Waltero™

RF-04 · WALTERO MIMIR GUIDE

Metering API Guide

Three ways to read meter data from the Waltero platform — high-level metering application, raw platform-api, and the data-audit changelog

VERSION	1.6
DATE	2026-05-26
AUDIENCE	Partner integrators and customers

Contents

This guide covers three ways to read meter data from the Waltero platform.

Part 1 — Daily consumption from the Metering application. One call, one response, all meters in your organization, consumption already converted to the meter's display units. Start here unless you specifically need raw values or a custom aggregation.

Part 2 — Raw meter readings from platform-api. The lower-level walkthrough: find devices → fetch their meter type → read `meter.value.processed` → apply the multiplier yourself. Use this when you need raw readings, an aggregation interval Part 1 doesn't offer, or fields beyond consumption (battery, RSSI, temperature, …).

Part 3 — Meter value changelog (data-audit API). Track every correction to a sensor's stored readings. Use this to answer "what data has changed since my last call?" — useful for replicating Waltero readings into your own system without missing post-hoc corrections.

The full OpenAPI 3 spec for platform-api ships alongside this guide as `platform-api.openapi.json` (see Part 2 §2.7 for Postman import).

Conventions (apply to all three parts)

Item	Value
Base URL (EMEA prod)	`https://qs0iya7cfk.execute-api.eu-central-1.amazonaws.com/prod`
Auth	Cognito JWT — `Authorization: Bearer <token>`
Login	`POST /api/auth/login` with `{username, password}` → `{token, accessToken, refreshToken, user}`
Token TTL	24 h. Re-login via `POST /api/auth/login` to get a fresh token. (The dedicated refresh endpoints are not currently usable for service integrations — see Part 2 §2.8.)

Is the base URL fixed? Yes — for your EMEA prod tenant, the host above is stable and won't change as new customers onboard. Other regions and environments (APAC prod, EMEA staging, on-prem deployments) have different hosts but identical endpoint paths and request/response contracts, so only the host needs to change if you later integrate against another region.

```
BASE='https://qs0iya7cfk.execute-api.eu-central-1.amazonaws.com/prod'

TOKEN=$(curl -s -X POST "$BASE/api/auth/login" \
  -H 'Content-Type: application/json' \
  -d '{"username":"you@example.com","password":"…"}' | jq -r .token)
```

Part 1 — The Metering Application

How to fetch **daily consumption per meter** for every meter in your organization. One call per organization. The response contains one row per meter per day, with consumption already in the meter's display units — no client-side multiplication, no delta math, no rollover handling.

1.1 Why `/v3/`?

The metering application is versioned. As long as you pin `/v3/` in your URL, the response shape is frozen. The unversioned URL follows whatever Waltero has flagged as the current default — that pointer can move at any time and a new version may add, rename, or remove fields. **Pin `/v3/` in any URL your integration depends on.**

1.2 Find your organization UUID

`GET /api/auth/me` returns your user record plus `accessible_organizations` — every organization UUID you can act on, including descendants in the org hierarchy:

```
curl -s "$BASE/api/auth/me" \
  -H "Authorization: Bearer $TOKEN" | jq '{
    primary_org:    .user.organizations[0].organization_id,
    all_accessible: .accessible_organizations
  }'
```

```
{
  "primary_org": "00000000-0000-0000-0000-000000000001",
  "all_accessible": [
    "00000000-0000-0000-0000-000000000001",
    "00000000-0000-0000-0000-000000000002"
  ]
}
```

The `id` is what you'll pass as the path parameter `{orgId}` in every URL below. Alternatively, list orgs by name:

```
curl -s "$BASE/api/organizations/?search=YourOrgName&limit=20" \
  -H "Authorization: Bearer $TOKEN" \
| jq '.data[] | {id, name, device_count}'
```

1.3 Fetch daily consumption per meter

```
ORG='00000000-0000-0000-0000-000000000001'
START='2026-05-18T22:00:00.000Z'
END='2026-05-25T22:00:00.000Z'

curl -s "$BASE/api/organizations/$ORG/applications/metering/v3/data/device-daily-
consumption\
?startDate=$START\
&endDate=$END\
&resolution=day" \
  -H "Authorization: Bearer $TOKEN" | jq
```

Response


```
{
  "data": [
    { "deviceId": "11111111-1111-1111-1111-111111111101",
      "deviceName": "Building 1", "serialNumber": 1001,
      "date": "2026-05-19", "consumption": 520 },
    { "deviceId": "11111111-1111-1111-1111-111111111101",
      "deviceName": "Building 1", "serialNumber": 1001,
      "date": "2026-05-20", "consumption": 450 },
    { "deviceId": "11111111-1111-1111-1111-111111111102",
      "deviceName": "Building 2", "serialNumber": 1002,
      "date": "2026-05-19", "consumption": 310 }
  ],
  "meta": {
    "count":      147,
    "resultKind": "table",
    "appliedFilters": {
      "organizationId": "00000000-0000-0000-0000-000000000001",
      "startDate":      "2026-05-18T22:00:00.000Z",
      "endDate":        "2026-05-25T22:00:00.000Z",
      "resolution":     "day",
      "_timezone":      "UTC",
      "_rangeMode":     "exclusiveEnd"
    }
  }
}
```


One row per `(deviceId, date)`. Row count is approximately `nMeters × nDays` — see "Nullable rows" below.

Field	Meaning
`deviceId`	UUID of the meter device.
`deviceName`	Human-readable name. **Not always set** — when the name field is empty the API falls back to the device's external ID (typically a string, but can be a number when only the serial is set). Code defensively.
`serialNumber`	Meter serial number.
`date`	UTC calendar day (`YYYY-MM-DD`). **Omitted** on no-data rows — see below.
`consumption`	Consumption for that day, in the meter's display units (e.g. litres). **May be `null`** on days with no data.

Nullable rows

A meter with no readings for a given day still appears in the response, but the row has `consumption: null` and the `date` field is **omitted entirely**:

```
{ "deviceId": "…", "deviceName": "…", "serialNumber": 20256, "consumption": null }
```

Filter these out before computing totals:

```
… | jq '[.data[] | select(.date != null and .consumption != null)]'
```

1.4 Parameters

Param	Where	Required	Notes
`{orgId}`	path	yes	The organization scope. Picked up from the path — no need to repeat as a query param.
`startDate`	query	yes	UTC, ISO-8601, e.g. `2026-05-18T22:00:00.000Z`.
`endDate`	query	yes	UTC, ISO-8601. **Exclusive** — a row at exactly `endDate` is not included.
`resolution`	query	yes	`day`, `week`, `month`, or `year`. Controls the bucket size and the `date` label.

Date semantics

Day labels are **UTC calendar days**. The API takes UTC `startDate` / `endDate`, computes day buckets in UTC, and returns `date` strings like `2026-05-19` corresponding to UTC midnight boundaries.

Resolution → `date` label format

`resolution`	`date` value example
`day`	`2026-05-19`
`week`	`2026-W21` (ISO week)
`month`	`2026-05`
`year`	`2026`

1.5 Common variations

A — Total consumption per meter for the period

Sum the rows for each device client-side:

```
curl -s "$BASE/api/organizations/$ORG/applications/metering/v3/data/device-daily-
consumption\
?startDate=$START&endDate=$END&resolution=day" \
  -H "Authorization: Bearer $TOKEN" \
| jq '[.data
       | group_by(.deviceId)[]
       | { deviceName:   .[0].deviceName,
           serialNumber: .[0].serialNumber,
           total:        ([.[].consumption // 0] | add) }]
     | sort_by(-.total)'
```

B — Weekly or monthly consumption instead of daily

Same call, change `resolution`:

```
curl -s "$BASE/api/organizations/$ORG/applications/metering/v3/data/device-daily-
consumption\
?startDate=2026-01-01T00:00:00.000Z\
&endDate=2026-05-25T22:00:00.000Z\
&resolution=month" \
  -H "Authorization: Bearer $TOKEN" | jq '.data[0:3]'
```

```
[
  { "deviceId": "11111111-1111-1111-1111-111111111101", "deviceName": "Building 1",
    "serialNumber": 1001, "date": "2026-01", "consumption": 14800 },
  { "deviceId": "11111111-1111-1111-1111-111111111101", "deviceName": "Building 1",
    "serialNumber": 1001, "date": "2026-02", "consumption": 13950 },
  { "deviceId": "11111111-1111-1111-1111-111111111101", "deviceName": "Building 1",
    "serialNumber": 1001, "date": "2026-03", "consumption": 15120 }
]
```

C — One specific meter

Filter client-side by serial number:

```
curl -s "$BASE/api/organizations/$ORG/applications/metering/v3/data/device-daily-
consumption\
?startDate=$START&endDate=$END&resolution=day" \
  -H "Authorization: Bearer $TOKEN" \
| jq --arg sn 1001 '[.data[] | select(.serialNumber == ($sn|tonumber))]'
```

D — The list of meters in your organization

```
curl -s "$BASE/api/organizations/$ORG/applications/metering/v3/data/meter-list" \
  -H "Authorization: Bearer $TOKEN" | jq '.data'
```

```
[
  { "deviceId": "11111111-1111-1111-1111-111111111101",
    "deviceName": "", "serialNumber": 1001, "externalMeterId": "Building 1" },
  { "deviceId": "11111111-1111-1111-1111-111111111102",
    "deviceName": "", "serialNumber": 1002, "externalMeterId": "Building 2" }
]
```

Sorted by `serialNumber` ascending.

1.6 Errors

HTTP	When
401	Token missing or expired (24 h). Re-login via `POST /api/auth/login`.
403	Organization outside your `accessible_organizations`. The API does not distinguish "no access" from "doesn't exist" — both return 403.
404	`{"code":"VERSION_NOT_AVAILABLE"}` — the version you pinned (e.g. `/v99/`) isn't deployed.
400	`INVALID_VERSION` if `/vN/` is not a positive integer.
504	Underlying query timeout — narrow the date range or use a coarser `resolution`.

Part 2 — Raw meter readings from platform-api

If the Metering application from Part 1 doesn't fit your use case — you need raw readings, an aggregation interval other than `day` / `week` / `month` / `year`, or fields beyond consumption (battery, RSSI, temperature, etc.) — read the data directly from platform-api. The flow is:

1. Find the **organization UUID(s)** you can access.
2. List the **devices** in an organization.
3. For each device, get its **details** — which gives you the meter type (with multiplier).
4. Read `meter.value.processed` for the time range you want.
5. Multiply the raw value by the meter type's `multiplier` to get the dashboard-equivalent number.

2.1 Find your organization UUID

Two ways, both authenticated with the JWT from the top-of-guide conventions.

A — from the logged-in user. `GET /api/auth/me` returns your user record plus `accessible_organizations` — every organization UUID you can act on, including descendants in the org hierarchy:

```
curl -s "$BASE/api/auth/me" \
  -H "Authorization: Bearer $TOKEN" | jq '{
    primary_org:    .user.organizations[0].organization_id,
    all_accessible: .accessible_organizations
  }'
```

```
{
  "primary_org": "f47ac10b-58cc-4372-a567-0e02b2c3d479",
  "all_accessible": [
    "f47ac10b-58cc-4372-a567-0e02b2c3d479",
    "a1b2c3d4-e5f6-4a5b-8c9d-1e2f3a4b5c6d"
  ]
}
```

B — list organizations directly. `GET /api/organizations/?search=<name>` returns full org records (name, address, device count, parent org):

```
curl -s "$BASE/api/organizations/?search=Beulco&limit=20" \
  -H "Authorization: Bearer $TOKEN" \
| jq '.data[] | {id, name, parentorganizationid, device_count}'
```

Either way, the `id` (a UUID) is what you'll pass everywhere as `organizationId`.

2.2 List devices in an organization

> **Device IDs are unchanged.** The UUIDs you already have from the legacy API are the same UUIDs in this API — there is no remapping. The same physical meter has the same `id` in both APIs, and that ID is stable for the life of the device.

```
ORG='f47ac10b-58cc-4372-a567-0e02b2c3d479'

curl -s "$BASE/api/devices/?organizationId=$ORG&limit=100&page=1" \
  -H "Authorization: Bearer $TOKEN" \
| jq '.data[] | {id, name, serialnumber, externalmeterid, isactivated}'
```

The response is paginated: `{"data": [...], "pagination": {"page", "limit", "total", "pages"}}`. `limit` defaults to 20, max 100 — page through with `?page=N` for orgs with more than 100 devices. Useful query parameters:

Param	Effect
`organizationId=<uuid>`	Required — the org you want to list devices for.
`search=<text>`	Match on device name, ID, or IMEI.
`status=online | offline | error`	Filter by current status.
`sortBy=serialnumber | externalmeterid`, `sortOrder=…`	Sort order.

2.3 Get device details (and the meter type)

```
DEV='<deviceId from §2.2>'

curl -s "$BASE/api/devices/$DEV" \
  -H "Authorization: Bearer $TOKEN" | jq
```

The device record includes a `metertype` object that the API has already resolved using a **DDB-first, RDS-fallback** chain — so you don't need to do anything different for legacy devices, the right value is already there.

```
{
  "id":           "28b2e23d-8d3b-4559-9afe-6b723614888c",
  "name":         "Example meter",
  "serialnumber": 1234,
  "metertypeid":  "12e2c7b1-fb15-42f6-ad9c-a28e450912df",
  "metertype": {
    "id":             "12e2c7b1-fb15-42f6-ad9c-a28e450912df",
    "name":           "Example water meter",
    "manufacturer":   "Acme",
    "medium":          null,
    "unit":           "m³",
    "multiplier":     0.001,
    "expectedDigits":  null,
```

```
    "source":          "global",
    "organizationId":   null,
    "kind":            "MULTIPLIER"
  }
}
```

The fields you'll use are `multiplier` (for §2.5) and `unit` (for display). The remaining fields (`medium`, `expectedDigits`, `source`, `organizationId`) are diagnostic — read-only metadata that you can ignore unless you're building admin tooling.

The same enrichment is applied to every row of the device list from §2.2, so for a bulk pull you can read `multiplier` straight off the listing without a per-device call:

```
curl -s "$BASE/api/devices/?organizationId=$ORG&limit=100" \
  -H "Authorization: Bearer $TOKEN" \
| jq '.data[] | {id, name, multiplier: .metertype.multiplier}'
```

Meter-type resolution rules

The platform applies this fallback chain server-side, so you should not need to do it yourself:

1. **DDB assignment** — the org-scoped meter type assigned via the dashboard. Carries `name`, `manufacturer`, `unit` label, `multiplier`, `kind`.
2. **Legacy RDS assignment** — `devices.metertypeid` JOIN `metertype`. Carries `name` (from `metertype.model`), `manufacturer` (from `metertype.make`), and `multiplier` (from `metertype.unit` — the column name is misleading; the value is a number, not a unit string). `unit` label is `null`.
3. **No usable assignment** — `device.metertype` may be absent **or** it may be a placeholder object whose `multiplier` is `null` (or not a positive finite number).

Your only client-side rule: **if `device.metertype?.multiplier` is not a positive finite number, treat the multiplier as `1` (pass-through).** This single check covers both flavours of case 3. Sample code in §2.5.

> **Heads-up:** do **not** call *`GET /api/devices/{id}/meter-type`* for this. That endpoint reads DDB only and returns 404 for legacy devices, even when the device record itself has a perfectly good *`metertype`* object.

2.4 Get `meter.value.processed` for a time range

```
START='2026-04-22T00:00:00Z'
END='2026-04-29T00:00:00Z'

curl -s "$BASE/api/telemetry/?deviceIds=$DEV\
&startDate=$START&endDate=$END\
&telemetry=meter.value.processed\
&interval=1h&function=last" \
  -H "Authorization: Bearer $TOKEN" | jq
```

- `interval` accepts `1m`, `5m`, `15m`, `30m`, `1h`, `6h`, `12h`, `1d`, `7d`. Omit `interval` / `function` to get raw rows.
- `function`: `avg | sum | min | max | count | first | last`. For meter readings you almost always want `last`.

- `deviceIds` accepts a comma-separated list (max **50** per call) — the same call works for one device or many.

> **Use** ***`meter.value.processed`***. It is the preferred customer-facing field name and the one the API itself recommends. The legacy name *`meter.value.corrected`* is still accepted as an alias on this endpoint but cannot be combined with *`.processed`* in a single request — doing so returns HTTP 400 *`CONFLICTING_FIELD_ALIASES`*. (The audit log in Part 3 still uses *`.corrected`* as its canonical name — see §3.2.)

Response shape (one block per device, rows grouped by **measure category** — `telemetry` / `connection` / `meta` — not by field name):

```
{
  "devices": {
    "<deviceId>": {
      "metadata":   { "deviceId": "…", "timeRange": {…}, "aggregation": {…} },
      "data": {
        "telemetry": [
          { "timestamp": "2026-04-22T00:00:00.000Z", "meter.value.processed":
211120 },
          { "timestamp": "2026-04-23T00:00:00.000Z", "meter.value.processed":
213440 },
          { "timestamp": "2026-04-29T00:00:00.000Z", "meter.value.processed":
222350 }
        ]
      },
      "pagination": { "count": 155, "hasMore": false }
    }
  },
  "summary": { "requested": 1, "successful": 1, "failed": 0, "totalReadings": 155 }
}
```

Each row carries a `timestamp` plus every requested field as its own key. If you also asked for `connection=rssi`, you'd get a sibling `data.connection` array with `{timestamp, rssi}` rows. See §2.6 for the catalogue of field names and their units.

Top-level **`summary`** (multi-device pulls): tells you which devices returned data. If `summary.failed > 0`, inspect each device's block for an absent `data` or empty rows — partial failures don't fail the request.

Per-device **`pagination`**: `hasMore: true` means a single page hit the underlying engine's row cap. Narrow the time window or use a coarser `interval` to get the rest.

2.5 Convert raw → display value

Multiply the raw `meter.value.processed` from §2.4 by the `multiplier` from §2.3:

```
def converted(raw, device):
    mt = device.get("metertype") or {}
    m = mt.get("multiplier")
    if not (isinstance(m, (int, float)) and m > 0):
```

```
        m = 1
    return raw * m
```

For a device with `multiplier = 0.001` and the time-series rows from §2.4:

```
2026-04-22T00:00 → 211120 × 0.001 = 211.120 m³
2026-04-23T00:00 → 213440 × 0.001 = 213.440 m³
2026-04-29T00:00 → 222350 × 0.001 = 222.350 m³
```

That's it.

2.6 Field reference (units & names)

Measure	Field	Type / Unit	Notes
`telemetry`	`meter.value`	number, meter's native units (raw display digits)	The reading as it came off the device.
`telemetry`	`meter.value.processed`	number, same units as `meter.value`	**Use this one.** It's `meter.value` with backend corrections (suspicious-value replacement, recovery) applied. Multiply by `device.metertype.multiplier` for the dashboard value (see §2.5). Legacy alias: `meter.value.corrected` (same field, same value — see note below).
`telemetry`	`meter.value.corrected`	alias	Legacy name for `meter.value.processed`. Still accepted on the telemetry endpoint but cannot be combined with `.processed` in the same request (HTTP 400 `CONFLICTING_FIELD_ALIASES`). **This is also the canonical name used by the audit log in Part 3** — Part 3 filters must use `.corrected`.
`telemetry`	`battery_level`	number, **millivolts (mV)** — e.g. `4955` = 4.955 V	The platform exposes raw millivolts. The value is the same one you receive today, so your existing mV → percentage conversion

Measure	Field	Type / Unit	Notes
			can keep working unchanged — just feed `battery_level` into it. The Waltero platform itself does not produce a percentage.
`telemetry`	`temperature.1`, `temperature.2`	number, degrees Celsius (°C) — e.g. `21.4`	On-board temperature sensors. Use the schema endpoint below to see which indices a specific device emits.
`telemetry`	`meter.digits.N`, `meter.scores.N`	number	Per-digit OCR readings and per-digit confidence scores. `N` is `0..K` where `K` depends on the meter (typically 4 or 5). Use the schema endpoint to see which indices a specific device emits.
`telemetry`	`status`	string, e.g. `"Ok"`, `"Failed read"`	Per-reading status from the device.
`telemetry`	`execution_time`	number, milliseconds	Per-reading on-device processing time. Present on most sensors.
`connection`	`rssi`	number, dBm	Cellular / WiFi signal strength.
`connection`	`rsrp`, `rsrq`, `sinr`	number, dBm / dB / dB	LTE link-quality metrics.
`connection`	`iccid`, `imei`, `imsi`	string	SIM card identifier, device IMEI, subscriber IMSI.
`connection`	`connection_mode`	`"lt"` (LTE) | `"wf"` (WiFi)	Which radio the device is using. (The OpenAPI example shows `"lte"`; the live API returns the two-letter form `"lt"`.)
`meta`	`fw`	string, sha256 hex	Firmware identity (changes per build).

To list every field a specific device actually emits:

```
curl -s "$BASE/api/telemetry/$DEV/schema" \
  -H "Authorization: Bearer $TOKEN" | jq
```

```
{
  "telemetry":  ["acc.x", "acc.y", "acc.z", "battery_level", "boot_reason",
                 "execution_time",
                 "meter.digits.0", "meter.digits.1", "meter.digits.2",
"meter.digits.3", "meter.digits.4", "meter.digits.5",
                 "meter.scores.0", "meter.scores.1", "meter.scores.2",
"meter.scores.3", "meter.scores.4", "meter.scores.5",
                 "meter.value", "meter.value.corrected", "meter.value.processed",
                 "read_counter", "status", "temperature.1", "temperature.2"],
  "connection": ["connection_mode", "iccid", "imei", "imsi", "rsrp", "rsrq",
"rssi", "sinr"],
  "meta":       ["fw"]
}
```


This is the source of truth for "what fields can I ask for on this device" — useful for confirming the field name on a meter you haven't seen before. The exact set varies per device variant; the example above shows a typical real sensor.

2.7 OpenAPI spec & Postman collection

The full OpenAPI 3 spec for platform-api is shipped alongside this guide as `platform-api.openapi.json`. It contains every endpoint (auth, organizations, devices, telemetry, alarms, …) with parameters, request bodies, and example responses.

Import into Postman (gives you a ready-made collection — the equivalent of the legacy Postman collection):

1. Open Postman → Import → Files → choose `platform-api.openapi.json`.
2. In the import dialog, set "Import as" → **Postman Collection** (default in recent Postman versions).
3. Click Import — you'll get one folder per tag (Authentication, Devices, Telemetry, Organizations, …) with a request for every endpoint, parameters pre-filled, and example payloads from the spec.

Set up two Postman environment variables and you're done:

Variable	Example value
`BASE`	`https://qs0iya7cfk.execute-api.eu-central-1.amazonaws.com/prod`
`TOKEN`	(the `token` value returned by `POST /api/auth/login`)

The same OpenAPI file also works with code generators (`openapi-generator`, `swagger-codegen`) if you'd rather generate a typed client than hand-write requests.

2.8 Errors you might hit

HTTP	When
401	Token missing or expired (24 h). The dedicated refresh endpoints (`GET /api/auth/refresh-token`, `POST /api/auth/refresh`) are currently **not** working for non-browser integrations — re-login via `POST /api/auth/login` to get a fresh token.
403	Missing permission (`telemetry:read`, `devices:read`), org outside your `accessible_organizations`, or the device id is unknown / belongs to another org. The API does not distinguish "no access" from "doesn't exist" — both return 403.

HTTP	When
400	Missing required query param (e.g. `organizationId` on `/api/devices/`, `startDate` / `endDate` on `/api/telemetry/`), or `startDate > endDate` — body: `{ "code": "VALIDATION_ERROR", "message": "Start date must be before end date" }`. Also returned for `CONFLICTING_FIELD_ALIASES` if you request `meter.value.processed` and `meter.value.corrected` in the same call.
504	Underlying Timestream timeout — narrow the date range or use a coarser `interval`.

Part 3 — Meter value changelog (data-audit API)

Need to know **what changed** in a sensor's readings since your last call? The data-audit API surfaces every correction that has been applied to a device's stored values — when the correction was made, what the previous value was, what the new value is, and who or what made it.

Typical use cases:

- Replicate Waltero readings into your own system without missing post-hoc corrections (the platform sometimes re-resolves earlier readings when an upstream meter sends a delayed or recovered value).
- Show an audit trail of "what corrected itself today" in a customer-facing app.
- Reconcile a periodic export against the current state of truth.

> **Scope of this part.** This part covers the **single-device** endpoint only. An org-level endpoint also exists (`GET /api/organizations/{orgId}/data-audit`) but as of 2026-05 it has a pagination defect when total matched rows across devices exceed `limit` — older rows can be dropped without a `next_cursor`. Until that's fixed, prefer the per-device endpoint described below; if you need org-wide coverage, iterate the devices yourself.

3.1 The endpoint

```
GET /api/devices/{deviceId}/data-audit
```

```
DEV='<deviceId>'
FROM='2026-05-01T00:00:00Z'
TO='2026-05-25T23:59:59Z'

curl -s "$BASE/api/devices/$DEV/data-audit\
?from=$FROM&to=$TO\
&field=meter.value.corrected\
&limit=200" \
  -H "Authorization: Bearer $TOKEN" | jq
```

The endpoint returns **one row per changed field** (not per write — see §3.4).

3.2 Query parameters

Param	Where	Required	Notes
`{deviceId}`	path	yes	The device whose audit log you want.
`from`	query	yes	ISO-8601 UTC, **inclusive** lower bound on `audit_timestamp`.
`to`	query	yes	ISO-8601 UTC, **inclusive** upper bound on `audit_timestamp`.
`limit`	query	no	1–200, default 50. See §3.5 about how the limit interacts with filters.
`cursor`	query	no	Opaque cursor returned by the previous page. See §3.5.

Param	Where	Required	Notes
`measure_name`	query	no	Filter: `telemetry` | `connection` | `meta` | `other`.
`field`	query	no	Filter by specific field name. **For a meter value changelog, use `field=meter.value.corrected`** — see note below.

Field naming asymmetry. The audit log stores the meter value under its canonical name *`meter.value.corrected`*, even though Parts 1 and 2 prefer the customer-facing alias *`meter.value.processed`*. The audit endpoint does **not** apply the alias: filtering with *`field=meter.value.processed`* returns 0 rows. Use *`.corrected`* here.

Range cap: 90 days. A request with `to − from > 90 d` is rejected with HTTP 400. For longer histories, paginate the window in 90-day chunks.

***`audit_timestamp`* vs *`record_timestamp`*.** The *`from`* / *`to`* filter applies to *`audit_timestamp`* (when the audit row was written). That's the correct filter for "show me what changed since my last call." It is **not** the same as *`record_timestamp`* (when the underlying meter reading is dated): a correction made today for a reading from last week shows up in **today's** audit page, not last week's.

3.3 Response shape


```
{
  "items": [
    {
      "device_id":         "77859b0d-688e-4bf8-87c7-7207592817f8",
      "event_id":          "491f02e0-09a6-4e53-928a-24f858d10500",
      "audit_timestamp":   "2026-05-25T11:30:01.831Z",
      "record_timestamp":  1779701324,
      "measure_name":      "telemetry",
      "field":             "meter.value.corrected",
      "old_value":         184337,
      "new_value":         184338,
      "old_data_type":     "double",
      "new_data_type":     "double",
      "change_type":       "Modified",
      "source":            "cloud.command-handler.sensor-data-update",
      "actor":             null,
      "organization_ids_at_write_time": [],
      "ingestion_time":    "2026-05-25T11:28:35.025Z",
      "request_id":        null
    }
  ],
  "next_cursor": "eyJkZXZpY2VfaWQiOnsiUyI6Ijc3ODU5YjBkLi4ifS…"
}
```

Field	Meaning
`event_id`	UUID identifying one logical correction event. Two audit rows can share an `event_id` (e.g. each meter recovery emits both a `status` row and a `meter.value.corrected` row atomically).
`audit_timestamp`	When the audit row was written. **This is what `from` / `to` filter on.**
`record_timestamp`	Unix epoch seconds — the timestamp of the underlying Timestream record that was mutated. Use this to align the correction with the reading it changed.
`field`	Which field changed. For a meter-value-only changelog, filter `field=meter.value.corrected`.
`old_value`, `new_value`	The values before and after the change, in the **raw** unit (same as `meter.value.processed` / `meter.value.corrected` in Part 2). Multiply by `device.metertype.multiplier` to convert to display units — see §2.5.
`change_type`	`Added`, `Modified`, or `Removed`.
`source`	Free-form emitter identifier. Common values: `cloud.command-handler.sensor-data-update` (internal recovery / meter-filter), `cloud.platform.api.telemetry.bulk_update_measures` (a dashboard `PATCH /api/telemetry/measures`).
`actor`	The authenticated user that triggered the change, or `null` when emitted by a server-side originator. Populated for dashboard PATCH edits, `null` for cmd-handler-driven corrections.
`organization_ids_at_write_time`	The device's org membership snapshot at the moment the audit row was written. **Currently always `[]` for rows produced by `cloud.command-handler.sensor-data-update`** (a known producer-side gap). Populated for dashboard PATCH edits. Don't rely on this field for access scoping.

3.4 Why two rows share an `event_id`

A correction often updates more than one field atomically. The most common pattern for meter recoveries is a single event that emits two rows:

- `field = "status"`, `old_value = "Failed read"`, `new_value = "Ok"`
- `field = "meter.value.corrected"`, `old_value = N`, `new_value = M`

Both carry the same `event_id`. If you only care about value changes, filter `field=meter.value.corrected` and you'll see one row per event — the corrections.

3.5 "What's changed since my last call?" — polling pattern

This is the recommended polling loop. It assumes you store `last_to` between runs (e.g. in a database or a settings file).

```
# 1. On the first run, pick a start point — e.g. "everything from the past
#    week" or a specific date your last import covers.
last_to='2026-05-18T00:00:00Z'
```

```
# 2. On every run:
now=$(date -u +"%Y-%m-%dT%H:%M:%SZ")

cursor=""
while :; do
  url="$BASE/api/devices/$DEV/data-audit?
from=$last_to&to=$now&field=meter.value.corrected&limit=200"
  [ -n "$cursor" ] && url="$url&cursor=$(printf %s "$cursor" | jq -sRr @uri)"

  resp=$(curl -s -H "Authorization: Bearer $TOKEN" "$url")
  echo "$resp" | jq -c '.items[]' >> changelog.ndjson

  cursor=$(echo "$resp" | jq -r '.next_cursor // empty')
  [ -z "$cursor" ] && break
done

# 3. Persist the new high-water mark for the next run.
last_to="$now"
```

A few things worth knowing about this loop:

- **Idempotency.** `event_id` is unique per correction. If you de-duplicate by `event_id` on your side, an occasional re-run with overlapping `from`/`to` is harmless.
- **No data is silently dropped on the single-device endpoint.** The presence/absence of `next_cursor` is the authoritative "is there more?" signal — keep following it until it's absent. **Don't** assume `items.length == limit` means "this was the last page."
- **DDB `Limit` is applied before the filter.** A request with `limit=200&field=meter.value.corrected` against a device with mixed fields can return fewer than 200 items per page while still having more matches on subsequent pages. This is correct behaviour — keep paging until `next_cursor` is absent. (Concretely: with `limit=10` against a device whose audit log alternates `status` and `meter.value.corrected`, you'll typically see ~5 matched rows per page.)
- **Range cap.** A single request can cover at most 90 days. If `now − last_to > 90 days`, narrow the window — page through several 90-day chunks.
- **24 h token TTL.** If your loop runs longer than the 24 h token TTL, re-issue `POST /api/auth/login` and retry.

Worked example

For one device over a recent week with `field=meter.value.corrected`, a typical correction stream looks like this (one row per event):

```
audit_timestamp        record_timestamp (UTC)   old → new (raw)   Δ
2026-05-25T11:30:01Z   2026-05-25T09:28:44Z     184337 → 184338   +1
2026-05-25T11:30:01Z   2026-05-25T06:28:46Z     184333 → 184337   +4
2026-05-25T11:30:01Z   2026-05-25T00:28:42Z     184330 → 184333   +3
2026-05-25T11:30:00Z   2026-05-24T20:28:43Z     184215 → 184330   +115
2026-05-25T11:30:00Z   2026-05-24T19:28:40Z     184105 → 184215   +110
…
```

`audit_timestamp` clusters around the daily batch run on the cmd-handler side; `record_timestamp` points at the underlying reading that was re-resolved. Multiply each value by `device.metertype.multiplier` (Part 2 §2.5) to get the display value.

3.6 Permissions

Reading the audit log requires `telemetry:read` **or** `device_telemetry:read`, plus device-level access. The same permissions you already use for the telemetry endpoints in Part 2 are sufficient.

3.7 Errors

HTTP	When
400	Missing or invalid `from` / `to`; `from > to`; range > 90 days. Body contains a `code` like `AUDIT_INVALID_RANGE` or `AUDIT_RANGE_TOO_WIDE`.
401	Token missing or expired. Re-login via `POST /api/auth/login`.
403	Missing `telemetry:read` / `device_telemetry:read`, or the device id is unknown / belongs to another org.
404	The device id doesn't exist in your accessible scope.
504	Underlying query timeout — narrow the time window.