

Package: conductoR (via r-universe)

October 12, 2026

Title Perform a Contract-Typed Analytical Plan Across Analytics Components

Version 0.2.0

Description Declares an analytical plan as a directed acyclic graph of nodes, each bound to a tool that returns an 'orchestraManifest' result, and performs it in topological order. Every edge is typed by the inferential target the upstream node declares, and a mismatched handoff stops the run before the downstream node executes rather than coercing across a contract boundary. Two epistemic operators are built in: a power-aware abstention gate that refuses to pass a null verdict the design could not have resolved, and a triangulation operator that reconciles any number of independent lenses on one question. A plan serialises to 'YAML' and back, compiles to a 'targets' pipeline, and every run writes a provenance record with the integrity chain of the manifests it produced.

License MIT + file LICENSE

URL <https://github.com/max578/conductoR>

BugReports <https://github.com/max578/conductoR/issues>

Encoding UTF-8

Language en-AU

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Depends R (>= 4.3.0)

Imports jsonlite, orchestraManifest (>= 0.1.0), S7, stats, utils, yaml

Suggests agridat, jsonvalidate, knitr, PESTO, rmarkdown, targets, testthat (>= 3.0.0)

VignetteBuilder knitr

Remotes max578/orchestraManifest, max578/PESTO

Config/roxygen2/version 8.1.0

Repository <https://max578.r-universe.dev>
Date/Publication 2026-10-12 14:21:50 UTC
RemoteUrl <https://github.com/max578/conductorR>
RemoteRef main
RemoteSha 3ecdb1b23a9e3dec317a1d0643c1d4eef0fb3c4d

Contents

compile_to_targets	2
conduct	3
conduct_step	6
decision_to_manifest	7
edge	8
lens_to_manifest	9
node	10
oap_node	12
oap_plan	14
plan_from_json	15
plan_from_yaml	16
plan_node	17
plan_order	17
plan_schema	18
plan_to_yaml	19
plan_validate	20
register_tool	21
resolve_tool	22
run_provenance	22
same_plan	23
terminal_target	24
Index	26

compile_to_targets	<i>Compile a plan to a targets script</i>
--------------------	---

Description

Writes a `_targets.R` in which each node is one `targets::tar_target()` calling `conduct_step()`, so the plan inherits caching and skipping of up-to-date nodes from 'targets' while every edge stays type-checked. Target names are the node ids prefixed with `n_`.

Usage

`compile_to_targets(plan, script_file, preamble = character(0))`

Arguments

plan	An oap_plan .
script_file	The path to write.
preamble	Lines of R placed at the top of the script. They must bind <code>.score</code> to the plan and <code>.data</code> to the data object, and make any registered tools available in the process that runs the pipeline.

Details

A plan whose nodes use `when`, `foreach` or `on_refusal` is refused with an error of class `conductoR_unsupported`, since those are applied by [conduct\(\)](#). A `select` node compiles like any other node.

Value

script_file, invisibly.

Author(s)

Max Moldovan

See Also

[terminal_target\(\)](#), [conduct_step\(\)](#).

Examples

```
p <- oap_plan(id = "chain", nodes = list(
  node("a", tool = "stats::median"),
  node("b", tool = "stats::median", consumes = list(edge("a"))))
))
f <- tempfile(fileext = ".R")
compile_to_targets(p, f, preamble = c("library(conductoR)",
                                     ".score <- readRDS('plan.rds')",
                                     ".data <- list()"))
cat(readLines(f), sep = "\n")
```

conduct

Perform a plan

Description

Walks the plan in topological order and performs each node with [conduct_step\(\)](#). Every in-edge is type-checked before its node runs, so a mismatched handoff stops the run before the downstream tool executes. An abstaining gate in `halt-branch` mode skips every node downstream of it; the skipped nodes are recorded in the provenance, not dropped.

Usage

```
conduct(plan, data = list(), out_dir = NULL, verbose = TRUE, label = NULL)
```

Arguments

plan	An oap_plan .
data	The data object every tool receives.
out_dir	A directory to write the provenance record into, or NULL to write nothing. See run_provenance() for the files.
verbose	Print one line per node as it runs.
label	A short string stored with the run, such as "blinded" or "unblinded", or NULL.

Value

A list with elements plan_id, plan_hash, input_hash, label, bus (the manifest each node emitted, named by node id), provenance (one record per node), conditions (one row per when evaluated: node, condition, value), items (one row per foreach item: node, item, instance, status, run_id), stopped_by (the node whose refusal ended the run, or NA), n_nodes, n_skipped, seconds and terminal (the manifest of the last node that ran). Returned invisibly.

The abstention gate

An abstain_gate node reads the one verdict it consumes and abstains when any of these holds: the upstream emitter abstained; the verdict is "no_effect" and the metric n is below `params$n_min` (default 25); the verdict is "no_effect" and the power proxy $|effect_mean| / (noise_sd / \sqrt{n})$ is below `params$z_floor` (default 2); or any metric named in `params$floors` (a named list of minima) is below its floor. `params$mode` sets what an abstention does downstream: "flag" (default) records it and continues, "halt-branch" skips the descendants, "reroute" tells a downstream triangulation to drop the gated lens and reconcile the survivors.

Triangulation

A triangulate node reads every lens it consumes, each a manifest whose summary metrics carry `effect_present` or `influence_present`, and at most one gate. It emits one of seven consensus headlines: `causal_effect_corroborated` (two or more lenses, all with an effect), `no_effect_corroborated` (two or more, none with an effect), `single_lens_effect` and `single_lens_no_effect` (one lens), `disagreement_flagged` (mixed), and `abstain` (the gate abstained in flag mode, or no lens survived a reroute).

Conditions

A node with a when condition runs only when the condition is TRUE. When it is FALSE, the node and every node downstream of it are skipped, and the run records the condition, its value and the reason. A condition reads results of nodes upstream: `<node>.headline`, `<node>.abstained`, `<node>.n_rows` (rows of its params table) and `<node>.metrics.<name>`. It may use numbers written plainly, quoted strings, TRUE, FALSE, the comparisons `==`, `!=`, `<`, `<=`, `>`, `>=`, `%in%` with `c(...)`, the logical operators `&`, `|`, `!` (`&&` and `||` read the same) and brackets. A function call, `$`, `::`, backticks,

an assignment and anything else not listed here are refused when the plan is built. A condition is never run as R code.

A select node keeps the rows of its input's params table for which `params$rule` is TRUE. The rule follows the same grammar, with column names in place of references. Rows where the rule gives NA are dropped and counted in the metric `n_na`.

Running a node once per item

A compute node with `foreach` runs once for each item: a fixed list, or each distinct value of a column of the params table of a node it consumes (`list(from = <node>, column = <name>)`). Each instance receives its item as `node@params$item` and is recorded as `<id>[<item>]`. More items than `foreach_max` (default 50) stop the run with an error. The node emits one manifest whose params table has one row per item, with its status (ran, refused or skipped), and whose outputs holds the instance manifests. With no items, nothing runs and downstream nodes receive that record, with `n_items = 0`.

Refusals

A refusal is a result that abstains: its summary says abstained. A node's `on_refusal` sets what a refusal does: "continue" (default) passes it on, "halt-branch" skips every node downstream of it, "stop" ends the run; the nodes not yet performed are recorded as skipped. In a `foreach`, "halt-branch" and "stop" also skip the items not yet run. A tool that fails with an error stops the run in every case: an error is not a refusal.

Blinded runs

Every run records `input_hash`, a SHA-256 of data, and `plan_hash`, a SHA-256 of the plan together with the code of every tool it references (see [same_plan\(\)](#)). To blind an analysis, run the plan on data whose treatment labels are hidden, freeze the plan, then run it once on the true data. [same_plan\(\)](#) then shows that both runs used one plan on different data.

Author(s)

Max Moldovan

See Also

[conduct_step\(\)](#), [run_provenance\(\)](#), [plan_to_yaml\(\)](#), [same_plan\(\)](#).

Examples

```
lens_a <- function(inputs, data, node) {
  lens_to_manifest("effect", TRUE, "lens_a", "lens_a:demo")
}
lens_b <- function(inputs, data, node) {
  lens_to_manifest("effect", TRUE, "lens_b", "lens_b:demo")
}
p <- oap_plan(id = "two_lenses", nodes = list(
  node("a", lens_a, emits = "treatment_effects"),
  node("b", lens_b, emits = "treatment_effects"),
```

```

    node("consensus", operator = "triangulate", emits = "decisions",
          consumes = list(edge("a", "treatment_effects"),
                           edge("b", "treatment_effects")))
  ))
run <- conduct(p, verbose = FALSE)
run$terminal@summary$headline

# Run a check once per site, but only when lens a found an effect.
check <- function(inputs, data, node) {
  lens_to_manifest("effect", TRUE, "check",
    paste0("check:", node@params$item))
}
p2 <- oap_plan(id = "per_site", nodes = list(
  node("a", lens_a, emits = "treatment_effects"),
  node("check", check, emits = "treatment_effects",
    consumes = list(edge("a", "treatment_effects")),
    when = "a.headline == \"effect\"",
    foreach = c("site_1", "site_2"))
))
run2 <- conduct(p2, verbose = FALSE)
run2$items

```

conduct_step

Perform one node

Description

Type-checks every input the node consumes, dispatches on the node's operator and checks that the output declares the inferential target the node promised. `conduct()` calls this once per node, and a pipeline written by `compile_to_targets()` calls it once per target.

Usage

```
conduct_step(node, inputs = list(), data = list())
```

Arguments

node	An oap_node .
inputs	A named list of upstream manifests, one per edge in <code>node@consumes</code> , named by the upstream node id.
data	The data object every tool receives as its second argument.

Value

The `orchestraManifest::orchestra_manifest` the node emitted.

Errors

An input that does not declare the contract its edge was typed with stops the step with a message containing TYPED-EDGE VIOLATION. An input that fails `orchestraManifest::consume_manifest()` stops with that guard's own message. A tool that returns something other than a manifest, or a manifest with a different target from `node@emits`, stops after the tool has run.

when, foreach and on_refusal are applied by `conduct()`; `conduct_step()` performs the node once, as given.

Author(s)

Max Moldovan

See Also

`conduct()`.

Examples

```
lens <- node("lens", emits = "treatment_effects",
  tool = function(inputs, data, node) {
    lens_to_manifest("effect", TRUE, "example", "example:lens")
  })
m <- conduct_step(lens)
m@inferential_target
```

decision_to_manifest *Lift a decision into a decisions manifest*

Description

A decision node turns a consensus and its supporting metrics into an action recommendation. The headline "insufficient_evidence" is recorded as an abstention.

Usage

```
decision_to_manifest(
  headline,
  emitter,
  method,
  metrics = list(),
  emitter_version = "0.0.0",
  run_id = NULL,
  consumed = list()
)
```

Arguments

headline	The recommended action, or "insufficient_evidence".
emitter	The name of the component or method family emitting the verdict.
method	A single string naming the method, conventionally "component:method".
metrics	A named list of scalars behind the verdict.
emitter_version	The emitting component's version string.
run_id	A run identifier, or NULL to derive one from the verdict.
consumed	The upstream manifests the decision was derived from. Their lineage is recorded in the result.

Value

An `orchestraManifest::orchestra_manifest` with `inferential_target = "decisions"`.

Author(s)

Max Moldovan

See Also

`lens_to_manifest()`.

Examples

```
lens <- lens_to_manifest("effect", TRUE, "ols", "ols:slope_t")
d <- decision_to_manifest("apply_nitrogen", "agronomy", "agronomy:rule",
  metrics = list(rate_kg_ha = 90),
  consumed = list(lens))
d@summary$headline
length(d@consumed_manifests)
```

edge

Declare a typed edge

Description

Declare a typed edge

Usage

```
edge(from, contract = "any")
```

Arguments

from	The id of the upstream node.
contract	The inferential target the upstream manifest must declare, or "any" to accept every target.

Value

A list with elements from and contract.

Author(s)

Max Moldovan

See Also

[node\(\)](#).

Examples

```
edge("lens", "treatment_effects")
```

lens_to_manifest	<i>Lift a binary verdict into a treatment-effects manifest</i>
------------------	--

Description

Any method that answers "is there an effect?" on a shared question can take part in a plan through this emitter: a kernel test, a classical regression slope test, a rank correlation. The effect_present flag is what triangulation reads; n, effect_mean and noise_sd, when supplied in metrics, are what the abstention gate reads.

Usage

```
lens_to_manifest(
  headline,
  effect_present,
  emitter,
  method,
  metrics = list(),
  emitter_version = "0.0.0",
  run_id = NULL
)
```

Arguments

headline	The verdict label, such as "effect" or "no_effect".
effect_present	Whether the method found an effect.
emitter	The name of the component or method family emitting the verdict.
method	A single string naming the method, conventionally "component:method".
metrics	A named list of scalars behind the verdict.
emitter_version	The emitting component's version string.
run_id	A run identifier, or NULL to derive one from the verdict.

Value

An [orchestraManifest::orchestra_manifest](#) with `inferential_target = "treatment_effects"`.

Author(s)

Max Moldovan

See Also

[decision_to_manifest\(\)](#), [conduct\(\)](#).

Examples

```
m <- lens_to_manifest("effect", TRUE, "ols", "ols:slope_t",
  metrics = list(p_value = 0.01, n = 136L))
m@summary$headline
orchestraManifest::verify_manifest(m)$ok
```

node

Declare a node

Description

Builds an [oap_node](#) so a plan reads as a declaration. `tool` is either a closure or a string reference; a string is stored as `tool_ref` and resolved by [resolve_tool\(\)](#) when the node runs.

Usage

```
node(
  id,
  tool = function(...) NULL,
  consumes = list(),
  emits = "any",
  operator = "compute",
```

```

    role = "compute",
    params = list(),
    when = "",
    foreach = NULL,
    foreach_max = 50L,
    on_refusal = "continue"
  )

```

Arguments

id	A single non-empty string, unique within the plan.
tool	A closure function(inputs, data, node) returning an orchestraManifest::orchestra_manifest , or a string reference in one of the forms resolve_tool() accepts.
consumes	A list of edges, each as returned by edge() .
emits	The inferential target the node's output must declare, one of orchestraManifest::inferential_target or "any".
operator	One of "compute", "abstain_gate", "triangulate" or "select".
role	A free-text label for the component the node stands for.
params	A named list of operator parameters. A "select" node needs rule, a row condition written in the grammar described in conduct() .
when	A condition that must be TRUE for the node to run, or "" to always run it. See the <i>Conditions</i> section of conduct() .
foreach	NULL to run the node once; a list of items to run it once per item; or list(from = <node id>, column = <column>) to take the items from a column of the params table of a node this node consumes.
foreach_max	The largest number of items a foreach may run. More items stop the run with an error.
on_refusal	What an abstaining result of this node does: "continue" passes it on, "halt-branch" skips every node downstream, "stop" ends the run.

Value

An [oap_node](#).

Author(s)

Max Moldovan

See Also

[edge\(\)](#), [oap_plan](#).

Examples

```
lens <- node("lens", tool = "my_lens", emits = "treatment_effects")
lens@tool_ref
check <- node("check", tool = "my_check", emits = "treatment_effects",
             consumes = list(edge("lens", "treatment_effects")),
             when = "lens.metrics.p_value < 0.05",
             foreach = c("site_a", "site_b"))
check@foreach
gate <- node("gate", consumes = list(edge("lens", "treatment_effects")),
            emits = "decisions", operator = "abstain_gate",
            params = list(n_min = 25L))
gate@operator
```

oap_node

A node of an analytical plan

Description

One step of a plan, bound to a tool with the uniform signature `tool(inputs, data, node)` that returns an [orchestraManifest::orchestra_manifest](#). The node declares which upstream nodes it consumes, each with the contract it expects that input to carry, and the inferential target its own output must declare.

Usage

```
oap_node(
  id = character(0),
  role = "compute",
  tool = function(...) NULL,
  tool_ref = "",
  consumes = list(),
  emits = "any",
  operator = "compute",
  params = list(),
  when = "",
  foreach = NULL,
  foreach_max = 50L,
  on_refusal = "continue"
)
```

Arguments

id	A single non-empty string, unique within the plan.
role	A free-text label for the component the node stands for.
tool	The closure that performs the node, or an unused stand-in when <code>tool_ref</code> is set.

tool_ref	A tool reference in one of the forms resolve_tool() accepts, or "" for an in-session closure.
consumes	A list of edges, each as returned by edge() .
emits	The inferential target the node's output must declare, one of orchestraManifest::inferential_target or "any".
operator	One of "compute", "abstain_gate", "triangulate" or "select".
params	A named list of operator parameters. A "select" node needs rule, a row condition written in the grammar described in conduct() .
when	A condition that must be TRUE for the node to run, or "" to always run it. See the <i>Conditions</i> section of conduct() .
foreach	NULL to run the node once; a list of items to run it once per item; or <code>list(from = <node id>, column = <column>)</code> to take the items from a column of the params table of a node this node consumes.
foreach_max	The largest number of items a foreach may run. More items stop the run with an error.
on_refusal	What an abstaining result of this node does: "continue" passes it on, "halt-branch" skips every node downstream, "stop" ends the run.

Details

operator distinguishes ordinary computation from the two epistemic node types the engine performs itself. A "compute" node calls its tool. An "abstain_gate" node reads the one verdict it consumes and passes or abstains, see [conduct\(\)](#) for the policy. A "triangulate" node reconciles every lens it consumes into one consensus.

A tool may be an in-session closure or a string reference that [resolve_tool\(\)](#) resolves when the node runs. The reference form is what makes a plan portable, see [plan_to_yaml\(\)](#).

A "select" node keeps the rows of its one input's params table that meet `params$rule`, and emits them under the input's inferential target.

when, foreach, foreach_max and on_refusal are applied by [conduct\(\)](#); see its sections for what each does. Node ids may not contain dots, square brackets or white space, because conditions refer to nodes as `<node>.<field>` and foreach instances are named `<node>[<item>]`.

Value

An oap_node object.

Author(s)

Max Moldovan

See Also

[node\(\)](#) for the declarative constructor, [oap_plan](#) for the plan.

Examples

```
n <- oap_node(id = "lens", emits = "treatment_effects")
n@id
```

oap_plan

An analytical plan

Description

A plan is a list of [oap_node](#) objects with unique ids whose edges all point at nodes within the plan. Acyclicity is checked when the plan is ordered, see [plan_order\(\)](#).

Usage

```
oap_plan(id = character(0), nodes = list(), meta = list())
```

Arguments

id	A single string naming the plan.
nodes	A list of oap_node objects.
meta	A named list of free-form metadata, such as the question the plan answers. Serialised with the plan.

Value

An oap_plan object.

Author(s)

Max Moldovan

See Also

[conduct\(\)](#) to perform a plan, [plan_to_yaml\(\)](#) to save one.

Examples

```
p <- oap_plan(id = "one_lens", nodes = list(
  node("lens", emits = "treatment_effects")
))
plan_order(p)
```

plan_from_json	<i>Read a plan from JSON</i>
----------------	------------------------------

Description

The JSON spelling of a plan holds the same fields as the YAML spelling and follows the schema returned by [plan_schema\(\)](#). A plan written as JSON, for example as structured output from an AI model, can be read here and saved as YAML with [plan_to_yaml\(\)](#).

Usage

```
plan_from_json(path)
```

Arguments

path	A file path, or a string of JSON text.
------	--

Value

An [oap_plan](#). A plan with problems is an error that lists all of them; see [plan_validate\(\)](#).

Author(s)

Max Moldovan

See Also

[plan_validate\(\)](#), [plan_from_yaml\(\)](#).

Examples

```
p <- plan_from_json('{
  "id": "one_lens",
  "nodes": [
    {"id": "lens", "tool": "my_lens", "emits": "treatment_effects"}
  ]
}')
plan_order(p)
```

plan_from_yaml	<i>Read a plan from YAML</i>
----------------	------------------------------

Description

Tool references are kept as strings and resolved when the plan is performed, so a plan loads without its tools present.

Usage

```
plan_from_yaml(path)
```

Arguments

path	A file path, or a string of YAML text.
------	--

Details

The plan is checked by [plan_validate\(\)](#) before it is built, and a plan with problems is an error that lists all of them, including any unquoted yes, no, on or off and any number in scientific notation.

Value

An [oap_plan](#).

Author(s)

Max Moldovan

See Also

[plan_to_yaml\(\)](#), [plan_validate\(\)](#), [plan_from_json\(\)](#).

Examples

```
f <- system.file("scores", "dual_lens_gate.oap.yaml", package = "conductorR")
p <- plan_from_yaml(f)
plan_order(p)
```

plan_node	<i>Fetch a node by id</i>
-----------	---------------------------

Description

Fetch a node by id

Usage

```
plan_node(plan, id)
```

Arguments

plan	An oap_plan .
id	The id of the node.

Value

The [oap_node](#) with that id. An unknown id is an error.

Author(s)

Max Moldovan

Examples

```
p <- oap_plan(id = "one", nodes = list(node("a")))
plan_node(p, "a")@id
```

plan_order	<i>Topological order of a plan</i>
------------	------------------------------------

Description

Orders the nodes so that every node follows the nodes it consumes. A plan with a cycle is an error.

Usage

```
plan_order(plan)
```

Arguments

plan	An oap_plan .
------	-------------------------------

Value

A character vector of node ids in execution order.

Author(s)

Max Moldovan

Examples

```
p <- oap_plan(id = "chain", nodes = list(
  node("b", consumes = list(edge("a"))),
  node("a")
))
plan_order(p)
```

plan_schema

The JSON Schema of the plan language

Description

The schema (JSON Schema 2020-12) describes every field a plan may hold, its type and its allowed values. Editors can use it to check a plan as it is written, and an AI model can use it to write a plan as structured output. [plan_validate\(\)](#) makes the same checks and adds those a schema cannot express: references between nodes, cycles and the condition grammar.

Usage

```
plan_schema()
```

Value

The path of the schema file, oap_plan_v0.2.schema.json.

Author(s)

Max Moldovan

See Also

[plan_validate\(\)](#).

Examples

```
s <- jsonlite::fromJSON(plan_schema(), simplifyVector = FALSE)
names(s$properties)
```

plan_to_yaml

Write a plan as YAML

Description

Every compute node must reference its tool by name (see [node\(\)](#)), since a closure cannot be written to a file. Operator nodes carry no tool.

Usage

```
plan_to_yaml(plan, path = NULL)
```

Arguments

plan	An oap_plan .
path	A file to write, or NULL to return the YAML text.

Details

Every string is written in double quotes and every number in plain decimal form, so the file reads back to the same plan. The line `plan_version: "0.2"` is written only when the plan uses a field added in version 0.2: `when`, `foreach`, `foreach_max`, `on_refusal` or a `select` node.

Value

path invisibly when a file was written, otherwise the YAML text as a single string.

Author(s)

Max Moldovan

See Also

[plan_from_yaml\(\)](#), [register_tool\(\)](#).

Examples

```
p <- oap_plan(id = "one_lens", nodes = list(
  node("lens", tool = "my_lens", emits = "treatment_effects"),
  node("gate", operator = "abstain_gate", emits = "decisions",
    consumes = list(edge("lens", "treatment_effects")),
    params = list(n_min = 25L))
))
cat(plan_to_yaml(p))
```

plan_validate	<i>Check a plan file and list every problem</i>
---------------	---

Description

Reads a plan written in YAML or JSON and checks it against the plan language: the fields and their types, the operators and inferential targets, the condition grammar, and the references between nodes. Every problem is reported, not only the first, each with the field, the node, the rule it breaks and how to fix it. A person or an AI model can repair a plan from this table and check it again.

Usage

```
plan_validate(x)
```

Arguments

x	The path of a .yaml, .yml or .json plan file, or the plan's text. Text that starts with an opening brace is read as JSON.
---	---

Details

In a YAML plan, an unquoted yes, no, on or off is refused: write true or false for a logical, or quote the word for text. A number in scientific notation (1e3) is refused with its plain form (1000). The structure checks follow the JSON Schema returned by [plan_schema\(\)](#).

Value

A data frame with one row per problem and the columns field, node (empty for a plan-level field), rule and fix. A valid plan gives zero rows.

Author(s)

Max Moldovan

See Also

[plan_schema\(\)](#), [plan_from_yaml\(\)](#), [plan_from_json\(\)](#).

Examples

```
f <- system.file("scores", "dual_lens_gate.oap.yaml", package = "conductorR")
nrow(plan_validate(f))

plan_validate('
id: broken
nodes:
- id: gate
  operator: abstain_gate
  consumes: [{from: lens, contract: treatment_effects}]')
```

```
    params: {mode: off, n_min: 1e3}
  ')
```

register_tool	<i>Register a tool by name</i>
---------------	--------------------------------

Description

A registered tool can be referenced from a node by its name, which is what lets a plan be written to a file and performed again in another session. The registry is per session; a plan loaded from a file needs its tools registered again before it is performed.

Usage

```
register_tool(name, fn)
```

Arguments

name	A single string.
fn	A closure function(inputs, data, node) returning an orchestraManifest::orchestra_manifest .

Value

name, invisibly.

Author(s)

Max Moldovan

See Also

[resolve_tool\(\)](#), [plan_to_yaml\(\)](#).

Examples

```
register_tool("always_effect", function(inputs, data, node) {
  lens_to_manifest("effect", TRUE, "example", "example:constant")
})
resolve_tool("always_effect")
```

resolve_tool	<i>Resolve a tool reference</i>
--------------	---------------------------------

Description

Three reference forms are accepted. "<name>" is a tool registered with [register_tool\(\)](#). "pkg::fn" is an exported function of an installed package. "script:<path>:<fn>" sources an R script into a fresh environment and returns the function it defines under that name; the script runs with the privileges of the session, so only reference scripts you would source yourself.

Usage

```
resolve_tool(ref)
```

Arguments

ref	A single string in one of the three forms.
-----	--

Value

The function the reference names. An unknown reference is an error.

Author(s)

Max Moldovan

See Also

[register_tool\(\)](#).

Examples

```
resolve_tool("stats::median")
```

run_provenance	<i>A run's provenance as a table</i>
----------------	--------------------------------------

Description

A run's provenance as a table

Usage

```
run_provenance(run)
```

Arguments

run The list returned by `conduct()`.

Value

A data frame with one row per node in execution order and columns node, operator, role, consumes (upstream ids joined with +), emits, emitter, run_id, data_hash, abstained, seconds, status (ran, refused or skipped) and reason (why a node was skipped, otherwise empty).

Files

When `conduct()` is given `out_dir`, it writes three files there: `run.rds` (the whole run object), `run_card.md` (this table and the terminal verdict) and `provenance.md` (a front-matter record with the chain of node ids and payload hashes in execution order). When the run skipped nodes, evaluated when conditions or ran foreach items, both Markdown files list them with their reasons, values and statuses.

Author(s)

Max Moldovan

See Also

`conduct()`.

Examples

```
lens <- node("lens", emits = "treatment_effects",
  tool = function(inputs, data, node) {
    lens_to_manifest("effect", TRUE, "example", "example:lens")
  })
run <- conduct(oap_plan(id = "one", nodes = list(lens)), verbose = FALSE)
run_provenance(run)[, c("node", "emits", "abstained")]
```

same_plan

Check that two runs used the same plan

Description

Compares the plan hashes and input hashes that `conduct()` records. The plan hash covers the plan and the code of every tool it references, so two runs with equal plan hashes ran the same analysis. Use it after a blinded analysis: the blinded run and the unblinded run should share a plan hash and differ in input hash.

Usage

```
same_plan(run_a, run_b)
```

Arguments

run_a, run_b Two runs returned by `conduct()`.

Value

A list with same_plan and same_input (logical), and the four hashes and two labels compared.

Author(s)

Max Moldovan

See Also

`conduct()`.

Examples

```
lens <- function(inputs, data, node) {
  lens_to_manifest(if (mean(data$d) > 0) "effect" else "no_effect",
    mean(data$d) > 0, "demo", "demo:mean")
}
p <- oap_plan(id = "blind_demo", nodes = list(
  node("lens", lens, emits = "treatment_effects")))
d <- c(0.3, 0.1, 0.2, 0.4)
flip <- c(1, -1, -1, 1)
blinded <- conduct(p, data = list(d = d * flip), verbose = FALSE,
  label = "blinded")
unblinded <- conduct(p, data = list(d = d), verbose = FALSE,
  label = "unblinded")
same_plan(blinded, unblinded)$same_plan
```

terminal_target

The target that holds a plan's terminal result

Description

The target that holds a plan's terminal result

Usage

```
terminal_target(plan)
```

Arguments

plan An `oap_plan`.

Value

The name of the target, as a string, that `compile_to_targets()` maps the last node in topological order to.

Author(s)

Max Moldovan

See Also

`compile_to_targets()`.

Examples

```
p <- oap_plan(id = "chain", nodes = list(
  node("b", consumes = list(edge("a"))),
  node("a")
))
terminal_target(p)
```

Index

`compile_to_targets`, [2](#)
`compile_to_targets()`, [6](#), [25](#)
`conduct`, [3](#)
`conduct()`, [3](#), [6](#), [7](#), [10](#), [11](#), [13](#), [14](#), [23](#), [24](#)
`conduct_step`, [6](#)
`conduct_step()`, [2](#), [3](#), [5](#)

`decision_to_manifest`, [7](#)
`decision_to_manifest()`, [10](#)

`edge`, [8](#)
`edge()`, [11](#), [13](#)

`lens_to_manifest`, [9](#)
`lens_to_manifest()`, [8](#)

`node`, [10](#)
`node()`, [9](#), [13](#), [19](#)

`oap_node`, [6](#), [10](#), [11](#), [12](#), [14](#), [17](#)
`oap_plan`, [3](#), [4](#), [11](#), [13](#), [14](#), [15–17](#), [19](#), [24](#)
`orchestraManifest::consume_manifest()`,
[7](#)
`orchestraManifest::inferential_targets()`,
[11](#), [13](#)
`orchestraManifest::orchestra_manifest`,
[6](#), [8](#), [10–12](#), [21](#)

`plan_from_json`, [15](#)
`plan_from_json()`, [16](#), [20](#)
`plan_from_yaml`, [16](#)
`plan_from_yaml()`, [15](#), [19](#), [20](#)
`plan_node`, [17](#)
`plan_order`, [17](#)
`plan_order()`, [14](#)
`plan_schema`, [18](#)
`plan_schema()`, [15](#), [20](#)
`plan_to_yaml`, [19](#)
`plan_to_yaml()`, [5](#), [13–16](#), [21](#)
`plan_validate`, [20](#)
`plan_validate()`, [15](#), [16](#), [18](#)

`register_tool`, [21](#)
`register_tool()`, [19](#), [22](#)
`resolve_tool`, [22](#)
`resolve_tool()`, [10](#), [11](#), [13](#), [21](#)
`run_provenance`, [22](#)
`run_provenance()`, [4](#), [5](#)

`same_plan`, [23](#)
`same_plan()`, [5](#)

`terminal_target`, [24](#)
`terminal_target()`, [3](#)