CHECKLIST

Spark Pipeline Anomaly Detection Checklist

A practical checklist for detecting anomalies in Apache Spark big data pipelines using volume, freshness, schema, content, runtime, and lineage signals.

Spark Pipeline Anomaly Detection Checklist

Use this checklist to evaluate whether a Spark pipeline deviation is operationally significant, validate the impact, and decide the right response.

---

1) Define what ?normal? looks like

- [ ] Identify the pipeline?s critical datasets, stages, and outputs.
- [ ] Document the expected schedule for each run.
- [ ] Record normal variation by hour, day, week, or partition.
- [ ] Define the business impact of each dataset if it is incomplete or wrong.
- [ ] Identify which deviations are acceptable and which are not.
- [ ] Note any planned source volatility, batch delays, or seasonal patterns.

2) Choose the signals to monitor

Monitor more than one signal at once.

- [ ] Volume: input and output row counts.

- [ ] Freshness: arrival time, missing partitions, delayed partitions, duplicate partitions.
- [ ] Schema: missing fields, extra fields, type changes, nesting changes.
- [ ] Content: null rates, value ranges, cardinality, distribution shifts.
- [ ] Runtime: stage duration, job duration, retries, shuffle read/write, spill, skew.
- [ ] Lineage: unexpected downstream changes when upstream sources are unchanged.

3) Set baseline and threshold rules

- [ ] Use hard rules for invariants: zero rows, schema contract violations, invalid types.
- [ ] Use rolling baselines or tolerance bands for variable datasets.
- [ ] Set thresholds by dataset and partition, not globally.
- [ ] Require sustained deviation for non-critical alerts.
- [ ] Define which signals should trigger immediate action versus investigation only.
- [ ] Review thresholds periodically as data volumes and schedules change.

4) Capture metrics at the right pipeline boundaries

Collect metrics where they are most meaningful.

- [ ] Before transformation: source volume, freshness, partition completeness.
- [ ] After joins and filters: row count, deduplication rate, key coverage.
- [ ] After schema-sensitive steps: schema snapshot and field-level checks.
- [ ] Before publish: output counts, null rates, distribution checks.
- [ ] At the Spark job level: stage runtime, shuffle behavior, retries, memory pressure.

5) Validate whether an anomaly is real

A deviation should be confirmed before action is taken.

- [ ] Compare the current run to the recent baseline.

- [ ] Check whether the deviation persists across partitions or reruns.
- [ ] Confirm the issue is not explained by a known business cycle.
- [ ] Check whether only one stage changed or multiple stages changed.
- [ ] Determine whether the anomaly is isolated to data, performance, or both.
- [ ] Verify whether the deviation is attributable to a source, transform, or publish step.

6) Investigate likely root causes

Use the signal pattern to narrow the cause.

- [ ] Volume drop: missing source data, late partition, filter regression, upstream outage.
- [ ] Volume spike: duplicate ingestion, join expansion, upstream reprocessing.
- [ ] Freshness issue: delayed producer, scheduler problem, partition misalignment.
- [ ] Schema drift: upstream deployment, contract break, serialization change.
- [ ] Null spike: missing join keys, source quality issue, parsing failure.
- [ ] Runtime slowdown: skew, spill, bad partitioning, resource contention, shuffle pressure.
- [ ] Lineage change: unexpected source change, dependency drift, hidden upstream update.

7) Decide the operational response

- [ ] Alert only if the deviation is meaningful and actionable.
- [ ] Quarantine output if data correctness is uncertain.
- [ ] Retry only if the issue appears transient and safe to repeat.
- [ ] Continue with warning if the deviation is acceptable but noteworthy.
- [ ] Escalate to the owning team when the anomaly affects downstream consumers.
- [ ] Record the decision and rationale for later review.

8) Make alerts useful to operators

Every alert should answer these questions:

- [ ] What changed?
- [ ] How much did it change?
- [ ] Where in the pipeline did it appear first?
- [ ] Is it new, repeated, or persistent?
- [ ] What baseline was used for comparison?
- [ ] What action is recommended?

9) Keep false positives under control

- [ ] Use dataset-specific baselines.
- [ ] Account for known batch schedules and late-arriving data.
- [ ] Exclude planned releases and maintenance windows from alerting if appropriate.
- [ ] Alert on combinations of signals, not one noisy metric alone.
- [ ] Tune thresholds after reviewing real incidents and false alarms.
- [ ] Reassess thresholds when data volume, seasonality, or pipeline design changes.

10) Verify production readiness before relying on the detector

- [ ] Metrics are collected consistently on every run.
- [ ] Baselines are stored in a reproducible location.
- [ ] The pipeline can attach anomalies to a specific stage.
- [ ] Alert routing is tested and reaches the right owner.
- [ ] Recovery actions are documented and practiced.
- [ ] The detector has been validated on historical incidents or test data.
- [ ] Operators can distinguish data drift from infrastructure drift.

Quick decision guide

Signal pattern	Likely interpretation	Suggested action
Row count drops sharply and freshness is late	Missing or delayed upstream data	Investigate source and partition arrival
Schema changes and nulls spike	Contract break or parsing issue	Quarantine or block publish until verified
Stage runtime increases with shuffle and spill	Skew or partitioning problem	Inspect execution metrics and data layout
Output count is stable but downstream quality drops	Hidden join or content issue	Check key coverage and value distribution
Anomaly repeats for multiple runs	Persistent operational issue	Escalate and open an incident

---

Minimal implementation checklist

- [ ] Baseline the last known good runs.
- [ ] Monitor counts, freshness, schema, nulls, and runtime together.
- [ ] Compare new runs against expected ranges.
- [ ] Flag sustained deviations, not just single outliers.
- [ ] Attach each anomaly to a pipeline stage.
- [ ] Preserve evidence for debugging and audit.
- [ ] Define whether the pipeline should alert, quarantine, retry, or continue.

Final reminder

A Spark anomaly detector is most useful when it helps answer one question quickly: is this change meaningful enough to act on before downstream consumers are affected?

If the answer is yes, the detector should make the signal obvious, the scope clear, and the next step easy.