

Package ‘pipeflow’

September 27, 2026

Title Fast Interactive Data Analysis Pipelines

Version 0.4.0

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 ggplot2, gridExtra, knitr, mockery, rmarkdown, targets,
testthat, visNetwork

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 2026-09-27 18:50:02 UTC

Contents

dim.pipeflow	2
Extract.pipeflow	3
pip_add	9
pip_clone	11
pip_collect	12
pip_data	14
pip_graph	15
pip_lock	16
pip_new	17
pip_remove	18
pip_rename	19
pip_replace	20
pip_reset	21
pip_run	22
pip_set_params	25
pip_tag	26
pip_view	27
print.pipeflow	28
rbind.pipeflow	29
Index	31

dim.pipeflow	<i>Dimensions of a pipeflow pipeline or view</i>
--------------	--

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 'pipeline'
x[i, j] <- value

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

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

Arguments

<code>x</code>	A pipeline pipeline or view object.
<code>i</code>	The row or field selector. For <code>[</code> and <code>[<-</code> , the rows to select: integer row indices, character step names, or a boolean filter expression evaluated in the context of the step table. For <code>[[</code> , a step-table column name, a meta field, or — in the two-index form — a step name or integer row index. For <code>[[<-</code> , 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.
<code>j</code>	For <code>[</code> , an optional character vector of step-table column names to extract. For <code>[<-</code> , the single step property to assign. For <code>[[</code> and <code>[[<-</code> , the column or step property to extract or assign.
<code>view</code>	For <code>[</code> , 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 <code>j</code> is provided.
<code>value</code>	The value to assign (<code>[[<-</code> , <code>[<-</code>).

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, and the [data.table filter operators](#) re-exported by pipeline (e.g. `%like%`, `%chin%`, `%between%`) are available. `p[]` returns a copy of the pipeline.

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 absolute row indices 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\(\)](#) but tags and execution mode are kept
- `p[[step, "params"]] <- list(...)` – update the step’s parameters; same as `pip_set_params(p, list(...))`
- `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.
- `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, "out"]] <- value` – set the step’s stored output (mostly useful for debugging)

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.

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

`p[i, j] <- value` assigns the step property `j` to the selected steps (see `[[<-`.pipeline above for the supported properties), mirroring the row and column selection of the extraction form `p[i, j]`. `i` selects rows like in the extraction form (including negative row indices, which select all rows but the excluded ones) and `j` must be a single property name; `p[, j] <- value` selects all steps. With more than one selected row, a value of length 1 is replicated to all selected rows and a value whose length equals the number of selected rows 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 selected 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(...)`. With a single selected row, value is stored as-is (like `p[i, j] <- value`). 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))`).

If `value` is a 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` (the rows of both sides are aligned by position and must have the same length). In this form, `i` is required and `j` must be omitted. The properties are copied in the order `step`, `fun`, `params`, `out`, `state`, `time`, `tags`, `locked`, `exec`: the step is renamed to the source name (a no-op when it is unchanged), the function is replaced with the source function (same as `pip_replace()`; its dependencies are recomputed against the steps of `p`), the unbound parameters of the source are applied on top of the new function defaults, and the runtime state and annotation columns are overwritten (locked steps included). The structural columns `nodeId`, `depends` and `unbound` are not copied. 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 pipeline view (if `view = TRUE`) or a new 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 view"
v[["view"]]          # row indices of 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

```

pip_add	<i>Add a step</i>
---------	-------------------

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),
  after = length(x),
  params = list(),
  exec = "auto"
)

```

Arguments

<code>x</code>	A pipeflow pipeline object.
<code>step</code>	Unique step name.
<code>fun</code>	Function to execute for the step. Each function parameter must have a default value. Default values that are simple constants are resolved immediately. Default values that are formulas like <code>~other_step</code> are treated as dependencies to those steps and resolved to the respective output values at runtime once the step is executed.
<code>tags</code>	Optional character vector of tags belonging to the step. Can also be adjusted later using <code>[pip_tag()]</code> .

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.
params	Optional named list of parameter values, which will be merged with the defaults of fun (if overlapping names, the default values in fun take precedence). There are two use cases for params: 1. Provide param values programmatically when adding steps at runtime 2. Provide extra param values to "mark" dependencies that are defined in pipelines nested in a step, which ensures that the step (and with that the pipeline in the step) is re-executed when one of the respective param values change.
exec	Execution mode for this step. One of "auto", "split", "reduce" or "plain". Using execution mode <code>exec = split</code> , the output of the step is marked as partitioned output. In this mode, any step that depends on the split step (directly or indirectly) will have its output automatically mapped partition-wise during step execution. The reduce mode expects partitioned input and passes it through without mapping, while plain mode only accepts non-partitioned input and always intends to execute a single call. In summary: • auto: map if partitioned input appears, otherwise single call • split: single call, then mark output as partitioned • reduce: single call, but only valid with partitioned input • plain: single call, only valid with non-partitioned input

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.

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

# --- Provide parameter values programmatically with 'params' ---
p3 <- pip_new("params-demo") |>
  pip_add("load", \(x = 1, ...) c(x, ...), params = list(size = 42))
pip_run(p3)
p3
p3[["load", "out"]] # c(1, size = 42) - extra params passed via `...`

```

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)
```

Arguments

x	A pipeflow pipeline object.
name	Optional name for the cloned pipeline. If NULL, the original name is used.

Value

A cloned pipeflow pipeline object.

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
```

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)

pip_collect_out(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 <code>data.table</code> 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 always takes one of two shapes:

- **Flat:** a named list whose elements are the step outputs directly, e.g. `list(s1 = 1, s2 = 2)`.
- **Grouped:** a named list whose elements are themselves named lists of step outputs, 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 contains exactly one step, which, for example, is always the case for `by = "step"` as step names are unique. If any group contains more than one step (or `simplify = FALSE`) the result is *grouped*. For list columns such as tags, a step with several entries contributes its output to every corresponding group, and steps without an entry (e.g. untagged steps) are omitted. You can use `pip_view()` to further narrow the selection before collecting.

Value

A named list of outputs

Lifecycle

Deprecated

`pip_collect_out()` is a legacy alias for `pip_collect()`. It raises a deprecation warning and will be removed in a future release.

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	<i>Build pipeline graph data</i>
-----------	----------------------------------

Description

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

Usage

```
pip_graph(x, include_upstream = FALSE)

pip_get_graph(x, include_upstream = FALSE)
```

Arguments

x	A pipeflow pip or view.
include_upstream	Logical. Only relevant for views. If TRUE, add all upstream dependencies of selected steps.

Details

Node shapes reflect execution mode:

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

Value

A named list with two `data.frames`: nodes and edges.

Lifecycle

Deprecated

`pip_get_graph()` is a legacy alias for `pip_graph()`. It raises a deprecation warning and will be removed in a future release.

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")

graph <- pip_graph(p)
graph$nodes # data.frame: id, label, shape, color
```

```

graph$edges # data.frame: from, to, arrows

# 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)
}

```

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.

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.

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
```

pip_remove	<i>Remove a step</i>
------------	----------------------

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, invisibly. For a view, the view with its remapped selector.

Note

If called on a view, the step to be removed must be part of the view, since removing steps drops rows from the pipeline, the row positions of the view will be shifted after the removal operation. For views, you therefore need to assign the result back to the view: `v <- pip_remove(v, ...)`. Views that are not reassigned (or other views on the same pipeline), keep their old row positions and therefore can become invalid after a step removal.

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
v <- pip_remove(v, "transform")
v[["step"]]                # view is remapped and stays valid
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	<i>Rename a step</i>
------------	----------------------

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

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 are automatically marked as outdated and will re-run on the next `pip_run()`.

Usage

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

Arguments

x	A pipeflow pipeline or view object.
step	Unique step name.
fun	Function to execute for the step. Each function parameter must have a default value. Default values that are simple constants are resolved immediately. Default values that are formulas like <code>~other_step</code> are treated as dependencies to those steps and resolved to the respective output values at runtime once the step is executed.
tags	Optional character vector of tags belonging to the step. Can also be adjusted later using <code>[pip_tag()]</code> .
params	Optional named list of parameter values, which will be merged with the defaults of fun (if overlapping names, the default values in fun take precedence). There are two use cases for params: 1. Provide param values programmatically when adding steps at runtime 2. Provide extra param values to "mark" dependencies that are defined in pipelines nested in a step, which ensures that the step (and with that the pipeline in the step) is re-executed when one of the respective param values change.
exec	Execution mode for this step. One of "auto", "split", "reduce" or "plain". Using execution mode <code>exec = split</code> , the output of the step is marked as partitioned output. In this mode, any step that depends on the split step (directly or indirectly) will have its output automatically mapped partition-wise during step execution. The reduce mode expects partitioned input and passes it through without mapping, while plain mode only accepts non-partitioned input and always intends to execute a single call. In summary: • auto: map if partitioned input appears, otherwise single call • split: single call, then mark output as partitioned • reduce: single call, but only valid with partitioned input • plain: single call, only valid with non-partitioned input

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)))
```

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. Any pending restart counters are cleared, but parameters, tags, and locked flags are left unchanged.

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

Locked steps are skipped: their state and output are preserved. If all selected steps are locked, a warning is issued 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
```

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 = pipeflow_lgr, force = FALSE, progress = NULL)
```

Arguments

x	A pipeflow pip or view
lgr	A logging function of the form function(level, msg, ...). To suppress logging, you can set lgr = NULL.
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.

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()`)
 - a step it depends on was re-executed or replaced
 - the last run did not reach it either on purpose (see section 'Running views below) or because a run was aborted early due to failed step
- done: the step was executed successfully with its current inputs
- failed: the step raised an error during its last execution

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()`.

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 that were not processed are marked "outdated".

Runtime errors:

If a step fails with an error, the failing step's state is set to "failed", the run is aborted (no further steps are executed), and the pipeline run state is set to "failed". Steps that have not been executed are marked "outdated", so a subsequent run retries 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 the `.self` argument.

- `.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, the steps that have not been executed are marked "outdated", and a subsequent `pip_run()` will continue where the run left off.

In both cases 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)
```

pip_set_params	<i>Get or set unbound parameters</i>
----------------	--------------------------------------

Description

`pip_get_params()` returns the current default values of all unbound parameters of a pipeline or view, that is, parameters wired to another step's output via `~step_name` are excluded. `pip_set_params()` updates these parameters for the whole pipeline or or view and marks the affected steps and their downstream dependents as outdated.

Usage

```
pip_get_params(x)

pip_set_params(x, params = list())
```

Arguments

x	A pipeflow pip or view.
params	Named list of parameters to set (only used by <code>pip_set_params()</code>).

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. For `pip_set_params()`, the updated pipeline or view, invisibly.

Note

Parameters of locked steps are never changed and their state remains unchanged.

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

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

pip_tag	<i>Add or remove tags</i>
---------	---------------------------

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	<i>Create a pipeline view</i>
----------	-------------------------------

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 <i>all</i> filters, • "union" keeps steps that match <i>any</i> 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.

Value

A pipeflow_view object.

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	<i>Print pipeflow objects</i>
----------------	-------------------------------

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.row.names", default = TRUE),
  class = getOption("pipeflow.print.class", default = FALSE),
  header = TRUE,
  ...
)

```

Arguments

object	A pipeflow pipeline or view, for <code>utils::str()</code> .
...	Other arguments passed to <code>print.data.table</code>

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 <code>core</code> or <code>all</code> to print the core or all columns, respectively, or an explicit character vector of columns to be printed. The <code>params</code> column lists the names of the step's parameters.
topn	The number of rows to be printed from the beginning and end of tables with more than <code>nrows</code> rows.
nrows	The number of rows printed before truncation is enforced.
row.names	If <code>TRUE</code> , row indices will be printed alongside <code>x</code> .
class	If <code>TRUE</code> , the resulting output will include above each column its storage class (or a self-evident abbreviation thereof).
header	If <code>TRUE</code> , a header with the pipeline name and number of steps, and a footer with the run state and the time of the last run, will be printed.

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.

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()`.

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", \(x = 5) x * 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
```

Index

`[.pipeflow (Extract.pipeflow), 3`
`[<-.pipeflow (Extract.pipeflow), 3`
`[[.pipeflow (Extract.pipeflow), 3`
`[[<-.pipeflow (Extract.pipeflow), 3`
`$.pipeflow (Extract.pipeflow), 3`
`$<-.pipeflow (Extract.pipeflow), 3`

`base::dim(), 3`
`base::length(), 3`
`base::nrow(), 3`
`base::stop(), 23`

`data.table` filter operators, 4
`dim.pipeflow, 2`

`Extract.pipeflow, 3`

`length.pipeflow (dim.pipeflow), 2`

`pip_add, 9`
`pip_add(), 17`
`pip_clone, 11`
`pip_clone(), 14`
`pip_collect, 12`
`pip_collect(), 13`
`pip_collect_out (pip_collect), 12`
`pip_data, 14`
`pip_data(), 5`
`pip_get_graph (pip_graph), 15`
`pip_get_params (pip_set_params), 25`
`pip_get_params(), 25`
`pip_graph, 15`
`pip_graph(), 15`
`pip_lock, 16`
`pip_lock(), 5`
`pip_new, 17`
`pip_remove, 18`
`pip_rename, 19`
`pip_replace, 20`
`pip_replace(), 5, 6`
`pip_reset, 21`

`pip_reset(), 23`
`pip_run, 22`
`pip_run(), 16, 17, 20, 21, 23, 27`
`pip_set_params, 25`
`pip_set_params(), 16, 23, 25, 27`
`pip_tag, 26`
`pip_tag(), 5, 16`
`pip_unlock (pip_lock), 16`
`pip_unlock(), 5, 16`
`pip_untag (pip_tag), 26`
`pip_untag(), 16`
`pip_view, 27`
`pip_view(), 4, 13, 26`
`print.pipeflow, 28`

`rbind.pipeflow, 29`

`str.pipeflow (print.pipeflow), 28`

`utils::str(), 28`

`visNetwork::visNetwork(), 15`