

USAspending → IDAS *Investigative Module*: Technical Specification (v1.2.1)

Prepared by: Bee R. Davis
Senior Software Engineer
`bee.r.davis@nasa.gov`
NASA Johnson Space Center (JSC)
2101 NASA Parkway, Houston, TX 77058

October 8, 2026

Contents

1	Scope & Non-Goals	4
2	Inputs & Outputs	5
2.1	Inputs	6
2.2	Outputs	7
3	Execution & Orchestration (Local + Databricks)	8
3.1	Runners at a glance	8
3.2	Local execution (developer / CI)	9
3.3	Databricks orchestration	9
3.4	Mode selection and fallbacks	12
3.5	Operational notes	12
4	Locking & Concurrency Control	12
4.1	Goals and non-goals	12
4.2	Lock scope and naming	13
4.3	Lock file schema	13
4.4	Acquisition protocol	14
4.5	Heartbeat and renewal	15
4.6	Release and cleanup	15
4.7	Stale lock reclamation	15
4.8	Failure modes and exit codes	15
4.9	Configuration knobs	16

4.10	Operational guidance	16
4.11	Test matrix (examples)	16
5	Algorithms & Signals	16
5.1	Row parity checks	16
5.2	Natural key estimation (Silver de-duplication)	18
5.3	Schema fingerprinting and drift classification	19
5.4	Parquet health checks	20
5.5	Sampling policy for evidence	20
5.6	Memory management and persistence	21
5.7	Complexity and limits	21
5.8	Finding codes and severities	21
6	Schema Registry & Proposal Workflow	22
6.1	Objectives	22
6.2	Tables	22
6.3	SemVer rules and drift classification	24
6.4	Bootstrap logic	24
6.5	Proposal creation	25
6.6	Governance and approval	25
6.7	Permissions and audit	26
6.8	Examples	26
7	Evidence Packaging & Retention	27
7.1	Bundle layout	27
7.2	Sampling caps	27
7.3	Retention policy	27
7.4	Packaging	27
8	Security, Privacy, Compliance	28
8.1	Data handling	28
8.2	Access control	28
8.3	Auditability	28
8.4	Secrets & tokens	28
9	Cost & Sizing Guidance	28
9.1	Reference cluster	28
9.2	Typical costs	28

9.3 Throughput tips	29
10 Runbook & Troubleshooting	29
10.1 Common issues and fixes	29
10.2 Operational checklist	29

TL;DR

What it is. A read-only, forensic *triage* module for the USAspending → IDAS pipeline that turns raw extracts into *defensible, ticket-ready evidence* of high-risk patterns associated with waste, fraud, and abuse (WFA). It does not adjudicate fraud; it quickly surfaces the *most suspicious* cases with reproducible proofs for analysts and auditors.

How it finds risk. The module:

- **Hardens the data first** (so signals are trustworthy): Phase-1 CSV↔Parquet row parity; Bronze↔Silver parity; schema fingerprinting against a governed registry; Parquet health checks.
- **Then scans for WFA typologies** (configurable “screens”):
 - *Split-threshold patterns*: many awards just under a configured limit (e.g., micro-purchase/competition thresholds) within narrow time/vendor windows.
 - *Duplicate/near-duplicate awards*: repeated natural key or stable vendor/date/amount tuples across slices.
 - *Round-dollar anomalies*: unusual concentrations at exact multiples or endings (e.g., 10,000; 25,000).
 - *End-of-period spikes*: disproportionate obligations near fiscal period boundaries.
 - *Vendor identity inconsistencies*: one UEI used with multiple unrelated recipient names (or vice-versa).
 - *Geo/type mismatches*: recipient geography not aligned with award type or policy configuration.
- **Scores and explains** each finding (PASS/WARN/FAIL), attaches bounded samples, and records whether checks ran in **approx** or **exact** mode with any fallback reason.

What you get. A deterministic investigation ID (INV-YYYYMMDDThhmmssZ-<uuid4>) and a bundle under \$OUTPUT_ROOT/\$INV_ID/ containing `report.md`, `findings.jsonl`, samples (Parquet), schema/manifests snapshots, metrics, and structured logs. Outputs are reproducible, link back to the Databricks run, and are ready to attach to an IG or audit ticket.

Where it runs. As a Databricks Job (recommended), a notebook, or a CLI—same engine, same evidence. Concurrency is controlled by a lock with TTL and heartbeat; timestamps are UTC; the module never mutates Bronze/Silver/Gold.

What it is not. Not a fraud verdict system. It is a *fast, explainable filter* that elevates the highest-risk patterns with the exact rows and computations used to flag them, enabling human review and downstream casework.

1 Scope & Non-Goals

In Scope

- **Row-parity diagnostics** across ingestion stages:
 - API CSV ↔ Parquet (Phase 1 integrity).
 - Bronze ↔ Silver (Phase 3 normalization).

- **Schema assessment** for Silver:
 - Detect and classify *schema drift* vs. *schema evolution* using a governed Schema Registry (SemVer rules).
 - Fingerprint includes *name, type, nullability, and stable metadata* to capture impactful changes.
- **Natural-key quality** estimation and selection for Silver de-duplication:
 - Sample-then-refine estimator with configurable thresholds (**info/warn/error**) and epsilon guard.
 - Transparent recording of the method used (**estimate** vs. **exact**).
- **Parquet health checks**:
 - Identification of corrupt files and path-level reporting.
 - Tiny-file detection and summarization (operational hygiene).
- **Evidence packaging** for incidents and reviews:
 - Human-readable **report.md**, machine-readable **findings.jsonl**, and bounded row samples.
 - Ticket-ready outputs with clear PASS/WARN/FAIL outcomes and run URLs.
- **Operational guardrails**:
 - Read-only operation (no data mutation); all timestamps in UTC.
 - Concurrency locking with TTL and heartbeat; evidence retention policy.
 - Mode transparency (**approx** vs. **exact**) with auto-fallback and explicit reason.
- **Deployment modes**: Databricks Job, notebook, or CLI; all produce identical evidence bundles.

Out of Scope

- Automatic data repair, reformatting, or mutation of Bronze/Silver/Gold datasets.
- Cross-domain reconciliations outside USAspending awards (e.g., financial ledgers).
- Pipeline scheduling & orchestration logic (owned by Phase 2 Airflow and Phase 3 Databricks jobs).
- Gold model design or KPI semantics (the module is read-only; Gold may be referenced for context only).
- Cost enforcement and quota management (reported, not enforced).

2 Inputs & Outputs

This section specifies the module’s inputs (data dependencies, parameters, environment), pre-flight validation, and outputs (status codes, evidence bundle, logs/metrics). Config precedence is: **CLI flags** > **Notebook params** > **Environment** > **defaults**.

2.1 Inputs

Data dependencies (read-only)

- **Phase 1 artifacts (optional):**
 - RAW_BASE/parquet/... (Parquet derived from API CSV).
 - RAW_BASE/bronze/**/manifest.jsonl (one JSON object per line).
- **Bronze Delta table (Phase 3):** staging table (append-only, with lineage).
- **Silver Delta table (Phase 3):** normalized & deduped awards.
- **Schema Registry (Phase 4b):**
 - schema_registry_schemas: canonical Silver schema versions (SemVer).
 - schema_registry_proposals: pending evolutions & approvals.

Parameter surface (CLI/Notebook/Job)

Name	Type	Default	Notes
-catalog	string	usaspending	Unity Catalog name
-schema	string	default	UC schema (database)
-bronze-table	string	bronze_awards	Fully-qualified
-silver-table	string	silver_awards	Fully-qualified
-raw-base	uri	dbfs:/...	Phase 1 root
-output-root	uri	dbfs:/...	Evidence root
-mode	enum	approx	approx exact
-exact-max-rows	int	1000000	Exact fallback
-natkey-info-rate	float	0.01	INFO threshold
-natkey-warn-rate	float	0.05	WARN threshold
-natkey-error-rate	float	0.10	ERROR threshold
-natkey-eps	float	0.005	Sampling epsilon
-sample-cap-per-finding	int	500	Max rows/finding
-sample-cap-total	int	2000	Max rows total
-lock-ttl-mins	int	45	Lock TTL
-priority	enum	normal	normal expedited
-schema-registry-bootstrap	enum	auto	auto off

Environment variables.

Name	Example	Default	Purpose
RAW_BASE	dbfs:/...	(above)	Phase 1 root
OUTPUT_ROOT	dbfs:/...	(above)	Evidence root
TZ	UTC	UTC	All timestamps

2.2 Outputs

Exit status

- **0** — Success
- **2** — Acceptance failure
- **3** — Runtime error
- **4** — Pre-flight failure
- **5** — Lock conflict
- **6** — Invalid arguments

Evidence bundle

Investigation ID: `INV-YYYYMMDDThhmmssZ-uuid4hex`

Evidence bundle layout

```
INV-20250919T182715Z-5f1a8f2e/  
|-- report.md  
|-- findings.jsonl  
|-- config.json  
|-- metrics.json  
|-- logs.ndjson  
|-- samples/  
| +-- FND_ROW_PARITY_001.parquet  
+-- artifacts/  
    |-- manifests_snapshot.jsonl  
    |-- schema_bronze.json  
    +-- schema_silver.json
```

Files.

- `report.md`: Human-readable summary
- `findings.jsonl`: Machine-readable findings
- `config.json`: Resolved configuration
- `metrics.json`: Run-level counters
- `logs.ndjson`: Structured logs (UTC)

Finding object (JSONL).

findings.jsonl example

```
{  
  "investigation_id": "INV-20250919T182715Z-5f1a8f2e",  
  "timestamp_utc": "2025-09-19T18:27:18Z",  
  "code": "ROW_PARITY",  
  "severity": "FAIL",  
  "message": "Silver row count deviates",  
}
```

```

"details": {
  "bronze_rows": 31625,
  "silver_rows": 6325
}
}

```

Schema fingerprints. Fingerprints incorporate name, type, and nullability.

Schema fingerprint pseudocode

```

def schema_fingerprint(cols_types):
    norm = [{"name": n.lower(), "type": t.lower(),
            "nullable": nullable}
            for n, t, nullable in cols_types]
    norm = sorted(norm, key=lambda x: x["name"])
    blob = json.dumps(norm)
    return hashlib.sha256(blob.encode()).hexdigest()

```

Side effects

- Evidence bundle under \$OUTPUT_ROOT/\$INV_ID/
- Schema Registry updates (if enabled)
- **No writes** to Bronze/Silver/Gold tables

3 Execution & Orchestration (Local + Databricks)

This section describes how the Investigative Module is executed locally and on Databricks, how Jobs are wired, how authentication works, and where outputs and logs land. The design keeps the same core engine across all runners (CLI, Notebook, Job). Configuration precedence remains: **CLI flags** > **Notebook params** > **Environment** > **defaults**.

3.1 Runners at a glance

- **Local CLI** (developer laptop or CI): fastest feedback loop for unit/integration tests.
- **Databricks Notebook** (ad-hoc): parameterized analysis, quick iteration, visual inspection.
- **Databricks Job** (recommended): repeatable, auditable, produces evidence bundles in DBFS, and posts run URLs to the tracking ticket.

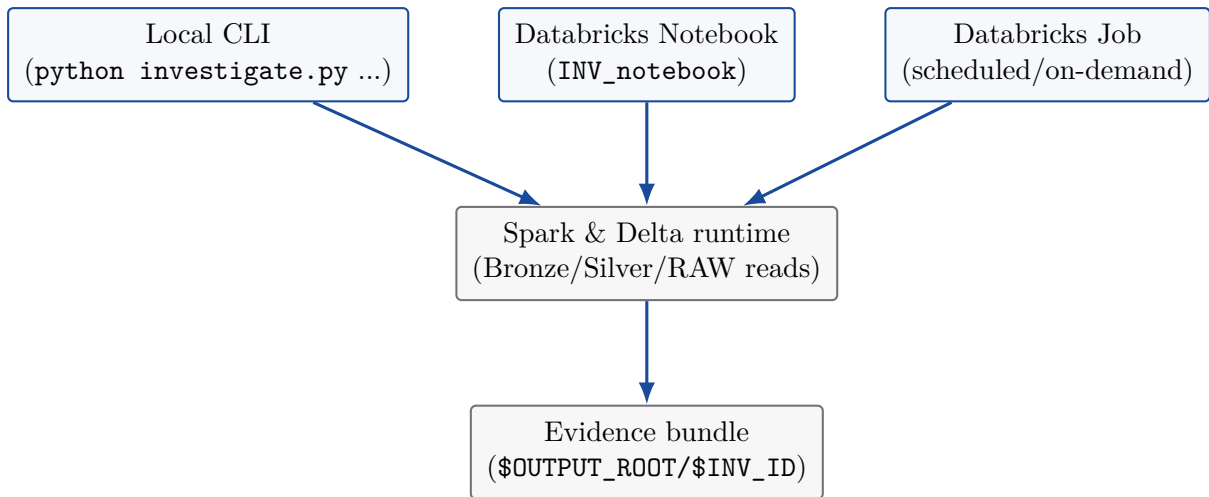


Figure 1: Runners share the same core engine; only orchestration differs.

3.2 Local execution (developer / CI)

Local runs are useful for fast iteration and for CI smoke tests against small fixtures.

Local CLI (illustrative)

```

# Create and activate a virtualenv (once per machine)
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -U pip wheel
python -m pip install -r requirements.txt

# Run with explicit parameters (override env/defaults)
python -m investigations.run \
  --catalog usaspending \
  --schema default \
  --bronze-table bronze_awards \
  --silver-table silver_awards \
  --raw-base "file:///path/to/raw" \
  --output-root "file:///tmp/investigations" \
  --mode approx --priority normal

```

Local outputs. When using `file://` URIs, the evidence bundle is written to the local filesystem, under `$OUTPUT_ROOT/$INV_ID/`. Structured logs stream to stdout and to `logs.ndjson` inside the bundle.

3.3 Databricks orchestration

Databricks is the recommended runtime for production investigations. The module is executed via:

1. **Workspace import:** notebooks and utils under a common folder (§3.3.2).
2. **Job creation:** JSON-based job definition that references the imported notebooks.
3. **Single-user cluster:** small, ephemeral, with auto-termination.

4. **Evidence write:** to DBFS \$OUTPUT_ROOT (e.g. dbfs:/mnt/idas/investigations).

3.3.1 Authentication (new CLI)

Use the new Databricks CLI with browser SSO. Create a named profile (e.g., `idas-dev`).

Databricks CLI (new) — one-time setup per machine

```
# Install the new CLI (already installed in many environments)
# databricks --version -> should print a 0.2xx+ version

databricks auth login \
  --host https://<your-workspace>.cloud.databricks.com \
  --profile idas-dev

# Verify identity
databricks current-user me --profile idas-dev
```

Profiles and scripts. All automation scripts accept `DATABRICKS_PROFILE` (or `-profile`) so that the same command line works across dev/stage/prod workspaces.

3.3.2 Workspace bootstrap & job creation

A helper script (e.g., `bootstrap_job.sh`) performs three idempotent actions:

1. **Copy Phase 1 artifacts (optional)** into `$RAW_BASE` on DBFS, if present locally.
2. **Import** the module's `notebooks/` and `utils/` into a configurable workspace folder (default: `/Shared/chihiro/databricks-compute/...`).
3. **Create or update** a Databricks Job from a JSON template, patching notebook paths to the chosen workspace base.

Bootstrap (excerpt)

```
WS_BASE="/Shared/chihiro"
RAW_BASE="dbfs:/mnt/idas/usaspending/raw"
PROFILE="idas-dev"

# Ensure DBFS directories exist and import notebooks/utils
databricks --profile "$PROFILE" fs mkdirs "$RAW_BASE"
databricks --profile "$PROFILE" workspace mkdirs "$WS_BASE/databricks-compute"
databricks --profile "$PROFILE" workspace import-dir --overwrite \
  ./databricks-compute/notebooks "$WS_BASE/databricks-compute/notebooks"
databricks --profile "$PROFILE" workspace import-dir --overwrite \
  ./databricks-compute/utils "$WS_BASE/databricks-compute/utils"

# Create/patch Job from JSON (paths rewritten to start at $WS_BASE)
databricks --profile "$PROFILE" jobs create --json-file patched_job.json
```

3.3.3 Cluster shape & runtime

A small, single-node, auto-terminating cluster is sufficient for most investigations. Key settings:

- **Profile:** Single-node (`spark.databricks.cluster.profile=singleNode`).
- **Workers:** `num_workers=0` (driver-only).
- **Auto-termination:** 20–30 minutes.
- **Runtime:** Current DBR with Scala 2.13 and Delta Lake preinstalled.
- **Security mode:** `SINGLE_USER` when evidence may include PII-like fields.

Cluster essentials (illustrative)

```
{
  "num_workers": 0,
  "spark_version": "17.2.x-scala2.13",
  "spark_conf": {
    "spark.master": "local[*,4]",
    "spark.databricks.cluster.profile": "singleNode"
  },
  "node_type_id": "i4i.4xlarge",
  "autotermination_minutes": 20,
  "data_security_mode": "SINGLE_USER",
  "custom_tags": {"ResourceClass": "SingleNode" }
}
```

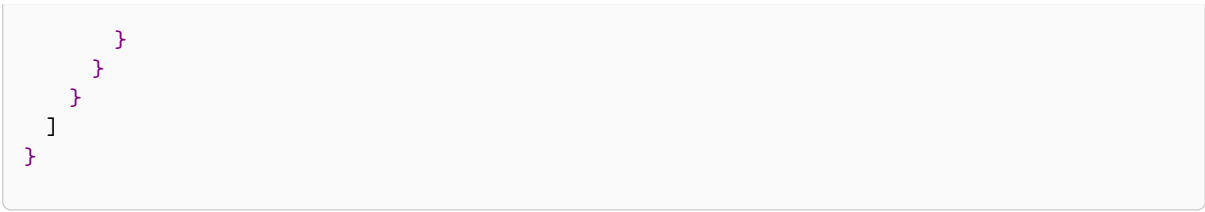
3.3.4 Job parameters & wiring

Jobs pass parameters to the notebook (or Python task) which forwards them to the engine. At minimum:

- `catalog`, `schema`, `bronze_table`, `silver_table`
- `raw_base`, `output_root`
- `mode`, `priority`
- `schema_registry_bootstrap`, `lock_ttl_mins`

Job task (illustrative)

```
{
  "name": "Investigate: Bronze<->Silver QC",
  "tasks": [
    {
      "task_key": "INV",
      "notebook_task": {
        "notebook_path": "/Shared/chihiro/databricks-compute/notebooks/investigate",
        "base_parameters": {
          "catalog": "usaspending",
          "schema": "default",
          "bronze_table": "bronze_awards",
          "silver_table": "silver_awards",
          "raw_base": "dbfs:/mnt/idas/usaspending/raw",
          "output_root": "dbfs:/mnt/idas/investigations",
          "mode": "approx",
          "priority": "normal",
          "schema_registry_bootstrap": "auto",
          "lock_ttl_mins": "45"
        }
      }
    }
  ]
}
```



3.3.5 Run monitoring & outputs

The Job returns a run URL (UI) and writes structured logs to the driver output and to `logs.ndjson` in the evidence bundle. The `report.md` includes a top-level PASS/WARN/FAIL summary and hyperlinks to sampled Parquet files.

Permissions. The Job's service principal (or single user) must have:

1. **Unity Catalog READ** on Bronze & Silver tables (or workspace/DBFS READ on paths if not using UC).
2. **WRITE** on `$OUTPUT_ROOT` and its `/locks` subfolder.
3. **READ** on `$RAW_BASE` if Phase 1 evidence is consulted.

3.4 Mode selection and fallbacks

Requested mode is recorded as `mode_requested`. The engine determines `mode_effective`:

- **Exact** is used only when both sides stay below `-exact-max-rows`; otherwise:
- **Approx** is used and `fallback_reason="exact_max_rows_exceeded"` is recorded.

Mode and fallback are echoed in `config.json` and each finding in `findings.jsonl`.

3.5 Operational notes

- **Idempotency:** repeated runs with the same inputs produce new `INV_IDs` and do not mutate data tables.
- **Locking:** a DBFS lock with heartbeat protects shared outputs; see §4 (next) for details.
- **Costs:** single-node ephemeral clusters keep runs inexpensive; see budgeting guidance in §9.

4 Locking & Concurrency Control

This section specifies how the Investigative Module prevents concurrent runs from writing to the same evidence output path, how stale locks are reclaimed, and what metadata is recorded for auditability. The design is file-based and works in Databricks (DBFS) and local file systems.

4.1 Goals and non-goals

- **Goals**

- Single-writer protection for evidence bundles under `$OUTPUT_ROOT`.
- Predictable behavior under contention: fast failure with clear messaging.
- Automatic cleanup: TTL expiry and safe reclamation of stale locks.
- Observability: lock owner, scope, TTL, last heartbeat, and run URL are recorded.
- **Non-goals**
 - Distributed mutual exclusion across multiple unrelated outputs.
 - Cross-pipeline locking (this module only protects its own outputs).
 - Breaking active locks; expedited priority does *not* preempt an active run.

4.2 Lock scope and naming

Locks are stored under a dedicated directory:

- `$LOCK_ROOT` defaults to `$OUTPUT_ROOT/locks`.
- One lock file per logical scope. Scope types:
 - **global**: guards the entire evidence root.
 - **table**: guards a specific table, e.g., Bronze or Silver.
 - **chunk**: guards a specific chunk (rare; used for targeted workflows).

Lock file paths (examples).

- Global: `$LOCK_ROOT/lock_global.json`
- Table: `$LOCK_ROOT/lock_table_usaspending_default_silver_awards.json`
- Chunk: `$LOCK_ROOT/lock_chunk_<hash8>.json`

4.3 Lock file schema

Each lock file is a single JSON document. Versioning allows non-breaking extensions.

Lock JSON (schema v1.0.0)

```
{
  "schema_version": "1.0.0",
  "lock_id": "LCK-20250919T182640Z-3f0c4c5a1b7d48bfa5f78122b0b0e9cd",
  "scope": {
    "type": "global", // global | table | chunk
    "table_fqn": "usaspending.default.silver_awards", // present when type=table
    "chunk_key": "FY2021_Q4_NASA_A" // present when type=chunk
  },
  "owner": {
    "runner": "job", // job | notebook | cli
    "user_name": "bee.r.davis@nasa.gov",
    "cluster_id": "0224-ABCD1234-INVEST",
    "run_id": "987654321",
    "run_url": "https://.../jobs/runs/run?run_id=987654321",
    "host": "driver",
    "pid": 12345
  }
}
```

```

},
"priority": "normal", // normal | expedited
"ttl_minutes": 45,
"heartbeat_period_sec": 45,
"created_at_utc": "2025-09-19T18:26:40Z",
"last_heartbeat_utc": "2025-09-19T18:27:25Z",
"expires_at_utc": "2025-09-19T19:11:40Z",
"notes": "Investigate Bronze<->Silver parity"
}

```

4.4 Acquisition protocol

Acquisition is best-effort atomic using create-if-absent semantics and rename-on-claim. On Databricks, use `dbutils.fs` or the Databricks CLI. On local files, use POSIX open with exclusive flags.

Acquire lock (pseudocode)

```

def acquire_lock(scope, ttl_minutes=45, hb_sec=45):
    path = lock_path_for(scope) # e.g., LOCK_ROOT/lock_global.json
    now = utcnow()

    # 1) Read if exists
    if exists(path):
        lock = read_json(path)
        if lock["expires_at_utc"] > now:
            # Active lock -> fail fast with details and exit code 5
            return Fail(active_lock=lock, code=5)
        else:
            # Stale lock detected; attempt safe reclamation
            if not try_reclaim(path, expected=lock):
                # Another contender won the race; re-check
                return Fail(contention=True, code=5)

    # 2) Compose new lock body
    new_lock = {
        "schema_version": "1.0.0",
        "lock_id": make_lock_id(), # LCK-<timestamp>-<uuid4hex>
        "scope": scope,
        "owner": owner_info(),
        "priority": priority(),
        "ttl_minutes": ttl_minutes,
        "heartbeat_period_sec": hb_sec,
        "created_at_utc": now,
        "last_heartbeat_utc": now,
        "expires_at_utc": now + ttl_minutes
    }

    # 3) Try to create atomically
    tmp = path + ".tmp." + uuid4hex()
    write_json(tmp, new_lock) # write temp
    if not atomic_mv(tmp, path): # best-effort atomic rename
        # Someone won the race; read current and return conflict
        cur = read_json(path)
        return Fail(conflict=cur, code=5)

    return Ok(lock=new_lock, path=path)

```

`try_reclaim(path, expected).`

- Confirm on-disk JSON still matches `expected.lock_id` and is expired.
- Rename existing to a backup (e.g., `path + ".stale.<ts>"`) to avoid clobber.
- If rename fails, the file likely changed; treat as contention and fail.

4.5 Heartbeat and renewal

The owner updates `last_heartbeat_utc` and extends `expires_at_utc` at a fixed cadence until release.

Heartbeat loop (pseudocode)

```
def heartbeat(path, lock_id, ttl_minutes, hb_sec):
    period = max(15, min(60, hb_sec)) # clamp to [15, 60] seconds
    while not shutdown_event.is_set():
        sleep(period + jitter(0, 5)) # small jitter prevents sync storms
        now = utcnow()
        lock = read_json(path)
        if lock["lock_id"] != lock_id:
            # Lost ownership; stop beating
            return
        lock["last_heartbeat_utc"] = now
        lock["expires_at_utc"] = now + ttl_minutes
        write_json(path + ".hb.tmp", lock)
        atomic_mv(path + ".hb.tmp", path)
```

4.6 Release and cleanup

On normal completion:

1. Stop the heartbeat loop.
2. Delete the lock file. If deletion fails, the TTL will expire the lock.

4.7 Stale lock reclamation

A lock is stale when `expires_at_utc < now`. Reclamation is guarded to avoid false positives:

- The module only reclaims if the on-disk `lock_id` matches what it read immediately before the attempt.
- If another contender updated the file during reclamation, the rename will fail, and the contender should re-check or abort with exit code 5.
- Expedited priority does not preempt an active lock; it only influences alerting and governance routing outside of this module.

4.8 Failure modes and exit codes

- **Code 5** (Lock conflict): existing non-expired lock; include current owner and expiry in the error.

- **Code 4** (Pre-flight): cannot create lock directory, or storage permissions error.
- **Code 3** (Runtime): unexpected IO or serialization error while beating or releasing.

4.9 Configuration knobs

- `-lock-ttl-mins` (default 45): time-to-live for the lock.
- `-priority` (normal|expedited): influences notifications, not lock semantics.
- `LOCK_ROOT`: custom location for lock files; defaults to `$OUTPUT_ROOT/locks`.

4.10 Operational guidance

- **Diagnostics:** to inspect a lock on Databricks, run `databricks fs head dbfs:/.../locks/lock_global.j`.
- **Manual recovery:** only reclaim manually if you confirm the owning run is terminated. Prefer rerunning the investigation rather than editing a live lock.
- **CI behavior:** concurrent CI jobs should use distinct `$OUTPUT_ROOT` or distinct lock scopes to avoid contention.

4.11 Test matrix (examples)

- **T1:** Two concurrent acquires on the same scope -> one succeeds, one fails with code 5.
- **T2:** Heartbeat refresh extends expiry; verify `expires_at_utc` is monotonic.
- **T3:** Simulate crash (no release); verify new run can reclaim after TTL.
- **T4:** Stale-lock race: two reclaimers; ensure at most one wins via rename check.
- **T5:** Permission error on lock directory -> code 4 and actionable message.

5 Algorithms & Signals

This section defines the algorithms that generate findings (PASS/WARN/FAIL). It covers: row-parity checks, natural key estimation, schema fingerprinting and drift classification, Parquet health checks, the sampling policy for evidence, memory guardrails, and limits.

5.1 Row parity checks

Row parity compares record counts (and optionally concrete set differences) across stages.

5.1.1 Phase 1: CSV -> Parquet

Signal: `ROW_PARITY_P1`

Inputs: CSV count from manifest vs. Parquet count from Phase 1 output.

Default threshold: 1 percent (configurable).

CSV->Parquet parity (approx by default, exact optional)

```
def row_parity_p1(csv_count:int, pq_count:int, threshold:float=0.01):
    # ratio is absolute relative difference
    diff = abs(csv_count - pq_count)
    base = max(1, csv_count) # guard divide-by-zero
    ratio = diff / base
    severity = "PASS" if ratio <= threshold else "FAIL"
    return {"ratio": ratio, "threshold": threshold, "severity": severity}
```

Exact mode (optional). For small datasets or targeted investigations, compute set difference on a stable row hash that is derived from normalized CSV rows vs. normalized Parquet rows. This requires aligning the same canonical stringification of rows (whitespace, case, nulls). Controlled by `-mode exact` and guarded by `-exact-max-rows`.

Exact set difference (guarded)

```
def exact_diff(left_df, right_df, max_rows:int=1_000_000):
    if left_df.count() > max_rows or right_df.count() > max_rows:
        return {"mode_effective": "approx", "fallback_reason": "exact_max_rows_exceeded"}
    # left minus right and right minus left (Spark: exceptAll)
    only_left = left_df.exceptAll(right_df)
    only_right = right_df.exceptAll(left_df)
    return {
        "mode_effective": "exact",
        "left_minus_right": only_left,
        "right_minus_left": only_right
    }
```

5.1.2 Phase 3: Bronze -> Silver

Signal: ROW_PARITY_P3

Inputs: Bronze count vs. Silver count for the same slice. Default threshold is 5 percent when using Bronze manifest expectations; can be tightened when using exact joins.

Bronze->Silver parity

```
def row_parity_p3(bronze_count:int, silver_count:int, threshold:float=0.05):
    diff = abs(bronze_count - silver_count)
    base = max(1, bronze_count)
    ratio = diff / base
    severity = "PASS" if ratio <= threshold else "FAIL"
    return {"ratio": ratio, "threshold": threshold, "severity": severity}
```

Approx vs. exact in Phase 3. Approx mode reports counts and ratio only. Exact mode joins on the selected natural key (Section 5.2) and computes symmetric differences, but is guarded by `-exact-max-rows`.

5.2 Natural key estimation (Silver de-duplication)

Signal: NATKEY_QUALITY

Goal: Choose a stable natural key column for de-duplication in Silver and report its health.

Candidate list (ordered example): generated_unique_award_id, unique_award_id, award_id, award_id_piid, piid, fain, uri, contract_award_unique_key.

Thresholds (configurable):

- info: `-natkey-info-rate` (default 0.01)
- warn: `-natkey-warn-rate` (default 0.05)
- error: `-natkey-error-rate` (default 0.10)

Estimator: sample-then-refine with epsilon guard

```
def estimate_natkey(df, candidates, eps=0.005,
                   info=0.01, warn=0.05, error=0.10,
                   sample_frac=0.1, sample_cap=1_000_000):
    best = None
    sample = df.sample(withReplacement=False, fraction=sample_frac) \
        .limit(sample_cap) \
        .cache()
    try:
        for name in candidates:
            if name not in df.columns:
                continue
            # Fast null presence check
            has_any_null = sample.filter(F.col(name).isNull()).limit(1).count() > 0
            # Estimate null rate from sample
            est_nulls = sample.filter(F.col(name).isNull()).count() if has_any_null else 0
            est_rate = est_nulls / max(1, sample.count())
            # If close to a threshold, compute exact null rate on full DF
            close_to_threshold = any(abs(est_rate - t) <= eps for t in [info, warn, error])
            if close_to_threshold:
                total = df.count()
                exact_nulls = df.filter(F.col(name).isNull()).count()
                rate = exact_nulls / max(1, total)
            else:
                rate = est_rate

            # Duplicate rate on sample (cheap proxy); refine if promising
            approx_dup = 1.0 - (sample.select(name).distinct().count() /
                               max(1, sample.select(name).count()))
            score = (1.0 - rate) * (1.0 - approx_dup)

            best = max(best or {}, {"name": name, "rate": rate,
                                    "approx_dup": approx_dup, "score": score},
                       key=lambda x: x.get("score", -1))
        # Severity based on null rate
        sev = "PASS"
        r = best.get("rate", 1.0) if best else 1.0
        if r > error: sev = "FAIL"
        elif r > warn: sev = "WARN"
        elif r > info: sev = "INFO"
        return {"selected": best, "severity": sev}
    finally:
        sample.unpersist(blocking=False)
```

5.3 Schema fingerprinting and drift classification

Signal: SCHEMA_DRIFT

Fingerprint: name, type, and nullability are normalized, sorted, JSON-encoded, and hashed.

Schema fingerprint (lowercase names/types + nullability)

```
def schema_fingerprint(cols_types):
    # cols_types: list of (name, type, nullable)
    norm = [{"name": n.lower(), "type": t.lower(), "nullable": bool(nullable)}
            for n, t, nullable in cols_types]
    norm = sorted(norm, key=lambda x: x["name"])
    blob = json.dumps(norm, separators=(",", ":"), sort_keys=True)
    return hashlib.sha256(blob.encode("utf-8")).hexdigest()
```

Classification rules (SemVer guidance).

- Fingerprint match to active registry version -> PASS.
- Additive columns (no type/nullability changes to existing) -> candidate MINOR evolution.
- Widening nullability (not null -> nullable) -> candidate MINOR, warn.
- Type changes on existing columns -> breaking -> candidate MAJOR, fail.
- Column drops or renames -> breaking unless mapped by approved rename map -> MAJOR, fail.
- Cosmetic-only (case changes, comments) -> PATCH, pass with note.

Drift detection outline

```
def detect_drift(current_schema, registry_active, registry_rename_map):
    fp_current = schema_fingerprint(current_schema)
    if fp_current == registry_active["fingerprint"]:
        return {"severity": "PASS", "kind": "match"}

    # Case-insensitive compare to detect cosmetic or rename candidates
    names_now = {c["name"].lower(): c for c in current_schema}
    names_reg = {c["name"].lower(): c for c in registry_active["columns"]}

    added = [n for n in names_now if n not in names_reg]
    dropped = [n for n in names_reg if n not in names_now]
    type_changes = []
    nullability_widen = []
    for n in (set(names_now.keys()) & set(names_reg.keys())):
        a, b = names_now[n], names_reg[n]
        if a["type"].lower() != b["type"].lower():
            type_changes.append(n)
        elif (b.get("nullable") is False) and (a.get("nullable") is True):
            nullability_widen.append(n)

    # Apply approved rename map before final decision
    unresolved_drops = [n for n in dropped if n not in registry_rename_map]
    if type_changes or unresolved_drops:
        return {"severity": "FAIL", "kind": "breaking",
                "type_changes": type_changes, "dropped": unresolved_drops}
```

```

if added or nullability_widen:
    return {"severity": "WARN", "kind": "minor",
            "added": added, "nullability_widen": nullability_widen}

# Fallback: fingerprint mismatch but no structural deltas found
return {"severity": "INFO", "kind": "cosmetic_or_metadata"}

```

5.4 Parquet health checks

Signals: PARQUET_CORRUPT, PARQUET_TINY_FILES

Corrupt file detection (single-pass). Use Spark read with default behavior (do not ignore corrupt files). Catch errors and extract offending paths from exception messages. No secondary PyArrow crawl is used.

Corrupt file detection

```

def detect_corrupt_parquet(spark, path_glob):
    try:
        _ = spark.read.parquet(path_glob).limit(1).count()
        return {"severity": "PASS", "corrupt_paths": []}
    except Exception as e:
        paths = extract_paths_from_exception(e) # parse message for path(s)
        sev = "FAIL" if paths else "WARN"
        return {"severity": sev, "corrupt_paths": paths, "error": str(e)[:2000]}

```

Tiny file detection. Small Parquet files hurt performance. Default threshold is 4 MiB per data file and a warn if more than 500 files in the scan set.

Tiny file heuristic

```

def tiny_files(fs_ls_rows, size_bytes_threshold=4*1024*1024, warn_count_threshold=500):
    files = [r for r in fs_ls_rows if r["path"].endswith(".parquet")]
    tiny = [r for r in files if r["size"] <= size_bytes_threshold]
    severity = "PASS"
    if len(files) > warn_count_threshold or len(tiny) > 0:
        severity = "WARN"
    return {"severity": severity,
            "total_files": len(files),
            "tiny_files": len(tiny),
            "tiny_sample": [r["path"] for r in tiny[:20]]}

```

5.5 Sampling policy for evidence

Row samples are bounded by two caps: `-sample-cap-per-finding` (default 500) and `-sample-cap-total` (default 2000). Allocation is round-robin across findings.

Round-robin sample allocator

```
def allocate_samples(findings, cap_per=500, cap_total=2000):
    remaining = cap_total
    allocated = {f["code"]: 0 for f in findings}
    # continue assigning one row at a time to each finding until exhausted
    while remaining > 0:
        progressed = False
        for f in findings:
            code = f["code"]
            if allocated[code] < cap_per and f["sample_source"].has_next():
                row = f["sample_source"].next()
                write_sample_row(code, row) # append to parquet shard per finding
                allocated[code] += 1
                remaining -= 1
                progressed = True
                if remaining == 0:
                    break
        if not progressed:
            break
    return allocated
```

5.6 Memory management and persistence

Intermediate frames may be cached when reused; prefer disk-backed storage for large frames.

- Use `persist(StorageLevel.DISK_ONLY)` for large intermediate results.
- Always call `unpersist(blocking=False)` in a `finally` block.
- Avoid caching raw Bronze or Silver unless reused across multiple checks.
- For exact diffs, repartition by natural key before set difference to balance shuffles.

5.7 Complexity and limits

- Count-based parity is $O(N)$ scans on each side.
- Exact symmetric difference requires at least one shuffle (potentially expensive).
- Natural key estimation uses a sample of at most $1e6$ rows; exact counts only when near thresholds.
- Corrupt detection stops at first reader exception; tiny-file scan is $O(\text{number of files})$.

5.8 Finding codes and severities

This module emits the following canonical finding codes:

- **ROW_PARITY_P1**: CSV vs Parquet (Phase 1). PASS if ratio ≤ 1 percent; otherwise FAIL.
- **ROW_PARITY_P3**: Bronze vs Silver (Phase 3). PASS if ratio $\leq$ configured threshold (default 5 percent); otherwise FAIL.

- **NATKEY_QUALITY**: Natural key null/dup assessment. PASS/WARN/FAIL based on configured rates.
- **SCHEMA_DRIFT**: Comparison to Schema Registry. PASS on match; WARN on additive/widen-only; FAIL on breaking changes.
- **PARQUET_CORRUPT**: Reader errors indicate corrupt files. FAIL when any corrupt paths are confirmed.
- **PARQUET_TINY_FILES**: Too many or too-small files. WARN by default with file counts and samples.

Mode transparency. Each finding records `mode_requested`, `mode_effective`, and (if applicable) `fallback_reason`. This is also echoed in `config.json`.

6 Schema Registry & Proposal Workflow

This section defines how the module governs Silver schema changes through a simple registry (two Delta tables) and an approval workflow. It covers table design, SemVer rules, bootstrap, proposal creation, governance, permissions, and examples.

6.1 Objectives

- Provide an authoritative record of approved Silver schemas over time.
- Detect drift vs. the active version and classify changes per SemVer.
- Allow engineers to submit proposals for schema evolution with artifacts.
- Support an expedited path for urgent, production-blocking changes.

6.2 Tables

Two Delta tables in Unity Catalog (or path-based if UC is not available):

schema_registry_schemas Canonical, approved versions.

- `version_semver` (STRING, PK within registry)
- `fingerprint` (STRING, SHA256 of normalized columns)
- `created_at_utc` (TIMESTAMP)
- `created_by` (STRING)
- `status` (STRING: active, deprecated)
- `columns_json` (STRING: JSON array of name, type, nullable)
- `rename_map_json` (STRING: JSON object of old->new, optional)
- `notes` (STRING, optional)

schema_registry_proposals Pending or approved evolutions.

- proposal_id (STRING, UUID)
- submitted_at_utc (TIMESTAMP)
- submitted_by (STRING)
- base_version (STRING, references schema_registry_schemas)
- proposed_version (STRING, SemVer)
- change_type (STRING: patch, minor, major)
- status (STRING: pending, approved, rejected, superseded)
- priority (STRING: normal, expedited)
- fingerprint_new (STRING)
- diff_summary_json (STRING: JSON lists of added, dropped, type_changes, nullability_widen)
- artifacts_path (STRING: DBFS path to proposal evidence)
- approved_by (STRING, nullable)
- approved_at_utc (TIMESTAMP, nullable)
- effective_at_utc (TIMESTAMP, nullable)
- notes (STRING, optional)

DDL (illustrative)

DDL for registry tables (Spark SQL)

```
-- Adjust <catalog>.<schema> as needed
CREATE TABLE IF NOT EXISTS <catalog>.<schema>.schema_registry_schemas (
  version_semver STRING,
  fingerprint STRING,
  created_at_utc TIMESTAMP,
  created_by STRING,
  status STRING,
  columns_json STRING,
  rename_map_json STRING,
  notes STRING
)
USING DELTA;

CREATE TABLE IF NOT EXISTS <catalog>.<schema>.schema_registry_proposals (
  proposal_id STRING,
  submitted_at_utc TIMESTAMP,
  submitted_by STRING,
  base_version STRING,
  proposed_version STRING,
  change_type STRING,
  status STRING,
  priority STRING,
  fingerprint_new STRING,
```

```

diff_summary_json STRING,
artifacts_path STRING,
approved_by STRING,
approved_at_utc TIMESTAMP,
effective_at_utc TIMESTAMP,
notes STRING
)
USING DELTA;

```

6.3 SemVer rules and drift classification

The module classifies observed schema changes relative to the active registry version:

- PATCH: cosmetic only (case changes, comments). Pass.
- MINOR: additive columns or widening nullability (not null -> nullable). Warn.
- MAJOR: column type changes, drops, or renames not covered by rename map. Fail.

On WARN or FAIL, the module can optionally submit a proposal automatically when `-propose-on-drift auto` is set (Section 7 details API/flags).

6.4 Bootstrap logic

If the registry is enabled and empty, a pre-flight bootstrap can initialize it from the current Silver table schema. Controlled by `-schema-registry-bootstrap` or env `SCHEMA_REGISTRY_BOOTSTRAP=auto|off`. Default is `auto`.

Behavior.

1. If tables do not exist, create them.
2. If `schema_registry_schemas` is empty and `bootstrap=auto`:
 - Snapshot the current Silver schema.
 - Compute fingerprint (Section 5).
 - Insert version 1.0.0 with `status=active`.
3. If non-empty, do nothing.

Bootstrap snippet (illustrative)

Bootstrap if empty

```

def bootstrap_registry_if_needed(spark, silver_fqn, schemas_fqn,
                                created_by="investigator", now_utc=None):
    now_utc = now_utc or F.current_timestamp()
    if spark.table(schemas_fqn).limit(1).count() > 0:
        return False # already bootstrapped

    df = spark.table(silver_fqn)
    cols = [{"name": f.name, "type": f.dataType.simpleString(), "nullable": f.nullable}

```

```

        for f in df.schema.fields]
fp = schema_fingerprint([(c["name"], c["type"], c["nullable"]) for c in cols])
payload = [( "1.0.0", fp, # version_semver, fingerprint
            )]
(spark.createDataFrame(payload, "version_semver string, fingerprint string")
 .withColumn("created_at_utc", F.current_timestamp())
 .withColumn("created_by", F.lit(created_by))
 .withColumn("status", F.lit("active"))
 .withColumn("columns_json", F.to_json(F.array([F.struct(
     F.lit(c["name"]).alias("name"),
     F.lit(c["type"]).alias("type"),
     F.lit(c["nullable"]).alias("nullable")) for c in cols])))
 .withColumn("rename_map_json", F.lit(None).cast("string"))
 .withColumn("notes", F.lit("bootstrap from current Silver")))
 .write.mode("append").saveAsTable(schemas_fqn))
return True

```

6.5 Proposal creation

When drift is detected, the module can write a row into `schema_registry_proposals`. This is gated by flags:

- `-propose-on-drift auto|off` (default: auto)
- `-priority normal|expedited` (default: normal)

Proposal snippet (illustrative)

Insert proposal row

```

def submit_proposal(spark, proposals_fqn, base_version, proposed_version,
                    change_type, fingerprint_new, diff_summary_json,
                    artifacts_path, submitted_by, priority="normal"):
    row = [{
        "proposal_id": str(uuid.uuid4()),
        "submitted_at_utc": datetime.utcnow().isoformat(timespec="seconds") + "Z",
        "submitted_by": submitted_by,
        "base_version": base_version,
        "proposed_version": proposed_version,
        "change_type": change_type, # patch / minor / major
        "status": "pending",
        "priority": priority, # normal / expedited
        "fingerprint_new": fingerprint_new,
        "diff_summary_json": diff_summary_json, # JSON string
        "artifacts_path": artifacts_path,
        "approved_by": None,
        "approved_at_utc": None,
        "effective_at_utc": None,
        "notes": None
    }]
    spark.createDataFrame(row).write.mode("append").saveAsTable(proposals_fqn)

```

6.6 Governance and approval

States. pending -> approved -> effective; or pending -> rejected.

RACI (concise)

Role	Responsibility	Action	Notes
Submitter	Propose change	Create proposal	Provides artifacts, diff summary
Data Steward	Review	Approve/Reject	Validates semantics, mappings
Governance Board	Oversight	Approve major	Enforces SemVer and policy
Platform Eng	Apply	Activate version	Updates status to active, deploys

Expedited path. Proposals with `priority=expedited` trigger on-call notification and a 24-hour SLA. Use only for production-blocking incidents. The job and evidence must link to the incident.

6.7 Permissions and audit

- SELECT on both tables for investigators and readers.
- INSERT on proposals for engineers and stewards.
- UPDATE on schemas status only for platform eng or governance service account.
- All writes are audited via standard UC audit logs and structured module logs.

6.8 Examples

Example schema version record

schema_registry_schemasrow

```
{
  "version_semver": "1.2.0",
  "fingerprint": "5f13...9a7b",
  "created_at_utc": "2025-09-22T13:40:05Z",
  "created_by": "governance-service",
  "status": "active",
  "columns_json": "[{\"name\":\"natural_key\",\"type\":\"string\",\"nullable\":false}, ...]",
  "rename_map_json": "{\"recipient_uei\":\"recipient_id\"}",
  "notes": "Added recipient_id (minor)."
}
```

Example proposal record

schema_registry_proposalsrow

```
{
  "proposal_id": "79c4b8a4-62d1-4b2c-9a5d-ccf9fdc64a51",
  "submitted_at_utc": "2025-09-23T02:11:59Z",
  "submitted_by": "bee.r.davis@nasa.gov",
  "base_version": "1.2.0",
  "proposed_version": "2.0.0",
  "change_type": "major",
}
```

```

"status": "pending",
"priority": "expedited",
"fingerprint_new": "b71e...2c54",
"diff_summary_json": "{\"dropped\": [\"award_id_piid\"], \"type_changes\": [\"recipient_id\"]
  }",
"artifacts_path": "dbfs:/mnt/idas/investigations/INV-20250923T020900Z-.../artifacts/",
"approved_by": null,
"approved_at_utc": null,
"effective_at_utc": null,
"notes": "Blocking downstream gold refresh."
}

```

7 Evidence Packaging & Retention

7.1 Bundle layout

Evidence bundle (ASCII tree)

```

INV-YYYYMMDDThhmmssZ-<uuid4hex>/
  report.md
  findings.jsonl
  config.json
  metrics.json
  logs.ndjson
  samples/
    FND_ROW_PARITY_001_sample.parquet
    FND_NATKEY_NULLS_002_sample.parquet
    ...
  artifacts/
    manifests_snapshot.jsonl
    schema_bronze.json
    schema_silver.json
    schema_registry_match.json
    diff_summary.csv

```

7.2 Sampling caps

- Max 500 rows per finding, 2000 rows per investigation (round-robin allocation).
- Allocation status and any truncation are recorded in `metrics.json`.

7.3 Retention policy

- Suggested: retain 30 days in dev/test, 90 days in prod (configurable).
- Old bundles can be purged by a scheduled job; do not delete artifacts referenced by open tickets.

7.4 Packaging

- Optional: create a tarball named `$INV_ID.tar.gz` for attachment to tickets.
- Large Parquet samples can be pruned to a minimal reproducible set upon closure.

8 Security, Privacy, Compliance

8.1 Data handling

- Read-only operations; no mutation of Bronze/Silver/Gold tables.
- All timestamps in UTC; outputs are scoped to the evidence root.
- Single-user security mode recommended when outputs may include sensitive fields.

8.2 Access control

- UC READ on Bronze/Silver (or DBFS READ if path-based).
- WRITE on \$OUTPUT_ROOT and \$OUTPUT_ROOT/locks.
- Access to RAW Phase 1 evidence is optional but recommended when row parity includes CSV->Parquet.

8.3 Auditability

- Structured logs (logs.ndjson) include investigation_id, principal, and parameters used.
- Job run URL should be embedded in report.md and each finding.

8.4 Secrets & tokens

- Use Databricks secrets for any credentials (if ever required for future sources).
- No secrets are written to evidence bundles or logs.

9 Cost & Sizing Guidance

9.1 Reference cluster

Single-node cluster (example)

```
{
  "num_workers": 0,
  "spark_version": "17.2.x-scala2.13",
  "spark_conf": {"spark.databricks.cluster.profile": "singleNode" },
  "node_type_id": "i4i.4xlarge",
  "autotermination_minutes": 20,
  "data_security_mode": "SINGLE_USER"
}
```

9.2 Typical costs

- Reference run: USD 0.15–0.80 for small-to-medium chunks (region/sku dependent).
- Exact mode is more expensive; the module auto-falls back to approx when row counts exceed -exact-max-rows.

9.3 Throughput tips

- Prefer approx mode for routine checks; reserve exact for small suspect subsets.
- Ensure Parquet file sizes are healthy (128–512 MB target) to reduce I/O overhead.
- Avoid wide shuffles in ad-hoc analysis; filter early, select minimal columns.

10 Runbook & Troubleshooting

10.1 Common issues and fixes

- **Auth error:** Re-run Databricks CLI login and verify with `databricks current-user me -profile <name>`.
- **Lock conflict:** Another run is active. Either wait for TTL expiry or, if truly stale, reclaim after `expires_utc`.
- **UC permission denied:** Confirm READ on Bronze/Silver and WRITE on `$OUTPUT_ROOT`.
- **Exact fallback:** When counts exceed `-exact-max-rows`, mode becomes `approx`; recorded in `config.json`.
- **Corrupt Parquet:** A finding will list offending paths; re-stage affected chunks or re-run Phase 1 for that slice.
- **Schema mismatch:** If legitimate evolution, file a proposal; otherwise, revert upstream changes or hotfix transform.

10.2 Operational checklist

1. Confirm inputs exist and are readable (RAW/Bronze/Silver).
2. Ensure profile auth and cluster policy compliance.
3. Run job with appropriate parameters and priority.
4. Attach `report.md` and key samples to the tracking ticket.
5. For schema evolution, shepherd proposal through governance.