Package 'pipeflow'

November 15, 2026

Title Fast Interactive Data Analysis Pipelines

Version 0.6.0

Maintainer Roman Pahl <roman.pahl@gmail.com>

Description A lightweight and intuitive framework for building interactive data analysis pipelines. You add R functions one by one, and 'pipeflow' wires them into a pipeline that stays consistent as you go. Modify, remove, or insert steps at any stage, manage all parameters in one place, fast execution (C++-powered DAG) for interactive use and Shiny backends.

License MIT + file LICENSE

URL https://rpahl.github.io/pipeflow/, https://github.com/rpahl/pipeflow

BugReports https://github.com/rpahl/pipeflow/issues

Depends R (>= 4.2.0)

Imports data.table, Rcpp (>= 1.1.1), stats

Suggests bench, ggplot2, gridExtra, knitr, mockery, parallel, rlang, rmarkdown, targets, testthat, visNetwork, withr

LinkingTo Rcpp

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Language en-US

NeedsCompilation yes

Author Roman Pahl [aut, cre]

Repository CRAN

Date/Publication 2026-11-15 19:00:02 UTC

Contents

as.function.pipeflow . . . 2
dim.pipeflow . . . 6
Extract.pipeflow . . . 8
pipeflow_lgr . . . 14
pip_add . . . 15
pip_clone . . . 19
pip_collect . . . 20
pip_data . . . 22
pip_graph . . . 23
pip_inner . . . 24
pip_lock . . . 25
pip_new . . . 26
pip_output_value . . . 27
pip_param_value . . . 29
pip_remove . . . 31
pip_rename . . . 32
pip_replace . . . 33
pip_reset . . . 35
pip_run . . . 36
pip_run_state . . . 41
pip_set_params . . . 42
pip_tag . . . 46
pip_view . . . 47
print.pipeflow . . . 48
rbind.pipeflow . . . 50

Index **52**

`as.function.pipeflow` *Turn a pipeline into a function*

Description

Returns a function that runs a copy of the pipeline `x` and returns the output of selected steps. The function arguments are the unbound parameters of `x`, with their current values as defaults. As any other function, it can be added to an "outer" pipeline with `pip_add()`, in which case the unbound parameters of `x` become parameters of that pipeline. To bind parameters of `x` to output of steps of the outer pipeline, pass them as references in `params` of `pip_add()`, e.g. `pip_add(outer, "inner", f_inner, params = list(df = ~outer_step))`.

Usage

```
## S3 method for class 'pipeflow'
as.function(
  x,
  out = NULL,
```

```
  cache = TRUE,
  on_error = c("stop", "continue"),
  lgr = NULL,
  ...
)
```

Arguments

x	A pipeflow pipeline (not a view; use `x[i, view = FALSE]` to turn a part of a pipeline into a function).
out	The result of the function. `NULL` (the default) returns the outputs of all steps as a named list, see `pip_collect()`; a step name returns the output of that step; several step names return a named list of their outputs. A function with at least one argument is called with (a light copy of) the run inner pipeline and returns the result, see Details.
cache	Whether to keep the run copies of `x` for reuse, `TRUE` (the default) or `FALSE`. The function keeps one copy per outer step and partition, see Details. With `FALSE`, a step mapped over partitions can run the function in parallel, see the `map` argument of `pip_run()`.
on_error	What to do when an inner step fails: `"stop"` (the default) ends the inner run at that step, `"continue"` runs the inner steps that do not depend on it, see Details and `pip_run()`.
lgr	The logger of the inner run, see `pip_run()`. `NULL` (the default) runs the inner pipeline silently. To activate logging, set e.g. `lgr = pipeflow_lgr`.
...	Not used.

Details

By default, the inner pipeline is run silently, but warnings and messages of its steps are still passed on to the caller. A failing inner step results in an error of the caller which in turn makes the outer step fail. Param values of the outer pipeline reach the function unwrapped (see `pip_param_value()`). The arguments of the function are values, also if they are formulas: unlike in `pip_set_params()`, a formula argument does not bind a param of `x` to a step.

Choosing the output:

With step names in `out`, the function returns their outputs. If `out` is a function, it is called with the run inner pipeline and its result is returned, e.g. `out = \(p) pip_collect(p, by = "tags")` groups the outputs by tags. That pipeline is a light copy that shares its values with the kept one, so returning it (or a view of it) keeps no extra copy of the data. Don't modify the values in place (e.g. with `:=` on a `data.table` output).

Errors of inner steps:

If an inner step fails, the error names it, e.g. `step 'fit': negative`. As a step of an outer pipeline, the error also names the outer step and, if that step is mapped over partitions, the key: `step 'by_batch': key 'D': step 'fit': negative`. Nested pipeline functions add one step per level.

Continuing after errors of inner steps:
With `on_error = "continue"` the inner pipeline is run like `pip_run(on_error = "continue")`: inner steps that do not depend on a failed step still run, and a step that refers to it via `~try(step)` receives the failure. The function still fails if a step named in `out` failed or was not run. Since `out = NULL` or a function `out` requests *all* steps, name the relevant steps in `out` to profit from a continuing inner run.

Pipeline functions with partitioned inputs:
If the step with the pipeline function receives partitioned inputs (from an earlier step with `exec = "split"`), the function is called once per partition, caching one inner pipeline of `x` per step and partition key. A call updates the params and inputs of the respective pipeline, so that only outdated inner steps are run again.

- A partition caches one pipeline for each key, which hold their inputs and outputs, so they need memory.
- A step without partitioned inputs keeps a single pipeline.
- Pure calls of the pipeline function (outside of a step) always share one pipeline, e.g. `lapply(subsets, f_pip)` don't cache anything.
- Do *not* use the same pipeline function in two different pipelines - this avoids name clash of keys, which would prevent proper caching (not wrong but slower).
- Cached pipelines are stored in the function. If you save the function or send it to another process, they are copied, including their data and output, so new runs work independently from the original. A function that has not been called yet, or one made with `cache = FALSE`, has no pipelines and therefore is small to copy.
- In case of an error, the error message includes the key of the partition and the name of the failed inner step, see 'Errors of inner steps'.

Limits:

- Params that several steps of `x` share by name become one argument, with the default of the first step as in `pip_get_params()`. Every call sets this value in all of these steps, so a step whose own default differs from it runs with the value of the argument.
- Inputs are compared as they are. A change made in place to an environment or another reference object (e.g. R6) is not seen: as for `pip_set_params()` of a pipeline, which needs `force = TRUE`, the steps stay up to date. A forced run of the outer pipeline (`pip_run(force = TRUE)`) does not force the inner pipelines either, but you can force them by setting `cache = FALSE`.

Value

A function with the unbound params of `x` as arguments.

Note

The conversion is done by taking a snapshot of `x`. Later changes of `x` therefore don't have an impact on the function. Also all steps start as new, so the pipeline function result does not depend on whether `x` has been run before, except for locked steps, which keep their outputs.

See Also

`pip_add()`, `pip_inner()` to look at the kept inner pipelines

Examples

```
inner <- pip_new("inner") |>
  pip_add("input", \(input_df = NULL) input_df) |>
  pip_add("fit", \(data = ~input, degree = 1) {
    lm(mpg ~ poly(wt, degree, raw = TRUE), data = data)
  }) |>
  pip_add("coefs", \(fit = ~fit, digits = 2) round(coef(fit), digits))

# As a plain function: the unbound params are its arguments
f_inner <- as.function(inner, out = "coefs")
f_inner(input_df = mtcars, digits = 1)

# As a step: one inner pipeline
single <- pip_new("single") |>
  pip_add("df", \(df = mtcars) df) |>
  pip_add("coefs", f_inner, params = list(input_df = ~df))

pip_run(single, lgr = NULL)
single[["coefs", "out"]]

# Only the inner steps after the changed param run again
pip_set_params(single, list(digits = 3))
pip_run(single, lgr = NULL)
single[["coefs", "out"]]

# With partioned inputs: one inner pipeline per cylinder number
outer <- pip_new("outer") |>
  pip_add("df", \(df = mtcars) df) |>
  pip_add("by_cyl", \(df = ~df) split(df, df$cyl), exec = "split") |>
  pip_add(
    "coefs",
    as.function(inner, out = "coefs"),
    params = list(input_df = ~by_cyl)
  ) |>
  pip_add("table", \(coefs = ~coefs) do.call(rbind, coefs), exec = "reduce")

pip_get_params(outer)
pip_run(outer, lgr = NULL)
outer[["table", "out"]]

# Only the steps after the changed param run again, for each partition
pip_set_params(outer, list(digits = 3))
print(outer)
pip_run(outer, lgr = NULL)
outer[["table", "out"]]

# Don't use the same function for the same step name in two pipelines:
# this breaks the caching (results are still correct, but it is slower)
a <- pip_new("a") |>
  pip_add("df", \(df = mtcars) df) |>
  pip_add("coefs", f_inner, params = list(input_df = ~df))
b <- pip_new("b") |>
```

```
  pip_add("df", \(df = mtcars[mtcars$am == 1, ]) df) |>
  pip_add("coefs", f_inner, params = list(input_df = ~df))

pip_run(a, lgr = NULL)
pip_run(b, lgr = NULL)

# The inner pipeline now holds the input of `b`, so all its steps run again
# for `a`, not only the ones after `digits`
pip_set_params(a, list(digits = 3))
pip_run(a, lgr = NULL)
a[["coefs", "out"]]

# Better: make one function per pipeline
pip_replace(
  a, "coefs", as.function(inner, out = "coefs"),
  params = list(input_df = ~df)
)

# A function as out: the outputs grouped by tags
tagged <- pip_new("tagged") |>
  pip_add("input", \(input_df = NULL) input_df) |>
  pip_add("n", \(data = ~input) nrow(data), tags = "summary") |>
  pip_add("mpg", \(data = ~input) mean(data$mpg), tags = "summary") |>
  pip_add("fit", \(data = ~input) lm(mpg ~ wt, data = data), tags = "model")

f_tagged <- as.function(tagged, out = \(p) pip_collect(p, by = "tags"))
res <- f_tagged(input_df = mtcars)
names(res)
res$summary

# With on_error = "continue", the inner pipeline handles a failed inner
# step itself: `coefs` is run with the failure of `fit` and reports it.
robust <- pip_new("robust") |>
  pip_add("input", \(input_df = NULL) input_df) |>
  pip_add("fit", \(data = ~input) {
    if (nrow(data) < 3) stop("too few rows to fit")
    lm(mpg ~ wt, data = data)
  }) |>
  pip_add("coefs", \(fit = ~try(fit), digits = 2) {
    if (inherits(fit, "pipeflow_failure")) NULL else round(coef(fit), digits)
  })

f_robust <- as.function(robust, out = "coefs", on_error = "continue")
f_robust(input_df = mtcars)

# the continued inner run signals a warning instead of an error
suppressWarnings(f_robust(input_df = mtcars[1:2, ]))
```

`dim.pipeflow` *Dimensions of a pipeflow pipeline or view*

Description

Treats a pipeline as a table of steps: `length()` and `nrow()` return the number of steps, `ncol()` returns the number of columns of the underlying step table, and `dim()` returns both. Views report only the steps covered by the view.

Usage

```
## S3 method for class 'pipeflow'
length(x)

## S3 method for class 'pipeflow'
dim(x)
```

Arguments

`x`	A pipeflow pipeline or view

Value

- `length()`, `nrow()`: the number of steps
- `ncol()`: the number of columns of the step table
- `dim()`: vector with the number of steps and columns.

See Also

`base::dim()`, `base::nrow()`, `base::length()`

Examples

```
p <- pip_new() |>
  pip_add("s1", \(x = 1) x) |>
  pip_add("s2", \(x = ~s1) x + 1) |>
  pip_add("s3", \(x = ~s2) x * 2)

dim(p)
nrow(p)
ncol(p)
length(p)
length(p) == nrow(p) # TRUE

# A view reports only the number of selected (visible) steps
v <- pip_view(p, step = c("s2", "s3"))
length(v) # 2
nrow(v) # 2
```

`Extract.pipeflow` *Extract or replace parts of a pipeline*

Description

A pipeline can be subset, read from and written to with the usual extract and replace operators, treating it as a table of steps. The extraction operators mirror base R, and the replacement operators update steps (or their properties) in place.

Usage

```
## S3 method for class 'pipeflow'
x[i, j, view = TRUE]

## S3 method for class 'pipeflow'
x[[i, j = NULL]]

## S3 replacement method for class 'pipeflow'
x[[i, j]] <- value

## S3 replacement method for class 'pipeflow'
x[i, j] <- value

## S3 method for class 'pipeflow'
x$i

## S3 replacement method for class 'pipeflow'
x$i <- value
```

Arguments

`x`	A pipeflow pipeline or view object.
`i`	The row or field selector. For `[` and `[<-`, the rows to select: integer row indices, character step names, or a boolean filter expression evaluated in the context of the step table. For `[[`, a step-table column name, a meta field, or — in the two-index form — a step name or integer row index. For `[[<-`, a step name, an integer row index, or one of the meta fields `"name"` and `"view"`. For a view, indices are relative to the covered steps and step names must be part of the view.
`j`	For `[`, an optional character vector of step-table column names to extract. For `[<-`, the single step property to assign. For `[[` and `[[<-`, the column or step property to extract or assign.
`view`	For `[`, if `TRUE` (default), a view referencing the selected steps is returned. If `FALSE`, a new pipeline is returned that includes the selected steps and all their upstream dependencies. Ignored when `j` is provided. With no selection, `v[, view = FALSE]` turns a view into a standalone pipeline of all its covered steps, which is also the way to get a copy of a view, as `pip_clone()` requires a full pipeline.

value The value to assign ([[<-, [<-).

Details

Extract or subset a pipeline (`x[i, j, view = TRUE]`):

`p[...]` selects steps from a pipeline. By default, a lightweight `pip_view()` is returned that references the selected steps without copying them. Set `view = FALSE` to instead get a new, self-contained pipeline that includes all required upstream dependencies.

Three forms of row selection are supported:

- **row indices** (relative to the current selection), e.g. `p[2:3]`,
- **step names**, e.g. `p[c("load", "fit")]`,
- a **boolean filter** that is evaluated in the context of the pipeline's step table, e.g. `p[state == "new"]` or `p[step %in% c("load", "fit") & tags %like% "io"]`.

Boolean filters may reference the columns of the step table (`step`, `fun`, `params`, `depends`, `tags`, `state`, ...) directly as variables. The filter operators of **data.table** (`%like%`, `%ilike%`, `%flike%`, `%plike%`, `%chin%`, `%between%`, `%inrange%` and `%notin%`) are available in the filter without attaching **data.table**. `p[]` returns a copy of the pipeline.

As the columns come first, a variable named like a column can't be used as a selector: with `step <- "load"`, `p[step]` selects all steps (by the `step` column), and `[` warns. Use another name, e.g. `p[steps]`, or `pip_view(p, step = step)`.

Note that `tags` is a list column, so `%like%` matches the pattern against the tags of each step converted to a single string, e.g. `"character(0)"` for a step without tags and `'c("io", "daily")'` for a step with two tags. A pattern can therefore also match untagged steps (e.g. `tags %like% "a"`), and anchored patterns such as `"^io$"` miss steps with several tags. To match tags exactly, use the `tags` argument of `pip_view()`, e.g. `pip_view(p, tags = "io")`.

When `j` is provided, the selected steps are returned as a step-table extraction rather than a pipeline. `j` must be a character vector of column names: `p[, j]` keeps those columns for all steps and `p[i, j]` first selects the rows and then keeps those columns. The result in this case is a `data.table`.

If done on a view, row selection is relative to the covered steps, and step names must be part of the view. For validated, programmatic filters with dedicated arguments (such as matching *any* of a set of tags) use `pip_view()` instead.

Extract values from a pipeline or view (`x[[i, j]]`):

A pipeline can be read like a data.frame of steps: `p[[column]]` returns a column of the step table, and `p[[row, column]]` extracts a single cell. In addition, a few meta fields are accessible by name.

Meta fields:

The following meta fields are available via `p[["..."]]` (or `p$...`):

- `name`: the name of the pipeline.
- `view`: the names of the steps covered by a view, or `NULL` for a full pipeline.
- `pipenv`: the shared inner environment holding the pipeline's state. All views and extracted subsets reference the same environment, so mutations are shared. The step table is available as `p[["pipenv"]][["data"]]` (see also `pip_data()`).

Virtual methods:

All pipeline functions are exposed as *virtual* methods via `p$...` (or `p[["..."]]`). For example, `p$add(...)` is shorthand for `pip_add(p, ...)` and returns the updated pipeline.

Step-table columns:
`p[["column"]]` returns the raw values of a column of the step table, named by the step names. For views, the column is restricted to the steps covered by the view. The two-index form `p[[row, column]]` extracts a single cell, where `row` is an integer row index or a step name.

Assign values to meta fields or step properties (`x[[i, j]] <- value`)**:**

Assigning meta fields:

- `p[["name"]] <- value` – sets the name of the pipeline
- `p[["view"]] <- steps` – allows to define the pipeline view explicitly via a character vector of step names, which is equivalent to `p[steps]` or `p[steps, ]`. Assigning `p[["view"]] <- NULL` clears the view and returns a full pipeline.

Assigning step properties:
The two-index form `p[[step, property]] <- value` provides interactive shortcuts for modifying a single step, where `step` is a step name or an integer row index and `property` selects what to update:

- `p[[step, "step"]] <- newName` – rename the step; same as `pip_rename(p, step, newName)`).
  - `p[[step, "step"]] <- NULL` – remove the step, together with its downstream dependencies; same as `pip_remove(p, step, force = TRUE)`
- `p[[step, "fun"]] <- fun` – replace the step's function; same as `pip_replace()`, that is, tags and execution mode are kept
- `p[[step, "params"]] <- list(...)` – update the step's parameters; same as `pip_set_params(pip_view(p, step = step), list(...), scope = "view")`
- `p[[step, "tags"]] <- tags` – set the step's tags explicitly to `tags` (a character vector); in contrast, `pip_tag()` will just add tags.
  - `p[[step, "tags"]] <- NULL` clears all tags
- `p[[step, "locked"]] <- TRUE|FALSE` – lock or unlock the step; same as `pip_lock()` / `pip_unlock()`
- `p[[step, "exec"]] <- mode` – set the step's execution mode. A new mode marks the step and its downstream steps as `"outdated"`.
- `p[[step, "state"]] <- state` – set the step's state.
- `p[[step, "time"]] <- time` – set the step's time stamp (a single `POSIXct` value).
- `p[[step, "time_start"]] <- time` – set the start time of the step's last run (a single `POSIXct` value, which can be `NA` for a step that never ran).
- `p[[step, "duration"]] <- duration` – set the duration of the step's last run (a single `difftime` value, which can be `NA` for a step that never ran).
- `p[[step, "out"]] <- value` – set the step's stored output (mostly useful for debugging). The downstream steps are marked as `"outdated"`, so that the next run computes them from the new output.

Assigning to any other step-table column, or to a meta field other than `name` and `view`, is not supported.
Views are supported as well. For a view, `i` is interpreted relative to the steps covered by the view: an integer refers to the n-th visible step, and a step name must be part of the view. Because views share the pipeline environment, step properties are written through to the originating pipeline, while `name` only renames the view itself.

Which assignments outdate steps:
Results that may no longer match the steps are marked as `"outdated"`, so that the next `pip_run()` computes them again. As always, steps that never ran stay `"new"`.

- `fun`: the step becomes `"new"` and its downstream steps `"outdated"`.
- `params` and `exec`: the step and its downstream steps, if a value changes.
- `out`: the downstream steps, while the step keeps its state.
- `step` (a new name), `state`, `time`, `time_start`, `duration`, `tags` and `locked` don't change any result and outdate no step.

This holds for all forms of assignment: `[[<-`, `[<-` (which assigns row by row) and assignments through a view, which also outdate the downstream steps that are not part of the view.

Bulk assignment of step properties (`x[i, j] <- value`):

`p[i, j] <- value` assigns the step property `j` to the selected steps (see `[[<-.pipeflow` above for the supported properties), mirroring the row and column selection of the extraction form `p[i, j]`: `i` selects rows (negative indices select all rows but the excluded ones), `j` must be a single property name, and `p[, j] <- value` selects all steps.

- With a *single selected row*, `value` is stored *as-is* (like `p[[i, j]] <- value`).
- With *more than one selected row*, a value of length 1 is *replicated* to all rows and a value with one element per row is assigned *element-wise* (`p[i, j] <- value` behaves like `p[[i[k], j]] <- value[[k]]`). As in base R, longer values are *recycled* if the number of rows is a multiple of the value length, otherwise an error is raised.
- To assign the *same list value* (e.g. a set of tags) to all rows, wrap it in `list(...)`.
- For the `params` property, the value of each row must *itself be a list*, e.g. `p[1:2, "params"] <- list(list(a = 1), list(b = 2))`.

The selected steps are assigned one after another, but the assignment changes all of them or none: if it fails for one step, the pipeline is restored as it was.
If `value` is a `pipeflow` pipeline or view, the writable step properties of its steps are copied to the selected rows: `p[i, ] <- q[i, ]` copies the steps of `q` selected by `i` into the steps of `p` selected by `i`. In this form, `i` is required and `j` must be omitted.

- The rows of both sides are *aligned by position* and must have the same length.
- The step is *renamed* to the source name (a no-op when it is unchanged), and its function, params and execution mode are *replaced* with those of the source (same as `pip_replace()` with the params and `exec` of the source).
- The runtime state and annotation columns (`out`, `state`, `time`, `time_start`, `duration`, `tags`, `locked`) are *overwritten*, locked steps included.
- A copied step refers to the steps that the source step depends on *by name*, also if it was defined with a relative reference such as `~-1`, and these steps are *looked up in* `p`.
- The structural columns `nodeId`, `depends` and `unbound` are not copied but *recomputed* against the steps of `p`.
- If a step can't be copied (e.g. because it refers to a step that is not in `p`), `p` is *left as it was*.

Instead of `p[i, j] <- q`, use `p[i, j] <- q[i, j]` to copy the values of a single property; the one-column table returned by the extraction form is unwrapped and assigned element-wise.

Extract and assign via $ (`x$i / x$i <- value`):

`$` is a shortcut for the one-index form of `[[ / [[<-`: `p$name` is equivalent to `p[["name"]]`, and `p$name <- value` to `p[["name"]] <- value`. This works for meta fields, step-table columns, and the virtual methods (e.g. `p$add(...)` or `p$run()`).

Value

For `[`, a pipeflow view (if `view = TRUE`) or a new pipeflow pipeline (if `view = FALSE`); if `j` is provided, a `data.table` with the selected rows and columns. For `[[`, the extracted step-table column, single cell, or meta field. For the assignment forms, the updated pipeline, invisibly.

Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 5) seq_len(n), tags = c("io", "daily")) |>
  pip_add("square", \(x = ~load) x^2, tags = "model") |>
  pip_add("total", \(x = ~square) sum(x), tags = c("model", "report"))

# By default, `[` returns a view into the selected steps.
p["total"] # view with a single step

# Select by name vector or integer row index
p[c("load", "square")][["step"]]   # view -> "load", "square"
p[1:2, view = FALSE][["step"]]     # pipeline -> "load", "square"

# Boolean filters are evaluated against the step table
p[tags %like% "model"][["step"]]  # "square", "total"

# view = FALSE extracts a pipeline with all upstream dependencies
p[tags %like% "report", view = FALSE][["step"]] # "load", "square", "total"

# No arguments returns a copy of the pipeline
p2 <- p[]
pip_run(p2)
p
p2

# Two-index extraction selects step-table columns by name
p[, "step"]                      # one-column data.table
p[c("load", "square"), "out"]   # selected rows, one column

# Works with views - selection is relative to the covered steps
v <- pip_view(p, tags = "model")
v[1L, "out"]
v[, "step"]
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x + 1)
pip_run(p)

# Meta fields
p[["name"]]              # "pipe"
p[["view"]]              # NULL - not a view
p[["pipenv"]]            # the inner pipeline environment
p[["pipenv"]][["data"]]  # the underlying step table

# Virtual methods
p$add("s3", \(x = ~fit) x * 10)
```

```
p$run()
p$restart   # a function; call p$restart() to request a restart
p$halt      # a function; call p$halt() to halt the current run

# Column access, named by steps
p[["step"]]   # c(load = "load", fit = "fit")
p[["out"]]    # c(load = 1, fit = 2)

# Single cell: p[[row, column]]
p[["fit", "depends"]]    # "load"
p[[2, "state"]]          # state of the second step

# Views behave analogously:
v <- pip_view(p, step = c("load", "fit"))
v[["name"]]              # "pipe"
v[["view"]]              # "load", "fit" - the covered steps

v[["step"]]              # c(load = "load", fit = "fit")
v[["out"]]               # c(load = 1, fit = 2)
v[["fit", "out"]]        # output of the "fit" step
p <- pip_new("pipe") |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load, k = 2) x * k) |>
  pip_add("report", \(x = ~fit) x)

# Assign the pipeline name via its meta field
p[["name"]] <- "demo"

# Replace a step's function (tags and exec mode are kept)
p[["fit", "fun"]] <- \(x = ~load, k = 2) x * k

# Update parameters, tags, locking and the execution mode
p[["fit", "params"]] <- list(k = 5)
p[["fit", "tags"]] <- c("model", "daily")
p[["fit", "locked"]] <- TRUE
p[["fit", "locked"]] <- FALSE
p[["fit", "exec"]] <- "plain"

# Rename a step; dependent steps are updated as well
p[["load", "step"]] <- "read"

# Assign by row index to set the state, time stamp or stored output
p[[2, "state"]] <- "outdated"
p[[2, "time"]] <- Sys.time() - 3600
p[[2, "out"]] <- 42
p

# Restrict to a view covering selected steps, then clear it again
p[["view"]] <- c("read", "fit")
p[["step"]]                  # view -> "read", "fit"
p[["view"]] <- NULL
p[["step"]]                  # full pipeline again
p <- pip_new("pipe") |>
```

```
  pip_add("load", \(n = 5) seq_len(n)) |>
  pip_add("fit", \(x = ~load, k = 2) x * k) |>
  pip_add("report", \(x = ~fit) x)

# Assign a property element-wise or replicated to selected steps
p[c("load", "fit"), "state"] <- c("outdated", "done")
p[1:2, "tags"] <- c("io", "model")
p[1:2, "tags"] <- list(c("new", "tag"))  # same tags for both steps

# All steps at once; read-only columns are rejected
p[, "state"] <- "outdated"
try(p[c("load", "fit"), "depends"] <- "load")  # read-only

# Copy the writable properties from another pipeline
q <- pip_clone(p)
q[c("load", "fit"), "out"] <- list(1:5, 6:10)
p[c("load", "fit"), ] <- q[c("load", "fit"), ]
p[c("load", "fit"), "tags"] <- q[c("load", "fit"), "tags"]
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x + 1)

p$name                # same as p[["name"]]
p$step                # same as p[["step"]]
p$name <- "renamed"   # same as p[["name"]] <- "renamed"

p$run                 # a virtual method; call p$run() to run the pipeline
```

`pipeflow_lgr` *Default logger of pipeflow*

Description

The logging function that `pip_run()` uses by default. It writes one line per message to the console in the form `<level> [<UTC time stamp>]: <msg>`, for example `info [2026-01-01 12:34:56.789 UTC]: Step 1/2 load`.

Usage

```
pipeflow_lgr(level, msg)
```

Arguments

`level`	The log level, one of `"info"`, `"warn"` or `"error"`.
`msg`	The message, a single string.

Details

It is exported so that custom loggers can build on it, for example to add a prefix, or to drop some levels and pass the rest on to the default logger. A logger is any function with the arguments `level` and `msg`. `pip_run()` calls it as `lgr(level = level, msg = msg)`.

Value

NULL, invisibly.

See Also

pip_run() for how loggers are used and how to switch logging off with lgr = NULL. The option pipeflow.lgr sets another default logger for all runs.

Examples

```
pipeflow_lgr("info", "Hello")

p <- pip_new("demo") |>
    pip_add("load", \(n = 3) seq_len(n)) |>
    pip_add("double", \(x = ~load) x * 2)

# Add a prefix to every message
prefixed <- function(level, msg) {
    pipeflow_lgr(level, paste("[demo]", msg))
}
pip_run(p, lgr = prefixed)

# Drop the "info" messages and pass the rest on
quiet <- function(level, msg) {
    if (level != "info") {
        pipeflow_lgr(level, msg)
    }
}
pip_run(p, lgr = quiet, force = TRUE)
```

---

pip_add *Add a step*

Description

Adds a named step to the pipeline. Each step is a function whose parameters either hold constant defaults or reference the output of a prior step using formula notation (~step_name). Dependencies are validated when the step is added.

Usage

```
pip_add(
  x,
  step,
  fun,
  tags = character(0),
  params = list(),
  exec = "auto",
  after = length(x)
)
```

Arguments

x	A pipeflow pipeline object.
step	Unique step name, or a pipeline or view whose steps are copied into x (see 'Copying steps' below), in which case fun, tags, params and exec must not be given. A step name must not start with -, which marks a relative reference like ~-1.
fun	Function to execute for the step. Each function parameter needs a value: a default value in fun or an entry in params. Default values that are simple constants are resolved immediately. Default values that are formulas like ~other_step are treated as dependencies to those steps and resolved to the respective output values at runtime once the step is executed. A reference wrapped in try(), like ~try(other_step), accepts a failed input when the pipeline was run with on_error = "continue" (see section 'Continuing after errors' in pip_run()).
tags	Optional character vector of tags belonging to the step. Can also be adjusted later using [pip_tag()].
params	Optional named list of parameter values. They override the defaults of fun and can be used to give a value to parameters of fun that have no default. As in fun, a formula like ~other_step refers to the output of another step. Names that are no parameters of fun are an error, unless fun has a ... argument, which then takes them. Use cases for params are: 1. Set param values programmatically, e.g. when adding steps in a loop or at runtime. 2. Use existing functions whose parameters have no defaults, e.g. pip_add(p, "avg", mean, params = list(x = ~data, na.rm = TRUE)). 3. Provide extra param values to "mark" dependencies that are defined in pipelines nested in a step, which ensures that it is re-executed when one of the nested param values change. See vignette("v05b-nested-pipeline", package = "pipeflow").
exec	Execution mode for this step, which controls how partitioned input is handled. One of: • "auto": map over the partitions if an input is partitioned, otherwise a single call. Several partitioned inputs must have the same keys in the same order. • "split": single call, then mark the output as partitioned. • "reduce": single call, only valid with partitioned input (or, via ~try(), a failure object). • "plain": single call, only valid with non-partitioned input. A mapped step that fails for some partition keys in a run with on_error = "continue" keeps the outputs of the other keys (see section 'Continuing after errors' in pip_run()).
after	Optional position after which the new step should be inserted (defaults to last position). Can be a step name or an integer index. If set to 0, the new step will be inserted at the beginning of the pipeline.

Details

If `after` was specified, the new step will be inserted after the given step or position. Be aware that in contrast to adding a step at the end, inserting a step in the middle is a rather expensive operation as it requires re-wiring parts of the internal pipeline structure, especially if the new step is inserted at an early position.

Copying steps:

If an entire pipeline or view is provided as `step`, its steps are added in their order, with their functions, current params, tags and exec modes. Upstream steps of views are *not* copied, which means that references (including relative references like `~-1`) are resolved (and must be valid) within `x`. If this blocks a step from being added (e.g., its name already exists in `x`, or it refers to a step not existing in `x`), `x` is left unchanged. Outputs and locks are not copied either. To copy a view `v` *with* its upstream steps, use `pip_add(x, v[, view = FALSE])` (see `[`), to bind whole pipelines, rbind().

Params with the same name:

A param name is meant to have one value in the whole pipeline.Therefore, `pip_add()`, `pip_replace()` and rbind() signal a warning if the default param value of a new step differs from the value of the same param in the other steps. To give steps different values on purpose, give the params different names, silence the warning with `suppressWarnings(..., classes = "pipeflow_param_conflict")`, or turn it off with `options(pipeflow.warn_param_conflicts = FALSE)`.

The pipeline as `.self`:

A step function can refer to the pipeline that is being run as `.self` without declaring it as an argument, for example to set parameters or add steps at runtime, or to call `.self$restart()` or `.self$halt()` (see `pip_run()`). `.self` is always the full pipeline, also when a view is run. pipeflow provides it each time the step runs, but only if the body of the step function contains the name `.self` (also within functions defined in the body). Indirect access, such as `get(".self")`, is not detected.

A step that changes its own function, params or exec mode via `.self`, ends the run as `"outdated"`. The same is true for other steps changed via `.self`. As these steps and their downstream steps run again in the next run, `pip_run()` warns (class `pipeflow_outdated_before`) if a step changes itself or a done step before it, unless the step halts the run or the run is restarted. Lastly, `.self` is only valid while the step runs and released once the step was returned. For this reason, a step that returns a function referring to `.self` fails with `object '.self' not found` when called later.

Outputs that keep an environment:

An output that keeps the environment it was made in, such as a function or a ggplot made in the step, also keeps the environment of the step function. If that function was defined where the pipeline is (e.g. in a function that builds it), saving the output saves the whole pipeline too. Define step functions in a package or at top level, or return a form without an environment (e.g. an image instead of a ggplot).

Value

The updated pipeline, invisibly.

Examples

```
# --- Tags, and view filtering ---
p <- pip_new("analysis") |>
  pip_add("load", \(n = 5) seq_len(n), tags = c("io", "raw")) |>
  pip_add("clean", \(x = ~load) x * 2, tags = c("io", "process")) |>
  pip_add("fit", \(x = ~clean) sum(x), tags = c("model", "core", "daily")) |>
  pip_add("report", \(x = ~fit) paste("result:", x), tags = "report")

pip_run(p)
p

# Filter by tag using pip_view — keeps steps with any matching tag
pip_view(p, tags = "daily")
pip_view(p, tags = "core")
pip_view(p, tags = c("raw", "report"))

# --- Split / reduce execution modes ---
q <- pip_new("split-demo") |>
  pip_add("data", \(x = iris) x) |>
  pip_add("split", \(x = ~data) split(x, x$Species),
    exec = "split"
  ) |>
  pip_add("stats", \(x = ~split) summary(x)) |>
  pip_add("combine", \(x = ~stats) do.call(rbind, x),
    exec = "reduce"
  )

pip_run(q)
q[["stats", "out"]]   # partitioned list — one summary per species
q[["combine", "out"]] # combined table

# --- Insert a step at a specific position with 'after' ---
p2 <- pip_new("insert-demo") |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x + 1)
p2
pip_add(p2, "clean", \(x = ~load) x * 2, after = "load")
p2  # "load", "clean", "fit" — inserted after "load"

# after = 0 inserts the new step at the very beginning
pip_add(p2, "preload", \(x = 1) x, after = 0)
p2

# --- Copy steps of another pipeline, without their upstream steps ---
other <- pip_new("other-data") |> pip_add("load", \(x = 10) x)
pip_add(other, p2["fit"]) # "fit" now refers to "load" of `other`
pip_run(other)
other[["fit", "out"]] # 11

# --- Provide parameter values with 'params' ---
# 'params' override the defaults of the function ...
p3 <- pip_new("params-demo") |>
```

```
  pip_add("data", \(x = c(1, NA, 3)) x) |>
  pip_add("scale", \(x = ~data, factor = 1) x * factor,
    params = list(factor = 10)
  ) |>
  # ... and give values to parameters without default, e.g. of mean()
  pip_add("avg", mean, params = list(x = ~scale, na.rm = TRUE))
pip_run(p3)
p3
p3[["avg", "out"]] # 20

# Extra params are passed via `...`
p4 <- pip_new("dots-demo") |>
  pip_add("load", \(x = 1, ...) c(x, ...), params = list(size = 42))
pip_run(p4)
p4[["load", "out"]] # c(1, size = 42)
```

`pip_clone` *Clone a pipeline*

Description

Creates an independent copy of the pipeline. Changes to the cloned pipeline do not affect the original pipeline, and vice versa.

Usage

```
pip_clone(x, name = NULL, out = NULL)
```

Arguments

`x`	A pipeflow pipeline object.
`name`	Optional name for the cloned pipeline. If `NULL`, the original name is used.
`out`	`NULL` (all outputs) or a character vector of the steps whose outputs the copy keeps (`character(0)`: none), see "Copies without some outputs".

Value

A cloned pipeflow pipeline object.

Copies without some outputs

With `out`, the copy keeps the outputs of the named steps only, e.g. to send the results of a run back from another R process (a parallel worker) without the data of the steps in between. Everything else is copied: the steps, their params, times and failures, and the states except for these changes:

- A step that was `"done"` and loses its output becomes `"outdated"`, so the copy never passes on a missing output: running it runs these steps again, and with them the steps after them, also those that kept their output.

- Failed steps keep their output (the failure, or for a mapped step the outputs and failures of its keys), which tells why the steps after them are not done.
- Locked steps keep their output, as a run passes it on.

The copy still holds the step functions, with their environments: a function defined inside another function keeps that function's frame and the data in it (see "Outputs that keep an environment" in `pip_add()`).

Examples

```
p <- pip_new("original") |>
  pip_add("s1", \(x = 1) x) |>
  pip_add("s2", \(x = ~s1) x + 1)

# Clone produces a fully independent copy
cp <- pip_clone(p, name = "copy")
pip_add(cp, "s3", \(x = ~s2) x * 10)

# As a result, the clone has the new step ...
cp

# ... while the original is left unchanged
p

# A copy with the output of the last step only, e.g. to return from a worker
p <- pip_new("worker") |>
  pip_add("data", \(n = 1e5) rnorm(n)) |>
  pip_add("summary", \(x = ~data) summary(x))
pip_run(p)
result <- pip_clone(p, out = "summary")
result[["summary", "out"]]
is.null(result[["data", "out"]])

# The step without output is outdated, so a run computes it again
result
```

`pip_collect` *Collect step outputs*

Description

Returns the outputs of the steps in a pipeline or view. With `by = "step"` (the default) the result is a named list of step outputs. With any other column, the outputs are grouped by the values of that column (typically `"tags"`).

Usage

```
pip_collect(x, by = "step", as.table = FALSE, simplify = TRUE)
```

Arguments

x	A pipeflow pip or view.
by	Single step-table column name to group by.
as.table	If TRUE, return a data.table instead of a named list.
simplify	If TRUE (default), if the list of collected outputs contains exactly one step per group, the result is flattened by one level, otherwise it is returned as a grouped list.

Details

The result is one of two shapes:

- **Flat**: a named list of step outputs, e.g. `list(s1 = 1, s2 = 2)`.
- **Grouped**: a named list of such lists, one per group, e.g. `list(io = list(s1 = 1, s2 = 2), model = list(s3 = 4))`.

With `simplify = TRUE` (the default) the result is flat whenever every group has exactly one step, as is always the case for `by = "step"`. Otherwise, or with `simplify = FALSE`, it is grouped. For list columns such as `tags`, a step with several entries contributes to every group, and steps without an entry are omitted. Use `pip_view()` to narrow the selection first.

Value

A named list of outputs

Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x, tags = "io") |>
  pip_add("clean", \(x = ~load) x + 1, tags = "io") |>
  pip_add("model", \(x = ~clean) x * 2, tags = "model")
pip_run(p)

# By default, a flat named list with one entry per step
pip_collect(p)

# The same output as a data.table
pip_collect(p, as.table = TRUE)

# Group the outputs by tag ...
pip_collect(p, by = "tags")

# ... which is equivalent to
list(
  io = pip_view(p, tags = "io") |> pip_collect(),
  model = pip_view(p, tags = "model") |> pip_collect()
)

# Grouped table output
pip_collect(p, by = "tags", as.table = TRUE)
```

```
# Keep single-step groups nested
pip_collect(p, simplify = FALSE)

# Collect output from a view
v <- p[step %in% c("clean", "model"), ]
pip_collect(v)
pip_collect(v, as.table = TRUE)
```

`pip_data` *Access the underlying step table*

Description

This is a convenience wrapper for accessing the internal `data.table`, which normally is reachable via `p[["pipenv"]][["data"]]`.

Usage

```
pip_data(x)
```

Arguments

`x`	A pipeflow pipeline or view.

Details

The internal `data.table` is holding the pipeline steps, one row per step. Unless you know what you are doing, this table should not be modified directly, as this can corrupt the pipeline including any views that are derived from it. If you want to experiment, consider cloning the pipeline first with `pip_clone()`.

Value

The underlying step table as a `data.table`.

Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 5) seq_len(n)) |>
  pip_add("model", \(x = ~load) sum(x))

pip_data(p)
```

pip_graph *Build pipeline graph data*

Description

Builds graph data (nodes and edges) describing the pipeline's step structure, suitable for visualisation with `visNetwork::visNetwork()`, or the graph as a mermaid flowchart.

Usage

```
pip_graph(x, include_upstream = FALSE, format = c("visNetwork", "mermaid"))
```

Arguments

`x`	A pipeflow pip or view.
`include_upstream`	Logical. Only relevant for views. If `TRUE`, add all upstream dependencies of selected steps.
`format`	The output format, either `"visNetwork"` or `"mermaid"`.

Details

Node shapes reflect execution mode:

- `auto/plain`: `hexagon`
- `reduce`: `dot`
- `split`: `star`

Node colors reflect the step state (`new`, `done`, `outdated` or `failed`). Hovering over a node shows its `title`: the state, the exec mode, the tags, whether it is locked, the start of its last run (and its duration if it succeeded), the failed keys of a mapped step, its parameters and the steps it depends on.

An edge is dashed if the step refers to the other step only with `~try()`, i.e. if the step also runs when the other step failed (see `pip_run()`).

Mermaid:

With `format = "mermaid"`, the graph is returned as the text of a mermaid flowchart. It renders without extra packages in a code block marked `mermaid`, e.g. on GitHub or in a Quarto document. It has the same colors and dashed edges as the visNetwork graph, but no hover text. The shapes are a hexagon for `auto/plain`, a trapezoid that widens for `split` and one that narrows for `reduce`.

Value

For `format = "visNetwork"`, a named list with two `data.frames`: `nodes` (columns `id`, `label`, `shape`, `color` and `title`) and `edges` (columns `from`, `to`, `arrows` and `dashes`). For `format = "mermaid"`, a single string with the mermaid flowchart.

Examples

```
p <- pip_new()
pip_add(p, "load", \(x = 1) x, tags = "io")
pip_add(p, "clean", \(x = ~load) x + 1, tags = "io")
pip_add(p, "fit", \(x = ~clean) x * 2, tags = "model")
pip_add(p, "report", \(fit = ~try(fit)) class(fit))

graph <- pip_graph(p)
graph$nodes # data.frame: id, label, shape, color, title
graph$edges # data.frame: from, to, arrows, dashes

# For a view, include_upstream = TRUE adds upstream deps to the graph
v <- pip_view(p, step = "fit")
pip_graph(v, include_upstream = TRUE)

if (require("visNetwork", quietly = TRUE)) {
  do.call(what = visNetwork::visNetwork, args = graph)
}

# The graph as a mermaid flowchart
cat(pip_graph(p, format = "mermaid"))
```

---

pip_inner *Inner pipelines of a pipeline function step*

Description

Returns the inner pipelines that the function of `step`, made by `as.function.pipeflow()`, has kept, e.g. to debug an inner run or to look at the steps behind the returned output.

Usage

```
pip_inner(x, step)
```

Arguments

`x`	A pipeflow pipeline object, the "outer" pipeline.
`step`	Name of a step of `x` whose function was made by `as.function.pipeflow()`.

Value

For a step that is mapped over partitions, a list of pipelines named by partition key, otherwise the pipeline of the last call of the step. The pipelines are copies that share the values of the kept ones, so don't modify the values in place. An empty list is returned if the function has kept nothing (yet), e.g. before the first run or with `cache = FALSE`.

See Also

`as.function.pipeflow()`

Examples

```
inner <- pip_new("inner") |>
  pip_add("data", \(input_df = NULL) input_df) |>
  pip_add("fit", \(data = ~data) lm(mpg ~ wt, data = data))

outer <- pip_new("outer") |>
  pip_add("df", \(df = mtcars) df) |>
  pip_add("by_cyl", \(df = ~df) split(df, df$cyl), exec = "split") |>
  pip_add(
    "fits",
    as.function(inner, out = "fit"),
    params = list(input_df = ~by_cyl)
  )
pip_run(outer, lgr = NULL)

fits <- pip_inner(outer, "fits")
names(fits)
fits[["6"]]
```

---

pip_lock *Lock or unlock steps*

Description

Locks or unlocks all steps of a pipeline, or a subset of steps defined by a view. Locked steps are skipped during `pip_run()` and are protected against modification from `pip_set_params()`, `pip_tag()` or `pip_untag()`. Calling `pip_unlock()` removes the lock again.

Usage

```
pip_lock(x)

pip_unlock(x)
```

Arguments

x	A pipeflow pip or view.

Details

A locked step passes its stored output to its dependents. Locking a step that is not `"done"` therefore marks its downstream steps (except locked ones) as `"outdated"`, as they may have run with a failure of the step instead (see section 'Continuing after errors' in `pip_run()`).

Value

The updated pipeline or view, invisibly.

Examples

```
p <- pip_new() |>
  pip_add("x", \(x = 1) x) |>
  pip_add("y", \(y = 2) y) |>
  pip_add("sum", \(x = 1, y = 2) x + y)
(pip_run(p))
p[["sum", "out"]] # 3

# Lock "sum" step via a view so it cannot be overwritten
pip_set_params(p, params = list(x = 10, y = 20))
p[["sum", "params"]] # x = 10, y = 20
pip_lock(p["sum", ])
(pip_run(p))
p[["sum", "out"]] # still 3

# Note that locking also prevents any parameter updates
pip_set_params(p, params = list(x = 100, y = 200))
p[["x", "params"]] # x = 100
p[["y", "params"]] # y = 200
p[["sum", "params"]] # still x = 10, y = 20

# Unlock everything to allow updates again
pip_unlock(p)
pip_set_params(p, params = list(x = 100, y = 200))
(pip_run(p))
```

`pip_new` *Create a pipeline*

Description

Creates a new, empty pipeline. Add steps with `pip_add()` and execute them with `pip_run()`.

Usage

```
pip_new(name = "pipe")
```

Arguments

`name`	The name of the pipeline used for display and logging.

Value

A pipeflow pipeline object.

Saving pipelines

A pipeline can be saved with `saveRDS()` and read back with `readRDS()` (or serialized in other ways, e.g. to send it to another R process), with its steps, parameters, outputs and states. The dependency graph of the steps is kept outside of R and is not saved, but it is rebuilt from the steps the first time it is needed. Views that are saved in the same object as their pipeline, as in `saveRDS(list(p, v), file)`, are read as views of the read pipeline. A view that is saved on its own is read with its own copy of the pipeline.

Examples

```
p <- pip_new("demo") |>
    pip_add("numbers", \(n = 5) seq_len(n)) |>
    pip_add("squared", \(x = ~numbers) x^2) |>
    pip_add("total",   \(x = ~squared) sum(x))
p
str(p)
p[["name"]]  # "demo"
p[["view"]]  # initially NULL

# Inner pipeline environment (for advanced usage)
ls(p[["pipenv"]])                # shows "data"
ls(p[["pipenv"]], all = TRUE)    # also shows hidden variables

# Save the pipeline and read it back
file <- tempfile(fileext = ".rds")
saveRDS(p, file)
p2 <- readRDS(file)
pip_run(p2)
p2
unlink(file)
```

---

pip_output_value *Annotated step outputs*

Description

`pip_output_value()` is a hook to annotate or postprocess the output of a step, for example to stamp a result with the step it came from or the time it was computed. It is applied to the output right after the step function returns, and its result is what is stored as the step's output. The default method returns `x` unchanged. Methods are meant for your own classes, not for base classes such as `numeric`.

Usage

```
pip_output_value(x, ctx = list(), ...)
```

Arguments

`x`	The output value of a step, or one of its partitions if the output is partitioned.
`ctx`	Named list describing the context of the value: • `step`: the name of the step, as it was before the step ran. • `key`: the partition key, or `NULL` if the output is not partitioned. • `time`: the time stamp of the step's run, which is the same for all partition keys and is also stored as the step's `time`.
`...`	Currently unused, reserved for future extensions. Methods should accept it.

Details

The generic is applied once to the whole output, except for a partitioned output (see `exec` in `pip_add()`): there it is applied once per partition, with the key in `ctx$key`. To annotate an output element-wise, write a method that walks over the elements, e.g. for a plain list:

```
pip_output_value.list <- function(x, ctx = list(), ...) {
    x[] <- lapply(x, FUN = pip_output_value, ctx = ctx)
    x
}
```

It is *not* applied to the outputs of failed steps, which remain the failure condition, nor to the inputs a step receives from other steps. An error in a method is a step error and, for a partitioned output, names the partition key.

Value

The value that is stored as the step's output.

See Also

`pip_param_value()`, `pip_run()`

Examples

```
# Stamp every output with the step it came from
stamped <- function(x) structure(x, class = "stamped")

pip_output_value.stamped <- function(x, ctx = list(), ...) {
    attr(x, "step") <- ctx[["step"]]
    x
}
registerS3method("pip_output_value", "stamped", pip_output_value.stamped)

p <- pip_new() |>
    pip_add("make", \(n = 3) stamped(seq_len(n)))
pip_run(p, lgr = NULL)
attr(p[["make", "out"]], "step")
```

`pip_param_value` *Annotated parameter values*

Description

`pip_param_value()` is an extension point for handling non-trivial parameter values (e.g. values with units or labels, or parameter objects that carry UI metadata) as step parameters. It can be thought of as a hook that allows you to preprocess or unwrap parameter values before they are passed to the step function. When a step runs, the generic is applied to each unbound argument of the step, that is, to the values that come from `params` or from the defaults of the step function, and its result is what the step function receives. The stored parameters are not changed, and a method should not change them either: it is called with the stored value, so a method that modifies a reference-backed `x` such as an environment or an R6 object in place changes the step's parameter on every run.

Usage

```
pip_param_value(x, ...)

pip_param_value(x, ...) <- value
```

Arguments

`x`	The parameter value as stored in the step.
`...`	For `pip_param_value()`, currently unused, reserved for future extensions. For `` `pip_param_value<-`() ``, the options given to `...` of `pip_set_params()`, which can't be named like its arguments (e.g. `scope`). Methods should accept it.
`value`	The plain value to set into the stored value `x`.

Details

The default method returns `x` unchanged. Methods are meant for your own classes, not for base classes such as `numeric` or `character`. Note that this is a hook that unwraps a stored value, not an accessor: use `pip_get_params()` to read parameters from a pipeline.

The generic is *not* applied to the outputs of other steps (arguments defined as `~step`), to partitions or to failure objects. An error in a method is an error of the step and is handled like any other step error, including `pip_run(on_error = "continue")`.

`` `pip_param_value<-`() `` is the write side of the same extension point. It turns a plain value into the value that is stored in the step, so that `pip_set_params()` can keep the annotations of a stored object when it is given a plain value. Dispatch is on `x`, the value as it is currently stored, and `value` is the requested value. The default method just returns `value`. A method decides what a plain `value` and what an object of its own class means, for example whether to merge it into the stored object or to use it as is.

Like `pip_param_value()`, this is a hook, not an accessor: `x` is a stored value, and the parameters of a pipeline are changed with `pip_set_params()`. The two sides belong together, so that applying `pip_param_value()` to the result of the setter gives back the plain value that was set:

```
pip_param_value(`pip_param_value<-`(x, value = v))   # v
```

pip_set_params() computes the stored value of every affected step *before* it changes any parameters in the pipeline. If one or more methods fail, an error of class `pipeflow_set_params_error` reports all failures. This two-phase approach ensures that either all parameter changes are applied successfully or, in case of an error, the pipeline stays unchanged. For that to hold, a method should not modify `x` in place as this would change pipeline parameters directly. A reference-backed `x` such as an environment or an R6 object therefore should be copied first.

Value

The value the step function receives for the argument.

See Also

pip_set_params(), pip_output_value(), pip_run()

Examples

```
# A value that carries annotations
my_number <- structure(
    list(value = 1.23, label = "Number", descr = "I am an example"),
    class = "foo"
)

# Step functions then simply see the plain value
pip_param_value.foo <- function(x, ...) x$value
registerS3method("pip_param_value", "foo", pip_param_value.foo)

p <- pip_new() |>
    pip_add("double", \(x = 1) x * 2)
pip_set_params(p, list(x = my_number))
pip_run(p, lgr = NULL)
pip_collect(p)

# The write side keeps the annotations when a plain value is set
`pip_param_value<-.foo` <- function(x, ..., value) {
    if (inherits(value, "foo")) {
        return(value)
    }
    x$value <- value
    x
}
registerS3method("pip_param_value<-", "foo", `pip_param_value<-.foo`)

pip_set_params(p, list(x = 42))
pip_get_params(p) # still a "foo" object, now holding 42
pip_run(p, lgr = NULL)
pip_collect(p)
```

pip_remove *Remove a step*

Description

Removes a pipeline step by its name. If other steps depend on it, an error is given and the removal of the step is blocked, unless `force` was set to `TRUE`, which will remove the selected step together with all its downstream dependent steps.

Usage

```
pip_remove(x, step, force = FALSE)
```

Arguments

`x`	A pipeflow pip or view
`step`	`string` the name of the step to be removed.
`force`	`logical` if `TRUE` the step is removed together with all its downstream dependencies.

Value

The updated pipeline or view, invisibly.

Note

If called on a view, the step to be removed must be part of the view. The view, as well as any other view on the same pipeline, stays valid: the removed steps just drop out of it.

Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("transform", \(x = ~load) x * 2) |>
  pip_add("model", \(x = ~transform) x + 10)

# Removing a leaf step (nothing depends on it) works directly
pip_remove(p, "model")
p                        # "load", "transform"

# Trying to remove a step that others depend on raises an error:
# pip_remove(p, "load")  # Error!

# If a view is passed, the step must be part of the view.
v <- pip_view(p, step = "transform")
try(pip_remove(v, "load"))  # Error: "load" is not part of the view

# Views stay valid, the removed steps just drop out of them
```

```
v <- pip_view(p, step = c("load", "transform"))
pip_remove(v, "transform")
v[["step"]]             # "load"
p                       # "load"

# force = TRUE removes the step and all its downstream dependents
pip_remove(p, "load", force = TRUE)
p                       # pipeline is now empty
```

`pip_rename` *Rename a step*

Description

Renames the selected step and updates dependency references in downstream steps.

Usage

```
pip_rename(x, from, to)
```

Arguments

`x`	A pipeflow pip or view
`from`	Existing step name
`to`	New step name, which must not start with `-` (see `pip_add()`)

Value

The updated pipeline, invisibly.

Examples

```
p <- pip_new() |>
  pip_add("s1", \(x = 1) x) |>
  pip_add("s2", \(x = ~s1) x + 1)           # "s2" depends on "s1"

# Downstream dependency references are updated automatically
pip_rename(p, from = "s1", to = "load_data")
p

# Trying to rename to an existing step name raises an error:
try(pip_rename(p, "load_data", to = "s2"))  # step 's2' already exists!

# If a view is passed, the step must be part of the view
v <- pip_view(p, step = c("load_data", "s2"))
pip_rename(v, from = "load_data", to = "input")
p[["step"]]                                 # "input", "s2"

v2 <- pip_view(p, step = "s2")
try(pip_rename(v2, from = "input", to = "data"))
```

`pip_replace` *Replace a step*

Description

Replaces a step's function while keeping it in the same position in the pipeline. Downstream steps that have run before are automatically marked as outdated and will re-run on the next `pip_run()`.

Usage

```
pip_replace(
  x,
  step,
  fun,
  tags = NULL,
  params = list(),
  exec = NULL,
  keep_params = FALSE
)
```

Arguments

`x`	A pipeflow pipeline or view object.
`step`	Name of the step to replace.
`fun`	Function to execute for the step. Each function parameter needs a value: a default value in `fun` or an entry in `params`. Default values that are simple constants are resolved immediately. Default values that are formulas like `~other_step` are treated as dependencies to those steps and resolved to the respective output values at runtime once the step is executed. A reference wrapped in `try()`, like `~try(other_step)`, accepts a failed input when the pipeline was run with `on_error = "continue"` (see section 'Continuing after errors' in `pip_run()`).
`tags`	Optional character vector of tags belonging to the step. If `NULL` (the default), the current tags of the step are kept. Use `character(0)` to remove all tags.
`params`	Optional named list of parameter values. They override the defaults of `fun` and can be used to give a value to parameters of `fun` that have no default. As in `fun`, a formula like `~other_step` refers to the output of another step. Names that are no parameters of `fun` are an error, unless `fun` has a `...` argument, which then takes them. Use cases for `params` are: 1. Set param values programmatically, e.g. when adding steps in a loop or at runtime. 2. Use existing functions whose parameters have no defaults, e.g. `pip_add(p, "avg", mean, params = list(x = ~data, na.rm = TRUE))`. 3. Provide extra param values to "mark" dependencies that are defined in pipelines nested in a step, which ensures that it is re-executed when one of the nested param values change. See `vignette("v05b-nested-pipeline", package = "pipeflow")`.

exec — Execution mode for the step, see pip_add(). If NULL (the default), the current execution mode of the step is kept.

keep_params — Logical. If TRUE, the new step keeps the current values of the unbound params of the old step that are also parameters of the new function, unless params give them or their default in the new function is a step reference (~step). The current values include the defaults of the old function, so a changed default of a kept param has no effect. Bound params are never kept, so replacing a step still unbinds them, and neither are params that went into the ... of the old function: give them in params again. If FALSE (the default), all params are taken from the new function and params.

Details

The step keeps its tags and execution mode by default, while output, state and lock status are always reset, like for a freshly added step. The parameters are taken from the new function and params, unless keep_params = TRUE.

- tags = NULL and exec = NULL mean *keep the current value*. To change them, pass explicit values.
- tags = character(0) *removes all tags* and exec = "auto" *resets* the execution mode to its default.

As in pip_add(), a warning is signalled if a param of the new function has another value than the same param in other steps (see Params with the same name in pip_add()). So if you set a param with pip_set_params() and replace the step, the new function's default causes this warning, unless keep_params = TRUE keeps the value that was set.

Value

The updated pipeline, invisibly.

Examples

```
p <- pip_new() |>
    pip_add("load", \(n = 5) seq_len(n)) |>
    pip_add("double", \(x = ~load) x * 2)
pip_run(p)
p

# Replace "load" — downstream steps are automatically marked "outdated"
pip_replace(p, "load", \(n = 3) seq_len(n))
p

# Re-run to bring everything up to date
pip_run(p)
p

# If a view is passed, the step must be part of the view
v <- pip_view(p, step = "double")
pip_replace(v, "double", \(x = ~load) x * 3)
p[["state"]]                                # "double" is "new" again
```

```
try(pip_replace(v, "load", \(n = 2) seq_len(n)))

# Tags and execution mode are kept unless set explicitly
pip_tag(p, "io")
pip_replace(p, "load", \(n = 4) seq_len(n))
p[, "tags"]
pip_replace(p, "load", \(n = 4) seq_len(n), tags = character(0))
p[, "tags"]

# Keep the params set before
pip_set_params(p, list(n = 10))
pip_replace(p, "load", \(n = 4) rev(seq_len(n)), keep_params = TRUE)
p[["load", "params"]]                       # n = 10, not 4
```

`pip_reset` *Reset a pipeline to its initial state*

Description

Resets all unlocked steps of a pipeline (or a subset of steps defined by a view) to state `"new"` and clears their outputs, so a subsequent `pip_run()` re-executes the cleaned steps from scratch. Parameters, tags, and locked flags are left unchanged. Further effects:

Usage

```
pip_reset(x)
```

Arguments

`x`	A pipeflow pip or view. If a view is given, only the steps covered by the view are reset.

Details

- *Downstream steps* that have run are marked `"outdated"`, as they ran with the cleared outputs.
- Any pending *restart counters* are cleared.
- If *all steps* of the pipeline are selected, the *run state* is set to `"ready"` and the time of the last run is cleared. A view that covers only some steps leaves both alone, as they describe the whole pipeline.

Locked steps are skipped: their state and output are preserved, also if they are downstream of a reset step. If all selected steps are locked, a message is shown and nothing is changed.

Value

The updated pipeline or view, invisibly.

Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 3) seq_len(n)) |>
  pip_add("square", \(x = ~load) x^2)

pip_run(p)
p[["state"]] # "done", "done"

# Locked steps keep their state and output when resetting
pip_lock(pip_view(p, step = "square"))
pip_reset(p)
p
p[["state"]] # "new", "done"
p[["out"]]   # NULL, (x^2 result)

pip_unlock(p)
pip_reset(p)
p
p[["state"]] # "new", "new"
p[["out"]]   # NULL, NULL

# Resetting a step outdates its downstream steps
pip_run(p)
pip_reset(pip_view(p, step = "load"))
p[["state"]] # "new", "outdated"
```

`pip_run` *Run a pipeline*

Description

Executes all pending steps in order. On repeated runs, steps that are already `"done"` are skipped and only steps that are still `"new"` or were marked `"outdated"` (because one of their dependencies changed) are executed. Use `force = TRUE` to re-execute every step.

Usage

```
pip_run(
  x,
  lgr = getOption("pipeflow.lgr", default = pipeflow_lgr),
  force = FALSE,
  progress = NULL,
  on_error = c("stop", "continue"),
  map = NULL
)
```

Arguments

x	A pipeflow pip or view
lgr	A logging function with the arguments level ("info", "warn" or "error") and msg, which is called as lgr(level = level, msg = msg). The default is the option pipeflow.lgr if set, otherwise pipeflow_lgr(), which can also be used to build custom loggers. To suppress logging, you can set lgr = NULL, or for all runs options(pipeflow.lgr = function(level, msg) NULL). Setting the option to NULL removes it, so that pipeflow_lgr() is used again.
force	Logical indicating if all steps should be forced to run, regardless of whether they are outdated or not.
progress	Optional callback of the form function(value, detail) called before each step, with the position of the step in the run and its name. If it returns FALSE, the run is halted before the step, as by .self$halt() (see section 'Runtime control flow via restart and halt' below).
on_error	What to do when a step fails: "stop" (the default) aborts the run, "continue" goes on with all steps that do not depend on a failed step (see section 'Continuing after errors' below).
map	Optional function like lapply() that runs the calls of a step mapped over partitions (exec = "auto", see pip_add()), for example parallel::mclapply or future.apply::future_lapply. It is called as map(X, FUN) and must return a list of the results of FUN for the elements of X. The default NULL calls them one after another (see section 'Mapping in parallel' below).

Details

Step states:

A step can take the following states:

- new: the step has been added but not yet executed, or was reset via pip_reset().
- outdated: a step can get outdated in the following scenarios:
  - one of the step's parameters were changed (e.g. via pip_set_params())
  - the step's execution mode was changed (p[[step, "exec"]] <- mode)
  - a step it depends on was re-executed, replaced or reset, got a new execution mode, or its output was set (p[[step, "out"]] <- value)
  - a step it depends on was executed, but the run did not reach the step itself, either on purpose (see section 'Running views' below) or because the run was aborted early due to a failed step
- done: the step was executed successfully with its current inputs
- failed: the step raised an error during its last execution

Only steps that have run before can become outdated: a step in state "new" stays "new" until it is executed. A "done" step is skipped unless force = TRUE was set. For all other states, the step will be re-executed in the next pip_run(). A step whose input is still "new" (and not locked), for example a step that an earlier step added via .self during the run, is skipped and keeps its state, as the input has no output yet.

Time stamps:

The step table has two time stamps and a duration per step:

- `time_start`: the start of the step's last run. It is set each time the step is run, also if the run fails. It is `NA` for a step that has not run since it was added, replaced (see `pip_replace()`) or reset (see `pip_reset()`). A skipped step (because it is `"done"`, locked or has a failed or new input) keeps its value.
- `duration`: how long the step's last run took (a `difftime` in seconds), also if the run failed. It is `NA` and kept in the same cases as `time_start`.
- `time`: the time the step was added or replaced, or the end of its last successful run. A failed run does not change it. It is the time stamp that `pip_output_value()` gets in `ctx`.

A step ran in a run if its `time_start` is not before a `Sys.time()` taken before the run, and it succeeded if its state is `"done"` and `time >= time_start`.

Running views:

When `x` is a view, the requested rows are run together with their upstream dependencies, so the steps covered by the view are brought up to date even if their inputs come from steps outside the view. The rest of the pipeline is not executed; downstream steps of executed steps are marked `"outdated"`. Steps whose inputs were not re-executed (for example, because they were already `"done"`) keep their state.

Runtime errors:

By default, a failing step stops the run: the step gets state `"failed"` and its error is stored as its output, the pipeline run state becomes `"failed"`, and steps after it that were not reached are marked `"outdated"`, so the next run retries them.

The error message names the step and, for a mapped step, the partition key. The error has class `pipeflow_step_error` and keeps the original error in its field `parent`: `tryCatch(pip_run(p), pipeflow_step_error = function(e) e$parent)`.

Continuing after errors:

With `on_error = "continue"`, the run does not stop at a failed step:

- The failed step gets state `"failed"`, and its error is stored as its output (as above).
- Steps that depend on a failed step are not run, and neither are their own dependents. All other steps are run.
- To run a step despite a failed input, refer to that input via `try()`, e.g. `x = ~try(fit)` (see `pip_add()`). The argument then receives a failure object (an error condition) whose message tells which step failed, e.g. `"step 'fit' failed: negative"`.
- Locked steps are never run, so a failed input does not affect them.

A step mapped over partitions fails only for the keys whose call failed:

- Its output keeps the results of the other keys and holds a failure object for each failed key, e.g. `"step 'fit' failed: key 'b': negative"`. The step gets state `"failed"`, and printing the pipeline shows its output as e.g. `<partitioned: 1/3 failed>`.
- A mapped step after it is not run for the failed keys, whose output is then a failure object too (`"step 'show' was not run: step 'fit' failed: key 'b': negative"`), unless it takes the input via `try()`.
- A step that is not mapped (e.g. `exec = "reduce"`) is not run, unless it takes the input via `try()`. Then it receives the results of all keys, including the failure objects.

At the end of the run, the pipeline run state is "continued", and a warning of class pipeflow_run_failed lists the failed steps (with the number of failed keys) and the steps that were not run.

The next run retries the failed steps and the steps that were not run. Once a failed step succeeds, the steps after it are run again, including those that received failure objects. A mapped step is only run again for its failed keys, unless something changed that affects all keys (its params or exec mode, an input step, or force = TRUE). See vignette("v05c-error-handling", package = "pipeflow").

Mapping in parallel:

With map, the keys of a mapped step are run by map, for example in parallel processes. This applies to all mapped steps of the run, but not to inner pipelines of functions made by as.function.pipeflow(), which run in parallel only if made with cache = FALSE (the cached inner pipelines would be lost in another process). For backends that need more arguments, pass a wrapper, e.g. \(X, FUN) parallel::mclapply(X, FUN,

Errors are handled as without map, but all keys of a step run before the error of the first failed key is raised. Warnings, messages and random numbers are up to the backend, e.g. future.apply::future_lapply relays warnings and messages and needs future.seed = TRUE, while parallel::mclapply does not relay them.

Runtime control flow via restart and halt:

A running pipeline can be interrupted via the restart() and halt() virtual methods. They are intended for advanced, self-modifying pipelines and are most often called from within a step function via .self (see section 'The pipeline as .self' in pip_add()).

- .self$restart(force = TRUE, times = 1L): aborts the current run after the current step has finished and restarts it from the first step. The default parameters are force = TRUE and times = 1L, that is, the above call is the same as just invoking .self$restart(). To skip steps that are already in state "done", set force = FALSE. The times parameter limits the number of consecutive restarts within a single pip_run() call. If a view is being run, a restart covers the view steps together with their upstream dependencies.
- p$halt(): aborts the current run after the current step has finished. This is a *controlled halt* and deliberately distinct from base::stop(): no error is raised and the pipeline is not marked as "failed". The run simply ends with run state "halted" (see pip_run_state()) and a subsequent pip_run() will continue where the run left off.

A run can also be halted from outside: if the progress callback returns FALSE, the run ends before the next step, e.g. when a user cancels the run in a Shiny app.

In all cases, steps downstream of executed steps that have not been executed until the restart or halt happens are marked as "outdated".

Value

The updated pipeline or view, invisibly.

See Also

vignette("v06-self-modify-pipeline", package = "pipeflow") for an advanced example of dynamic pipelines.

Examples

```
p <- pip_new() |>
pip_add("load", \(n = 3) seq_len(n)) |>
  pip_add("prep", \(x = ~load, weight = 1) x * weight) |>
  pip_add("square", \(x = ~prep) x^2) |>
  pip_add("total", \(x = ~square) sum(x))

pip_run(p)
p

pip_set_params(p, list(weight = 2))
p

# Already-done steps are skipped on a second run
pip_run(p) # first step skipped

# lgr = NULL suppresses log output
pip_run(p, lgr = NULL)

# force = TRUE re-executes every step regardless of state
pip_run(p, force = TRUE)
p

# Run only a subset of steps via a view;
# upstream dependencies are automatically included
v <- pip_view(p, step = "total")
pip_run(v)

# Halt or restart pipeline at runtime (for advanced usage)
p <- pip_new("restart") |>
  pip_add("load", \(n = 3) seq_len(n)) |>
  pip_add("check", \(x = ~load) {
    if (length(x) > 10L) .self$halt()
  }) |>
  pip_add("model", \(x = ~load) {
    if (length(x) == 3L) {
      .self$set_params(list(n = 5))
      .self$restart()
    }
    x * 2
  })

pip_run(p)
p

pip_set_params(p, list(n = 15)) # now halt() in 'check' step is triggered
pip_run(p)

# Continue after errors and pass failures to the steps that allow them
p <- pip_new("continue") |>
  pip_add("data", \(n = 5) seq_len(n)) |>
  pip_add("check", \(x = ~data) stop("check failed")) |>
```

```
  pip_add("count", \(x = ~check) length(x)) |>
  pip_add("total", \(x = ~data) sum(x)) |>
  pip_add("report", \(total = ~total, n = ~try(count)) {
    if (inherits(n, "pipeflow_failure")) conditionMessage(n) else n
  })

withCallingHandlers(
  pip_run(p, lgr = NULL, on_error = "continue"),
  pipeflow_run_failed = \(w) {
    message("failed: ", toString(names(w$failed)))
    invokeRestart("muffleWarning")
  }
)
p
p[["report", "out"]]

# Run the keys of a mapped step with a map function, e.g. in parallel
p <- pip_new("map") |>
  pip_add("parts", \(x = list(a = 1, b = 2, c = 3)) x, exec = "split") |>
  pip_add("square", \(x = ~parts) x^2) |>
  pip_add("total", \(x = ~square) sum(unlist(x)), exec = "reduce")

pip_run(p, map = lapply)
p[["total", "out"]]

if (.Platform$OS.type == "unix") {
  mc2 <- \(X, FUN) parallel::mclapply(X, FUN, mc.cores = 2)
  pip_run(p, force = TRUE, map = mc2)
}
```

`pip_run_state` *Get the run state of a pipeline*

Description

Returns the state of the last (or current) run of a pipeline. Views share the run state of their pipeline.

Usage

```
pip_run_state(x)
```

Arguments

x	A pipeflow pip or view.

Details

The run state is one of

- `"ready"`: the pipeline is new, was reset, or its last run finished without failures.

- `"running"`: a run is in progress (seen from within a step).
- `"restart"`: a step requested a restart, which `pip_run()` carries out right after the step.
- `"halted"`: a step or the `progress` callback halted the run (see section 'Runtime control flow via restart and halt' in `pip_run()`).
- `"failed"`: the last run stopped on an error (`on_error = "stop"`).
- `"continued"`: the last run had failed steps but went on (`on_error = "continue"`).

`pip_reset()` sets the run state back to `"ready"`.

Value

A single character string.

See Also

`pip_run()`, `pip_reset()`

Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 3) seq_len(n)) |>
  pip_add("fail", \(x = ~load) stop("boom"))

pip_run_state(p) # "ready"
try(pip_run(p, lgr = NULL))
pip_run_state(p) # "failed"

suppressWarnings(pip_run(p, on_error = "continue", lgr = NULL))
pip_run_state(p) # "continued"

pip_reset(p)
pip_run_state(p) # "ready"
```

---

pip_set_params *Get or set unbound parameters*

Description

`pip_get_params()` returns the current default values of all unbound parameters of a pipeline or view (i.e., parameters bound to another step via `~step_name` are excluded). `pip_set_params()` updates these parameters in all steps that have them, except for locked steps. Steps whose values change are marked as outdated, together with their downstream dependents. Values that are `identical()` to the stored ones are skipped and don't outdate any step, unless `force = TRUE`.

Usage

```
pip_get_params(x, simplify = TRUE)

pip_set_params(
  x,
  params = list(),
  unknown = c("warn", "ignore", "error"),
  force = FALSE,
  scope = c("pipeline", "view"),
  ...
)
```

Arguments

x	A pipeflow pip or view.
simplify	If TRUE pip_get_params() returns one value per parameter name, otherwise the values of all steps, grouped by name.
params	A named list of parameters to set. One-sided formulas bind parameters to a step, see Binding parameters to steps.
unknown	How to handle undefined params: • "warn" (default) signals a warning and sets the remaining parameters, • "ignore" does so silently, • "error" fails before anything is changed
force	Logical. If TRUE, every step affected by a parameter update is outdated even if the values were identical() to the stored ones.
scope	Defines the scope of steps for the parameter update if x is a view: • "pipeline" (default) sets them in all steps of the pipeline that have the param • "view" only in the steps of the view (except bindings, see Views and scope)
...	Passed on to the methods of `pip_param_value<-`()

Value

For pip_get_params(), a named list of unbound parameter values; if the same parameter name appears in multiple steps, the first occurrence in pipeline order is returned. With simplify = FALSE, a named list with an element per parameter name, which is a list of the values named by step, e.g. list(x = list(s1 = 1, s2 = 2), y = list(s1 = 3)). For pip_set_params(), the updated pipeline or view, invisibly.

Views and scope

On a view, a parameter name is updated if it appears in one of the view's steps, otherwise it is considered unknown (see unknown). With scope = "pipeline", the value is spread to every unbound step of the whole pipeline, so that all names get the same value, as pip_get_params() assumes. With scope = "view", the update is limited to the steps of the view.

Bindings ignore `scope`: a one-sided formula binds a param only in the steps of the view. For example, in `pip_set_params(v, list(alpha = 5, data = ~load))`, `alpha` is set in the whole pipeline while `data` is only bound in the steps of `v`.

Binding parameters to steps

A one-sided formula binds a parameter to the output of a step, e.g. `pip_set_params(p, list(model = ~fit))`, as in `pip_add()`.

- The parameter is bound in *all selected steps*, except for *locked steps*. To bind it only in certain steps, select them with a view.
- The bound steps *must come after* the step that the formula refers to, otherwise an error is signalled and nothing is changed.
- A *relative reference* such as `~-1` is not allowed, but `~try(fit)` works as in `pip_add()`.

A bound parameter is no longer unbound: `pip_get_params()` doesn't return it, and `pip_set_params()` can't change it any more. To unbind or rebind it, replace the step with `pip_replace()`. A one-sided formula therfore can't be set as value. Give it as a string instead and turn it into a formula in the step, e.g. with `stats::as.formula()`.

Locked steps

Parameters of locked steps are never changed and their state remains unchanged. `pip_get_params()` still returns them, so if a locked step comes first among the steps that share a parameter name, `pip_get_params()` keeps returning the locked step's value after the parameter was set. To read the parameters of the unlocked steps only, use a view that excludes the locked ones, `pip_get_params(p[!locked])`.

An environment (or a reference object such as an R6 object) that was modified in place is identical to the stored one, so setting it again doesn't outdate the step unless `force = TRUE`.

Storing annotated values

A new value does not simply replace the stored one. For a step whose param currently holds `x`, the pipeline basically does `pip_param_value(x) <- new_value` and stores the result. For plain values, this is just `new_value`. For an annotated object (e.g. a value with units or labels), a method for its class can instead merge `new_value` into `x`, so that the annotations survive (see `pip_param_value()`). Here, the merging follows these rules:

- *All or nothing*: the merged values of all affected steps are computed before anything is changed. If a method fails, nothing is set and an error of class `pipeflow_set_params_error` reports every failure at once.
- *Comparison*: the merged value is what is compared with the stored one, so setting a plain value that an annotated object already holds doesn't outdate the step.

See Also

`pip_param_value()` for storing and passing on annotated parameter values.

Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 10) seq_len(n)) |>
  pip_add("scale", \(x = ~load, factor = 0.5) x * factor)

# See all unbound/adjustable parameters before running
pip_get_params(p) # list(n = 10, factor = 0.5)
(pip_run(p))

# Updating params marks affected steps (and their dependents) outdated
pip_set_params(p, params = list(n = 5, factor = 2.0))
p
(pip_run(p))

# Setting the same values again doesn't outdate any step
pip_set_params(p, params = list(n = 5, factor = 2.0))
p

# Use force = TRUE to outdate the steps anyway
pip_set_params(p, params = list(n = 5, factor = 2.0), force = TRUE)
p

# Setting a parameter that is not defined in the pipeline yields a warning

pip_set_params(p, params = list(nope = 1))

# ... which can be silenced or turned into an error
pip_set_params(p, params = list(nope = 1), unknown = "ignore")
try(pip_set_params(p, params = list(nope = 1), unknown = "error"))

# Show the values of all steps with simplify = FALSE
d <- suppressWarnings(
  pip_new() |>
    pip_add("s1", \(x = 1, y = 2) x + y) |>
    pip_add("s2", \(x = 3) x)
)
pip_get_params(d, simplify = FALSE)

# pip_get_params() includes locked steps; a view can exclude them
pip_lock(p["load"])
pip_get_params(p) # list(n = 5, factor = 2.0)
pip_get_params(p[!locked]) # list(factor = 2.0)

# On a view, a value is set in the whole pipeline unless scope = "view"
r <- pip_new() |>
  pip_add("fit", \(alpha = 1) alpha) |>
  pip_add("plot", \(x = ~fit, alpha = 1) c(x, alpha), tags = "rep")
pip_set_params(pip_view(r, tags = "rep"), list(alpha = 5))
r[["params"]] # alpha = 5 in both steps
pip_set_params(pip_view(r, tags = "rep"), list(alpha = 2), scope = "view")
r[["params"]] # alpha = 5 in "fit", 2 in "plot"
```

```
# A formula binds a parameter to the output of a step
q <- pip_new() |>
  pip_add("load", \(n = 5) seq_len(n)) |>
  pip_add("fit_mean", \(x = ~load) mean(x)) |>
  pip_add("fit_max", \(x = ~load) max(x)) |>
  pip_add("report", \(model = NA) paste("result:", model)) |>
  pip_add("report_mean", \(model = NA) paste("mean:", model))

# A view selects the steps to bind, here only "report_mean"
pip_set_params(q["report_mean"], params = list(model = ~fit_mean))

# Without a view, the param is bound in all steps that have it unbound
pip_set_params(q, params = list(model = ~fit_max))
q
pip_run(q)
q[["report", "out"]]
q[["report_mean", "out"]]
```

`pip_tag` *Add or remove tags*

Description

Adds tags to, or removes tags from, all steps of a pipeline or a subset of steps defined by a view. Tagged steps can later be selected via `pip_view()`.

Usage

```
pip_tag(x, tags = character())

pip_untag(x, tags = character())
```

Arguments

`x`	A pipeflow pip or view.
`tags`	Character vector of tags to add to or remove from each selected step.

Value

The updated pipeline or view, invisibly.

Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x + 1)

# Tag every step in the pipeline at once
pip_tag(p, c("daily", "core"))
```

```
p
p[, "tags"] # both steps have c("daily", "core")

# Add an extra tag to only one step via a view
p[step == "fit"] |> pip_tag("model")
p
p[, "tags"] # "fit" also has "model"

# Remove "daily" from all steps
pip_untag(p, "daily")
p[, "tags"]
```

`pip_view` *Create a pipeline view*

Description

Creates a filtered view showing only a selected subset of steps. A view references the underlying pipeline without copying it, so operations like `pip_run()` and `pip_set_params()` applied to a view work directly on the underlying pipeline, but are restricted to the steps defined by the view.

Usage

```
pip_view(x, ..., join = c("intersect", "union"), fixed = TRUE)
```

Arguments

`x`	A pipeflow pipeline or view.
`...`	Named filters, which can be one or more of `step`, `params`, `state`, `tags`, `exec`, and `depends`. Each filter value is a character vector of values to keep, or - if `fixed` is `FALSE` - a regular expression. The `params` filter matches against the actual parameter names of each step.
`join`	How individual filters are combined: • `"intersect"` (the default) keeps steps that match *all* filters, • `"union"` keeps steps that match *any* filter. Within a single filter, multiple values are always treated as alternatives (OR).
`fixed`	If TRUE, values in `...` are treated as fixed strings, otherwise they are treated as regular expressions.

Details

A view covers only the selected steps, not their upstream steps, as does `p[i]` (see `[`). So `pip_reset()` or `pip_lock()` on a view leave the upstream steps alone. Running a view is different: `pip_run()` also runs the upstream steps that are not done (see section 'Running views' there).

A view keeps covering its steps when steps are added, removed, renamed or replaced, also through the pipeline or another view. Removed steps drop out of the view, and new steps are not added to it, even if they match the filters.

Value

A view: a `pipeflow` object that shares the pipeline's state and covers the selected steps.

Examples

```
p <- pip_new() |>
  pip_add("load", \(a = 1) a, tags = c("io", "core", "daily")) |>
  pip_add("fit", \(b = 2) b + 1, tags = c("model")) |>
  pip_add("eval_fit", \(fit = ~fit) fit,
    tags = c("model", "daily", "report")
  )
p

# Filter by one or more column values
pip_view(p, state = "new")
pip_view(p, step = c("load", "fit"))

# Filter by tag — keeps steps that have *any* of the given tags
pip_view(p, tags = "daily")

# Combine filters: step pattern AND state (logical AND)
pip_view(p, step = "fit", state = "new")

# Combine filters as a union (step OR state)
pip_view(p, step = "load", tags = "report", join = "union")

# Use a regex pattern
pip_view(p, step = "fit$", fixed = FALSE)

# Filter by parameter names — steps with any of the given parameters
pip_view(p, params = c("a", "fit"))

# Views are composable: create a view-of-view for progressive narrowing
v1 <- pip_view(p, tags = "daily")
print(v1) # load, eval_fit
v2 <- pip_view(v1, tags = "report")
print(v2) # eval_fit only
```

---

print.pipeflow *Print pipeflow objects*

Description

Print pipeflow objects

Usage

```
## S3 method for class 'pipeflow'
str(object, ...)
```

```
## S3 method for class 'pipeflow'
print(
  x,
  rows = integer(),
  cols = getOption("pipeflow.print.cols", default = "core"),
  topn = getOption("pipeflow.print.topn", default = 5),
  nrows = getOption("pipeflow.print.nrows", default = 50),
  row.names = getOption("pipeflow.print.rownames", default = TRUE),
  class = getOption("pipeflow.print.class", default = FALSE),
  header = TRUE,
  ...
)
```

Arguments

object	A pipeflow pipeline or view, for utils::str().
...	Other arguments passed to print.data.table
x	A pipeflow pipeline or view.
rows	Row indices to be printed. If empty, all rows are printed.
cols	The columns to be printed. Can be either one of core or all to print the core or all columns, respectively, or an explicit character vector of columns to be printed. Some columns are shown in short form: • params: the names of the step's parameters. • out: a condition (the output of a step that failed in a run with on_error = "continue") as <kind/class>, e.g. <error/simpleError>; a matrix, array or table as <class[dims]>, e.g. <matrix[4x2]>; a function or environment (e.g. an R6 object) as <class>; strings longer than 40 characters are cut off and marked with "...". • duration: e.g. 0.251 s or 2 min 5 s.
topn	The number of rows to be printed from the beginning and end of tables with more than nrows rows.
nrows	The number of rows printed before truncation is enforced.
row.names	If TRUE, row indices will be printed alongside x.
class	If TRUE, the resulting output will include above each column its storage class (or a self-evident abbreviation thereof).

header	If TRUE, a header with the pipeline name and number of steps, and a footer will be printed. The footer shows the run state, the time and the duration of the last run, and how many of the printed steps are in each state, e.g.
	`<ready> last run: 2026-10-25 14:03:12 (0.251 s)`
	`steps: 3 done, 1 failed, 2 new`

Value

Invisibly returns x.

Examples

```
p <- pip_new("demo") |>
  pip_add("load", \(n = 5) seq_len(n), tags = c("io", "raw")) |>
  pip_add("square", \(x = ~load) x^2, tags = "compute") |>
  pip_add("total", \(x = ~square) sum(x), tags = "compute")

print(p) # core columns: step, params, depends, state, tags
print(p, cols = "all") # all step-table columns
print(p, rows = 2:3) # print only steps 2 and 3

v <- pip_view(p, tags = "compute")
print(v)
```

`rbind.pipeflow` *Bind pipelines*

Description

Binds two or more pipelines together by concatenating their steps. If the pipelines have steps with the same name, the step names of later pipelines are automatically adapted to avoid name clashes. A single pipeline is returned unchanged. A warning is signalled if an unbound param of a later pipeline has another value than the same param in the earlier ones (see Params with the same name in `pip_add()`).

Usage

```
## S3 method for class 'pipeflow'
rbind(..., deparse.level = 1)
```

Arguments

`...`	Two or more pipeflow pipeline objects.
`deparse.level`	Not used, for compatibility with the generic `rbind()`.

Details

To feed steps of the second pipeline with the output of steps of the first one, bind their params with `pip_set_params()` afterwards, e.g. `pip_set_params(ab, list(data = ~prep))`. This binds the param in all steps that have it unbound, so to bind only the steps of the second pipeline, select them with a view, e.g. `ab[-seq_along(a)]`. Locked steps are not bound: unlock them before and lock them again afterwards, e.g. with `pip_unlock()` and `pip_lock()` on the same view.

Value

A new pipeflow pipeline object representing the bound pipelines.

Examples

```
a <- pip_new("a") |>
  pip_add("prep", \(x = 1) x * 2) |>
  pip_add("fit", \(x = ~prep) x + 10)

# "prep" exists in both pipelines; the one from b gets a numeric suffix
b <- pip_new("b") |> pip_add("prep", \(y = 5) y * 3)

ab <- rbind(a, b)
ab[["step"]] # "prep", "fit", "prep2" (step name conflict auto-resolved)
ab

# Any number of pipelines can be combined
abc <- rbind(a, b, b)
abc

# Feed the param `data` of the second pipeline with the output of "prep"
d <- pip_new("d") |> pip_add("scale", \(data = 1, k = 3) data * k)
ad <- rbind(a, d)
pip_set_params(ad[-seq_along(a)], list(data = ~prep))
ad[["scale", "depends"]]
pip_get_params(ad) # `data` is no longer a param
pip_run(ad)
ad[["scale", "out"]]
```

Index

[, *17*, *47*
[.pipeflow *(*Extract.pipeflow*)*, 8
[<-.pipeflow *(*Extract.pipeflow*)*, 8
[[.pipeflow *(*Extract.pipeflow*)*, 8
[[<-.pipeflow *(*Extract.pipeflow*)*, 8
$.pipeflow *(*Extract.pipeflow*)*, 8
$<-.pipeflow *(*Extract.pipeflow*)*, 8

as.function.pipeflow, 2
as.function.pipeflow(), *24*, *39*

base::dim(), *7*
base::length(), *7*
base::nrow(), *7*
base::stop(), *39*

dim.pipeflow, 6

Extract.pipeflow, 8

identical(), *42, 43*

lapply(), *37*
length.pipeflow *(*dim.pipeflow*)*, 6

pip_add, 15
pip_add(), *2*, *4*, *17*, *20*, *26*, *28*, *32*, *34*, *37–39*, *44*, *50*
pip_clone, 19
pip_clone(), *8*, *22*
pip_collect, 20
pip_collect(), *3*
pip_data, 22
pip_data(), *9*
pip_get_params *(*pip_set_params*)*, 42
pip_get_params(), *4*, *29*, *42–44*
pip_graph, 23
pip_inner, 24
pip_inner(), *4*
pip_lock, 25
pip_lock(), *10*, *47*, *50*
pip_new, 26
pip_output_value, 27
pip_output_value(), *30*, *38*
pip_param_value, 29
pip_param_value(), *3*, *28*, *44*
pip_param_value<- *(*pip_param_value*)*, 29
pip_remove, 31
pip_rename, 32
pip_replace, 33
pip_replace(), *10, 11*, *17*, *38*, *44*
pip_reset, 35
pip_reset(), *37, 38*, *42*, *47*
pip_run, 36
pip_run(), *3*, *11*, *14–17*, *23*, *25, 26*, *28*, *30*, *33*, *35*, *37*, *39*, *42*, *47*
pip_run_state, 41
pip_run_state(), *39*
pip_set_params, 42
pip_set_params(), *3, 4*, *25*, *29, 30*, *34*, *37*, *42–44*, *47*, *50*
pip_tag, 46
pip_tag(), *10*, *25*
pip_unlock *(*pip_lock*)*, 25
pip_unlock(), *10*, *25*, *50*
pip_untag *(*pip_tag*)*, 46
pip_untag(), *25*
pip_view, 47
pip_view(), *9*, *21*, *46*
pipeflow_lgr, 14
pipeflow_lgr(), *37*
print.pipeflow, 48

rbind(), *17*
rbind.pipeflow, 50
readRDS(), *27*

saveRDS(), *27*
stats::as.formula(), *44*
str.pipeflow *(*print.pipeflow*)*, 48

utils::str(), *49*

visNetwork::visNetwork(), *23*