admin audit feed / event log	Validation results are saved and viewable in Data Docs. [27]	Typically stored via custom integrations/ metrics repositories. [33]	Cloud tiers emphasize audit logs/ RBAC. [34]	Cloud tiers include audit logs (per pricing tiers). [35]	Platform UI includes monitoring history; billing/ consumption views exist. [36]	Logs + monitors; “Audit Trail” monitor type exists. [37]	Issue repre aggre event group histor
Cross-reference against benchmarks	Expectation suites can encode “golden” constraints. [38]	Metrics repository + anomaly checks pattern fits “compare to history.” [33]	Checks can compare aggregates; advanced automation varies. [39]	Explicitly compares “reference” vs “current.” [18]	Historical baselines + impact/lineage help triage benchmark drift. [19]	Compare service metrics over time; SLO-style comparisons are typical. [40]	Trend comp releas envir segm comm

Sentinel capability	Great Expectations	Deequ	Soda	Evidently	Monte Carlo	Datadog	Sentr
Middleware / hook-oriented integration	Usually pipeline/ETL integration rather than runtime middleware. [42]	Spark library, integrated into jobs. [28]	CLI/library scans, integrated into orchestration/ CI. [43]	Python library + platform ingestion. [44]	Vendor integrations; not "your middleware." [45]	Agents/SDKs and monitors; integrates with runtime telemetry. [46]	SDKs captu in-app

Deployment model, language support, and cost model comparison

This table focuses on what matters when you're deciding whether to build vs adapt.

Tool	Primary deployment model	Typical integration point	Language focus	Cost model notes
Great Expectations	OSS framework + hosted GX Cloud. [48]	Batch validation (Checkpoints/Actions) + report rendering (Data Docs). [15]	Python. [49]	GX Core is Apache 2.0 and positioned as always free; cloud has tiers. [48]
Deequ	OSS library (Apache 2.0). [28]	Spark job: compute metrics, verify constraints, anomaly checks. [50]	Scala/Spark ecosystem (plus wrappers). [51]	Open-source library; operational cost is Spark compute. [52]
Soda	Soda Core + commercial tiers (Soda Cloud / paid plans). [24]	CLI/library scans that run checks against warehouses. [53]	Python library + YAML DSL (SodaCL). [6]	**License nuance:** Soda Core moved to ELv2 Jan 27, 2026 (free internal use, restrictions on managed service). [54]
Evidently	OSS library + Evidently Cloud tiers. [55]	Python evaluation + drift tests; platform adds dashboards/ alerting. [56]	Python. [57]	Public pricing tiers shown; enterprise adds audit logs/private cloud. [35]
Monte Carlo	Commercial SaaS (request pricing). [58]	Data platform integrations; baseline anomaly monitoring. [12]	Vendor product; not a library focus. [45]	Pricing is typically "request pricing"; consumption/billing views exist. [59]

Tool	Primary deployment model	Typical integration point	Language focus	Cost model notes
Datadog	Commercial SaaS. [60]	Agents/SDKs; monitors (APM/log/error tracking). [61]	Polyglot; broad agent ecosystem (varies by feature). [46]	Complex pricing by product/unit; official docs describe billing models. [62]
Sentry	SaaS + self-hosted option. [63]	SDK instrumentation; issues grouped by fingerprint; breadcrumbs. [64]	Polyglot SDKs; platform UI. [65]	Event-based pricing tiers; licensing goals emphasize self-host availability. [66]

Architectural patterns for middleware and post-query validation

Core placements

A Sentinel-like layer can sit in multiple places; the "best" placement depends on what you want to protect (UI correctness? downstream data integrity? SLA?), and what you can afford (latency/cost).

API gateway / edge proxy placement
At the edge, you can enforce coarse validation and attach metadata headers, but you're usually blind to deep module context unless you call out to a sidecar service. Systems like Envoy demonstrate a general pattern: an HTTP filter can call an external service and decide allow/deny, and it explicitly models fail-open vs fail-closed via configuration (e.g., `failure_mode_allow`). [67]
Kong demonstrates response-phase manipulation with plugins that transform upstream responses "on the fly before returning it to the client." [68]

Service middleware placement
This is the most natural match to "per-module" checks: your API handler calls module X, gets result R, then runs validator rules for module X, returning R plus warnings. In Node ecosystems, the middleware pattern is explicit and standardized (functions have access to request/response and `next()`), which makes insertion predictable. [69]
Framework-specific hooks can be even cleaner: NestJS interceptors run around controller execution and can be scoped globally/controller/method. [70]
GraphQL is similarly hookable: Apollo Server plugins expose lifecycle hooks and explicitly call out `didResolveOperation` as a place for "extra validation." [71]

Database triggers / query interceptors
If "module output" is derived from persisted state, validating at the DB layer prevents bad writes (strong), but it's often a blunt instrument for UI-level anomalies (and can kill write throughput if you overdo it). PostgreSQL documents triggers firing BEFORE/AFTER and at statement vs row granularity; row-level triggers can execute once per modified row. [72]
At the ORM/driver layer, hooks can validate post-query results: Prisma client extensions can "hook into the query life-cycle and modify an incoming query or its result." [73]

SQLAlchemy similarly exposes before/after cursor execute events, supporting instrumentation around raw statements. [74]

Event-driven consumers
If your system emits events that later drive UI or analytics, validating events in consumers can catch anomalies without impacting request latency. Kafka exposes a `ConsumerInterceptor` plugin interface to intercept (and possibly mutate) records received by consumers, with monitoring/logging as a stated use case. [75]
Schema registries formalize contracts at the message boundary: Confluent Schema Registry supports Avro, Protobuf, and JSON Schema, with compatibility/evolution rules. [76]

Recommended baseline architecture

Below is a "fast path + async path" architecture that matches your requirements while staying honest about latency.

```
flowchart LR
  U[Client UI] -->|request| API[API Layer]
  API --> MOD[Module/Service]
  MOD -->|result| SENT[Sentinel Hook\n(post-query/middleware)]
  SENT -->|validated result + warnings| API
  API -->|response w/ flags| U

  SENT -->|emit validation event| BUS[(Event/Queue)]
  SENT -->|write| STORE[(Audit Store)]
  BUS --> ASYNC[Async Validators\n(baselines, x-ref)]
  ASYNC --> STORE

  STORE --> ADMIN[Admin Audit Feed UI]
  STORE --> ALERT[Alert Router]
  ALERT --> ONCALL[Pager/Chat/Email]
```

This split mirrors how "learned baseline" systems operate: Monte Carlo stresses historical patterns for detecting abnormal behavior, while tools like Evidently compare reference vs current distributions—both are more stable when computed asynchronously or on sampled data rather than inline. [77]

Performance and latency considerations

If you put validation inline, the only sane rule is: **inline checks must be predictable and bounded**.

Practical guardrails that show up repeatedly in mature systems:

- **Precompile what you can**: JSON Schema validators like AJV compile schemas into efficient JavaScript validation code. The docs emphasize compilation to optimized functions for speed. [78]
- **Fail-open vs fail-closed must be explicit**: Envoy's external authorization filter documents a configuration that determines behavior when the external service is unavailable

(`failure_mode_allow`). Your Sentinel needs the same explicitness, because "validation service down" should not silently become "site down" unless you chose that. [67]
- **Avoid heavy statistical checks in the hot path**: drift detection libraries explicitly run statistical tests over distributions and compare datasets (reference/current). That's great, but it's compute and data hungry by design. [18]
- **Sampling is not a cop-out; it's how you survive**: Datadog's APM model emphasizes tracking service-level hits/errors/latency and supports analytics monitors for spikes in slow requests. This is essentially "validation via telemetry," computed as aggregated metrics, not by fully validating every payload synchronously. [40]

Failure modes you must design for

A Sentinel introduces failure modes you don't have today; pretending otherwise is how systems die in production.

Key categories and mitigation patterns:

1) **Validator failure** (bug, exception, outdated rule)
If the validator throws, you need defined behavior: skip validation, downgrade to "unknown," or block response. The "multiple plugins can run; only one error returns" dynamic in Apollo's lifecycle is a reminder that hook ordering and error aggregation are real complexities. [71]

2) **Rule/config skew** (UI expects different semantics than server rules)
The solution is versioned rule bundles and a visible audit trail. Great Expectations' habit of persisting validation results and rendering them in Data Docs is basically the "make it inspectable" pattern you want. [27]

3) **Backpressure / overload** (too many anomalies → too many logs/alerts)
Datadog Error Tracking and Sentry both solve a version of this by **grouping** events into issues, not treating every event as a unique snowflake. [79]

4) **Dependency outage** (audit store down, queue down)
Your runtime path must degrade safely. Again: this is why the event/audit writes should be best-effort with buffering, not mandatory for serving. The fail-open/closed knob in Envoy is the right mental model. [80]

Rule definition formats and example schemas in JSON and TypeScript

What to copy from mature ecosystems

A JSON/TypeScript rule system is basically your in-house version of:

- Great Expectations "Expectation Suites" (a named collection of assertions). [38]
- SodaCL checks (a declarative language for checks, YAML-based). [81]
- JSON Schema (formal structural validation vocabulary; modern drafts like 2020-12 are current). [82]

- OpenAPI 3.1 schema dialect alignment with JSON Schema (useful if your API is already OAS-described). [83]
- "Reference vs current" comparison from drift tooling (Evidently explicitly frames evaluation as comparing two datasets). [18]

Suggested sentinel rule model

Below is a **suggested** shape that supports:

- per-module checks (fast)
- cross-reference checks (benchmark/history)
- severity and confidence
- composability and versioning

TypeScript interfaces

```ts
// sentinel/rules/types.ts

export type Severity = "info" | "warning" | "error" | "critical";

export type Confidence =
  | { kind: "static"; value: number }                  // 0..1 chosen by author
  | { kind: "statistical"; pValue?: number; score: number } // derived
  | { kind: "model"; modelId: string; score: number }; // if you ever go ML

export interface RuleContext {
  moduleId: string;           // "billing.quote" / "catalog.search" etc.
  routeId?: string;           // optional: endpoint/operation id
  userTier?: string;          // optional: for selective enforcement
  env: "dev" | "staging" | "prod";
  requestId: string;
  timestampMs: number;
}

export interface ValidationFlag {
  flagId: string;
  moduleId: string;
  ruleId: string;
  severity: Severity;
  confidence: Confidence;
  shortMessage: string;       // UI-friendly
  details?: Record<string, unknown>; // admin/debug
  remediation?: string;       // what to do next
  tags?: string[];
}

export interface ValidationResult<T> {
```

```ts
  output: T;                  // original output (or patched output if allowed)
  flags: ValidationFlag[];
  validatorVersion: string;
  rulesetVersion: string;
  tookMs: number;
}

export type Selector =
  | { kind: "jsonpath"; expr: string }
  | { kind: "pointer"; expr: string }; // e.g., RFC6901-like pointer

export type Check =
  | { type: "schema"; schemaId: string; selector?: Selector }
  | { type: "required"; selector: Selector }
  | { type: "range"; selector: Selector; min?: number; max?: number }
  | { type: "enum"; selector: Selector; allowed: (string | number | boolean)[] }
  | { type: "regex"; selector: Selector; pattern: string }
  | { type: "custom"; fn: string; args?: Record<string, unknown> };

export interface Rule {
  id: string;
  moduleId: string;
  description: string;
  severity: Severity;
  confidence: Confidence;

  // Conditions: only run when these match
  when?: {
    env?: ("dev" | "staging" | "prod")[];
    routeId?: string[];
    featureFlag?: string;
  };

  checks: Check[];

  // Optional: cross-reference
  crossRef?: {
    kind: "benchmark" | "history" | "secondary_source";
    sourceId: string;
    selector: Selector;
    compare: "eq" | "approx" | "delta" | "ratio";
    tolerance?: number;
    window?: { minutes: number }; // if history-based
  };

  // Behavior knobs
  mode: "observe" | "enforce"; // enforce could block/patch (be careful)
}
```

JSON rule bundle format

```
{
  "rulesetVersion": "2026-02-15.1",
  "validatorVersion": "1.0.0",
  "schemas": {
    "catalog.search.response": {
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "type": "object",
      "required": ["items", "total"],
      "properties": {
        "total": { "type": "integer", "minimum": 0 },
        "items": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["id", "price"],
            "properties": {
              "id": { "type": "string", "minLength": 1 },
              "price": { "type": "number", "minimum": 0 }
            }
          }
        }
      }
    }
  },
  "rules": [
    {
      "id": "catalog.search.schema",
      "moduleId": "catalog.search",
      "description": "Response must conform to the catalog search schema.",
      "severity": "error",
      "confidence": { "kind": "static", "value": 0.95 },
      "mode": "observe",
      "checks": [{ "type": "schema", "schemaId": "catalog.search.response" }]
    },
    {
      "id": "catalog.search.total_matches_items",
      "moduleId": "catalog.search",
      "description": "total should be >= items.length",
      "severity": "warning",
      "confidence": { "kind": "static", "value": 0.7 },
      "mode": "observe",
      "checks": [
        { "type": "custom", "fn": "totalGteItemsLength", "args": { "pathTotal":
"/total", "pathItems": "/items" } }
      ]
```

```
    },
    {
      "id": "catalog.search.volume_drift",
      "moduleId": "catalog.search",
      "description": "Row-count-like drift: item count deviates from 7d
baseline.",
      "severity": "warning",
      "confidence": { "kind": "statistical", "score": 0.8 },
      "mode": "observe",
      "checks": [],
      "crossRef": {
        "kind": "history",
        "sourceId": "sentinel.metrics.catalog.search.items_length",
        "selector": { "kind": "pointer", "expr": "/items" },
        "compare": "delta",
        "tolerance": 0.5,
        "window": { "minutes": 10080 }
      }
    }
  ]
}
```

Why this model is aligned with “known good” patterns:

- Using JSON Schema draft 2020-12 is consistent with current JSON Schema guidance. [84]
- AJV-style compilation is a realistic approach to making schema validation fast in Node/TS runtimes. [85]
- “Reference/current” comparisons mirror how drift detection tooling frames the problem. [18]

Confidence scoring approach

A Sentinel that only says “pass/fail” is fine for CI, but mediocre for UI. If you want inline warnings that users won’t ignore, you want something like:

- **Severity** = user impact if wrong
- **Confidence** = how sure you are the output is wrong or suspicious

This is basically the same philosophy as anomaly tooling: drift is detected vs not detected based on statistical tests, and learned-baseline tools alert on pattern violations rather than single thresholds. [86]

A pragmatic confidence rubric (recommendation, not a standard):

- 0.9–1.0: structural violations (schema mismatch, required field missing)
- 0.7–0.9: invariant violations (total < items.length, negative price)
- 0.4–0.7: baseline deviations (counts/ratios drifting)
- 0.2–0.4: heuristics (“looks weird”: empty results unusually often)

UI and UX patterns for inline warnings and admin audit feeds

Inline warnings that don't destroy usability

For end-user UI, the goal is: **communicate uncertainty without blocking workflows** (unless you have high confidence and high severity).

Common patterns that are actually supported by established design systems:

- **Badges** for status/notifications on icons or navigation items (small, persistent). [87]
- **Status indicator palettes** (red/orange/yellow/green/blue) to convey severity levels consistently. [88]
- **Error message design** should be visible, constructive, and respect user effort. [89]

A concrete UI pattern for Sentinel warnings:

- Show a small badge (e.g., "⚠") next to a module output section
- Hover/expand reveals: short message, severity, confidence, timestamp
- Provide "Learn more" that opens an explanation panel with rule id + remediation

Here is a simple **wireframe mockup**:

```
[ Quote Summary ]                                      [⚠ 2]
------------------------------------------------------------
Total: $12,430
Line items:  8
Tax:         $0

⚠ Data quality warning (Confidence: 0.72)
• Tax is 0 but items include taxable products.
  View details ▸   Dismiss ▸
```

Admin audit feed and grouping model

Your admin feed should copy what Sentry/Datadog learned the hard way:

- **Don't show a raw firehose**; show grouped issues.
- Let engineers drill into raw events when needed.
- Make grouping explainable.

Sentry's model: "issues" are groups of similar events based on a fingerprint; grouping info is inspectable. [90]
Datadog's model: Error Tracking groups large volumes into a single issue abstraction, and monitors can page when an issue is new or regresses. [8]

A sentinel-specific audit feed could group by:

- (moduleId, ruleId, normalized field path)
- optionally include “environment” and “routeId”
- optionally include a “fingerprint” hash, borrowing Sentry terminology

Diagram: UI flow from validation → warning → audit

```
sequenceDiagram
  participant UI as UI
  participant API as API
  participant S as Sentinel
  participant A as Audit Store
  participant Admin as Admin Feed

  UI->>API: Request module output
  API->>S: Run post-query validation
  S-->>API: Return flags + confidence + severity
  API-->>UI: Response + inline warnings
  S->>A: Persist validation event
  Admin->>A: Query grouped issues + trends
  A-->>Admin: Issue list + drill-down
```

Visual examples of “badge + status + feed” metaphors

Alerting, escalation, retention, and systemic analytics

Alerting strategy that avoids “alert fatigue hell”

If you do this wrong, your Sentinel becomes a **factory for meaningless noise**. If you do it right, it becomes a radar.

The most robust pattern is multi-stage:

1) **Inline UI warning** (user-facing)
2) **Audit event** (admin-visible)
3) **Grouped issue** (engineering triage)
4) **Monitor + escalation** (on-call only when thresholds crossed)

Datadog models “monitors” across many categories and explicitly supports Error Tracking monitors for new/high-impact/regressing issues. [20]
Sentry similarly treats issues as the primary unit of triage rather than raw events. [21]

Retention and aggregation model

A Sentinel audit store should generally keep:

- **Raw validation events** (short TTL, e.g., 7–30 days)
- **Aggregated counters** by (moduleId, ruleId, day/hour, env) (longer retention)
- **Issue state** (open/closed, firstSeen/lastSeen, regression flag)

This approach is consistent with the "issue aggregation" framing in Datadog Error Tracking ("an issue is an aggregation of error data"). [91]

Systemic issue analytics

Useful analytics you can compute cheaply:

- "Top rules by fire rate" and "top modules by warning rate"
- warning rate correlated with deploys (release id)
- confidence distribution over time
- mean time to acknowledge / resolve (issue lifecycle metrics)

This is the same *shape* of operational insight that APM platforms target (hits/errors/latency measure and alert at service level). [40]

Security, privacy, and data governance considerations

Logging and audit data can become a liability

Your Sentinel will produce metadata about user-visible outputs. That's useful—and also a prime way to accidentally leak personal data or secrets into logs.

OWASP's Logging Cheat Sheet is explicit that security logging needs concentrated guidance and that logging mechanisms and collected event data must be protected from misuse; logs can contain personal and other sensitive information. [92]
General privacy guidance similarly emphasizes mitigation methods to protect users from privacy threats. [93]

Practical governance requirements (recommendations):

- **Data minimization by default**: log only fingerprints, counts, and rule ids; avoid raw payloads.
- **Redaction**: if you must log snippets for debugging, redact PII/credentials.
- **Access control**: admin audit feed must be role-gated; avoid exposing it broadly.
- **Retention limits**: don't keep sensitive event details forever.
- **Tamper resistance**: audit logs should be append-only or integrity-protected if they're used for compliance.

License and dependency governance (important nuance)

If you plan to reuse or embed third-party rule engines:

- Great Expectations Core is Apache 2.0 and explicitly positioned as remaining free/open-source. [94]
- Deequ is Apache 2.0. [28]
- Soda Core's ELv2 shift (Jan 27, 2026) is the big red flag: ELv2 permits free internal use but restricts offering the software as a hosted/managed service and other behaviors. If your Sentinel is part of a product you sell as a service, you must read this carefully. [95]

Recommended implementation plan and minimal viable feature set

Opinionated recommendation

If you want "lightweight middleware/post-query validation," **do not start by copying data observability SaaS patterns wholesale into the request path**. Those products are great, but they assume asynchronous monitoring, baselining, lineage graphs, etc. Your Sentinel should start as a **fast deterministic validator + event emitter**, then graduate into baselines and cross-references off the hot path.

This matches how Deequ frames scalable verification (translate checks into efficient aggregations; support incremental computation), and how baseline-driven tools emphasize learned patterns over manual thresholds. [96]

Minimal viable feature set

An MVP that is genuinely shippable:

- **Middleware/post-query hook** integrated into your server framework
- If Node/Express-style: middleware pattern is well-defined and consistent. [69]
- If GraphQL: Apollo plugin hooks allow validation around operations. [71]
- If DB/ORM-centric modules: Prisma query extensions can hook into query lifecycle/results. [97]
- **Rule bundle loader**
- loads JSON ruleset (versioned)
- compiles JSON Schema validators (AJV-style) once and caches them. [85]
- **Rule evaluation engine**
- runs fast checks
- emits normalized flags (severity, confidence, message, rule id)
- **Audit event writer**

- persist to DB or log pipeline
- must be non-blocking / buffered

- **UI warning surface**

- badges + expandable details (non-blocking), aligned with badge/status patterns. [98]

- **Basic grouping + alerting**

- group by fingerprint
- alert only on regressions or sustained high rates (copy Datadog/Sentry issue-first mental model). [99]

Incremental phases

Phase 1: Deterministic inline validation
Schema + invariants + required fields + ranges. (Fast, bounded.)

Phase 2: Audit feed + grouping
Store events, compute fingerprints, issue list + drill-down (Sentry-like grouping). [100]

Phase 3: Baselines + cross-reference
Move baseline checks async. Use "reference vs current" (Evidently style) for drift-like checks and "learn normal patterns" (Monte Carlo style) for anomaly checks, but computed in consumers/batch. [86]

Phase 4: Policy-driven enforcement
Only after you've proven low false-positive rates. Borrow the "fail-open vs fail-closed is explicit" lesson (Envoy). [80]

Sample code snippets and pseudocode for hooks and rule evaluation

Express-style post-query middleware sketch

```
// sentinel/middleware.ts
import type { Request, Response, NextFunction } from "express";

export function sentinelMiddleware({ evaluateRules, emitEvent }) {
  return async function (req: Request, res: Response, next: NextFunction) {
    // Attach a helper so handlers can validate module outputs consistently
    res.locals.validateModuleOutput = async function
validateModuleOutput(moduleId: string, output: unknown) {
      const ctx = {
        moduleId,
        routeId: req.route?.path,
```

```
        env: process.env.NODE_ENV === "production" ? "prod" : "dev",
        requestId: req.header("x-request-id") ?? crypto.randomUUID(),
        timestampMs: Date.now(),
      };

      const result = await evaluateRules(ctx, output);

      // Emit best-effort audit event (never block response)
      void emitEvent({
        ...ctx,
        flags: result.flags,
        rulesetVersion: result.rulesetVersion,
        tookMs: result.tookMs,
      });

      return result; // { output, flags, ... }
    };

    return next();
  };
}
```

This is aligned with the standard Express model: middleware functions have access to request/response and must call `next()` to continue. [69]

Prisma post-query hook sketch

```
// sentinel/prisma-extension.ts
import { PrismaClient } from "@prisma/client";

export function withSentinel(prisma: PrismaClient, { evaluateRules,
emitEvent }) {
  return prisma.$extends({
    query: {
      $allModels: {
        async $allOperations({ model, operation, args, query }) {
          const output = await query(args); // run original query

          // Only validate selected operations or models
          const moduleId = `db.${model}.${operation}`;

          const ctx = {
            moduleId,
            env: process.env.NODE_ENV === "production" ? "prod" : "dev",
            requestId: "unknown", // pass through from request context in real
code
```

```
            timestampMs: Date.now(),
          };

          const result = await evaluateRules(ctx, output);
          void emitEvent({ ...ctx, flags: result.flags });

          return output; // keep mutations minimal; consider returning patched
output carefully
        },
      },
    },
  });
}
```

This follows Prisma's documented query lifecycle hook capability ("hook into the query life-cycle and modify an incoming query or its result"). [73]

Rule evaluation engine pseudocode

```
function evaluateRules(ctx, output):
  rules = rulesByModuleId[ctx.moduleId]
  flags = []

  for rule in rules:
    if !conditionsMatch(rule.when, ctx): continue

    for check in rule.checks:
      outcome = runCheck(check, output)
      if outcome.failed:
        flags.append(makeFlag(rule, outcome))

    if rule.crossRef exists:
      // Inline only if cheap; otherwise enqueue async comparison
      if isCheapCrossRef(rule.crossRef):
        outcome = compareAgainstBenchmark(rule.crossRef, output)
        if outcome.failed:
          flags.append(makeFlag(rule, outcome))
      else:
        enqueueCrossRefJob(ctx, rule, outputSummary(output))

  // Optional: dedupe flags by (ruleId, selector)
  return { output, flags, rulesetVersion, validatorVersion, tookMs }
```

Optional: schema validation compilation strategy

If using JSON Schema + AJV, compile once per ruleset version and reuse compiled functions. AJV explicitly describes converting schemas into efficient JS validation code and emphasizes performance via compilation. [78]

Prioritized sources and references

Primary/most authoritative sources (use these as your “spine”):

Great Expectations documentation (Expectation Suites, Checkpoints, Data Docs). [101]
Deequ research paper (VLDB 2018): *Automating Large-Scale Data Quality Verification* (details on declarative constraints and efficient execution). [52]
Deequ library (awslabs/deequ) and AWS blog introducing Deequ. [51]
Soda documentation (SodaCL and Soda Core architecture) plus the Soda Core license update (Jan 27, 2026). [102]
Evidently documentation (drift detection logic; open-source vs cloud). [103]
Monte Carlo documentation (architecture; table/pipeline observability; learned patterns) and pricing request flow. [104]
Datadog documentation (APM monitors, Error Tracking, pricing/billing docs). [105]
Sentry documentation (Issues, Issue Grouping/fingerprints, breadcrumbs). [9]
JSON Schema specification (2020-12) and validation vocabulary. [106]
OpenAPI 3.1 specification alignment with JSON Schema (if you want contract-like schemas for API outputs). [83]
Express middleware docs (if your runtime is Node/Express-like) and Apollo Server plugin hooks (if GraphQL). [107]
Envoy external authorization filter docs (excellent reference for “external hook” + fail-open/fail-closed thinking). [67]
OWASP Logging Cheat Sheet and privacy guidance (for audit-feed safety). [108]

Secondary but useful extensions (good for implementation details and adjacent patterns):

Kafka ConsumerInterceptor API docs (event-driven validation hooks). [109]
Confluent Schema Registry fundamentals (multi-format schemas and compatibility). [110]
Material Design badge guidelines and NN/g error-message guidance (UI warning patterns). [111]

---

[1] [15] [42] Data Validation workflow
https://docs.greatexpectations.io/docs/0.18/oss/guides/validation/validate_data_overview?utm_source=chatgpt.com

[2] [12] Table - Monte Carlo
https://docs.getmontecarlo.com/docs/pipeline-observability?utm_source=chatgpt.com

[3] [9] [14] [21] [64] [90] Issues
https://docs.sentry.io/product/issues/?utm_source=chatgpt.com

[4] [10] [38] [101] Expectation Suite
https://docs.greatexpectations.io/docs/0.18/reference/learn/terms/expectation_suite/?utm_source=chatgpt.com

5 28 51 awslabs/deequ
https://github.com/awslabs/deequ?utm_source=chatgpt.com

6 81 102 Write SodaCL checks | Soda v3
https://docs.soda.io/soda-v3/soda-cl-overview?utm_source=chatgpt.com

7 Soda Core License Update: Moving to Elastic License 2.0
https://soda.io/blog/soda-core-license-update-moving-to-elastic-license?utm_source=chatgpt.com

8 79 91 Error Tracking for Logs
https://docs.datadoghq.com/logs/error_tracking/?utm_source=chatgpt.com

11 44 Evidently AI - Documentation: What is Evidently?
https://docs.evidentlyai.com/?utm_source=chatgpt.com

13 40 61 105 APM Monitor
https://docs.datadoghq.com/monitors/types/apm/?utm_source=chatgpt.com

16 23 33 Large-scale Data Quality Verification How to Unit-test Your ...
https://2019.berlinbuzzwords.de/sites/2019.berlinbuzzwords.de/files/media/documents/buzzwords2019_deequ.pdf?utm_source=chatgpt.com

17 Add automated monitoring checks
https://docs.soda.io/soda-cl-overview/automated-monitoring?utm_source=chatgpt.com

18 56 86 103 Data drift
https://docs.evidentlyai.com/metrics/explainer_drift?utm_source=chatgpt.com

19 45 77 104 Monte Carlo at a Glance
https://docs.getmontecarlo.com/docs/architecture?utm_source=chatgpt.com

20 25 31 37 Monitor Types
https://docs.datadoghq.com/monitors/types/?utm_source=chatgpt.com

22 27 Data Docs | Great Expectations
https://docs.greatexpectations.io/docs/0.18/reference/learn/terms/data_docs?utm_source=chatgpt.com

24 39 53 Soda Core | Soda v3
https://docs.soda.io/soda-v3/overview-main?utm_source=chatgpt.com

26 100 Issue Grouping
https://docs.sentry.io/concepts/data-management/event-grouping/?utm_source=chatgpt.com

29 Reports and Tests Overview | Evidently Documentation
https://docs-old.evidentlyai.com/user-guide/tests-and-reports/introduction?utm_source=chatgpt.com

30 Data Quality
https://www.montecarlodata.com/platform/data-quality/?utm_source=chatgpt.com

32 Using Breadcrumbs
https://docs.sentry.io/product/issues/issue-details/breadcrumbs/?utm_source=chatgpt.com

34 Soda Data Quality | Transparent Pricing
https://soda.io/pricing?utm_source=chatgpt.com

35 Evidently AI - Pricing
https://www.evidentlyai.com/pricing?utm_source=chatgpt.com

36 Billing - Monte Carlo
https://docs.getmontecarlo.com/docs/billing?utm_source=chatgpt.com

41 Performance Monitoring
https://docs.sentry.io/product/sentry-basics/performance-monitoring/?utm_source=chatgpt.com

43 Test data quality during CI/CD development | Soda v3
https://docs.soda.io/soda-v3/use-case-guides/quick-start-dev?utm_source=chatgpt.com

46 Application Performance Monitoring (APM)
https://www.datadoghq.com/product/apm/?utm_source=chatgpt.com

47 65 Sentry Docs | Application Performance Monitoring & Error ...
https://docs.sentry.io/?utm_source=chatgpt.com

48 94 GX Core: a powerful, flexible data quality solution
https://greatexpectations.io/gx-core/?utm_source=chatgpt.com

49 great-expectations
https://pypi.org/project/great-expectations/?utm_source=chatgpt.com

50 52 96 Automating Large-Scale Data Quality Verification
https://www.vldb.org/pvldb/vol11/p1781-schelter.pdf?utm_source=chatgpt.com

54 95 Soda Core License Update: Moving to Elastic License 2.0
https://soda.io/old/blog/soda-core-license-update-moving-to-elastic-license?utm_source=chatgpt.com

55 Open-source vs. Cloud
https://docs.evidentlyai.com/faq/oss_vs_cloud?utm_source=chatgpt.com

57 evidentlyai/evidently
https://github.com/evidentlyai/evidently?utm_source=chatgpt.com

58 59 Pricing
https://www.montecarlodata.com/request-for-pricing/?utm_source=chatgpt.com

60 Datadog Pricing
https://www.datadoghq.com/pricing/?utm_source=chatgpt.com

62 Datadog Pricing Models
https://docs.datadoghq.com/account_management/billing/pricing/?utm_source=chatgpt.com

63 Self-Hosted Sentry
https://develop.sentry.dev/self-hosted/?utm_source=chatgpt.com

66 Plans and Pricing
https://sentry.io/pricing/?utm_source=chatgpt.com

67 80 External Authorization
https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/security/ext_authz_filter?utm_source=chatgpt.com

68 Response Transformer - Plugin - Kong Docs
https://developer.konghq.com/plugins/response-transformer/?utm_source=chatgpt.com

69 107 Using middleware
https://expressjs.com/en/guide/using-middleware.html?utm_source=chatgpt.com

70 Interceptors | NestJS - A progressive Node.js ...
https://docs.nestjs.com/interceptors?utm_source=chatgpt.com

71 Apollo Server Plugin Event Reference
https://www.apollographql.com/docs/apollo-server/integrations/plugins-event-reference?utm_source=chatgpt.com

72 Documentation: 18: CREATE TRIGGER
https://www.postgresql.org/docs/current/sql-createtrigger.html?utm_source=chatgpt.com

73 97 Create custom Prisma Client queries - Extensions
https://www.prisma.io/docs/orm/prisma-client/client-extensions/query?utm_source=chatgpt.com

74 Core Events
https://docs.sqlalchemy.org/en/latest/core/events.html?utm_source=chatgpt.com

75 109 ConsumerInterceptor (kafka 2.1.0 API)
https://kafka.apache.org/21/javadoc/org/apache/kafka/clients/consumer/ConsumerInterceptor.html?utm_source=chatgpt.com

76 Schema Registry for Confluent Platform
https://docs.confluent.io/platform/current/schema-registry/index.html?utm_source=chatgpt.com

78 85 Getting started | Ajv JSON schema validator
https://ajv.js.org/guide/getting-started.html?utm_source=chatgpt.com

82 84 106 JSON Schema - Specification [#section]
https://json-schema.org/specification?utm_source=chatgpt.com

83 OpenAPI Specification 3.1.0 Released
https://www.openapis.org/blog/2021/02/18/openapi-specification-3-1-released?utm_source=chatgpt.com

87 98 Badge – Material Design 3
https://m3.material.io/components/badges?utm_source=chatgpt.com

88 Status indicators
https://carbondesignsystem.com/patterns/status-indicator-pattern/?utm_source=chatgpt.com

89 Error-Message Guidelines
https://www.nngroup.com/articles/error-message-guidelines/?utm_source=chatgpt.com

92 108 Logging Cheat Sheet
https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html?utm_source=chatgpt.com

93 User Privacy Protection - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/User_Privacy_Protection_Cheat_Sheet.html?utm_source=chatgpt.com

99 Error Tracking Monitors
https://docs.datadoghq.com/error_tracking/monitors/?utm_source=chatgpt.com

110 Formats, Serializers, and Deserializers for Schema ...
https://docs.confluent.io/platform/current/schema-registry/fundamentals/serdes-develop/index.html?utm_source=chatgpt.com

111 Badge – Material Design 3
https://m3.material.io/components/badges/guidelines?utm_source=chatgpt.com