

Workflows

From the task to the signed run

Technical reference for the engine, the capability registry, the access log and the sixty-one core tasks.

Contents

Understanding

1. Mental model	3
2. The execution model	4
3. Anatomy of a workflow	5
4. The step and its settings	7
5. Substitution and context	9
6. Contract and implementation	10

Running

7. States	13
8. Run, dry run, debug	14
9. The debugger	15
10. Triggers and arming	16
11. Composition and iteration	17

Governing

12. Capabilities and consent	19
13. The access log	21
14. The panel and the commands	22

Reference

15. Task catalogue	25
16. What the assistant sees	32
17. Four workflows to copy	33
18. Troubleshooting	36
19. Limits and persistence	37
20. Declaring a task	39

Understanding

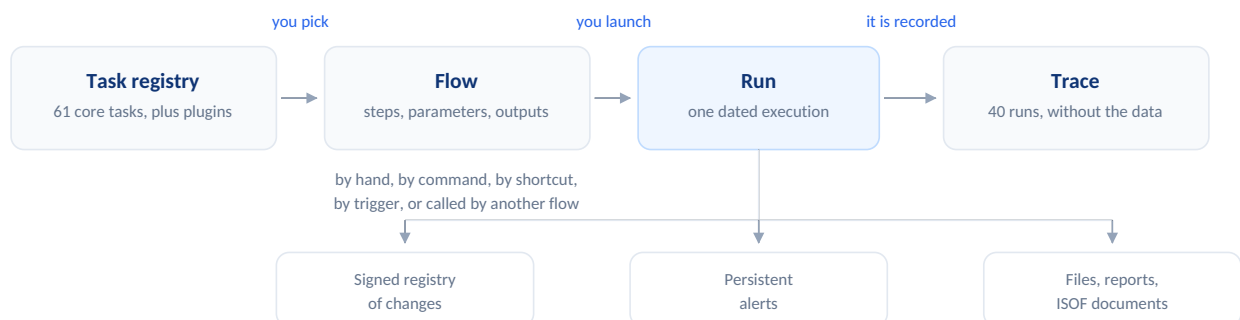
What a workflow is, how it is written, and what you are allowed to rely on when you write one.

1. Mental model

Five words describe the whole system, as long as you keep them apart. Two of them, the flow and the run, carry the most useful distinction: one is a piece of text, the other is a dated event.

Word	What it is	Where it lives
Task	An action declared by the core or by a plugin, with its inputs, its outputs and its write flag. You do not write a task in the interface, you pick one.	In memory, in the task registry, rebuilt every time the page loads.
Flow	What people call a workflow: a series of steps, some parameters, declared outputs, sometimes a trigger. A piece of text that does nothing until you run it.	In local storage, under <code>isofind_workflows</code> .
Step	One link of the flow: a task, its arguments, a condition, a fallback. Its identifier is the address other steps use to read its outputs.	In the flow definition.
Run	One execution of the flow, at a date, with parameters, in a mode. Two runs of the same flow need not look alike.	In memory while it executes, then in the trace, stripped of its data.
Turn	One iteration of <code>core.pour_chaque</code> . Each turn is a complete run of the called flow, with its own trace, folded under its parent.	Same as a run.

The life cycle



An imported flow also goes through the consent screen before its first run, and the list of granted accesses then travels with it, including into the flows it calls.

Three ideas that explain most of the design

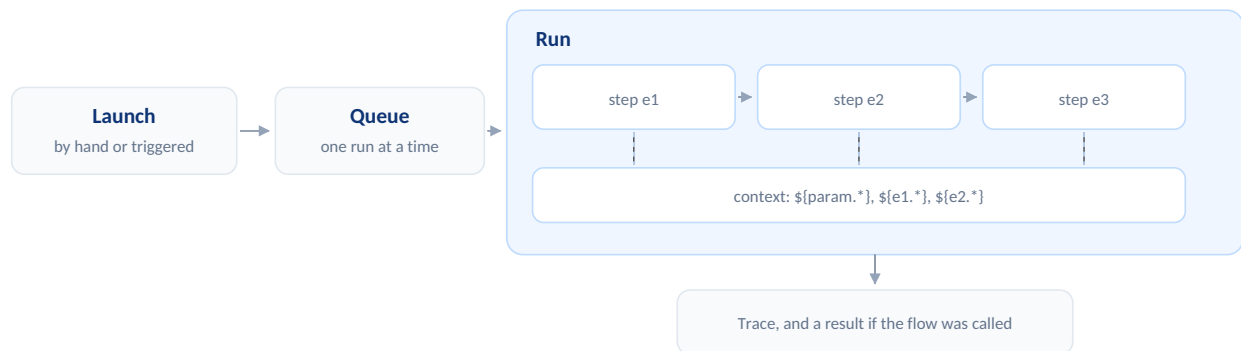
1. **The flow does nothing, the run does everything.** Editing a flow changes no recorded run, and reading a run replays nothing. It is also why the trace keeps the parameters of the run rather than those of the flow at the moment you read it.

2. **The engine trusts neither the model nor a file it received.** What protects is never an instruction written somewhere, but a tool that was not handed over, a capability that was not granted, or a substitution that evaluates nothing.
3. **Whatever runs unwatched has to leave something behind.** That explains the persistent trace, the alerts that survive a reload, the step by step detail in the panel, and the fact that a task relays the real server message rather than a code.

2. The execution model

A task differs from a command on one point that governs everything else: it runs without a screen.

Domain commands open windows. An import asks for a file, a record asks for a form, and that is fine as long as someone is there to answer. An engine chaining them would produce a slideshow of dialogs, one click per step, which would save almost nothing. Tasks are therefore declared separately: they take parameters, they return a result, and nothing obliges them to address anyone while they work.



Steps follow one another in a single direction. Each one adds to the context that later steps read, and none of them can send execution back to a step already passed.

A workflow enters the registry as an agent

The change registry separates the actor, who decides, from the agent, who executes. A write made by a workflow is recorded as `Colin Ferrari, via workflow "Blank correction v3"`, and the mention grows where it should: the trigger that launched it, and the signatory whose machine the flow was written on if it came from elsewhere.

Faced with forty identical corrections, an auditor first asks whether they are looking at a gesture repeated forty times or at an automatic rule, since the two do not carry the same responsibility.

Which tasks make the list

A task enters the registry if it is screenless, deterministic, and if its parameters are enough to describe its effect. Anything that involves a human choice stays a command. You automate carrying out a decision, not the decision itself.

What is missing on purpose

- **Deletion.** A workflow can be received, copied, imported. Giving it the power to erase would mean accepting that a file arriving by email could empty a database. The registry would keep a record of the deletion, which is already a lot, but a record does not replace data.
- **Code execution.** It would make plugin sandboxing decorative, since writing a workflow would be enough to escape it. That is also why writing files stays out of the model's reach: a written file can be a script.
- **Expression evaluation.** Substitution is literal, with no `eval` anywhere, the debugger included.
- **Going back.** The chain is linear, with conditions. Writing a program in a graphical interface produces something you can neither read back, nor test, nor version; when a loop becomes necessary, IsoFind already runs Python and R.

The iteration of `core.pour_chaque` does not walk back that last point. The number of turns is the length of a list known before starting, it is capped, and nothing inside decides to carry on.

3. Anatomy of a workflow

A workflow is a JSON object, kept in local storage under `isofind_workflows` and reloaded at startup.

The smallest workflow worth having

Three things are required: an identifier, a name, and at least one step carrying a task. Everything else has a default.

```
{
  id: "wf-inventory",
  nom: "Inventory of the day",
  etapes: [
    { id: "count", tache: "core.inventaire", args: { par: "element" } },

    { id: "say", tache: "core.alerter",
      args: { message: "${count.nombre} rows in the inventory" } }
  ]
}
```

It runs, it dry runs, it shows up in the palette, and it can take a trigger with nothing added. The `count` identifier on the first step is what makes its output readable by the second: without it the step would still run, and the message would come out empty.

Field names stay in French. They are the keys the engine reads, so `etapes`, `tache` and `parametres` are written that way in a stored flow, whatever language the interface is in.

The full shape

```

{
  id: "wf-lx8k2",           // stable, used as key and as call target
  nom: "Blank correction",   // what the user sees and can call
  description: "...",

  parametres: [             // read as ${param.name} inside steps
    { nom: "element", default: "Sr" }
  ],

  sorties: [                // what it returns to a calling workflow
    { nom: "total", valeur: "${e2.lignes}" }
  ],

  etapes: [
    { id: "e1", tache: "core.sql", args: { sql: "SELECT ..." },
      si: null, continuerSiEchec: false, repli: null }
  ],

  declencheur: {            // declaring is not arming: see « arme »
    type: "evenement" | "horaire",
    evenement: "lims.pull.done",
    si: "evenement.resultat.fichiers",
    heure: "03:00", jours: [1,2,3,4,5],
    simulation: false
  },
  arme: false,              // the second gesture, never imported

  origine: { signataire: "...", valide: true }, // present if imported
  capacites: ["samples:read", "files:write"]    // granted at import
}

```

Every workflow becomes a command. On save, the engine registers `workflow.<id>` in the command registry. The flow then appears in the palette, in the Actions panel and in universal search, and it can take a keyboard shortcut, instead of remaining a setting buried in a menu.

Parameters

Two fields only, a name and a default value. The default matters less for comfort than for automatic launching: a command, a shortcut or a trigger starts with nobody there to answer a dialog, so the workflow would start with empty variables, fail at the first substitution, and the user would blame the workflow rather than the way it was launched.

It also removes copies. "Blank correction" existed in four versions because it differed by one element; a parameter read as `${param.element}` deletes three of them.

When launched by hand, a workflow that declares parameters asks for their values, and one that declares none starts without asking anything. A confirmation before every launch would turn an automation into a form.

Declared outputs

They state what the workflow returns when another one calls it. Without a declaration, the caller receives the outputs of every named step, filed under their identifiers. That works, but it ties the caller to the internal structure of the callee, and renaming a step then breaks a flow you never opened.

The declaration has a second effect, inside loops. A flow that declares its outputs has its results kept turn by turn; without a declaration, only the state of each turn is kept, otherwise five hundred turns would hold all their steps in memory.

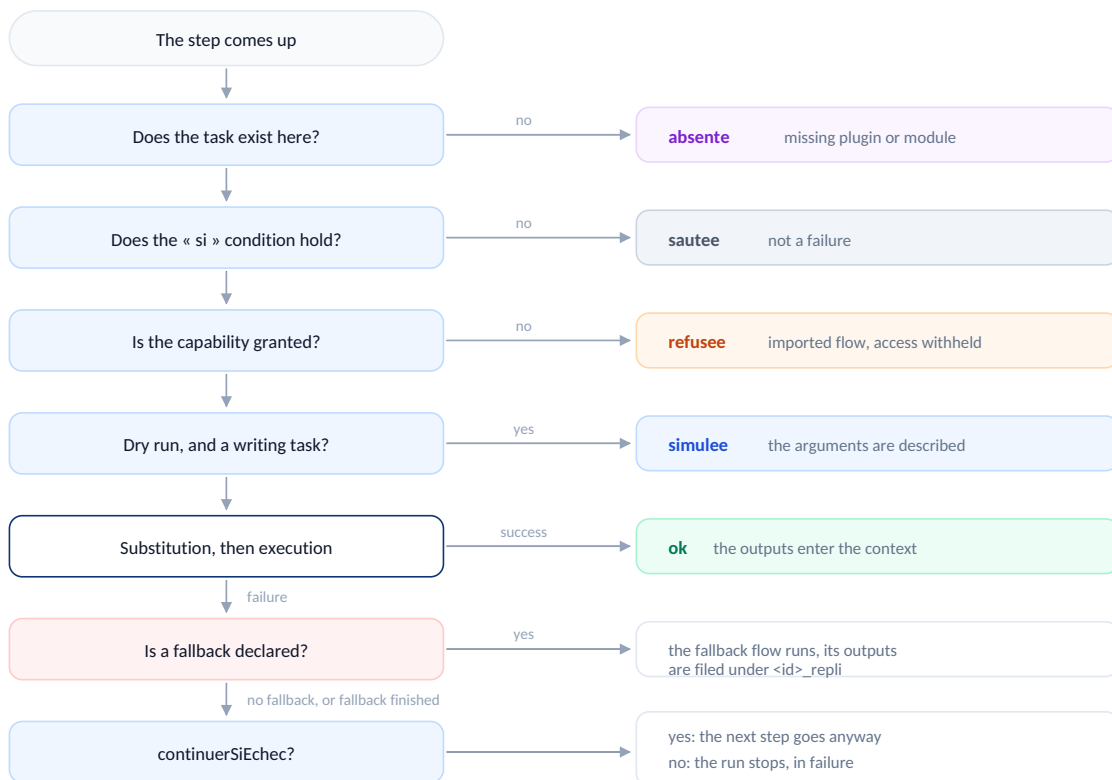
The result is computed only for a called flow. At the root nobody would read it, and it would hold on to the outputs of the forty runs kept in the trace.

4. The step and its settings

The editor knows no task in particular: it derives a step's fields from the `entrees` schema of the chosen task. A task brought by a plugin therefore becomes editable without touching the panel.

Field	Role	Default
<code>id</code>	The name under which the step's outputs become readable, as <code>\${id.output}</code> . A step without an identifier still runs, but leaves nothing behind.	<code>e1, e2, ...</code>
<code>tache</code>	The task identifier. The dropdown comes from the registry and is grouped by capability, which shows, at the moment of choosing, what the step will add to the consent screen.	<code>required</code>
<code>args</code>	One field per declared input. Changing task clears the arguments, otherwise the old content would stay in the file without being shown anywhere.	<code>{}</code>
<code>si</code>	A variable path, not an expression. The step is skipped when the value it names is falsy. To compare two numbers you need a <code>core.condition</code> step upstream and read its <code>vrai</code> output.	<code>null</code>
<code>continuerSiEchec</code>	Decides whether a failure stops the run. It also applies to the <i>absente</i> and <i>refusee</i> states.	<code>false</code>
<code>repli</code>	A workflow run if the step fails, picked from a list. It receives <code>\${param.erreur}</code> , <code>\${param.etape}</code> and <code>\${param.tache}</code> .	<code>null</code>

What the engine decides, in order



The order matters: the capability is checked before the arguments are prepared, so a refused or skipped step never started at all.

The fallback is a workflow, not a jump to another step

Sending a failed step back to an earlier step of the same flow would have introduced a backward branch, which is the loop the engine refuses elsewhere. A fallback workflow keeps the chain linear: it is written apart, read on its own, dry run on its own, and the call stack already prevents it from calling itself.

Its outputs are filed under `<id>_repli`, never under the identifier of the step that gave way. Filed there, a later step reading `#{e2.lignes}` would get the result on some days and the recovery on others, with nothing to tell them apart, and the workflow would look fine until the day it counted wrong.

In the trace, a fallback is not an extra step: it carries the rank of the one it rescues, which keeps "step 3 of 5" true even when three steps had a fallback.

Fallback and carry on are set separately. The first says what to do with the failure, the second whether it stops the run. Merging them would run the rest of the workflow on missing variables every time you only wanted to be told. A failed fallback does not hide the original failure: that one keeps rising.

Input types the editor knows

type	Control shown	Note
texte	Free field	Accepts substitutions
nombre	Free field	Converted by the task

<code>boolean</code>	Checkbox	Substitution returns text, so tasks test both <code>true</code> and <code>'true'</code>
<code>objet</code>	Free field	An object or its JSON text, both accepted
<code>liste</code>	Free field	In practice a reference to an output
<code>sql</code>	Monospaced field	Placeholder <code>SELECT ...</code>
<code>workflow</code>	Dropdown of flows	A free field would give a step that only fails at run time, often at night
<code>commande</code>	Free field	A command identifier

The flow being edited never appears in a workflow dropdown, neither for a fallback nor for a call: the engine would refuse the circular call at run time, and offering it would amount to offering an error.

5. Substitution and context

A step reads the outputs of earlier ones with `${dotted.path}`. Substitution stays literal, and nothing written in a workflow is ever evaluated as code.

A reference on its own, or a reference inside text

When the value is exactly `${e1.lignes}`, the engine returns the value itself. Otherwise an array of three identifiers would arrive as `1,2,3` at a task expecting a list.

When the reference sits inside text, each occurrence is replaced by its textual form, and a missing value leaves an empty string. That is what lets you write an alert message such as `${e2.erreurs} files rejected last night`.

What the context holds

Path	Content
<code>\${param.name}</code>	The run parameters: declared defaults, overridden by the launch variables.
<code>\${e2.lignes}</code>	The outputs of step <code>e2</code> , filed under its identifier rather than merged flat, otherwise two steps both producing <code>lignes</code> would overwrite each other.
<code>\${e2_repli.x}</code>	The outputs of the fallback workflow of step <code>e2</code> .
<code>\${e3.sorties.total}</code>	For a <code>core.workflow</code> step: what the called flow declared it returns.
<code>\${evenement...}</code>	The event payload, for a triggered run.
<code>\${param.element}, \${param.index}, \${param.total}</code>	Inside a loop body: the item of the turn, its rank and the number of turns.

Substitution cases

Assuming step `e1` returned `{ nombre: 238, lignes: [...], colonnes: [...] }` and that the `element` parameter holds `Sr`.

Written in the argument	Received by the task	Note
<code>\${e1.nombre}</code>	<code>238</code>	Reference on its own: the value keeps its type.
<code>\${e1.lignes}</code>	<code>[...]</code>	The array itself, not its textual form.
<code>\${e1}</code>	<code>{ nombre, lignes, colonnes }</code>	The whole output. That is what <code>core.pour_chaque</code> and <code>core.exporter_fichier</code> expect.
<code>\${e1.nombre} rows</code>	<code>"238 rows"</code>	A reference inside text: the result is text.
<code>\${e1.missing}</code>	<code>undefined</code>	No error. The task will complain, or nobody will.
<code>\${e1.missing} rows</code>	<code>" rows"</code>	A missing value leaves an empty string, silently.
<code>\${param.element}</code>	<code>"Sr"</code>	Parameters are text, even when they hold a number.
<code>\${e3.sorties.total}</code>	<code>the declared value</code>	The path walks objects with no depth limit.
<code>e1.nombre</code>	<code>"e1.nombre"</code>	Without the braces nothing is substituted. Except in the <code>si</code> field, which expects a bare path.
<code>\${e1.lignes[0]}</code>	<code>undefined</code>	Bracket indexing does not exist. The path splits on dots, so <code>\${e1.lignes.0}</code> is the first item.

Substitution does not protect what it inserts. A parameter placed inside an SQL query goes in as it is: a quote in the value breaks the query, and the error message will come from the database. For a criterion coming from a parameter, prefer a task that takes that criterion as an argument, such as `core.analyses` or `core.echantillons`, over a hand written query.

Why parameters have their own namespace. A step can be called `element`. Without the `param` prefix, `${element}` would mean its outputs on some days and a parameter on others, depending on which was written first, and an ambiguity settled by writing order always catches someone out.

A reference that names nothing does not fail the step. It produces a missing value, and it is the task that complains, sometimes several steps later. This is the most common failure, and the first thing the debugger is there to diagnose.

6. Contract and implementation

Nothing recompiles here. A saved workflow is text that quotes names, and those names are resolved at launch, sometimes months later. The line between what is promised and what is free to move therefore deserves to be spelled out.

In every task file, the declaration is the contract and the `executer` function is the implementation. That is also why the catalogue in section 15 is produced from the declarations: what you read there is exactly what a flow may rely on.

Object	Contract	Implementation
Task	Its identifier, its <code>ecrit</code> and <code>outil</code> flags, its capability, the name and type of its inputs, the name of its outputs and the meaning of the values they carry.	The <code>executer</code> function, the routes it calls, the shape of the requests, the wording of error messages, the file the task is declared in.
Flow	Its identifier, its name (since a call may use it), its parameters and their defaults, its declared outputs.	Its steps, their order, their identifiers, its conditions and fallbacks, as long as the declared outputs keep their meaning.
Step	Its identifier, quoted by later steps and by breakpoints.	The task used and its arguments.
Engine	Literal substitution, the state names, the linear order, the agent passed to every task, the fact that a write is skipped in a dry run.	The queue, the stack, the debug gate, the timers, the way the trace is stripped down.
Trace	The fields kept for a run: steps, states, messages, durations, output shape, parameters.	Its encoding, its size, the storage key.

What breaks flows already saved

- **Renaming a task output.** Every flow that read `#{e1.old_name}` receives a missing value, with no error at load time, because nothing checks those strings before execution.
- **Making an optional input required.** Existing flows do not supply it. Adding an optional input with a default, on the other hand, breaks nothing.
- **Renaming a flow.** Calls by identifier keep working; calls by name do not.
- **Renaming a step** when the flow does not declare its outputs: the caller was reading the identifier and now finds nothing. Breakpoints set on that step are lost as well.
- **Changing a task's capability.** An imported flow granted the old one sees its steps turn *refusee* until the accesses are reviewed.
- **Turning `ecrit` from false to true.** The task leaves conversation mode and becomes skipped in a dry run, which changes the behaviour of flows that use it without anyone having touched them.

Limits are guard rails, not promises

The caps listed in section 19 protect the machine and the database. They can change, and a flow should not lean on their exact value. A flow counting on five hundred turns to treat a list of four hundred and eighty items is a flow that will fall over the day the list grows, with `tronque` as its only warning.

The line with the server follows the same logic. Routes are the implementation of tasks, and the checks they perform, on allowed roots, visibility and write caps, are the security contract. Nothing is duplicated in the page: a second set of checks would end up diverging, and one of the two would become the door nobody watches.

Running

What happens at launch: the states, the three modes, stepping through, triggers and composition.

7. States

Six step states and four run states. Keeping them apart is what lets you read a trace back without guessing what happened.

Step

State	What happened	Stops the run	What later steps can read
ok	Executed without error.	no	Its outputs, under its identifier.
sautée	The si condition was falsy, or the step was skipped from the debugger. The message then says who decided, since a missing step with no author is a riddle for whoever reads the trace.	no	Nothing.
absente	Task unknown on this installation, usually a flow written for another machine or for a plugin that is not there.	yes, unless <code>continuerSiEchec</code>	Nothing.
refusee	Capability not granted to an imported flow. The refusal is read in the trace next to "skipped" and "absent", rather than as a breakdown.	yes, unless <code>continuerSiEchec</code>	Nothing.
simulee	Write described and not executed. The trace keeps the arguments it would have received, and the panel shows them under that heading rather than as a result.	no	Nothing: the described arguments live in the trace, not in the context.
echec	Error raised by the task, with its real message.	yes, unless <code>continuerSiEchec</code>	Nothing, apart from the fallback outputs under <code><id>_repli</code> .

Four of these six states leave nothing in the context. A later step reading `${e2.lignes}` then gets a missing value, with no error, which explains most of the failures listed in section 18.

Run

State	What it means
-------	---------------

encours	The run is working, or waiting for an order from the debugger.
ok	Every step succeeded, or those that gave way carried <code>continuerSiEchec</code> .
echec	A step gave way and nothing asked to carry on. The run message is the step's.
annule	Stopped by hand, or from the debugger. Writes already under way went through to the end.

A called flow computes its result either way, on success as on failure. With `continuerSiEchec`, the steps that did succeed produced something, and the caller must be able to read it to decide what comes next.

A run also knows how many steps to expect before it starts, since the flow already exists. That is what lets an outside observer, a progress badge or the tracking panel, show progress while it runs rather than afterwards.

Cancelling raises a flag, it does not interrupt. A task in flight that writes to the database finishes its write. Cutting it off midway would leave the database in a state the registry could not describe. A run paused in the debugger would never see that flag, so cancelling also closes the debug session.

8. Run, dry run, debug

Three ways to launch the same workflow. The mode is a launch argument and not a setting of the flow, so the same workflow can be run, dry run or debugged without being modified.

Call	Reads	Writes	Use
<code>executer</code>	executed	executed	Normal launch, by hand or triggered.
<code>simuler</code>	executed	described, not performed	Watch a flow work before arming it.
<code>deboguer</code>	executed	executed	Step by step, stopping before each retained step.

A dry run makes no outputs up

A writing task returns an empty object, so later steps see missing variables. That is deliberate: making up plausible outputs would give a dry run that succeeds where the real execution would fail, and the first real launch would be a leap in the dark all over again.

A workflow called from a dry run is dry run too. Without that, dry running a parent flow would really perform the writes of its children.

One consequence worth knowing: a flow whose read depends on an earlier write will fail in a dry run, and that is the expected behaviour rather than a shortcoming of the dry run.

Arming without a dry run. On software whose provenance is signed, the first real execution had better not be the first execution at all. It is also the moment you notice that a criterion names four thousand rows instead of twelve.

A trigger can itself be set to dry run. An armed flow then works on every event without writing anything, which is a way of watching for a few nights what it would do, alerts included, since `core.alerter` stays active in a dry run and says where its messages came from.

A queue rather than a refusal

One root run works at a time. Refusing the others held up while a human was clicking, since they saw the message and started again. With triggers, two acquisitions fifteen seconds apart would lose the second one, silently, at night. Extra runs therefore wait their turn, in a queue capped at twenty: beyond that, a chatty trigger would turn into a memory leak.

A nested run never goes through the queue and executes inside its parent's turn. Queued, it would wait behind that parent, which is waiting for it to finish.

9. The debugger

A linear chain nearly always fails the same way: an argument arrives empty because `#{e2.lignes}` did not name what you thought. The trace says so afterwards, once the context is gone.

Stepping is obtained by slipping a gate in front of every link of the chain. Outside a session that gate returns `undefined`, the chain does not even take one more turn of the event loop, and the debugger costs nothing while it sleeps.

A session starts paused, before the first step. A "debug" button that behaved like "run" as long as no breakpoint was set would force you to prepare the session before being able to conduct it, when what you want to see first is step 1 leaving.

The orders

Order	Effect	Stepping afterwards
pas	Runs the step and stops at the next one.	armed
sauter	Skips the step without running it. It is marked <i>sautee</i> , with a message crediting the decision to the debugger.	armed
continuer	Lets it run to the next breakpoint.	disarmed
arreter	Cancels the run.	session closed

What a stop shows

The rank and title of the step, its arguments, its condition, the real context of the run, and two things computed before it leaves: whether it writes, and whether it would be refused. Showing them at that moment still leaves the choice of not letting it go. Seeing "access not granted" before taking the step also saves you from looking for the cause in the workflow when it lies in the consent given at import.

The evaluation field only substitutes

It applies the engine's literal substitution to the context of the stop. What it shows is therefore exactly what the step would receive, and nothing typed into it is executed. The rule forbidding evaluation inside a workflow holds for its debugger too.

It takes both forms: a bare path such as `e2.lignes`, and text containing references, such as `step ${param.etape} gave way`. A reference in the middle of a sentence is common in alert messages, and that is precisely where you want to check what will come out.

- **Breakpoints live outside workflows**, in local storage under `isofind_workflow_points`. Filed in the definition, they would leave with an exported workflow and land on a colleague who asked for nothing.
- **The anchor** is the step identifier where there is one, its rank otherwise. A breakpoint set on a step without an identifier therefore follows the rank, and moves at the first insertion.
- **Debugging goes down with the call**. A flow called from a session stops at its first step while its parent waits its turn in the chain, which gives the "step into" of a development environment with nothing added. A called flow inherits the stepping state, but not the initial pause, otherwise a loop would reopen a pause on every turn.
- **One last stop** happens after the final step, otherwise the outputs of that step would never be visible.
- **The context is neither copied nor persisted**. It carries the real outputs, tables of measurements included, and it is detached from the run at the end of the session.
- **Fifteen minutes without a gesture** close the session. A paused run occupies the engine, hence the queue, and a debug session left open before going to lunch would make the afternoon triggers wait without anyone connecting the two.

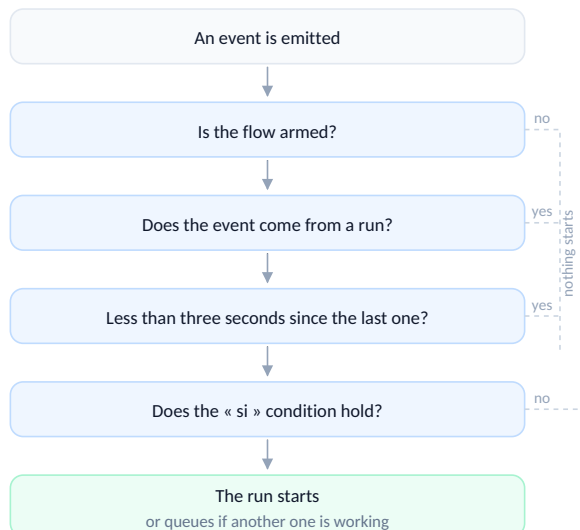
10. Triggers and arming

An acquisition lands at three in the morning. If the blank correction waits for a human to click, it will wait until morning, and the automation will only have saved a form.

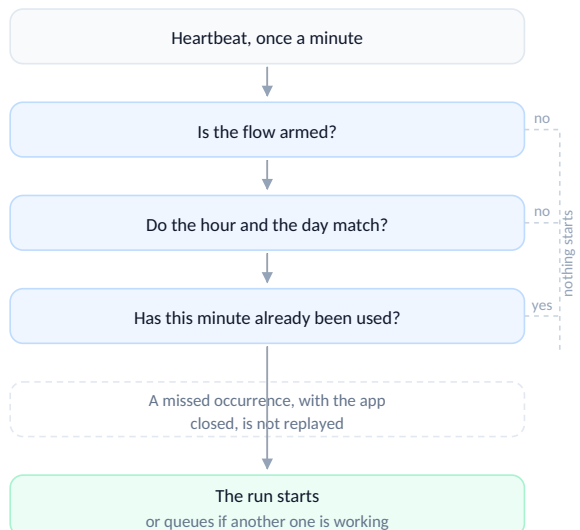
Trigger	When	Readable payload
(none)	By hand, by command, by shortcut.	-
<code>lims.pull.done</code>	After a LIMS acquisition.	<code>\${evenement.resultat...}</code>
<code>lims.pull.error</code>	After a failed acquisition.	<code>\${evenement...}</code>
<code>workflow.finished</code>	After another workflow.	<code>\${evenement.passage...}</code>
<code>@horaire</code>	At a fixed time, on chosen days.	-

The `si` condition applies to the event payload. A LIMS pull that ingested nothing still emits: without a condition, a workflow armed on `lims.pull.done` would set off every five seconds over nothing, and the trace would fill with empty runs. For the schedule, no day ticked means every day, which is the most common case and therefore asks for no input.

On an event



At a fixed time



On the left, the third question reads the other way round: an event emitted by a run is not allowed to relaunch a triggered flow, so there it is "yes" that stops everything. On the right, the current minute is the key that keeps a slot from firing twice.

Three locks

Arming is explicit, flow by flow, and never imported. Declaring a trigger does not arm it: you have to say so a second time, and that second time carries the word "will write" when the flow contains a write. Setting the trigger back to "none" disarms the flow in the same gesture.

A triggered workflow cannot be re-triggered by the events its own run emits. Without that, a flow listening to `workflow.finished` would call itself endlessly, at night, writing into a signed database.

A three second cooldown separates two triggers of the same flow. The LIMS ingests file by file and each one emits, so ten files dropped together would otherwise launch ten concurrent runs over the same data.

Missed occurrences are not replayed. The application is closed in the evening; on Monday morning an engine that caught up would launch the seven runs of the past week at once, all writing to the database. The heartbeat is one minute, and the current minute serves as a key so that one slot does not fire twice.

The alert, last step of an unwatched workflow

A step that fails writes to the console and to the journal, which nobody reads in the morning. `core.alerter` leaves instead a message that survives a reload and comes back in the panel at the next startup. The alert carries the workflow name, its trigger, and whether it came from a dry run, since "3 files rejected" left by a dry run does not describe the state of the database.

Three levels exist, `info`, `attention` and `erreur`. The last two also raise an on screen notification when somebody is there to see it, without replacing the persistent alert, which is the only one that survives the night.

11. Composition and iteration

Composition keeps a twenty step workflow from being copied four times because it differs by one. It reopens the loop, which the stack and the depth limit close again.

core.workflow

Runs another workflow and waits for it. What it produced comes back under the `sorties` key rather than merged into the returned object: a flow declaring an output named `etat` would otherwise overwrite the run state, and the breakdown would show up at the next step, far from its cause. The target can be written either as an identifier or as a name, since an identifier like `wf-lx8k2` is not what a user remembers.

core.pour_chaque

Repeats a workflow over each item of a list. The list comes from an earlier step and is passed whole: the task knows how to find its rows, whether they are called `lignes`, `rows`, `echantillons`, `trouves` or `entrees`. An output that files its items under another name is passed directly, for instance `${e1.ids}`, since a plain array is accepted as is.

- Each turn runs the workflow with the item as a parameter, under the name given by `variable`, and also passes `index` and `total`.
- When the columns are known, the row arrives named. The body reads `${param.element.name}` instead of a position, which would shift the first time a column is added to the query.
- Parameters shared by every turn can be passed separately, which saves repeating them in the body.
- Turns run in series. In parallel, two turns writing at the same time into the same database would tread on each other.
- The body is a workflow, not a block of nested steps. A block would have required a two level editor and a two level trace, where a workflow is read on its own, run on its own to try it on one item, and reused elsewhere.
- A failure stops the loop, unless `continuerSiEchec`. The message says which turn gave way and how many had succeeded before it, because a loop that never started and a loop that fell over on the fortieth item are not picked up the same way.

Two caps, and they do not say the same thing. The one the author sets (`max`, fifty by default) bounds their intention. The engine's, five hundred, bounds the damage, including when the list comes from a `core.sql` whose criterion nobody read back. Beyond it, remaining items are left and `tronque` says so.

What closes the loop again

The engine keeps a stack of the flows in flight, root included, and refuses a circular call, direct or through a third flow. While the stack was pushed only by the nested caller, a flow calling itself got through on the first turn and was refused only on the second, once two writes had been made.

Nesting stops at three levels. A loop body that itself calls a flow therefore uses two of those three.

Governing

What a flow is allowed to touch, how you grant it, and what the log keeps of it.

12. Capabilities and consent

A signed document says who wrote the workflow. It does not say what that workflow touches, and that is the other question.

While a flow could read a database and write a CSV, the list of steps shown at import served as an answer. Now that it composes, iterates and falls back on another flow, that list can no longer be taken in at a glance, and a `core.pour_chaque` step shows nothing of what its loop body does. The question is therefore reduced to a handful of named accesses, computed by walking down into the called flows, and granted once and for all.

The vocabulary is the plugins' own. `samples:read`, `files:write` and `ui:notify` are the words a plugin already declares in its manifest, and that the user has already seen in the plugin consent screen. A second set of names would mean learning the same thing twice, and wondering, in front of `db:write`, whether it means `samples:write`.

The full vocabulary

Capability	What it grants	Trace
<code>samples:read</code>	Read the database: samples, analyses, measurements	<code>read</code>
<code>samples:write</code>	Write to the database, under your name, in the signed registry	<code>writes</code>
<code>files:read</code>	Read files, within the declared roots	<code>read</code>
<code>files:write</code>	Write files to disk	<code>writes</code>
<code>lims:read</code>	See acquisition sources and their log	<code>read</code>
<code>lims:write</code>	Ingest an acquisition into the database	<code>writes</code>
<code>methods:write</code>	Store methods in the library	<code>writes</code>
<code>campaigns:read</code>	See declared campaigns and their aliquot chains	<code>read</code>
<code>campaigns:write</code>	Write to a campaign descriptor, which is authoritative for its provenance	<code>writes</code>
<code>sites:read</code>	Read prospection sites, their boreholes and interpretations	<code>read</code>
<code>sites:solve</code>	Run a site's solvers: flow, gradient, profile	<code>read</code>
<code>nexus:run</code>	Query the Nexus engine and its reference databases	<code>read</code>
<code>notebook:run</code>	Run a saved notebook: the script does whatever its kernel allows	<code>writes</code>

<code>report:write</code>	Produce a report or an ISOF file	writes
<code>ui:notify</code>	Leave an alert you will see at next startup	read
<code>ui:command</code>	Open any IsoFind window	writes
<code>workflow:call</code>	Run other workflows, and repeat one over a list	writes

The `ecrit` flag does not mean the capability touches the database, but that it leaves a trace outside the run. It is what decides whether the box is ticked in advance, and it matches the only distinction you can ask someone to make in three seconds. `sites:solve` modifies nothing but costs machine time by the tens of seconds, and a consent screen that did not tell reading a site from launching a coupled solve would not say much about what is being granted.

Three Nexus tasks are filed against the look of their name: those that search the samples fall under `samples:read` rather than `nexus:run`, otherwise a flow never granted database reads would get there by that route.

From the task to the capability

1. The task declares its `capacite` field, which wins. That is how a plugin files its own inside the shared vocabulary.
2. Otherwise, the core table, gathered in a single file so that "what can a workflow touch" can be answered without opening fifteen. Tasks that ask for nothing appear there with a null capability, so that the omission reads as a choice.
3. Otherwise, a capability built from the task's source, `<source>:read` or `<source>:write` depending on its flag. It will show up as such on the screen, and an obscure line is better than an access granted without saying so.

A capability outside the vocabulary is treated as writing: in doubt, one box too many to tick costs less than an access granted in silence.

What the computation cannot know

The walk goes down into called flows and into fallbacks, four levels deep. A fallback is a workflow too, and what it touches is granted on the same footing, otherwise making a step fail would be enough to open a door nobody looked at. Two blind spots remain, and the screen announces them:

- **Unknown flows.** The workflow calls others that do not exist here. Those steps will fail, and their accesses could not be computed.
- **Undecidable target.** A call names its target through a substitution, known only at run time. The engine will then refuse any step outside the granted accesses.

Saying so matters as much as the computation, because a permission screen silent about its blind spots gives the certainty of having seen everything.

Who is constrained, and how far

A workflow written here has nothing to ask, since whoever launches it is whoever wrote it. An imported workflow carries an origin, and the engine then lets it run only the tasks whose capability was granted.

The constraint travels through the whole call tree, otherwise an imported flow would grant itself by composition what it was not given. A local flow called by an imported one therefore inherits the same list.

- At import, reads are ticked in advance and writes are not. As with arming, whatever commits you is said twice.
- The consent screen is not a window of its own: it is a block placed inside the import window, since stacking a second window would lose sight of what is being consented to.
- Accesses can be reviewed afterwards, without reimporting. Withdrawing three weeks later an access granted in front of a screen you were reading for the first time is a normal gesture; forbidding it would push people to delete the workflow and import it again, which would give it a fresh identifier and lose its history.
- The capability is checked before the arguments are substituted, so before any preparation takes place.
- The list of accesses a run worked under is kept with it in the trace. It is the first thing you look at in front of a change made by an automation received from elsewhere.
- When the capability registry is not loaded on the page, reads go through and writes are withheld. A missing security module must not amount to an authorisation, nor stall a page that only consults.

13. The access log

Who reaches the samples, and under what name. The log is written by the server; the page supplies the name.

Every task file calls the network its own way. Placing the header in all fifteen would mean maintaining fifteen places that have to agree, and the first file added afterwards would forget it with nothing to signal it, when an undeclared access looks exactly like an access that never happened. A single wrapper is therefore installed, and the scope that says which task to attribute a request to comes from the registry. A task brought by a plugin goes through the same door, so it is covered without its author having to think about it.

Header	Content
X-IsoFind-Agent-Kind	workflow, plugin, or the kind declared by the caller
X-IsoFind-Agent-Id	The flow identifier, which serves as the join key
X-IsoFind-Agent-Nom	The readable name, which can change without breaking anything
X-IsoFind-Capability	The capability of the running task
X-IsoFind-Run	The run reference
X-IsoFind-Task	The task identifier
X-IsoFind-Ambiguous	Set when two runs interleave

The identifier travels, the name comes along. Filed under the name, the log would lose the link at the first renamed workflow, and its old lines would point to a flow that no longer exists under that label.

Attribution is declarative, and it flags its doubts

The engine runs in the page, so these headers count as a declaration and not as proof, and the server treats them that way. When two simultaneous runs interleave their requests, the wrapper marks the ambiguity instead of attributing at random, and the log entry carries that reservation.

- **Nothing is refused here.** Refusing revoked accesses belongs to the server, which alone knows what has been revoked, and consenting to capabilities stays the engine's business.
- **Outbound requests get nothing.** These headers would trigger a CORS preflight on map tiles and geocoding, which would refuse it, and the declaration would break what it observes.
- **The scope holds until the promise settles,** not only during the synchronous part. A task chaining two requests would otherwise lose its name from the second one.
- **Consumers outside workflows name themselves too,** through an explicitly opened scope: the plugin bridge, which alone knows which plugin it is serving, and the notebook.
- **A stamp that cannot be applied does not stop the request.** The call leaves without a name, and the server records it as undeclared.

14. The panel and the commands

The Workflows panel lists, launches, and shows what happened. It sits in the left dock and has two tabs, the workflows and the runs.

Searching a long list

The filter matches the name, ignoring accents and case, and recognises a few state words, because the two questions you really put to a long list are not asked by name.

Tab	Words recognised
Workflows	<code>armed</code> , <code>imported</code> , <code>writes</code> and their French equivalents
Runs	The state codes as they are: <code>ok</code> , <code>echec</code> , <code>annule</code> , and <code>simulation</code>

The filter clears when you change tab, otherwise an "echec" kept on the way to the workflows would hide everything with no visible reason. Escape clears it without closing the panel. The counter only shows while a filter is doing something, as "4 of 12".

Armed flows rise to the top of the list: they are the only ones that leave without being watched, so the only ones whose state matters when you open the panel for no particular reason.

One gesture per line

Only "Run" stays on the line. Dry run, export, duplicate and delete live in the context menu, which knows how to mark what writes and grey out what is not possible. Four permanent links per line made four targets to aim at for gestures of which only one is frequent.

An imported flow carries a mark, with its signatory and the state of its signature, and gives access to reviewing its accesses. Importing is done through the arrow at the top of the panel, when the signed document module is present.

Reading a run

- The summary folds, and the whole line acts as the button.
- Each step shows its state, its duration, and its message where there is one.
- Outputs unfold under a summary that stands on its own: "11 rows" says what there is to see, where "outputs" makes you open it to find out whether there was anything.
- A table copies in spreadsheet format or exports as CSV, with all its rows and not the fifty on display. Column names stay as they are in the database in the file, even though the table shows them in plain form, so that an export can be matched back to a query.
- A dry run step shows the arguments it would have received, not a result.
- A trace read back after a reload shows the shape of the outputs, "238 rows, data not kept", rather than a silent step.
- The turns of a loop are folded under their parent, which announces how many there are. A run whose parent has dropped out of the trace rises to the root and says so, rather than disappearing.

The alert banner. Unread alerts appear at the top of the panel. The cross hides them for the current session without marking them read, and a fresh alert reopens the banner: having dismissed it at ten should not hide the failure of eleven. "Mark as read" remains the only gesture that clears them.

The commands

The panel listens to no keystroke. These are commands, so they enter the palette, the Actions panel and universal search, and each can take whatever shortcut the user wants.

Command	Effect
<code>core.workflows.open</code>	Open the panel
<code>core.workflows.new</code>	New workflow
<code>core.workflows.run</code>	Run the workflow chosen in the title bar
<code>core.workflows.dryrun</code>	Dry run it
<code>core.workflows.debug</code>	Debug it, opening the debug panel
<code>core.workflows.replay</code>	Replay the last root run, with its parameters

<code>core.workflows.stop</code>	Stop the current run
<code>core.workflows.filter</code>	Open the panel and focus the filter field
<code>workflow.<id></code>	One per saved workflow

Replay takes the parameters of the run rather than the defaults of the flow, and stays a dry run if the run it replays was one. Replaying with other values would amount to doing something else while claiming to reproduce.

Reference

The task catalogue, workflows to copy, troubleshooting, the limits, and adding a task.

15. Task catalogue

The sixty-one tasks shipped with the core, grouped by what they touch. This list stands in for architecture: a missing task is something nobody will be able to automate, and one task too many is a door that will have to be defended.

In each row, an argument followed by a star is required, its type follows in grey, and a default value appears in green. **writes** marks a task that modifies something, therefore absent from conversation mode and skipped in a dry run. **hidden** marks a task the assistant sees in no mode at all.

No access needed **null**

They do not reach outside their own run: nothing to ask for, and the omission is a choice.

Task	Inputs	Outputs	Flags
core.calcul Apply a geochemical formula	formule* texte parametres* texte element texte sql sql corrigees boolean = false	resultat nombre incertitude nombre lignes liste colonnes liste	read
core.carnet_noyaux Available notebook kernels	none	noyaux liste nombre nombre	read
core.condition Condition	valeur* texte operateur* texte = > reference* texte	vrai boolean	read hidden
core.formules Geochemical formula catalogue	domaine texte formule texte	formules liste constantes liste nombre nombre	read
core.journal Write to journal	message* texte	none	read hidden

Read the database **samples: read**

Task	Inputs	Outputs	Flags
core.analyses Analyses by element	element* texte limite nombre = 500 corrigees boolean = false	lignes liste colonnes liste nombre nombre	read
core.inventaire Database inventory	par texte = element	lignes liste colonnes liste graphique objet nombre nombre	read
core.schema Database schema	base texte = isofind table texte	bases liste tables liste	read

	colonne	texte	
core.derivees Derived values of a sample	sample_id* nombre actives boolean = false	derivees liste nombre nombre	read
core.graphique Draw a chart	sql sql element texte type texte = barres x texte y texte erreur texte couleur texte agreger texte par texte titre texte titre_x texte titre_y texte corrigees boolean = false	graphique objet lignes liste colonnes liste nombre nombre	read
core.correspondance Geographic matching	echantillon* texte seuil nombre	lignes liste colonnes liste nombre nombre	read
core.methodes Method library	type texte element texte	methodes liste nombre nombre	read
core.nexus_authenticite Nexus: authenticity check	element* texte = Sb delta* nombre incertitude nombre = 0.05 origine_declaree texte categorie texte seuil_confiance nombre = 0.8	score nombre verdict texte references liste	read
core.nexus_correspondance Nexus: quick match	element* texte = Sb delta* nombre incertitude nombre = 0.05 gabarit texte materiau texte seuil_sigma nombre = 3 conditions objet molecule texte	matches liste transformed_signature objet steps liste	read
core.nexus_tracabilite Nexus: trace a process	element* texte = Sb delta* nombre incertitude nombre = 0.05 procede texte materiau texte seuil_sigma nombre = 3 molecule texte	matches liste inversion_step objet method texte	read
core.rapport_modeles Report templates	none	lignes liste colonnes liste nombre nombre	read
core.sql SQL query	base* texte = isofind sql* sql	lignes liste colonnes liste nombre nombre	read
core.echantillon_detail Sample detail	nom* texte corrigees boolean = false	echantillon objet analyses liste	read
core.carte Sample map	echantillon* texte	carte objet	read

core.echantillons Select samples	filtre texte limite nombre = 100	ids liste noms liste nombre nombre	read
core.resume Summary statistics	element texte sql sql par texte sur texte corrigees boolean = false	lignes liste colonnes liste nombre nombre	read
core.verifier_chaine Verify a sample chain	id* nombre	valide boolean entrees nombre raison texte	read

Campaigns and aliquot chains, read [campaigns:read](#)

Task	Inputs	Outputs	Flags
core.chaine Aliquot chain	path* texte	chaine objet resume objet orphelins liste integrite_resume objet	read
core.campagnes Declared campaigns	nom texte projet_id texte	campagnes liste nombre nombre	read
core.provenance_recit Everything that led to this node	path* texte noeud* texte	etapes liste valeur texte repris_dans liste noeud texte	read
core.provenance_resoudre Find a sample across campaigns	terme* texte path texte	trouves liste terme texte	read
core.provenance_comparer What differs between two lineages	a* objet b* objet	differences liste etapes_identiques liste ecart_valeurs objet	read

Prospection sites, read [sites:read](#)

Task	Inputs	Outputs	Flags
core.sites Prospection sites	nom texte	sites liste nombre nombre	read
core.site_matrice Units and ratios of a site	site_id* nombre ratios texte interpretation_id nombre	units liste ratios liste	read

Site solvers [sites:solve](#)

They modify nothing, but cost machine time by the tens of seconds.

Task	Inputs	Outputs	Flags
core.gradient_isotopique Interpolated isotope gradient	site_id* nombre forages* liste ratio* texte interpretation_id nombre	grid objet ratio_key texte	read

```

nx nombre = 60
nz nombre = 40
anisotropie nombre = 5
puissance nombre = 2
distance_max nombre = 30
contact_permeabilite nombre
= 0

```

core.profil_isotopique Isotope profile of a transect	site_id* nombre forages* liste ratio texte interpretation_id nombre	boreholes liste layer_stats liste	read
core.flux_simuler Simulate flow	site_id* nombre forages* liste interpretation_id nombre nx nombre = 80 nz nombre = 60 vadose booleen = false recharge nombre = 0	head objet velocity objet grid objet	read

Nexus engine [nexus:run](#)

Task	Inputs	Outputs	Flags
core.nexus_gabarits Nexus: fractionation templates	none	templates liste	read
core.nexus_processus Nexus: identify a process	element* texte = Sb delta_initial* nombre delta_final* nombre incertitude_initiale nombre = 0.05 incertitude_finale nombre = 0.05 conditions objet	processes liste delta_change nombre	read
core.nexus_serie Nexus: monitor a series	element* texte = Sb mesures* liste site texte fenetre nombre = 3 sensibilite texte = medium	breakpoints liste trend objet statistics objet	read
core.nexus_demixage Nexus: source unmixing	element* texte = Sb delta_melange* nombre incertitude nombre = 0.05 sources* liste methode texte = nnls	fractions liste method texte	read

Read files [files:read](#)

Bounded to the declared roots, with paths resolved before comparison.

Task	Inputs	Outputs	Flags
core.fichier_lister List a folder	chemin texte	entrees liste nombre nombre	read
core.fichier_lire Read a file	chemin* texte	contenu texte lignes nombre	read

Acquisition, read `lims:read`

Task	Inputs	Outputs	Flags
<code>core.lims_evenements</code> Acquisition log	statut texte limite nombre = 40	evenements liste nombre nombre erreurs nombre	read
<code>core.lims_sources</code> Acquisition sources	none	sources liste nombre nombre actives nombre	read
<code>core.lims_sonder</code> Probe a source	source* texte	ok booleen message texte exemples liste	read

Write to the database `samples:write`

Task	Inputs	Outputs	Flags
<code>core.echantillon_creer</code> Create a sample	nom* texte materiau texte secteur texte lieu texte date texte description texte classification texte projet texte latitude nombre longitude nombre isotopes liste	sample_id nombre isotopes nombre	writes hidden
<code>core.enregistrer_derivee</code> Record a derived value	donnees* objet motif* texte nature texte = correction lignes_max nombre = 100 colonne texte	enregistrees nombre ignorees nombre sample_ids liste	writes hidden
<code>core.physico_enregistrer</code> Record a physico-chemical parameter	sample_id* nombre parametre* texte valeur* nombre incertitude nombre methode texte mesure_le texte notes texte	id nombre parametre texte	writes hidden
<code>core.rendement_enregistrer</code> Record a purification yield	sample_id* nombre element* texte rendement* nombre opérateur texte methode texte	element texte rendement nombre	writes hidden
<code>core.geochimie_enregistrer</code> Record concentrations	sample_id* nombre donnees* liste remplacer booleen = false	enregistrees nombre ids liste	writes hidden
<code>core.echantillon_modifier</code> Update a sample	sample_id* nombre champ texte valeur texte champs objet	sample_id nombre modifies liste	writes hidden

Campaigns write `campaigns:write`

Campaigns, write `campaigns:write`

A campaign descriptor is authoritative for its provenance.

Task	Inputs	Outputs	Flags
core.chaine_noeud Add a chain node	path* texte nom* texte type texte = aliquote quantite texte valeur texte reference texte fichiers liste note texte	noeud objet resume objet	writes hidden
core.chaine_rattacher Attach a file to the chain	path* texte id* texte chemin* texte repere texte detacher boolean = false	fait boolean chemin texte	writes hidden
core.campagne_declarer Declare a campaign	path* texte nom texte projet_id texte	racine texte fichiers nombre signalements liste	writes hidden
core.chaine_arete Link two nodes	path* texte de* texte vers* texte operation texte = aliquotage protocole texte fichiers liste note texte	arete objet resume objet	writes hidden

Method library `methods:write`

Task	Inputs	Outputs	Flags
core.methode_enregistrer Record a method	cle* texte nom* texte type* texte = purification element texte protocole objet materiel texte laboratoire texte auteur texte version texte = 1.0 doi texte reference texte	id nombre cle texte cree boolean	writes hidden

Write files `files:write`

Two formats, and the extension is imposed by the format.

Task	Inputs	Outputs	Flags
core.exporter_fichier Export to a file	donnees* objet fichier* texte format texte = csv horodater boolean = true	fichier texte lignes nombre octets nombre	writes hidden

Acquisition, ingest `files:write`

Acquisition, ingest `lims:write`

Task	Inputs	Outputs	Flags
<code>core.lims_tirer</code> Pull an acquisition	source texte	fichiers nombre echantillons nombre lignes nombre neuf booleen	writes

Reports and ISOF `report:write`

Task	Inputs	Outputs	Flags
<code>core.export_isof</code> Export to ISOF	ids* liste signer booleen = true provenance texte = complet	document objet echantillons nombre	writes
<code>core.rapport</code> Generate a report	modele* nombre echantillons texte titre texte projet texte	rapport objet	writes

Notebooks `notebook:run`

Task	Inputs	Outputs	Flags
<code>core.carnet_executer</code> Run a notebook	chemin* texte noyau texte continuerSiEchec booleen = false	cellules nombre reussies nombre sortie texte resultat objet	writes hidden

Alerting `ui:notify`

Task	Inputs	Outputs	Flags
<code>core.alerter</code> Raise an alert	message* texte niveau texte = info	posee booleen	read

Open a window `ui:command`

Task	Inputs	Outputs	Flags
<code>core.commande</code> Run a command	id* commande	none	writes

Compose and repeat `workflow:call`

Task	Inputs	Outputs	Flags
<code>core.pour_chaque</code> For each item	liste* objet workflow* workflow variable texte = element max nombre = 50 continuerSiEchec booleen = false	tours nombre reussis nombre echoues nombre resultats liste tronque booleen	writes

	parametres objet		
<code>core.workflow</code> Run a workflow	<code>workflow*</code> workflow <code>parametres</code> objet	etat texte etapes nombre duree nombre sorties objet	writes

The capability table also lists `core.contexte`, `core.doc`, `core.workflow_detail`, `core.workflow_liste`, `core.workflow_passages`, `core.workflow_simuler`, declared outside the task files above. They add no access, except `core.workflow_simuler`, filed under `workflow:call`.

16. What the assistant sees

Two flags carve out three surfaces. They are not documentation: they decide what the model can call.

Surface	What it receives
Conversation mode	Tasks where <code>ecrit</code> is false and <code>outil</code> is not false.
Agent mode	Tasks where <code>outil</code> is not false, writes included.
Workflows only	Tasks marked <code>outil: false</code> , editable inside a plan a human has read and adopted, invisible to the model.

Taking a tool away is not up for discussion, asking is

Telling the model to modify nothing would amount to a request, which a model that hallucinates, or that poisoned data manipulates, could ignore. Taking away the tools that write puts it in a position where the question no longer arises.

A task whose flag is missing is treated as writing, otherwise a plugin that forgets to declare it would have its task exposed to the least watched mode.

`outil: false` covers two different situations. Chaining pieces such as `core.condition` or `core.journal` answer no question a user asks, and a model that saw them would try them, since it tries what it is shown. Domain writes such as `core.echantillon_creer`, `core.enregistrer_derivee` or `core.exporter_fichier` are withheld for another reason: writing to a signed database because a human read a plan and writing because a model decided so mid sentence are two different things, and that is what plan mode is there to separate.

Two tasks are marked as writing without writing anything themselves: `core.workflow` and `core.pour_chaque`. The flow they launch may write, and they do not know what it does. Declaring them as reads would open, in conversation mode, a route to every write the model cannot call directly, multiplied by the length of a list in the second case.

The notice

Every task exposed to the model carries a bilingual notice saying when to call it, when to leave it alone, and how to read an empty answer. Four instructions recur:

- **Finding nothing is an answer.** Saying so keeps the model from looking elsewhere for an explanation nothing supports.
- **Identifiers are not invented.** The notice names the task that returns them and asks for them to be reused verbatim: a node identifier such as `n17` means nothing outside its campaign.
- **Computation belongs to the code.** The model picks the formula, the code applies it, propagates the uncertainty and returns the reference. A formula applied from memory is a formula whose version nobody will know, and "Rayleigh" names four of them.
- **A failure has to be learnable.** Tasks relay the real server message rather than a code: with "no such column: element" the model corrects itself; with "HTTP 500" it asks the same thing again or gives up. The SQL task goes further and returns the list of real tables when a query fails, which fixes the most common mistake, that of inventing a name.

17. Four workflows to copy

Four shapes that keep coming back, written here in their stored form rather than described. Column names and numeric values are to be replaced by those of your database.

Night acquisition, with a report in the morning

The probe at the front is what separates a useful test from a green light: ingesting from an unreachable source produces nothing, but ingesting from a half synced folder produces wrong data.

```
{
  nom: "Night acquisition",
  declencheur: { type: "horaire", heure: "03:00", jours: [] },
  arme: true,
  etapes: [
    { id: "probe", tache: "core.lims_sonder",
      args: { source: "neptune" } },
    { id: "pull", tache: "core.lims_tirer",
      args: { source: "neptune" }, si: "probe.ok" },
    { id: "rejects", tache: "core.lims_evenements",
      args: { statut: "erreur" }, continuerSiEchec: true },
    { id: "alert", tache: "core.alerter",
      args: { message: "${rejects.erreurs} file(s) rejected, "
              + "${pull.echantillons} sample(s) came in",
              niveau: "attention" },
      si: "rejects.erreurs" }
  ]
}
```

The pull is idempotent: running it again ingests nothing twice. All zeros means there was nothing new, not that the acquisition failed, which is precisely why the `rejects` step is there.

Parameterised correction, with a derived write

The computation is done in code, with uncertainty propagated, and the correction is added to the measurement instead of replacing it. The motive is required, and it travels with the value into the registry.

```
{
  nom: "Blank correction",
  parametres: [
    { nom: "element",      default: "Sr" },
    { nom: "delta_blanc",  default: "-0.8" },
    { nom: "q_blanc",      default: "0.12" }
  ],
  etapes: [
    { id: "calc", tache: "core.calcul",
      args: {
        formule: "correction_blanc",          // id returned by core.formules
        element: "${param.element}",          // the source, hence the @ columns
        parametres: "delta_mesure=@ratio_value; q_mesure=@amount; "
                    + "delta_blanc=${param.delta_blanc}; q_blanc=${param.q_blanc}"
      } },
    { id: "store", tache: "core.enregistrer_derivee",
      args: { donnees: "${calc}",              // the whole output, not one column
              motif: "Session blank, ${param.element}",
              nature: "blanc", lignes_max: 200 } }
  ],
  sorties: [ { nom: "corrected", valeur: "${store.enregistrees}" } ]
}
```

Names prefixed with `@` point at columns of the source and must exist there. The output of the computation carries the measurement identifier, the corrected value and its uncertainty, which lets the next step attach each value to the right measurement without the flow having to name a column.

Repeating over a selection

The body is an ordinary workflow, which you can run on its own against one sample to try it before putting it in a loop.

```
// The calling flow
{
  nom: "Check a selection",
  parametres: [ { nom: "filtre", default: "sediment" } ],
  etapes: [
    { id: "list", tache: "core.echantillons",
      args: { filtre: "${param.filtre}", limite: 200 } },
    { id: "loop", tache: "core.pour_chaque",
      args: { liste: "${list.ids}",           // a plain array, accepted as is
    }
```

```

        workflow: "Check one sample",
        variable: "id", max: 200, continuerSiEchec: true } },

    { id: "report", tache: "core.alerter",
      args: { message: "${loop.echoues} check(s) failed out of ${loop.tours}" },
      si: "loop.echoues" }
  ]
}

// The loop body
{
  nom: "Check one sample",
  etapes: [
    { id: "chain", tache: "core.verifier_chaine",
      args: { id: "${param.id}" } }
  ],
  sorties: [
    { nom: "valide", valeur: "${chain.valide}" },
    { nom: "raison", valeur: "${chain.raison}" }
  ]
}

```

A negative check is not a failure. `core.verifier_chaine` returns `valide: false` without raising an error, so `echoues` counts only the turns that really gave way. To count invalid chains, read the loop's `resultats`, which carry the declared outputs of each turn.

Weekly summary dropped on disk

A workflow that computes without putting anything anywhere computes for nobody: the table lives in the memory of a page that will be closed.

```

{
  nom: "Weekly summary",
  declencheur: { type: "horaire", heure: "07:00", jours: [1] },
  arme: true,
  etapes: [
    { id: "summary", tache: "core.resume",
      args: { element: "Sr", par: "material_type" } },

    { id: "file", tache: "core.exporter_fichier",
      args: { donnees: "${summary}", fichier: "reports/sr_summary",
        format: "csv", horodater: true } },

    { id: "say", tache: "core.alerter",
      args: { message: "Summary written: ${file.fichier}" } }
  ]
}

```

The extension is imposed by the format: a name ending in anything else is rebuilt, with the folder left intact. The timestamp keeps one run from overwriting the day before's, and is switched off explicitly when someone else reads the same file at the same place every time. A criterion that finds nothing produces a file with its headers only, because a missing file cannot be told from a breakdown in the morning.

18. Troubleshooting

Common failures, and where to start. Most of them show up in a minute with the debugger, stopping on the step before the one that fails.

Symptom	Likely cause	Where to look
An argument arrives empty, or the task complains about missing data	The reference names a step without an identifier, or an output that does not carry that name	The debugger's evaluation field, stopped on that step. The catalogue gives the exact output names.
Step <i>absente</i>	Task not declared on this machine: missing plugin, or a module the edition does not mount	The task list in the editor. Campaign tasks, for instance, are only declared if the edition mounts the module.
Step <i>refusee</i>	Capability not granted to an imported flow, or a task's capability changed since import	Review the accesses, from the flow's context menu.
The loop leaves items behind	A cap was reached, either the one set or the engine's	The step's <code>tronque</code> output.
The triggered workflow does not start	Not armed, condition falsy, three second cooldown, a triggered run already working, or an open debug session holding the engine	The Runs tab, and the arming box in the editor.
A schedule was not honoured	The application was closed at that time, and missed occurrences are not replayed	Nothing to fix: this is the intended behaviour.
The outputs vanished after a reload	They are not persisted, only their shape is	The "data not kept" line under the step. Running it again returns them.
The dry run fails where the real run would pass	A read depends on an earlier write, which was skipped and returned nothing	Expected. Check the reading steps separately.
"Circular call" or "nesting too deep"	A flow calls itself, directly or through a third one, or the chain passes three levels	The message names the offending flow.
The access log carries an ambiguity mark	Two runs interleaved their requests	Normal, and flagged as such; attribution is not reliable for those lines.
Alerts come back after being dismissed	The cross hides for the session, it does not mark as read	"Mark as read" in the banner.

The useful reflex. Set the breakpoint on the step after the one that fails rather than on the failing step itself: you then see what the first one really produced, rather than what you thought it would produce.

19. Limits and persistence

The engine constants

Limit	Value	Reason
Runs kept	40	Persistent trace, stripped down. Ten only if the quota is full.
Queue	20	An unbounded queue would be a memory leak.
Nesting	3	Call depth between workflows.
Capability walk	4	Depth into called flows and fallbacks.
Loop turns	50 / 500	The author's cap, the engine's cap.
Cooldown between triggers	3 s	Ten files dropped together do not make ten runs.
Schedule heartbeat	60 s	The current minute serves as the key.
Debug inactivity	15 min	A paused run occupies the queue.
Alerts kept	50	They stay until read.

What survives the page

Key	Content
<code>isofind_workflows</code>	The definitions
<code>isofind_workflow_passages</code>	The trace, stripped down
<code>isofind_workflow_points</code>	Breakpoints, outside the flows
<code>isofind_workflow_alertes</code>	Unread alerts

The trace keeps the shape of the outputs, not the outputs

One `core.sql` output can carry ten thousand isotope values. Writing them to local storage would saturate the quota and put measurements somewhere nothing compartments. What remains is what a diagnosis needs: which step, which state, which message, how long.

The number of rows and the column names are kept, since those are names and not measurements. Without them, a run read back shows a silent step and you conclude the workflow produced nothing; with them you read "238 rows, not kept", and you know there is nothing to do but run it again.

The parameters and the list of granted accesses survive as well: they weigh a few values, and they are what makes it possible to replay a run identically and to know under what rights it worked.

A trace read back carries a mark: a fresh run and a run read back look alike feature for feature, and one has its data where the other no longer does. Without that mark the panel would show "this workflow produced nothing" where the truth is "its outputs are no longer in memory", which blames the workflow for a shortcoming that is not its own.

Events emitted

Event	When
<code>workflow.started</code>	The run starts
<code>workflow.step</code>	A step changes state
<code>workflow.finished</code>	The run ends, whatever its state
<code>workflow.queued</code>	A run enters the queue
<code>workflow.changed</code>	A flow is saved or deleted
<code>workflow.tasks.changed</code>	A task is declared or withdrawn
<code>workflow.debug.arret</code>	The run stops on a step
<code>workflow.debug.repris</code>	An order was given
<code>workflow.debug.fin</code>	The session closes
<code>workflow.debug.points</code>	Breakpoints changed
<code>workflow.alerte</code>	An alert is left or marked read

The panel listens to the first five and repaints as it goes. A panel that only showed the result at the end would make it look as though nothing were happening, and the user would launch it again.

Exporting a run

A run export is not an ISOF. An ISOF is signed, carries its provenance and can be verified. A run export is ordinary JSON, good for dropping a result into a spreadsheet or an email, and worth nothing as proof. The file therefore carries its own warning, telling three cases apart: a dry run, a run read back whose outputs were dropped, and a complete but unsigned run.

CSV files exported from the panel carry an encoding mark at the front, without which a spreadsheet reads UTF-8 as latin-1 and accented words come out unreadable, which the user blames on the export rather than on the spreadsheet.

20. Declaring a task

The registry is open. A plugin declares its tasks and they appear in the editor, in the picker, in the capability computation and in the access log, without a single core file changing.

```
K.workflows.declarerTache({
  id: "myplugin.file",
  source: "myplugin",          // used as the group in the editor

  ecrit: true,                 // missing means « writes »: the careful default
  outil: false,                // invisible to the model, workflows only
  capacite: "samples:write", // otherwise <source>:read | <source>:write

  title: { fr: "Ranger un lot", en: "File a batch" },
  notice: { fr: "...", en: "..." }, // read by the model, if outil

  entrees: [
    { nom: "lot", type: "texte", requis: true, default: null,
      libelle: { fr: "Identifiant du lot", en: "Batch id" } }
  ],
  sorties: [ { nom: "filed", type: "nombre" } ],

  executer: function (args, ctx) {
    // ctx.passage, ctx.etape, ctx.contexte, ctx.agent, ctx.simulation
    return Promise.resolve({ filed: 3 });
  }
});
```

The declaration returns a token that knows how to withdraw itself, which a plugin needs when it unloads: its task then disappears from the registry, and flows that used it fall cleanly on the *absente* state rather than on an error.

Six habits that hold the rest together

1. **Return a promise of an object.** The keys of that object are what later steps will read as ``${id.key}``. A task returning only a boolean would leave the workflow blind to what just happened.

2. **Relay the real message.** Read the response body before trying to parse it: on an error, parsing fails before the cause could be read, and the caller receives a complaint about invalid JSON instead of the reason for the refusal.
3. **Accept a string as readily as an object.** The engine substitutes strings: a boolean arrives as text as often as as a boolean, and an object written in the editor arrives as JSON text. Testing only the expected form makes a correct step fail, or silently falls back on a default.
4. **Do not reimplement the limits.** Routes already bound access to the declared roots, visibility and the caps. A second check would end up diverging from the first, and one of the two would become the door nobody watches.
5. **Go through the registry rather than the route.** When an operation already has a registry in the page, such as acquisition, calling the route directly would bypass the locks it applies, and a workflow would become the way an unauthorised source gets out anyway.
6. **Only declare a task if its module is there.** Declared without its route, it appears in the list and fails when called: the model gets a not found error where it expects an answer, and concludes the data does not exist.