Manual 08 · every node, what it does, how to use it

Node Reference

One entry per node of the palette, grouped as in the studio — Input, Transform, Output, Flow control and DevOps — with its ports, how it executes, its main settings, the pitfalls worth knowing and an example.

CONTENTS

01 How to read this reference
- 1.1 Settings common to every node
- 1.2 Choosing the right node

02 Input nodes
- 2.1 DB Source
- 2.2 File Input
- 2.3 HTTP Source
- 2.4 FTP/SFTP Source
- 2.5 LDAP Source
- 2.6 GitHub Source
- 2.7 Dir Watcher
- 2.8 ActiveMQ
- 2.9 MQTT
- 2.10 Webhook Receiver
- 2.11 Watchdog
- 2.12 Bridge In

03 Transform nodes
- 3.1 Log
- 3.2 Data Quality
- 3.3 Union
- 3.4 Filter
- 3.5 Transform fields
- 3.6 Join
- 3.7 TMap
- 3.8 Aggregate
- 3.9 JSON Parser
- 3.10 XML Parser
- 3.11 Script
- 3.12 Window
- 3.13 Materialize
- 3.14 Explode
- 3.15 Report Generator
- 3.16 Pivot / Unpivot
- 3.17 LDAP Auth

04 Output nodes
- 4.1 JSON Serializer
- 4.2 XML Serializer
- 4.3 DB Sink
- 4.4 File Output
- 4.5 ActiveMQ Sink
- 4.6 MQTT Sink
- 4.7 FTP/SFTP Sink
- 4.8 Mail Sink
- 4.9 Webhook Responder
- 4.10 Bridge Out

05 Flow control
- 5.1 Stop
- 5.2 Start
- 5.3 End
- 5.4 Error Handler

06 DevOps
- 6.1 Shell
- 6.2 SSH

Copyright © 2025–2026 Marco Dongu. All rights reserved.

01 How to read this reference

Each entry starts with a line like this one:

`source_file` in: optional trigger / parameters row out: `output` row by row (streaming)

Part	Meaning
`source_file`	the node's type, as it appears in artifacts, logs and the runtime profile's lists
in	its input ports. Sources accept an optional input: a row that starts them or brings parameters (a file name, a date for the query)
out	its output ports: `output` for good rows, `reject` for refused rows with the reason, others named after their role. *(optional)* ports appear only with certain settings
execution	*row by row*: streams, memory stays flat · *stream*: events as they arrive, may run until stopped · *holds its whole input*: needs memory for all its rows · *keeps one side in memory*: the join's right side

Settings common to every node

Every node has, besides its own panel:

Tab	Settings
General	display name (used in logs and the monitor), short label, description, notes, **Enabled**
Advanced	**timeout** (0 = none; when it expires the node fails like for any other error), **On error** — *Error handler*, *Catch on node*, *Retry, then error handler*, *Retry, then catch on node* — with retry count and delay

Tip. Every node that reads or writes a system has a **Test connection** on its resource and, for sources, a **Preview** on real data. Use them before the first run: they use the same engine code as the run.

Choosing the right node

You want to...	Use
compute or rename fields of each row	**Transform** (one expression per field), **Script** (several statements)
keep, drop or route rows by condition	**Filter**; **Script** with `skip` / `reject` for complex logic
look up data in another source	**TMap** (several lookups, several outputs), **Join** (two flows, SQL-like types)
total, count, average by group	**Aggregate**
rank, running totals, previous/next row	**Window**
turn rows into columns or back	**Pivot / Unpivot**
one row into many	**Explode**, **JSON/XML Parser** with arrays, **Script** with `emit`
check data quality and score it	**Data Quality**; `expect` rules in a **Script**
reuse a dataset in several nodes of the lane	**Materialize**

You want to...	Use
pass rows to another lane	**Bridge Out → Bridge In**
stop on purpose	**Stop**

02 Input nodes

DB Source

`source_db` in: optional trigger / parameters row out: `output` row by row (streaming)

Reads rows from a database table.

Reads rows from a database through a **DB resource** of the lane: PostgreSQL, MySQL/MariaDB and SQLite with built-in drivers; Oracle, Informix and any ODBC source through the system's ODBC drivers.

- **Query mode**: a table (with optional schema, order, limit and offset) or your own SQL. In your SQL, `${field}` takes the value of that field from the **incoming row** — connect upstream the node that computes it. Values travel as query **parameters**, never pasted into the SQL text.
- **Detect schema** reads the column names and types; **Preview** shows real rows.
- **Query timeout** is the maximum wait for the **first** row; after that, rows stream without a limit. For a limit on the server, set `statement_timeout` (PostgreSQL) or `max_statement_time` (MariaDB) on the database user.
- **Filter pushdown**: when the node feeds only a Filter (reject not connected), the Filter's conditions are added to the query as a `WHERE`; the result does not change, the database sends fewer rows (*FPEL Language Reference*, chapter 7).

Query mode: SQL — sql

```sql
SELECT order_id, customer_id, amount, region
FROM sales.orders
WHERE order_date = ${batch_date}          -- field of the incoming row, sent as a parameter
```

File Input

`source_file` in: optional trigger / parameters row out: `output` row by row (streaming)

Reads records from a local file.

Reads a local file — or a mounted network folder — in CSV, TSV, Excel (`xlsx`, `xlsm`, `xlsb`, `xls`, `ods`), JSON, JSON Lines, XML, plain text or binary.

- **Where the file comes from**: a fixed path or a **glob** (`/data/in/*.csv` reads every match, in name order), a lane variable, or a field of the incoming rows — one file per row, in order, so a *Dir Watcher* can feed it directly; no row means no file to read (a warning, not an error). A path may contain `${NAME}` of a shared or lane variable: `${IN_DIR}/orders.csv`.
- **CSV/TSV**: delimiter, quote, escape, comment character, header yes/no, rows to skip, encoding (UTF-8, UTF-16, ISO-8859-1, Windows-1252, ASCII). Quoted fields may contain line breaks; a UTF-8 BOM is removed.

- **Mapping**: each output field has a name, the source column, a type and a text transformation. With a mapping only the mapped fields go out; without one, every column goes out as text. A value that cannot be converted to its type becomes null — a visible null rather than a silent zero.
- **Duplicate column names** are renamed `name_2`, `name_3`…; **short lines** get the missing fields as null, with a warning.
- **Excel** is read whole into memory before the first row goes out: for very large sheets, export to CSV.

Example. `customers_*.csv`, separator `;`, encoding Windows-1252, mapping `id → integer`, `birth_date → date`, `email → string (lower)`.

HTTP Source

`source_http` in: optional trigger / parameters row out: `output`, `reject` row by row (streaming)

Retrieves data from an HTTP endpoint.

Calls an HTTP or HTTPS endpoint and turns the answer into rows.

- **Method, URL, headers, query parameters and body**; values can come from the incoming row or variables.
- **Authentication**: none, API key, Basic, Bearer, Digest, OAuth2 client credentials, OAuth2 authorization code. With OAuth2 the token is renewed once on a 401 and the request repeated.
- **Response**: JSON with fields extracted by path, the raw JSON as an object, CSV text, or binary as base64.
- **Pagination**: by page number, offset, cursor or `Link` header, up to a maximum number of pages.
- Failed requests go to `reject` with the status and the message.

Example. `GET https://api.example.com/v2/orders?since={last_run}`, Bearer `${API_TOKEN}`, records at `$.data`, cursor pagination on `$.next_cursor`.

FTP/SFTP Source

`source_ftp` in: optional trigger / parameters row out: `output` row by row (streaming)

Reads files from FTP, FTPS or SFTP servers.

Downloads files from FTP, FTPS or SFTP and reads them with the same reader as the File Input (same formats, encodings and mapping). Each row also carries `_filename`, `_filepath`, `_filesize`, `_modified_at`.

- **Protocols**: `ftp` (plain text: only on trusted networks), `ftps` (explicit TLS before login, protected data channel, certificate verified; implicit FTPS on port 990 is not supported), `sftp` (SSH, password or key; the server's host key is checked against `known_hosts`).
- **Search**: name filter, subfolders, order, maximum files per run; GZIP and ZIP archives.
- **Wait for files** polls until at least one file matches. **Continuous** keeps listening: each group of ready files (unchanged between two checks) restarts the lane, and processed files are remembered in a state file, so a file is read once.
- **After reading**: *Move* to an archive folder or *Delete* — only for files read without errors.

LDAP Source

`ldap_source` in: optional trigger / parameters row out: `output` row by row (streaming)

Queries an LDAP directory and produces one row per entry.

Searches an LDAP directory and emits one row per entry.

- **Resource**: host, port, TLS mode (`ldaps`, `starttls` or `plain`), certificate verification, bind DN and password of a service account.
- **Base DN**, **scope** (base, one level, subtree), **LDAP filter**, **attributes** to return (comma-separated), page size for large directories.
- **Multiple values**: an attribute with several values (such as `memberOf`) as a list, or joined into text.

Example. Base `ou=people,dc=example,dc=com`, scope subtree, filter `(&(objectClass=person)(department=Sales))`, attributes `uid,cn,mail,memberOf`.

GitHub Source

`github_source` in: optional trigger / parameters row out: `output` row by row (streaming)

Fetches data from GitHub (repos, issues/PRs, commits) via REST API.

Reads from the GitHub REST API — read only.

- **Entity**: *Repos* of an organisation or user, *Issues + Pull Requests* of a repository (state open/closed/all, with or without PRs), *Commits* of a repository (optional branch).
- **Mode**: *From configuration* (owner and repo written in the panel) or *Per-row* (owner and repo from fields of the incoming rows — for example the list produced by a *Repos* node).
- Items per page and a maximum number of items; authentication through the GitHub resource's token (`${GITHUB_TOKEN}`).

Dir Watcher

`dir_watcher` in: optional trigger / parameters row out: `output`, `reject` stream — events as they arrive

Watches a directory for new files (watch) or enumerates its contents (scan).

Emits one row per file of a **local** folder.

- **Scan** lists the files present — with pattern, subfolders, minimum size, maximum age, order and limit.
- **Watch** listens to the operating system's events (new, modified, renamed, deleted). *Once* waits for the first group of events and ends; *Continuous* keeps listening until stopped, and each group of events restarts the lane — only if the group contains at least one event the node would emit (pattern and event types). With *Continuous*, choose *New files only* unless you really need updates: copying a file produces several events. On mounted network folders events often do not arrive: use Scan on a schedule instead.
- **File stability**: a file is emitted only when its size and date stay unchanged for the set time — never while it is still being written.
- **De-duplication**: by path, by path and modification date, or by content (SHA-256). With a state file it survives between runs, so a daily Scan processes only new files. A file counts as processed only if the nodes downstream finished without errors.

Example. Scan `/data/inbox`, pattern `invoice_*.xml`, de-duplication *by content* with a state file: run it every 10 minutes from the scheduler and each invoice is processed once.

ActiveMQ

`source_activemq` in: optional trigger / parameters row out: `output`, `reject` stream — events as they arrive

ActiveMQ consumer — receives messages from queues or topics over STOMP.

Consumes messages from ActiveMQ (Classic or Artemis) over **STOMP** (port 61613). OpenWire and AMQP are not supported: a node saved with them stops with a clear error.

- Queue or topic; for topics a **durable subscription** keeps the messages that arrive while the flow is stopped.
- Prefetch, receive timeout (0 = no limit), maximum messages (0 = no limit).
- Each message becomes a row with its headers and body; TLS from the resource.

MQTT

`source_mqtt` in: optional trigger / parameters row out: `output` stream — events as they arrive

MQTT subscriber — receives messages from MQTT/MQTTS brokers.

Subscribes to MQTT topics (MQTT 3.1.1) and emits one row per message.

- Broker, port, client id, user and password from the resource; **TLS** on port 8883, with the broker's certificate verified.
- Topic filter with wildcards (`sensors/+/temperature`, `plant/#`), QoS, payload format (JSON fields or raw text), listening time and queue size.

Webhook Receiver

`webhook_receiver` in: optional trigger / parameters row out: `output` stream — events as they arrive

Receives webhooks — shared server with HMAC, buffering and dedup.

Receives HTTP calls on a port of the run machine and emits one row per call: `event_id`, `event_type`, `source_ip`, `webhook_path`, `payload`, `headers`, `received_at`, `signature_valid`, plus the fields of a JSON body.

- Several receivers can share a port with different paths.
- **HMAC** (SHA-256 or SHA-1) with a shared secret: a request with a missing or wrong signature is answered **401** and never becomes a row; accepted rows carry `signature_valid = true`. Without a secret, `signature_valid` is null.
- **IP allowlist**, de-duplication of repeated events, a buffer with a policy when full: drop the oldest, drop the newest, or refuse with **503** so the sender retries.
- Plain HTTP: in production put it behind a reverse proxy with TLS.

Example. Path `/hooks/github`, secret `${GITHUB_HOOK_SECRET}`, header `X-Hub-Signature-256`, then Filter `event_type == "push"`: forged calls never reach the Filter.

Watchdog

`watchdog` in: optional trigger / parameters row out: `output` stream — events as they arrive

Monitors services via HEAD — unblocks the flow on the expected header.

Checks a service with HTTP `HEAD` (or `GET`) requests and lets the flow continue when the expected header is present.

- **Gate**: waits until the condition is true, emits once and ends — "start loading when the upstream service says it is ready".
- **Stream**: emits on every positive check and stays active. **Edge**: emits only when the state changes.
- Interval, timeout, authentication (Bearer, Basic, API key header) and the header to expect.

Bridge In

`bridge_in` in: none out: `output` stream — events as they arrive

Entry port from another lane — receives from the bridge channel.

Receives the rows that the Bridge Out it is linked to publishes from another lane. The link is chosen in the panel, from the list of Bridge Out of the other lanes (*Studio User Guide*, chapter 5).

- **Delivery**: *As it arrives* (default) passes rows on immediately; *All at once* holds them until the sending lane has delivered everything, then passes them — if the sender stops halfway, nothing passes.
- **Wait timeout**: maximum wait for the first row; `0` = no limit.

---

03 Transform nodes

Log

`log` in: `input` out: `output` row by row (streaming)

Transparent node — logs the rows passing through for debugging. Does not modify the data.

Writes the rows that pass through to a log, without changing them — for debugging.

- **Destination**: the studio's log panel, a dedicated Log Viewer window, or both.
- **Level** (debug, info, warn, error), message prefix and **template** with `{field}` placeholders.
- **Sampling**: the first N rows, one every N, or a percentage; a character limit per row.

> **Warning.** Log rows contain real data. In artifacts for production keep Log nodes disabled, or let the artifact's log level filter them (*Build & Publish*, chapter 4).

Data Quality

`data_quality` in: `input` out: `output` row by row (streaming)

Checks each row against rules, repairs where a rule says so and adds a quality field (_dq) — it does not filter.

Checks each row against rules, repairs where a rule says so, and adds a field (default `_dq`). It does **not** filter: every row goes out; separate them downstream with a Filter on `_dq.valid`, `_dq.level` or `_dq.score`.

- `_dq` contains `score` (0–1), `valid`, `level` (`ok`, `warn`, `error`), `repaired`, the score per dimension and the list of `issues` (rule, field, severity, message, original and new value).

- Checks: not null, not empty, minimum/maximum length, range, pattern, is numeric / date / e-mail / URL, in list / not in list, comparison between fields, referential (lookup), FPEL expression. Repairs: set a default, set null or empty text, copy from another field or from the previous row, concatenate fields, lookup in a Materialize dataset, FPEL expression.
- The score of a dimension is passed rules / rules of that dimension; the total is the weighted mean of the dimensions that have rules.

Example. Rule *email matches pattern* `^[^@]+@[^@]+\.[a-z]{2,}$` (validity, error); rule *country in* `IT, FR, DE` with repair *lookup* in the Materialize `countries`; then Filter `_dq.level ≠ "error"`.

Union

`union` in: `input_main` · several inputs out: `output` stream — events as they arrive

Merges N flows into one — concat, interleave or zip modes.

Merges several flows into one; the **Mapping** tab decides, for each output field, which field of each flow feeds it.

- *Concat*: the first flow to the end, then the second, and so on. *Mix*: rows in arrival order. *Zip*: one row per flow merged into one (for flows of different lengths: truncate, fill with null or error).
- A field missing in a flow: null, omitted, or error before reading. *Source field* adds the name of the originating flow.

Filter

`filter` in: `input` out: one per route / stream you define row by row (streaming)

Routes rows to N outputs based on conditions — first-match.

Routes each row to the **first** route whose condition is true; rows matching none go to `reject`.

- Conditions are written in three ways: **Visual** (field, operator, value joined by AND/OR), **Template** (ready-made cases for dates, numbers, text, null) or **Code** (an FPEL condition). Visual and template show their FPEL equivalent.
- **Case sensitive** applies to visual and template; in Code use `lower()`.
- **Branch execution**: *All in parallel* (default) or *Ordered with wait* — route #2 starts only when everything downstream of route #1 has finished. Typical use: master rows before detail rows, for foreign keys.

Route	Condition
#1 `domestic`	`country == "IT"`
#2 `eu`	`country in ["FR", "DE", "ES", "AT"]`
`reject`	everything else

Transform fields

`transform` in: `input` out: `output` row by row (streaming)

Transforms, renames and converts the fields of each row with FPEL expressions.

Computes, renames and converts the fields of each row, one FPEL expression per output field; fields you do not touch pass unchanged.

- Each field starts from **pass unchanged**; choose a transformation from the list (with search) — templates are grouped by field type — or write a custom expression.
- **Preview** runs the node on sample rows with the real engine; click a computed value to see **how** it was computed.

Output field	Expression
`full_name`	`concat_ws(" ", first_name, last_name)`
`email`	`lower(trim(email))`
`amount_vat`	`round(amount * 1.22, 2)`
`segment`	`case when amount > 10000 then "key" when amount > 1000 then "mid" else "small" end`

Join

`join` in: `input_left`, `input_right` · several inputs out: `output`, `reject` keeps one side in memory

Joins two flows on a key field.

Joins the left flow (streamed) with a right side kept in memory by key (hash join).

- **Right side**: the flow on the right input, or the dataset of a Materialize of the lane.
- **Types**: INNER, LEFT, RIGHT, FULL, CROSS, ANTI (left rows without a match), SEMI (left rows with a match, only their fields) and CUSTOM. With INNER, SEMI and CUSTOM, unmatched left rows go to `reject`.
- **CUSTOM** takes any FPEL condition on `left.field` and `right.field` — compares every left row with every right row, so keep the right side small.
- Composite keys, case sensitivity, a prefix for right fields with the same name, multiple matches (all, first, last, error), null keys (excluded, equal, error).

Example — price bands. CUSTOM `left.price ⩾ right.min_price && left.price ⩽ right.max_price`, right side = a small table of bands. Type numeric fields at the source: two texts compare as text (`"15" ⩽ "9"` is true).

TMap

`tmap` in: `input_main` · several inputs out: one per route / stream you define row by row (streaming)

Visual multi-input/output transformer with mapping, lookup joins and conditional routing.

The visual mapper: one **main** input read row by row, any number of **lookup** inputs kept in memory by key, and any number of **outputs**, each with its own fields and filter.

- Lookups are resolved in dependency order (a lookup can join on another one). *inner*: rows without a match go to the **rejected** output; *left*: the lookup's fields are null; *first*: only the first match.
- **Join keys**: a field with an optional function (trim, upper, year, substring, regex extract, FPEL expression); composite keys.
- **Expressions** use qualified fields: `Orders.id`, `"Customer data".name`. Renaming an input updates the expressions.
- **Output types** are applied: `"12"` on an integer field goes out as 12; a value that cannot be converted stays as it is and is counted in the log.

- **Output filters** are independent: a row can go to several outputs.
- **Transforms** are computed once per *combination* (a main row with three lookup matches makes three), before the output filters, and shared by all outputs. The button *per combination / per main row* in the transform header computes it once per main row instead — useful with `next()` for one number per order. `next()` written in an output field advances only for the rows that output emits.

Example. Main *Orders*, lookup *Customers* on `Orders.customer_id = Customers.id` (left), lookup *Countries* on `Customers.country = Countries.code` (inner). Output *invoices* with filter `Orders.amount > 0`, output *anomalies* with filter `Customers.id is null`.

Aggregate

`aggregate` in: `input` out: `output` holds its whole input in memory

Groups rows and computes aggregate functions.

Groups rows and computes aggregate functions; it holds its whole input in memory.

- **Group by** one or more fields; **nulls** in the grouping fields: excluded, or a separate group.
- **Functions**: `count`, `count_distinct`, `sum`, `avg`, `min`, `max`, `median`, `std_dev`, `variance`, `first`, `last`, `string_agg`, `array_agg`, `json_agg`, each with an optional alias.
- **HAVING** filters the groups after aggregation; **Order by** sorts the result on several fields, each with its direction (`orders DESC, region ASC`); a **limit** keeps the top N.
- The input can be the connected flow or a Materialize dataset.

Functions — json

```
{ "count": "*", "sum": "amount", "avg": "amount", "max": "order_date" }
```

Gives `count_all`, `sum_amount`, `avg_amount`, `max_order_date` per group — or the aliases you choose.

JSON Parser

`json_parser` in: `input` out: one per route / stream you define row by row (streaming)

Extracts and transforms JSON data into structured flows.

Reads a JSON field (text or already an object) and produces one output per **stream**: each stream has a JSONPath (for example `$.lines`) and its fields.

- With *Array* on, one row per element; otherwise only the first element. Fields are looked up in the element first, then in the whole document — a lines stream can also take `$.customer` of the header.
- Field **types** are applied (`"12"` on an integer field becomes 12). A missing field: null, a default, skip the element, or reject it.
- **Parent fields** adds chosen fields of the incoming row to each output row — the order lines carry the order's `order_id`.
- Unreadable JSON: to `reject` (`invalid_json`), or dropped and counted.

input field "document" json

```
{ "order": { "id": "A-17", "customer": "C-3" },
  "lines": [ { "sku": "X1", "qty": 2 }, { "sku": "Y9", "qty": 1 } ] }
```

Stream *lines* at `$.lines`, fields `sku`, `qty`, `order_id ← $.order.id`: two rows, `X1/2/A-17` and `Y9/1/A-17`.

XML Parser

`xml_parser` in: `input` out: one per route / stream you define row by row (streaming)

Extracts and transforms XML data into structured flows.

Reads an XML field and produces one output per **stream** (a base path). With *Repeating* on, one row per element found.

- Fields are resolved relative to the element (`qty`, `@id`, `text()`), or with an absolute path: if it is not under the stream, the element's ancestor is used (implicit master-detail).
- Navigation is by tag names, not full XPath. The `@` button turns the last step into an attribute.
- *Apply field types* converts values to the field types; unreadable XML goes to `reject` (`invalid_xml`) if enabled.

Script

`script` in: `input` (optional) out: `output` (optional), `output` (optional), `reject` (optional) row by row (streaming)

Transforms, filters or drops each row with instructions and expressions.

Runs FPEL statements on each row — or generates rows from nothing. See the *FPEL Language Reference*, chapter 4, for the statements and many examples.

- **Flow**: one input; the row goes out at the end of the body. **Generate**: no input; rows go out only with `emit`.
- Ports: `output`, and `reject` when enabled (used by `reject` and `expect … else reject`).
- `expect` rules are counted per node and shown in the studio and in the monitor.

Window

`window` in: `input` out: `output` holds its whole input in memory

Computes window functions (ROW_NUMBER, RANK, LAG, LEAD, CUMSUM, MOVING_AVG...).

Computes window functions over partitions of the data; it holds its whole input in memory.

- **Partition by** one or more fields; **Order by** within each partition, several fields with their directions (`order_date ASC, id DESC`).
- **Functions**: ranking (`row_number`, `rank`, `dense_rank`, `percent_rank`, `cume_dist`, `ntile`, `topn_flag`), navigation (`lag`, `lead`, `first_value`, `last_value`, `nth_value`), cumulative (`cumsum`, `cumcount`, `cumprod`), moving (`moving_avg`, `moving_sum`, `moving_min`, `moving_max`, `moving_stddev` over N rows), analytical (`ratio_to_report`, `delta`, `change_detect`, `sessionize`, `streak`, `interpolate`).

Example. Partition by `customer_id`, order by `order_date ASC`: `row_number` → `order_seq` (1 = first order), `lag(amount)` → `previous_amount`, `cumsum(amount)` → `lifetime_value`.

Materialize

`materialize` in: `input` out: `output`, `reject` holds its whole input in memory

Keeps a flow in memory; published in the lane, other nodes read it by name (Explode, Join, Window, Aggregate, Pivot, Data Quality).

Keeps a flow in memory under a name; in the same lane, Explode, Join, Window, Aggregate, Pivot and Data Quality read it by name — without an arrow and without consuming it.

- *Passthrough*: rows continue downstream while being stored. *Buffer → Signal*: stores everything, publishes the dataset, then emits one status row (`name`, `row_count`, `status`, `completed_at`, `elapsed_ms`, `truncated`) that triggers the next nodes.
- **Key field** indexes the dataset (Data Quality uses it for lookups); on repeated keys keep all, the first, the last, or fail.
- **Row limit**: beyond it, fail or truncate (extra rows to `reject`).
- The dataset lives for one execution of the lane.

Explode

`explode` in: `input` out: `output`, `reject` holds its whole input in memory

Turns dense structures (a Materialize dataset, an array or object field) into a flow of rows.

From one to many: a dataset or a collection becomes a flow of rows.

- *From Materialize*: emits a lane dataset row by row (the input, if connected, is only a start signal).
- *From flow field*: for each incoming row, explodes an array or object field (also a string holding JSON). *Propagate parent fields* copies the parent's fields to each child.
- **Mapping** chooses the output fields, names, types and text transformations.
- Empty or null structure: skip (parent to `reject`), emit one row with nulls, or error.

Example. A row with `tags = ["red","large"]` exploded on `tags` with parent fields → two rows, each with the order fields and `tag = "red"` / `"large"`.

Report Generator

`report_generator` in: `input` out: `output`, `reject` holds its whole input in memory

Buffers the flow and generates a PDF/HTML/Excel report with tables and charts.

Collects the flow and produces a report in **PDF**, **HTML** or **Excel**, with a title, tables and charts; the document goes out in a field (base64 for binary formats) to be written by a File Output, an FTP Sink or attached by a Mail node.

- Template and colour theme, language (English or Italian), report title and subtitle.
- **Columns** with formats (text, number, currency, date) and conditional styles on cells or whole rows.
- **Charts and KPIs**: category field (X axis), value field (Y axis), aggregation (sum, average, count).

Example. Monthly sales by region: columns `region`, `orders`, `revenue (currency)`; bar chart X = `region`, Y = `revenue`, sum; a red style on rows where `revenue < target`. Then *File Output* → `/reports/sales_${date}.pdf`, content from the report field.

Pivot / Unpivot

`pivot` in: `input` out: `output` holds its whole input in memory

Reshapes the table structure — rows into columns (Pivot) or columns into rows (Unpivot).

Reshapes a table: **Pivot** turns rows into columns, **Unpivot** columns into rows.

- Pivot: identity fields (the rows of the result), the pivot field (its distinct values become columns) and the value field with an aggregation; nulls kept or replaced with 0; an optional total column. Columns are discovered from the data and ordered as you choose.
- Unpivot: the columns to turn into `key` / `value` pairs, with or without nulls.

region	month	revenue	→	region	Jan	Feb	Mar
North	Jan	100		North	100	120	90
North	Feb	120		South	80	70	95

LDAP Auth

`ldap_auth` in: `input` out: `output`, `reject` row by row (streaming)

Authenticates the credentials of each row against LDAP (search-then-bind).

Checks the credentials found in each row against LDAP (search, then bind as the user found): rows with valid credentials go to `output`, the others to `reject`.

- User field and password field of the row; login attribute (`uid`, `sAMAccountName`, `mail`…); base DN.
- Optional: an additional filter, a **required group**, attributes to return with the authenticated row.

Warning. Rows carry passwords. Keep the flow short — receive, authenticate, discard the password field — and never send it to a Log node or a file.

04 Output nodes

JSON Serializer

`json_serializer` in: `input` · several inputs out: `output` holds its whole input in memory

Serializes flow rows into JSON strings with a configurable structure.

Collects **all** the rows of its input flows and builds **one** JSON document following the tree you draw: objects, arrays (one element per row of the flow driving them, or per group) and leaves linked to fields. One row goes out with the document in a field (default `content`).

- Leaves can have an FPEL **expression** and a **condition** (false → the key is omitted).
- *Field types: applied* converts leaves to string, number or boolean.

XML Serializer

`xml_serializer` in: `input` · several inputs out: `output` holds its whole input in memory

Serializes flow rows into XML strings with configurable structure and namespaces.

Builds XML documents from the rows, following the tree you draw: elements, attributes, CDATA, namespaces; elements repeated per row or per group.

- Leaves and attributes can have an FPEL **expression** and a **condition**.
- **Check the output** verifies each document is well-formed (not an XSD validation); a bad document stops the node, goes to `reject`, is written empty or is skipped.

DB Sink

`sink_db` in: `input` out: `output`, `reject` row by row (streaming)

Writes rows into a database table.

Writes rows into a database table through a DB resource.

Write mode	Effect
INSERT	inserts new rows
UPSERT	inserts, or updates on key conflict
UPDATE	updates existing rows matched by the key columns
DELETE	deletes rows matched by the key columns
TRUNC+INS	empties the table, then inserts
MERGE	SQL `MERGE` with an `ON` condition (PostgreSQL 15+, Oracle)

- **Key columns** for UPSERT, UPDATE, DELETE and MERGE; *Load constraints* reads the primary key from the database.
- **Table options**: create if missing, or recreate (`DROP + CREATE` — dangerous, it asks).
- **Pre-write / post-write SQL**, batch size and flush interval, a **transaction** group (the lane's Transactions tab).
- **Query mode** instead of a write mode: a parametrised SQL statement per row (`{field}` placeholders become parameters) or a **stored procedure** per row or per batch (rows as JSON).
- On constraint errors: stop, or skip the row (to `reject` with the database's message). **Generated SQL preview** shows the statement.

Example. UPSERT into `dw.customers`, key `customer_id`, batch 1 000, pre-write SQL `SET search_path TO dw`, in the native transaction group *load* with the order tables.

File Output

`sink_file` in: `input` out: `output`, `reject` row by row (streaming)

Writes rows to a file.

Writes rows to a local file (or a mounted network folder).

- **Formats**: CSV, TSV, Excel `.xlsx`, JSON (array or one object per line), JSON Lines, XML; or the value of a field — HTML or text as text, Excel or binary from base64 (PDF, images: for example the output of a Report Generator).

- **Path variables**: `${date}`, `${datetime}`, `${uuid}`, `${seq}` (first free number), `${partition}`; any other `${NAME}` is a shared or lane variable (`${OUT_DIR}/report.csv`), and an unknown name is an error. Missing folders are created.
- **Mode**: overwrite, append (with header only on a new file), new file (`out_1.csv`, `out_2.csv`...), or fail if the file exists.
- **Columns** (Mapping tab): which fields, in which order, with name, type and text transformation. Without a mapping, fields are written in their input order.
- **Encoding**: UTF-8, UTF-16 LE with BOM, ISO-8859-1, Windows-1252, ASCII; characters that cannot be represented become `?` and are counted.
- **Partitions**: one file per value of a field, per date (year/month/day/hour) or per maximum size.
- **Output**: a status row per file written (`rows_written`, `bytes_written`, `file_path`...) or the rows themselves after writing. **After writing**: a shell command per file (`${file}`, variable `FP_FILE`) or a webhook.

Example. `${OUT_DIR}/sales/${partition}/sales_${date}.csv`, partition by field `region`, UTF-8, header on → `/data/out/sales/North/sales_2026-11-07.csv`, one file per region.

ActiveMQ Sink

`sink_activemq` in: `input` out: `output` row by row (streaming)

ActiveMQ producer — publishes messages to queues or topics.

Publishes each row as a message to an ActiveMQ queue or topic over STOMP. Each message is sent with a receipt: an error frame from the broker becomes the error of that row. Options: destination and its type, serialization of the row, persistent delivery, priority, time to live, correlation id from a field.

MQTT Sink

`sink_mqtt` in: `input` out: `output` row by row (streaming)

MQTT publisher — publishes messages to topics.

Publishes each row as a message to an MQTT topic — a fixed topic, or one taken from a field of the row. QoS, retain flag, serialization of the row (JSON or a field); TLS from the resource, with the broker's certificate verified.

Example. Topic from the field `alert_topic` (computed upstream as `f"plant/{line}/alerts"`), QoS 1, payload = the row as JSON.

FTP/SFTP Sink

`sink_ftp` in: `input` out: `output`, `reject` row by row (streaming)

Writes files to FTP, FTPS or SFTP servers.

Collects all rows and uploads one file in a single transfer when the input ends — FTP, FTPS (explicit TLS) or SFTP (host key checked against `known_hosts`).

- Remote folder and file name with `{date}`, `{datetime}`, `{timestamp}`, `{run_id}`.
- Mode: overwrite, append (CSV, TSV, JSON Lines or field value), new file (`_1`, `_2`...), or fail if it exists.

- Content: rows in CSV, TSV, JSON, JSON Lines, XML, Excel — or the value of a field (text or base64). GZIP or ZIP compression; **atomic** upload (`name.tmp` , then rename); permissions (`644`).

Mail Sink

`mail_sink` in: `input` out: `output` , `reject` row by row (streaming)

Sends email via SMTP, SendGrid, Amazon SES or Mailgun.

Sends e-mail through **SMTP**, **SendGrid**, **Amazon SES** or **Mailgun** — one message per row, or one message summarising the flow.

- From, to, cc, bcc, subject and body (text or HTML) with `{field}` placeholders; attachments from base64 fields (the output of a Report Generator).
- SMTP security: `STARTTLS` (587), `SSL/TLS` (465) or none; without a user the node does not try authentication (internal relays).
- Priority high or low adds the standard headers.

Example. Connected to the Error Handler's `error_out` : to `ops@example.com` , subject `Stilton: {_error_node_type} failed in {_error_lane_id}` , body `{_error_message} at {_error_at}` .

Webhook Responder

`webhook_responder` in: `input` out: none row by row (streaming)

Responds to HEAD/GET with synthetic headers from the current row.

Answers `HEAD` and/or `GET` requests on a webhook path with headers built from the **current row** — a lightweight status endpoint for other systems (for example a Watchdog in another flow).

- Methods: HEAD only, GET only, or both; with GET the body can return the headers as JSON.
- *Monitor* mode reads lane variables and stays active.

Bridge Out

`bridge_out` in: `input` out: `output` (optional), `output` (optional) row by row (streaming)

Exit port from the lane flow — publishes to the bridge channel.

Publishes the lane's flow on a channel; the Bridge In linked to it, in another lane, receives it. The channel name is a label: the link is a code kept by the studio (*Studio User Guide*, chapter 5). If the other lane is slower, this one waits.

- **Output to current lane**: none (default), *Passthrough* (the same rows also continue in this lane), *Signal* (one row `channel` , `rows_sent` , `status` , `sent_at` when the delivery is done).
- **Sync**: *Fire & Forget*, or *Wait for ack* (ends only when the other lane has received everything; status `delivered`).
- A failure here reaches the other lane only if the failing node is **critical**.

05 Flow control

Stop

`stop` in: `input` out: none row by row (streaming)

Flow control: deliberately stops the lane (rollback + closing connections) when the flow reaches this node. It is not a failure. Multi-instance — typically downstream of a reject or of a filter handle.

Ends the lane deliberately when rows reach it: open transactions are rolled back and connections closed. It is **not** a failure — the run reports it as a stop with your message.

- **Trigger**: *Immediate* (on the first row) or *After input* (when the branch is exhausted).

Example. After a Filter route `duplicate_batch` with condition `batch_id == var("LAST_BATCH")` : Stop with message *"batch already loaded"* — nothing is written twice.

Start

`lane_start` in: none out: `output` row by row (streaming)

Optional visual marker of where the lane begins. It has no effect on execution: all the nodes of a lane start together and only data connections order them.

An optional visual marker of where a lane begins. It does not change the execution: all nodes of a lane start together and only the arrows order them.

End

`lane_end` in: `input` out: none row by row (streaming)

Optional visual marker of where the lane ends. It has no effect on execution.

An optional visual marker of where a lane ends. It does not change the execution.

Error Handler

`error_handler` in: `catch` out: `error_out` row by row (streaming)

Central collector for lane errors — always active, cannot be deleted. Automatically receives every error not handled by catch/reject (and a copy of the handled ones, if "Centralized log" is active).

One per lane, created with it and not removable. It receives the errors that the lane's nodes delegate to it — no arrow needed — as rows of `_error_*` fields, and emits them on `error_out` according to its rules (*Studio User Guide*, chapter 6).

- Rules evaluated from the top; the first that matches decides: **Emit** (log and `error_out`), **Log only**, **Ignore**, **Stop the lane**.
- Per node: **Critical** (an error always stops the lane) and **Exclude from log**.
- An error caught on a node's `catch` port follows only that path and does not arrive here.

06 DevOps

Shell

`shell_exec` in: `input` out: `output` row by row (streaming)

Runs local bash/shell commands — output into the flow.

Runs a command on the machine that runs the flow — with `/bin/sh -c` on Linux, `cmd /C` on Windows — and turns its output into rows. Write the command for the shell of the machine that will run the artifact.

- Command (pipes, redirections and `&&` allowed), working directory, extra environment variables (JSON object), timeout.
- **Run for each row**, or once.
- **Output**: one row per output line, or the whole output in one field; stderr captured or not; exit code ≠ 0: stop or continue.

Warning. A Shell node runs whatever its command says, with the rights of the account that runs the flow. Production machines should deny `shell_exec` in their runtime profile unless they really need it (*Runtime Operations*, chapter 4).

SSH

`ssh_exec` in: `input` out: `output` row by row (streaming)

Runs commands on a remote host via SSH.

Runs a command on a remote host over SSH and turns its output into rows.

- Host, port, user; authentication by password, private key, or key with passphrase — from the SSH resource of the lane or from the node.
- The server's **host key** is always checked against `~/.ssh/known_hosts` of the account that runs the flow; an unknown or changed key is refused before any credential is sent.
- Command, run for each row or once, output as lines or a single field, stderr captured or not, exit code handling, timeouts.

Example. `df -h /data | tail -1` on the file server every morning, output one field, then a Script that extracts the percentage used and an `expect usage < 90 else warn` .