HumAIn Flow

User Guide

Version of 16 September 2026

Contents

Introduction

HumAIn Flow is a web application for designing, validating and running workflows made of logical blocks, data connections and execution dependencies.

The application supports two main ways of working:

- building the flow by hand in the visual editor
- generating or reshaping the flow through the assistant

Once a flow exists you can save it, review its validation errors, run it from the `Tasks` tab, and — for flows that involve language models — study how a deliberately introduced bias, or a mitigation meant to counter one, changes what the flow produces.

This guide covers the standard use of the editor, of executions, and of the bias tooling. It describes what you see on screen and what each action does, not how the platform is implemented internally.

Signing in and finding your way around

After signing in you land on the main screen. The top bar contains:

- `Editor`: where flows are designed
- `Tasks`: where flow instances are inspected and run
- the user menu, under your username: `Change password`, and `Admin users` for accounts that have it

Areas you are not entitled to use are not shown at all.

Notifications — confirmations, warnings and errors — appear briefly at the bottom of the screen, colour-coded, and can be dismissed by clicking them.

The Editor layout

The `Editor` tab has three main areas:

- a left panel listing the available `Blocks` and `Containers`
- the central workflow canvas
- a right panel holding the `Assistant` and, when relevant, the `Errors` panel

The Tasks layout

The `Tasks` tab has:

- the list of executions in the left column
- the detail of the selected execution in the central area

Core concepts

Flow

A flow is the definition of a workflow. It contains:

- blocks
- containers
- data connections
- execution dependencies

A flow also carries a status. The one that matters day to day is whether it is `EXECUTABLE` — a flow that still has validation errors stays a `DRAFT` and cannot be run.

Block

A block represents a single step of the workflow. Typical examples:

- LLM blocks, which send a prompt to a language model
- input and output blocks
- human interaction blocks
- conditional blocks, which route the flow down one branch or another
- agent blocks, which let a model call external tools

Container

A container holds a `subFlow` — a nested flow. Use one to group part of the workflow into a reusable, more readable unit, or to repeat a part of the workflow over a list of items.

A container without a `subFlow` is considered incomplete.

Data connections

Standard connections link:

- a source output
- to a destination input

They exist to carry values between nodes.

Execution dependencies

Dependencies carry no data. They only impose an order of execution.

The labels visible on the nodes are:

- `Depends on`: this node must wait for another node
- `Prerequisite of`: this node unblocks the execution of another

Use a dependency when you want to guarantee the right order between two steps without passing a value as input.

Global inputs

Some values are not produced by a node but supplied when the execution starts — a document to analyse, a topic, a threshold. These are the flow's global inputs. They are declared once on the flow and can then be referenced by any block that needs them, instead of being wired node by node.

A block that references a global input which the flow does not declare is a validation error (`GLOBAL_INPUT_NOT_DECLARED`). Global inputs belong to the top-level flow: a subflow inside a container cannot declare its own.

Projects

A project groups related flows under one name. It gives you three things: a folder in the flows list, a set of shared values every flow in it can read, and an order in which those flows run when you run the project as a whole.

Projects are private to whoever created them. A flow can be published and shared; the project structure around it cannot.

Creating a project

In the editor sidebar, the `New` button above the flows list offers `New project` alongside `Empty flow`, `Create with AI` and `From JSON`. A project needs a name; the description is optional. Both can be changed later from the same dialog.

Organising flows into projects

Once projects exist, the flows list is grouped by project, with a `No project` group for everything else. Each group header shows the name, the description and how many flows it holds, and `Filters & sorting` gains a project filter.

There are two ways to put a flow in a project:

- from the project group's `⋮` menu, `New flow in this project`, which creates an empty flow already assigned;
- from a flow's own `⋮` menu, `Move to project`, which also offers `Remove from project`.

When a project group holds more than one flow, arrows appear beside each entry: `Run earlier in this project` and `Run later in this project`. That order is what a project run follows.

Shared context

A project's `⋮` menu offers `Shared context…` — values that every flow in the project can read at run time. Each entry has a name, a type (`TEXT`, `BOOLEAN`, `JSON` or `CSV`) and a value, and the dialog shows the placeholder to use it with, which you can copy:

```
${{project.myVariable}}
```

They work exactly like the flow's own global inputs, one level up: define a value once for the project, reference it from any flow inside it.

Names must be unique, must be usable in a placeholder (start with a letter, then letters, digits, `_`, `.` or `-`), and must not start with `project.`, `global.`, `context.` or `vars.`. Saving replaces the whole context, so an entry you remove from the dialog is removed for good.

Running a project

The play button on the project group header runs every executable flow in the project. It is available only when at least one of them is executable; the ones that are not are skipped, and the notification tells you how many.

The flows run **one at a time, in the project's order**, each starting only after the previous one has succeeded. The application takes you to the `Tasks` tab with the first execution selected, and executions started this way carry a chip with the project name.

Creating the run does not fill anything in for you. Each execution still needs its own inputs and credentials, exactly as it would on its own. A flow that fails stops the run, and a flow waiting for inputs or a credential blocks it — supply what is missing and run the project again to carry on from where it stopped.

Deleting a project

`Delete project` deletes the project **and every flow in it**, finalized flows included — the ordinary protection on a finalized flow does not apply here. The confirmation dialog lists the flows that will go, marks the finalized ones, and asks you to type the project name before the button becomes available.

Past executions are kept: each one holds its own snapshot of the flow it ran, so deleting a project does not erase its history.

Building a flow by hand

1. Open the editor

Go to the `Editor` tab. The sidebar lists the flows you can open, with a search box and a `Filters & sorting` disclosure offering visibility toggles, an ordering control and, once you have projects, a project filter. Each row shows whether the flow is public or private, who wrote it and when it was created.

The `New` button offers four ways to start:

- `Empty flow`
- `Create with AI`, which hands the job to the assistant
- `From JSON`, which imports a flow file
- `New project`

Each flow's `⋮` menu offers `Clone flow`, `Export flow` — which downloads it as JSON — `Move to project` and `Delete flow`.

2. Add blocks or containers

From the left panel:

- look for the block or container type you need
- drag it onto the canvas

If the catalogue is not ready yet, a loader is shown in place of the empty-list message.

3. Move and organise the nodes

You can:

- drag nodes around the canvas
- select and move nodes
- clone a node with the clone icon
- delete a node with the delete icon

Deleting a node also removes:

- the data connections attached to it
- the dependency edges attached to it

4. Connect the nodes

To pass data:

- connect an output to an input

To impose order only:

- connect `Prerequisite of` on the source node

- to `Depends on` on the destination node

Dependency connections are drawn differently from data connections: a thinner, dashed line.

5. Rename nodes

For both blocks and containers:

- click the pencil icon next to the name
- edit the name
- confirm with `Save`

6. Configure the parameters

Clicking a node shows its parameters.

For editable fields:

- use the pencil button on the parameter
- edit the value in the dialog
- save

For long texts:

- a long field is truncated on the node
- open it in full with the eye icon

In read-only subflow views the long fields keep the eye icon, so you can still read them completely.

Placeholders in prompts and parameters

A parameter can reference a value produced elsewhere — a connected input or a global input — through a placeholder. On the node, placeholders are rendered as distinct segments rather than plain text, so you can tell at a glance which part of a prompt is literal and which part will be substituted at run time.

During an execution, the same preview shows the resolved value once it is available; until then the original placeholder stays visible.

Working with containers

Putting a subflow inside a container

A container can receive a subflow in more than one way:

- by importing an existing flow
- by dragging a selection of nodes into it

When a subflow is replaced:

- the structural configuration of the container is rebuilt
- parameters belonging to the previous subflow are not preserved

Importing a flow into the container

If the container type supports it:

- click `Import flow`
- choose one of the available flows
- confirm

Viewing the subflow

If the container holds a subflow:

- a `View Flow` button appears
- it opens a read-only preview window

From that view you can:

- explore the subflow
- open long parameters in read-only mode with the eye icon

Validation errors inside a subflow

A subflow is validated together with the flow that contains it. An error produced inside a subflow is reported on the container that holds it — that is the node you can actually see and act on — and the inner node responsible for it is highlighted when you open the subflow view.

Required parameters and local validation

If a node has missing parameters, a warning indicator appears on it.

For blocks, the warning lists the missing required fields.

For containers:

- if the `subFlow` is missing, only `Subflow` is listed
- the inner fields of the `FlowData`, such as `Blocks`, `Connections` or `Dependencies`, are not reported as missing

Technical type fields are never shown as user parameters. These include:

- `type`
- `typeName`
- `containerType`
- `configurationType`
- `configurationClass`

Fields that belong to a branch of the configuration you have not selected are hidden as well: a parameter that only applies when another field has a particular value appears when that value is chosen, and does not count as missing before then.

Using the assistant

The assistant lives in the right panel of the editor.

Create and refine modes

The assistant changes behaviour depending on the state of the open flow:

- with no flow open, or with an open flow that is still empty, it works in `Create` mode
- with a flow that already contains nodes or connections, it works in `Refine` mode

An empty flow therefore does not block assisted creation.

What you can ask

Typical requests:

- create a flow from scratch
- modify an existing flow
- explain what a flow does
- help fix validation problems

What you get back

When the assistant returns a draft:

- the flow is loaded into the editor
- you can keep editing it by hand
- you can save it like any other flow

The assistant proposes; nothing is persisted until you save.

Saving a flow

While you work in the editor the flow can be modified but not yet saved.

Saving

Use the `Save` button in the flow toolbar.

Saving:

- updates the flow on the backend
- recomputes validation when needed

Renaming the flow

The flow title uses explicit actions:

- `Save`
- `Cancel`

It is not saved automatically when the field loses focus.

The validation errors panel

The right rail has a dedicated icon for flow errors.

When it appears

The icon:

- is always visible in the right rail
- is disabled when there are no errors
- becomes active when the flow has validation errors

How errors are loaded

Errors are requested from the backend:

- after saving, when the flow is not `EXECUTABLE`
- when opening a flow that is already a `DRAFT`

What the panel shows

For each error you see:

- a code
- a readable message

Noisy metadata such as `entity`, `field` and `id` is not shown on the card.

Node highlighting

When an error names related nodes, those nodes are highlighted on the canvas. If the error comes from inside a container's subflow, the container is highlighted while you are looking at the main flow, and the inner node is highlighted once you open the subflow.

Stale validation

If you make a structural change without saving, the errors panel shows a notice:

`Validation will be recomputed after save.`

Purely graphical moves of the nodes do not count as structural changes.

Published and Finalized

If you own the flow, two controls appear in the toolbar:

- `Published`
- `Finalized`

Published

Controls the visibility of the flow. You can publish and unpublish freely.

Finalized

Marks the flow as definitive and no longer modifiable.

Once finalized:

- the content of the flow becomes read-only
- the flow cannot be un-finalized
- the flow cannot be deleted
- publishing and unpublishing remain available

Finalize a flow when you intend to use it as a stable baseline — for instance as the reference version for a bias experiment.

Running a flow

When a flow is valid and executable, `Execute` becomes available.

What happens when you click Execute

The application:

- switches to the `Tasks` tab immediately
- creates the execution in the background
- shows a loader until the new execution is ready

This avoids the perceived delay before the tab changes.

Working in the Tasks tab

The `Tasks` tab shows a list of executions on the left and the detail on the right.

The execution list

Each entry shows:

- the name
- the status
- the date and time
- a `Simulated` badge when the run was simulated
- the run number, when the execution is part of a group of repeated runs

The detail of an execution

In the detail you can:

- see the graph in read-only mode
- fill in the required inputs
- read outputs and logs
- take the actions that are available for the current state: start, simulate, cancel, resume, re-run

Execution inputs

When an execution step needs manual input you provide it in the dedicated panel.

How inputs are saved

Inputs are not sent automatically when a field loses focus. The behaviour is:

- you edit the value
- the draft stays local
- you press `Save` to send it

This applies to multi-value fields as well.

Text and list inputs

A text input is edited in place; long values can be opened in a larger editor. A list input lets you add and remove entries, each one edited on its own row.

File inputs

An input that expects a file is filled through a drop zone: drag a file onto it, or click it to pick one.

Once a file is chosen it is uploaded straight away and the zone reports its state:

- while the transfer is running, a progress indicator and the file name
- when it is finished, the file name and a `Replace` action
- if the transfer fails, the error and the possibility to try again

A note under the zone states the behaviour explicitly: the file is stored as soon as it is chosen, not when you press `Save`. To change your mind, use `Replace` and pick another file.

Prompts and parameters that reference an uploaded file show its file name, not the internal storage path.

Files on global inputs

A file can also be supplied as a global input of the flow, so that several blocks use the same document without uploading it more than once. A block input that accepts a file can either take a file of its own or point at a global input; when it points at a global input, the file is chosen once with the rest of the execution inputs.

Finding your way around the panel

The panel keeps a running count of how many inputs are provided out of how many are required, and separates `Global inputs` from `Node inputs`, each group badged with how many of its values are still missing, or ticked when none are.

At the bottom, a bar states whether you have unsaved changes and how many. It is the only thing that sends them.

A few conveniences are worth knowing:

- a list input can be filled in one go with `Import from JSON array`;
- a long value can be opened in a larger editing box;
- `Copy from another run` fills the panel from a previous execution. The values arrive as unsaved changes, so you can review them before saving. Files and credentials are not copied — an uploaded file belongs to the run it was uploaded to, and credentials never leave the server.

Credentials and authorizations

Some blocks need a secret before they can run: an API key for a language model provider, an authorization header for an HTTP call. The application treats these two cases differently on purpose.

	Provider credential	**Runtime authorization**
Typical case	an LLM provider's API key	a header value for an HTTP call
What you supply	a **saved credential**, chosen from a list	the value itself, typed into a masked field
Where it is kept	encrypted in the vault, on the server	with the execution

A provider key is never pasted as a plain value: it always goes through the vault.

When an execution asks for a credential

An execution that needs one says so before it will start. A banner above the graph reads:

> This execution needs a Vault credential for `<provider>` before it can start.

and the start button is disabled, with a tooltip naming the missing provider. `Choose credential` takes you to the requirement in the `Inputs` tab.

Each outstanding requirement appears as its own card, headed `Vault credential for <provider>`, listing the steps that need it, with a dropdown of the credentials you already have for that provider and an `Add credential` button for when you have none or want a new one.

Adding a credential

`Add credential` asks for:

- the **provider**, which is fixed to the one being asked for;
- a **label**, so you can recognise it later;
- an optional **description**;
- the **API key** itself.

The value is encrypted and stored, and is never shown again — not in this dialog, not anywhere else. Only your own active credentials for that provider are offered.

Once a requirement is satisfied, the amber card becomes a green row naming the credential in force, with `Change` to pick a different one. A credential that cannot be used — unknown, disabled, or belonging to a different provider — is refused at the moment you choose it rather than in the middle of the run, and the reason is shown under the picker.

Runtime authorizations

Requirements that are not provider credentials appear in the `Provider Authorizations` section of the inputs panel. Each card names the provider and the field, describes what is wanted, lists the steps that need it, and offers a masked input with `Show` / `Hide` and its own `Save`.

Credentials and reruns

A rerun carries over the authorizations that were already provided, so it usually starts ready to go. Copying inputs from another run does **not** carry them over.

Human interaction blocks

Human interaction blocks can ask for a confirmation or for a manual answer.

The interaction dialog

When a node requires it, a dedicated dialog opens.

In the non-chat case you can:

- confirm the current input
- edit the answer
- send it with `Send Output`

In the chat case you can:

- carry on the conversation
- send a final answer

These buttons really do send

The sending buttons perform a real call to the backend. Specifically:

- `Send Output`
- `Confirm Input`
- sending a chat message
- sending the final answer

all go through the interaction endpoint of the execution.

Reading an execution

The detail of an execution has a header, the graph, and a side panel with five tabs: `Inputs`, `Intermediate`, `Logs`, `Output` and `Bias impact`.

The header

The header carries the run's name, a fullscreen toggle, and any badges that apply — `Simulated`, `Subflow` for one iteration of a container's subflow, and the bias variant badges described later. A `Details` disclosure shows the execution identifier and, for a subflow run, its parent, container, iteration and role. Below it, five tiles: status, start, end, duration and steps.

Two banners appear when they apply:

- *Execution cancelled. This execution is closed. Start a new execution to continue.*
- *Execution suspended after service restart. Resume the execution to continue.* — paired with a `Resume execution` button.

Outcomes

Above the graph, a collapsible `Outcome(s)` section lists what each End node concluded: the outcome code, its label, and the step it came through, with its payload rendered as text or as a JSON tree. An End node that received no value shows only its label, and says so.

Task nodes

In the execution viewer:

- containers show a `View Subflow` button
- dependency ports are shown only when they are actually connected
- long texts can be opened in read-only mode

Outputs

Outputs are grouped by node. For each node you see the node title and the list of its individual outputs.

If an output exceeds the expected length it is truncated, and the eye icon opens it in full.

When a response is an array it is not shown as a single blob: it is expanded into separate entries, `Item 1`, `Item 2`, and so on.

Logs

In the logs tab:

- the content scrolls automatically to the bottom as it refreshes
- the readable text is shown
- the raw JSON detail block is not displayed

Intermediate inputs

The `Intermediate` tab shows the inputs each step actually received, grouped by node, with the eye icon for long values. For agent and LLM steps it is what was really sent to the model — the first place to look when a step produces something unexpected, because it distinguishes a bad prompt from a bad answer.

The execution tree

When a run has container iterations, an `Execution tree` panel appears under the executions rail. It shows the run, its container steps — badged `waiting` when a container is waiting on its subflow — and one child entry per iteration. Clicking any of them opens that execution.

This is the only place to reach *past* iterations of a container; the canvas shows the current one.

Containers waiting for a person

When more than one container is waiting for a human answer, a bar above the viewer lists them, one button per container and iteration, so you can go straight to the one you mean to answer.

Simulated runs

A simulated run is one where the flow's **human interaction steps are answered by a language model instead of by a person**, so the flow can go from end to end without anyone sitting at the dialog.

Starting one

Next to the ordinary start button in the execution toolbar there is a second play button, marked with a robot icon rather than a person. The two tooltips say which is which:

- `Run – you answer the human steps`
- `Run simulated – an LLM answers the human steps`

The simulated button only appears when the flow actually has interactive steps and all of them can be simulated. It is otherwise subject to the same conditions as a normal start: the inputs filled in, the credentials provided, the execution not yet started, and not a subflow run.

Clicking it opens `Simulation Settings`, which asks for the provider and the model, and — under `Model parameters` — optionally for temperature, top P, top K, maximum tokens and a seed. Each is left at the server's default when empty.

The seed deserves attention. It fixes the randomness, so the same inputs produce the same answer. Without it, two runs of the same flow differ for reasons that have nothing to do with anything you changed — which is precisely what you are trying to rule out when you compare runs.

What is different about a simulated run

- The interactive nodes cannot be opened during the run; the interaction dialog will not appear.
- The execution header carries a `Simulated` badge and, under `Details`, the simulator that answered: `provider / model`.
- The execution list marks the run with the same badge.
- Containers pass the simulation down to their subflow runs automatically.

Only the *answering of interactive steps* is simulated. LLM blocks, HTTP calls and agent steps run exactly as they normally would.

Repeating a simulated run

A rerun inherits the simulator of the run it repeats, but not the decision to simulate — that stays an explicit act. When you rerun a simulated execution, a notice says so:

> The run this one repeats was simulated with `<model>`. Start it simulated, with the same model, or the comparison will also be comparing two simulators.

The settings dialog preselects that model for you.

Re-running and comparing executions

Re-running

Each run in the executions list has a replay button, `Rerun execution`, available once the run has reached a final state. Subflow runs cannot be rerun on their own.

A rerun carries over the workflow inputs, the global input descriptors, the credentials that were provided, and the simulator descriptor. It is **created, not started**: you still press play, or the simulate button.

Re-running is also how you find out how much a flow varies on its own, which is worth knowing before you read any comparison — a difference no larger than the flow's ordinary run-to-run variation is not evidence of anything.

Run history

Runs of the same flow are grouped into one collapsible card in the executions rail, showing the flow name, the project chip when there is one, the number of runs, the latest status and when it last ran. Collapsed, it offers `Open latest execution`. Expanded, it lists each run by number, with a badge saying whether it is an original `Run` or a `Rerun`, and — for a rerun — which run it repeats.

Above the list there is a search box and a `Filters & sorting` disclosure with status toggles (`All`, `Init`, `Running`, `Paused`, `Final`) and an ordering control.

> These run groups are the rerun history of a single flow. They are a different thing from a project run, which is one execution of each of several flows.

Comparing two runs

On a group with more than one run, the `Compare` button turns on a picking mode: tick exactly two runs — the bar keeps count — and confirm. The comparison opens in place of the execution viewer.

It shows:

- a header naming the two runs, a toggle between `Only differences` and `All values`, and a `Close` button;
- a summary line — *X of Y nodes differ* — followed by the caveat that model output varies between runs on its own, so a difference is not by itself evidence of a change in behaviour;
- one section per outcome and per node, each marked `equal`, `changed`, `only-left` or `only-right`, with the status each side reached and the differing values side by side, highlighted inline.

Two runs that share no node at all are almost certainly runs of different versions of the flow, and the view says so rather than lining up unrelated steps. If the two runs were answered differently — one simulated and one not, or simulated with different models — a warning says that part of what follows comes from that, not from the runs themselves.

This is the general-purpose comparison. A bias report does the same work, but knows which intervention was applied and can therefore speak about its effect.

Bias and mitigation: the idea

A flow that runs successfully is not necessarily a flow you should trust. A prompt can lean on a stereotype, a decision step can favour one kind of answer, an automated recommendation can be taken at face value by the next step. The bias tooling exists to turn that worry into a measurement: you state the risk on the node where you suspect it, you inject the suspect behaviour on purpose, and you look at what actually changed downstream.

The work has three stages:

1. **Annotate** — record the risk on the node, in the editor.
2. **Probe** — optionally give the annotation an executable behaviour, so an experiment can switch it on.
3. **Experiment and read the report** — run the flow with the probe active, compare it against a baseline run, and see whether the outputs, the branches or the final outcome moved.

The same machinery works in both directions. A **bias probe** injects the problematic behaviour to see what it does. A **mitigation probe** injects a corrective instruction to see whether it helps. You can also activate both at once.

> Nothing here scores a flow as fair or unfair. The platform shows you what changed; the judgement remains yours.

Annotating a node

Where annotations live

Bias annotations are attached to a node in the **editor**, on the node card itself. Not every block type accepts them — where the feature is not offered, the trigger does not appear.

A node that carries annotations shows a small badge with the count. Hovering it tells you how many there are and the highest severity among them, and states when one of them carries an executable probe.

Clicking the `Bias annotations` trigger opens the list of annotations for that node, with `Add bias annotation` in its header and `Edit` / `Remove` on each card. A node can hold up to 50.

During an execution the same list is reachable from the node in the execution canvas, in read-only mode.

What an annotation records

The form is driven by the server, so the exact set of fields follows the platform version. Currently it asks for:

Field	Required	What it is for
`Category`	yes	The kind of risk: selection, automation, historical, accessibility, measurement, confirmation or exclusion bias, or a transparency risk. The form shows a description of the option you pick.
`Severity`	yes	Low, Medium or High.
`Issue`	yes	The risk itself, in prose. Up to 2000 characters.
`Rationale`	no	Why it matters. Up to 4000 characters.
`Mitigation`	no	A possible corrective action, in prose. Up to 4000 characters. This field is documentation — it is not executed.
`Status`	no	Where the review stands: Proposed, Confirmed, Mitigated or Dismissed. New annotations start as Proposed.
`Source`	no	Manual or Automated. Annotations you write by hand are Manual.
`Analysis identifier`	no	For annotations produced by an external analysis.
`Bias probe`	no	The executable behaviour to measure. See below.
`Mitigation probe`	no	The executable correction to apply. See below.

Length-limited fields show a live character counter.

An annotation with no probe is perfectly useful: it documents a risk for whoever reviews the flow. It just cannot be run.

Behavioral probes

A probe is what makes an annotation executable. It is described as *an optional controlled behaviour used only by bias experiments* — it never affects a normal run of the flow, only a run started from an experiment.

A probe has an **activation mode**, which says how the behaviour is injected. Which modes are on offer depends on the node type, so the editor loads them per node:

Mode	What it does	Where it applies
`PROMPT_DIRECTIVE`	Adds an experimental behavioural directive to the node's prompt.	LLM blocks, MCP agent and agent-chat blocks, human interaction and decision blocks, and conditional or switch blocks that are configured to use an LLM.
`INPUT_TRANSFORMATION`	Rewrites selected textual inputs before the node runs.	Any block.
`OUTPUT_TRANSFORMATION`	Rewrites textual outputs after the node runs.	Any block.
`ROUTING_OVERRIDE`	Forces a conditional, switch or human decision block down a named branch.	Conditional, switch and human decision blocks.
`MOCK_RESPONSE`	Skips an external call and returns a configured answer instead.	HTTP server-call blocks.

The fields you fill in follow from the mode:

- **Prompt directive** asks for an `Experimental instruction` — the text added to the prompt.
- **Input transformation** and **output transformation** ask for a `Transformation template`. `${original}` is the only placeholder, and stands for the value being transformed; if you leave it out, your instruction is simply prepended to the original value. Input transformation additionally offers a `Target inputs` checklist of the node's real input ports — leave them all unchecked to target every textual input, and note that multiple inputs are transformed item by item.
- **Routing override** asks for the `Forced output branch`, chosen from the node's actual output ports.
- **Mock response** asks for one typed value per output port rather than an instruction — a checkbox for boolean outputs, a JSON value for structured ones, plain text otherwise.

Every probe also has an `Expected impact` field. It records what you expect to observe; it does not change the execution.

> A prompt directive on a human interaction node biases the *simulator's* prompt. If a real person answers that step, the probe does nothing.

A probe counts as **executable**, and is therefore selectable in an experiment, once it has an activation mode and the content that mode needs — a non-empty instruction, or at least one mock output for a mock response. The annotation list marks such annotations with an `Executable probe` badge.

Bias probe or mitigation probe

The two probe editors are identical. What differs is intent, and which side of the annotation an experiment switches on:

- the **bias probe** is *the behaviour to measure* — inject the suspect behaviour and see what moves;
- the **mitigation probe** is *the correction to apply* — inject the corrective instruction and see whether the answer improves.

Every experiment dialog has an `Intervention direction` selector that decides which of the two fires:

- `BIAS` — only annotations that carry a bias probe are offered, and only bias probes run.
- `MITIGATION` — only annotations that carry a mitigation probe are offered.
- `BOTH` — only annotations that carry *both* are offered, and both fire in the same run.

Take care with the word *mitigation*: the `Mitigation` text field and the `Mitigated` status are documentation, while the mitigation **probe** is the executable one.

Running an experiment

The two shapes of experiment

There are two, and you choose between them by choosing which control you click — there is no single dialog with a toggle.

	Isolated step experiment	**Full-flow biased rerun**
Started from	the chart icon on a **node**, in the execution canvas	the `Create biased rerun` button on the execution **toolbar**
What runs	that one block, repeated with the probe on	the whole flow, as a new execution
Baseline	the output already recorded in the completed run	the run you started from
Report	produced immediately, in the same dialog	produced later, when you click `Compare with baseline`

Both need a **finished** baseline execution: an experiment compares a run against a run, so the baseline has to be complete.

Measuring one step in isolation

The chart icon appears on nodes that carry an executable probe. It is shown disabled, rather than hidden, when the experiment is not currently possible, and the tooltip says why — the execution has not finished yet, or the node cannot be replayed on its own.

Nodes that cannot be replayed on their own are the user-interactive ones (chat interaction, human decision, human interactive) and containers. For those, the tooltip points you at `Create biased rerun` instead.

The `Measure bias impact` dialog asks for:

- the `Intervention direction`;
- which `Executable annotations` to activate — all of them are pre-selected, and changing the direction drops the ones that no longer qualify;
- `Repetitions`, between 1 and 10, 3 by default — how many times to run the biased variant;
- whether to `Include raw outputs in the report`, on by default;
- the external side effect policy, described below.

The experiment is queued on the server and the dialog reports its progress. When it finishes, the report replaces the form in the same window.

Two things are worth knowing about how an isolated experiment works. The baseline is **not** re-run: it is the output already recorded in the completed execution, and only the variants are executed. And because only one block runs, an isolated report has **no downstream section** — nothing downstream was executed to observe.

Closing the dialog does not cancel the job. It keeps running, and its report appears in the `Bias impact` tab when it is done.

Creating a biased rerun

`Create biased rerun` is on the toolbar above the execution graph. It needs a top-level run — a subflow run cannot be a baseline — that has reached a final state and contains at least one node with an activatable probe. When one of those conditions is missing, the `Bias impact` tab states which one.

The dialog lists one section per candidate node:

- for ordinary nodes, a checkbox per eligible annotation. Unlike the isolated dialog, **nothing is pre-selected** here;
- for containers, a single `Activate biases in the subflow` checkbox, which turns on every executable probe on the nodes inside it.

Confirming creates a **new execution** of the flow, with the same inputs and the probes active, and takes you to it. No report is produced yet — the variant simply runs.

A variant run is clearly labelled. Its header carries a `Bias variant`, `Mitigation variant` or `Bias + mitigation` badge, with the reminder that probes were active and the result is therefore not a baseline, and an expandable block showing the experiment, baseline and intervention identifiers. In the run picker, such runs are suffixed `· bias variant`.

Comparing with the baseline

Once the variant has finished, `Compare with baseline` — the compare icon, shown only on a variant run — produces the report. The comparison walks both runs and records what differs.

Comparisons are cached: asking again for the same pair returns the report that already exists, and a report built by an older version of the comparison is recomputed in place, keeping its identity and its assessment history.

External side effects

Some blocks call the outside world — HTTP server calls, MCP agents, MCP agent chats. Repeating them in an experiment would repeat their effects, so every experiment dialog has an `External side effects` section. When the nodes you selected can make such calls, a warning appears above it.

Policy	What happens
`Block external calls`	**The default.** The experiment refuses to run anything that would reach an external service.
`Mock external calls`	The call is not made. The block is bypassed and its outputs are filled with placeholders.
`Allow after confirmation`	Real calls are made. You are asked to confirm before the experiment starts.

Choosing `Allow after confirmation` and pressing the run button opens a confirmation dialog first; your answer is recorded on the resulting execution, so the choice remains auditable afterwards.

Three practical consequences:

- With the default policy, an experiment on a flow containing an HTTP or MCP node will **fail** rather than run — the message says the side effects were blocked. This is the most common reason for an experiment to error.
- The isolated experiment checks this up front, so it fails immediately. A full-flow rerun only discovers it when the offending step is reached, in the middle of the run.
- With `Mock external calls`, everything downstream sees a placeholder instead of a real answer. Differences downstream of a mocked node therefore partly reflect the mock, not the intervention. The report lists every mocked call for exactly this reason.

Reading a bias impact report

The same report viewer is used everywhere: inline at the end of an isolated experiment, in the compare dialog, and when you open a stored report from the `Bias impact` tab.

The headline

The top of the report tells you, before any number, whether the thing that matters changed:

- `Final decision changed` or `Final decision unchanged` — set when the flow's final outcome differed, or when it took a different branch somewhere.
- `X of Y subjects changed`, when the compared output is a list and each element is a subject.
- The strongest numeric delta, written in the flow's own units, for example `Score 5 → 8 (+3)`.
- A chip about simulation, when either run was simulated.
- A chip summarising the model assessment, when one has been requested.

Below them, a one-sentence summary generated by the server, which leads with the decision, then the subjects, then the averages, then the counts of changed downstream nodes and routing changes.

A collapsed section carries the identifiers — baseline execution, biased execution, experiment, node, annotations, repetitions — for when you need to trace a report back to its runs.

Immediate impact

What the activated node itself produced, baseline against variant.

Three figures are given: whether the output changed at all, the change rate, and the maximum **text difference**, which runs from 0 for identical to 1 for maximally different.

For list-shaped outputs there is a `Per subject` breakdown, showing only the changed subjects by default. Each subject expands into a two-column diff with word-level highlighting. For container nodes, the breakdown is per iteration, and states the status each side ended on.

The numbers come in three flavours, in increasing order of usefulness:

- a normalised **text difference**, which tells you *that* something moved;
- the **share of list elements** that changed, which tells you *how widely*;
- **labelled numeric deltas**, which tell you *how much*, in the flow's own units. These are extracted from plainly labelled numbers such as `Score: 7` or `confidence = 0.8`, which is why they are sometimes absent.

List elements are paired by position: element *i* is treated as the same subject on both sides.

Downstream impact

Every node downstream of the activated one, with whether the change propagated, the status each side reached, and the same per-field and per-subject drill-down. A checkbox narrows the list to the changed nodes.

An isolated experiment always shows this section empty, and says so.

Routing and outcome changes

The two largest things an intervention can do:

- an **outcome** change — the flow ended on a different outcome;
- a **routing** change — the flow took a different branch at a named node.

A routing change is reported only when both runs produced exactly one branch at that node and the two differ.

Mocked side effects

Present only when something was mocked. Each entry names the node and the kind of call that was skipped. Read it as a caveat on everything downstream of those nodes.

Warnings

Always shown, even when empty. The ones you will actually meet:

- on an isolated report, that the baseline was captured from the completed run and only the variants were repeated;
- on a full-flow report, that observed differences may also include ordinary non-determinism from external models;
- that only the first 200 elements of a long list, or the first 20 000 characters of a long text, were compared;
- that the two runs did not perform the same number of container iterations;
- that the interactive steps were simulated on one side and answered directly on the other, or simulated with different models — in which case part of the difference does not come from the intervention at all.

Warnings are not failures. They tell you how far to trust the numbers above them.

Highlighting the report on the canvas

`Highlight on canvas` closes the report and paints it onto the execution graph. A legend appears above the canvas, nodes carrying an active annotation and nodes whose output changed get their own badges, and connections on a branch that changed are marked. `Back to normal view` restores the ordinary canvas.

Asking a model to assess a report

A report measures differences; it does not say whether a difference matters. `Evaluate impact with LLM` asks a model to read the compared pairs and give an opinion. You choose the provider and model in the usual picker.

Each assessment yields:

- an **impact level** — `NONE`, `COSMETIC`, `SUBSTANTIVE` or `DECISIVE` — how far the meaning of the output moved;
- an **attribution** — `INJECTION`, `NON_DETERMINISM` or `UNCLEAR` — whether the change carries the intervention's fingerprint or looks like ordinary model variation;
- a narrative, plus a badge on each individual subject and field.

Assessments accumulate: the report keeps a history, newest first, so a second opinion does not erase the first. The per-subject badges follow the most recent one. Pairs that are identical are never sent to a model.

Two limits are worth stating plainly, and the viewer states them itself. This is an assessment by a model, not a measurement. And with a single run per side, a difference cannot be separated from ordinary model variation with certainty — which is the argument for raising `Repetitions`, and for setting a seed on the judge model when you want a repeatable opinion.

> A report saved with `Include raw outputs` switched off cannot be assessed later: the outputs it would need were not kept. If you may want a model's opinion, leave that box ticked.

The Bias impact tab

Every report produced for an execution is listed in the `Bias impact` tab of the execution panel, alongside `Inputs`, `Intermediate`, `Logs` and `Output`.

Each row shows the kind of report, whether anything changed, the date, the summary sentence, how many annotations were activated, and whether a model has assessed it. Clicking a row opens the full report.

When the tab is empty it explains what would fill it: on a variant run, that comparing it with the run it repeats is what produces the report; on an ordinary run, what a report is and how many nodes carry a probe that could be activated.

Administration

Accounts with the administrator role reach a separate workspace through `Admin users` in the user menu. It has a `Back to editor` link and a sidebar with two groups: `Auth`, holding `Users` and `Create user`, and `Stats`, holding `Operations`.

Users

The list shows each account with its username and email. For each one you can change the role between `USER` and `ADMIN` — the `Save role` button stays disabled until you actually change it — reset the password, or delete the account.

Create user

Provisions a new account: username, email, role and password. A live checklist below the password field shows which policy rules the password currently satisfies.

Operations

Aggregated usage figures, for everyone or for one account at a time. The cards cover flows (with how many are published and finalized), executions (running, succeeded, failed), simulations, and — once a user is selected — authentication figures such as login count and average session length, alongside the last flow update and last execution.

Practical tips

- save often after structural changes
- check the errors panel before running
- use `Connections` to pass data and `Dependencies` only to impose order
- use containers to isolate reusable parts of the flow
- if a node looks incomplete, check its missing parameters before running
- if a text is truncated, use the eye icon instead of widening the node
- finalize the flow you intend to use as a baseline, so later runs remain comparable
- before trusting a difference between two runs, re-run the baseline once and see how much the flow moves on its own
- when a comparison has to mean something, simulate both runs with the same model and a fixed seed
- leave `Include raw outputs` ticked on an experiment you may want a model to assess later
- annotate risks as you build, even without a probe: an annotation with no probe is still the record of a concern

Common problems

I see a loader instead of the blocks in the sidebar

The block or container catalogue is still loading. Wait for the backend to return the type descriptors.

A node looks empty when I open it

The type schema may not be available yet. Clicking the node makes the application retry the load.

A container is reported as incomplete

Check that it has a `Subflow`. A container without one is considered missing.

The flow stays DRAFT after saving

Open the validation errors panel. The flow may have been saved and still not be executable.

An error points at a container, not at a block

The error comes from inside the container's subflow. Open the subflow with `View Flow`: the node responsible is highlighted there.

I cannot see a whole answer in the tasks view

If the value is long or truncated, use the eye icon to open the full content in read-only mode.

The start button is disabled and I cannot see why

Check the banner above the graph. An execution that still needs a provider credential cannot start, and the tooltip on the start button names the provider it is waiting for.

The experiment button on a node is greyed out

Its tooltip says why. Either the execution has not finished yet — an experiment needs a completed baseline — or the node cannot be replayed on its own, which is the case for interactive steps and containers. For those, use `Create biased rerun` on the toolbar instead.

My experiment failed with a side-effect error

The flow contains an HTTP or MCP node and the policy was left at `Block external calls`. Choose `Mock external calls`, or `Allow after confirmation` if the real calls are safe to repeat.

I cannot ask a model to assess my report

The report was saved without its raw outputs, so there is nothing left to assess. Run the experiment again with `Include raw outputs in the report` ticked.

A comparison shows differences I did not cause

Read the warnings. Model output varies between runs on its own; if the two runs were simulated differently, or not simulated at all, part of the difference comes from that. Fix a seed and simulate both sides the same way, then compare again.

My uploaded file does not appear in the prompt

Check that the input is the one the block actually reads. A block input can either hold its own file or point at a global input; if it points at a global input, the file must be provided with the execution inputs.

Conclusion

HumAIn Flow lets you work visually and with assistance. The recommended path is:

1. create or open a flow
2. build it by hand or with the assistant
3. save it
4. fix any validation errors
5. run it from the `Tasks` tab
6. follow inputs, outputs and logs to completion

And, when the question is not only *does the flow work* but *what is it sensitive to*:

1. finalize the flow and record a baseline run
2. annotate the steps you want to probe
3. run the experiment and read the report