Package 'pipeflow'

October 13, 2026

Title Fast Interactive Data Analysis Pipelines

Version 0.5.0

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, rlang, 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 2026-10-13 14:50:07 UTC

Contents

dim.pipeflow . . . 2
Extract.pipeflow . . . 3
pipeflow_lgr . . . 9
pip_add . . . 10
pip_clone . . . 13
pip_collect . . . 14
pip_data . . . 15
pip_graph . . . 16
pip_lock . . . 17
pip_new . . . 18
pip_param_value . . . 19
pip_remove . . . 20
pip_rename . . . 21
pip_replace . . . 22
pip_reset . . . 24
pip_run . . . 25
pip_run_state . . . 29
pip_set_params . . . 30
pip_tag . . . 32
pip_view . . . 33
print.pipeflow . . . 34
rbind.pipeflow . . . 36

Index **37**

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

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(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 `[[<-.pipeflow` 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 `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` (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 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.

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

Arguments

x	A pipeflow pipeline 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 `~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 is run with `on_error = "continue"`: instead of the output of the referenced step, the argument then receives a condition of class `pipeflow_failure` (see section 'Continuing after errors' in `pip_run()`). The function can refer to the running pipeline as `.self` (see section 'The pipeline as `.self`' in `pip_add()`).
tags	Optional character vector of tags belonging to the step. Can also be adjusted later using `[pip_tag()]`.
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 `exec = split`, 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. If a split pipeline fails in a run with `on_error = "continue"`, a `reduce` step also accepts the resulting failure object when it takes it via `~try()`. 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 or, via `~try()`, a failure object • 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.

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. `.self` is a reserved name: it can't be an argument of `fun` or a name in `params`.

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 `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 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 *Build pipeline graph data*

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.

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_param_value *Unwrap a parameter value before a step function is called*

Description

`pip_param_value()` is an extension point for packages that store annotated values (e.g. values with units or labels, or parameter objects that carry UI metadata) as step parameters. When a step runs, the generic is applied once 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.

Usage

```
pip_param_value(x, ...)
```

Arguments

`x`	The parameter value as stored in the step.
`...`	Currently unused, reserved for future extensions. Methods should accept it.

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")`.

Value

The value the step function receives for the argument.

See Also

`pip_set_params()`, `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)
```

---

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

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)
```

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 `~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 is run with `on_error = "continue"`: instead of the output of the referenced step, the argument then receives a condition of class `pipeflow_failure` (see section 'Continuing after errors' in `pip_run()`). The function can refer to the running pipeline as `.self` (see section 'The pipeline as `.self`' in `pip_add()`).
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, 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 the step, see `pip_add()`. If `NULL` (the default), the current execution mode of the step is kept.

Details

By default, the step keeps its tags and execution mode: `tags = NULL` and `exec = NULL` mean "keep the current value". To change them, pass explicit values, where `tags = character(0)` removes all tags and `exec = "auto"` resets the execution mode to its default. Parameters, output, state and lock status are always reset, like for a freshly added step.

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"]
```

`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,
  on_error = c("stop", "continue")
)
```

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 `pipeflow_lgr()`, which can also be used to build custom loggers. 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.
`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).

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
  - 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()`.

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:

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 downstream of executed steps that were not reached are marked `"outdated"`, so a subsequent run retries them.

The error is signalled as a condition of class `c("pipeflow_step_error", "error", "condition")`. Its message is the message of the original error, prefixed with `"step '<step>': "`. Its field `step` names the failed step and its field `parent` holds the original error condition with its class, which can be caught via `tryCatch(pip_run(p), pipeflow_step_error = function(e) e$parent)`. If the step was mapped over partitioned inputs (see `exec` in `pip_add()`), `parent` is a condition of class `c("pipeflow_key_error", "error", "condition")` instead. Its message is prefixed with `"key '<key>': "`, its field `key` names the failed partition key and its field `parent` holds the original error condition.

Continuing after errors:

With `on_error = "continue"`, a failing step gets state `"failed"`, its error condition is stored as its output, and the run goes on. For a step mapped over partitioned inputs, the stored condition is the `pipeflow_key_error` described above. An input of a step is *failed* if its step failed or was not run because a step upstream of it failed in this run.

- A step with a failed input is not run, and counts as failed for its own dependents, unless the argument receiving that input refers to its step via `try()`, e.g. `x = ~try(other_step)` (see `pip_add()`).
- A step whose failed inputs all use `try()` references is run, and each of these arguments receives a condition of class `c("pipeflow_failure", "error", "condition")` with the fields `message`, `step` (the step of the input), `failed_step` (the step that actually failed) and `parent` (the original error condition).
- Locked steps are the exception: they are never run, so a failed input does not affect them. They keep their state and output, and their dependents receive that output as usual.

Steps that ran (including those that received failure objects) are `"done"`. Steps that were not run are treated like steps not reached by an aborted run: a `"done"` step becomes `"outdated"`, a `"new"` one stays `"new"`, so the next run retries them together with the failed steps. Once a failed step succeeds, the steps downstream of it, including those that received failure objects, are run again.

If any step failed, the pipeline run state is set to `"continued"` and, once all states and outputs are stored, a warning of class `c("pipeflow_run_failed", "warning", "condition")` is signalled. Its field `failed` is a named list with the original conditions of the failed steps and its field `skipped` names the steps that were not run.

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.

In both 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"]]
```

`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 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 (parameters wired to another step's output via ~step_name are excluded), including those of locked steps. pip_set_params() updates these parameters for the whole pipeline or view. Steps whose values change are marked as outdated, together with their downstream dependents (steps that have never run stay "new"). Values that are identical() to the stored ones are skipped and don't outdate any step, unless force = TRUE.

Usage

```
pip_get_params(x)

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

Arguments

x	A pipeflow pip or view.
params	Named list of parameters to set

unknown — How to handle `params` that are not defined in any of the selected steps, including locked ones: `"warn"` (default) signals a warning and sets the remaining parameters, `"ignore"` does so silently, and `"error"` fails before anything is changed.

force — Logical. If `TRUE`, every step that has one of the `params` is outdated (together with its downstream steps), even if the values are `identical()` to the stored ones. This refreshes steps whose inputs are environments or reference objects that were modified in place.

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. `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.

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`.

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"))
```

`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 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. The params column lists the names of the step's parameters. In the out column, some outputs are shown in short form: 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]>; and a function or environment (e.g. an R6 object) as <class>. Strings longer than 40 characters are cut off and marked with "...".
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 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(), *27*

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

Extract.pipeflow, 3

identical(), *30, 31*

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

pip_add, 10
pip_add(), *11*, *18*, *23*, *26, 27*
pip_clone, 13
pip_clone(), *16*
pip_collect, 14
pip_collect(), *14*
pip_collect_out *(pip_collect)*, 14
pip_data, 15
pip_data(), *5*
pip_get_graph *(pip_graph)*, 16
pip_get_params *(pip_set_params)*, 30
pip_get_params(), *20*, *30, 31*
pip_graph, 16
pip_graph(), *17*
pip_lock, 17
pip_lock(), *6*
pip_new, 18
pip_param_value, 19
pip_remove, 20
pip_rename, 21
pip_replace, 22
pip_replace(), *5, 6*
pip_reset, 24
pip_reset(), *26*, *29, 30*
pip_run, 25
pip_run(), *9–12*, *17, 18*, *20*, *22–24*, *26, 27*, *29, 30*, *33*
pip_run_state, 29
pip_run_state(), *27*
pip_set_params, 30
pip_set_params(), *17*, *20*, *26*, *30, 31*, *33*
pip_tag, 32
pip_tag(), *5*, *17*
pip_unlock *(pip_lock)*, 17
pip_unlock(), *6*, *17*
pip_untag *(pip_tag)*, 32
pip_untag(), *17*
pip_view, 33
pip_view(), *4, 5*, *14*, *32*
pipeflow_lgr, 9
pipeflow_lgr(), *25*
print.pipeflow, 34

rbind.pipeflow, 36
readRDS(), *19*

saveRDS(), *19*
str.pipeflow *(print.pipeflow)*, 34

utils::str(), *35*

visNetwork::visNetwork(), *16*