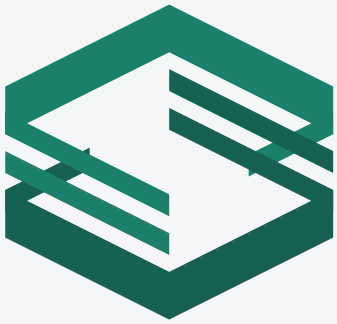
Manual 02 · design, connect, test

Studio User Guide

How a Stilton project is organised — pool, lanes, nodes and resources — how lanes talk to each other through bridges, how errors are routed, and the tools of the studio you use every day: environments, history, validation, previews, the log and the monitor panel.



CONTENTS

01 **The studio window**
- 1.1 Keyboard shortcuts

02 **Pool, lanes and nodes**
- 2.1 Working with lanes
- 2.2 Connecting nodes

03 **Configuring a node**
- 3.1 Example — a file source
- 3.2 Previews on real data
- 3.3 Expressions

04 **Resources: connections to the outside**
- 4.1 Example — a PostgreSQL resource
- 4.2 Transactions

05 **Bridges between lanes**
- 5.1 Bridge Out settings
- 5.2 Bridge In settings
- 5.3 When the sending lane fails

06 **Errors: reject, catch and the error handler**
- 6.1 The Error Handler
- 6.2 Stopping on purpose

07 **Variables, profiles and secrets**
- 7.1 Shared variables and profiles
- 7.2 Secrets

08 **Saving, versions and undo**
- 8.1 Projects from other versions of Stilton
- 8.2 Importing lanes from another project

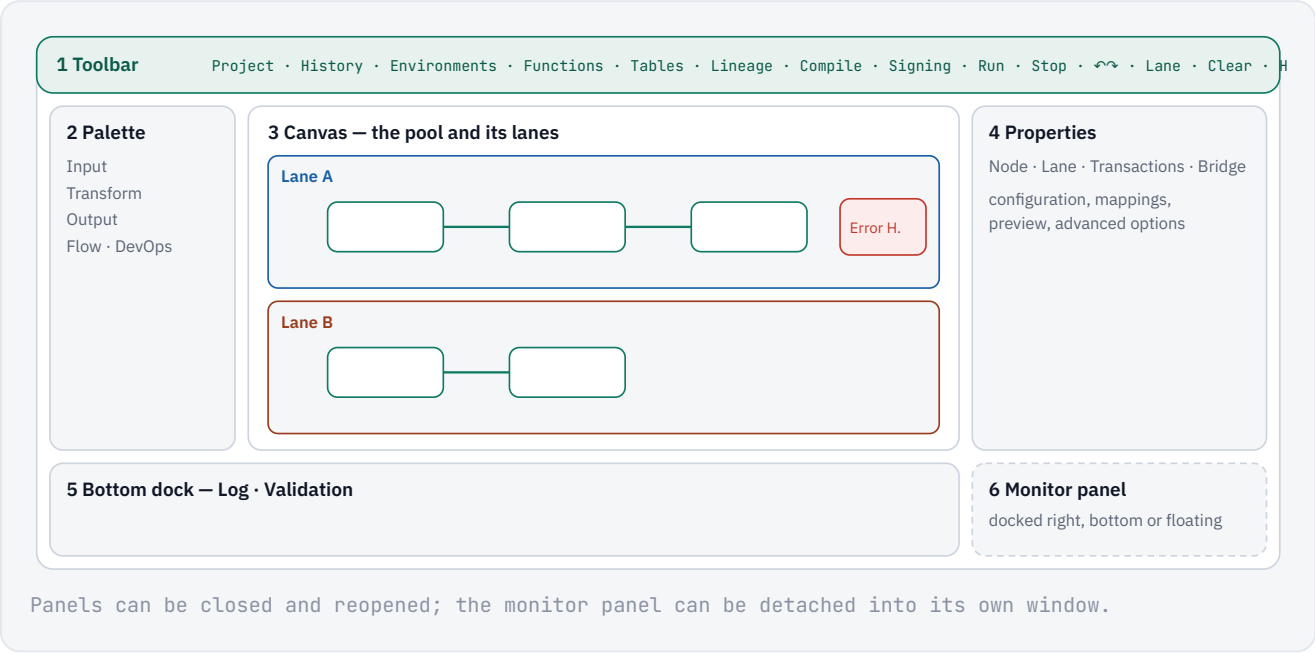
09 **Validation, log and monitor**
- 9.1 Field lineage and "explain this value"

10 **A complete example in two lanes**

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

01 The studio window

Everything happens in one window. The canvas is in the middle; around it, the tools that describe and test what is on it.



Panels can be closed and reopened; the monitor panel can be detached into its own window.

#	Area	What you do there
1	**Toolbar**	open and save projects, manage environments and versions, compile and sign, run and stop
2	**Palette**	drag a node type onto a lane: *Input*, *Transform*, *Output*, *Flow*, *DevOps*
3	**Canvas**	the pool with its lanes; nodes and the connections between them
4	**Properties**	settings of the selected node, of its lane (variables, resources), transactions and bridges
5	**Bottom dock**	*Log*: what the last run did. *Validation*: problems found while you design — click one to jump to the node
6	**Monitor panel**	timing and throughput per node, memory, timeline of the last run

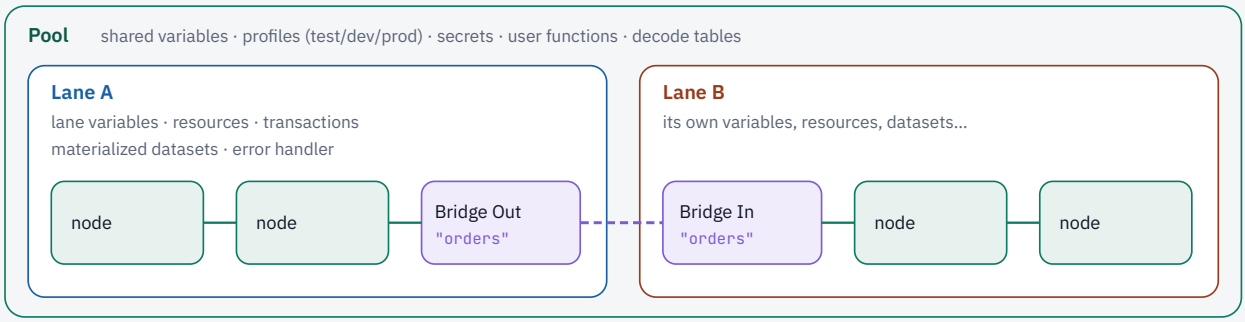
Keyboard shortcuts

Keys	Action
`Ctrl+N`	new project
`Ctrl+O`	open a project
`Ctrl+S` / `Ctrl+Shift+S`	save / save as
`Ctrl+Z`	undo
`Ctrl+Shift+Z` or `Ctrl+Y`	redo
`Delete`	remove the selected nodes or connections

On macOS the same shortcuts use `Cmd`.

02 Pool, lanes and nodes

A project contains one **pool**. The pool contains **lanes**; each lane contains **nodes** connected by arrows. The structure is not decoration: it decides what can see what.



Lanes are isolated: the only way rows cross from one lane to another is a bridge.

A lane is the unit of isolation. Resources (connections), transactions, lane variables and materialized datasets belong to one lane. Two lanes never share a database connection, a transaction or a dataset; they exchange rows only through bridges. This is also what lets a lane become a separate process in the future without redesigning the flow.

All lanes start together. When you press **Run**, every lane starts at the same time and runs in parallel. Inside a lane, every node starts too: the arrows decide the order, because a node waits for the rows of the node before it. The *Start* and *End* nodes of the *Flow* section are optional visual markers; they do not change the execution.

Each lane has its own Error Handler. It is created with the lane and cannot be deleted; it receives the errors of the lane's nodes without any arrow (see *Errors* below).

Working with lanes

- **Lane** in the toolbar adds a lane at the bottom of the pool. Rename it, change its colour, collapse it or change its height from the lane header.
- Select a lane to see its tab **Lane** in the properties: lane variables and resources.
- **Clear** removes all nodes and connections but keeps lanes, resources and variables — useful to redraw a flow on the same infrastructure. It asks for confirmation.
- **Project → Import lanes...** brings lanes of another project next to these (see *Importing lanes from another project*, below).

Connecting nodes

Drag from the small circle on the right of a node (an **output port**) to the circle on the left of another node (its **input port**). Some nodes have more than one output port:

Port	Carries	Example
`output`	the rows that went well	every source and transform
`reject`	rows the node refused, with the reason	a Filter row that matches no condition; a CSV line that cannot be parsed
named outputs	one route per condition or per output table	*Filter* routes, *TMap* outputs
`catch`	rows that caused an error, when the node handles its own errors	a DB Sink row that violates a constraint
`error_out`	errors collected by the lane's Error Handler	send them to a log file or an e-mail

The validation panel flags a connection that cannot work — for example a mandatory input left unconnected, or a field used downstream that no upstream node produces.

---

03 Configuring a node

Click a node to open its settings in the **Node** tab. Every node has three groups of settings:

- **General** — display name (shown on the canvas and in logs), short label, description, notes, and **Enabled** (a disabled node is skipped).
- **The node's own panel** — what the node does: the file to read, the query to run, the mapping to apply.
- **Advanced** — timeout, retry and what happens on error.

> **Tip.** Give every node a clear display name. Logs, the monitor and the example verifier identify nodes by name: `Read orders` is easier to find in a log than `source_db_3`.

Example — a file source

Setting	Value
Display name	`Read customers`
File	`data/customers.csv`
Format	CSV, separator `,`, first row is header
Schema	press **Detect** to read the column names from the file, then adjust the types

Relative paths are resolved from the **project's folder** in the studio (your home folder while the project has never been saved), and from the **current directory** for the runner. Keep data next to the project, run the runner from the project's folder, and the same paths work in both.

Previews on real data

Several panels have a **Preview** button: file and database sources read a sample of the real data; a Transform shows what its expressions produce on sample rows. The preview runs the real engine on that single node, so what you see is what the run will do — not a simulation.

Expressions

Computed fields, filter conditions and scripts are written in **FPEL**, Stilton's expression language. A few examples:

```text
concat(first_name, " ", last_name)
round(amount * 1.22, 2)
case when country == "IT" then "domestic" else "foreign" end
upper(trim(email)) ?? "unknown"
```

The function picker next to every expression field lists all functions with a description. The *FPEL Language Reference* manual covers the language in full.

04 Resources: connections to the outside

A **resource** describes a connection — a database, an HTTP API, an FTP server, an MQTT broker, an SSH server, an LDAP directory, GitHub, a webhook server. It belongs to a lane and is used by the nodes of that lane.

01 Select the lane and open its **Lane** tab, section *Resources*, and add one (*New DB*, *New API*, *New FTP*, *New MQTT*, ...).

02 Fill in the connection settings. Use references instead of literal values for anything that changes between environments or is secret: `${DB_HOST}`, `${DB_PASSWORD}`.

03 Press **Test connection**.

04 In the nodes, choose the resource from the list. A resource offers shortcuts — *Add as input*, *Add as output* — that create a node already linked to it.

Example — a PostgreSQL resource

Field	Value
Name	`warehouse`
Host	`${DB_HOST}`
Port	`5432`

Field	Value
Database	`sales`
User	`etl_reader`
Password	`${DB_PASSWORD}`

`DB_HOST` is a shared variable whose value changes with the profile (test, production); `DB_PASSWORD` is a secret, whose value is never saved in the project.

Transactions

The **Transactions** tab of a lane groups database outputs so that they commit or roll back together.

Mode	Members	Behaviour
Native	outputs on **one** database resource	one connection, autocommit off; COMMIT at the end of the lane, ROLLBACK if any member failed
XA	outputs on several **PostgreSQL** resources	two-phase commit across databases

Example. An order header goes to `orders` and its lines to `order_lines`, both on the `warehouse` resource. Put both DB Sinks in the same native group: if writing a line fails, the header is rolled back too, and no half-written order remains.

The rules that decide commit or rollback, what each database supports and how transactions meet catch ports, the error handler, the Stop node and bridges are in **Manual 10 — Transactions**.

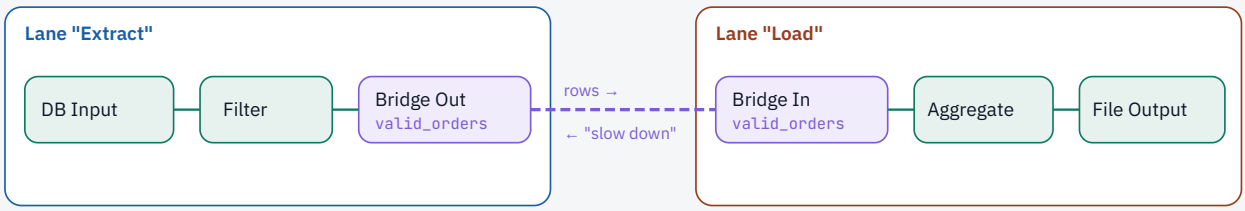
05 Bridges between lanes

A bridge moves rows from one lane to another. It has two halves linked to each other: a **Bridge Out** at the end of the sending flow and a **Bridge In** at the start of the receiving flow.

How they are linked. The link is a **channel code** that the studio creates and keeps; the channel *name* is only a label, set on the Bridge Out, which you can rename at any time without breaking anything.

- A new bridge links **by itself only when there is no doubt**: exactly one free bridge of the opposite type in the other lanes. Otherwise it stays free.
- To link one by hand, open its panel: the section *Linked Bridge In / Source Bridge Out* lists **every bridge of the opposite type with its lane**, its channel name and whether it is already linked. **Link** connects it, **Unlink** frees it. One Bridge Out feeds one Bridge In; linking a bridge that was already taken frees its previous partner.
- The **Bridge** tab of the properties lists every channel of the pool, complete or not.

Projects saved before this version keep their pairs: the channel code of an old bridge is its name.



If "Load" is slower, "Extract" waits (backpressure): no buffer to size, no rows lost.

Bridge Out settings

Setting	Options	Meaning
Channel name	text	a label for the channel; renaming does not break the link
Linked Bridge In	list of the Bridge In of the other lanes	**Link** / **Unlink** (see above)
Output to current lane	**None** (default) · **Passthrough** · **Signal**	Passthrough also sends the same rows to the following nodes of this lane; Signal emits one row (`channel` , `rows_sent` , `status` , `sent_at`) when the delivery is done
Sync	**Fire & Forget** (default) · **Wait for ack**	with *Wait for ack* the node finishes only when the other lane has received everything (status `delivered`); if that lane stops first, the node fails

Bridge In settings

Setting	Options	Meaning
Source Bridge Out	list of the Bridge Out of the other lanes, with their lane	**Link** / **Unlink**; the channel name is the Bridge Out's
Delivery	**As it arrives** (default) · **All at once**	*All at once* holds the rows until the sending lane has delivered everything, then passes them on; if the sender stops halfway, nothing passes — no partial data
Wait timeout	seconds, `0` = no limit	maximum wait for the first row; when it expires the node fails

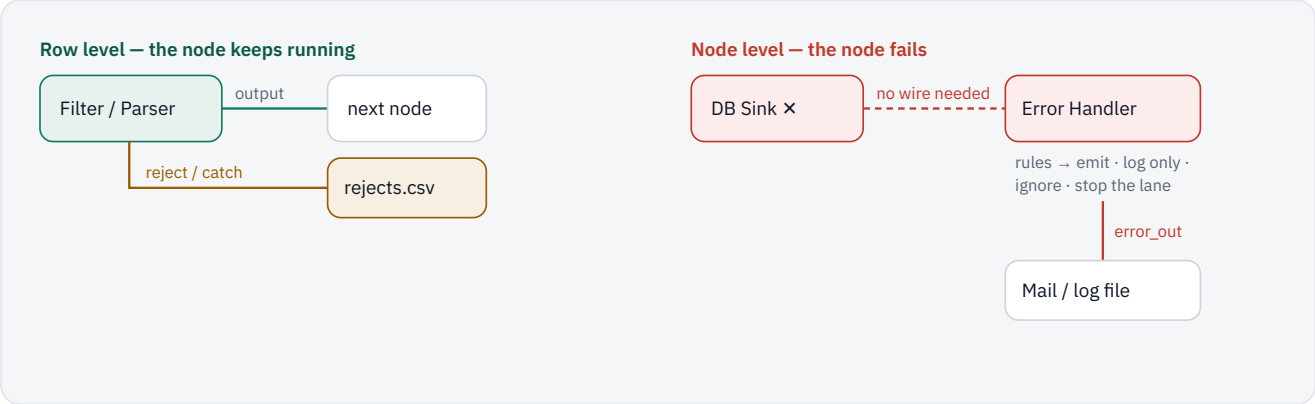
When the sending lane fails

Important. A failure in the sending lane reaches the receiving lane only if the failing node is marked **critical** (in the Error Handler, see below). Otherwise the sending lane's Bridge Out closes normally — perhaps with zero rows — and the receiving lane goes on: if it writes to a sink, it writes an empty result.

Rule of thumb: if the receiving lane must not run on incomplete data, mark the source nodes of the sending lane as critical **and** set the Bridge In to *All at once*.

06 Errors: reject, catch and the error handler

Stilton separates two kinds of problems.



Row level — a row is not valid but the node can continue. The row leaves on the `reject` port (or `catch`, see below) with fields that explain why. Connect the port to a file or a table and nothing is lost silently.

Node level — the node itself fails: a connection drops, a query is wrong, a timeout expires. What happens depends on the node's **On error** setting (tab *Advanced*):

On error	What happens
Error handler — the lane decides (default)	the error goes to the lane's Error Handler as a row of `_error_*` fields
Catch on node	the node gets a `catch` port: the rows that caused the error leave there, and the node continues
Retry, then error handler	tries again *Retry count* times, waiting *Delay between retries*; then as above
Retry, then catch on node	tries again, then as *Catch on node*

The Error Handler

Each lane's Error Handler receives every error the lane's nodes delegate to it, as soon as it happens. Its rules are evaluated from top to bottom; the first that matches decides.

Rule	Example
when Always → Emit	default behaviour: log the error and send it on `error_out`
when Node type is `mail_sink` → Log only	a failed notification is logged but not forwarded
when Node type is `source_http` → Stop the lane	without the API data the lane must not continue

The rows on `error_out` carry `_error_message`, `_error_node_id`, `_error_node_type`, `_error_lane_id`, `_error_at`, `_error_source` and `_error_critical`. Connect a File Output or a Mail node to keep or send them.

In the Error Handler's mapping panel each node of the lane has two switches:

- **Critical** — an error in this node always stops the whole lane, whatever the rules say. A rule can raise the severity, never lower it.
- **Exclude from log** — the node's errors are not copied to `error_out`.

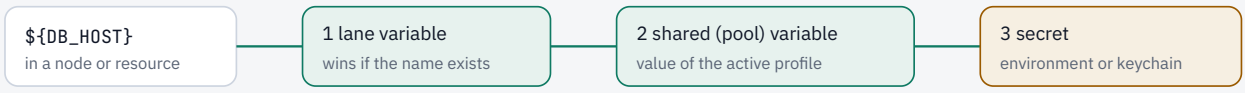
Example. A lane reads orders from an API and writes them to a database. Mark the API source **critical**: if the API is down, the lane stops at once instead of writing nothing and ending green. Leave the mail notification non-critical with a *Log only* rule: a mail server problem should not undo a successful load.

Stopping on purpose

The **Stop** node (section *Flow*) ends the lane deliberately when rows reach it — with rollback of open transactions and closing of connections. It is not a failure. Typical use: after a Filter, when finding a certain row means the batch must not be processed.

07 Variables, profiles and secrets

Values that change between environments, or that must stay secret, never go literally into node settings. They are written as references, `${NAME}`, and resolved at run time: in the fields of a **resource**, in the **paths** of File Input and File Output, and — for secrets — in the credential fields of a node. In an expression a variable is read with `var("NAME")`. In the other text settings of some nodes (a URL, a mail body) `${name}` stands for a **field of the row**, not a variable: see the node notes.



A secret is resolved only on the machine that runs: its value never enters the project or the artifact.

Shared variables and profiles

Open **Environments** in the toolbar.

01 Under *Shared variables (pool)* add the variables of the project with their default value: `DB_HOST = localhost`, `BATCH_DATE = 2026-01-01`.

02 Create profiles — for example `test` and `prod` — and give each the values that differ from the default.

03 Choose the **Active profile (used at Run)**.

Variable	Default	`test`	`prod`
`DB_HOST`	`localhost`	`db-test.internal`	`db.internal`

Variable	Default	`test`	`prod`
`OUT_DIR`	`out`	`/data/test/out`	`/data/out`
`THRESHOLD`	`100`	—	`5000`

With `prod` active, `${DB_HOST}` becomes `db.internal` and `${THRESHOLD}` becomes `5000`; with `test` active, `THRESHOLD` keeps the default `100`.

Profiles can be exported to a file and imported back (**Export…** / **Import from file…**), to prepare them outside the studio or share them:

prod.env.json — json

```
{ "profile": "prod", "values": { "DB_HOST": "db.internal", "OUT_DIR": "/data/out", "THRESHOLD": "5000" } }
```

Lane variables (tab *Lane*) are visible only to that lane and override a shared variable with the same name. Expressions read variables with `var("NAME")`; settings use `${NAME}`. A lane variable of type **counter** declares how `next("NAME")` counts — start, step, and whether it restarts every run or every session of a continuous flow; each node that calls `next` keeps its own sequence (*FPEL Reference*, *Counters*).

Secrets

In *Environments* press **+ secret** and give it a name, such as `DB_PASSWORD`. Only the name is saved. Use `${DB_PASSWORD}` wherever the value is needed. To test on your own computer, type the value under *Secrets — values on this machine (keychain)*: it goes into the system keychain, not into the project. The *Runtime Operations* manual explains how to provide secrets on the machine that runs.

08 Saving, versions and undo

- **Project → Save** writes a single `.ffplan` file. It contains the flow, the profiles and the **history** of previous versions; it never contains secret values.
- Every save moves the previous version into the history (the last 20 are kept). **History** lists them: you can **Restore** one, add a comment to the current version, or **Save new checkpoint** with a name — for example *"before switching to the new API"*.
- **Undo / Redo** work on the canvas and on node settings.
- Opening another project or closing the window with unsaved changes asks for confirmation.

Tip. The history travels inside the file, so it is handy for small steps. For team work keep the `.ffplan` files in Git as well: the two do not compete.

Projects from other versions of Stilton

A project saved by an earlier version **always opens**. What can be updated is updated on the spot; what cannot is kept, never thrown away. When something changed, the studio shows a **compatibility report** in three groups — click an entry to jump to the node:

Group	What it means	What you do
Updated automatically	The node was brought to the current version (new defaults, renamed settings)	Nothing — save to keep the update
To review	Something worth a look, such as a connection to a node that no longer exists, or a file from a newer studio	Read the note, check the flow
Must be replaced or rewritten	A node type this studio does not support	Replace or remove the node

A node type the studio does not know becomes a red, striped **incompatible node**. It keeps its name, its connections and all its original data — saving the project loses nothing, and a version of Stilton that knows the type restores the original node by itself. The validation marks it as an error, so the flow cannot run or be compiled until you replace it (for example with the node that took its place) or delete it.

For example, a project with a node `kafka_source` opened in a studio without Kafka shows *"Node type "kafka_source" is not known to this version of Stilton Studio"* in the report and in **Problems**; the rest of the project runs normally once that node is replaced.

A project saved by a **newer** studio opens too, with the note *newer version — save as copy* next to its name. Nodes it does not know become incompatible nodes, and **Save** asks for a new file: the original stays untouched for the newer studio.

Note. Each `.ffplan` records its file format (`formatVersion`) and the studio version that saved it (`studioVersion`). From the beta on, every release is tested against the projects saved by the previous releases.

Importing lanes from another project

Project → Import lanes... copies lanes of another `.ffplan` next to those of the open project — the way to reuse a flow written once, as parallel lanes that you then connect with bridges.

01 Choose the file, then the lanes to bring in (each shows its nodes and bridges).

02 Review what will happen and **confirm the bridges**: bridges linked to each other among the imported lanes stay linked; every other imported bridge arrives **unlinked**, and for each one you choose what to connect it to — the list shows the free bridges of this project with their lanes. A proposal appears when exactly one free bridge here has the same channel name, but nothing is linked until you press **Import**.

03 The lanes arrive at the bottom of the pool with new identifiers, so they never clash with this project's ones. Each lane header shows **from** : it remembers the project, the file and the lane it comes from.

What else is brought along:

Item of the other project	Here it...
shared variable that does not exist here	is added to the shared variables, with its values for the profiles that exist here
shared variable with the same name and value	is shared

Item of the other project	Here it...
shared variable with the same name and a different value	becomes a variable of the imported lanes (it takes precedence there), and the report says so
user function or decode table that does not exist here	is added
user function or decode table with the same name but different content	**this project's version is kept**, and the report warns you: check the imported expressions
resources, transactions, lane variables	travel with their lane

The imported lanes are a **copy**: changing them does not change the original project, and vice versa.

---

09 Validation, log and monitor

Validation (bottom dock) checks the project while you design: missing inputs, fields used but not produced, expressions that do not compile, settings that contradict each other. Each problem shows its node: click it to select the node on the canvas. Fix the errors before compiling: compilation stops on problems that would make the plan wrong — an expression that does not compile, a user-function example that is not true.

Log shows the messages of the last run in the studio, node by node, with rows in, rows out and rejects.

The **Monitor panel** shows the last run as numbers and charts:

Section	Shows
Nodes — timing and throughput	time spent and rows per second of each node
Drops per node	rows rejected or discarded, per node
Memory	memory of the studio and the engine against time. **This run** shows the run in progress or the last one, from its start; **Session** puts the runs of the session side by side, with the pauses between them compressed — if each run starts higher than the one before, memory is not being given back
Timeline — nodes × memory	when each node was active, against memory use
Resource connections	the connections opened by the run

Memory is sampled every 150 ms while a flow runs. A long run keeps its whole history: older stretches are drawn at a coarser resolution, keeping their peaks. A gap in the line means no samples arrived, for example while the computer was asleep.

Use it to find the slow node of a flow before you deploy it.

Field lineage and "explain this value"

- **Lineage** (toolbar) answers *"where does this field come from, and what does it feed?"*: choose a node and a field, and see the chain of nodes and expressions that produce it and those that use it.
- In the previews of transforms, **explain this value** shows how a single output value was computed, step by step.

10 A complete example in two lanes

The goal: read yesterday's orders from PostgreSQL, keep the valid ones, total them by region, and write a CSV — with invalid orders kept apart and errors e-mailed.

Lane "Extract"

01 **DB Input** `Read orders` on resource `warehouse`, query `SELECT * FROM orders WHERE order_date = '${BATCH_DATE}'`. Mark it **critical** in the Error Handler.

02 **Filter** `Valid orders`, route `ok` with condition `amount > 0 and customer_id is not null`. Connect `reject` to a **File Output** `Rejected orders` → `${OUT_DIR}/rejected.csv`.

03 **Bridge Out** `To load`, channel name `valid_orders`, sync *Wait for ack*.

Lane "Load"

04 **Bridge In** `From extract`, delivery *All at once*. It is the only free bridge of the pool, so it links to `To load` by itself; otherwise choose `Extract · To load` in its panel.

05 **Aggregate** `By region`, group by `region`, functions `{"count": "*", "sum": "amount"}`.

06 **File Output** `Totals` → `${OUT_DIR}/totals_${BATCH_DATE}.csv`.

07 In the lane's Error Handler, connect `error_out` to a **Mail** node that sends the errors to the operations team.

With `test` active the run reads the test database and writes into `/data/test/out`; compiled with `prod`, the same flow becomes an artifact for production (see *Build & Publish*). If the database is unreachable, *Read orders* is critical: lane "Extract" stops, the bridge never confirms the delivery, *From extract* fails, and lane "Load" writes nothing — no empty totals file.