Package ‘targets’

January 31, 2025

Title Dynamic Function-Oriented 'Make'-Like Declarative Pipelines

Description Pipeline tools coordinate the pieces of computationally
demanding analysis projects.
The 'targets' package is a 'Make'-like pipeline tool for statistics and
data science in R. The package skips costly runtime for tasks
that are already up to date,
orchestrates the necessary computation with implicit parallel computing,
and abstracts files as R objects. If all the current output matches
the current upstream code and data, then the whole pipeline is up
to date, and the results are more trustworthy than otherwise.
The methodology in this package
borrows from GNU 'Make' (2015, ISBN:978-9881443519)
and 'drake' (2018, <doi:10.21105/joss.00550>).

Version 1.10.1

License MIT + file LICENSE

URL `https://docs.ropensci.org/targets/`,
`https://github.com/ropensci/targets`

BugReports `https://github.com/ropensci/targets/issues`

Depends R (>= 3.5.0)

Imports base64url (>= 1.4), callr (>= 3.7.0), cli (>= 2.0.2),
codetools (>= 0.2.16), data.table (>= 1.12.8), igraph (>=
2.0.0), knitr (>= 1.34), ps (>= 1.8.0), R6 (>= 2.4.1), rlang
(>= 1.0.0), secretbase (>= 0.5.0), stats, tibble (>= 3.0.1),
tidyselect (>= 1.1.0), tools, utils, vctrs (>= 0.2.4), yaml (>=
2.2.1)

Suggests autometric (>= 0.1.0), bslib, clustermq (>= 0.9.2), crew (>=
0.9.0), curl (>= 4.3), DT (>= 0.14), dplyr (>= 1.0.0), fst (>=
0.9.2), future (>= 1.19.1), future.batchtools (>= 0.9.0),
future.callr (>= 0.6.0), gargle (>= 1.2.0), googleCloudStorageR
(>= 0.7.0), gt (>= 0.2.2), keras (>= 2.2.5.0), markdown (>=
1.1), nanonext (>= 0.12.0), rmarkdown (>= 2.4), parallelly (>=
1.35.0), paws.common (>= 0.6.4), paws.storage (>= 0.4.0),
pkgload (>= 1.1.0), processx (>= 3.4.3), qs2, reprex (>=

2.0.0), rstudioapi (>= 0.11), R.utils (>= 2.6.0), shiny (>= 1.5.0), shinybusy (>= 0.2.2), shinyWidgets (>= 0.5.4), tarchetypes, testthat (>= 3.0.0), torch (>= 0.1.0), usethis (>= 1.6.3), visNetwork (>= 2.1.2)

Encoding UTF-8

Language en-US

VignetteBuilder knitr

Config/testthat/edition 3

RoxygenNote 7.3.2

NeedsCompilation no

Author William Michael Landau [aut, cre] (<https://orcid.org/0000-0003-1878-3253>),
Matthew T. Warkentin [ctb],
Mark Edmondson [ctb] (<https://orcid.org/0000-0002-8434-3881>),
Samantha Oliver [rev] (<https://orcid.org/0000-0001-5668-1165>),
Tristan Mahr [rev] (<https://orcid.org/0000-0002-8890-5116>),
Eli Lilly and Company [cph, fnd]

Maintainer William Michael Landau <will.landau.oss@gmail.com>

Repository CRAN

Date/Publication 2025-01-31 16:50:02 UTC

Contents

targets-package 5
tar_active 5
tar_assert 6
tar_backoff 9
tar_branches 10
tar_branch_index 11
tar_branch_names 12
tar_call 13
tar_cancel 14
tar_canceled 15
tar_completed 16
tar_condition 17
tar_config_get 18
tar_config_projects 20
tar_config_set 21
tar_config_unset 26
tar_config_yaml 28
tar_crew 29
tar_cue 30
tar_definition 32
tar_delete 34
tar_deps 36

tar_described_as 37
tar_destroy 39
tar_dispatched 42
tar_edit 43
tar_engine_knitr 43
tar_envir 45
tar_envvars 46
tar_errored 48
tar_exist_meta 49
tar_exist_objects 50
tar_exist_process 51
tar_exist_progress 51
tar_exist_script 52
tar_format 53
tar_format_get 56
tar_github_actions 57
tar_glimpse 58
tar_group 61
tar_helper 63
tar_interactive 64
tar_invalidate 65
tar_language 66
tar_load 67
tar_load_everything 70
tar_load_globals 71
tar_make 73
tar_make_clustermq 77
tar_make_future 80
tar_manifest 84
tar_mermaid 86
tar_meta 90
tar_meta_delete 93
tar_meta_download 94
tar_meta_sync 96
tar_meta_upload 97
tar_name 99
tar_network 100
tar_newer 103
tar_noninteractive 104
tar_objects 105
tar_older 106
tar_option_get 108
tar_option_reset 109
tar_option_set 110
tar_outdated 118
tar_path_script 121
tar_path_script_support 122
tar_path_store 123

tar_path_target 123
tar_pattern 125
tar_pid 127
tar_poll 128
tar_process 129
tar_progress 130
tar_progress_branches 132
tar_progress_summary 133
tar_prune 135
tar_prune_list 137
tar_read 139
tar_renv 141
tar_repository_cas 143
tar_repository_cas_local 148
tar_repository_cas_local_gc 150
tar_reprex 153
tar_resources 154
tar_resources_aws 156
tar_resources_clustermq 159
tar_resources_crew 161
tar_resources_custom_format 162
tar_resources_feather 164
tar_resources_fst 165
tar_resources_future 166
tar_resources_gcp 168
tar_resources_network 169
tar_resources_parquet 171
tar_resources_qs 172
tar_resources_repository_cas 174
tar_resources_url 175
tar_script 177
tar_seed_create 178
tar_seed_get 180
tar_seed_set 181
tar_sitrep 183
tar_skipped 186
tar_source 187
tar_target 189
tar_test 196
tar_timestamp 197
tar_toggle 199
tar_traceback 200
tar_unblock_process 201
tar_unscript 202
tar_unversion 203
tar_validate 204
tar_visnetwork 205
tar_watch 209

tar_watch_server 212
tar_watch_ui 214
tar_workspace 215
tar_workspaces 217
use_targets 218
use_targets_rmd 219

Index **221**

`targets-package` *targets: Dynamic Function-Oriented Make-Like Declarative Pipelines for R*

Description

A pipeline toolkit for Statistics and data science in R, the `targets` package brings function-oriented programming to Make-like declarative pipelines. `targets` orchestrates a pipeline as a graph of dependencies, skips steps that are already up to date, runs the necessary computations with optional parallel workers, abstracts files as R objects, and provides tangible evidence that the results are reproducible given the underlying code and data. The methodology in this package borrows from GNU Make (2015, ISBN:978-9881443519) and `drake` (2018, doi:10.21105/joss.00550).

See Also

Other help: `tar_reprex()`, `use_targets()`, `use_targets_rmd()`

`tar_active` *Show if the pipeline is running.*

Description

Return `TRUE` if called in a target or `_targets.R` and the pipeline is running.

Usage

```
tar_active()
```

Value

Logical of length 1, `TRUE` if called in a target or `_targets.R` and the pipeline is running (`FALSE` otherwise).

See Also

Other utilities: `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_active() # FALSE
tar_script({
  library(targets)
  library(tarchetypes)
  message("Pipeline running? ", tar_active())
  tar_target(x, tar_active())
})
tar_manifest() # prints "Pipeline running? FALSE"
tar_make() # prints "pipeline running? TRUE"
tar_read(x) # TRUE
})
}
```

`tar_assert` *Assertions*

Description

These functions assert the correctness of user inputs and generate custom error conditions as needed. Useful for writing packages built on top of `targets`.

Usage

```
tar_assert_chr(x, msg = NULL)

tar_assert_dbl(x, msg = NULL)

tar_assert_df(x, msg = NULL)

tar_assert_equal_lengths(x, msg = NULL)

tar_assert_envir(x, msg = NULL)

tar_assert_expr(x, msg = NULL)

tar_assert_flag(x, choices, msg = NULL)

tar_assert_file(x)

tar_assert_finite(x, msg = NULL)

tar_assert_function(x, msg = NULL)

tar_assert_function_arguments(x, args, msg = NULL)
```

```
tar_assert_ge(x, threshold, msg = NULL)

tar_assert_identical(x, y, msg = NULL)

tar_assert_in(x, choices, msg = NULL)

tar_assert_not_dirs(x, msg = NULL)

tar_assert_not_dir(x, msg = NULL)

tar_assert_not_in(x, choices, msg = NULL)

tar_assert_inherits(x, class, msg = NULL)

tar_assert_int(x, msg = NULL)

tar_assert_internet(msg = NULL)

tar_assert_lang(x, msg = NULL)

tar_assert_le(x, threshold, msg = NULL)

tar_assert_list(x, msg = NULL)

tar_assert_lgl(x, msg = NULL)

tar_assert_name(x)

tar_assert_named(x, msg = NULL)

tar_assert_names(x, msg = NULL)

tar_assert_nonempty(x, msg = NULL)

tar_assert_null(x, msg = NULL)

tar_assert_not_expr(x, msg = NULL)

tar_assert_nzchar(x, msg = NULL)

tar_assert_package(package, msg = NULL)

tar_assert_path(path, msg = NULL)

tar_assert_match(x, pattern, msg = NULL)

tar_assert_nonmissing(x, msg = NULL)
```

```
tar_assert_positive(x, msg = NULL)

tar_assert_scalar(x, msg = NULL)

tar_assert_store(store)

tar_assert_target(x, msg = NULL)

tar_assert_target_list(x)

tar_assert_true(x, msg = NULL)

tar_assert_unique(x, msg = NULL)

tar_assert_unique_targets(x)
```

Arguments

`x`	R object, input to be validated. The kind of object depends on the specific assertion function called.
`msg`	Character of length 1, a message to be printed to the console if `x` is invalid.
`choices`	Character vector of choices of `x` for certain assertions.
`args`	Character vector of expected function argument names. Order matters.
`threshold`	Numeric of length 1, lower/upper bound for assertions like `tar_assert_le()`/`tar_assert_ge()`.
`y`	R object, value to compare against `x`.
`class`	Character vector of expected class names.
`package`	Character of length 1, name of an R package.
`path`	Character, file path.
`pattern`	Character of length 1, a `grep` pattern for certain assertions.
`store`	Character of length 1, path to the data store of the pipeline.

See Also

Other utilities to extend targets: `tar_condition`, `tar_language`, `tar_test()`

Examples

```
tar_assert_chr("123")
try(tar_assert_chr(123))
```

`tar_backoff` *Superseded: exponential backoff*

Description

Superseded: configure exponential backoff while polling for tasks during the pipeline.

Usage

```
tar_backoff(min = 0.001, max = 0.1, rate = 1.5)
```

Arguments

`min`	Positive numeric of length 1, minimum polling interval in seconds. Must be at least `sqrt(.Machine$double.eps)`.
`max`	Positive numeric of length 1, maximum polling interval in seconds. Must be at least `sqrt(.Machine$double.eps)`.
`rate`	Positive numeric of length 1, greater than or equal to 1. Multiplicative rate parameter that allows the exponential backoff minimum polling interval to increase from `min` to `max`. Actual polling intervals are sampled uniformly from the current minimum to `max`.

Details

This function is superseded and is now only relevant to other superseded functions `tar_make_clustermq()` and `tar_make_future()`. `tar_make()` uses `crew` in an efficient non-polling way, making exponential backoff unnecessary.

Backoff

In high-performance computing it can be expensive to repeatedly poll the priority queue if no targets are ready to process. The number of seconds between polls is `runif(1, min, max(max, min * rate ^ index))`, where `index` is the number of consecutive polls so far that found no targets ready to skip or run, and `min`, `max`, and `rate` are arguments to `tar_backoff()`. (If no target is ready, `index` goes up by 1. If a target is ready, `index` resets to 0. For more information on exponential, backoff, visit `https://en.wikipedia.org/wiki/Exponential_backoff`). Raising `min` or `max` is kinder to the CPU etc. but may incur delays in some instances.

See Also

Other utilities: `tar_active()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_option_set(backoff = tar_backoff(min = 0.001, max = 0.1, rate = 1.5))
})
}
```

`tar_branches` *Reconstruct the branch names and the names of their dependencies.*

Description

Given a branching pattern, use available metadata to reconstruct branch names and the names of each branch's dependencies. The metadata of each target must already exist and be consistent with the metadata of the other targets involved.

Usage

```
tar_branches(
  name,
  pattern = NULL,
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`name`	Symbol, name of the target.
`pattern`	Language to define branching for a target (just like in `tar_target()`) or `NULL` to get the pattern from the `targets` pipeline script specified in the `script` argument (default: `_targets.R`).
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

The results from this function can help you retroactively figure out correspondences between upstream branches and downstream branches. However, it does not always correctly predict what the names of the branches will be after the next run of the pipeline. Dynamic branching happens while the pipeline is running, so we cannot always know what the names of the branches will be in advance (or even how many there will be).

Value

A `tibble` with one row per branch and one column for each target (including the branched-over targets and the target with the pattern.)

See Also

Other branching: `tar_branch_index()`, `tar_branch_names()`, `tar_pattern()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, head(letters, 2)),
    tar_target(z, head(LETTERS, 2)),
    tar_target(dynamic, c(x, y, z), pattern = cross(z, map(x, y)))
  )
}, ask = FALSE)
tar_make()
tar_branches(dynamic)
tar_branches(dynamic, pattern = cross(z, map(x, y)))
})
}
```

`tar_branch_index` *Integer branch indexes*

Description

Get the integer indexes of individual branch names within their corresponding dynamic branching targets.

Usage

```
tar_branch_index(names, store = targets::tar_config_get("store"))
```

Arguments

`names`	Character vector of branch names.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A named integer vector of branch indexes.

See Also

Other branching: `tar_branch_names()`, `tar_branches()`, `tar_pattern()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(4)),
    tar_target(y, 2 * x, pattern = map(x)),
    tar_target(z, y, pattern = map(y))
  )
}, ask = FALSE)
tar_make()
names <- c(
  tar_meta(y, children)$children[[1]][c(2, 3)],
  tar_meta(z, children)$children[[1]][2]
)
names
tar_branch_index(names) # c(2, 3, 2)
})
}
```

`tar_branch_names` *Branch names*

Description

Get the branch names of a dynamic branching target using numeric indexes. `tar_branch_names()` expects an unevaluated symbol for the name argument, whereas `tar_branch_names_raw()` expects a character string for `name`.

Usage

```
tar_branch_names(name, index, store = targets::tar_config_get("store"))

tar_branch_names_raw(name, index, store = targets::tar_config_get("store"))
```

Arguments

`name`	Name of the dynamic branching target. `tar_branch_names()` expects an unevaluated symbol for the name argument, whereas `tar_branch_names_raw()` expects a character string for `name`.
`index`	Integer vector of branch indexes.
`store`	Character string, directory path to the `targets` data store of the pipeline.

Value

A character vector of branch names.

See Also

Other branching: `tar_branch_index()`, `tar_branches()`, `tar_pattern()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(4)),
    tar_target(y, 2 * x, pattern = map(x)),
    tar_target(z, y, pattern = map(y))
  )
}, ask = FALSE)
tar_make()
tar_branch_names(z, c(2, 3))
})
}
```

`tar_call` *Identify the called* `targets` *function.*

Description

Get the name of the currently running `targets` interface function. Returns `NULL` if not invoked inside a target or `_targets.R` (i.e. if not directly invoked by `tar_make()`, `tar_visnetwork()`, etc.).

Usage

```
tar_call()
```

Value

Character of length 1, name of the currently running `targets` interface function. For example, suppose you have a call to `tar_call()` inside a target or `_targets.R`. Then if you run `tar_make()`, `tar_call()` will return `"tar_make"`.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_call() # NULL
tar_script({
  library(targets)
  library(tarchetypes)
  message("called function: ", tar_call())
  tar_target(x, tar_call())
})
tar_manifest() # prints "called function: tar_manifest"
tar_make() # prints "called function: tar_make"
tar_read(x) # "tar_make"
})
}
```

`tar_cancel` *Cancel a target mid-execution under a custom condition.*

Description

Cancel a target while its command is running if a condition is met.

Usage

```
tar_cancel(condition = TRUE)
```

Arguments

`condition`	Logical of length 1, whether to cancel the target.

Details

Must be invoked by the target itself. `tar_cancel()` cannot interrupt a target from another process.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(tar_target(x, tar_cancel(1 > 0)))
tar_make() # Should cancel target x.
})
}
```

`tar_canceled` *List canceled targets.*

Description

List targets whose progress is `"canceled"`.

Usage

```
tar_canceled(names = NULL, store = targets::tar_config_get("store"))
```

Arguments

`names`	Optional, names of the targets. If supplied, the output is restricted to the selected targets. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A character vector of canceled targets.

See Also

Other progress: `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_poll()`, `tar_progress()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_skipped()`, `tar_watch()`, `tar_watch_server()`, `tar_watch_ui()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_canceled()
tar_canceled(starts_with("y_")) # see also any_of()
})
}
```

`tar_completed` *List completed targets.*

Description

List targets whose progress is `"completed"`.

Usage

```
tar_completed(names = NULL, store = targets::tar_config_get("store"))
```

Arguments

`names`	Optional, names of the targets. If supplied, the output is restricted to the selected targets. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A character vector of completed targets.

See Also

Other progress: tar_canceled(), tar_dispatched(), tar_errored(), tar_poll(), tar_progress(), tar_progress_branches(), tar_progress_summary(), tar_skipped(), tar_watch(), tar_watch_server(), tar_watch_ui()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_completed()
tar_completed(starts_with("y_")) # see also any_of()
})
}
```

`tar_condition` *Conditions*

Description

These functions throw custom `targets`-specific error conditions. Useful for error handling in packages built on top of `targets`.

Usage

```
tar_message_run(...)

tar_throw_file(...)

tar_throw_run(..., class = character(0))

tar_throw_validate(...)

tar_warn_deprecate(...)

tar_warn_run(...)

tar_warn_validate(...)

tar_message_validate(...)
```

```
tar_print(...)

tar_error(message, class)

tar_warning(message, class)

tar_message(message, class)
```

Arguments

`...`	zero or more objects which can be coerced to character (and which are pasted together with no separator) or a single condition object.
`class`	Character vector of S3 classes of the message.
`message`	Character of length 1, text of the message.

See Also

Other utilities to extend targets: `tar_assert`, `tar_language`, `tar_test()`

Examples

```
try(tar_throw_validate("something is not valid"))
```

`tar_config_get` *Get configuration settings.*

Description

Read the custom settings for the current project in the optional YAML configuration file.

Usage

```
tar_config_get(
  name,
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```

Arguments

`name`	Character of length 1, name of the specific configuration setting to retrieve.
`config`	Character of length 1, file path of the YAML configuration file with `targets` project settings. The `config` argument specifies which YAML configuration file that `tar_config_get()` reads from or `tar_config_set()` writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always

	`_targets.yaml` unless you set another default path using the `TAR_CONFIG` environment variable, e.g. `Sys.setenv(TAR_CONFIG = "custom.yaml")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.
`project`	Character of length 1, name of the current `targets` project. Thanks to the `config` R package, `targets` YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The `project` argument allows you to set or get a configuration setting for a specific project for a given call to `tar_config_set()` or `tar_config_get()`. The default project is always called `"main"` unless you set another default project using the `TAR_PROJECT` environment variable, e.g. `Sys.setenv(tar_project = "custom")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.

Value

The value of the configuration setting from the YAML configuration file (default: `_targets.yaml`) or the default value if the setting is not available. The data type of the return value depends on your choice of `name`.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

Configuration

For several key functions like `tar_make()`, the default values of arguments are controlled though `tar_config_get()`. `tar_config_get()` retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with `tar_config_set()`. The default file path of this YAML file is `_targets.yaml`, and you can set another path globally using the `TAR_CONFIG` environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the `TAR_PROJECT` environment variable. The structure of the YAML file follows rules similar to the `config` R package, e.g. projects can inherit settings from one another using the `inherits` field. Exceptions include:

1. There is no requirement to have a configuration named `"default"`.
2. Other projects do not inherit from the default project‘ automatically.
3. Not all fields need values because `targets` already has defaults.

`targets` does not actually invoke the `config` package. The implementation in `targets` was written from scratch without viewing or copying any part of the source code of `config`.

See Also

Other configuration: tar_config_projects(), tar_config_set(), tar_config_unset(), tar_config_yaml(), tar_envvars(), tar_option_get(), tar_option_reset(), tar_option_set()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(list(tar_target(x, 1 + 1)))
tar_config_get("store") # "_targets"
store_path <- tempfile()
tar_config_set(store = store_path)
tar_config_get("store") # Shows a temp file.
tar_make() # Writes to the custom data store identified in _targets.yaml.
tar_read(x) # tar_read() knows about _targets.yaml too.
file.exists("_targets") # FALSE
file.exists(store_path) # TRUE
})
}
```

tar_config_projects *List projects.*

Description

List the names of projects defined in `_targets.yaml`.

Usage

```
tar_config_projects(config = Sys.getenv("TAR_CONFIG", "_targets.yaml"))
```

Arguments

config	Character of length 1, file path of the YAML configuration file with targets project settings. The config argument specifies which YAML configuration file that tar_config_get() reads from or tar_config_set() writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always _targets.yaml unless you set another default path using the TAR_CONFIG environment variable, e.g. Sys.setenv(TAR_CONFIG = "custom.yaml"). This also has the effect of temporarily modifying the default arguments to other functions such as tar_make() because the default arguments to those functions are controlled by tar_config_get().

Value

Character vector of names of projects defined in `_targets.yaml`.

Storage access

Several functions like tar_make(), tar_read(), tar_load(), tar_meta(), and tar_progress() read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the tarchetypes package such as tar_render() and tar_quarto().

Configuration

For several key functions like tar_make(), the default values of arguments are controlled though tar_config_get(). tar_config_get() retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with tar_config_set(). The default file path of this YAML file is _targets.yaml, and you can set another path globally using the TAR_CONFIG environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the TAR_PROJECT environment variable. The structure of the YAML file follows rules similar to the config R package, e.g. projects can inherit settings from one another using the inherits field. Exceptions include:

1. There is no requirement to have a configuration named "default".
2. Other projects do not inherit from the default project' automatically.
3. Not all fields need values because targets already has defaults.

targets does not actually invoke the config package. The implementation in targets was written from scratch without viewing or copying any part of the source code of config.

See Also

Other configuration: tar_config_get(), tar_config_set(), tar_config_unset(), tar_config_yaml(), tar_envvars(), tar_option_get(), tar_option_reset(), tar_option_set()

Examples

```
yaml <- tempfile()
tar_config_set(store = "my_store_a", config = yaml, project = "project_a")
tar_config_set(store = "my_store_b", config = yaml, project = "project_b")
tar_config_projects(config = yaml)
```

tar_config_set *Set configuration settings.*

Description

tar_config_set() writes special custom settings for the current project to an optional YAML configuration file.

Usage

```
tar_config_set(
  inherits = NULL,
  as_job = NULL,
  garbage_collection = NULL,
  label = NULL,
  label_width = NULL,
  level_separation = NULL,
  reporter_make = NULL,
  reporter_outdated = NULL,
  script = NULL,
  seconds_meta_append = NULL,
  seconds_meta_upload = NULL,
  seconds_reporter = NULL,
  seconds_reporter_outdated = NULL,
  seconds_interval = NULL,
  store = NULL,
  shortcut = NULL,
  use_crew = NULL,
  workers = NULL,
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```

Arguments

`inherits`	Character of length 1, name of the project from which the current project should inherit configuration settings. The current project is the `project` argument, which defaults to `Sys.getenv("TAR_PROJECT", "main")`. If the `inherits` argument `NULL`, the `inherits` setting is not modified. Use `tar_config_unset()` to delete a setting.
`as_job`	Logical of length 1, `as_job` argument of `tar_make()`. `TRUE` to run as an RStudio IDE / Posit Workbench job, `FALSE` to run as a `callr` process in the main R session (depending on the `callr_function` argument). If `as_job_` is `TRUE`, then the `rstudioapi` package must be installed.
`garbage_collection`	Deprecated. Use the `garbage_collection` argument of `tar_option_set()` instead to run garbage collection at regular intervals in a pipeline, or use the argument of the same name in `tar_target()` to activate garbage collection for a specific target.
`label`	Argument of `tar_glimpse()` and `tar_visnetwork()` to control node labels.
`label_width`	Argument of `tar_glimpse()` and `tar_visnetwork()` to control the maximum width (number of characters wide) of the node labels.
`level_separation`	Argument of `tar_visnetwork()` and `tar_glimpse()` to control the space between hierarchical levels.

`reporter_make` Character of length 1, `reporter` argument to `tar_make()` and related functions that run the pipeline. If the argument `NULL`, the setting is not modified. Use `tar_config_unset()` to delete a setting.

`reporter_outdated`
Character of length 1, `reporter` argument to `tar_outdated()` and related functions that do not run the pipeline. If the argument `NULL`, the setting is not modified. Use `tar_config_unset()` to delete a setting.

`script` Character of length 1, path to the target script file that defines the pipeline (`_targets.R` by default). This path should be either an absolute path or a path relative to the project root where you will call `tar_make()` and other functions. When `tar_make()` and friends run the script from the current working directory. If the argument `NULL`, the setting is not modified. Use `tar_config_unset()` to delete a setting.

`seconds_meta_append`
Argument of `tar_make()`, `tar_make_clustermq()`, and `tar_make_future()`. Positive numeric of length 1 with the minimum number of seconds between saves to the local metadata and progress files in the data store. This is an aggressive optimization setting not recommended for most users: higher values might make the pipeline run faster, but unsaved work (in the event of a crash) is not up to date.

When the pipeline ends, all the metadata and progress data is saved immediately, regardless of `seconds_meta_append`. When the pipeline is just skipping targets, the actual interval between saves is `max(1, seconds_meta_append)` to reduce overhead.

`seconds_meta_upload`
Argument of `tar_make()`, `tar_make_clustermq()`, and `tar_make_future()`. Positive numeric of length 1 with the minimum number of seconds between uploads of the metadata and progress data to the cloud (see `https://books.ropensci.org/targets/cloud-storage.html`). Higher values generally make the pipeline run faster, but unsaved work (in the event of a crash) may not be backed up to the cloud. When the pipeline ends, all the metadata and progress data is uploaded immediately, regardless of `seconds_meta_upload`.

`seconds_reporter`
Argument of `tar_make()`, `tar_make_clustermq()`, and `tar_make_future()`. Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console (for the aforementioned `tar_make()`-like functions only). This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is `max(1, seconds_reporter)` to reduce overhead.

`seconds_reporter_outdated`
Argument of `tar_outdated()` and other related functions that do not run the pipeline. Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console for `tar_outdated()`.

`seconds_interval`	Deprecated on 2023-08-24 (`targets` version 1.2.2.9001). Use `seconds_meta_append`, `seconds_meta_upload`, and `seconds_reporter` instead.
`store`	Character of length 1, path to the data store of the pipeline. If `NULL`, the `store` setting is left unchanged in the YAML configuration file (default: `_targets.yaml`). Usually, the data store lives at `_targets`. Set `store` to a custom directory to specify a path other than `_targets/`. The path need not exist before the pipeline begins, and it need not end with "_targets", but it must be writeable. For optimal performance, choose a storage location with fast read/write access. If the argument `NULL`, the setting is not modified. Use `tar_config_unset()` to delete a setting.
`shortcut`	logical of length 1, default `shortcut` argument to `tar_make()` and related functions. If the argument `NULL`, the setting is not modified. Use `tar_config_unset()` to delete a setting.
`use_crew`	Logical of length 1, whether to use `crew` in `tar_make()` if the `controller` option is set in `tar_option_set()` in the target script (`_targets.R`). See `https://books.ropensci.org/targets/crew.html` for details.
`workers`	Positive numeric of length 1, `workers` argument of `tar_make_clustermq()` and related functions that run the pipeline with parallel computing among targets. If the argument `NULL`, the setting is not modified. Use `tar_config_unset()` to delete a setting.
`config`	Character of length 1, file path of the YAML configuration file with `targets` project settings. The `config` argument specifies which YAML configuration file that `tar_config_get()` reads from or `tar_config_set()` writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always `_targets.yaml` unless you set another default path using the `TAR_CONFIG` environment variable, e.g. `Sys.setenv(TAR_CONFIG = "custom.yaml")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.
`project`	Character of length 1, name of the current `targets` project. Thanks to the `config` R package, `targets` YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The `project` argument allows you to set or get a configuration setting for a specific project for a given call to `tar_config_set()` or `tar_config_get()`. The default project is always called `"main"` unless you set another default project using the `TAR_PROJECT` environment variable, e.g. `Sys.setenv(tar_project = "custom")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.

Value

`NULL` (invisibly)

Configuration

For several key functions like `tar_make()`, the default values of arguments are controlled though `tar_config_get()`. `tar_config_get()` retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with `tar_config_set()`. The default file path of this YAML file is `_targets.yaml`, and you can set another path globally using the `TAR_CONFIG` environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the `TAR_PROJECT` environment variable. The structure of the YAML file follows rules similar to the `config` R package, e.g. projects can inherit settings from one another using the `inherits` field. Exceptions include:

1. There is no requirement to have a configuration named `"default"`.
2. Other projects do not inherit from the default project‘ automatically.
3. Not all fields need values because `targets` already has defaults.

`targets` does not actually invoke the `config` package. The implementation in `targets` was written from scratch without viewing or copying any part of the source code of `config`.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other configuration: `tar_config_get()`, `tar_config_projects()`, `tar_config_unset()`, `tar_config_yaml()`, `tar_envvars()`, `tar_option_get()`, `tar_option_reset()`, `tar_option_set()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(list(tar_target(x, 1 + 1)))
tar_config_get("store") # NULL (data store defaults to "_targets/")
store_path <- tempfile()
tar_config_set(store = store_path)
tar_config_get("store") # Shows a temp file.
tar_make() # Writes to the custom data store identified in _targets.yaml.
tar_read(x) # tar_read() knows about _targets.yaml too.
file.exists("_targets") # FALSE
file.exists(store_path) # TRUE
})
}
```

tar_config_unset *Unset configuration settings.*

Description

Unset (i.e. delete) one or more custom settings for the current project from the optional YAML configuration file. After that, `tar_option_get()` will return the original default values for those settings for the project.

Usage

```
tar_config_unset(
  names = character(0),
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```

Arguments

`names`	Character vector of configuration settings to delete from the current project.
`config`	Character of length 1, file path of the YAML configuration file with `targets` project settings. The `config` argument specifies which YAML configuration file that `tar_config_get()` reads from or `tar_config_set()` writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always `_targets.yaml` unless you set another default path using the `TAR_CONFIG` environment variable, e.g. `Sys.setenv(TAR_CONFIG = "custom.yaml")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.
`project`	Character of length 1, name of the current `targets` project. Thanks to the `config` R package, `targets` YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The `project` argument allows you to set or get a configuration setting for a specific project for a given call to `tar_config_set()` or `tar_config_get()`. The default project is always called `"main"` unless you set another default project using the `TAR_PROJECT` environment variable, e.g. `Sys.setenv(tar_project = "custom")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.

Value

`NULL` (invisibly)

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

Configuration

For several key functions like `tar_make()`, the default values of arguments are controlled though `tar_config_get()`. `tar_config_get()` retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with `tar_config_set()`. The default file path of this YAML file is `_targets.yaml`, and you can set another path globally using the `TAR_CONFIG` environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the `TAR_PROJECT` environment variable. The structure of the YAML file follows rules similar to the `config` R package, e.g. projects can inherit settings from one another using the `inherits` field. Exceptions include:

1. There is no requirement to have a configuration named `"default"`.
2. Other projects do not inherit from the default project‘ automatically.
3. Not all fields need values because `targets` already has defaults.

`targets` does not actually invoke the `config` package. The implementation in `targets` was written from scratch without viewing or copying any part of the source code of `config`.

See Also

Other configuration: `tar_config_get()`, `tar_config_projects()`, `tar_config_set()`, `tar_config_yaml()`, `tar_envvars()`, `tar_option_get()`, `tar_option_reset()`, `tar_option_set()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(list(tar_target(x, 1 + 1)))
tar_config_get("store") # "_targets"
store_path <- tempfile()
tar_config_set(store = store_path)
tar_config_get("store") # Shows a temp file.
tar_config_unset("store")
tar_config_get("store") # _targets
})
}
```

tar_config_yaml *Read* _targets.yaml.

Description

Read the YAML content of _targets.yaml.

Usage

```
tar_config_yaml(config = Sys.getenv("TAR_CONFIG", "_targets.yaml"))
```

Arguments

config	Character of length 1, file path of the YAML configuration file with targets project settings. The config argument specifies which YAML configuration file that tar_config_get() reads from or tar_config_set() writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always _targets.yaml unless you set another default path using the TAR_CONFIG environment variable, e.g. Sys.setenv(TAR_CONFIG = "custom.yaml"). This also has the effect of temporarily modifying the default arguments to other functions such as tar_make() because the default arguments to those functions are controlled by tar_config_get().

Value

Nested list of fields defined in _targets.yaml.

Storage access

Several functions like tar_make(), tar_read(), tar_load(), tar_meta(), and tar_progress() read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the tarchetypes package such as tar_render() and tar_quarto().

Configuration

For several key functions like tar_make(), the default values of arguments are controlled though tar_config_get(). tar_config_get() retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with tar_config_set(). The default file path of this YAML file is _targets.yaml, and you can set another path globally using the TAR_CONFIG environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the TAR_PROJECT environment variable. The structure of the YAML file follows rules similar to the config R package, e.g. projects can inherit settings from one another using the inherits field. Exceptions include:

1. There is no requirement to have a configuration named `"default"`.
2. Other projects do not inherit from the default project' automatically.
3. Not all fields need values because `targets` already has defaults.

`targets` does not actually invoke the `config` package. The implementation in `targets` was written from scratch without viewing or copying any part of the source code of `config`.

See Also

Other configuration: `tar_config_get()`, `tar_config_projects()`, `tar_config_set()`, `tar_config_unset()`, `tar_envvars()`, `tar_option_get()`, `tar_option_reset()`, `tar_option_set()`

Examples

```
yaml <- tempfile()
tar_config_set(store = "my_store_a", config = yaml, project = "project_a")
tar_config_set(store = "my_store_b", config = yaml, project = "project_b")
str(tar_config_yaml(config = yaml))
```

`tar_crew` *Get crew worker info.*

Description

For the most recent run of the pipeline with `tar_make()` where a `crew` controller was started, get summary-level information of the workers.

Usage

```
tar_crew(store = targets::tar_config_get("store"))
```

Arguments

`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A data frame one row per `crew` controller (potentially multiple rows if `tar_option_get("controller")` is a controller group) and the following columns:

- `controller`: name of the `crew` controller.
- `targets`: number of times the controller attempted to run a target.
- `seconds`: number of seconds the workers spent running tasks.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other data: `tar_pid()`, `tar_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
if (requireNamespace("crew", quietly = TRUE)) {
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(controller = crew::crew_controller_local())
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_process()
tar_process(pid)
}
})
}
```

tar_cue *Declare the rules that cue a target.*

Description

Declare the rules that mark a target as outdated.

Usage

```
tar_cue(
  mode = c("thorough", "always", "never"),
  command = TRUE,
  depend = TRUE,
  format = TRUE,
  repository = TRUE,
  iteration = TRUE,
  file = TRUE,
  seed = TRUE
)
```

Arguments

`mode`	Cue mode. If `"thorough"`, all the cues apply unless individually suppressed. If `"always"`, then the target always runs. If `"never"`, then the target does not run unless the metadata does not exist or the last run errored.
`command`	Logical, whether to rerun the target if command changed since last time.
`depend`	Logical, whether to rerun the target if the value of one of the dependencies changed.
`format`	Logical, whether to rerun the target if the user-specified storage format changed. The storage format is user-specified through `tar_target()` or `tar_option_set()`.
`repository`	Logical, whether to rerun the target if the user-specified storage repository changed. The storage repository is user-specified through `tar_target()` or `tar_option_set()`.
`iteration`	Logical, whether to rerun the target if the user-specified iteration method changed. The iteration method is user-specified through `tar_target()` or `tar_option_set()`.
`file`	Logical, whether to rerun the target if the file(s) with the return value changed or at least one is missing.
`seed`	Logical, whether to rerun the target if pseudo-random number generator seed either changed or is `NA`. The reproducible deterministic target-specific seeds are controlled by `tar_option_get("seed")` and the target names. See `tar_option_set()` for details.

Target invalidation rules

`targets` uses internal metadata and special cues to decide whether a target is up to date (can skip) or is outdated/invalidated (needs to rerun). By default, `targets` moves through the following list of cues and declares a target outdated if at least one is cue activated.

1. There is no metadata record of the target.
2. The target errored last run.
3. The target has a different class than it did before.
4. The cue mode equals `"always"`.
5. The cue mode does not equal `"never"`.
6. The `command` metadata field (the hash of the R command) is different from last time.
7. The `depend` metadata field (the hash of the immediate upstream dependency targets and global objects) is different from last time.
8. The storage format is different from last time.
9. The iteration mode is different from last time.
10. A target's file (either the one in `_targets/objects/` or a dynamic file) does not exist or changed since last time.

The user can suppress many of the above cues using the `tar_cue()` function, which creates the cue argument of `tar_target()`. Cues objects also constitute more nuanced target invalidation rules. The `tarchetypes` package has many such examples, including `tar_age()`, `tar_download()`, `tar_cue_age()`, `tar_cue_force()`, and `tar_cue_skip()`.

Dependency-based invalidation and user-defined functions

If the cue of a target has `depend = TRUE` (default) then the target is marked invalidated/outdated when its upstream dependencies change. A target's dependencies include upstream targets, user-defined functions, and other global objects populated in the target script file (default: `_targets.R`). To determine if a given dependency changed since the last run of the pipeline, `targets` computes hashes. The hash of a target is computed on its files in storage (usually a file in `_targets/objects/`). The hash of a non-function global object dependency is computed directly on its in-memory data. User-defined functions are hashed in the following way:

1. Deparse the function with `targets:::tar_deparse_safe()`. This function computes a string representation of the function body and arguments. This string representation is invariant to changes in comments and whitespace, which means trivial changes to formatting do not cue targets to rerun.
2. Manually remove any literal pointers from the function string using `targets:::mask_pointers()`. Such pointers arise from inline compiled C/C++ functions.
3. Using static code analysis (i.e. `tar_deps()`, which is based on `codetools::findGlobals()`) identify any user-defined functions and global objects that the current function depends on. Append the hashes of those dependencies to the string representation of the current function.
4. Compute the hash of the final string representation using `targets:::hash_object()`.

Above, (3) is important because user-defined functions have dependencies of their own, such as other user-defined functions and other global objects. (3) ensures that a change to a function's dependencies invalidates the function itself, which in turn invalidates any calling functions and any targets downstream with the `depend` cue turned on.

See Also

Other targets: `tar_target()`

Examples

```
# The following target will always run when the pipeline runs.
x <- tar_target(x, download_data(), cue = tar_cue(mode = "always"))
```

`tar_definition` *For developers only: get the definition of the current target.*

Description

For developers only: get the full definition of the target currently running. This target definition is the same kind of object produced by `tar_target()`.

Usage

```
tar_definition(
  default = targets::tar_target_raw("target_name", quote(identity()))
)
```

Arguments

`default`	Environment, value to return if `tar_definition()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.

Details

Most users should not use `tar_definition()` because accidental modifications could break the pipeline. `tar_definition()` only exists in order to support third-party interface packages, and even then the returned target definition is not modified..

Value

If called from a running target, `tar_definition()` returns the target object of the currently running target. See the "Target objects" section for details.

Target objects

Functions like `tar_target()` produce target objects, special objects with specialized sets of S3 classes. Target objects represent skippable steps of the analysis pipeline as described at `https://books.ropensci.org/targets/`. Please read the walkthrough at `https://books.ropensci.org/targets/walkthrough.html` to understand the role of target objects in analysis pipelines.

For developers, `https://wlandau.github.io/targetopia/contributing.html#target-factories` explains target factories (functions like this one which generate targets) and the design specification at `https://books.ropensci.org/targets-design/` details the structure and composition of target objects.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
class(tar_definition())
tar_definition()$name
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(
  tar_target(x, tar_definition()$settings$memory, memory = "transient")
)
tar_make(x)
tar_read(x)
})
}
```

tar_delete *Delete target output values.*

Description

Delete the output values of targets in `_targets/objects/` (or the cloud if applicable) but keep the records in the metadata.

Usage

```
tar_delete(
  names,
  cloud = TRUE,
  batch_size = 1000L,
  verbose = TRUE,
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Optional, names of the targets to delete. If supplied, the `names` argument restricts the targets which are deleted. The value is a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`cloud`	Logical of length 1, whether to delete objects from the cloud if applicable (e.g. AWS, GCP). If `FALSE`, files are not deleted from the cloud.
`batch_size`	Positive integer between 1 and 1000, number of target objects to delete from the cloud with each HTTP API request. Currently only supported for AWS. Cannot be more than 1000.
`verbose`	Logical of length 1, whether to print console messages to show progress when deleting each batch of targets from each cloud bucket. Batched deletion with verbosity is currently only supported for AWS.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

If you have a small number of data-heavy targets you need to discard to conserve storage, this function can help. Local external files files (i.e. `format = "file"` and `repository = "local"`) are not deleted. For targets with `repository` not equal `"local"`, `tar_delete()` attempts to delete the file and errors out if the deletion is unsuccessful. If deletion fails, either log into the cloud platform and manually delete the file (e.g. the AWS web console in the case of `repository = "aws"`) or call `tar_invalidate()` on that target so that `targets` does not try to delete the object. For patterns recorded in the metadata, all the branches will be deleted. For patterns no longer in the metadata, branches are left alone.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

Cloud target data versioning

Some buckets in Amazon S3 or Google Cloud Storage are "versioned", which means they track historical versions of each data object. If you use `targets` with cloud storage (`https://books.ropensci.org/targets/cloud-storage.html`) and versioning is turned on, then `targets` will record each version of each target in its metadata.

Functions like `tar_read()` and `tar_load()` load the version recorded in the local metadata, which may not be the same as the "current" version of the object in the bucket. Likewise, functions `tar_delete()` and `tar_destroy()` only remove the version ID of each target as recorded in the local metadata.

If you want to interact with the *latest* version of an object instead of the version ID recorded in the local metadata, then you will need to delete the object from the metadata.

1. Make sure your local copy of the metadata is current and up to date. You may need to run `tar_meta_download()` or `tar_meta_sync()` first.
2. Run `tar_unversion()` to remove the recorded version IDs of your targets in the local metadata.
3. With the version IDs gone from the local metadata, functions like `tar_read()` and `tar_destroy()` will use the *latest* version of each target data object.
4. Optional: to back up the local metadata file with the version IDs deleted, use `tar_meta_upload()`.

See Also

Other clean: `tar_destroy()`, `tar_invalidate()`, `tar_prune()`, `tar_prune_list()`, `tar_unversion()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
tar_delete(starts_with("y")) # Only deletes y1 and y2.
tar_make() # y1 and y2 rerun but return the same values, so z is up to date.
})
}
```

tar_deps *Code dependencies*

Description

List the dependencies of a function or expression. tar_deps() expects the expr argument to be an unevaluated expression, whereas tar_deps_raw() expects expr to be an evaluated expression object. Functions can be passed normally in either case.

Usage

```
tar_deps(expr)

tar_deps_raw(expr)
```

Arguments

expr	An R expression or function. tar_deps() expects the expr argument to be an unevaluated expression, whereas tar_deps_raw() expects expr to be an evaluated expression object. Functions can be passed normally in either case.

Details

targets detects the dependencies of commands using static code analysis. Use tar_deps() to run the code analysis and see the dependencies for yourself.

Value

Character vector of the dependencies of a function or expression.

See Also

tar_branches(), tar_network()

Other inspect: tar_manifest(), tar_network(), tar_outdated(), tar_sitrep(), tar_validate()

Examples

```
tar_deps(x <- y + z)
tar_deps(quote(x <- y + z))
tar_deps({
  x <- 1
  x + a
})
tar_deps(function(a = b) map_dfr(data, ~do_row(.x)))
tar_deps_raw(function(a = b) map_dfr(data, ~do_row(.x)))
```

`tar_described_as` *Select targets using their descriptions.*

Description

Select a subset of targets in the `_targets.R` file based on their custom descriptions.

Usage

```
tar_described_as(
  described_as = NULL,
  tidyselect = TRUE,
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script")
)
```

Arguments

`described_as`	A `tidyselect` expression to select targets based on their descriptions. For example, `described_as = starts_with("survival model")` matches all targets in the pipeline whose `description` arguments of `tar_target()` start with the text string `"survival model"`.
`tidyselect`	If `TRUE`, return a call to `tidyselect::all_of()` identifying the selected targets, which can then be supplied to any `tidyselect`-compatible `namesargument of downstream functions` return a simple character vector of target names.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.

`envir`	An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

Details

Targets with empty descriptions are ignored.

Value

If `tidyselect` is `TRUE`, then `tar_described_as()` returns a call to `tidyselect::all_of()` which can be supplied to the `names` argument of functions like `tar_manifest()` and `tar_make()`. This allows functions like `tar_manifest()` and `tar_make()` to focus on only the targets with the matching descriptions. If `tidyselect` is `FALSE`, then `tar_described_as()` returns a simple character vector of the names of all the targets in the pipeline with matching descriptions.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(b2, TRUE, description = "blue two"),
    tar_target(b3, TRUE, description = "blue three"),
    tar_target(g2, TRUE, description = "green two"),
    tar_target(g3, TRUE, description = "green three"),
    tar_target(g4, TRUE, description = "green three")
  )
}, ask = FALSE)
tar_described_as(starts_with("green"), tidyselect = FALSE)
tar_make(names = tar_described_as(starts_with("green")))
tar_progress() # Only `g2`, `g3`, and `g4` ran.
})
}
```

`tar_destroy` *Destroy the data store.*

Description

Destroy the data store written by the pipeline.

Usage

```
tar_destroy(
  destroy = c("all", "cloud", "local", "meta", "process", "progress", "objects",
    "scratch", "workspaces", "user"),
  batch_size = 1000L,
  verbose = TRUE,
  ask = NULL,
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`destroy` Character of length 1, what to destroy. Choices:

- `"all"`: entire data store (default: `_targets/`) including cloud data, as well as download/upload scratch files.
- `"cloud"`: cloud data, including metadata as well as target object data from targets with `tar_target(..., repository = "aws")`. Also deletes temporary staging files in `file.path(tempdir(), "targets")` that may have been accidentally left over from incomplete uploads or downloads.
- `"local"`: all the local files in the data store but nothing on the cloud.
- `"meta"`: metadata file at `meta/meta` in the data store, which invalidates all the targets but keeps the data.
- `"process"`: progress data file at `meta/process` in the data store, which resets the metadata of the main process.
- `"progress"`: progress data file at `meta/progress` in the data store, which resets the progress tracking info.
- `"objects"`: all the target return values in `objects/` in the data store but keep progress and metadata. Dynamic files are not deleted this way.
- `"scratch"`: temporary files in saved during `tar_make()` that should automatically get deleted except if R crashed.

- "workspaces": compressed lightweight files in workspaces/ in the data store with the saved workspaces of targets. See tar_workspace() for details.
- "user": custom user-supplied files in the user/ folder in the data store.

batch_size	Positive integer between 1 and 1000, number of target objects to delete from the cloud with each HTTP API request. Currently only supported for AWS. Cannot be more than 1000.
verbose	Logical of length 1, whether to print console messages to show progress when deleting each batch of targets from each cloud bucket. Batched deletion with verbosity is currently only supported for AWS.
ask	Logical of length 1, whether to pause with a menu prompt before deleting files. To disable this menu, set the TAR_ASK environment variable to "false". usethis::edit_r_environ() can help set environment variables.
script	Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. If the script does not exist, then cloud metadata will not be deleted.
store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Details

The data store is a folder created by tar_make() (or tar_make_future() or tar_make_clustermq()). The details of the data store are explained at https://books.ropensci.org/targets/data.html#local-data-store. The data store folder contains the output data and metadata of the targets in the pipeline. Usually, the data store is a folder called _targets/ (see tar_config_set() to customize), and it may link to data on the cloud if you used AWS or GCP buckets. By default, tar_destroy() deletes the entire _targets/ folder (or wherever the data store is located), including custom user-supplied files in _targets/user/, as well as any cloud data that the pipeline uploaded. See the destroy argument to customize this behavior and only delete part of the data store, and see functions like tar_invalidate(), tar_delete(), and tar_prune() to remove information pertaining to some but not all targets in the pipeline. After calling tar_destroy() with default arguments, the entire data store is gone, which means all the output data from previous runs of the pipeline is gone (except for input/output files tracked with tar_target(..., format = "file")). The next run of the pipeline will start from scratch, and it will not skip any targets.

Value

NULL (invisibly).

Storage access

Several functions like tar_make(), tar_read(), tar_load(), tar_meta(), and tar_progress() read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the tarchetypes package such as tar_render() and tar_quarto().

Cloud target data versioning

Some buckets in Amazon S3 or Google Cloud Storage are "versioned", which means they track historical versions of each data object. If you use targets with cloud storage (https://books.ropensci.org/targets/cloud-storage.html) and versioning is turned on, then targets will record each version of each target in its metadata.

Functions like tar_read() and tar_load() load the version recorded in the local metadata, which may not be the same as the "current" version of the object in the bucket. Likewise, functions tar_delete() and tar_destroy() only remove the version ID of each target as recorded in the local metadata.

If you want to interact with the *latest* version of an object instead of the version ID recorded in the local metadata, then you will need to delete the object from the metadata.

1. Make sure your local copy of the metadata is current and up to date. You may need to run tar_meta_download() or tar_meta_sync() first.
2. Run tar_unversion() to remove the recorded version IDs of your targets in the local metadata.
3. With the version IDs gone from the local metadata, functions like tar_read() and tar_destroy() will use the *latest* version of each target data object.
4. Optional: to back up the local metadata file with the version IDs deleted, use tar_meta_upload().

See Also

Other clean: tar_delete(), tar_invalidate(), tar_prune(), tar_prune_list(), tar_unversion()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(tar_target(x, 1 + 1))
})
tar_make() # Creates the _targets/ data store.
tar_destroy()
print(file.exists("_targets")) # Should be FALSE.
})
}
```

tar_dispatched *List dispatched targets.*

Description

List the targets with progress status "dispatched".

Usage

```
tar_dispatched(names = NULL, store = targets::tar_config_get("store"))
```

Arguments

names	Optional, names of the targets. If supplied, the function restricts its output to these targets. You can supply symbols or tidyselect helpers like any_of() and starts_with().
store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Details

A target is "dispatched" if it is sent off to be run. Depending on your high-performance computing configuration via the crew package, the may not actually start right away. This may happen if the target is ready to start but all available parallel workers are busy.

Value

A character vector of dispatched targets.

See Also

Other progress: tar_canceled(), tar_completed(), tar_errored(), tar_poll(), tar_progress(), tar_progress_branches(), tar_progress_summary(), tar_skipped(), tar_watch(), tar_watch_server(), tar_watch_ui()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_dispatched()
tar_dispatched(starts_with("y_")) # see also any_of()
})
}
```

`tar_edit` *Open the target script file for editing.*

Description

Open the target script file for editing. Requires the `usethis` package.

Usage

```
tar_edit(script = targets::tar_config_get("script"))
```

Arguments

`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

Details

The target script file is an R code file that defines the pipeline. The default path is `_targets.R`, but the default for the current project can be configured with `tar_config_set()`

See Also

Other scripts: `tar_github_actions()`, `tar_helper()`, `tar_renv()`, `tar_script()`

`tar_engine_knitr` *Target Markdown* `knitr` *engine*

Description

`knitr` language engine that runs `targets` code chunks in Target Markdown.

Usage

```
tar_engine_knitr(options)
```

Arguments

`options`	A named list of `knitr` chunk options.

Value

Character, output generated from `knitr::engine_output()`.

Target Markdown interactive mode

Target Markdown has two modes:

1. Non-interactive mode. This is the default when you run `knitr::knit()` or `rmarkdown::render()`. Here, the code in `targets` code chunks gets written to special script files in order to set up a `targets` pipeline to run later.
2. Interactive mode: here, no scripts are written to set up a pipeline. Rather, the globals or targets in question are run in the current environment and the values are assigned to that environment.

The mode is interactive if `!isTRUE(getOption("knitr.in.progress"))`, is `TRUE`. The `knitr.in.progress` option is `TRUE` when you run `knitr::knit()` or `rmarkdown::render()` and `NULL` if you are running one chunk at a time interactively in an integrated development environment, e.g. the notebook interface in RStudio: `https://bookdown.org/yihui/rmarkdown/notebook.html`. You can choose the mode with the `tar_interactive` chunk option. (In `targets` 0.6.0, `tar_interactive` defaults to `interactive()` instead of `!isTRUE(getOption("knitr.in.progress"))`.)

Target Markdown chunk options

Target Markdown introduces the following `knitr` code chunk options. Most other standard `knitr` code chunk options should just work in non-interactive mode. In interactive mode, not all

- `tar_globals`: Logical of length 1, whether to define globals or targets. If `TRUE`, the chunk code defines functions, objects, and options common to all the targets. If `FALSE` or `NULL` (default), then the chunk returns formal targets for the pipeline.
- `tar_interactive`: Logical of length 1, whether to run in interactive mode or non-interactive mode. See the "Target Markdown interactive mode" section of this help file for details.
- `tar_name`: name to use for writing helper script files (e.g. `_targets_r/targets/target_script.R`) and specifying target names if the `tar_simple` chunk option is `TRUE`. All helper scripts and target names must have unique names, so please do not set this option globally with `knitr::opts_chunk$set()`.
- `tar_script`: Character of length 1, where to write the target script file in non-interactive mode. Most users can skip this option and stick with the default `_targets.R` script path. Helper script files are always written next to the target script in a folder with an `"_r"` suffix. The `tar_script` path must either be absolute or be relative to the project root (where you call `tar_make()` or similar). If not specified, the target script path defaults to `tar_config_get("script")` (default: `_targets.R`; helpers default: `_targets_r/`). When you run `tar_make()` etc. with a non-default target script, you must select the correct target script file either with the `script` argument or with `tar_config_set(script = ...)`. The function will `source()` the script file from the current working directory (i.e. with `chdir = FALSE` in `source()`).

- tar_simple: Logical of length 1. Set to TRUE to define a single target with a simplified interface. In code chunks with tar_simple equal to TRUE, the chunk label (or the tar_name chunk option if you set it) becomes the name, and the chunk code becomes the command. In other words, a code chunk with label targetname and command mycommand() automatically gets converted to tar_target(name = targetname, command = mycommand()). All other arguments of tar_target() remain at their default values (configurable with tar_option_set() in a tar_globals = TRUE chunk).

See Also

https://books.ropensci.org/targets/literate-programming.html

Other Target Markdown: tar_interactive(), tar_noninteractive(), tar_toggle()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
# Register the engine.
if (requireNamespace("knitr", quietly = TRUE)) {
  knitr::knit_engines$set(targets = targets::tar_engine_knitr)
}
# Then, `targets` code chunks in a knitr report will run
# as described at
# <https://books.ropensci.org/targets/literate-programming.html>.
}
```

---

tar_envir — *For developers only: get the environment of the current target.*

Description

For developers only: get the environment where a target runs its command. Designed to be called while the target is running. The environment inherits from tar_option_get("envir").

Usage

```
tar_envir(default = parent.frame())
```

Arguments

default	Environment, value to return if tar_envir() is called on its own outside a targets pipeline. Having a default lets users run things without tar_make(), which helps peel back layers of code and troubleshoot bugs.

Details

Most users should not use tar_envir() because accidental modifications to parent.env(tar_envir()) could break the pipeline. tar_envir() only exists in order to support third-party interface packages, and even then the returned environment is not modified.

Value

If called from a running target, `tar_envir()` returns the environment where the target runs its command. If called outside a pipeline, the return value is whatever the user supplies to `default` (which defaults to `parent.frame()`).

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
tar_envir()
tar_envir(default = new.env(parent = emptyenv()))
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(tar_target(x, tar_envir(default = parent.frame())))
tar_make(x)
tar_read(x)
})
}
```

`tar_envvars` *Show* `targets` *environment variables.*

Description

Show all the special environment variables available for customizing `targets`.

Usage

```
tar_envvars(unset = "")
```

Arguments

`unset`	Character of length 1, value to return for any environment variable that is not set.

Details

You can customize the behavior of `targets` with special environment variables. The sections in this help file describe each environment variable, and the `tar_envvars()` function lists their current values.

If you modify environment variables, please set them in project-level `.Renviron` file so you do not lose your configuration when you restart your R session. Modify the project-level `.Renviron` file with usethis::edit_r_environ(scope = "project"). Restart your R session after you are done editing.

For targets that run on parallel workers created by tar_make_clustermq() or tar_make_future(), only the environment variables listed by tar_envvars() are specifically exported to the targets. For all other environment variables, you will have to set the values manually, e.g. a project-level .Renviron file (for workers that have access to the local file system).

Value

A data frame with one row per environment variable and columns with the name and current value of each. An unset environment variable will have a value of "" by default. (Customize with the unset argument).

TAR_ASK

The TAR_ASK environment variable accepts values "true" and "false". If TAR_ASK is not set, or if it is set to "true", then targets asks permission in a menu before overwriting certain files, such as the target script file (default: _targets.R) in tar_script(). If TAR_ASK is "false", then targets overwrites the old files with the new ones without asking. Once you are comfortable with tar_script(), tar_github_actions(), and similar functions, you can safely set TAR_ASK to "false" in either a project-level or user-level .Renviron file.

TAR_CONFIG

The TAR_CONFIG environment variable controls the file path to the optional YAML configuration file with project settings. See the help file of tar_config_set() for details.

TAR_PROJECT

The TAR_PROJECT environment variable sets the name of project to set and get settings when working with the YAML configuration file. See the help file of tar_config_set() for details.

TAR_WARN

The TAR_WARN environment variable accepts values "true" and "false". If TAR_WARN is not set, or if it is set to "true", then targets throws warnings in certain edge cases, such as target/global name conflicts and dangerous use of devtools::load_all(). If TAR_WARN is "false", then targets does not throw warnings in these cases. These warnings can detect potentially serious issues with your pipeline, so please do not set TAR_WARN unless your use case absolutely requires it.

See Also

Other configuration: tar_config_get(), tar_config_projects(), tar_config_set(), tar_config_unset(), tar_config_yaml(), tar_option_get(), tar_option_reset(), tar_option_set()

Examples

```
tar_envvars()
```

tar_errored *List errored targets.*

Description

List targets whose progress is "errored".

Usage

```
tar_errored(names = NULL, store = targets::tar_config_get("store"))
```

Arguments

names	Optional, names of the targets. If supplied, the output is restricted to the selected targets. The object supplied to names should be NULL or a tidyselect expression like any_of() or starts_with() from tidyselect itself, or tar_described_as() to select target names based on their descriptions.
store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Value

A character vector of errored targets.

Storage access

Several functions like tar_make(), tar_read(), tar_load(), tar_meta(), and tar_progress() read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the tarchetypes package such as tar_render() and tar_quarto().

See Also

Other progress: tar_canceled(), tar_completed(), tar_dispatched(), tar_poll(), tar_progress(), tar_progress_branches(), tar_progress_summary(), tar_skipped(), tar_watch(), tar_watch_server(), tar_watch_ui()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
```

```
    library(tarchetypes)
    list(
      tar_target(x, seq_len(2)),
      tar_target(y, 2 * x, pattern = map(x))
    )
  }, ask = FALSE)
  tar_make()
  tar_errored()
  tar_errored(starts_with("y_")) # see also any_of()
  })
  }
```

tar_exist_meta   *Check if target metadata exists.*

Description

Check if the target metadata file `_targets/meta/meta` exists for the current project.

Usage

```
tar_exist_meta(store = targets::tar_config_get("store"))
```

Arguments

`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

To learn more about data storage in `targets`, visit `https://books.ropensci.org/targets/data.html`.

Value

Logical of length 1, whether the current project's metadata exists.

See Also

Other existence: `tar_exist_objects()`, `tar_exist_process()`, `tar_exist_progress()`, `tar_exist_script()`

Examples

```
tar_exist_meta()
```

`tar_exist_objects` *Check if local output data exists for one or more targets.*

Description

Check if output target data exists in either `_targets/objects/` or the cloud for one or more targets.

Usage

```
tar_exist_objects(
  names,
  cloud = TRUE,
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Character vector of target names. Not `tidyselect`-compatible.
`cloud`	Logical of length 1, whether to include cloud targets in the output (e.g. `tar_target(..., repository = "aws")`).
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

If a target has no metadata or if the `repository` argument of `tar_target()` was set to `"local"`, then the `_targets/objects/` folder is checked. Otherwise, if there is metadata and `repsitory` is not `"local"`, then `tar_exist_objects()` checks the cloud repository selected.

Value

Logical of length `length(names)`, whether each given target has an existing file in either `_targets/objects/` or the cloud.

See Also

Other existence: `tar_exist_meta()`, `tar_exist_process()`, `tar_exist_progress()`, `tar_exist_script()`

Examples

```
tar_exist_objects(c("target1", "target2"))
```

tar_exist_process *Check if process metadata exists.*

Description

Check if the process metadata file `_targets/meta/process` exists for the current project.

Usage

```
tar_exist_process(store = targets::tar_config_get("store"))
```

Arguments

`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

To learn more about data storage in `targets`, visit `https://books.ropensci.org/targets/data.html`.

Value

Logical of length 1, whether the current project's metadata exists.

See Also

Other existence: `tar_exist_meta()`, `tar_exist_objects()`, `tar_exist_progress()`, `tar_exist_script()`

Examples

```
tar_exist_process()
```

tar_exist_progress *Check if progress metadata exists.*

Description

Check if the progress metadata file `_targets/meta/progress` exists for the current project.

Usage

```
tar_exist_progress(store = targets::tar_config_get("store"))
```

Arguments

`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

To learn more about data storage in `targets`, visit `https://books.ropensci.org/targets/data.html`.

Value

Logical of length 1, whether the current project's metadata exists.

See Also

Other existence: `tar_exist_meta()`, `tar_exist_objects()`, `tar_exist_process()`, `tar_exist_script()`

Examples

```
tar_exist_progress()
```

`tar_exist_script` *Check if the target script file exists.*

Description

Check if the target script file exists for the current project. The target script is `_targets.R` by default, but the path can be configured for the current project using `tar_config_set()`.

Usage

```
tar_exist_script(script = targets::tar_config_get("script"))
```

Arguments

`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

Value

Logical of length 1, whether the current project's metadata exists.

See Also

Other existence: `tar_exist_meta()`, `tar_exist_objects()`, `tar_exist_process()`, `tar_exist_progress()`

Examples

```
tar_exist_script()
```

`tar_format` *Define a custom target storage format.*

Description

Define a custom target storage format for the `format` argument of `tar_target()` or `tar_option_set()`.

Usage

```
tar_format(
  read = NULL,
  write = NULL,
  marshal = NULL,
  unmarshal = NULL,
  convert = NULL,
  copy = NULL,
  substitute = list(),
  repository = NULL
)
```

Arguments

`read`	A function with a single argument named `path`. This function should read and return the target stored at the file in the argument. It should have no side effects. See the "Format functions" section for specific requirements. If `NULL`, the `read` argument defaults to `readRDS()`.
`write`	A function with two arguments: `object` and `path`, in that order. This function should save the R object `object` to the file path at `path` and have no other side effects. The function need not return a value, but the file written to `path` must be a single file, and it cannot be a directory. See the "Format functions" section for specific requirements. If `NULL`, the `write` argument defaults to `saveRDS()` with `version = 3`.
`marshal`	A function with a single argument named `object`. This function should marshal the R object and return an in-memory object that can be exported to remote parallel workers. It should not read or write any persistent files. See the Marshalling section for details. See the "Format functions" section for specific requirements. If `NULL`, the `marshal` argument defaults to just returning the original object without any modifications.

`unmarshal`	A function with a single argument named `object`. This function should unmarshal the (marshalled) R object and return an in-memory object that is appropriate and valid for use on a parallel worker. It should not read or write any persistent files. See the Marshalling section for details. See the "Format functions" section for specific requirements. If `NULL`, the `unmarshal` argument defaults to just returning the original object without any modifications.
`convert`	The `convert` argument is a function with a single argument named `object`. It accepts the object returned by the command of the target and changes it into an acceptable format (e.g. can be saved with the `read` function). The `convert` ensures the in-memory copy of an object during the running pipeline session is the same as the copy of the object that is saved to disk. The function should be idempotent, and it should handle edge cases like `NULL` values (especially for `error = "null"` in `tar_target()` or `tar_option_set()`). If `NULL`, the `convert` argument defaults to just returning the original object without any modifications.
`copy`	The `copy` argument is a function with a single function named `object`. It accepts the object returned by the command of the target and makes a deep copy in memory. This method does is relevant to objects like `data.tables` that support in-place modification which could cause unpredictable side effects from target to target. In cases like these, the target should be deep-copied before a downstream target attempts to use it (in the case of `data.table` objects, using `data.table::copy()`). If `NULL`, the `copy` argument defaults to just returning the original object without any modifications.
`substitute`	Named list of values to be inserted into the body of each custom function in place of symbols in the body. For example, if `write = function(object, path) saveRDS(object, path, version = VERSION)` and `substitute = list(VERSION = 3)`, then the `write` function will actually end up being `function(object, path) saveRDS(object, path, version = 3)`. Please do not include temporary or sensitive information such as authentication credentials. If you do, then `targets` will write them to metadata on disk, and a malicious actor could steal and misuse them. Instead, pass sensitive information as environment variables using `tar_resources_custom_format()`. These environment variables only exist in the transient memory spaces of the R sessions of the local and worker processes.
`repository`	Deprecated. Use the `repository` argument of `tar_target()` or `tar_option_set()` instead.

Value

A character string of length 1 encoding the custom format. You can supply this string directly to the `format` argument of `tar_target()` or `tar_option_set()`.

Marshalling

If an object can only be used in the R session where it was created, it is called "non-exportable". Examples of non-exportable R objects are Keras models, Torch objects, `xgboost` matrices, `xml2` documents, `rstan` model objects, `sparklyr` data objects, and database connection objects. These objects cannot be exported to parallel workers (e.g. for `tar_make_future()`) without special treatment. To send an non-exportable object to a parallel worker, the object must be marshalled: converted into a form that can be exported safely (similar to serialization but not always the same). Then, the worker must unmarshal the object: convert it into a form that is usable and valid in the current R session. Arguments `marshal` and `unmarshal` of `tar_format()` let you control how marshalling and unmarshalling happens.

Format functions

In `tar_format()`, functions like `read`, `write`, `marshal`, and `unmarshal` must be perfectly pure and perfectly self-sufficient. They must load or namespace all their own packages, and they must not depend on any custom user-defined functions or objects in the global environment of your pipeline. `targets` converts each function to and from text, so it must not rely on any data in the closure. This disqualifies functions produced by `Vectorize()`, for example.

The `write` function must write only a single file, and the file it writes must not be a directory.

The functions to read and write the object should not do any conversions on the object. That is the job of the `convert` argument. The `convert` argument is a function that accepts the object returned by the command of the target and changes it into an acceptable format (e.g. can be saved with the `read` function). Working with the `convert` function is best because it ensures the in-memory copy of an object during the running pipeline session is the same as the copy of the object that is saved to disk.

See Also

Other storage: `tar_load()`, `tar_load_everything()`, `tar_objects()`, `tar_read()`

Examples

```
# The following target is equivalent to the current superseded
# tar_target(name, command(), format = "keras").
# An improved version of this would supply a `convert` argument
# to handle NULL objects, which are returned by the target if it
# errors and the error argument of tar_target() is "null".
tar_target(
  name = keras_target,
  command = your_function(),
  format = tar_format(
    read = function(path) {
      keras::load_model_hdf5(path)
    },
    write = function(object, path) {
      keras::save_model_hdf5(object = object, filepath = path)
    },
    marshal = function(object) {
      keras::serialize_model(object)
    },
    unmarshal = function(object) {
      keras::unserialize_model(object)
    }
  )
)
# And the following is equivalent to the current superseded
# tar_target(name, torch::torch_tensor(seq_len(4)), format = "torch"),
# except this version has a `convert` argument to handle
# cases when `NULL` is returned (e.g. if the target errors out
# and the `error` argument is "null" in tar_target()
# or tar_option_set())
tar_target(
  name = torch_target,
  command = torch::torch_tensor(),
  format = tar_format(
    read = function(path) {
      torch::torch_load(path)
    },
    write = function(object, path) {
      torch::torch_save(obj = object, path = path)
    },
    marshal = function(object) {
      con <- rawConnection(raw(), open = "wr")
      on.exit(close(con))
      torch::torch_save(object, con)
      rawConnectionValue(con)
    },
    unmarshal = function(object) {
      con <- rawConnection(object, open = "r")
      on.exit(close(con))
      torch::torch_load(con)
    }
  )
)
```

tar_format_get *Current storage format.*

Description

Get the storage format of the target currently running.

Usage

```
tar_format_get()
```

Details

This function is meant to be called inside a target in a running pipeline. If it is called outside a target in the running pipeline, it will return the default format given by `tar_option_get("format")`.

Value

A character string, storage format of the target currently running in the pipeline. If called outside a target in the running pipeline, `tar_format_get()` will return the default format given by `tar_option_get("format")`.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
tar_target(x, tar_format_get(), format = "qs")
```

`tar_github_actions` *Set up GitHub Actions to run a targets pipeline*

Description

Writes a GitHub Actions workflow file so the pipeline runs on every push to GitHub. Historical runs accumulate in the `targets-runs` branch, and the latest output is restored before `tar_make()` so up-to-date targets do not rerun.

Usage

```
tar_github_actions(
  path = file.path(".github", "workflows", "targets.yaml"),
  ask = NULL
)
```

Arguments

`path`	Character of length 1, file path to write the GitHub Actions workflow file.
`ask`	Logical, whether to ask before writing if the workflow file already exists. If `NULL`, defaults to `Sys.getenv("TAR_ASK")`. (Set to `"true"` or `"false"` with `Sys.setenv()`). If `ask` and the `TAR_ASK` environment variable are both indeterminate, defaults to `interactive()`.

Details

Steps to set up continuous deployment:

1. Ensure your pipeline stays within the resource limitations of GitHub Actions and repositories, both for storage and compute. For storage, you may wish to reduce the burden with an alternative repository (e.g. `tar_target(..., repository = "aws")`).

2. Ensure Actions are enabled in your GitHub repository. You may have to visit the Settings tab.
3. Call `targets::tar_renv(extras = character(0))` to expose hidden package dependencies.
4. Set up `renv` for your project (with `renv::init()` or `renv::snapshot()`). Details at `https://rstudio.github.io/renv/articles/ci.html`.
5. Commit the `renv.lock` file to the `main` (recommended) or `master` Git branch.
6. Run `tar_github_actions()` to create the workflow file. Commit this file to `main` (recommended) or `master` in Git.
7. Push your project to GitHub. Verify that a GitHub Actions workflow runs and pushes results to `targets-runs`. Subsequent runs will only recompute the outdated targets.

Value

Nothing (invisibly). This function writes a GitHub Actions workflow file as a side effect.

See Also

Other scripts: `tar_edit()`, `tar_helper()`, `tar_renv()`, `tar_script()`

Examples

```
tar_github_actions(tempfile())
```

`tar_glimpse` *Visualize an abridged fast dependency graph.*

Description

Analyze the pipeline defined in the target script file (default: `_targets.R`) and visualize the directed acyclic graph of targets. Unlike `tar_visnetwork()`, `tar_glimpse()` does not account for metadata or progress information, which means the graph renders faster. Also, `tar_glimpse()` omits functions and other global objects by default (but you can include them with `targets_only = FALSE`).

Usage

```
tar_glimpse(
  targets_only = TRUE,
  names = NULL,
  shortcut = FALSE,
  allow = NULL,
  exclude = ".Random.seed",
  label = targets::tar_config_get("label"),
  label_width = targets::tar_config_get("label_width"),
  level_separation = targets::tar_config_get("level_separation"),
  degree_from = 1L,
  degree_to = 1L,
  zoom_speed = 1,
  physics = FALSE,
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

targets_only	Logical, whether to restrict the output to just targets (FALSE) or to also include global functions and objects.
names	Names of targets. The graph visualization will operate only on these targets (and unless shortcut is TRUE, all the targets upstream as well). Selecting a small subgraph using names could speed up the load time of the visualization. Unlike allow, names is invoked before the graph is generated. Set to NULL to check/run all the targets (default). Otherwise, the object supplied to names should be a tidyselect expression like any_of() or starts_with() from tidyselect itself, or tar_described_as() to select target names based on their descriptions.
shortcut	Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. If TRUE, then the function only checks the targets in names and uses stored metadata for information about upstream dependencies as needed. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, shortcut = TRUE only works if you set names.
allow	Optional, define the set of allowable vertices in the graph. Unlike names, allow is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to NULL to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or tidyselect helpers like starts_with().
exclude	Optional, define the set of exclude vertices from the graph. Unlike names, exclude is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to NULL to exclude no vertices. Otherwise, you can supply symbols or tidyselect helpers like any_of() and starts_with().
label	Character vector of one or more aesthetics to add to the vertex labels. Currently, the only option is "description" to show each target's custom description, or character(0) to suppress it.
label_width	Positive numeric of length 1, maximum width (in number of characters) of the node labels.
level_separation	Numeric of length 1, levelSeparation argument of visNetwork::visHierarchicalLayout(). Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If level_separation is NULL, the levelSeparation argument of visHierarchicalLayout() defaults to 150.

`degree_from`	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_from` controls the number of edges the neighborhood extends upstream.
`degree_to`	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_to` controls the number of edges the neighborhood extends downstream.
`zoom_speed`	Positive numeric of length 1, scaling factor on the zoom speed. Above 1 zooms faster than default, below 1 zooms lower than default.
`physics`	Logical of length 1, whether to implement interactive physics in the graph, e.g. edge elasticity.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.
`envir`	An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A `visNetwork` HTML widget object.

Dependency graph

The dependency graph of a pipeline is a directed acyclic graph (DAG) where each node indicates a target or global object and each directed edge indicates where a downstream node depends on an upstream node. The DAG is not always a tree, but it never contains a cycle because no target is allowed to directly or indirectly depend on itself. The dependency graph should show a natural progression of work from left to right. `targets` uses static code analysis to create the graph, so the order of `tar_target()` calls in the `_targets.R` file does not matter. However, targets does not support self-referential loops or other cycles. For more information on the dependency graph, please read `https://books.ropensci.org/targets/targets.html#dependencies`.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other visualize: `tar_mermaid()`, `tar_visnetwork()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_glimpse()
tar_glimpse(allow = starts_with("y")) # see also any_of()
})
}
```

`tar_group` *Group a data frame to iterate over subsets of rows.*

Description

Like `dplyr::group_by()`, but for patterns. `tar_group()` allows you to map or cross over subsets of data frames. Requires `iteration = "group"` on the target. See the example.

Usage

```
tar_group(x)
```

Arguments

`x`	Grouped data frame from `dplyr::group_by()`

Details

The goal of `tar_group()` is to post-process the return value of a data frame target to allow downstream targets to branch over subsets of rows. It takes the groups defined by `dplyr::group_by()` and translates that information into a special `tar_group` is a column. `tar_group` is a vector of positive integers from 1 to the number of groups. Rows with the same integer in `tar_group` belong to the same group, and branches are arranged in increasing order with respect to the integers in `tar_group`. The assignment of `tar_group` integers to group levels depends on the orderings inside the grouping variables and not the order of rows in the dataset. `dplyr::group_keys()` on the grouped data frame shows how the grouping variables correspond to the integers in the `tar_group` column.

Value

A data frame with a special `tar_group` column that `targets` will use to find subsets of your data frame.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
# The tar_group() function simply creates
# a tar_group column to partition the rows
# of a data frame.
data.frame(
  x = seq_len(6),
  id = rep(letters[seq_len(3)], each = 2)
) %>%
  dplyr::group_by(id) %>%
  tar_group()
# We use tar_group() below to branch over
# subsets of a data frame defined with dplyr::group_by().
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
library(dplyr)
library(targets)
library(tarchetypes)
list(
  tar_target(
    data,
    data.frame(
      x = seq_len(6),
      id = rep(letters[seq_len(3)], each = 2)
    ) %>%
      group_by(id) %>%
      tar_group(),
    iteration = "group"
  ),
  tar_target(
    sums,
    sum(data$x),
    pattern = map(data),
    iteration = "vector"
  )
)
})
tar_make()
tar_read(sums) # Should be c(3, 7, 11).
})
}
```

`tar_helper` *Write a helper R script.*

Description

Write a helper R script for a `targets` pipeline. Could be supporting functions or the target script file (default: `_targets.R`) itself.

`tar_helper()` expects an unevaluated expression for the code argument, whereas `tar_helper_raw()` expects an evaluated expression object.

Usage

```
tar_helper(path = NULL, code = NULL, tidy_eval = TRUE, envir = parent.frame())

tar_helper_raw(path = NULL, code = NULL)
```

Arguments

`path`	Character of length 1, path to write (or overwrite) code. If the parent directory does not exist, `tar_helper_raw()` creates it. `tar_helper()` overwrites the file if it already exists.
`code`	Code to write to `path`. `tar_helper()` expects an unevaluated expression for the `code` argument, whereas `tar_helper_raw()` expects an evaluated expression object.
`tidy_eval`	Logical, whether to use tidy evaluation on code. If turned on, you can substitute expressions and symbols using `!!` and `!!!`. See examples below.
`envir`	Environment for tidy evaluation.

Details

`tar_helper()` is a specialized version of `tar_script()` with flexible paths and tidy evaluation.

Value

`NULL` (invisibly)

See Also

Other scripts: `tar_edit()`, `tar_github_actions()`, `tar_renv()`, `tar_script()`

Examples

```
# Without tidy evaluation:
path <- tempfile()
tar_helper(path, code = x <- 1)
tar_helper_raw(path, code = quote(x <- 1)) # equivalent
writeLines(readLines(path))
# With tidy evaluation:
y <- 123
tar_helper(path, x <- !!y)
writeLines(readLines(path))
```

`tar_interactive` *Run if Target Markdown interactive mode is on.*

Description

In Target Markdown, run the enclosed code only if interactive mode is activated. Otherwise, do not run the code.

Usage

```
tar_interactive(code)
```

Arguments

`code`	R code to run if Target Markdown interactive mode is turned on.

Details

Visit <books.ropensci.org/targets/literate-programming.html> to learn about Target Markdown and interactive mode.

Value

If Target Markdown interactive mode is turned on, the function returns the result of running the code. Otherwise, the function invisibly returns `NULL`.

See Also

Other Target Markdown: tar_engine_knitr(), tar_noninteractive(), tar_toggle()

Examples

```
tar_interactive(message("In interactive mode."))
```

---

tar_invalidate *Delete one or more metadata records (e.g. to rerun a target).*

Description

Delete the metadata of records in _targets/meta/meta but keep the return values of targets in _targets/objects/.

Usage

```
tar_invalidate(names, store = targets::tar_config_get("store"))
```

Arguments

names	Names of the targets to remove from the metadata list. The object supplied to names should be a tidyselect expression like any_of() or starts_with() from tidyselect itself, or tar_described_as() to select target names based on their descriptions.
store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Details

This function forces one or more targets to rerun on the next tar_make(), regardless of the cues and regardless of how those targets are stored. After tar_invalidate(), you will still be able to locate the data files with tar_path_target() and manually salvage them in an emergency. However, tar_load() and tar_read() will not be able to read the data into R, and subsequent calls to tar_make() will attempt to rerun those targets. For patterns recorded in the metadata, all the branches will be invalidated. For patterns no longer in the metadata, branches are left alone.

Value

NULL (invisibly).

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other clean: `tar_delete()`, `tar_destroy()`, `tar_prune()`, `tar_prune_list()`, `tar_unversion()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
tar_invalidate(starts_with("y")) # Only invalidates y1 and y2.
tar_make() # y1 and y2 rerun but return same values, so z is up to date.
})
}
```

`tar_language` *Language*

Description

These functions help with metaprogramming in packages built on top of `targets`.

Usage

```
tar_deparse_language(expr)

tar_deparse_safe(expr, collapse = "\n", backtick = TRUE)

tar_tidy_eval(expr, envir, tidy_eval)

tar_tidyselect_eval(names_quosure, choices)
```

Arguments

`expr`	A language object to modify or deparse.
`collapse`	Character of length 1, delimiter in deparsing.
`backtick`	logical indicating whether symbolic names should be enclosed in backticks if they do not follow the standard syntax.
`envir`	An environment to find objects for tidy evaluation.
`tidy_eval`	Logical of length 1, whether to apply tidy evaluation.
`names_quosure`	An `rlang` quosure with `tidyselect` expressions.
`choices`	A character vector of choices for character elements returned by tidy evaluation.

Details

- `tar_deparse_language()` is a wrapper around `tar_deparse_safe()` which leaves character vectors and `NULL` objects alone, which helps with subsequent user input validation.
- `tar_deparse_safe()` is a wrapper around `base::deparse()` with a custom set of fast default settings and guardrails to ensure the output always has length 1.
- `tar_tidy_eval()` applies tidy evaluation to a language object and returns another language object.
- `tar_tidyselect_eval()` applies `tidyselect` selection with some special guardrails around `NULL` inputs.

See Also

Other utilities to extend targets: `tar_assert`, `tar_condition`, `tar_test()`

Examples

```
tar_deparse_language(quote(run_model()))
```

`tar_load` *Load the values of targets.*

Description

Load the return values of targets into the current environment (or the environment of your choosing). For a typical target, the return value lives in a file in `_targets/objects/`. For dynamic files (i.e. `format = "file"`) the paths loaded in place of the values. `tar_load_everything()` is shorthand for `tar_load(everything())` to load all targets.

`tar_load()` uses non-standard evaluation in the `names` argument (example: `tar_load(names = everything())`), whereas `tar_load_raw()` uses standard evaluation for `names` (example: `tar_load_raw(names = quote(everything()))`).

Usage

```
tar_load(
  names,
  branches = NULL,
  meta = targets::tar_meta(targets_only = TRUE, store = store),
  strict = TRUE,
  silent = FALSE,
  envir = parent.frame(),
  store = targets::tar_config_get("store")
)

tar_load_raw(
  names,
  branches = NULL,
  meta = tar_meta(store = store),
  strict = TRUE,
  silent = FALSE,
  envir = parent.frame(),
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Names of the targets to load. `tar_load()` uses non-standard evaluation in the `names` argument (example: `tar_load(names = everything())`), whereas `tar_load_raw()` uses standard evaluation for `names` (example: `tar_load_raw(names = quote(everything()))`). The object supplied to `names` should be a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`branches`	Integer of indices of the branches to load for any targets that are patterns.
`meta`	Data frame of target metadata from `tar_meta()`.
`strict`	Logical of length 1, whether to error out if one of the selected targets is in the metadata but cannot be loaded. Set to `FALSE` to just load the targets in the metadata that can be loaded and skip the others.
`silent`	Logical of length 1. Only relevant when `strict` is `FALSE`. If `silent` is `FALSE` and `strict` is `FALSE`, then a message will be printed if a target is in the metadata but cannot be loaded. However, load failures will not stop other targets from being loaded.
`envir`	R environment in which to load target return values.
`store`	Character of length 1, directory path to the data store of the pipeline.

Value

Nothing.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

Cloud target data versioning

Some buckets in Amazon S3 or Google Cloud Storage are "versioned", which means they track historical versions of each data object. If you use `targets` with cloud storage (`https://books.ropensci.org/targets/cloud-storage.html`) and versioning is turned on, then `targets` will record each version of each target in its metadata.

Functions like `tar_read()` and `tar_load()` load the version recorded in the local metadata, which may not be the same as the "current" version of the object in the bucket. Likewise, functions `tar_delete()` and `tar_destroy()` only remove the version ID of each target as recorded in the local metadata.

If you want to interact with the *latest* version of an object instead of the version ID recorded in the local metadata, then you will need to delete the object from the metadata.

1. Make sure your local copy of the metadata is current and up to date. You may need to run `tar_meta_download()` or `tar_meta_sync()` first.
2. Run `tar_unversion()` to remove the recorded version IDs of your targets in the local metadata.
3. With the version IDs gone from the local metadata, functions like `tar_read()` and `tar_destroy()` will use the *latest* version of each target data object.
4. Optional: to back up the local metadata file with the version IDs deleted, use `tar_meta_upload()`.

See Also

Other storage: `tar_format()`, `tar_load_everything()`, `tar_objects()`, `tar_read()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
ls() # Does not have "y1", "y2", or "z".
tar_load(starts_with("y"))
ls() # Has "y1" and "y2" but not "z".
tar_load_raw(quote(any_of("z")))
ls() # Has "y1", "y2", and "z".
})
}
```

---

tar_load_everything *Load the values of all available targets.*

Description

Shorthand for `tar_load(everything())` to load all targets with entries in the metadata.

Usage

```
tar_load_everything(
  branches = NULL,
  meta = tar_meta(targets_only = TRUE, store = store),
  strict = TRUE,
  silent = FALSE,
  envir = parent.frame(),
  store = targets::tar_config_get("store")
)
```

Arguments

`branches`	Integer of indices of the branches to load for any targets that are patterns.
`meta`	Data frame of target metadata from `tar_meta()`.
`strict`	Logical of length 1, whether to error out if one of the selected targets is in the metadata but cannot be loaded. Set to `FALSE` to just load the targets in the metadata that can be loaded and skip the others.
`silent`	Logical of length 1. Only relevant when `strict` is `FALSE`. If `silent` is `FALSE` and `strict` is `FALSE`, then a message will be printed if a target is in the metadata but cannot be loaded. However, load failures will not stop other targets from being loaded.
`envir`	R environment in which to load target return values.
`store`	Character of length 1, directory path to the data store of the pipeline.

Value

Nothing.

See Also

Other storage: `tar_format()`, `tar_load()`, `tar_objects()`, `tar_read()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
ls() # Does not have "y1", "y2", or "z".
tar_load_everything()
ls() # Has "y1", "y2", and "z".
})
}
```

`tar_load_globals` *Load globals for debugging, testing, and prototyping*

Description

Load user-defined packages, functions, global objects, and settings defined in the target script file (default: `_targets.R`). This function is for debugging, testing, and prototyping only. It is not recommended for use inside a serious pipeline or to report the results of a serious pipeline.

Usage

```
tar_load_globals(
  envir = parent.frame(),
  script = targets::tar_config_get("script")
)
```

Arguments

`envir`	Environment to source the target script (default: `_targets.R`). Defaults to the calling environment.
`script`	Character of length 1, path to the target script file that defines the pipeline (`_targets.R` by default). This path should be either an absolute path or a path relative to the project root where you will call `tar_make()` and other functions. When `tar_make()` and friends run the script from the current working directory. If the argument `NULL`, the setting is not modified. Use `tar_config_unset()` to delete a setting.

Details

This function first sources the target script file (default: `_targets.R`) to loads all user-defined functions, global objects, and settings into the current R process. Then, it loads all the packages defined in `tar_option_get("packages")` (default: `(.packages())`) using `library()` with `lib.loc` defined in `tar_option_get("library")` (default: `NULL`).

Value

`NULL` (invisibly).

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other debug: `tar_traceback()`, `tar_workspace()`, `tar_workspaces()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(packages = "callr")
  analyze_data <- function(data) {
    summary(data)
  }
  list(
    tar_target(x, 1 + 1),
    tar_target(y, 1 + 1)
  )
}, ask = FALSE)
tar_load_globals()
print(analyze_data)
print("callr" %in% (.packages()))
})
}
```

`tar_make` *Run a pipeline of targets.*

Description

Run the pipeline you defined in the targets script file (default: `_targets.R`). `tar_make()` runs the correct targets in the correct order and stores the return values in `_targets/objects/`. Use `tar_read()` to read a target back into R, and see `https://docs.ropensci.org/targets/reference/index.html#clean` to manage output files.

Usage

```
tar_make(
  names = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  reporter = targets::tar_config_get("reporter_make"),
  seconds_meta_append = targets::tar_config_get("seconds_meta_append"),
  seconds_meta_upload = targets::tar_config_get("seconds_meta_upload"),
  seconds_reporter = targets::tar_config_get("seconds_reporter"),
  seconds_interval = targets::tar_config_get("seconds_interval"),
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store"),
  garbage_collection = NULL,
  use_crew = targets::tar_config_get("use_crew"),
  terminate_controller = TRUE,
  as_job = targets::tar_config_get("as_job")
)
```

Arguments

`names`	Names of the targets to run or check. Set to `NULL` to check/run all the targets (default). The object supplied to `names` should be a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`shortcut`	Logical of length 1, how to interpret the `names` argument. If `shortcut` is `FALSE` (default) then the function checks all targets upstream of `names` as far back as the dependency graph goes. `shortcut = TRUE` increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. It relies on stored metadata for information about upstream dependencies. `shortcut = TRUE` only works if you set `names`.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to `tar_config_get("reporter_make")`. Choices:

- `"silent"`: print nothing.
- `"summary"`: print a running total of the number of each targets in each status category (queued, dispatched, skipped, completed, canceled, or errored). Also show a timestamp (`"%H:%M %OS2"` `strptime()` format) of the last time the progress changed and printed to the screen.
- `"timestamp"`: same as the `"verbose"` reporter except that each .message begins with a time stamp.
- `"timestamp_positives"`: same as the `"timestamp"` reporter except without messages for skipped targets.
- `"verbose"`: print messages for individual targets as they start, finish, or are skipped. Each individual target-specific time (e.g. "3.487 seconds") is strictly the elapsed runtime of the target and does not include steps like data retrieval and output storage.
- `"verbose_positives"`: same as the `"verbose"` reporter except without messages for skipped targets.

`seconds_meta_append`

Positive numeric of length 1 with the minimum number of seconds between saves to the local metadata and progress files in the data store. his is an aggressive optimization setting not recommended for most users: higher values generally make the pipeline run faster, but unsaved work (in the event of a crash) is not up to date. When the pipeline ends, all the metadata and progress data is saved immediately, regardless of `seconds_meta_append`.

When the pipeline is just skipping targets, the actual interval between saves is `max(1, seconds_meta_append)` to reduce overhead.

`seconds_meta_upload`

Positive numeric of length 1 with the minimum number of seconds between uploads of the metadata and progress data to the cloud (see `https://books.ropensci.org/targets/cloud-storage.html`). Higher values generally make the pipeline run faster, but unsaved work (in the event of a crash) may not be backed up to the cloud. When the pipeline ends, all the metadata and progress data is uploaded immediately, regardless of `seconds_meta_upload`.

`seconds_reporter`

Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is `max(1, seconds_reporter)` to reduce overhead.

`seconds_interval`

Deprecated on 2023-08-24 (targets version 1.2.2.9001). Use `seconds_meta_append`, `seconds_meta_upload`, and `seconds_reporter` instead.

`callr_function` A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.

`callr_arguments`
: A list of arguments to `callr_function`.

`envir`
: An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc.

  The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

`script`
: Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

`store`
: Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

`garbage_collection`
: Deprecated. Use the `garbage_collection` argument of `tar_option_set()` instead to run garbage collection at regular intervals in a pipeline, or use the argument of the same name in `tar_target()` to activate garbage collection for a specific target.

`use_crew`
: Logical of length 1, whether to use `crew` if the `controller` option is set in `tar_option_set()` in the target script (`_targets.R`). See `https://books.ropensci.org/targets/crew.html` for details.

`terminate_controller`
: Logical of length 1. For a `crew`-integrated pipeline, whether to terminate the controller after stopping or finishing the pipeline. This should almost always be set to `TRUE`, but `FALSE` combined with `callr_function = NULL` will allow you to get the running controller using `tar_option_get("controller")` for debugging purposes. For example, `tar_option_get("controller")$summary()` produces a worker-by-worker summary of the work assigned and completed, `tar_option_get("controller")$queue` is the list of unresolved tasks, and `tar_option_get("controller")$results` is the list of tasks that completed but were not collected with `pop()`. You can manually terminate the controller with `tar_option_get("controller")$summary()` to close down the dispatcher and worker processes.

`as_job`
: `TRUE` to run as an RStudio IDE / Posit Workbench job, if running on RStudio IDE / Posit Workbench. `FALSE` to run as a `callr` process in the main R session (depending on the `callr_function` argument). If `as_job` is `TRUE`, then the `rstudioapi` package must be installed.

Value

NULL except if callr_function = callr::r_bg(), in which case a handle to the callr background process is returned. Either way, the value is invisibly returned.

Storage access

Several functions like tar_make(), tar_read(), tar_load(), tar_meta(), and tar_progress() read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the tarchetypes package such as tar_render() and tar_quarto().

See Also

Other pipeline: tar_make_clustermq(), tar_make_future()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make(starts_with("y")) # Only processes y1 and y2.
# Distributed computing with crew:
if (requireNamespace("crew", quietly = TRUE)) {
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(controller = crew::controller_local())
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
}
})
}
```

tar_make_clustermq *Superseded. Run a pipeline with persistent* clustermq *workers.*

Description

Superseded. Use `tar_make()` with `crew`: `https://books.ropensci.org/targets/crew.html`.

Usage

```
tar_make_clustermq(
  names = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  reporter = targets::tar_config_get("reporter_make"),
  seconds_meta_append = targets::tar_config_get("seconds_meta_append"),
  seconds_meta_upload = targets::tar_config_get("seconds_meta_upload"),
  seconds_reporter = targets::tar_config_get("seconds_reporter"),
  seconds_interval = targets::tar_config_get("seconds_interval"),
  workers = targets::tar_config_get("workers"),
  log_worker = FALSE,
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store"),
  garbage_collection = NULL
)
```

Arguments

`names`	Names of the targets to run or check. Set to `NULL` to check/run all the targets (default). The object supplied to `names` should be a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`shortcut`	Logical of length 1, how to interpret the `names` argument. If `shortcut` is `FALSE` (default) then the function checks all targets upstream of `names` as far back as the dependency graph goes. `shortcut = TRUE` increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. It relies on stored metadata for information about upstream dependencies. `shortcut = TRUE` only works if you set `names`.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to `tar_config_get("reporter_make")`. Choices: • `"silent"`: print nothing. • `"summary"`: print a running total of the number of each targets in each status category (queued, dispatched, skipped, completed, canceled, or errored). Also show a timestamp (`"%H:%M %OS2"` `strptime()` format) of the last time the progress changed and printed to the screen.

- `"timestamp"`: same as the `"verbose"` reporter except that each .message begins with a time stamp.
- `"timestamp_positives"`: same as the `"timestamp"` reporter except without messages for skipped targets.
- `"verbose"`: print messages for individual targets as they start, finish, or are skipped. Each individual target-specific time (e.g. "3.487 seconds") is strictly the elapsed runtime of the target and does not include steps like data retrieval and output storage.
- `"verbose_positives"`: same as the `"verbose"` reporter except without messages for skipped targets.

`seconds_meta_append`
: Positive numeric of length 1 with the minimum number of seconds between saves to the local metadata and progress files in the data store. his is an aggressive optimization setting not recommended for most users: higher values generally make the pipeline run faster, but unsaved work (in the event of a crash) is not up to date. When the pipeline ends, all the metadata and progress data is saved immediately, regardless of `seconds_meta_append`.

  When the pipeline is just skipping targets, the actual interval between saves is `max(1, seconds_meta_append)` to reduce overhead.

`seconds_meta_upload`
: Positive numeric of length 1 with the minimum number of seconds between uploads of the metadata and progress data to the cloud (see `https://books.ropensci.org/targets/cloud-storage.html`). Higher values generally make the pipeline run faster, but unsaved work (in the event of a crash) may not be backed up to the cloud. When the pipeline ends, all the metadata and progress data is uploaded immediately, regardless of `seconds_meta_upload`.

`seconds_reporter`
: Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is `max(1, seconds_reporter)` to reduce overhead.

`seconds_interval`
: Deprecated on 2023-08-24 (targets version 1.2.2.9001). Use `seconds_meta_append`, `seconds_meta_upload`, and `seconds_reporter` instead.

`workers`
: Positive integer, number of persistent `clustermq` workers to create.

`log_worker`
: Logical, whether to write a log file for each worker. Same as the `log_worker` argument of `clustermq::Q()` and `clustermq::workers()`.

`callr_function`
: A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.

callr_arguments
: A list of arguments to callr_function.

envir
: An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

  The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with tar_option_set() from within the target script file. In other words, if you call tar_option_set(envir = envir1) in an interactive session and then tar_make(envir = envir2, callr_function = NULL), then envir2 will be used.

script
: Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.

store
: Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

garbage_collection
: Deprecated. Use the garbage_collection argument of tar_option_set() instead to run garbage collection at regular intervals in a pipeline, or use the argument of the same name in tar_target() to activate garbage collection for a specific target.

Details

tar_make_clustermq() is like tar_make() except that targets run in parallel on persistent workers. A persistent worker is an R process that runs for a long time and runs multiple targets during its lifecycle. Persistent workers launch as soon as the pipeline reaches an outdated target with deployment = "worker", and they keep running until the pipeline starts to wind down.

To configure tar_make_clustermq(), you must configure the clustermq package. To do this, set global options clustermq.scheduler and clustermq.template inside the target script file (default: _targets.R). To read more about configuring clustermq for your scheduler, visit https://mschubert.github.io/clustermq/articles/userguide.html#configuration # nolint or https://books.ropensci.org/targets/hpc.html. clustermq is not a strict dependency of targets, so you must install clustermq yourself.

Value

NULL except if callr_function = callr::r_bg(), in which case a handle to the callr background process is returned. Either way, the value is invisibly returned.

Storage access

Several functions like tar_make(), tar_read(), tar_load(), tar_meta(), and tar_progress() read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the tarchetypes package such as tar_render() and tar_quarto().

See Also

Other pipeline: tar_make(), tar_make_future()

Examples

```
if (!identical(tolower(Sys.info()[["sysname"]]), "windows")) {
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  options(clustermq.scheduler = "multiprocess") # Does not work on Windows.
  tar_option_set()
  list(tar_target(x, 1 + 1))
}, ask = FALSE)
tar_make_clustermq()
})
}
}
```

tar_make_future *Superseded. Run a pipeline of targets in parallel with transient* future *workers.*

Description

Superseded. Use tar_make() with crew: https://books.ropensci.org/targets/crew.html.

Usage

```
tar_make_future(
  names = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  reporter = targets::tar_config_get("reporter_make"),
  seconds_meta_append = targets::tar_config_get("seconds_meta_append"),
  seconds_meta_upload = targets::tar_config_get("seconds_meta_upload"),
  seconds_reporter = targets::tar_config_get("seconds_reporter"),
  seconds_interval = targets::tar_config_get("seconds_interval"),
  workers = targets::tar_config_get("workers"),
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store"),
  garbage_collection = NULL
)
```

Arguments

`names`	Names of the targets to run or check. Set to `NULL` to check/run all the targets (default). The object supplied to `names` should be a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`shortcut`	Logical of length 1, how to interpret the `names` argument. If `shortcut` is `FALSE` (default) then the function checks all targets upstream of `names` as far back as the dependency graph goes. `shortcut = TRUE` increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. It relies on stored metadata for information about upstream dependencies. `shortcut = TRUE` only works if you set `names`.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to `tar_config_get("reporter_make")`. Choices: • `"silent"`: print nothing. • `"summary"`: print a running total of the number of each targets in each status category (queued, dispatched, skipped, completed, canceled, or errored). Also show a timestamp (`"%H:%M %OS2"` `strptime()` format) of the last time the progress changed and printed to the screen. • `"timestamp"`: same as the `"verbose"` reporter except that each .message begins with a time stamp. • `"timestamp_positives"`: same as the `"timestamp"` reporter except without messages for skipped targets. • `"verbose"`: print messages for individual targets as they start, finish, or are skipped. Each individual target-specific time (e.g. "3.487 seconds") is strictly the elapsed runtime of the target and does not include steps like data retrieval and output storage. • `"verbose_positives"`: same as the `"verbose"` reporter except without messages for skipped targets.
`seconds_meta_append`	Positive numeric of length 1 with the minimum number of seconds between saves to the local metadata and progress files in the data store. his is an aggressive optimization setting not recommended for most users: higher values generally make the pipeline run faster, but unsaved work (in the event of a crash) is not up to date. When the pipeline ends, all the metadata and progress data is saved immediately, regardless of `seconds_meta_append`. When the pipeline is just skipping targets, the actual interval between saves is `max(1, seconds_meta_append)` to reduce overhead.

`seconds_meta_upload`
: Positive numeric of length 1 with the minimum number of seconds between uploads of the metadata and progress data to the cloud (see https://books.ropensci.org/targets/cloud-storage.html). Higher values generally make the pipeline run faster, but unsaved work (in the event of a crash) may not be backed up to the cloud. When the pipeline ends, all the metadata and progress data is uploaded immediately, regardless of `seconds_meta_upload`.

`seconds_reporter`
: Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is `max(1, seconds_reporter)` to reduce overhead.

`seconds_interval`
: Deprecated on 2023-08-24 (targets version 1.2.2.9001). Use `seconds_meta_append`, `seconds_meta_upload`, and `seconds_reporter` instead.

`workers`
: Positive integer, maximum number of transient `future` workers allowed to run at any given time.

`callr_function`
: A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.

`callr_arguments`
: A list of arguments to `callr_function`.

`envir`
: An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc.

  The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

`script`
: Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

`store`
: Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function

call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

garbage_collection
: Deprecated. Use the garbage_collection argument of tar_option_set() instead to run garbage collection at regular intervals in a pipeline, or use the argument of the same name in tar_target() to activate garbage collection for a specific target.

Details

This function is like tar_make() except that targets run in parallel with transient future workers. It requires that you declare your future::plan() inside the target script file (default: _targets.R). future is not a strict dependency of targets, so you must install future yourself.

To configure tar_make_future() with a computing cluster, see the future.batchtools package documentation.

Value

NULL except if callr_function = callr::r_bg(), in which case a handle to the callr background process is returned. Either way, the value is invisibly returned.

Storage access

Several functions like tar_make(), tar_read(), tar_load(), tar_meta(), and tar_progress() read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the tarchetypes package such as tar_render() and tar_quarto().

See Also

Other pipeline: tar_make(), tar_make_clustermq()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  future::plan(future::multisession, workers = 2)
  list(
    tar_target(x, 1 + 1),
    tar_target(y, 1 + 1)
  )
}, ask = FALSE)
tar_make_future()
})
}
```

tar_manifest *Produce a data frame of information about your targets.*

Description

Along with tar_visnetwork() and tar_glimpse(), tar_manifest() helps check that you constructed your pipeline correctly.

Usage

```
tar_manifest(
  names = NULL,
  fields = tidyselect::any_of(c("name", "command", "pattern", "description")),
  drop_missing = TRUE,
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script")
)
```

Arguments

names	Names of the targets to show. Set to NULL to show all the targets (default). Otherwise, the object supplied to names should be a tidyselect expression like any_of() or starts_with() from tidyselect itself, or tar_described_as() to select target names based on their descriptions.
fields	Names of the fields, or columns, to show. Set to NULL to show all the fields (default). Otherwise, the value of fields should be a tidyselect expression like starts_with() to select the columns to show in the output. Possible fields are below. All of them can be set in tar_target(), tar_target_raw(), or tar_option_set(). • name: Name of the target. • command: the R command that runs when the target runs. • description: custom free-form text description of the target, if available. • pattern: branching pattern of the target, if applicable. • format: Storage format. • repository: Storage repository. • iteration: Iteration mode for branching. • error: Error mode, what to do when the target fails. • memory: Memory mode, when to keep targets in memory. • storage: Storage mode for high-performance computing scenarios. • retrieval: Retrieval mode for high-performance computing scenarios. • deployment: Where/whether to deploy the target in high-performance computing scenarios.

- `priority`: Numeric of length 1 between 0 and 1. Controls which targets get deployed first when multiple competing targets are ready simultaneously. Targets with priorities closer to 1 get dispatched earlier (and polled earlier in `tar_make_future()`).
- `resources`: A list of target-specific resource requirements for `tar_make_future()`.
- `cue_mode`: Cue mode from `tar_cue()`.
- `cue_depend`: Depend cue from `tar_cue()`.
- `cue_expr`: Command cue from `tar_cue()`.
- `cue_file`: File cue from `tar_cue()`.
- `cue_format`: Format cue from `tar_cue()`.
- `cue_repository`: Repository cue from `tar_cue()`.
- `cue_iteration`: Iteration cue from `tar_cue()`.
- `packages`: List columns of packages loaded before running the target.
- `library`: List column of library paths to load the packages.

`drop_missing` Logical of length 1, whether to automatically omit empty columns and columns with all missing values.

`callr_function` A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.

`callr_arguments`
A list of arguments to `callr_function`.

`envir` An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc.

The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

`script` Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

Value

A data frame of information about the targets in the pipeline. Rows appear in topological order (the order they will run without any influence from parallel computing or priorities).

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other inspect: `tar_deps()`, `tar_network()`, `tar_outdated()`, `tar_sitrep()`, `tar_validate()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2),
    tar_target(m, z, pattern = map(z), description = "branching over z"),
    tar_target(c, z, pattern = cross(z))
  )
}, ask = FALSE)
tar_manifest()
tar_manifest(fields = any_of(c("name", "command")))
tar_manifest(fields = any_of("command"))
tar_manifest(fields = starts_with("cue"))
})
}
```

`tar_mermaid` `mermaid.js` *dependency graph.*

Description

Visualize the dependency graph with a static `mermaid.js` graph.

Usage

```
tar_mermaid(
  targets_only = FALSE,
  names = NULL,
  shortcut = FALSE,
  allow = NULL,
  exclude = ".Random.seed",
  outdated = TRUE,
  label = targets::tar_config_get("label"),
  label_width = targets::tar_config_get("label_width"),
  legend = TRUE,
  color = TRUE,
  reporter = targets::tar_config_get("reporter_outdated"),
  seconds_reporter = targets::tar_config_get("seconds_reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include global functions and objects.
`names`	Names of targets. The graph visualization will operate only on these targets (and unless `shortcut` is `TRUE`, all the targets upstream as well). Selecting a small subgraph using `names` could speed up the load time of the visualization. Unlike `allow`, `names` is invoked before the graph is generated. Set to NULL to check/run all the targets (default). Otherwise, the object supplied to `names` should be a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`shortcut`	Logical of length 1, how to interpret the `names` argument. If `shortcut` is `FALSE` (default) then the function checks all targets upstream of `names` as far back as the dependency graph goes. If `TRUE`, then the function only checks the targets in `names` and uses stored metadata for information about upstream dependencies as needed. `shortcut = TRUE` increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, `shortcut = TRUE` only works if you set `names`.
`allow`	Optional, define the set of allowable vertices in the graph. Unlike `names`, `allow` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or `tidyselect` helpers like `starts_with()`.
`exclude`	Optional, define the set of exclude vertices from the graph. Unlike `names`, `exclude` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to exclude no vertices. Otherwise, you can supply symbols or `tidyselect` helpers like `any_of()` and `starts_with()`.
`outdated`	Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and setting `outdated` to `FALSE` is a nice way to speed up the graph if you only want to see dependency relationships and pipeline progress.

`label`	Character vector of one or more aesthetics to add to the vertex labels. Can contain `"description"` to show each target's custom description, `"time"` to show total runtime, `"size"` to show total storage size, or `"branches"` to show the number of branches in each pattern. You can choose multiple aesthetics at once, e.g. `label = c("description", "time")`. Only the description is enabled by default.
`label_width`	Positive numeric of length 1, maximum width (in number of characters) of the node labels.
`legend`	Logical of length 1, whether to display the legend.
`color`	Logical of length 1, whether to color the graph vertices by status.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: * `"forecast_interactive"` (default): use the forecast reporter if the session is interactive (see `base::interactive()`), otherwise use the silent reporter. * `"silent"`: print nothing. * `"forecast"`: print running totals of the checked and outdated targets found so far.
`seconds_reporter`	Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is `max(1, seconds_reporter)` to reduce overhead.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.
`envir`	An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

`store` Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

`mermaid.js` is a JavaScript library for constructing static visualizations of graphs.

Value

A character vector of lines of code of the `mermaid.js` graph. You can visualize the graph by copying the text into a public online `mermaid.js` editor or a `mermaid` GitHub code chunk (`https://github.blog/2022-02-14-incl`
Alternatively, you can render it inline in an R Markdown or Quarto document using a `results = "asis"` code chunk like so:

~~~
```{r, results = "asis", echo = FALSE}
cat(c("```{mermaid}", targets::tar_mermaid(), "```"), sep = "\n")
```
~~~

Dependency graph

The dependency graph of a pipeline is a directed acyclic graph (DAG) where each node indicates a target or global object and each directed edge indicates where a downstream node depends on an upstream node. The DAG is not always a tree, but it never contains a cycle because no target is allowed to directly or indirectly depend on itself. The dependency graph should show a natural progression of work from left to right. `targets` uses static code analysis to create the graph, so the order of `tar_target()` calls in the `_targets.R` file does not matter. However, targets does not support self-referential loops or other cycles. For more information on the dependency graph, please read `https://books.ropensci.org/targets/targets.html#dependencies`.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other visualize: `tar_glimpse()`, `tar_visnetwork()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
```

```
  library(targets)
  library(tarchetypes)
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2, description = "sum of two other sums")
  )
})
# Copy the text into a mermaid.js online editor
# or a mermaid GitHub code chunk:
tar_mermaid()
})
}
```

tar_meta *Read a project's metadata.*

Description

Read the metadata of all recorded targets and global objects.

Usage

```
tar_meta(
  names = NULL,
  fields = NULL,
  targets_only = FALSE,
  complete_only = FALSE,
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Optional, names of the targets. If supplied, `tar_meta()` only returns metadata on these targets. You can supply symbols or `tidyselect` helpers like `any_of()` and `starts_with()`. If `NULL`, all names are selected.
`fields`	Optional, names of columns/fields to select. If supplied, `tar_meta()` only returns the selected metadata columns. If `NULL`, all fields are selected. You can supply symbols or `tidyselect` helpers like `any_of()` and `starts_with()`. The `name` column is always included first no matter what you select. Choices: • `name`: name of the target or global object. • `type`: type of the object: either `"function"` or `"object"` for global objects, and `"stem"`, `"branch"`, `"map"`, or `"cross"` for targets. • `data`: hash of the output data. • `command`: hash of the target's deparsed command. • `depend`: hash of the immediate upstream dependencies of the target.

- `seed`: random number generator seed with which the target ran. A target's random number generator seed is a deterministic function of its name. In this way, each target runs with a reproducible seed so someone else running the same pipeline should get the same results, and no two targets in the same pipeline share the same seed. (Even dynamic branches have different names and thus different seeds.) You can recover the seed of a completed target with `tar_meta(your_target, seed)` and run `tar_seed_set()` on the result to locally recreate the target's initial RNG state.
- `path`: A list column of paths to target data. Usually, each element is a single path, but there could be multiple paths per target for dynamic files (i.e. `tar_target(format = "file")`).
- `time`: `POSIXct` object with the time the target's data in storage was last modified. If the target stores no local file, then the time stamp corresponds to the time the target last ran successfully. Only targets that run commands have time stamps: just non-branching targets and individual dynamic branches. Displayed in the current time zone of the system. If there are multiple outputs for that target, as with file targets, then the maximum time is shown.
- `size`: hash of the sum of all the bytes of the files at `path`.
- `bytes`: total file size in bytes of all files in `path`.
- `format`: character, one of the admissible data storage formats. See the `format` argument in the `tar_target()` help file for details.
- `iteration`: character, either `"list"` or `"vector"` to describe the iteration and aggregation mode of the target. See the `iteration` argument in the `tar_target()` help file for details.
- `parent`: for branches, name of the parent pattern.
- `children`: list column, names of the children of targets that have them. These include buds of stems and branches of patterns.
- `seconds`: number of seconds it took to run the target.
- `warnings`: character string of warning messages from the last run of the target. Only the first 50 warnings are available, and only the first 2048 characters of the concatenated warning messages.
- `error`: character string of the error message if the target errored.

`targets_only`	Logical, whether to just show information about targets or also return metadata on functions and other global objects.
`complete_only`	Logical, whether to return only complete rows (no `NA` values).
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

A metadata row only updates when the target completes. `tar_progress()` shows information on targets that are running. That is why the number of branches may disagree between `tar_meta()` and `tar_progress()` for actively running pipelines.

Value

A data frame with one row per target/object and the selected fields.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

Cloud metadata

Metadata files help `targets` read data objects and decide if the pipeline is up to date. Usually, these metadata files live in files in the local `_targets/meta/` folder in your project, e.g. `_targets/meta/meta`. But in addition, if you set `repository` to anything other than `"local"` in `tar_option_set()` in `_targets.R`, then `tar_make()` continuously uploads the metadata files to the bucket you specify in `resources`. `tar_meta_delete()` will delete those files from the cloud, and so will `tar_destroy()` if `destroy` is set to either `"all"` or `"cloud"`.

Other functions in `targets`, such as `tar_meta()`, `tar_visnetwork()`, `tar_outdated()`, and `tar_invalidate()`, use the local metadata only and ignore the copies on the cloud. So if you are working on a different computer than the one running the pipeline, you will need to download the cloud metadata to your current machine using `tar_meta_download()`. Other functions `tar_meta_upload()`, `tar_meta_sync()`, and `tar_meta_delete()` also manage metadata across the cloud and the local file system.

Remarks:

- The `repository_meta` option in `tar_option_set()` is actually what controls where the metadata lives in the cloud, but it defaults to `repository`.
- Like `tar_make()`, `tar_make_future()` and `tar_make_clustermq()` also continuously upload metadata files to the cloud bucket specified in `resources`.
- `tar_meta_download()` and related functions need to run `_targets.R` to detect `tar_option_set()` options `repository_meta` and `resources`, so please be aware of side effects that may happen running your custom `_targets.R` file.

See Also

Other metadata: `tar_meta_delete()`, `tar_meta_download()`, `tar_meta_sync()`, `tar_meta_upload()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_meta()
tar_meta(starts_with("y_")) # see also any_of()
})
}
```

`tar_meta_delete` *Delete metadata.*

Description

Delete the project metadata files from the local file system, the cloud, or both.

Usage

```
tar_meta_delete(
  meta = TRUE,
  progress = TRUE,
  process = TRUE,
  crew = TRUE,
  verbose = TRUE,
  delete = "all",
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`meta`	Logical of length 1, whether to process the main metadata file at `_targets/meta/meta`.
`progress`	Logical of length 1, whether to process the progress file at `_targets/meta/progress`.
`process`	Logical of length 1, whether to process the process file at `_targets/meta/process`.
`crew`	Logical of length 1, whether to process the `crew` file at `_targets/meta/crew`. Only exists if running `targets` with `crew`.
`verbose`	Logical of length 1, whether to print informative console messages.
`delete`	Character of length 1, which location to delete the files. Choose `"local"` for local files, `"cloud"` for files on the cloud, or `"all"` to delete metadata files from both the local file system and the cloud.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

store	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

See Also

Other metadata: `tar_meta()`, `tar_meta_download()`, `tar_meta_sync()`, `tar_meta_upload()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(
    resources = tar_resources(
      aws = tar_resources_aws(
        bucket = "YOUR_BUCKET_NAME",
        prefix = "YOUR_PROJECT_NAME"
      )
    ),
    repository = "aws"
  )
  list(
    tar_target(x, data.frame(x = seq_len(2), y = seq_len(2)))
  )
})
tar_make()
tar_meta_delete()
})
}
```

---

tar_meta_download *download local metadata to the cloud.*

Description

download local metadata files to the cloud location (repository, bucket, and prefix) you set in `tar_option_set()` in `_targets.R`.

Usage

```
tar_meta_download(
  meta = TRUE,
  progress = TRUE,
  process = TRUE,
  crew = TRUE,
  verbose = TRUE,
  strict = FALSE,
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`meta`	Logical of length 1, whether to process the main metadata file at `_targets/meta/meta`.
`progress`	Logical of length 1, whether to process the progress file at `_targets/meta/progress`.
`process`	Logical of length 1, whether to process the process file at `_targets/meta/process`.
`crew`	Logical of length 1, whether to process the `crew` file at `_targets/meta/crew`. Only exists if running `targets` with `crew`.
`verbose`	Logical of length 1, whether to print informative console messages.
`strict`	Logical of length 1. `TRUE` to error out if the file does not exist in the bucket, `FALSE` to proceed without an error or warning. If `strict` is `FALSE` and `verbose` is `TRUE`, then an informative message will print to the R console.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

See Also

Other metadata: `tar_meta()`, `tar_meta_delete()`, `tar_meta_sync()`, `tar_meta_upload()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(
    resources = tar_resources(
      aws = tar_resources_aws(
        bucket = "YOUR_BUCKET_NAME",
        prefix = "YOUR_PROJECT_NAME"
      )
    ),
    repository = "aws"
  )
  list(
    tar_target(x, data.frame(x = seq_len(2), y = seq_len(2)))
  )
})
tar_make()
tar_meta_download()
})
}
```

---

tar_meta_sync — *Synchronize cloud metadata.*

Description

Synchronize metadata in a cloud bucket with metadata in the local data store.

Usage

```
tar_meta_sync(
  meta = TRUE,
  progress = TRUE,
  process = TRUE,
  crew = TRUE,
  verbose = TRUE,
  prefer_local = TRUE,
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`meta`	Logical of length 1, whether to process the main metadata file at `_targets/meta/meta`.
`progress`	Logical of length 1, whether to process the progress file at `_targets/meta/progress`.
`process`	Logical of length 1, whether to process the process file at `_targets/meta/process`.
`crew`	Logical of length 1, whether to process the `crew` file at `_targets/meta/crew`. Only exists if running `targets` with `crew`.
`verbose`	Logical of length 1, whether to print informative console messages.
`prefer_local`	Logical of length 1 to control which copy of each metadata file takes precedence if the local hash and cloud hash are different but the time stamps are the same. Set to `TRUE` to upload the local data file in that scenario, `FALSE` to download the cloud file.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

store	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

`tar_meta_sync()` synchronizes the local and cloud copies of all the metadata files of the pipeline so that both have the most recent copy. For each metadata file, if the local file does not exist or is older than the cloud file, then the cloud file is downloaded to the local file path. Conversely, if the cloud file is older or does not exist, then the local file is uploaded to the cloud. If the time stamps of these files are equal, use the `prefer_local` argument to determine which copy takes precedence.

See Also

Other metadata: `tar_meta()`, `tar_meta_delete()`, `tar_meta_download()`, `tar_meta_upload()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(
    resources = tar_resources(
      aws = tar_resources_aws(
        bucket = "YOUR_BUCKET_NAME",
        prefix = "YOUR_PROJECT_NAME"
      )
    ),
    repository = "aws"
  )
  list(
    tar_target(x, data.frame(x = seq_len(2), y = seq_len(2)))
  )
}, ask = FALSE)
tar_make()
tar_meta_sync()
})
}
```

`tar_meta_upload` *Upload local metadata to the cloud.*

Description

Upload local metadata files to the cloud location (repository, bucket, and prefix) you set in `tar_option_set()` in `_targets.R`.

Usage

```
tar_meta_upload(
  meta = TRUE,
  progress = TRUE,
  process = TRUE,
  crew = TRUE,
  verbose = TRUE,
  strict = FALSE,
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`meta`	Logical of length 1, whether to process the main metadata file at `_targets/meta/meta`.
`progress`	Logical of length 1, whether to process the progress file at `_targets/meta/progress`.
`process`	Logical of length 1, whether to process the process file at `_targets/meta/process`.
`crew`	Logical of length 1, whether to process the `crew` file at `_targets/meta/crew`. Only exists if running `targets` with `crew`.
`verbose`	Logical of length 1, whether to print informative console messages.
`strict`	Logical of length 1. `TRUE` to error out if the file does not exist locally, `FALSE` to proceed without an error or warning. If `strict` is `FALSE` and `verbose` is `TRUE`, then an informative message will print to the R console.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

See Also

Other metadata: `tar_meta()`, `tar_meta_delete()`, `tar_meta_download()`, `tar_meta_sync()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(
    resources = tar_resources(
      aws = tar_resources_aws(
        bucket = "YOUR_BUCKET_NAME",
        prefix = "YOUR_PROJECT_NAME"
      )
    ),
    repository = "aws"
  )
  list(
    tar_target(x, data.frame(x = seq_len(2), y = seq_len(2)))
  )
}, ask = FALSE)
tar_make()
tar_meta_upload()
})
}
```

tar_name *Get the name of the target currently running.*

Description

Get the name of the target currently running.

Usage

```
tar_name(default = "target")
```

Arguments

`default`	Character, value to return if `tar_name()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.

Value

Character of length 1. If called inside a pipeline, `tar_name()` returns name of the target currently running. Otherwise, the return value is `default`.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
tar_name()
tar_name(default = "custom_target_name")
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(tar_target(x, tar_name()), ask = FALSE)
tar_make()
tar_read(x)
})
}
```

`tar_network` *Return the vertices and edges of a pipeline dependency graph.*

Description

Analyze the pipeline defined in the target script file (default: `_targets.R`) and return the vertices and edges of the directed acyclic graph of dependency relationships.

Usage

```
tar_network(
  targets_only = FALSE,
  names = NULL,
  shortcut = FALSE,
  allow = NULL,
  exclude = NULL,
  outdated = TRUE,
  reporter = targets::tar_config_get("reporter_outdated"),
  seconds_reporter = targets::tar_config_get("seconds_reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include imported global functions and objects.
`names`	Names of targets. The graph visualization will operate only on these targets (and unless `shortcut` is `TRUE`, all the targets upstream as well). Selecting a small subgraph using `names` could speed up the load time of the visualization. Unlike `allow`, `names` is invoked before the graph is generated. Set to NULL to check/run all the targets (default). Otherwise, the object supplied to `names` should be a `tidyselect` expression like `any_of()` or `starts_with()` from

	`tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`shortcut`	Logical of length 1, how to interpret the `names` argument. If `shortcut` is `FALSE` (default) then the function checks all targets upstream of `names` as far back as the dependency graph goes. If `TRUE`, then the function only checks the targets in `names` and uses stored metadata for information about upstream dependencies as needed. `shortcut = TRUE` increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, `shortcut = TRUE` only works if you set `names`.
`allow`	Optional, define the set of allowable vertices in the graph. Unlike `names`, `allow` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or `tidyselect` helpers like `starts_with()`.
`exclude`	Optional, define the set of exclude vertices from the graph. Unlike `names`, `exclude` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to exclude no vertices. Otherwise, you can supply symbols or `tidyselect` helpers like `any_of()` and `starts_with()`.
`outdated`	Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and setting `outdated` to `FALSE` is a nice way to speed up the graph if you only want to see dependency relationships and pipeline progress.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: * `"forecast_interactive"` (default): use the forecast reporter if the session is interactive (see `base::interactive()`), otherwise use the silent reporter. * `"silent"`: print nothing. * `"forecast"`: print running totals of the checked and outdated targets found so far.
`seconds_reporter`	Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is `max(1, seconds_reporter)` to reduce overhead.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.
`envir`	An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc.

	The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A list with two data frames: `vertices` and `edges`. The vertices data frame has one row per target and columns with the the type of the target or object (stem, branch, map, cross, function, or object), each target's description, and each target's status (up to date, outdated, dispatched, completed, canceled, or errored), as well as metadata if available (seconds of runtime, bytes of storage, and number of dynamic branches). The edges data frame has one row for every edge and columns `to` and `from` to mark the starting and terminating vertices.

Dependency graph

The dependency graph of a pipeline is a directed acyclic graph (DAG) where each node indicates a target or global object and each directed edge indicates where a downstream node depends on an upstream node. The DAG is not always a tree, but it never contains a cycle because no target is allowed to directly or indirectly depend on itself. The dependency graph should show a natural progression of work from left to right. `targets` uses static code analysis to create the graph, so the order of `tar_target()` calls in the `_targets.R` file does not matter. However, targets does not support self-referential loops or other cycles. For more information on the dependency graph, please read `https://books.ropensci.org/targets/targets.html#dependencies`.

See Also

Other inspect: `tar_deps()`, `tar_manifest()`, `tar_outdated()`, `tar_sitrep()`, `tar_validate()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1, description = "y2 info"),
    tar_target(z, y1 + y2, description = "z info")
  )
}, ask = FALSE)
tar_network(targets_only = TRUE)
})
}
```

`tar_newer` *List new targets*

Description

List all the targets whose last successful run occurred after a certain point in time.

Usage

```
tar_newer(
  time,
  names = NULL,
  inclusive = FALSE,
  store = targets::tar_config_get("store")
)
```

Arguments

`time`	A `POSIXct` object of length 1, time threshold. Targets newer than this time stamp are returned. For example, if `time = Sys.time - as.difftime(1, units = "weeks")` then `tar_newer()` returns targets newer than one week ago.
`names`	Names of eligible targets. Targets excluded from `names` will not be returned even if they are newer than the given `time`. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`inclusive`	Logical of length 1, whether to include targets completed at exactly the `time` given.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

Only applies to targets with recorded time stamps: just non-branching targets and individual dynamic branches. As of targets version 0.6.0, these time stamps are available for these targets regardless of storage format. Earlier versions of targets do not record time stamps for remote storage such as format = "url" or repository = "aws" in tar_target().

Value

A character vector of names of old targets with recorded timestamp metadata.

See Also

Other time: tar_older(), tar_timestamp()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(tar_target(x, seq_len(2)))
}, ask = FALSE)
tar_make()
# targets newer than 1 week ago
tar_newer(Sys.time() - as.difftime(1, units = "weeks"))
# targets newer than 1 week from now
tar_newer(Sys.time() + as.difftime(1, units = "weeks"))
# Everything is still up to date.
tar_make()
# Invalidate all targets targets newer than 1 week ago
# so they run on the next tar_make().
invalidate_these <- tar_newer(Sys.time() - as.difftime(1, units = "weeks"))
tar_invalidate(any_of(invalidate_these))
tar_make()
})
}
```

---

tar_noninteractive *Run if Target Markdown interactive mode is not on.*

Description

In Target Markdown, run the enclosed code only if interactive mode is not activated. Otherwise, do not run the code.

Usage

```
tar_noninteractive(code)
```

Arguments

`code`	R code to run if Target Markdown interactive mode is not turned on.

Details

Visit <books.ropensci.org/targets/literate-programming.html> to learn about Target Markdown and interactive mode.

Value

If Target Markdown interactive mode is not turned on, the function returns the result of running the code. Otherwise, the function invisibly returns `NULL`.

See Also

Other Target Markdown: `tar_engine_knitr()`, `tar_interactive()`, `tar_toggle()`

Examples

```
tar_noninteractive(message("Not in interactive mode."))
```

`tar_objects` *List saved targets*

Description

List targets currently saved to `_targets/objects/` or the cloud. Does not include local files with `tar_target(..., format = "file", repository = "local")`.

Usage

```
tar_objects(
  names = NULL,
  cloud = TRUE,
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Names of targets to select. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`cloud`	Logical of length 1, whether to include cloud targets in the output (e.g. `tar_target(..., repository = "aws")`).
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

Character vector of targets saved to `_targets/objects/`.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other storage: `tar_format()`, `tar_load()`, `tar_load_everything()`, `tar_read()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(tar_target(x, "value"))
}, ask = FALSE)
tar_make()
tar_objects()
tar_objects(starts_with("x")) # see also any_of()
})
}
```

`tar_older` *List old targets*

Description

List all the targets whose last successful run occurred before a certain point in time. Combine with `tar_invalidate()`, you can use `tar_older()` to automatically rerun targets at regular intervals. See the examples for a demonstration.

Usage

```
tar_older(
  time,
  names = NULL,
  inclusive = FALSE,
  store = targets::tar_config_get("store")
)
```

Arguments

time	A POSIXct object of length 1, time threshold. Targets older than this time stamp are returned. For example, if time = Sys.time() - as.difftime(1, units = "weeks") then tar_older() returns targets older than one week ago.
names	Names of eligible targets. Targets excluded from names will not be returned even if they are old. The object supplied to names should be NULL or a tidyselect expression like any_of() or starts_with() from tidyselect itself, or tar_described_as() to select target names based on their descriptions.
inclusive	Logical of length 1, whether to include targets completed at exactly the time given.
store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Details

Only applies to targets with recorded time stamps: just non-branching targets and individual dynamic branches. As of targets version 0.6.0, these time stamps are available for these targets regardless of storage format. Earlier versions of targets do not record time stamps for remote storage such as format = "url" or repository = "aws" in tar_target().

Value

A character vector of names of old targets with recorded timestamp metadata.

See Also

Other time: tar_newer(), tar_timestamp()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(tar_target(x, seq_len(2)))
}, ask = FALSE)
tar_make()
# targets older than 1 week ago
tar_older(Sys.time() - as.difftime(1, units = "weeks"))
# targets older than 1 week from now
tar_older(Sys.time() + as.difftime(1, units = "weeks"))
# Everything is still up to date.
tar_make()
# Invalidate all targets targets older than 1 week from now
# so they run on the next tar_make().
invalidate_these <- tar_older(Sys.time() + as.difftime(1, units = "weeks"))
tar_invalidate(any_of(invalidate_these))
tar_make()
})
}
```

tar_option_get *Get a target option.*

Description

Get a target option. These options include default arguments to tar_target() such as packages, storage format, iteration type, and cue. Needs to be called before any calls to tar_target() in order to take effect.

Usage

```
tar_option_get(name = NULL, option = NULL)
```

Arguments

name	Character of length 1, name of an option to get. Must be one of the argument names of tar_option_set().
option	Deprecated, use the name argument instead.

Details

This function goes well with tar_target_raw() when it comes to defining external interfaces on top of the targets package to create pipelines.

Value

Value of a target option.

See Also

Other configuration: tar_config_get(), tar_config_projects(), tar_config_set(), tar_config_unset(), tar_config_yaml(), tar_envvars(), tar_option_reset(), tar_option_set()

Examples

```
tar_option_get("format") # default format before we set anything
tar_target(x, 1)$settings$format
tar_option_set(format = "fst_tbl") # new default format
tar_option_get("format")
tar_target(x, 1)$settings$format
tar_option_reset() # reset the format
tar_target(x, 1)$settings$format
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(cue = tar_cue(mode = "always")) # All targets always run.
  list(tar_target(x, 1), tar_target(y, 2))
})
tar_make()
tar_make()
})
}
```

---

tar_option_reset *Reset all target options.*

Description

Reset all target options you previously chose with `tar_option_set()`. These options are mostly configurable default arguments to `tar_target()` and `tar_target_raw()`.

Usage

```
tar_option_reset()
```

Value

`NULL` (invisibly).

See Also

Other configuration: `tar_config_get()`, `tar_config_projects()`, `tar_config_set()`, `tar_config_unset()`, `tar_config_yaml()`, `tar_envvars()`, `tar_option_get()`, `tar_option_set()`

Examples

```
tar_option_get("format") # default format before we set anything
tar_target(x, 1)$settings$format
tar_option_set(format = "fst_tbl") # new default format
tar_option_get("format")
tar_target(x, 1)$settings$format
tar_option_reset() # reset all options
tar_target(x, 1)$settings$format
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(cue = tar_cue(mode = "always"))
  tar_option_reset() # Undo option above.
  list(tar_target(x, 1), tar_target(y, 2))
})
tar_make()
tar_make()
})
}
```

---

tar_option_set *Set target options.*

Description

Set target options, including default arguments to `tar_target()` such as packages, storage format, iteration type, and cue. Only the non-null arguments are actually set as options. See currently set options with `tar_option_get()`. To use `tar_option_set()` effectively, put it in your workflow's target script file (default: `_targets.R`) before calls to `tar_target()` or `tar_target_raw()`.

Usage

```
tar_option_set(
  tidy_eval = NULL,
  packages = NULL,
  imports = NULL,
  library = NULL,
  envir = NULL,
  format = NULL,
  repository = NULL,
  repository_meta = NULL,
  iteration = NULL,
  error = NULL,
  memory = NULL,
  garbage_collection = NULL,
  deployment = NULL,
  priority = NULL,
  backoff = NULL,
  resources = NULL,
  storage = NULL,
  retrieval = NULL,
  cue = NULL,
  description = NULL,
  debug = NULL,
  workspaces = NULL,
  workspace_on_error = NULL,
  seed = NULL,
  controller = NULL,
  trust_timestamps = NULL,
  trust_object_timestamps = NULL
)
```

Arguments

`tidy_eval`
: Logical, whether to enable tidy evaluation when interpreting `command` and `pattern`. If `TRUE`, you can use the "bang-bang" operator `!!` to programmatically insert the values of global objects.

`packages`
: Character vector of packages to load right before the target runs or the output data is reloaded for downstream targets. Use `tar_option_set()` to set packages globally for all subsequent targets you define.

`imports`
: Character vector of package names. For every package listed, `targets` tracks every dataset and every object in the package namespace as if it were part of the global namespace. As an example, say you have a package called `customAnalysisPackage` which contains an object called `analysis_function()`. If you write `tar_option_set(imports = "yourAnalysisPackage")` in your target script file (default: `_targets.R`), then a function called `"analysis_function"` will show up in the `tar_visnetwork()` graph, and any targets or functions referring to the symbol `"analysis_function"` will depend on the function `analysis_function()` from package `yourAnalysisPackage`. This is best combined with `tar_option_set(packages = "yourAnalysisPackage")` so that `analysis_function()` can actually be called in your code.

  There are several important limitations: 1. Namespaced calls, e.g. `yourAnalysisPackage::analysis_fu` are ignored because of the limitations in `codetools::findGlobals()` which powers the static code analysis capabilities of `targets`. 2. The `imports` option only looks at R objects and R code. It not account for low-level compiled code such as C/C++ or Fortran. 3. If you supply multiple packages, e.g. `tar_option_set(imports = c("p1", "p2"))`, then the objects in `p1` override the objects in `p2` if there are name conflicts. 4. Similarly, objects in `tar_option_get("envir")` override everything in `tar_option_get("imports")`.

`library`
: Character vector of library paths to try when loading `packages`.

`envir`
: Environment containing functions and global objects common to all targets in the pipeline. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

  If `envir` is the global environment, all the promise objects are diffused before sending the data to parallel workers in `tar_make_future()` and `tar_make_clustermq()`, but otherwise the environment is unmodified. This behavior improves performance by decreasing the size of data sent to workers.

  If `envir` is not the global environment, then it should at least inherit from the global environment or base environment so `targets` can access attached packages. In the case of a non-global `envir`, `targets` attempts to remove potentially high memory objects that come directly from `targets`. That includes `tar_target()` objects of class `"tar_target"`, as well as objects of class `"tar_pipeline"` or `"tar_algorithm"`. This behavior improves performance by decreasing the size of data sent to workers.

  Package environments should not be assigned to `envir`. To include package objects as upstream dependencies in the pipeline, assign the package to the `packages` and `imports` arguments of `tar_option_set()`.

`format` Optional storage format for the target's return value. With the exception of `format = "file"`, each target gets a file in `_targets/objects`, and each format is a different way to save and load this file. See the "Storage formats" section for a detailed list of possible data storage formats.

`repository` Character of length 1, remote repository for target storage. Choices:

- `"local"`: file system of the local machine.
- `"aws"`: Amazon Web Services (AWS) S3 bucket. Can be configured with a non-AWS S3 bucket using the `endpoint` argument of `tar_resources_aws()`, but versioning capabilities may be lost in doing so. See the cloud storage section of `https://books.ropensci.org/targets/data.html` for details for instructions.
- `"gcp"`: Google Cloud Platform storage bucket. See the cloud storage section of `https://books.ropensci.org/targets/data.html` for details for instructions.
- A character string from `tar_repository_cas()` for content-addressable storage.

Note: if `repository` is not `"local"` and `format` is `"file"` then the target should create a single output file. That output file is uploaded to the cloud and tracked for changes where it exists in the cloud. The local file is deleted after the target runs.

`repository_meta`

Character of length 1 with the same values as `repository` but excluding content-addressable storage (`"aws"`, `"gcp"`, `"local"`). Cloud repository for the metadata text files in `_targets/meta/`, including target metadata and progress data. Defaults to `tar_option_get("repository")` except in the case of content-addressable storage (CAS). When `tar_option_get("repository")` is a CAS repository, the default value of `repository_meta` is `"local"`.

`iteration` Character of length 1, name of the iteration mode of the target. Choices:

- `"vector"`: branching happens with `vctrs::vec_slice()` and aggregation happens with `vctrs::vec_c()`.
- `"list"`, branching happens with `[[]]` and aggregation happens with `list()`.
- `"group"`: `dplyr::group_by()`-like functionality to branch over subsets of a non-dynamic data frame. For `iteration = "group"`, the target must not by dynamic (the `pattern` argument of `tar_target()` must be left `NULL`). The target's return value must be a data frame with a special `tar_group` column of consecutive integers from 1 through the number of groups. Each integer designates a group, and a branch is created for each collection of rows in a group. See the `tar_group()` function to see how you can create the special `tar_group` column with `dplyr::group_by()`.

`error` Character of length 1, what to do if the target stops and throws an error. Options:

- `"stop"`: the whole pipeline stops and throws an error.
- `"continue"`: the whole pipeline keeps going.

- `"null"`: The errored target continues and returns `NULL`. The data hash is deliberately wrong so the target is not up to date for the next run of the pipeline. In addition, as of `targets` version 1.8.0.9011, a value of `NULL` is given to upstream dependencies with `error = "null"` if loading fails.
- `"abridge"`: any currently running targets keep running, but no new targets launch after that.
- `"trim"`: all currently running targets stay running. A queued target is allowed to start if:
  1. It is not downstream of the error, and
  2. It is not a sibling branch from the same `tar_target()` call (if the error happened in a dynamic branch).

  The idea is to avoid starting any new work that the immediate error impacts. `error = "trim"` is just like `error = "abridge"`, but it allows potentially healthy regions of the dependency graph to begin running. (Visit `https://books.ropensci.org/targets/debugging.html` to learn how to debug targets using saved workspaces.)

`memory` Character of length 1, memory strategy. Possible values:

- `"auto"`: new in `targets` version 1.8.0.9011, `memory = "auto"` is equivalent to `memory = "transient"` for dynamic branching (a non-null `pattern` argument) and `memory = "persistent"` for targets that do not use dynamic branching.
- `"persistent"`: the target stays in memory until the end of the pipeline (unless `storage` is `"worker"`, in which case `targets` unloads the value from memory right after storing it in order to avoid sending copious data over a network).
- `"transient"`: the target gets unloaded after every new target completes. Either way, the target gets automatically loaded into memory whenever another target needs the value.

For cloud-based dynamic files (e.g. `format = "file"` with `repository = "aws"`), the `memory` option applies to the temporary local copy of the file: `"persistent"` means it remains until the end of the pipeline and is then deleted, and `"transient"` means it gets deleted as soon as possible. The former conserves bandwidth, and the latter conserves local storage.

`garbage_collection`
A non-negative integer. If `0`, do not run garbage collection. If `1`, run garbage collection on every target that is not skipped, both locally and on all parallel workers. If `garbage_collection` is a positive integer `n`, then garbage collection runs every `n`'th target that is not skipped. For example, `garbage_collection = 3` will run garbage collection on every third active target, both locally and on all parallel workers.

`deployment` Character of length 1. If `deployment` is `"main"`, then the target will run on the central controlling R process. Otherwise, if `deployment` is `"worker"` and you set up the pipeline with distributed/parallel computing, then the target runs on a parallel worker. For more on distributed/parallel computing in `targets`, please visit `https://books.ropensci.org/targets/crew.html`.

`priority`	Numeric of length 1 between 0 and 1. Controls which targets get deployed first when multiple competing targets are ready simultaneously. Targets with priorities closer to 1 get dispatched earlier (and polled earlier in `tar_make_future()`).
`backoff`	An object from `tar_backoff()` configuring the exponential backoff algorithm of the pipeline. See `tar_backoff()` for details. A numeric argument for `backoff` is still allowed, but deprecated.
`resources`	Object returned by `tar_resources()` with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of `targets`. See `tar_resources()` for details.
`storage`	Character string to control when the output of the target is saved to storage. Only relevant when using `targets` with parallel workers (`https://books.ropensci.org/targets/crew.html`). Must be one of the following values: • `"main"`: the target's return value is sent back to the host machine and saved/uploaded locally. • `"worker"`: the worker saves/uploads the value. • `"none"`: `targets` makes no attempt to save the result of the target to storage in the location where `targets` expects it to be. Saving to storage is the responsibility of the user. Use with caution.
`retrieval`	Character string to control when the current target loads its dependencies into memory before running. (Here, a "dependency" is another target upstream that the current one depends on.) Only relevant when using `targets` with parallel workers (`https://books.ropensci.org/targets/crew.html`). Must be one of the following values: • `"main"`: the target's dependencies are loaded on the host machine and sent to the worker before the target runs. • `"worker"`: the worker loads the target's dependencies. • `"none"`: `targets` makes no attempt to load its dependencies. With `retrieval = "none"`, loading dependencies is the responsibility of the user. Use with caution.
`cue`	An optional object from `tar_cue()` to customize the rules that decide whether the target is up to date.
`description`	Character of length 1, a custom free-form human-readable text description of the target. Descriptions appear as target labels in functions like `tar_manifest()` and `tar_visnetwork()`, and they let you select subsets of targets for the names argument of functions like `tar_make()`. For example, `tar_manifest(names = tar_described_as(starts_with("survival model")))` lists all the targets whose descriptions start with the character string `"survival model"`.
`debug`	Character vector of names of targets to run in debug mode. To use effectively, you must set `callr_function = NULL` and restart your R session just before running. You should also `tar_make()`, `tar_make_clustermq()`, or `tar_make_future()`. For any target mentioned in debug, `targets` will force the target to run locally (with `tar_cue(mode = "always")` and `deployment = "main"` in the settings) and pause in an interactive debugger to help you diagnose problems. This is like inserting a `browser()` statement at the beginning of the target's expression, but without invalidating any targets.

`workspaces` Character vector of target names. Could be non-branching targets, whole dynamic branching targets, or individual branch names. `tar_make()` and friends will save workspace files for these targets even if the targets are skipped. Workspace files help with debugging. See `tar_workspace()` for details about workspaces.

`workspace_on_error`
Logical of length 1, whether to save a workspace file for each target that throws an error. Workspace files help with debugging. See `tar_workspace()` for details about workspaces.

`seed` Integer of length 1, seed for generating target-specific pseudo-random number generator seeds. These target-specific seeds are deterministic and depend on `tar_option_get("seed")` and the target name. Target-specific seeds are safely and reproducibly applied to each target's command, and they are stored in the metadata and retrievable with `tar_meta()` or `tar_seed()`.

Either the user or third-party packages built on top of `targets` may still set seeds inside the command of a target. For example, some target factories in the `tarchetypes` package assigns replicate-specific seeds for the purposes of reproducible within-target batched replication. In cases like these, the effect of the target-specific seed saved in the metadata becomes irrelevant and the seed defined in the command applies.

The `seed` option can also be `NA` to disable automatic seed-setting. Any targets defined while `tar_option_get("seed")` is `NA` will not set a seed. In this case, those targets will never be up to date unless they have `cue = tar_cue(seed = FALSE)`.

`controller` A controller or controller group object produced by the `crew` R package. `crew` brings auto-scaled distributed computing to `tar_make()`.

`trust_timestamps`
Logical of length 1, whether to use file system modification timestamps to check whether the target output data files in are up to date. This is an advanced setting and usually does not need to be set by the user except on old or difficult platforms.

If `trust_timestamps` was reset with `tar_option_reset()` or never set at all (recommended) then `targets` makes a decision based on the type of file system of the given file.

If `trust_timestamps` is `TRUE` (default), then `targets` looks at the timestamp first. If it agrees with the timestamp recorded in the metadata, then `targets` considers the file unchanged. If the timestamps disagree, then `targets` recomputes the hash to make a final determination. This practice reduces the number of hash computations and thus saves time.

However, timestamp precision varies from a few nanoseconds at best to 2 entire seconds at worst, and timestamps with poor precision should not be fully trusted if there is any possibility that you will manually change the file within 2 seconds after the pipeline finishes. If the data store is on a file system with low-precision timestamps, then you may consider setting `trust_timestamps` to `FALSE` so `targets` errs on the safe side and always recomputes the hashes of files.

To check if your file system has low-precision timestamps, you can run `file.create("x"); nanonext::` from within the directory containing the `_targets` data store and then check difftime(file.mtime("y"), file.mtime("x"), units = "secs"). If the value from difftime() is around 0.001 seconds (must be strictly above 0 and below 1) then you do not need to set trust_timestamps = FALSE.

trust_object_timestamps
Deprecated. Use trust_timestamps instead.

Value

NULL (invisibly).

Storage formats

targets has several built-in storage formats to control how return values are saved and loaded from disk:

- "rds": Default, uses saveRDS() and readRDS(). Should work for most objects, but slow.
- "auto": either "file" or "qs", depending on the return value of the target. If the return value is a character vector of existing files (and/or directories), then the format becomes "file" before tar_make() saves the target. Otherwise, the format becomes "qs".
- "qs": Uses qs2::qs_save() and qs2::qs_read(). Should work for most objects, much faster than "rds". Optionally configure settings through tar_resources() and tar_resources_qs().

  Prior to targets version 1.8.0.9014, format = "qs" used the qs package. qs has since been superseded in favor of qs2, and so later versions of targets use qs2 to save new data. To read existing data, targets first attempts qs2::qs_read(), and then if that fails, it falls back on qs::qread().
- "feather": Uses arrow::write_feather() and arrow::read_feather() (version 2.0). Much faster than "rds", but the value must be a data frame. Optionally set compression and compression_level in arrow::write_feather() through tar_resources() and tar_resources_feather(). Requires the arrow package (not installed by default).
- "parquet": Uses arrow::write_parquet() and arrow::read_parquet() (version 2.0). Much faster than "rds", but the value must be a data frame. Optionally set compression and compression_level in arrow::write_parquet() through tar_resources() and tar_resources_parquet(). Requires the arrow package (not installed by default).
- "fst": Uses fst::write_fst() and fst::read_fst(). Much faster than "rds", but the value must be a data frame. Optionally set the compression level for fst::write_fst() through tar_resources() and tar_resources_fst(). Requires the fst package (not installed by default).
- "fst_dt": Same as "fst", but the value is a data.table. Deep copies are made as appropriate in order to protect against the global effects of in-place modification. Optionally set the compression level the same way as for "fst".
- "fst_tbl": Same as "fst", but the value is a tibble. Optionally set the compression level the same way as for "fst".
- "keras": superseded by tar_format() and incompatible with error = "null" (in tar_target() or tar_option_set()). Uses keras::save_model_hdf5() and keras::load_model_hdf5(). The value must be a Keras model. Requires the keras package (not installed by default).

- "torch": superseded by tar_format() and incompatible with error = "null" (in tar_target() or tar_option_set()). Uses torch::torch_save() and torch::torch_load(). The value must be an object from the torch package such as a tensor or neural network module. Requires the torch package (not installed by default).
- "file": A dynamic file. To use this format, the target needs to manually identify or save some data and return a character vector of paths to the data (must be a single file path if repository is not "local"). (These paths must be existing files and nonempty directories.) Then, targets automatically checks those files and cues the appropriate run/skip decisions if those files are out of date. Those paths must point to files or directories, and they must not contain characters | or *. All the files and directories you return must actually exist, or else targets will throw an error. (And if storage is "worker", targets will first stall out trying to wait for the file to arrive over a network file system.) If the target does not create any files, the return value should be character(0).

  If repository is not "local" and format is "file", then the character vector returned by the target must be of length 1 and point to a single file. (Directories and vectors of multiple file paths are not supported for dynamic files on the cloud.) That output file is uploaded to the cloud and tracked for changes where it exists in the cloud. The local file is deleted after the target runs.
- "url": A dynamic input URL. For this storage format, repository is implicitly "local", URL format is like format = "file" except the return value of the target is a URL that already exists and serves as input data for downstream targets. Optionally supply a custom curl handle through tar_resources() and tar_resources_url(). in new_handle(), nobody = TRUE is important because it ensures targets just downloads the metadata instead of the entire data file when it checks time stamps and hashes. The data file at the URL needs to have an ETag or a Last-Modified time stamp, or else the target will throw an error because it cannot track the data. Also, use extreme caution when trying to use format = "url" to track uploads. You must be absolutely certain the ETag and Last-Modified time stamp are fully updated and available by the time the target's command finishes running. targets makes no attempt to wait for the web server.
- A custom format can be supplied with tar_format(). For this choice, it is the user's responsibility to provide methods for (un)serialization and (un)marshaling the return value of the target.
- The formats starting with "aws_" are deprecated as of 2022-03-13 (targets version > 0.10.0). For cloud storage integration, use the repository argument instead.

Formats "rds", "file", and "url" are general-purpose formats that belong in the targets package itself. Going forward, any additional formats should be implemented with tar_format() in third-party packages like tarchetypes and geotargets (for example: tarchetypes::tar_format_nanoparquet()). Formats "qs", "fst", etc. are legacy formats from before the existence of tar_format(), and they will continue to remain in targets without deprecation.

See Also

Other configuration: tar_config_get(), tar_config_projects(), tar_config_set(), tar_config_unset(), tar_config_yaml(), tar_envvars(), tar_option_get(), tar_option_reset()

Examples

```
tar_option_get("format") # default format before we set anything
```

```
tar_target(x, 1)$settings$format
tar_option_set(format = "fst_tbl") # new default format
tar_option_get("format")
tar_target(x, 1)$settings$format
tar_option_reset() # reset the format
tar_target(x, 1)$settings$format
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(cue = tar_cue(mode = "always")) # All targets always run.
  list(tar_target(x, 1), tar_target(y, 2))
})
tar_make()
tar_make()
})
}
```

tar_outdated *Check which targets are outdated.*

Description

Checks for outdated targets in the pipeline, targets that will be rerun automatically if you call `tar_make()` or similar. See `tar_cue()` for the rules that decide whether a target needs to rerun.

Usage

```
tar_outdated(
  names = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  branches = FALSE,
  targets_only = TRUE,
  reporter = targets::tar_config_get("reporter_outdated"),
  seconds_reporter = targets::tar_config_get("seconds_reporter_outdated"),
  seconds_interval = targets::tar_config_get("seconds_interval"),
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Names of the targets. `tar_outdated()` will check these targets and all upstream ancestors in the dependency graph. Set `names` to `NULL` to check/build all the targets (default). The object supplied to `names` should be `NULL` or a

	tidyselect expression like any_of() or starts_with() from tidyselect itself, or tar_described_as() to select target names based on their descriptions.
shortcut	Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. If TRUE, then the function only checks the targets in names and uses stored metadata for information about upstream dependencies as needed. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, shortcut = TRUE only works if you set names.
branches	Logical of length 1, whether to include branch names. Including branches could get cumbersome for large pipelines. Individual branch names are still omitted when branch-specific information is not reliable: for example, when a pattern branches over an outdated target.
targets_only	Logical of length 1, whether to just restrict to targets or to include functions and other global objects from the environment created by running the target script file (default: _targets.R).
reporter	Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: * "forecast_interactive" (default): use the forecast reporter if the session is interactive (see base::interactive()), otherwise use the silent reporter. * "silent": print nothing. * "forecast": print running totals of the checked and outdated targets found so far.
seconds_reporter	Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is max(1, seconds_reporter) to reduce overhead.
seconds_interval	Deprecated on 2023-08-24 (targets version 1.2.2.9001). Use seconds_meta_append, seconds_meta_upload, and seconds_reporter instead.
callr_function	A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.
callr_arguments	A list of arguments to callr_function.
envir	An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc. The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before

	running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

Requires that you define a pipeline with a target script file (default: `_targets.R`). (See `tar_script()` for details.)

Value

Names of the outdated targets.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other inspect: `tar_deps()`, `tar_manifest()`, `tar_network()`, `tar_sitrep()`, `tar_validate()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(list(tar_target(x, 1 + 1)))
tar_outdated()
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_outdated()
})
}
```

`tar_path_script` *Current target script path*

Description

Identify the file path to the target script of the pipeline currently running.

Usage

```
tar_path_script()
```

Value

Character, file path to the target script of the pipeline currently running. If called outside of the pipeline currently running, `tar_path_script()` returns `tar_config_get("script")`.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
tar_path_script()
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
script <- tempfile()
tar_script(tar_target(x, tar_path_script()), script = script, ask = FALSE)
tar_make(script = script)
tar_read(x)
})
}
```

tar_path_script_support *Directory path to the support scripts of the current target script*

Description

Identify the directory path to the support scripts of the current target script of the pipeline currently running.

Usage

```
tar_path_script_support()
```

Details

A target script (default: `_targets.R`) comes with support scripts if it is written by Target Markdown. These support scripts usually live in a folder called `_targets_r/`, but the path may vary from case to case. The `tar_path_scipt_support()` returns the path to the folder with the support scripts.

Value

Character, directory path to the target script of the pipeline currently running. If called outside of the pipeline currently running, `tar_path_script()` returns `tar_config_get("script")`.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
tar_path_script_support()
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
script <- tempfile()
tar_script(
  tar_target(x, tar_path_script_support()),
  script = script,
  ask = FALSE
)
tar_make(script = script)
tar_read(x)
})
}
```

`tar_path_store` *Current data store path*

Description

Identify the file path to the data store of the pipeline currently running.

Usage

```
tar_path_store()
```

Value

Character, file path to the data store of the pipeline currently running. If called outside of the pipeline currently running, `tar_path_store()` returns `tar_config_get("store")`.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_target()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
tar_path_store()
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(tar_target(x, tar_path_store()), ask = FALSE)
store <- tempfile()
tar_make(store = store)
tar_read(x, store = store)
})
}
```

`tar_path_target` *Identify the file path where a target will be stored.*

Description

Identify the file path where a target will be stored after the target finishes running in the pipeline.

Usage

```
tar_path_target(
  name = NULL,
  default = NA_character_,
  create_dir = FALSE,
  store = targets::tar_config_get("store")
)
```

Arguments

`name`	Symbol, name of a target. If `NULL`, `tar_path_target()` returns the path of the target currently running in a pipeline.
`default`	Character, value to return if `tar_path_target()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.
`create_dir`	Logical of length 1, whether to create `dirname(tar_path_target())` in `tar_path_target()` itself. This is useful if you are writing to `tar_path_target()` from inside a `storage = "none"` target and need the parent directory of the file to exist.
`store`	Character of length 1, path to the data store if `tar_path_target()` is called outside a running pipeline. If `tar_path_target()` is called inside a running pipeline, this argument is ignored and actual the path to the running pipeline's data store is used instead.

Value

Character, file path of the return value of the target. If not called from inside a running target, `tar_path_target(name = your_target)` just returns `_targets/objects/your_target`, the file path where `your_target` will be saved unless `format` is equal to `"file"` or any of the supported cloud-based storage formats.

For non-cloud storage formats, if you call `tar_path_target()` with no arguments while target `x` is running, the `name` argument defaults to the name of the running target, so `tar_path_target()` returns `_targets/objects/x`.

For cloud-backed formats, `tar_path_target()` returns the path to the staging file in `_targets/scratch/`. That way, even if you select a cloud repository (e.g. `tar_target(..., repository = "aws", storage = "none")`) then you can still manually write to `tar_path_target(create_dir = TRUE)` and the `targets` package will automatically hash it and upload it to the AWS S3 bucket. This does not apply to `format = "file"`, where you would never need `storage = "none"` anyway.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_source()`, `tar_store()`, `tar_unblock_process()`

Examples

```
tar_path_target()
tar_path_target(your_target)
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(tar_target(returns_path, tar_path_target()), ask = FALSE)
tar_make()
tar_read(returns_path)
})
}
```

`tar_pattern` *Emulate dynamic branching.*

Description

Emulate the dynamic branching process outside a pipeline. `tar_pattern()` can help you understand the overall branching structure that comes from the `pattern` argument of `tar_target()`.

Usage

```
tar_pattern(pattern, ..., seed = 0L)
```

Arguments

`pattern`	Function call with the pattern specification.
`...`	Named integers, each of length 1. Each name is the name of a dependency target, and each integer is the length of the target (number of branches or slices). Names must be unique.
`seed`	Integer of length 1, random number generator seed to emulate the pattern reproducibly. (The `sample()` pattern is random). In a real pipeline, the seed is automatically generated from the target name in deterministic fashion.

Details

Dynamic branching is a way to programmatically create multiple new targets based on the values of other targets, all while the pipeline is running. Use the `pattern` argument of `tar_target()` to get started. `pattern` accepts a function call composed of target names and any of the following patterns:

- `map()`: iterate over one or more targets in sequence.
- `cross()`: iterate over combinations of slices of targets.
- `slice()`: select one or more slices by index, e.g. `slice(x, index = c(3, 4))` selects the third and fourth slice or branch of `x`.
- `head()`: restrict branching to the first few elements.
- `tail()`: restrict branching to the last few elements.
- `sample()`: restrict branching to a random subset of elements.

Value

A `tibble` showing the kinds of dynamic branches that `tar_target()` would create in a real pipeline with the given `pattern`. Each row is a dynamic branch, each column is a dependency target, and each element is the name of an upstream bud or branch that the downstream branch depends on. Buds are pieces of non-branching targets ("stems") and branches are pieces of patterns. The returned bud and branch names are not the actual ones you will see when you run the pipeline, but they do communicate the branching structure of the pattern.

See Also

Other branching: `tar_branch_index()`, `tar_branch_names()`, `tar_branches()`

Examples

```
# To use dynamic map for real in a pipeline,
# call map() in a target's pattern.
# The following code goes at the bottom of
# your target script file (default: `_targets.R`).
list(
  tar_target(x, seq_len(2)),
  tar_target(y, head(letters, 2)),
  tar_target(dynamic, c(x, y), pattern = map(x, y)) # 2 branches
)
# Likewise for more complicated patterns.
list(
  tar_target(x, seq_len(2)),
  tar_target(y, head(letters, 2)),
  tar_target(z, head(LETTERS, 2)),
  tar_target(dynamic, c(x, y, z), pattern = cross(z, map(x, y))) #4 branches
)
# But you can emulate dynamic branching without running a pipeline
# in order to understand the patterns you are creating. Simply supply
# the pattern and the length of each dependency target.
# The returned data frame represents the branching structure of the pattern:
# One row per new branch, one column per dependency target, and
# one element per bud/branch in each dependency target.
tar_pattern(
  cross(x, map(y, z)),
  x = 2,
  y = 3,
  z = 3
)
tar_pattern(
  head(cross(x, map(y, z)), n = 2),
  x = 2,
  y = 3,
  z = 3
)
```

tar_pid *Get main process ID.*

Description

Get the process ID (PID) of the most recent main R process to orchestrate the targets of the current project.

Usage

```
tar_pid(store = targets::tar_config_get("store"))
```

Arguments

`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

The main process is the R process invoked by `tar_make()` or similar. If `callr_function` is not `NULL`, this is an external process, and the `pid` in the return value will not agree with `Sys.getpid()` in your current interactive session. The process may or may not be alive. You may want to check it with `ps::ps_is_running(ps::ps_handle(targets::tar_pid()))` before running another call to `tar_make()` for the same project.

Value

Integer with the process ID (PID) of the most recent main R process to orchestrate the targets of the current project.

See Also

Other data: `tar_crew()`, `tar_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
Sys.getpid()
tar_pid() # Different from the current PID.
})
}
```

---

tar_poll *Repeatedly poll progress in the R console.*

Description

Print the information in tar_progress_summary() at regular intervals.

Usage

```
tar_poll(
  interval = 1,
  timeout = Inf,
  fields = c("skipped", "dispatched", "completed", "errored", "canceled", "since"),
  store = targets::tar_config_get("store")
)
```

Arguments

interval	Number of seconds to wait between iterations of polling progress.
timeout	How many seconds to run before exiting.
fields	Optional character vector of names of progress data columns to read. Set to `NULL` to read all fields.
store	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

`NULL` (invisibly). Called for its side effects.

See Also

Other progress: `tar_canceled()`, `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_progress()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_skipped()`, `tar_watch()`, `tar_watch_server()`, `tar_watch_ui()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  list(
    tar_target(x, seq_len(100)),
    tar_target(y, Sys.sleep(0.1), pattern = map(x))
  )
}, ask = FALSE)
px <- tar_make(callr_function = callr::r_bg, reporter = "silent")
tar_poll()
})
}
```

`tar_process` *Get main process info.*

Description

Get info on the most recent main R process to orchestrate the targets of the current project.

Usage

```
tar_process(names = NULL, store = targets::tar_config_get("store"))
```

Arguments

`names`	Optional, names of the data points to return. If supplied, `tar_process()` returns only the rows of the names you select. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()`.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

The main process is the R process invoked by `tar_make()` or similar. If `callr_function` is not `NULL`, this is an external process, and the `pid` in the return value will not agree with `Sys.getpid()` in your current interactive session. The process may or may not be alive. You may want to check the status with `tar_pid() %in% ps::ps_pids()` before running another call to `tar_make()` for the same project.

Value

A data frame with metadata on the most recent main R process to orchestrate the targets of the current project. The output includes the `pid` of the main process.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other data: `tar_crew()`, `tar_pid()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_process()
tar_process(pid)
})
}
```

`tar_progress` *Read progress.*

Description

Read a project's target progress data for the most recent run of `tar_make()` or similar. Only the most recent record is shown.

Usage

```
tar_progress(
  names = NULL,
  fields = "progress",
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Optional, names of the targets. If supplied, the output is restricted to the selected targets. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`fields`	Optional, names of progress data columns to read. Set to `NULL` to read all fields.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A data frame with one row per target and the following columns:

- `name`: name of the target.
- `type`: type of target: `"stem"` for non-branching targets, `"pattern"` for dynamically branching targets, and `"branch"` for dynamic branches.
- `parent`: name of the target's parent. For branches, this is the name of the associated pattern. For other targets, the pattern is just itself.
- `branches`: number of dynamic branches of a pattern. 0 for non-patterns.
- `progress`: the most recent progress update of that target. Could be `"dispatched"`, `"completed"`, `"skipped"`, `"canceled"`, or `"errored"`. `"dispatched"` means the target was sent off to be run, but in the case of `tar_make()` with a `crew` controller, the target might not actually start running right away if the `crew` workers are all busy.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other progress: `tar_canceled()`, `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_skipped()`, `tar_watch()`, `tar_watch_server()`, `tar_watch_ui()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_progress()
tar_progress(starts_with("y_")) # see also any_of()
})
}
```

---

tar_progress_branches *Tabulate the progress of dynamic branches.*

Description

Read a project's target progress data for the most recent run of the pipeline and display the tabulated status of dynamic branches. Only the most recent record is shown.

Usage

```
tar_progress_branches(
  names = NULL,
  fields = NULL,
  store = targets::tar_config_get("store")
)
```

Arguments

names	Optional, names of the targets. If supplied, `tar_progress()` only returns progress information on these targets. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
fields	Optional, names of progress data columns to read. Set to `NULL` to read all fields.
store	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A data frame with one row per target per progress status and the following columns.

- `name`: name of the pattern.
- `progress`: progress status: `"dispatched"`, `"completed"`, `"canceled"`, or `"errored"`.

- `branches`: number of branches in the progress category.
- `total`: total number of branches planned for the whole pattern. Values within the same pattern should all be equal.

See Also

Other progress: `tar_canceled()`, `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_poll()`, `tar_progress()`, `tar_progress_summary()`, `tar_skipped()`, `tar_watch()`, `tar_watch_server()`, `tar_watch_ui()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, x, pattern = map(x)),
    tar_target(z, stopifnot(y < 1.5), pattern = map(y))
  )
}, ask = FALSE)
try(tar_make())
tar_progress_branches()
})
}
```

`tar_progress_summary` *Summarize target progress.*

Description

Summarize the progress of a run of the pipeline.

Usage

```
tar_progress_summary(
  fields = c("skipped", "dispatched", "completed", "errored", "canceled", "since"),
  store = targets::tar_config_get("store")
)
```

Arguments

`fields`	Optional character vector of names of progress data columns to read. Set to `NULL` to read all fields.

store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Value

A data frame with one row and the following optional columns that can be selected with fields. (time is omitted by default.)

- dispatched: number of targets that were sent off to run and did not (yet) finish. These targets may not actually be running, depending on the status and workload of parallel workers.
- completed: number of targets that completed without error or cancellation.
- errored: number of targets that threw an error.
- canceled: number of canceled targets (see tar_cancel()).
- since: how long ago progress last changed (Sys.time() - time).
- time: the time when the progress last changed (modification timestamp of the _targets/meta/progress file).

See Also

Other progress: tar_canceled(), tar_completed(), tar_dispatched(), tar_errored(), tar_poll(), tar_progress(), tar_progress_branches(), tar_skipped(), tar_watch(), tar_watch_server(), tar_watch_ui()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, x, pattern = map(x)),
    tar_target(z, stopifnot(y < 1.5), pattern = map(y), error = "continue")
  )
}, ask = FALSE)
try(tar_make())
tar_progress_summary()
})
}
```

tar_prune *Remove targets that are no longer part of the pipeline.*

Description

Remove target values from `_targets/objects/` and the cloud and remove target metadata from `_targets/meta/meta` for targets that are no longer part of the pipeline.

Usage

```
tar_prune(
  cloud = TRUE,
  batch_size = 1000L,
  verbose = TRUE,
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`cloud`	Logical of length 1, whether to delete objects from the cloud if applicable (e.g. AWS, GCP). If `FALSE`, files are not deleted from the cloud.
`batch_size`	Positive integer between 1 and 1000, number of target objects to delete from the cloud with each HTTP API request. Currently only supported for AWS. Cannot be more than 1000.
`verbose`	Logical of length 1, whether to print console messages to show progress when deleting each batch of targets from each cloud bucket. Batched deletion with verbosity is currently only supported for AWS.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.
`envir`	An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before

running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

`tar_prune()` is useful if you recently worked through multiple changes to your project and are now trying to discard irrelevant data while keeping the results that still matter. Global objects and local files with `format = "file"` outside the data store are unaffected. Also removes `_targets/scratch/`, which is only needed while `tar_make()`, `tar_make_clustermq()`, or `tar_make_future()` is running. To list the targets that will be pruned without actually removing anything, use `tar_prune_list()`.

Value

`NULL` except if `callr_function` is `callr::r_bg`, in which case a handle to the `callr` background process is returned. Either way, the value is invisibly returned.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

Cloud target data versioning

Some buckets in Amazon S3 or Google Cloud Storage are "versioned", which means they track historical versions of each data object. If you use `targets` with cloud storage (`https://books.ropensci.org/targets/cloud-storage.html`) and versioning is turned on, then `targets` will record each version of each target in its metadata.

Functions like `tar_read()` and `tar_load()` load the version recorded in the local metadata, which may not be the same as the "current" version of the object in the bucket. Likewise, functions `tar_delete()` and `tar_destroy()` only remove the version ID of each target as recorded in the local metadata.

If you want to interact with the *latest* version of an object instead of the version ID recorded in the local metadata, then you will need to delete the object from the metadata.

1. Make sure your local copy of the metadata is current and up to date. You may need to run `tar_meta_download()` or `tar_meta_sync()` first.
2. Run `tar_unversion()` to remove the recorded version IDs of your targets in the local metadata.
3. With the version IDs gone from the local metadata, functions like `tar_read()` and `tar_destroy()` will use the *latest* version of each target data object.
4. Optional: to back up the local metadata file with the version IDs deleted, use `tar_meta_upload()`.

See Also

tar_prune_inspect

Other clean: `tar_delete()`, `tar_destroy()`, `tar_invalidate()`, `tar_prune_list()`, `tar_unversion()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
# Remove some targets from the pipeline.
tar_script(list(tar_target(y1, 1 + 1)), ask = FALSE)
# Keep only the remaining targets in the data store.
tar_prune()
})
}
```

`tar_prune_list` *List targets that* `tar_prune()` *will remove.*

Description

List the targets that `tar_prune()` will remove. Does not actually remove any targets.

Usage

```
tar_prune_list(
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

callr_function	A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.
callr_arguments	A list of arguments to callr_function.
envir	An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc. The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with tar_option_set() from within the target script file. In other words, if you call tar_option_set(envir = envir1) in an interactive session and then tar_make(envir = envir2, callr_function = NULL), then envir2 will be used.
script	Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.
store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Details

See tar_prune() for details.

Value

If `callr_function` is `callr::r_bg`, the return value is a handle to the `callr` background process is returned. Otherwise, the return value is a character vector of target names identifying targets that `tar_prune()` will remove.

See Also

tar_prune

Other clean: `tar_delete()`, `tar_destroy()`, `tar_invalidate()`, `tar_prune()`, `tar_unversion()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
# Remove some targets from the pipeline.
tar_script(list(tar_target(y1, 1 + 1)), ask = FALSE)
# List targets that tar_prune() will remove.
tar_prune_list()
})
}
```

`tar_read` *Read a target's value from storage.*

Description

Read a target's return value from its file in `_targets/objects/`. For file targets (i.e. `format = "file"`) the paths are returned.

`tar_read()` expects an unevaluated symbol for the name argument, whereas `tar_read_raw()` expects a character string.

Usage

```
tar_read(
  name,
  branches = NULL,
  meta = tar_meta(store = store),
  store = targets::tar_config_get("store")
)

tar_read_raw(
  name,
  branches = NULL,
  meta = tar_meta(store = store),
  store = targets::tar_config_get("store")
)
```

Arguments

`name`	Name of the target to read. `tar_read()` expects an unevaluated symbol for the `name` argument, whereas `tar_read_raw()` expects a character string.
`branches`	Integer of indices of the branches to load if the target is a pattern.
`meta`	Data frame of metadata from `tar_meta()`. `tar_read()` with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call `tar_meta()` beforehand and supply it to the `meta` argument, then successive calls to `tar_read()` may run much faster.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

The target's return value from its file in `_targets/objects/`, or the paths to the custom files and directories if `format = "file"` was set.

Cloud target data versioning

Some buckets in Amazon S3 or Google Cloud Storage are "versioned", which means they track historical versions of each data object. If you use `targets` with cloud storage (`https://books.ropensci.org/targets/cloud-storage.html`) and versioning is turned on, then `targets` will record each version of each target in its metadata.

Functions like `tar_read()` and `tar_load()` load the version recorded in the local metadata, which may not be the same as the "current" version of the object in the bucket. Likewise, functions `tar_delete()` and `tar_destroy()` only remove the version ID of each target as recorded in the local metadata.

If you want to interact with the *latest* version of an object instead of the version ID recorded in the local metadata, then you will need to delete the object from the metadata.

1. Make sure your local copy of the metadata is current and up to date. You may need to run `tar_meta_download()` or `tar_meta_sync()` first.
2. Run `tar_unversion()` to remove the recorded version IDs of your targets in the local metadata.
3. With the version IDs gone from the local metadata, functions like `tar_read()` and `tar_destroy()` will use the *latest* version of each target data object.

4. Optional: to back up the local metadata file with the version IDs deleted, use `tar_meta_upload()`.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other storage: `tar_format()`, `tar_load()`, `tar_load_everything()`, `tar_objects()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(tar_target(x, 1 + 1))
})
tar_make()
tar_read(x)
tar_read_raw("x")
})
}
```

`tar_renv` *Set up package dependencies for compatibility with* `renv`

Description

Write package dependencies to a script file (by default, named `_targets_packages.R` in the root project directory). Each package is written to a separate line as a standard `library()` call (e.g. `library(package)`) so `renv` can identify them automatically.

Usage

```
tar_renv(
  extras = c("bslib", "crew", "gt", "markdown", "rstudioapi", "shiny", "shinybusy",
    "shinyWidgets", "visNetwork"),
  path = "_targets_packages.R",
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script")
)
```

Arguments

`extras`	Character vector of additional packages to declare as project dependencies.
`path`	Character of length 1, path to the script file to populate with `library()` calls.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.
`envir`	An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

Details

This function gets called for its side-effect, which writes package dependencies to a script for compatibility with `renv`. The generated file should **not** be edited by hand and will be overwritten each time `tar_renv()` is called.

The behavior of `renv` is to create and manage a project-local `R` library and keep a record of project dependencies in a file called `renv.lock`. To identify dependencies, `renv` crawls through code to find packages explicitly mentioned using `library()`, `require()`, or `::`. However, `targets` manages packages in a way that hides dependencies from `renv`. `tar_renv()` finds package dependencies that would be otherwise hidden to `renv` because they are declared using the `targets` API. Thus, calling `tar_renv` this is only necessary if using `tar_option_set()` or `tar_target()` to use specialized storage formats or manage packages.

With the script written by `tar_renv()`, `renv` is able to crawl the file to identify package dependencies (with `renv::dependencies()`). `tar_renv()` only serves to make your `targets` project compatible with `renv`, it is still the users responsibility to call `renv::init()` and `renv::snapshot()` directly to initialize and manage a project-local `R` library. This allows your `targets` pipeline to have its own self-contained `R` library separate from your standard `R` library. See `https://rstudio.github.io/renv/index.html` for more information.

Value

Nothing, invisibly.

Performance

If you use `renv`, then overhead from project initialization could slow down `tar_make()` and friends. If you experience slowness, please make sure your `renv` library is on a fast file system. (For example, slow network drives can severely reduce performance.) In addition, you can disable the slowest `renv` initialization checks. After confirming at `https://rstudio.github.io/renv/reference/config.html` that you can safely disable these checks, you can write lines `RENV_CONFIG_RSPM_ENABLED=false`, `RENV_CONFIG_SANDBOX_ENABLED=false`, and `RENV_CONFIG_SYNCHRONIZED_CHECK=false` in your user-level `.Renviron` file. If you disable the synchronization check, remember to call `renv::status()` periodically to check the health of your `renv` project library.

See Also

`https://rstudio.github.io/renv/articles/renv.html`

Other scripts: `tar_edit()`, `tar_github_actions()`, `tar_helper()`, `tar_script()`

Examples

```
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
  tar_script({
    library(targets)
    library(tarchetypes)
    tar_option_set(packages = c("tibble", "qs"))
    list()
  }, ask = FALSE)
  tar_renv()
  writeLines(readLines("_targets_packages.R"))
})
tar_option_reset()
```

`tar_repository_cas` *Define a custom content-addressable storage (CAS) repository (an experimental feature).*

Description

Define a custom storage repository that uses content-addressable storage (CAS).

Usage

```
tar_repository_cas(
  upload,
  download,
  exists = NULL,
  list = NULL,
  consistent = FALSE,
  substitute = base::list()
)
```

Arguments

`upload`	A function with arguments `key` and `path`, in that order. This function should upload the file or directory from `path` to the CAS system. `path` is where the file is originally saved to disk outside the CAS system. It could be a staging area or a custom `format = "file"` location. `key` denotes the name of the destination data object in the CAS system. To differentiate between `format = "file"` targets and non-`"file"` targets, the `upload` method can use `tar_format_get()`. For example, to make `tar_repository_cas_local()` efficient, `upload` moves the file if `targets::tar_format_get() == "file"` and copies it otherwise. See the "Repository functions" section for more details.
`download`	A function with arguments `key` and `path`, in that order. This function should download the data object at `key` from the CAS system to the file or directory at `path`. `key` denotes the name of the data object in the CAS system. `path` is a temporary staging area and not the final destination. Please be careful to avoid deleting the object at `key` from the CAS system. If the CAS system is a local file system, for example, `download` should copy the file and not simply move it (e.g. please avoid `file.rename()`). See the "Repository functions" section for more details.
`exists`	A function with a single argument `key`, where `key` is a single character string (`length(key)` is 1) to identify a single object in the CAS system. The `exists` function should check if there is a single object at a single `key` in the CAS system. It is ignored if `list` is given and `consistent` is `TRUE`. See the "Repository functions" section for more details.
`list`	Either `NULL` or an optional function with a single argument named `keys`. The `list` function increases efficiency by reducing repeated calls to the `exists` function (see above) or entirely avoiding them if `consistent` is ‘TRUE. The `list` function should return a character vector of keys that already exist in the CAS system. The `keys` argument of `list` is a character vector of CAS keys (hashes) which are already recorded in the pipeline metadata (`tar_meta()`). For greater efficiency, the `list` function can restrict its query to these existing keys instead of trying to list the billions of keys that could exist in a CAS system. See the source code of `tar_cas_l()` for an example of how this can work for a local file system CAS. See the "Repository functions" section for more details.
`consistent`	Logical. Set to `TRUE` if the storage platform is strongly read-after-write consistent. Set to `FALSE` if the platform is not necessarily strongly read-after-write consistent. A data storage system is said to have strong read-after-write consistency if a new object is fully available for reading as soon as the write operation finishes. Many modern cloud services like Amazon S3 and Google Cloud Storage have strong

read-after-write consistency, meaning that if you upload an object with a PUT request, then a GET request immediately afterwards will retrieve the precise version of the object you just uploaded.

Some storage systems do not have strong read-after-write consistency. One example is network file systems (NFS). On a computing cluster, if one node creates a file on an NFS, then there is a delay before other nodes can access the new file. `targets` handles this situation by waiting for the new file to appear with the correct hash before attempting to use it in downstream computations. `consistent = FALSE` imposes a waiting period in which `targets` repeatedly calls the `exists` method until the file becomes available (at time intervals configurable with `tar_resources_network()`). These extra calls to `exists` may come with a little extra latency / computational burden, but on systems which are not strongly read-after-write consistent, it is the only way `targets` can safely use the current results for downstream computations.

`substitute` Named list of values to be inserted into the body of each custom function in place of symbols in the body. For example, if `upload = function(key, path) do_upload(key, path, bucket = X)` and `substitute = list(X = "my_aws_bucket")`, then the `upload` function will actually end up being `function(key, path) do_upload(key, path, bucket = "my_aws_bucket")`.

Please do not include temporary or sensitive information such as authentication credentials. If you do, then `targets` will write them to metadata on disk, and a malicious actor could steal and misuse them. Instead, pass sensitive information as environment variables using `tar_resources_repository_cas()`. These environment variables only exist in the transient memory spaces of the R sessions of the local and worker processes.

Content-addressable storage

Normally, `targets` organizes output data based on target names. For example, if a pipeline has a single target `x` with default settings, then `tar_make()` saves the output data to the file `_targets/objects/x`. When the output of `x` changes, `tar_make()` overwrites `_targets/objects/x`. In other words, no matter how many changes happen to `x`, the data store always looks like this:

```
_targets/
    meta/
        meta
    objects/
        x
```

By contrast, with content-addressable storage (CAS), `targets` organizes outputs based on the hashes of their contents. The name of each output file is its hash, and the metadata maps these hashes to target names. For example, suppose target `x` has `repository = tar_repository_cas_local("my_cas")`. When the output of `x` changes, `tar_make()` creates a new file inside `my_cas/` without overwriting or deleting any other files in that folder. If you run `tar_make()` three different times with three different values of `x`, then storage will look like this:

```
_targets/
    meta/
```

```
        meta
my_cas/
    1fffeb09ad36e84a
    68328d833e6361d3
    798af464fb2f6b30
```

The next call to `tar_read(x)` uses `tar_meta(x)$data` to look up the current hash of `x`. If `tar_meta(x)$data` returns `"1fffeb09ad36e84a"`, then `tar_read(x)` returns the data from `my_cas/1fffeb09ad36e84a`. Files `my_cas/68328d833e6361d3` and and `my_cas/798af464fb2f6b30` are left over from previous values of `x`.

Because CAS accumulates historical data objects, it is ideal for data versioning and collaboration. If you commit the `_targets/meta/meta` file to version control alongside the source code, then you can revert to a previous state of your pipeline with all your targets up to date, and a colleague can leverage your hard-won results using a fork of your code and metadata.

The downside of CAS is the cost of accumulating many data objects over time. Most pipelines that use CAS should have a garbage collection system or retention policy to remove data objects when they no longer needed.

The `tar_repository_cas()` function lets you create your own CAS system for `targets`. You can supply arbitrary custom methods to upload, download, and check for the existence of data objects. Your custom CAS system can exist locally on a shared file system or remotely on the cloud (e.g. in an AWS S3 bucket). See the "Repository functions" section and the documentation of individual arguments for advice on how to write your own methods.

The `tar_repository_cas_local()` function has an example CAS system based on a local folder on disk. It uses `tar_cas_u()` for uploads, `tar_cas_d()` for downloads, and `tar_cas_l()` for listing keys.

Repository functions

In `tar_repository_cas()`, functions `upload`, `download`, `exists`, and `keys` must be completely pure and self-sufficient. They must load or namespace all their own packages, and they must not depend on any custom user-defined functions or objects in the global environment of your pipeline. `targets` converts each function to and from text, so it must not rely on any data in the closure. This disqualifies functions produced by `Vectorize()`, for example.

`upload` and `download` can assume `length(path)` is 1, but they should account for the possibility that `path` could be a directory. To simply avoid supporting directories, `upload` could simply call an assertion:

```
targets::tar_assert_not_dir(
  path,
  msg = "This CAS upload method does not support directories."
)
```

Otherwise, support for directories may require handling them as a special case. For example, `upload` and `download` could copy all the files in the given directory, or they could manage the directory as a zip archive.

Some functions may need to be adapted and configured based on other inputs. For example, you may want to define `upload = \(key, path) file.rename(path, file.path(folder, key))` but do not want to hard-code a value of `folder` when you write the underlying function. The `substitute` argument handles this situation. For example, if `substitute` is `list(folder = "my_folder")`, then `upload` will end up as `\(key, path) file.rename(path, file.path("my_folder", key))`.

Temporary or sensitive such as authentication credentials should not be injected this way into the function body. Instead, pass them as environment variables using `tar_resources_repository_cas()`.

See Also

Other content-addressable storage: `tar_repository_cas_local()`, `tar_repository_cas_local_gc()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  repository <- tar_repository_cas(
    upload = function(key, path) {
      if (dir.exists(path)) {
        stop("This CAS repository does not support directory outputs.")
      }
      if (!file.exists("cas")) {
        dir.create("cas", recursive = TRUE)
      }
      file.rename(path, file.path("cas", key))
    },
    download = function(key, path) {
      file.copy(file.path("cas", key), path)
    },
    exists = function(key) {
      file.exists(file.path("cas", key))
    },
    list = function(keys) {
      keys[file.exists(file.path("cas", keys))]
    },
    consistent = FALSE
  )
  write_file <- function(object) {
    writeLines(as.character(object), "file.txt")
    "file.txt"
  }
  list(
    tar_target(x, c(2L, 4L), repository = repository),
    tar_target(
      y,
      x,
      pattern = map(x),
      format = "qs",
      repository = repository
    ),
    tar_target(z, write_file(y), format = "file", repository = repository)
  )
})
tar_make()
tar_read(y)
tar_read(z)
list.files("cas")
tar_meta(any_of(c("x", "z")), fields = any_of("data"))
})
}
```

`tar_repository_cas_local`
Local content-addressable storage (CAS) repository (an experimental feature).

Description

Local content-addressable storage (CAS) repository.

Usage

```
tar_repository_cas_local(path = NULL, consistent = FALSE)
```

Arguments

`path`	Character string, file path to the CAS repository where all the data object files will be stored. `NULL` to default to `file.path(tar_config_get("store"), "cas")` (usually `"_targets/cas/"`).
`consistent`	Logical. Set to `TRUE` if the storage platform is strongly read-after-write consistent. Set to `FALSE` if the platform is not necessarily strongly read-after-write consistent. A data storage system is said to have strong read-after-write consistency if a new object is fully available for reading as soon as the write operation finishes. Many modern cloud services like Amazon S3 and Google Cloud Storage have strong read-after-write consistency, meaning that if you upload an object with a PUT request, then a GET request immediately afterwards will retrieve the precise version of the object you just uploaded. Some storage systems do not have strong read-after-write consistency. One example is network file systems (NFS). On a computing cluster, if one node creates a file on an NFS, then there is a delay before other nodes can access the new file. `targets` handles this situation by waiting for the new file to appear with the correct hash before attempting to use it in downstream computations. `consistent = FALSE` imposes a waiting period in which `targets` repeatedly calls the `exists` method until the file becomes available (at time intervals configurable with `tar_resources_network()`). These extra calls to `exists` may come with a little extra latency / computational burden, but on systems which are not strongly read-after-write consistent, it is the only way `targets` can safely use the current results for downstream computations.

Details

Pass to the `repository` argument of `tar_target()` or `tar_option_set()` to use a local CAS system.

Value

A character string from `tar_repository_cas()` which may be passed to the `repository` argument of `tar_target()` or `tar_option_set()` to use a local CAS system.

Content-addressable storage

Normally, `targets` organizes output data based on target names. For example, if a pipeline has a single target `x` with default settings, then `tar_make()` saves the output data to the file `_targets/objects/x`. When the output of `x` changes, `tar_make()` overwrites `_targets/objects/x`. In other words, no matter how many changes happen to `x`, the data store always looks like this:

```
_targets/
    meta/
        meta
    objects/
        x
```

By contrast, with content-addressable storage (CAS), `targets` organizes outputs based on the hashes of their contents. The name of each output file is its hash, and the metadata maps these hashes to target names. For example, suppose target `x` has `repository = tar_repository_cas_local("my_cas")`. When the output of `x` changes, `tar_make()` creates a new file inside `my_cas/` without overwriting or deleting any other files in that folder. If you run `tar_make()` three different times with three different values of `x`, then storage will look like this:

```
_targets/
    meta/
        meta
my_cas/
    1fffeb09ad36e84a
    68328d833e6361d3
    798af464fb2f6b30
```

The next call to `tar_read(x)` uses `tar_meta(x)$data` to look up the current hash of `x`. If `tar_meta(x)$data` returns `"1fffeb09ad36e84a"`, then `tar_read(x)` returns the data from `my_cas/1fffeb09ad36e84a`. Files `my_cas/68328d833e6361d3` and and `my_cas/798af464fb2f6b30` are left over from previous values of `x`.

Because CAS accumulates historical data objects, it is ideal for data versioning and collaboration. If you commit the `_targets/meta/meta` file to version control alongside the source code, then you can revert to a previous state of your pipeline with all your targets up to date, and a colleague can leverage your hard-won results using a fork of your code and metadata.

The downside of CAS is the cost of accumulating many data objects over time. Most pipelines that use CAS should have a garbage collection system or retention policy to remove data objects when they no longer needed.

The tar_repository_cas() function lets you create your own CAS system for targets. You can supply arbitrary custom methods to upload, download, and check for the existence of data objects. Your custom CAS system can exist locally on a shared file system or remotely on the cloud (e.g. in an AWS S3 bucket). See the "Repository functions" section and the documentation of individual arguments for advice on how to write your own methods.

The tar_repository_cas_local() function has an example CAS system based on a local folder on disk. It uses tar_cas_u() for uploads, tar_cas_d() for downloads, and tar_cas_l() for listing keys.

See Also

Other content-addressable storage: tar_repository_cas(), tar_repository_cas_local_gc()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  repository <- tar_repository_cas_local("cas")
  write_file <- function(object) {
    writeLines(as.character(object), "file.txt")
    "file.txt"
  }
  list(
    tar_target(x, c(2L, 4L), repository = repository),
    tar_target(
      y,
      x,
      pattern = map(x),
      format = "qs",
      repository = repository
    ),
    tar_target(z, write_file(y), format = "file", repository = repository)
  )
})
tar_make()
tar_read(y)
tar_read(z)
list.files("cas")
tar_meta(any_of(c("x", "z")), fields = any_of("data"))
})
}
```

---

tar_repository_cas_local_gc
Local CAS garbage collection

Description

Garbage collection for a local content-addressable storage system.

Usage

```
tar_repository_cas_local_gc(
  path = NULL,
  store = targets::tar_config_get("store")
)
```

Arguments

`path`	Character string, file path to the CAS repository where all the data object files will be stored. `NULL` to default to `file.path(tar_config_get("store"), "cas")` (usually `"_targets/cas/"`).
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

Deletes all the files in the local CAS which are not in `tar_meta(targets_only = TRUE)$data`, including all locally saved historical data of the pipeline. This clears disk space, but at the expense of removing historical data and data from other colleagues who worked on the same project.

Value

`NULL` (invisibly). Called for its side effects. Removes files from the CAS repository at `path`.

Content-addressable storage

Normally, `targets` organizes output data based on target names. For example, if a pipeline has a single target `x` with default settings, then `tar_make()` saves the output data to the file `_targets/objects/x`. When the output of `x` changes, `tar_make()` overwrites `_targets/objects/x`. In other words, no matter how many changes happen to `x`, the data store always looks like this:

```
_targets/
    meta/
        meta
    objects/
        x
```

By contrast, with content-addressable storage (CAS), `targets` organizes outputs based on the hashes of their contents. The name of each output file is its hash, and the metadata maps these hashes to target names. For example, suppose target `x` has `repository = tar_repository_cas_local("my_cas")`. When the output of `x` changes, `tar_make()` creates a new file inside `my_cas/` without overwriting or deleting any other files in that folder. If you run `tar_make()` three different times with three different values of `x`, then storage will look like this:

```
_targets/
    meta/
        meta
my_cas/
    1fffeb09ad36e84a
    68328d833e6361d3
    798af464fb2f6b30
```

The next call to `tar_read(x)` uses `tar_meta(x)$data` to look up the current hash of `x`. If `tar_meta(x)$data` returns `"1fffeb09ad36e84a"`, then `tar_read(x)` returns the data from `my_cas/1fffeb09ad36e84a`. Files `my_cas/68328d833e6361d3` and and `my_cas/798af464fb2f6b30` are left over from previous values of `x`.

Because CAS accumulates historical data objects, it is ideal for data versioning and collaboration. If you commit the `_targets/meta/meta` file to version control alongside the source code, then you can revert to a previous state of your pipeline with all your targets up to date, and a colleague can leverage your hard-won results using a fork of your code and metadata.

The downside of CAS is the cost of accumulating many data objects over time. Most pipelines that use CAS should have a garbage collection system or retention policy to remove data objects when they no longer needed.

The `tar_repository_cas()` function lets you create your own CAS system for `targets`. You can supply arbitrary custom methods to upload, download, and check for the existence of data objects. Your custom CAS system can exist locally on a shared file system or remotely on the cloud (e.g. in an AWS S3 bucket). See the "Repository functions" section and the documentation of individual arguments for advice on how to write your own methods.

The `tar_repository_cas_local()` function has an example CAS system based on a local folder on disk. It uses `tar_cas_u()` for uploads, `tar_cas_d()` for downloads, and `tar_cas_l()` for listing keys.

See Also

Other content-addressable storage: `tar_repository_cas()`, `tar_repository_cas_local()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(seed = NA, repository = tar_repository_cas_local())
  list(tar_target(x, sample.int(n = 9e9, size = 1)))
})
for (index in seq_len(3)) tar_make(reporter = "silent")
list.files("_targets/cas")
tar_repository_cas_local_gc()
list.files("_targets/cas")
tar_meta(names = any_of("x"), fields = any_of("data"))
})
}
```

`tar_reprex` *Reproducible example of* `targets` *with* `reprex`

Description

Create a reproducible example of a `targets` pipeline with the `reprex` package.

Usage

```
tar_reprex(pipeline = tar_target(example_target, 1), run = tar_make(), ...)
```

Arguments

`pipeline`	R code for the target script file `_targets.R`. `library(targets)` is automatically written at the top.
`run`	R code to inspect and run the pipeline.
`...`	Named arguments passed to `reprex::reprex()`.

Details

The best way to get help with an issue is to create a reproducible example of the problem and post it to `https://github.com/ropensci/targets/discussions` `tar_reprex()` facilitates this process. It is like `reprex::reprex({targets::tar_script(...); tar_make()})`, but more convenient.

Value

A character vector of rendered the reprex, invisibly.

See Also

Other help: `targets-package`, `use_targets()`, `use_targets_rmd()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_reprex(
  pipeline = {
    list(
      tar_target(data, data.frame(x = sample.int(1e3))),
      tar_target(summary, mean(data$x, na.rm = TRUE))
    )
  },
  run = {
    tar_visnetwork()
    tar_make()
  }
)
}
```

tar_resources *Target resources*

Description

Create a resources argument for tar_target() or tar_option_set().

Usage

```
tar_resources(
  aws = tar_option_get("resources")$aws,
  clustermq = tar_option_get("resources")$clustermq,
  crew = tar_option_get("resources")$crew,
  custom_format = tar_option_get("resources")$custom_format,
  feather = tar_option_get("resources")$feather,
  fst = tar_option_get("resources")$fst,
  future = tar_option_get("resources")$future,
  gcp = tar_option_get("resources")$gcp,
  network = tar_option_get("resources")$network,
  parquet = tar_option_get("resources")$parquet,
  qs = tar_option_get("resources")$qs,
  repository_cas = tar_option_get("resources")$repository_cas,
  url = tar_option_get("resources")$url
)
```

Arguments

aws	Output of function tar_resources_aws(). Amazon Web Services (AWS) S3 storage settings for tar_target(..., repository = "aws"). See the cloud storage section of https://books.ropensci.org/targets/data.html for details for instructions.
clustermq	Output of function tar_resources_clustermq(). Optional clustermq settings for tar_make_clustermq(), including the log_worker and template arguments of clustermq::workers(). clustermq workers are *persistent*, so there is not a one-to-one correspondence between workers and targets. The clustermq resources apply to the workers, not the targets. So the correct way to assign clustermq resources is through tar_option_set(), not tar_target(). clustermq resources in individual tar_target() calls will be ignored.

crew	Output of function tar_resources_crew() with target-specific settings for integration with the crew R package. These settings are arguments to the push() method of the controller or controller group object which control things like auto-scaling behavior and the controller to use in the case of a controller group.
custom_format	Output of function tar_resources_custom_format() with configuration details for tar_format() storage formats.
feather	Output of function tar_resources_feather(). Non-default arguments to arrow::read_feather() and arrow::write_feather() for arrow/feather-based storage formats. Applies to all formats ending with the "_feather" suffix. For details on formats, see the format argument of tar_target().
fst	Output of function tar_resources_fst(). Non-default arguments to fst::read_fst() and fst::write_fst() for fst-based storage formats. Applies to all formats ending with "fst" in the name. For details on formats, see the format argument of tar_target().
future	Output of function tar_resources_future(). Optional future settings for tar_make_future(), including the resources argument of future::future(), which can include values to insert in template placeholders in future.batchtools template files. This is how to supply the resources argument of future::future() for targets. Resources supplied through future::plan() and future::tweak() are completely ignored.
gcp	Output of function tar_resources_gcp(). Google Cloud Storage bucket settings for tar_target(..., repository = "gcp"). See the cloud storage section of https://books.ropensci.org/targets/data.html for details for instructions.
network	Output of function tar_resources_network(). Settings to configure how to handle unreliable network connections in the case of uploading, downloading, and checking data in situations that rely on network file systems or HTTP/HTTPS requests. Examples include retries and timeouts for internal storage management operations for storage = "worker" or format = "file" (on network file systems), format = "url", repository = "aws", and repository = "gcp". These settings do not apply to actions you take in the custom R command of the target.
parquet	Output of function tar_resources_parquet(). Non-default arguments to arrow::read_parquet() and arrow::write_parquet() for arrow/parquet-based storage formats. Applies to all formats ending with the "_parquet" suffix. For details on formats, see the format argument of tar_target().
qs	Output of function tar_resources_qs(). Non-default arguments to qs2::qs_read() and qs2::qs_save() for targets with format = "qs". For details on formats, see the format argument of tar_target().
repository_cas	Output of function tar_resources_repository_cas() with configuration details for tar_repository_cas() storage repositories.
url	Output of function tar_resources_url(). Non-default settings for storage formats ending with the "_url" suffix. These settings include the curl handle for extra control over HTTP requests. For details on formats, see the format argument of tar_target().

Value

A list of objects of class "tar_resources" with non-default settings of various optional backends for data storage and high-performance computing.

Resources

Functions tar_target() and tar_option_set() each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The tar_resources() function is a helper to supply those settings in the correct manner.

In targets version 0.12.2 and above, resources are inherited one-by-one in nested fashion from tar_option_get("resources"). For example, suppose you set tar_option_set(resources = tar_resources(aws = my_aws)), where my_aws equals tar_resources_aws(bucket = "x", prefix = "y"). Then, tar_target(data, get_data() will have bucket "x" and prefix "y". In addition, if new_resources equals tar_resources(aws = tar_resources_aws(bucket = "z"))), then tar_target(data, get_data(), resources = new_resources) will use the new bucket "z", but it will still use the prefix "y" supplied through tar_option_set(). (In targets 0.12.1 and below, options like prefix do not carry over from tar_option_set() if you supply non-default resources to tar_target().)

See Also

Other resources: tar_resources_aws(), tar_resources_clustermq(), tar_resources_crew(), tar_resources_custom_format(), tar_resources_feather(), tar_resources_fst(), tar_resources_future(), tar_resources_gcp(), tar_resources_network(), tar_resources_parquet(), tar_resources_qs(), tar_resources_repository_cas(), tar_resources_url()

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "qs",
  resources = tar_resources(
    qs = tar_resources_qs(preset = "fast"),
    future = tar_resources_future(resources = list(n_cores = 1))
  )
)
```

---

tar_resources_aws *Target resources: Amazon Web Services (AWS) S3 storage*

Description

Create the aws argument of tar_resources() to specify optional settings to AWS for tar_target(..., repository = "aws"). See the format argument of tar_target() for details.

Usage

```
tar_resources_aws(
  bucket = targets::tar_option_get("resources")$aws$bucket,
  prefix = targets::tar_option_get("resources")$aws$prefix,
  region = targets::tar_option_get("resources")$aws$region,
  endpoint = targets::tar_option_get("resources")$aws$endpoint,
 s3_force_path_style = targets::tar_option_get("resources")$aws$s3_force_path_style,
  part_size = targets::tar_option_get("resources")$aws$part_size,
  page_size = targets::tar_option_get("resources")$aws$page_size,
  max_tries = targets::tar_option_get("resources")$aws$max_tries,
  seconds_timeout = targets::tar_option_get("resources")$aws$seconds_timeout,
  close_connection = targets::tar_option_get("resources")$aws$close_connection,
  verbose = targets::tar_option_get("resources")$aws$verbose,
  ...
)
```

Arguments

`bucket`	Character of length 1, name of an existing bucket to upload and download the return values of the affected targets during the pipeline.
`prefix`	Character of length 1, "directory path" in the bucket where your target object and metadata will go. Please supply an explicit prefix unique to your `targets` project. In the future, `targets` will begin requiring explicitly user-supplied prefixes. (This last note was added on 2023-08-24: `targets` version 1.2.2.9000.)
`region`	Character of length 1, AWS region containing the S3 bucket. Set to `NULL` to use the default region.
`endpoint`	Character of length 1, URL endpoint for S3 storage. Defaults to the Amazon AWS endpoint if `NULL`. Example: To use the S3 protocol with Google Cloud Storage, set `endpoint = "https://storage.googleapis.com"` and `region = "auto"`. (A custom endpoint may require that you explicitly set a custom region directly in `tar_resources_aws()`. `region = "auto"` happens to work with Google Cloud.) Also make sure to create HMAC access keys in the Google Cloud Storage console (under Settings => Interoperability) and set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables accordingly. After that, you should be able to use S3 storage formats with Google Cloud storage buckets. There is one limitation, however: even if your bucket has object versioning turned on, `targets` may fail to record object versions. Google Cloud Storage in particular has this incompatibility.
`s3_force_path_style`	Logical of length 1, whether to use path-style addressing for S3 requests.
`part_size`	Positive numeric of length 1, number of bytes for each part of a multipart upload. (Except the last part, which is the remainder.) In a multipart upload, each part must be at least 5 MB. The default value of the `part_size` argument is `5 * (2 ^ 20)`.
`page_size`	Positive integer of length 1, number of items in each page for paginated HTTP requests such as listing objects.

`max_tries`	Positive integer of length 1, maximum number of attempts to access a network resource on AWS.
`seconds_timeout`	Positive numeric of length 1, number of seconds until an HTTP connection times out.
`close_connection`	Logical of length 1, whether to close HTTP connections immediately.
`verbose`	Logical of length 1, whether to print console messages when running computationally expensive operations such as listing objects in a large bucket.
`...`	Named arguments to functions in `paws.storage::s3()` to manage S3 storage. The documentation of these specific functions is linked from `https://www.paws-r-sdk.com/docs/s3/`. The configurable functions themselves are: • `paws.storage::s3()$head_object()` • `paws.storage::s3()$get_object()` • `paws.storage::s3()$delete_object()` • `paws.storage::s3()$put_object()` • `paws.storage::s3()$create_multipart_upload()` • `paws.storage::s3()$abort_multipart_upload()` • `paws.storage::s3()$complete_multipart_upload()` • `paws.storage::s3()$upload_part()` The named arguments in `...` must not be any of `"bucket"`, `"Bucket"`, `"key"`, `"Key"`, `"prefix"`, `"region"`, `"part_size"`, `"endpoint"`, `"version"`, `"VersionId"`, `"body"`, `"Body"`, `"metadata"`, `"Metadata"`, `"UploadId"`, `"MultipartUpload"`, or `"PartNumber"`.

Details

See the cloud storage section of `https://books.ropensci.org/targets/data.html` for details for instructions.

Value

Object of class `"tar_resources_aws"`, to be supplied to the aws argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix = "y")`. Then, `tar_target(data, get_data()` will have bucket `"x"` and prefix `"y"`. In addition, if `new_resources` equals `tar_resources(aws = tar_resources_aws(bucket = "z")))`, then `tar_target(data, get_data(), resources = new_resources)` will use the new bucket `"z"`, but it will still use the prefix `"y"` supplied through `tar_option_set()`. (In `targets` 0.12.1 and below, options like `prefix` do not carry over from `tar_option_set()` if you supply non-default resources to `tar_target()`.)

See Also

Other resources: `tar_resources()`, `tar_resources_clustermq()`, `tar_resources_crew()`, `tar_resources_custom_fo`
`tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_gcp()`,
`tar_resources_network()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_repository_cas()`,
`tar_resources_url()`

Examples

```
# Somewhere in you target script file (usually _targets.R):
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_target(
  name,
  command(),
  format = "qs",
  repository = "aws",
  resources = tar_resources(
    aws = tar_resources_aws(
      bucket = "yourbucketname",
      prefix = "_targets"
    ),
    qs = tar_resources_qs(preset = "fast"),
  )
)
}
```

tar_resources_clustermq

Target resources: `clustermq` *high-performance computing*

Description

Create the `clustermq` argument of `tar_resources()` to specify optional high-performance computing settings for `tar_make_clustermq()`. For details, see the documentation of the `clustermq` R package and the corresponding argument names in this help file.

Usage

```
tar_resources_clustermq(
  template = targets::tar_option_get("resources")$clustermq$template
)
```

Arguments

`template`	Named list, `template` argument to `clustermq::workers()`. Defaults to an empty list.

Details

clustermq workers are *persistent*, so there is not a one-to-one correspondence between workers and targets. The clustermq resources apply to the workers, not the targets. So the correct way to assign clustermq resources is through tar_option_set(), not tar_target(). clustermq resources in individual tar_target() calls will be ignored.

Value

Object of class "tar_resources_clustermq", to be supplied to the clustermq argument of tar_resources().

Resources

Functions tar_target() and tar_option_set() each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The tar_resources() function is a helper to supply those settings in the correct manner.

In targets version 0.12.2 and above, resources are inherited one-by-one in nested fashion from tar_option_get("resources"). For example, suppose you set tar_option_set(resources = tar_resources(aws = my_aws)), where my_aws equals tar_resources_aws(bucket = "x", prefix = "y"). Then, tar_target(data, get_data() will have bucket "x" and prefix "y". In addition, if new_resources equals tar_resources(aws = tar_resources_aws(bucket = "z"))), then tar_target(data, get_data(), resources = new_resources) will use the new bucket "z", but it will still use the prefix "y" supplied through tar_option_set(). (In targets 0.12.1 and below, options like prefix do not carry over from tar_option_set() if you supply non-default resources to tar_target().)

See Also

Other resources: tar_resources(), tar_resources_aws(), tar_resources_crew(), tar_resources_custom_format() tar_resources_feather(), tar_resources_fst(), tar_resources_future(), tar_resources_gcp(), tar_resources_network(), tar_resources_parquet(), tar_resources_qs(), tar_resources_repository_cas(), tar_resources_url()

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  resources = tar_resources(
    clustermq = tar_resources_clustermq(template = list(n_cores = 2))
  )
)
```

tar_resources_crew *Target resources:* crew *high-performance computing*

Description

Create the crew argument of tar_resources() to specify optional target settings.

Usage

```
tar_resources_crew(
  controller = targets::tar_option_get("resources")$crew$controller,
  scale = NULL,
  seconds_timeout = targets::tar_option_get("resources")$crew$seconds_timeout
)
```

Arguments

controller	Character of length 1. If tar_option_get("controller") is a crew controller group, the controller argument of tar_resources_crew() indicates which controller in the controller group to use. If you need heterogeneous workers, you can leverage this argument to send different targets to different worker groups.
scale	Deprecated in targets version 1.3.0.9002 (2023-10-02). No longer necessary.
seconds_timeout	Positive numeric of length 1, optional task timeout passed to the .timeout argument of mirai::mirai() (after converting to milliseconds).

Details

tar_resources_crew() accepts target-specific settings for integration with the crew R package. These settings are arguments to the push() method of the controller or controller group object which control things like auto-scaling behavior and the controller to use in the case of a controller group.

Value

Object of class "tar_resources_crew", to be supplied to the crew argument of tar_resources().

Resources

Functions tar_target() and tar_option_set() each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The tar_resources() function is a helper to supply those settings in the correct manner.

In targets version 0.12.2 and above, resources are inherited one-by-one in nested fashion from tar_option_get("resources"). For example, suppose you set tar_option_set(resources = tar_resources(aws = my_aws)), where my_aws equals tar_resources_aws(bucket = "x", prefix = "y"). Then, tar_target(data, get_data() will have bucket "x" and prefix "y". In addition, if new_resources equals tar_resources(aws = tar_resources_aws(bucket = "z"))), then tar_target(data, get_data(), resources = new_resources) will use the new bucket "z", but it will still use the prefix "y" supplied through tar_option_set(). (In targets 0.12.1 and below, options like prefix do not carry over from tar_option_set() if you supply non-default resources to tar_target().)

See Also

Other resources: tar_resources(), tar_resources_aws(), tar_resources_clustermq(), tar_resources_custom_for
tar_resources_feather(), tar_resources_fst(), tar_resources_future(), tar_resources_gcp(), tar_resources_network(), tar_resources_parquet(), tar_resources_qs(), tar_resources_repository_cas(), tar_resources_url()

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  resources = tar_resources(
    crew = tar_resources_crew(seconds_timeout = 5)
  )
)
```

---

tar_resources_custom_format
Target resources for custom storage formats

Description

Create the custom_format argument of tar_resources() to specify optional target settings for custom storage formats.

Usage

```
tar_resources_custom_format(
  envvars = targets::tar_option_get("resources")$custom_format$envvars
)
```

Arguments

envvars	Named character vector of environment variables. These environment variables are temporarily set just before each call to the storage methods you define in tar_format(). Specific methods like read can retrieve values from these environment variables using Sys.getenv(). Set envvars to NULL to omit entirely.

Details

tar_resources_custom_format() accepts target-specific settings to customize tar_format() storage formats.

Value

Object of class "tar_resources_custom_format", to be supplied to the custom_format argument of tar_resources().

Resources

Functions tar_target() and tar_option_set() each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The tar_resources() function is a helper to supply those settings in the correct manner.

In targets version 0.12.2 and above, resources are inherited one-by-one in nested fashion from tar_option_get("resources"). For example, suppose you set tar_option_set(resources = tar_resources(aws = my_aws)), where my_aws equals tar_resources_aws(bucket = "x", prefix = "y"). Then, tar_target(data, get_data() will have bucket "x" and prefix "y". In addition, if new_resources equals tar_resources(aws = tar_resources_aws(bucket = "z"))), then tar_target(data, get_data(), resources = new_resources) will use the new bucket "z", but it will still use the prefix "y" supplied through tar_option_set(). (In targets 0.12.1 and below, options like prefix do not carry over from tar_option_set() if you supply non-default resources to tar_target().)

See Also

Other resources: tar_resources(), tar_resources_aws(), tar_resources_clustermq(), tar_resources_crew(), tar_resources_feather(), tar_resources_fst(), tar_resources_future(), tar_resources_gcp(), tar_resources_network(), tar_resources_parquet(), tar_resources_qs(), tar_resources_repository_cas(), tar_resources_url()

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name = target_name,
  command = data.frame(x = 1),
  format = tar_format(
    read = function(path) {
      readRDS(file = path)
    },
    write = function(object, path) {
      version <- as.integer(Sys.getenv("SERIALIZATION", unset = "2"))
      saveRDS(object = object, file = path, version = version)
    }
  ),
  resources = tar_resources(
    custom_format = tar_resources_custom_format(
      envvars = c(SERIALIZATION = "3")
    )
  )
)
```

tar_resources_feather *Target resources: feather storage formats*

Description

Create the feather argument of `tar_resources()` to specify optional settings for feather data frame storage formats powered by the `arrow` R package. See the `format` argument of `tar_target()` for details.

Usage

```
tar_resources_feather(
  compression = targets::tar_option_get("resources")$feather$compression,
  compression_level = targets::tar_option_get("resources")$feather$compression_level
)
```

Arguments

`compression`	Character of length 1, `compression` argument of `arrow::write_feather()`. Defaults to `"default"`.
`compression_level`	Numeric of length 1, `compression_level` argument of `arrow::write_feather()`. Defaults to `NULL`.

Value

Object of class `"tar_resources_feather"`, to be supplied to the feather argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix = "y")`. Then, `tar_target(data, get_data()` will have bucket `"x"` and prefix `"y"`. In addition, if `new_resources` equals `tar_resources(aws = tar_resources_aws(bucket = "z")))`, then `tar_target(data, get_data(), resources = new_resources)` will use the new bucket `"z"`, but it will still use the prefix `"y"` supplied through `tar_option_set()`. (In `targets` 0.12.1 and below, options like `prefix` do not carry over from `tar_option_set()` if you supply non-default resources to `tar_target()`.)

See Also

Other resources: `tar_resources()`, `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_crew()`, `tar_resources_custom_format()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_gcp()`, `tar_resources_network()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_repository_cas()`, `tar_resources_url()`

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "feather",
  resources = tar_resources(
    feather = tar_resources_feather(compression = "lz4")
  )
)
```

`tar_resources_fst` *Target resources:* `fst` *storage formats*

Description

Create the `fst` argument of `tar_resources()` to specify optional settings for big data frame storage formats powered by the `fst` R package. See the `format` argument of `tar_target()` for details.

Usage

```
tar_resources_fst(compress = targets::tar_option_get("resources")$fst$compress)
```

Arguments

`compress`	Numeric of length 1, `compress` argument of `fst::write_fst()`. Defaults to `50`.

Value

Object of class `"tar_resources_fst"`, to be supplied to the `fst` argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix`

= "y"). Then, tar_target(data, get_data() will have bucket "x" and prefix "y". In addition, if new_resources equals tar_resources(aws = tar_resources_aws(bucket = "z"))), then tar_target(data, get_data(), resources = new_resources) will use the new bucket "z", but it will still use the prefix "y" supplied through tar_option_set(). (In targets 0.12.1 and below, options like prefix do not carry over from tar_option_set() if you supply non-default resources to tar_target().)

See Also

Other resources: tar_resources(), tar_resources_aws(), tar_resources_clustermq(), tar_resources_crew(), tar_resources_custom_format(), tar_resources_feather(), tar_resources_future(), tar_resources_gcp(), tar_resources_network(), tar_resources_parquet(), tar_resources_qs(), tar_resources_repository_cas(), tar_resources_url()

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "fst_tbl",
  resources = tar_resources(
    fst = tar_resources_fst(compress = 100)
  )
)
```

---

tar_resources_future *Target resources:* future *high-performance computing*

Description

Create the future argument of tar_resources() to specify optional high-performance computing settings for tar_make_future(). This is how to supply the resources argument of future::future() for targets. Resources supplied through future::plan() and future::tweak() are completely ignored. For details, see the documentation of the future R package and the corresponding argument names in this help file.

Usage

```
tar_resources_future(
  plan = NULL,
  resources = targets::tar_option_get("resources")$future$resources
)
```

Arguments

plan	A future::plan() object or NULL, a target-specific future plan. Defaults to NULL.
resources	Named list, resources argument to future::future(). This argument is not supported in some versions of future. For versions of future where resources is not supported, instead supply resources to future::tweak() and assign the returned plan to the plan argument of tar_resources_future(). The default value of resources in tar_resources_future() is an empty list.

Value

Object of class "tar_resources_future", to be supplied to the future argument of tar_resources().

Resources

Functions tar_target() and tar_option_set() each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The tar_resources() function is a helper to supply those settings in the correct manner.

In targets version 0.12.2 and above, resources are inherited one-by-one in nested fashion from tar_option_get("resources"). For example, suppose you set tar_option_set(resources = tar_resources(aws = my_aws)), where my_aws equals tar_resources_aws(bucket = "x", prefix = "y"). Then, tar_target(data, get_data() will have bucket "x" and prefix "y". In addition, if new_resources equals tar_resources(aws = tar_resources_aws(bucket = "z"))), then tar_target(data, get_data(), resources = new_resources) will use the new bucket "z", but it will still use the prefix "y" supplied through tar_option_set(). (In targets 0.12.1 and below, options like prefix do not carry over from tar_option_set() if you supply non-default resources to tar_target().)

See Also

Other resources: tar_resources(), tar_resources_aws(), tar_resources_clustermq(), tar_resources_crew(), tar_resources_custom_format(), tar_resources_feather(), tar_resources_fst(), tar_resources_gcp(), tar_resources_network(), tar_resources_parquet(), tar_resources_qs(), tar_resources_repository_cas(), tar_resources_url()

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  resources = tar_resources(
    future = tar_resources_future(resources = list(n_cores = 2))
  )
)
```

tar_resources_gcp — *Target resources: Google Cloud Platform (GCP) Google Cloud Storage (GCS)*

Description

Create the `gcp` argument of `tar_resources()` to specify optional settings for Google Cloud Storage for targets with `tar_target(..., repository = "gcp")`. See the `format` argument of `tar_target()` for details.

Usage

```
tar_resources_gcp(
  bucket = targets::tar_option_get("resources")$gcp$bucket,
  prefix = targets::tar_option_get("resources")$gcp$prefix,
  predefined_acl = targets::tar_option_get("resources")$gcp$predefined_acl,
  max_tries = targets::tar_option_get("resources")$gcp$max_tries,
  verbose = targets::tar_option_get("resources")$gcp$verbose
)
```

Arguments

`bucket`	Character of length 1, name of an existing bucket to upload and download the return values of the affected targets during the pipeline.
`prefix`	Character of length 1, "directory path" in the bucket where your target object and metadata will go. Please supply an explicit prefix unique to your `targets` project. In the future, `targets` will begin requiring explicitly user-supplied prefixes. (This last note was added on 2023-08-24: `targets` version 1.2.2.9000.)
`predefined_acl`	Character of length 1, user access to the object. See `?googleCloudStorageR::gcs_upload` for possible values. Defaults to `"private"`.
`max_tries`	Positive integer of length 1, number of tries accessing a network resource on GCP.
`verbose`	Logical of length 1, whether to print extra messages like progress bars during uploads and downloads. Defaults to `FALSE`.

Details

See the cloud storage section of `https://books.ropensci.org/targets/data.html` for details for instructions.

Value

Object of class `"tar_resources_gcp"`, to be supplied to the `gcp` argument of `tar_resources()`.

Resources

Functions tar_target() and tar_option_set() each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The tar_resources() function is a helper to supply those settings in the correct manner.

In targets version 0.12.2 and above, resources are inherited one-by-one in nested fashion from tar_option_get("resources"). For example, suppose you set tar_option_set(resources = tar_resources(aws = my_aws)), where my_aws equals tar_resources_aws(bucket = "x", prefix = "y"). Then, tar_target(data, get_data() will have bucket "x" and prefix "y". In addition, if new_resources equals tar_resources(aws = tar_resources_aws(bucket = "z"))), then tar_target(data, get_data(), resources = new_resources) will use the new bucket "z", but it will still use the prefix "y" supplied through tar_option_set(). (In targets 0.12.1 and below, options like prefix do not carry over from tar_option_set() if you supply non-default resources to tar_target().)

See Also

Other resources: tar_resources(), tar_resources_aws(), tar_resources_clustermq(), tar_resources_crew(), tar_resources_custom_format(), tar_resources_feather(), tar_resources_fst(), tar_resources_future(), tar_resources_network(), tar_resources_parquet(), tar_resources_qs(), tar_resources_repository_cas(), tar_resources_url()

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "qs",
  repository = "gcp",
  resources = tar_resources(
    gcp = tar_resources_gcp(
      bucket = "yourbucketname",
      prefix = "_targets"
    ),
    qs = tar_resources_qs(preset = "fast"),
  )
)
```

---

tar_resources_network *Target resources for network file systems.*

Description

In high-performance computing on network file systems, if storage = "worker" in tar_target() or tar_option_set(), then targets waits for hashes to synchronize before continuing the pipeline. These resources control the retry mechanism.

Usage

```
tar_resources_network(
  max_tries = targets::tar_option_get("resources")$network$max_tries,
  seconds_interval = targets::tar_option_get("resources")$network$seconds_interval,
  seconds_timeout = targets::tar_option_get("resources")$network$seconds_timeout,
  verbose = targets::tar_option_get("resources")$network$verbose
)
```

Arguments

`max_tries`	Positive integer of length 1. Max number of tries.
`seconds_interval`	Positive numeric of length 1, seconds between retries.
`seconds_timeout`	Positive numeric of length 1. Timeout length in seconds.
`verbose`	Logical of length 1, whether to print informative console messages.

Value

Object of class `"tar_resources_network"`, to be supplied to the network argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix = "y")`. Then, `tar_target(data, get_data()` will have bucket `"x"` and prefix `"y"`. In addition, if `new_resources` equals `tar_resources(aws = tar_resources_aws(bucket = "z")))`, then `tar_target(data, get_data(), resources = new_resources)` will use the new bucket `"z"`, but it will still use the prefix `"y"` supplied through `tar_option_set()`. (In `targets` 0.12.1 and below, options like `prefix` do not carry over from `tar_option_set()` if you supply non-default resources to `tar_target()`.)

See Also

Other resources: `tar_resources()`, `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_crew()`, `tar_resources_custom_format()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_gcp()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_repository_cas()`, `tar_resources_url()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
# Somewhere in you target script file (usually _targets.R):
```

```
tar_target(
  name = your_name,
  command = your_command(),
  storage = "worker",
  resources = tar_resources(
    network = tar_resources_network(max_tries = 3)
  )
)
}
```

---

tar_resources_parquet *Target resources: parquet storage formats*

Description

Create the parquet argument of `tar_resources()` to specify optional settings for parquet data frame storage formats powered by the `arrow` R package. See the `format` argument of `tar_target()` for details.

Usage

```
tar_resources_parquet(
   compression = targets::tar_option_get("resources")$parquet$compression,
  compression_level = targets::tar_option_get("resources")$parquet$compression_level
)
```

Arguments

`compression`	Character of length 1, `compression` argument of `arrow::write_parquet()`. Defaults to `"snappy"`.
`compression_level`	Numeric of length 1, `compression_level` argument of `arrow::write_parquet()`. Defaults to `NULL`.

Value

Object of class `"tar_resources_parquet"`, to be supplied to the parquet argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix = "y")`. Then, `tar_target(data, get_data()` will have bucket `"x"` and prefix `"y"`. In addition, if `new_resources` equals `tar_resources(aws = tar_resources_aws(bucket = "z")))`, then `tar_target(data, get_data(), resources = new_resources)` will use the new bucket `"z"`, but it will still use the prefix `"y"` supplied through `tar_option_set()`. (In `targets` 0.12.1 and below, options like `prefix` do not carry over from `tar_option_set()` if you supply non-default resources to `tar_target()`.)

See Also

Other resources: `tar_resources()`, `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_crew()`, `tar_resources_custom_format()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_gcp()`, `tar_resources_network()`, `tar_resources_qs()`, `tar_resources_repository_cas()`, `tar_resources_url()`

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "parquet",
  resources = tar_resources(
    parquet = tar_resources_parquet(compression = "lz4")
  )
)
```

---

tar_resources_qs *Target resources: qs storage formats*

Description

Create the `qs` argument of `tar_resources()` to specify optional settings for big data storage formats powered by the qs R package. See the `format` argument of `tar_target()` for details.

Usage

```
tar_resources_qs(
  compress_level = targets::tar_option_get("resources")$qs$compress_level,
  shuffle = targets::tar_option_get("resources")$qs$shuffle,
  nthreads = targets::tar_option_get("resources")$qs$nthreads,
  preset = NULL
)
```

Arguments

`compress_level`	Positive integer, `compress_level` argument of `qs2::qs_save()` to control the compression level.
`shuffle`	`TRUE` to use byte shuffling in `qs2::qs_save()` to improve compression at the cost of some computation time, `FALSE` to forgo byte shuffling.

`nthreads`	Positive integer, number of threads to use for functions in the `qs2` package to save and read the data.
`preset`	Deprecated in `targets` version 1.8.0.9014 (2024-11-11) and not used.

Value

Object of class `"tar_resources_qs"`, to be supplied to the qs argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix = "y")`. Then, `tar_target(data, get_data()` will have bucket `"x"` and prefix `"y"`. In addition, if `new_resources` equals `tar_resources(aws = tar_resources_aws(bucket = "z")))`, then `tar_target(data, get_data(), resources = new_resources)` will use the new bucket `"z"`, but it will still use the prefix `"y"` supplied through `tar_option_set()`. (In `targets` 0.12.1 and below, options like `prefix` do not carry over from `tar_option_set()` if you supply non-default resources to `tar_target()`.)

See Also

Other resources: `tar_resources()`, `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_crew()`, `tar_resources_custom_format()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_gcp()`, `tar_resources_network()`, `tar_resources_parquet()`, `tar_resources_repository_cas()`, `tar_resources_url()`

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "qs",
  resources = tar_resources(
    qs = tar_resources_qs(preset = "fast")
  )
)
```

tar_resources_repository_cas

Target resources for custom storage formats

Description

Create the `repository_cas` argument of `tar_resources()` to specify optional target settings for custom storage formats.

Usage

```
tar_resources_repository_cas(
  envvars = targets::tar_option_get("resources")$repository_cas$envvars
)
```

Arguments

`envvars`	Named character vector of environment variables. These environment variables are temporarily set just before each call to the storage methods you define in `tar_format()`. Specific methods like `read` can retrieve values from these environment variables using `Sys.getenv()`. Set `envvars` to `NULL` to omit entirely.

Details

`tar_resources_repository_cas()` accepts target-specific settings to customize `tar_repository_cas()` storage repositories.

Value

Object of class `"tar_resources_repository_cas"`, to be supplied to the `repository_cas` argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix = "y")`. Then, `tar_target(data, get_data()` will have bucket `"x"` and prefix `"y"`. In addition, if `new_resources` equals `tar_resources(aws = tar_resources_aws(bucket = "z")))`, then `tar_target(data, get_data(), resources = new_resources)` will use the new bucket `"z"`, but it will still use the prefix `"y"` supplied through `tar_option_set()`. (In `targets` 0.12.1 and below, options like `prefix` do not carry over from `tar_option_set()` if you supply non-default resources to `tar_target()`.)

See Also

Other resources: `tar_resources()`, `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_crew()`, `tar_resources_custom_format()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_gcp()`, `tar_resources_network()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`

Examples

```
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name = target_name,
  command = data.frame(x = 1),
  repository = tar_repository_cas(
    upload = function(key, path) {
      if (dir.exists(path)) {
        stop("This CAS repository does not support directory outputs.")
      }
      if (!file.exists("cas")) {
        dir.create("cas", recursive = TRUE)
      }
      file.copy(path, file.path("cas", key))
    },
    download = function(key, path) {
      file.copy(file.path("cas", key), path)
    },
    exists = function(key) {
      file.exists(file.path("cas", key))
    }
  ),
  resources = tar_resources(
    repository_cas = tar_resources_repository_cas(
      envvars = c(AUTHENTICATION_CREDENTIALS = "...")
    )
  )
)
```

`tar_resources_url` *Target resources: URL storage formats*

Description

Create the `url` argument of `tar_resources()` to specify optional settings for URL storage formats. See the `format` argument of `tar_target()` for details.

Usage

```
tar_resources_url(
  handle = targets::tar_option_get("resources")$url$handle,
  max_tries = targets::tar_option_get("resources")$url$max_tries,
  seconds_interval = targets::tar_option_get("resources")$url$seconds_interval,
  seconds_timeout = targets::tar_option_get("resources")$url$seconds_interval
)
```

Arguments

`handle`	Object returned by `curl::new_handle` or `NULL`. Defaults to `NULL`.
`max_tries`	Positive integer of length 1, maximum number of tries to access a URL.
`seconds_interval`	Nonnegative numeric of length 1, number of seconds to wait between individual retries while attempting to connect to the URL. Use `tar_resources_network()` instead.
`seconds_timeout`	Nonnegative numeric of length 1, number of seconds to wait before timing out while trying to connect to the URL. Use `tar_resources_network()` instead.

Value

Object of class `"tar_resources_url"`, to be supplied to the url argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner.

In `targets` version 0.12.2 and above, resources are inherited one-by-one in nested fashion from `tar_option_get("resources")`. For example, suppose you set `tar_option_set(resources = tar_resources(aws = my_aws))`, where `my_aws` equals `tar_resources_aws(bucket = "x", prefix = "y")`. Then, `tar_target(data, get_data()` will have bucket `"x"` and prefix `"y"`. In addition, if `new_resources` equals `tar_resources(aws = tar_resources_aws(bucket = "z")))`, then `tar_target(data, get_data(), resources = new_resources)` will use the new bucket `"z"`, but it will still use the prefix `"y"` supplied through `tar_option_set()`. (In `targets` 0.12.1 and below, options like `prefix` do not carry over from `tar_option_set()` if you supply non-default resources to `tar_target()`.)

See Also

Other resources: `tar_resources()`, `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_crew()`, `tar_resources_custom_format()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_gcp()`, `tar_resources_network()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_repository_cas()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "url",
  resources = tar_resources(
    url = tar_resources_url(handle = curl::new_handle())
  )
)
}
```

---

tar_script *Write a target script file.*

Description

The `tar_script()` function is a convenient way to create the required target script file (default: `_targets.R`) in the current working directory. It always overwrites the existing target script, and it requires you to be in the working directory where you intend to write the file, so be careful. See the "Target script" section for details.

Usage

```
tar_script(
  code = NULL,
  library_targets = TRUE,
  ask = NULL,
  script = targets::tar_config_get("script")
)
```

Arguments

code	R code to write to the target script file. If `NULL`, an example target script file is written instead.
library_targets	logical, whether to write a `library(targets)` line at the top of the target script file automatically (recommended). If `TRUE`, you do not need to explicitly put `library(targets)` in `code`.
ask	Logical, whether to ask before writing if the target script file already exists. If `NULL`, defaults to `Sys.getenv("TAR_ASK")`. (Set to `"true"` or `"false"` with `Sys.setenv()`). If `ask` and the `TAR_ASK` environment variable are both indeterminate, defaults to `interactive()`.
script	Character of length 1, where to write the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`.

Value

`NULL` (invisibly).

Target script file

Every targets project requires a target script file. The target script file is usually a file called _targets.R Functions tar_make() and friends look for the target script and run it to set up the pipeline just prior to the main task. Every target script file should run the following steps in the order below:

1. Package: load the targets package. This step is automatically inserted at the top of the target script file produced by tar_script() if library_targets is TRUE, so you do not need to explicitly include it in code.
2. Globals: load custom functions and global objects into memory. Usually, this section is a bunch of calls to source() that run scripts defining user-defined functions. These functions support the R commands of the targets.
3. Options: call tar_option_set() to set defaults for targets-specific settings such as the names of required packages. Even if you have no specific options to set, it is still recommended to call tar_option_set() in order to register the proper environment.
4. Targets: define one or more target objects using tar_target().
5. Pipeline: call list() to aggregate the targets from (4) into a list. Every target script file must return a pipeline object, which usually means ending with a call to list(). In practice, (4) and (5) can be combined together in the same function call.

See Also

Other scripts: tar_edit(), tar_github_actions(), tar_helper(), tar_renv()

Examples

```
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script() # Writes an example target script file.
# Writes a user-defined target script:
tar_script({
  library(targets)
  library(tarchetypes)
  x <- tar_target(x, 1 + 1)
  tar_option_set()
  list(x)
}, ask = FALSE)
writeLines(readLines("_targets.R"))
})
```

tar_seed_create *Create a seed for a target.*

Description

Create a seed for a target.

Usage

```
tar_seed_create(name, global_seed = NULL)
```

Arguments

`name`	Character of length 1, target name.
`global_seed`	Integer of length 1, the overarching global pipeline seed which governs the seeds of all the targets. Set to `NULL` to default to `tar_option_get("seed")`. Set to `NA` to disable seed setting in `targets` and make `tar_seed_create()` return `NA_integer_`.

Value

Integer of length 1, the target seed.

Seeds

A target's random number generator seed is a deterministic function of its name and the global pipeline seed from `tar_option_get("seed")`. Consequently,

```
1. Each target runs with a reproducible seed so that
   different runs of the same pipeline in the same computing
   environment produce identical results.
2. No two targets in the same pipeline share the same seed.
   Even dynamic branches have different names and thus different seeds.
```

You can retrieve the seed of a completed target with `tar_meta(your_target, seed)` and run `tar_seed_set()` on the result to locally recreate the target's initial RNG state. `tar_workspace()` does this automatically as part of recovering a workspace.

RNG overlap

In theory, there is a risk that the pseudo-random number generator streams of different targets will overlap and produce statistically correlated results. (For a discussion of the motivating problem, see the Section 6: "Random-number generation" in the `parallel` package vignette: `vignette(topic = "parallel", package = "parallel")`.) However, this risk is extremely small in practice, as shown by L'Ecuyer et al. (2017) doi:10.1016/j.matcom.2016.05.005 under "A single RNG with a 'random' seed for each stream" (Section 4: under "How to produce parallel streams and substreams"). `targets` and `tarchetypes` take the approach discussed in the aforementioned section of the paper using the `secretbase` package by Charlie Gao (2024) doi:10.5281/zenodo.10553140. To generate the 32-bit integer seed argument of `set.seed()` for each target, `secretbase` generates a cryptographic hash using the SHAKE256 extendable output function (XOF). `secretbase` uses algorithms from the `Mbed TLS` C library.

References

- Gao C (2024). `secretbase`: Cryptographic Hash and Extendable-Output Functions. R package version 0.1.0, doi:10.5281/zenodo.10553140.

- Pierre L'Ecuyer, David Munger, Boris Oreshkin, and Richard Simard (2017). Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. Mathematics and Computers in Simulation, 135, 3-17. doi:10.1016/j.matcom.2016.05.005.

See Also

Other pseudo-random number generation: `tar_seed_get()`, `tar_seed_set()`

`tar_seed_get` *Get the random number generator seed of the target currently running.*

Description

Get the random number generator seed of the target currently running.

Usage

```
tar_seed_get(default = 1L)
```

Arguments

`default`	Integer, value to return if `tar_seed_get()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.

Value

Integer of length 1. If invoked inside a `targets` pipeline, the return value is the seed of the target currently running, which is a deterministic function of the target name. Otherwise, the return value is `default`.

Seeds

A target's random number generator seed is a deterministic function of its name and the global pipeline seed from `tar_option_get("seed")`. Consequently,

```
1. Each target runs with a reproducible seed so that
   different runs of the same pipeline in the same computing
   environment produce identical results.
2. No two targets in the same pipeline share the same seed.
   Even dynamic branches have different names and thus different seeds.
```

You can retrieve the seed of a completed target with `tar_meta(your_target, seed)` and run `tar_seed_set()` on the result to locally recreate the target's initial RNG state. `tar_workspace()` does this automatically as part of recovering a workspace.

RNG overlap

In theory, there is a risk that the pseudo-random number generator streams of different targets will overlap and produce statistically correlated results. (For a discussion of the motivating problem, see the Section 6: "Random-number generation" in the `parallel` package vignette: `vignette(topic = "parallel", package = "parallel")`.) However, this risk is extremely small in practice, as shown by L'Ecuyer et al. (2017) doi:10.1016/j.matcom.2016.05.005 under "A single RNG with a 'random' seed for each stream" (Section 4: under "How to produce parallel streams and substreams"). `targets` and `tarchetypes` take the approach discussed in the aforementioned section of the paper using the `secretbase` package by Charlie Gao (2024) doi:10.5281/zenodo.10553140. To generate the 32-bit integer `seed` argument of `set.seed()` for each target, `secretbase` generates a cryptographic hash using the SHAKE256 extendable output function (XOF). `secretbase` uses algorithms from the `Mbed TLS` C library.

References

- Gao C (2024). `secretbase`: Cryptographic Hash and Extendable-Output Functions. R package version 0.1.0, doi:10.5281/zenodo.10553140.
- Pierre L'Ecuyer, David Munger, Boris Oreshkin, and Richard Simard (2017). Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. Mathematics and Computers in Simulation, 135, 3-17. doi:10.1016/j.matcom.2016.05.005.

See Also

Other pseudo-random number generation: `tar_seed_create()`, `tar_seed_set()`

Examples

```
tar_seed_get()
tar_seed_get(default = 123L)
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(tar_target(returns_seed, tar_seed_get()), ask = FALSE)
tar_make()
tar_read(returns_seed)
})
}
```

`tar_seed_set` *Set a seed to run a target.*

Description

`targets` generates its own target-specific seeds using `tar_seed_create()`. Use `tar_seed_set()` to set one of these seeds in R.

Usage

```
tar_seed_set(seed)
```

Arguments

`seed`	Integer of length 1, value of the seed to set with `set.seed()`.

Details

`tar_seed_set()` gives the user-supplied seed to `set.seed()` and sets arguments `kind = "default"`, `normal.kind = "default"`, and `sample.kind = "default"`.

Value

`NULL` (invisibly).

Seeds

A target's random number generator seed is a deterministic function of its name and the global pipeline seed from `tar_option_get("seed")`. Consequently,

```
1. Each target runs with a reproducible seed so that
   different runs of the same pipeline in the same computing
   environment produce identical results.
2. No two targets in the same pipeline share the same seed.
   Even dynamic branches have different names and thus different seeds.
```

You can retrieve the seed of a completed target with `tar_meta(your_target, seed)` and run `tar_seed_set()` on the result to locally recreate the target's initial RNG state. `tar_workspace()` does this automatically as part of recovering a workspace.

RNG overlap

In theory, there is a risk that the pseudo-random number generator streams of different targets will overlap and produce statistically correlated results. (For a discussion of the motivating problem, see the Section 6: "Random-number generation" in the `parallel` package vignette: `vignette(topic = "parallel", package = "parallel")`.) However, this risk is extremely small in practice, as shown by L'Ecuyer et al. (2017) doi:10.1016/j.matcom.2016.05.005 under "A single RNG with a 'random' seed for each stream" (Section 4: under "How to produce parallel streams and substreams"). `targets` and `tarchetypes` take the approach discussed in the aforementioned section of the paper using the `secretbase` package by Charlie Gao (2024) doi:10.5281/zenodo.10553140. To generate the 32-bit integer seed argument of `set.seed()` for each target, `secretbase` generates a cryptographic hash using the SHAKE256 extendable output function (XOF). `secretbase` uses algorithms from the `Mbed TLS` C library.

References

- Gao C (2024). `secretbase`: Cryptographic Hash and Extendable-Output Functions. R package version 0.1.0, doi:10.5281/zenodo.10553140.
- Pierre L'Ecuyer, David Munger, Boris Oreshkin, and Richard Simard (2017). Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. Mathematics and Computers in Simulation, 135, 3-17. doi:10.1016/j.matcom.2016.05.005.

See Also

Other pseudo-random number generation: tar_seed_create(), tar_seed_get()

Examples

```
seed <- tar_seed_create("target_name")
seed
sample(10)
tar_seed_set(seed)
sample(10)
tar_seed_set(seed)
sample(10)
```

---

tar_sitrep   *Show the cue-by-cue status of each target.*

Description

For each target, report which cues are activated. Except for the never cue, the target will rerun in tar_make() if any cue is activated. The target is suppressed if the never cue is TRUE. See tar_cue() for details.

Usage

```
tar_sitrep(
  names = NULL,
  fields = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  reporter = targets::tar_config_get("reporter_outdated"),
  seconds_reporter = targets::tar_config_get("seconds_reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

names	Optional, names of the targets. If supplied, tar_sitrep() only returns metadata on these targets. The object supplied to names should be NULL or a tidyselect expression like any_of() or starts_with() from tidyselect itself, or tar_described_as() to select target names based on their descriptions.
fields	Optional, names of columns/fields to select. If supplied, tar_sitrep() only returns the selected metadata columns. You can supply symbols or tidyselect helpers like any_of() and starts_with(). The name column is always included first no matter what you select. Choices:

- name: name of the target or global object.
- meta: Whether the meta cue is activated: TRUE if the target is not in the metadata (tar_meta()), or if the target errored during the last tar_make(), or if the class of the target changed.
- always: Whether mode in tar_cue() is "always". If TRUE, tar_make() always runs the target.
- never: Whether mode in tar_cue() is "never". If TRUE, tar_make() will only run if the meta cue activates.
- command: Whether the target's command changed since last time. Always TRUE if the meta cue is activated. Otherwise, always FALSE if the command cue is suppressed.
- depend: Whether the data/output of at least one of the target's dependencies changed since last time. Dependencies are targets, functions, and global objects directly upstream. Call tar_outdated(targets_only = FALSE) or tar_visnetwork(targets_only = FALSE) to see exactly which dependencies are outdated. Always NA if the meta cue is activated. Otherwise, always FALSE if the depend cue is suppressed.
- format: Whether the storage format of the target is different from last time. Always NA if the meta cue is activated. Otherwise, always FALSE if the format cue is suppressed.
- repository: Whether the storage repository of the target is different from last time. Always NA if the meta cue is activated. Otherwise, always FALSE if the format cue is suppressed.
- iteration: Whether the iteration mode of the target is different from last time. Always NA if the meta cue is activated. Otherwise, always FALSE if the iteration cue is suppressed.
- file: Whether the file(s) with the target's return value are missing or different from last time. Always NA if the meta cue is activated. Otherwise, always FALSE if the file cue is suppressed.

shortcut Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. If TRUE, then the function only checks the targets in names and uses stored metadata for information about upstream dependencies as needed. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Use with caution. shortcut = TRUE only works if you set names.

reporter Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: * "forecast_interactive" (default): use the forecast reporter if the session is interactive (see base::interactive()), otherwise use the silent reporter. * "silent": print nothing. * "forecast": print running totals of the checked and outdated targets found so far.

seconds_reporter

Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is `max(1, seconds_reporter)` to reduce overhead.

`callr_function` A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.

`callr_arguments`
A list of arguments to `callr_function`.

`envir` An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc.

The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

`script` Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

`store` Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

Caveats:

- `tar_cue()` allows you to change/suppress cues, so the return value will depend on the settings you supply to `tar_cue()`.
- If a pattern tries to branches over a target that does not exist in storage, then the branches are omitted from the output.
- `tar_sitrep()` is myopic. It only considers what happens to the immediate target and its immediate upstream dependencies, and it makes no attempt to propagate invalidation downstream.

Value

A data frame with one row per target/object and one column per cue. Each element is a logical to indicate whether the cue is activated for the target. See the `field` argument in this help file for details.

See Also

Other inspect: `tar_deps()`, `tar_manifest()`, `tar_network()`, `tar_outdated()`, `tar_validate()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_sitrep()
tar_meta(starts_with("y_")) # see also any_of()
})
}
```

`tar_skipped` *List skipped targets.*

Description

List targets whose progress is `"skipped"`.

Usage

```
tar_skipped(names = NULL, store = targets::tar_config_get("store"))
```

Arguments

`names`	Optional, names of the targets. If supplied, the output is restricted to the selected targets. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A character vector of skipped targets.

See Also

Other progress: `tar_canceled()`, `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_poll()`, `tar_progress()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_watch()`, `tar_watch_server()`, `tar_watch_ui()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_skipped()
tar_skipped(starts_with("y_")) # see also any_of()
})
}
```

`tar_source` *Run R scripts.*

Description

Run all the R scripts in a directory in the environment specified.

Usage

```
tar_source(
  files = "R",
  envir = targets::tar_option_get("envir"),
  change_directory = FALSE
)
```

Arguments

`files`	Character vector of file and directory paths to look for R scripts to run. Paths must either be absolute paths or must be relative to the current working directory just before the function call.
`envir`	Environment to run the scripts. Defaults to `tar_option_get("envir")`, the environment of the pipeline.

`change_directory`
: Logical, whether to temporarily change the working directory to the directory of each R script before running it.

Details

`tar_source()` is a convenient way to load R scripts in `_targets.R` to make custom functions available to the pipeline. `tar_source()` recursively looks for files ending in `.R` or `.r`, and it runs each with `eval(parse(text = readLines(script_file, warn = FALSE)), envir)`.

Value

`NULL` (invisibly)

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_store()`, `tar_unblock_process()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
# Running in tar_dir(), these files are written in tempdir().
dir.create("R")
writeLines("f <- function(x) x + 1", file.path("R", "functions.R"))
tar_script({
  tar_source()
  list(tar_target(x, f(1)))
})
tar_make()
tar_read(x) # 2
})
}
```

tar_target *Declare a target.*

Description

A target is a single step of computation in a pipeline. It runs an R command and returns a value. This value gets treated as an R object that can be used by the commands of targets downstream. Targets that are already up to date are skipped. See the user manual for more details.

`tar_target()` defines a target using non-standard evaluation. The `name` argument is an unevaluated symbol, and the `command` and `pattern` arguments are unevaluated expressions. Example: `tar_target(name = data, command = get_data())`.

`tar_target_raw()` defines a target with standard evaluation. The `name` argument is a character string, and the `command` and `pattern` arguments are evaluated expressions. Example: `tar_target_raw(name = "data", command = quote(get_data()))`. `tar_target_raw()` also has extra arguments `deps` and `string` for advanced customization.

Usage

```
tar_target(
  name,
  command,
  pattern = NULL,
  tidy_eval = targets::tar_option_get("tidy_eval"),
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  format = targets::tar_option_get("format"),
  repository = targets::tar_option_get("repository"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = isTRUE(targets::tar_option_get("garbage_collection")),
  deployment = targets::tar_option_get("deployment"),
  priority = targets::tar_option_get("priority"),
  resources = targets::tar_option_get("resources"),
  storage = targets::tar_option_get("storage"),
  retrieval = targets::tar_option_get("retrieval"),
  cue = targets::tar_option_get("cue"),
  description = targets::tar_option_get("description")
)

tar_target_raw(
  name,
  command,
  pattern = NULL,
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  deps = NULL,
  string = NULL,
  format = targets::tar_option_get("format"),
  repository = targets::tar_option_get("repository"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = isTRUE(targets::tar_option_get("garbage_collection")),
  deployment = targets::tar_option_get("deployment"),
  priority = targets::tar_option_get("priority"),
  resources = targets::tar_option_get("resources"),
  storage = targets::tar_option_get("storage"),
  retrieval = targets::tar_option_get("retrieval"),
  cue = targets::tar_option_get("cue"),
  description = targets::tar_option_get("description")
)
```

Arguments

`name` Symbol, name of the target. In `tar_target()`, `name` is an unevaluated symbol, e.g. `tar_target(name = data)`. In `tar_target_raw()`, `name` is a character string, e.g. `tar_target_raw(name = "data")`.

A target name must be a valid name for a symbol in R, and it must not start with a dot. Subsequent targets can refer to this name symbolically to induce a dependency relationship: e.g. `tar_target(downstream_target, f(upstream_target))` is a target named `downstream_target` which depends on a target `upstream_target` and a function `f()`.

In most cases, The target name is the name of its local data file in storage. Some file systems are not case sensitive, which means converting a name to a different case may overwrite a different target. Please ensure all target names have unique names when converted to lower case.

In addition, a target's name determines its random number generator seed. In this way, each target runs with a reproducible seed so someone else running the same pipeline should get the same results, and no two targets in the same pipeline share the same seed. (Even dynamic branches have different names and thus different seeds.) You can recover the seed of a completed target with `tar_meta(your_target, seed)` and run `tar_seed_set()` on the result to locally recreate the target's initial RNG state.

`command` R code to run the target. In `tar_target()`, `command` is an unevaluated expression, e.g. `tar_target(command = data)`. In `tar_target_raw()`, `command` is an evaluated expression, e.g. `tar_target_raw(command = quote(data))`.

`pattern` Code to define a dynamic branching branching for a target. In `tar_target()`, `pattern` is an unevaluated expression, e.g. `tar_target(pattern = map(data))`. In `tar_target_raw()`, `command` is an evaluated expression, e.g. `tar_target_raw(pattern = quote(map(data)))`.

To demonstrate dynamic branching patterns, suppose we have a pipeline with numeric vector targets `x` and `y`. Then, `tar_target(z, x + y, pattern = map(x,` y)) implicitly defines branches of z that each compute x[1] + y[1], x[2] + y[2], and so on. See the user manual for details.

tidy_eval	Logical, whether to enable tidy evaluation when interpreting command and pattern. If TRUE, you can use the "bang-bang" operator !! to programmatically insert the values of global objects.
packages	Character vector of packages to load right before the target runs or the output data is reloaded for downstream targets. Use tar_option_set() to set packages globally for all subsequent targets you define.
library	Character vector of library paths to try when loading packages.
format	Optional storage format for the target's return value. With the exception of format = "file", each target gets a file in _targets/objects, and each format is a different way to save and load this file. See the "Storage formats" section for a detailed list of possible data storage formats.

repository

Character of length 1, remote repository for target storage. Choices:

- "local": file system of the local machine.
- "aws": Amazon Web Services (AWS) S3 bucket. Can be configured with a non-AWS S3 bucket using the endpoint argument of tar_resources_aws(), but versioning capabilities may be lost in doing so. See the cloud storage section of https://books.ropensci.org/targets/data.html for details for instructions.
- "gcp": Google Cloud Platform storage bucket. See the cloud storage section of https://books.ropensci.org/targets/data.html for details for instructions.
- A character string from tar_repository_cas() for content-addressable storage.

Note: if repository is not "local" and format is "file" then the target should create a single output file. That output file is uploaded to the cloud and tracked for changes where it exists in the cloud. The local file is deleted after the target runs.

iteration

Character of length 1, name of the iteration mode of the target. Choices:

- "vector": branching happens with vctrs::vec_slice() and aggregation happens with vctrs::vec_c().
- "list", branching happens with [[]] and aggregation happens with list().
- "group": dplyr::group_by()-like functionality to branch over subsets of a non-dynamic data frame. For iteration = "group", the target must not by dynamic (the pattern argument of tar_target() must be left NULL). The target's return value must be a data frame with a special tar_group column of consecutive integers from 1 through the number of groups. Each integer designates a group, and a branch is created for each collection of rows in a group. See the tar_group() function to see how you can create the special tar_group column with dplyr::group_by().

error

Character of length 1, what to do if the target stops and throws an error. Options:

- "stop": the whole pipeline stops and throws an error.
- "continue": the whole pipeline keeps going.

- "null": The errored target continues and returns NULL. The data hash is deliberately wrong so the target is not up to date for the next run of the pipeline. In addition, as of targets version 1.8.0.9011, a value of NULL is given to upstream dependencies with error = "null" if loading fails.
- "abridge": any currently running targets keep running, but no new targets launch after that.
- "trim": all currently running targets stay running. A queued target is allowed to start if:
  1. It is not downstream of the error, and
  2. It is not a sibling branch from the same tar_target() call (if the error happened in a dynamic branch).

  The idea is to avoid starting any new work that the immediate error impacts. error = "trim" is just like error = "abridge", but it allows potentially healthy regions of the dependency graph to begin running. (Visit https://books.ropensci.org/targets/debugging.html to learn how to debug targets using saved workspaces.)

memory
: Character of length 1, memory strategy. Possible values:

- "auto": new in targets version 1.8.0.9011, memory = "auto" is equivalent to memory = "transient" for dynamic branching (a non-null pattern argument) and memory = "persistent" for targets that do not use dynamic branching.
- "persistent": the target stays in memory until the end of the pipeline (unless storage is "worker", in which case targets unloads the value from memory right after storing it in order to avoid sending copious data over a network).
- "transient": the target gets unloaded after every new target completes. Either way, the target gets automatically loaded into memory whenever another target needs the value.

For cloud-based dynamic files (e.g. format = "file" with repository = "aws"), the memory option applies to the temporary local copy of the file: "persistent" means it remains until the end of the pipeline and is then deleted, and "transient" means it gets deleted as soon as possible. The former conserves bandwidth, and the latter conserves local storage.

garbage_collection
: Logical: TRUE to run base::gc() just before the target runs, FALSE to omit garbage collection. In the case of high-performance computing, gc() runs both locally and on the parallel worker. All this garbage collection is skipped if the actual target is skipped in the pipeline. Non-logical values of garbage_collection are converted to TRUE or FALSE using isTRUE(). In other words, non-logical values are converted FALSE. For example, garbage_collection = 2 is equivalent to garbage_collection = FALSE.

deployment
: Character of length 1. If deployment is "main", then the target will run on the central controlling R process. Otherwise, if deployment is "worker" and you set up the pipeline with distributed/parallel computing, then the target runs on a parallel worker. For more on distributed/parallel computing in targets, please visit https://books.ropensci.org/targets/crew.html.

`priority`	Numeric of length 1 between 0 and 1. Controls which targets get deployed first when multiple competing targets are ready simultaneously. Targets with priorities closer to 1 get dispatched earlier (and polled earlier in `tar_make_future()`).
`resources`	Object returned by `tar_resources()` with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of `targets`. See `tar_resources()` for details.
`storage`	Character string to control when the output of the target is saved to storage. Only relevant when using `targets` with parallel workers (`https://books.ropensci.org/targets/crew.html`). Must be one of the following values: • `"main"`: the target's return value is sent back to the host machine and saved/uploaded locally. • `"worker"`: the worker saves/uploads the value. • `"none"`: `targets` makes no attempt to save the result of the target to storage in the location where `targets` expects it to be. Saving to storage is the responsibility of the user. Use with caution.
`retrieval`	Character string to control when the current target loads its dependencies into memory before running. (Here, a "dependency" is another target upstream that the current one depends on.) Only relevant when using `targets` with parallel workers (`https://books.ropensci.org/targets/crew.html`). Must be one of the following values: • `"main"`: the target's dependencies are loaded on the host machine and sent to the worker before the target runs. • `"worker"`: the worker loads the target's dependencies. • `"none"`: `targets` makes no attempt to load its dependencies. With `retrieval = "none"`, loading dependencies is the responsibility of the user. Use with caution.
`cue`	An optional object from `tar_cue()` to customize the rules that decide whether the target is up to date.
`description`	Character of length 1, a custom free-form human-readable text description of the target. Descriptions appear as target labels in functions like `tar_manifest()` and `tar_visnetwork()`, and they let you select subsets of targets for the `names` argument of functions like `tar_make()`. For example, `tar_manifest(names = tar_described_as(starts_with("survival model")))` lists all the targets whose descriptions start with the character string `"survival model"`.
`deps`	Optional character vector of the adjacent upstream dependencies of the target, including targets and global objects. If `NULL`, dependencies are resolved automatically as usual. The `deps` argument is only for developers of extension packages such as `tarchetypes`, not for end users, and it should almost never be used at all. In scenarios that at first appear to requires `deps`, there is almost always a simpler and more robust workaround that avoids setting `deps`.
`string`	Optional string representation of the command. Internally, the string gets hashed to check if the command changed since last run, which helps `targets` decide whether the target is up to date. External interfaces can take control of `string` to ignore changes in certain parts of the command. If `NULL`, the strings is just deparsed from `command` (default).

Value

A target object. Users should not modify these directly, just feed them to `list()` in your target script file (default: `_targets.R`).

Target objects

Functions like `tar_target()` produce target objects, special objects with specialized sets of S3 classes. Target objects represent skippable steps of the analysis pipeline as described at `https://books.ropensci.org/targets/`. Please read the walkthrough at `https://books.ropensci.org/targets/walkthrough.html` to understand the role of target objects in analysis pipelines.

For developers, `https://wlandau.github.io/targetopia/contributing.html#target-factories` explains target factories (functions like this one which generate targets) and the design specification at `https://books.ropensci.org/targets-design/` details the structure and composition of target objects.

Storage formats

`targets` has several built-in storage formats to control how return values are saved and loaded from disk:

- `"rds"`: Default, uses `saveRDS()` and `readRDS()`. Should work for most objects, but slow.
- `"auto"`: either `"file"` or `"qs"`, depending on the return value of the target. If the return value is a character vector of existing files (and/or directories), then the format becomes `"file"` before `tar_make()` saves the target. Otherwise, the format becomes `"qs"`.
- `"qs"`: Uses `qs2::qs_save()` and `qs2::qs_read()`. Should work for most objects, much faster than `"rds"`. Optionally configure settings through `tar_resources()` and `tar_resources_qs()`.

  Prior to `targets` version 1.8.0.9014, `format = "qs"` used the `qs` package. `qs` has since been superseded in favor of `qs2`, and so later versions of `targets` use `qs2` to save new data. To read existing data, `targets` first attempts `qs2::qs_read()`, and then if that fails, it falls back on `qs::qread()`.
- `"feather"`: Uses `arrow::write_feather()` and `arrow::read_feather()` (version 2.0). Much faster than `"rds"`, but the value must be a data frame. Optionally set `compression` and `compression_level` in `arrow::write_feather()` through `tar_resources()` and `tar_resources_feather()`. Requires the `arrow` package (not installed by default).
- `"parquet"`: Uses `arrow::write_parquet()` and `arrow::read_parquet()` (version 2.0). Much faster than `"rds"`, but the value must be a data frame. Optionally set `compression` and `compression_level` in `arrow::write_parquet()` through `tar_resources()` and `tar_resources_parquet()`. Requires the `arrow` package (not installed by default).
- `"fst"`: Uses `fst::write_fst()` and `fst::read_fst()`. Much faster than `"rds"`, but the value must be a data frame. Optionally set the compression level for `fst::write_fst()` through `tar_resources()` and `tar_resources_fst()`. Requires the `fst` package (not installed by default).
- `"fst_dt"`: Same as `"fst"`, but the value is a `data.table`. Deep copies are made as appropriate in order to protect against the global effects of in-place modification. Optionally set the compression level the same way as for `"fst"`.
- `"fst_tbl"`: Same as `"fst"`, but the value is a `tibble`. Optionally set the compression level the same way as for `"fst"`.

- "keras": superseded by tar_format() and incompatible with error = "null" (in tar_target() or tar_option_set()). Uses keras::save_model_hdf5() and keras::load_model_hdf5(). The value must be a Keras model. Requires the keras package (not installed by default).
- "torch": superseded by tar_format() and incompatible with error = "null" (in tar_target() or tar_option_set()). Uses torch::torch_save() and torch::torch_load(). The value must be an object from the torch package such as a tensor or neural network module. Requires the torch package (not installed by default).
- "file": A dynamic file. To use this format, the target needs to manually identify or save some data and return a character vector of paths to the data (must be a single file path if repository is not "local"). (These paths must be existing files and nonempty directories.) Then, targets automatically checks those files and cues the appropriate run/skip decisions if those files are out of date. Those paths must point to files or directories, and they must not contain characters | or *. All the files and directories you return must actually exist, or else targets will throw an error. (And if storage is "worker", targets will first stall out trying to wait for the file to arrive over a network file system.) If the target does not create any files, the return value should be character(0).

  If repository is not "local" and format is "file", then the character vector returned by the target must be of length 1 and point to a single file. (Directories and vectors of multiple file paths are not supported for dynamic files on the cloud.) That output file is uploaded to the cloud and tracked for changes where it exists in the cloud. The local file is deleted after the target runs.
- "url": A dynamic input URL. For this storage format, repository is implicitly "local", URL format is like format = "file" except the return value of the target is a URL that already exists and serves as input data for downstream targets. Optionally supply a custom curl handle through tar_resources() and tar_resources_url(). in new_handle(), nobody = TRUE is important because it ensures targets just downloads the metadata instead of the entire data file when it checks time stamps and hashes. The data file at the URL needs to have an ETag or a Last-Modified time stamp, or else the target will throw an error because it cannot track the data. Also, use extreme caution when trying to use format = "url" to track uploads. You must be absolutely certain the ETag and Last-Modified time stamp are fully updated and available by the time the target's command finishes running. targets makes no attempt to wait for the web server.
- A custom format can be supplied with tar_format(). For this choice, it is the user's responsibility to provide methods for (un)serialization and (un)marshaling the return value of the target.
- The formats starting with "aws_" are deprecated as of 2022-03-13 (targets version > 0.10.0). For cloud storage integration, use the repository argument instead.

Formats "rds", "file", and "url" are general-purpose formats that belong in the targets package itself. Going forward, any additional formats should be implemented with tar_format() in third-party packages like tarchetypes and geotargets (for example: tarchetypes::tar_format_nanoparquet()). Formats "qs", "fst", etc. are legacy formats from before the existence of tar_format(), and they will continue to remain in targets without deprecation.

See Also

Other targets: tar_cue()

Examples

```
# Defining targets does not run them.
data <- tar_target(target_name, get_data(), packages = "tidyverse")
analysis <- tar_target(analysis, analyze(x), pattern = map(x))
# In a pipeline:
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(
    tar_target(name = x, command = 1 + 1),
    tar_target_raw(name = "y", command = quote(x + y))
  )
})
tar_make()
tar_read(x)
})
# Tidy evaluation
tar_option_set(envir = environment())
n_rows <- 30L
data <- tar_target(target_name, get_data(!!n_rows))
print(data)
# Disable tidy evaluation:
data <- tar_target(target_name, get_data(!!n_rows), tidy_eval = FALSE)
print(data)
tar_option_reset()
}
```

`tar_test` *Test code in a temporary directory.*

Description

Runs a `test_that()` unit test inside a temporary directory to avoid writing to the user's file space. This helps ensure compliance with CRAN policies. Also isolates `tar_option_set()` options and environment variables specific to `targets` and skips the test on Solaris. Useful for writing tests for targetopia packages (extensions to `targets` tailored to specific use cases).

Usage

```
tar_test(label, code)
```

Arguments

`label`	Character of length 1, label for the test.
`code`	User-defined code for the test.

Value

`NULL` (invisibly).

See Also

Other utilities to extend targets: `tar_assert`, `tar_condition`, `tar_language`

Examples

```
tar_test("example test", {
  testing_variable_cafecfcb <- "only defined inside tar_test()"
  file.create("only_exists_in_tar_test")
})
exists("testing_variable_cafecfcb")
file.exists("only_exists_in_tar_test")
```

`tar_timestamp` *Get the timestamp(s) of a target.*

Description

Get the timestamp associated with a target's last successful run. `tar_timestamp()` expects the name argument to be an unevaluated symbol, whereas `tar_timestamp_raw()` expects name to be a character string.

Usage

```
tar_timestamp(
  name = NULL,
  format = NULL,
  tz = NULL,
  parse = NULL,
  store = targets::tar_config_get("store")
)

tar_timestamp_raw(
  name = NULL,
  format = NULL,
  tz = NULL,
  parse = NULL,
  store = targets::tar_config_get("store")
)
```

Arguments

name	Name of the target. If NULL (default) then tar_timestamp() will attempt to return the timestamp of the target currently running. Must be called inside a target's command or a supporting function in order to work. tar_timestamp() expects the name argument to be an unevaluated symbol, whereas tar_timestamp_raw() expects name to be a character string.
format	Deprecated in targets version 0.6.0 (2021-07-21).
tz	Deprecated in targets version 0.6.0 (2021-07-21).
parse	Deprecated in targets version 0.6.0 (2021-07-21).
store	Character string, directory path to the data store of the pipeline.

Details

tar_timestamp() checks the metadata in _targets/meta/meta, not the actual returned data of the target. The timestamp depends on the storage format of the target. If storage is local, e.g. formats like "rds" and "file", then the time stamp is the latest modification time of the target data files at the time the target last successfully ran. For non-local storage as with repository = "aws" and format = "url", targets chooses instead to simply record the time the target last successfully ran.

Value

If the target is not recorded in the metadata or cannot be parsed correctly, then tar_timestamp() returns a POSIXct object at 1970-01-01 UTC.

See Also

Other time: tar_newer(), tar_older()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  list(tar_target(x, 1))
}, ask = FALSE)
tar_make()
# Get the timestamp.
tar_timestamp(x)
# We can use the timestamp to cancel the target
# if it already ran within the last hour.
# Be sure to set `cue = tar_cue(mode = "always")`
# if you want the target to always check the timestamp.
tar_script({
  list(
  tar_target(
    x,
    tar_cancel((Sys.time() - tar_timestamp()) < 3600),
    cue = tar_cue(mode = "always")
  )
)}, ask = FALSE)
tar_make()
})
}
```

tar_toggle *Choose code to run based on Target Markdown mode.*

Description

Run one piece of code if Target Markdown mode interactive mode is turned on and another piece of code otherwise.

Usage

```
tar_toggle(interactive, noninteractive)
```

Arguments

`interactive`	R code to run if Target Markdown interactive mode is activated.
`noninteractive`	R code to run if Target Markdown interactive mode is not activated.

Details

Visit <books.ropensci.org/targets/literate-programming.html> to learn about Target Markdown and interactive mode.

Value

If Target Markdown interactive mode is not turned on, the function returns the result of running the code. Otherwise, the function invisibly returns `NULL`.

See Also

Other Target Markdown: `tar_engine_knitr()`, `tar_interactive()`, `tar_noninteractive()`

Examples

```
tar_toggle(
  message("In interactive mode."),
  message("Not in interactive mode.")
)
```

tar_traceback *Get a target's traceback*

Description

Return the saved traceback of a target. Assumes the target errored out in a previous run of the pipeline with workspaces enabled for that target. See `tar_workspace()` for details.

Usage

```
tar_traceback(
  name,
  envir = NULL,
  packages = NULL,
  source = NULL,
  characters = NULL,
  store = targets::tar_config_get("store")
)
```

Arguments

`name`	Symbol, name of the target whose workspace to read.
`envir`	Deprecated in `targets` > 0.3.1 (2021-03-28).
`packages`	Logical, whether to load the required packages of the target.
`source`	Logical, whether to run the target script file (default: `_targets.R`) to load user-defined global object dependencies into `envir`. If `TRUE`, then `envir` should either be the global environment or inherit from the global environment.
`characters`	Deprecated in `targets` 1.4.0 (2023-12-06).
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

Character vector, the traceback of a failed target if it exists.

See Also

Other debug: `tar_load_globals()`, `tar_workspace()`, `tar_workspaces()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tmp <- sample(1)
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(workspace_on_error = TRUE)
  list(
    tar_target(x, "loaded"),
    tar_target(y, stop(x))
  )
}, ask = FALSE)
try(tar_make())
tar_traceback(y, characters = 60)
})
}
```

`tar_unblock_process` *Unblock the pipeline process*

Description

`targets` tries to avoid running two concurrent instances of `tar_make()` on the same pipeline writing to the same data store. Sometimes it generates false positives (meaning `tar_make()` throws this error even though there is only one instance of the pipeline running.) If there is a false positive, `tar_unblock_process()` gets the pipeline unstuck by removing the `_targets/meta/process` file. This allows the next call to `tar_make()` to resume.

Usage

```
tar_unblock_process(store = targets::tar_config_get("store"))
```

Arguments

`store`	Character string, path to the data store (usually `"_targets"`).

Value

`NULL` (invisibly). Called for its side effects.

See Also

Other utilities: `tar_active()`, `tar_backoff()`, `tar_call()`, `tar_cancel()`, `tar_definition()`, `tar_described_as()`, `tar_envir()`, `tar_format_get()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_path_script()`, `tar_path_script_support()`, `tar_path_store()`, `tar_path_target()`, `tar_source()`, `tar_store()`

tar_unscript — *Remove target script helper files.*

Description

Remove target script helper files (default: `_targets_r/`) that were created by Target Markdown.

Usage

```
tar_unscript(script = targets::tar_config_get("script"))
```

Arguments

`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

Details

Target Markdown code chunks create R scripts in a folder called `_targets_r/` in order to aid the automatically supplied `_targets.R` file. Over time, the number of script files starts to build up, and `targets` has no way of automatically removing helper script files that are no longer necessary. To keep your pipeline up to date with the code chunks in the Target Markdown document(s), it is good practice to call `tar_unscript()` at the beginning of your first Target Markdown document. That way, extraneous/discarded targets are automatically removed from the pipeline when the document starts render.

If the target script is at some alternative path, e.g. `custom/script.R`, the helper scripts are in `custom/script_r/`. `tar_unscript()` works on the helper scripts as long as your project configuration settings correctly identify the correct target script.

Value

`NULL` (invisibly).

Examples

```
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_unscript()
})
```

tar_unversion *Delete cloud object version IDs from local metadata.*

Description

Delete version IDs from local metadata.

Usage

```
tar_unversion(
  names = tidyselect::everything(),
  store = targets::tar_config_get("store")
)
```

Arguments

`names`	Tidyselect expression to identify the targets to drop version IDs. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

`NULL` (invisibly).

Cloud target data versioning

Some buckets in Amazon S3 or Google Cloud Storage are "versioned", which means they track historical versions of each data object. If you use `targets` with cloud storage (`https://books.ropensci.org/targets/cloud-storage.html`) and versioning is turned on, then `targets` will record each version of each target in its metadata.

Functions like `tar_read()` and `tar_load()` load the version recorded in the local metadata, which may not be the same as the "current" version of the object in the bucket. Likewise, functions `tar_delete()` and `tar_destroy()` only remove the version ID of each target as recorded in the local metadata.

If you want to interact with the *latest* version of an object instead of the version ID recorded in the local metadata, then you will need to delete the object from the metadata.

1. Make sure your local copy of the metadata is current and up to date. You may need to run `tar_meta_download()` or `tar_meta_sync()` first.
2. Run `tar_unversion()` to remove the recorded version IDs of your targets in the local metadata.

3. With the version IDs gone from the local metadata, functions like `tar_read()` and `tar_destroy()` will use the *latest* version of each target data object.
4. Optional: to back up the local metadata file with the version IDs deleted, use `tar_meta_upload()`.

See Also

Other clean: `tar_delete()`, `tar_destroy()`, `tar_invalidate()`, `tar_prune()`, `tar_prune_list()`

---

tar_validate *Validate a pipeline of targets.*

Description

Inspect the pipeline for issues and throw an error or warning if a problem is detected.

Usage

```
tar_validate(
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.
`envir`	An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

`NULL` except if `callr_function = callr::r_bg()`, in which case a handle to the `callr` background process is returned. Either way, the value is invisibly returned.

See Also

Other inspect: `tar_deps()`, `tar_manifest()`, `tar_network()`, `tar_outdated()`, `tar_sitrep()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script(list(tar_target(x, 1 + 1)), ask = FALSE)
tar_validate()
})
}
```

`tar_visnetwork` *visNetwork dependency graph.*

Description

Visualize the pipeline dependency graph with a `visNetwork` HTML widget.

Usage

```
tar_visnetwork(
  targets_only = FALSE,
  names = NULL,
  shortcut = FALSE,
  allow = NULL,
  exclude = ".Random.seed",
  outdated = TRUE,
  label = targets::tar_config_get("label"),
  label_width = targets::tar_config_get("label_width"),
  level_separation = targets::tar_config_get("level_separation"),
  degree_from = 1L,
  degree_to = 1L,
  zoom_speed = 1,
  physics = FALSE,
  reporter = targets::tar_config_get("reporter_outdated"),
  seconds_reporter = targets::tar_config_get("seconds_reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::tar_callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include global functions and objects.
`names`	Names of targets. The graph visualization will operate only on these targets (and unless `shortcut` is `TRUE`, all the targets upstream as well). Selecting a small subgraph using `names` could speed up the load time of the visualization. Unlike `allow`, `names` is invoked before the graph is generated. Set to NULL to check/run all the targets (default). Otherwise, the object supplied to `names` should be a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.
`shortcut`	Logical of length 1, how to interpret the `names` argument. If `shortcut` is `FALSE` (default) then the function checks all targets upstream of `names` as far back as the dependency graph goes. If `TRUE`, then the function only checks the targets in `names` and uses stored metadata for information about upstream dependencies as needed. `shortcut = TRUE` increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, `shortcut = TRUE` only works if you set `names`.
`allow`	Optional, define the set of allowable vertices in the graph. Unlike `names`, `allow` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or `tidyselect` helpers like `starts_with()`.
`exclude`	Optional, define the set of exclude vertices from the graph. Unlike `names`, `exclude` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to exclude no vertices. Otherwise, you can supply symbols or `tidyselect` helpers like `any_of()` and `starts_with()`.
`outdated`	Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and setting `outdated` to `FALSE` is a nice way to speed up the graph if you only want to see dependency relationships and pipeline progress.
`label`	Character vector of one or more aesthetics to add to the vertex labels. Can contain `"description"` to show each target's custom description, `"time"` to show

total runtime, "size" to show total storage size, or "branches" to show the number of branches in each pattern. You can choose multiple aesthetics at once, e.g. label = c("description", "time"). Only the description is enabled by default.

label_width	Positive numeric of length 1, maximum width (in number of characters) of the node labels.
level_separation	Numeric of length 1, levelSeparation argument of visNetwork::visHierarchicalLayout(). Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If level_separation is NULL, the levelSeparation argument of visHierarchicalLayout() defaults to 150.
degree_from	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. degree_from controls the number of edges the neighborhood extends upstream.
degree_to	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. degree_to controls the number of edges the neighborhood extends downstream.
zoom_speed	Positive numeric of length 1, scaling factor on the zoom speed. Above 1 zooms faster than default, below 1 zooms lower than default.
physics	Logical of length 1, whether to implement interactive physics in the graph, e.g. edge elasticity.
reporter	Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: * "forecast_interactive" (default): use the forecast reporter if the session is interactive (see base::interactive()), otherwise use the silent reporter. * "silent": print nothing. * "forecast": print running totals of the checked and outdated targets found so far.
seconds_reporter	Positive numeric of length 1 with the minimum number of seconds between times when the reporter prints progress messages to the R console. This is an aggressive optimization setting not recommended for most users: higher values might make some pipelines run faster, but it becomes less clear which targets are actually running at any given moment. When the pipeline is just skipping targets, the actual interval between messages is max(1, seconds_reporter) to reduce overhead.
callr_function	A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.
callr_arguments	A list of arguments to callr_function.
envir	An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A `visNetwork` HTML widget object.

Dependency graph

The dependency graph of a pipeline is a directed acyclic graph (DAG) where each node indicates a target or global object and each directed edge indicates where a downstream node depends on an upstream node. The DAG is not always a tree, but it never contains a cycle because no target is allowed to directly or indirectly depend on itself. The dependency graph should show a natural progression of work from left to right. `targets` uses static code analysis to create the graph, so the order of `tar_target()` calls in the `_targets.R` file does not matter. However, targets does not support self-referential loops or other cycles. For more information on the dependency graph, please read `https://books.ropensci.org/targets/targets.html#dependencies`.

Storage access

Several functions like `tar_make()`, `tar_read()`, `tar_load()`, `tar_meta()`, and `tar_progress()` read or modify the local data store of the pipeline. The local data store is in flux while a pipeline is running, and depending on how distributed computing or cloud computing is set up, not all targets can even reach it. So please do not call these functions from inside a target as part of a running pipeline. The only exception is literate programming target factories in the `tarchetypes` package such as `tar_render()` and `tar_quarto()`.

See Also

Other visualize: `tar_glimpse()`, `tar_mermaid()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
```

```
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2, description = "sum of two other sums")
  )
})
tar_visnetwork()
tar_visnetwork(allow = starts_with("y")) # see also any_of()
})
}
```

`tar_watch` *Shiny app to watch the dependency graph.*

Description

Launches a background process with a Shiny app that calls `tar_visnetwork()` every few seconds. To embed this app in other apps, use the Shiny module in `tar_watch_ui()` and `tar_watch_server()`.

Usage

```
tar_watch(
  seconds = 10,
  seconds_min = 1,
  seconds_max = 60,
  seconds_step = 1,
  targets_only = FALSE,
  exclude = ".Random.seed",
  outdated = FALSE,
  label = NULL,
  level_separation = 150,
  degree_from = 1L,
  degree_to = 1L,
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main"),
  height = "650px",
  display = "summary",
  displays = c("summary", "branches", "progress", "graph", "about"),
  background = TRUE,
  browse = TRUE,
  host = getOption("shiny.host", "127.0.0.1"),
  port = getOption("shiny.port", targets::tar_random_port()),
  verbose = TRUE,
  supervise = TRUE,
  poll_connection = TRUE,
  stdout = "|",
  stderr = "|",
  title = "",
  theme = bslib::bs_theme(),
  spinner = TRUE
)
```

Arguments

`seconds`	Numeric of length 1, default number of seconds between refreshes of the graph. Can be changed in the app controls.
`seconds_min`	Numeric of length 1, lower bound of `seconds` in the app controls.
`seconds_max`	Numeric of length 1, upper bound of `seconds` in the app controls.
`seconds_step`	Numeric of length 1, step size of `seconds` in the app controls.
`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include global functions and objects.
`exclude`	Character vector of nodes to omit from the graph.
`outdated`	Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and setting `outdated` to `FALSE` is a nice way to speed up the graph if you only want to see dependency relationships and pipeline progress.
`label`	Label argument to `tar_visnetwork()`.
`level_separation`	Numeric of length 1, `levelSeparation` argument of `visNetwork::visHierarchicalLayout()`. Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If `level_separation` is `NULL`, the `levelSeparation` argument of `visHierarchicalLayout()` defaults to `150`.
`degree_from`	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_from` controls the number of edges the neighborhood extends upstream.
`degree_to`	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_to` controls the number of edges the neighborhood extends downstream.
`config`	Character of length 1, file path of the YAML configuration file with `targets` project settings. The `config` argument specifies which YAML configuration file that `tar_config_get()` reads from or `tar_config_set()` writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always `_targets.yaml` unless you set another default path using the `TAR_CONFIG` environment variable, e.g. `Sys.setenv(TAR_CONFIG = "custom.yaml")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.

`project`	Character of length 1, name of the current `targets` project. Thanks to the `config` R package, `targets` YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The `project` argument allows you to set or get a configuration setting for a specific project for a given call to `tar_config_set()` or `tar_config_get()`. The default project is always called `"main"` unless you set another default project using the `TAR_PROJECT` environment variable, e.g. `Sys.setenv(tar_project = "custom")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.
`height`	Character of length 1, height of the `visNetwork` widget and branches table.
`display`	Character of length 1, which display to show first.
`displays`	Character vector of choices for the display. Elements can be any of `"graph"`, `"summary"`, `"branches"`, or `"about"`.
`background`	Logical, whether to run the app in a background process so you can still use the R console while the app is running.
`browse`	Whether to open the app in a browser when the app is ready. Only relevant if `background` is `TRUE`.
`host`	Character of length 1, IPv4 address to listen on. Only relevant if `background` is `TRUE`.
`port`	Positive integer of length 1, TCP port to listen on. Only relevant if `background` is `TRUE`.
`verbose`	whether to print a spinner and informative messages. Only relevant if `background` is `TRUE`.
`supervise`	Whether to register the process with a supervisor. If `TRUE`, the supervisor will ensure that the process is killed when the R process exits.
`poll_connection`	Whether to have a control connection to the process. This is used to transmit messages from the subprocess to the main process.
`stdout`	The name of the file the standard output of the child R process will be written to. If the child process runs with the `--slave` option (the default), then the commands are not echoed and will not be shown in the standard output. Also note that you need to call `print()` explicitly to show the output of the command(s). IF `NULL` (the default), then standard output is not returned, but it is recorded and included in the error object if an error happens.
`stderr`	The name of the file the standard error of the child R process will be written to. In particular `message()` sends output to the standard error. If nothing was sent to the standard error, then this file will be empty. This argument can be the same file as `stdout`, in which case they will be correctly interleaved. If this is the string `"2>&1"`, then standard error is redirected to standard output. IF `NULL` (the default), then standard output is not returned, but it is recorded and included in the error object if an error happens.
`title`	Character of length 1, title of the UI.
`theme`	A call to `bslib::bs_theme()` with the `bslib` theme.
`spinner`	`TRUE` to add a busy spinner, `FALSE` to omit.

Details

The controls of the app are in the left panel. The seconds control is the number of seconds between refreshes of the graph, and the other settings match the arguments of `tar_visnetwork()`.

Value

A handle to `callr::r_bg()` background process running the app.

See Also

Other progress: `tar_canceled()`, `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_poll()`, `tar_progress()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_skipped()`, `tar_watch_server()`, `tar_watch_ui()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  sleep_run <- function(...) {
    Sys.sleep(10)
  }
  list(
    tar_target(settings, sleep_run()),
    tar_target(data1, sleep_run(settings)),
    tar_target(data2, sleep_run(settings))
  )
}, ask = FALSE)
# Launch the app in a background process.
tar_watch(seconds = 10, outdated = FALSE, targets_only = TRUE)
# Run the pipeline.
tar_make()
})
}
```

`tar_watch_server` *Shiny module server for tar_watch()*

Description

Use `tar_watch_ui()` and `tar_watch_server()` to include `tar_watch()` as a Shiny module in an app.

Usage

```
tar_watch_server(
  id,
  height = "650px",
  exclude = ".Random.seed",
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```

Arguments

`id`	Character of length 1, ID corresponding to the UI function of the module.
`height`	Character of length 1, height of the `visNetwork` widget and branches table.
`exclude`	Character vector of nodes to omit from the graph.
`config`	Character of length 1, file path of the YAML configuration file with `targets` project settings. The `config` argument specifies which YAML configuration file that `tar_config_get()` reads from or `tar_config_set()` writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always `_targets.yaml` unless you set another default path using the `TAR_CONFIG` environment variable, e.g. `Sys.setenv(TAR_CONFIG = "custom.yaml")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.
`project`	Character of length 1, name of the current `targets` project. Thanks to the `config` R package, `targets` YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The `project` argument allows you to set or get a configuration setting for a specific project for a given call to `tar_config_set()` or `tar_config_get()`. The default project is always called `"main"` unless you set another default project using the `TAR_PROJECT` environment variable, e.g. `Sys.setenv(tar_project = "custom")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.

Value

A Shiny module server.

See Also

Other progress: `tar_canceled()`, `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_poll()`, `tar_progress()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_skipped()`, `tar_watch()`, `tar_watch_ui()`

`tar_watch_ui` *Shiny module UI for tar_watch()*

Description

Use `tar_watch_ui()` and `tar_watch_server()` to include `tar_watch()` as a Shiny module in an app.

Usage

```
tar_watch_ui(
  id,
  label = "tar_watch_label",
  seconds = 10,
  seconds_min = 1,
  seconds_max = 60,
  seconds_step = 1,
  targets_only = FALSE,
  outdated = FALSE,
  label_tar_visnetwork = NULL,
  level_separation = 150,
  degree_from = 1L,
  degree_to = 1L,
  height = "650px",
  display = "summary",
  displays = c("summary", "branches", "progress", "graph", "about"),
  title = "",
  theme = bslib::bs_theme(),
  spinner = FALSE
)
```

Arguments

`id`	Character of length 1, ID corresponding to the UI function of the module.
`label`	Label for the module.
`seconds`	Numeric of length 1, default number of seconds between refreshes of the graph. Can be changed in the app controls.
`seconds_min`	Numeric of length 1, lower bound of `seconds` in the app controls.
`seconds_max`	Numeric of length 1, upper bound of `seconds` in the app controls.
`seconds_step`	Numeric of length 1, step size of `seconds` in the app controls.
`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include global functions and objects.
`outdated`	Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and

	setting `outdated` to `FALSE` is a nice way to speed up the graph if you only want to see dependency relationships and pipeline progress.
`label_tar_visnetwork`	Character vector, `label` argument to `tar_visnetwork()`.
`level_separation`	Numeric of length 1, `levelSeparation` argument of `visNetwork::visHierarchicalLayout()`. Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If `level_separation` is `NULL`, the `levelSeparation` argument of `visHierarchicalLayout()` defaults to `150`.
`degree_from`	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_from` controls the number of edges the neighborhood extends upstream.
`degree_to`	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_to` controls the number of edges the neighborhood extends downstream.
`height`	Character of length 1, height of the `visNetwork` widget and branches table.
`display`	Character of length 1, which display to show first.
`displays`	Character vector of choices for the display. Elements can be any of `"graph"`, `"summary"`, `"branches"`, or `"about"`.
`title`	Character of length 1, title of the UI.
`theme`	A call to `bslib::bs_theme()` with the `bslib` theme.
`spinner`	`TRUE` to add a busy spinner, `FALSE` to omit.

Value

A Shiny module UI.

See Also

Other progress: `tar_canceled()`, `tar_completed()`, `tar_dispatched()`, `tar_errored()`, `tar_poll()`, `tar_progress()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_skipped()`, `tar_watch()`, `tar_watch_server()`

`tar_workspace` *Load a saved workspace and seed for debugging.*

Description

Load the packages, environment, and random number generator seed of a target.

Usage

```
tar_workspace(
  name,
  envir = parent.frame(),
  packages = TRUE,
  source = TRUE,
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

`name`	Symbol, name of the target whose workspace to read.
`envir`	Environment in which to put the objects.
`packages`	Logical, whether to load the required packages of the target.
`source`	Logical, whether to run `_targets.R` to load user-defined global object dependencies into `envir`. If `TRUE`, then `envir` should either be the global environment or inherit from the global environment.
`script`	Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
`store`	Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

If you activate workspaces through the `workspaces` argument of `tar_option_set()`, then under the circumstances you specify, `targets` will save a special workspace file to a location in in `_targets/workspaces/`. The workspace file is a compact reference that allows `tar_workspace()` to load the target's dependencies and random number generator seed as long as the data objects are still in the data store (usually files in `_targets/objects/`). When you are done debugging, you can remove the workspace files using `tar_destroy(destroy = "workspaces")`.

Value

This function returns `NULL`, but it does load the target's required packages, as well as multiple objects into the environment (`envir` argument) in order to replicate the workspace where the error happened. These objects include the global objects at the time `tar_make()` was called and the dependency targets. The random number generator seed for the target is also assigned with `tar_seed_set()`.

See Also

Other debug: `tar_load_globals()`, `tar_traceback()`, `tar_workspaces()`

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tmp <- sample(1)
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(workspace_on_error = TRUE)
  list(
    tar_target(x, "loaded"),
    tar_target(y, stop(x))
  )
}, ask = FALSE)
# The following code throws an error for demonstration purposes.
try(tar_make())
exists("x") # Should be FALSE.
tail(.Random.seed) # for comparison to the RNG state after tar_workspace(y)
tar_workspace(y)
exists("x") # Should be TRUE.
print(x) # "loaded"
# Should be different: tar_workspace() runs
# tar_seed_set(tar_meta(y, seed)$seed)
tail(.Random.seed)
})
}
```

`tar_workspaces` *List saved target workspaces.*

Description

List target workspaces currently saved to `_targets/workspaces/`. See `tar_workspace()` for more information.

Usage

```
tar_workspaces(names = NULL, store = targets::tar_config_get("store"))
```

Arguments

`names`	Optional `tidyselect` selector to return a tactical subset of workspace names. If `NULL`, all names are selected. The object supplied to `names` should be `NULL` or a `tidyselect` expression like `any_of()` or `starts_with()` from `tidyselect` itself, or `tar_described_as()` to select target names based on their descriptions.

store	Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Value

Character vector of available workspaces to load with tar_workspace().

See Also

Other debug: tar_load_globals(), tar_traceback(), tar_workspace()

Examples

```
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) { # for CRAN
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
tar_script({
  library(targets)
  library(tarchetypes)
  tar_option_set(workspace_on_error = TRUE)
  list(
    tar_target(x, "value"),
    tar_target(y, x)
  )
}, ask = FALSE)
tar_make()
tar_workspaces()
tar_workspaces(contains("x"))
})
}
```

use_targets *Use targets*

Description

Set up targets for an existing project.

Usage

```
use_targets(
  script = targets::tar_config_get("script"),
  open = interactive(),
  overwrite = FALSE,
  scheduler = NULL,
  job_name = NULL
)
```

Arguments

`script`	Character of length 1, where to write the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`.
`open`	Logical of length 1, whether to open the file for editing in the RStudio IDE.
`overwrite`	Logical of length 1, `TRUE` to overwrite the the target script file, `FALSE` otherwise.
`scheduler`	Deprecated in `targets` version 1.5.0.9001 (2024-02-12).
`job_name`	Deprecated in `targets` version 1.5.0.9001 (2024-02-12).

Details

`use_targets()` writes an example `_targets.R` script to get started with a `targets` pipeline for the current project. Follow the comments in this script to adapt it as needed. For more information, please visit `https://books.ropensci.org/targets/walkthrough.html`.

Value

`NULL` (invisibly).

See Also

Other help: `tar_reprex()`, `targets-package`, `use_targets_rmd()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
use_targets(open = FALSE)
})
}
```

`use_targets_rmd` *Use targets with Target Markdown.*

Description

Create an example Target Markdown report to get started with `targets`.

Usage

```
use_targets_rmd(path = "_targets.Rmd", open = interactive())
```

Arguments

`path`	Character of length 1, output path of the Target Markdown report relative to the current active project.
`open`	Logical, whether to open the file for editing in the RStudio IDE.

Value

`NULL` (invisibly).

See Also

Other help: `tar_reprex()`, `targets-package`, `use_targets()`

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temp dir for CRAN.
use_targets(open = FALSE)
})
}
```

Index

∗ **Target Markdown**
    tar_engine_knitr, 43
    tar_interactive, 64
    tar_noninteractive, 104
    tar_toggle, 199
∗ **branching**
    tar_branch_index, 11
    tar_branch_names, 12
    tar_branches, 10
    tar_pattern, 125
∗ **clean**
    tar_delete, 34
    tar_destroy, 39
    tar_invalidate, 65
    tar_prune, 135
    tar_prune_list, 137
    tar_unversion, 203
∗ **configuration**
    tar_config_get, 18
    tar_config_projects, 20
    tar_config_set, 21
    tar_config_unset, 26
    tar_config_yaml, 28
    tar_envvars, 46
    tar_option_get, 108
    tar_option_reset, 109
    tar_option_set, 110
∗ **content-addressable storage**
    tar_repository_cas, 143
    tar_repository_cas_local, 148
    tar_repository_cas_local_gc, 150
∗ **data**
    tar_crew, 29
    tar_pid, 127
    tar_process, 129
∗ **debug**
    tar_load_globals, 71
    tar_traceback, 200
    tar_workspace, 215
    tar_workspaces, 217
∗ **existence**
    tar_exist_meta, 49
    tar_exist_objects, 50
    tar_exist_process, 51
    tar_exist_progress, 51
    tar_exist_script, 52
∗ **help**
    tar_reprex, 153
    targets-package, 5
    use_targets, 218
    use_targets_rmd, 219
∗ **inspect**
    tar_deps, 36
    tar_manifest, 84
    tar_network, 100
    tar_outdated, 118
    tar_sitrep, 183
    tar_validate, 204
∗ **metadata**
    tar_meta, 90
    tar_meta_delete, 93
    tar_meta_download, 94
    tar_meta_sync, 96
    tar_meta_upload, 97
∗ **pipeline**
    tar_make, 73
    tar_make_clustermq, 77
    tar_make_future, 80
∗ **progress**
    tar_canceled, 15
    tar_completed, 16
    tar_dispatched, 42
    tar_errored, 48
    tar_poll, 128
    tar_progress, 130
    tar_progress_branches, 132
    tar_progress_summary, 133
    tar_skipped, 186

tar_watch, 209
tar_watch_server, 212
tar_watch_ui, 214
∗ **pseudo-random number generation**
tar_seed_create, 178
tar_seed_get, 180
tar_seed_set, 181
∗ **resources**
tar_resources, 154
tar_resources_aws, 156
tar_resources_clustermq, 159
tar_resources_crew, 161
tar_resources_custom_format, 162
tar_resources_feather, 164
tar_resources_fst, 165
tar_resources_future, 166
tar_resources_gcp, 168
tar_resources_network, 169
tar_resources_parquet, 171
tar_resources_qs, 172
tar_resources_repository_cas, 174
tar_resources_url, 175
∗ **scripts**
tar_edit, 43
tar_github_actions, 57
tar_helper, 63
tar_renv, 141
tar_script, 177
∗ **storage**
tar_format, 53
tar_load, 67
tar_load_everything, 70
tar_objects, 105
tar_read, 139
∗ **targets**
tar_cue, 30
tar_target, 189
∗ **time**
tar_newer, 103
tar_older, 106
tar_timestamp, 197
∗ **utilities to extend targets**
tar_assert, 6
tar_condition, 17
tar_language, 66
tar_test, 196
∗ **utilities**
tar_active, 5
tar_backoff, 9
tar_call, 13
tar_cancel, 14
tar_definition, 32
tar_described_as, 37
tar_envir, 45
tar_format_get, 56
tar_group, 61
tar_name, 99
tar_path_script, 121
tar_path_script_support, 122
tar_path_store, 123
tar_path_target, 123
tar_source, 187
tar_unblock_process, 201
∗ **visualize**
tar_glimpse, 58
tar_mermaid, 86
tar_visnetwork, 205

any_of(), *15, 16*, *34*, *42*, *48*, *59*, *65*, *68*, *73*, *77*, *81*, *84*, *87*, *90*, *100, 101*, *103*, *105*, *107*, *119*, *129*, *131, 132*, *183*, *186*, *203*, *206*, *217*

base::interactive(), *88*, *101*, *119*, *184*, *207*
bslib::bs_theme(), *211*, *215*

cross *(*tar_pattern*)*, 125

head *(*tar_pattern*)*, 125

library(), *141*
list(), *178*, *194*

map *(*tar_pattern*)*, 125

qs2::qs_read(), *116*, *194*
qs2::qs_save(), *172*
qs::qread(), *116*, *194*

sample *(*tar_pattern*)*, 125
starts_with(), *15, 16*, *34*, *42*, *48*, *59*, *65*, *68*, *73*, *77*, *81*, *84*, *87*, *90*, *100, 101*, *103*, *105*, *107*, *119*, *129*, *131, 132*, *183*, *186*, *203*, *206*, *217*

tail *(*tar_pattern*)*, 125
tar_active, 5, *9*, *14, 15*, *33*, *38*, *46*, *57*, *62*, *99*, *121–124*, *188*, *201*

tar_assert, 6, *18*, *67*, *197*
tar_assert_chr *(tar_assert)*, 6
tar_assert_dbl *(tar_assert)*, 6
tar_assert_df *(tar_assert)*, 6
tar_assert_envir *(tar_assert)*, 6
tar_assert_equal_lengths *(tar_assert)*, 6
tar_assert_expr *(tar_assert)*, 6
tar_assert_file *(tar_assert)*, 6
tar_assert_finite *(tar_assert)*, 6
tar_assert_flag *(tar_assert)*, 6
tar_assert_function *(tar_assert)*, 6
tar_assert_function_arguments *(tar_assert)*, 6
tar_assert_ge *(tar_assert)*, 6
tar_assert_identical *(tar_assert)*, 6
tar_assert_in *(tar_assert)*, 6
tar_assert_inherits *(tar_assert)*, 6
tar_assert_int *(tar_assert)*, 6
tar_assert_internet *(tar_assert)*, 6
tar_assert_lang *(tar_assert)*, 6
tar_assert_le *(tar_assert)*, 6
tar_assert_lgl *(tar_assert)*, 6
tar_assert_list *(tar_assert)*, 6
tar_assert_match *(tar_assert)*, 6
tar_assert_name *(tar_assert)*, 6
tar_assert_named *(tar_assert)*, 6
tar_assert_names *(tar_assert)*, 6
tar_assert_nonempty *(tar_assert)*, 6
tar_assert_nonmissing *(tar_assert)*, 6
tar_assert_not_dir *(tar_assert)*, 6
tar_assert_not_dirs *(tar_assert)*, 6
tar_assert_not_expr *(tar_assert)*, 6
tar_assert_not_in *(tar_assert)*, 6
tar_assert_null *(tar_assert)*, 6
tar_assert_nzchar *(tar_assert)*, 6
tar_assert_package *(tar_assert)*, 6
tar_assert_path *(tar_assert)*, 6
tar_assert_positive *(tar_assert)*, 6
tar_assert_scalar *(tar_assert)*, 6
tar_assert_store *(tar_assert)*, 6
tar_assert_target *(tar_assert)*, 6
tar_assert_target_list *(tar_assert)*, 6
tar_assert_true *(tar_assert)*, 6
tar_assert_unique *(tar_assert)*, 6
tar_assert_unique_targets *(tar_assert)*, 6
tar_backoff, *5*, 9, *14*, *15*, *33*, *38*, *46*, *57*, *62*, *99*, *121–124*, *188*, *201*
tar_backoff(), *9*, *114*
tar_branch_index, *11*, 11, *13*, *126*
tar_branch_names, *11*, *12*, 12, *126*
tar_branch_names(), *12*, *13*
tar_branch_names_raw *(tar_branch_names)*, 12
tar_branch_names_raw(), *12*, *13*
tar_branches, 10, *12*, *13*, *126*
tar_branches(), *36*
tar_call, *5*, *9*, 13, *15*, *33*, *38*, *46*, *57*, *62*, *99*, *121–124*, *188*, *201*
tar_cancel, *5*, *9*, *14*, 14, *33*, *38*, *46*, *57*, *62*, *99*, *121–124*, *188*, *201*
tar_cancel(), *134*
tar_canceled, 15, *17*, *42*, *48*, *128*, *131*, *133*, *134*, *187*, *212*, *213*, *215*
tar_cas_d(), *146*, *150*, *152*
tar_cas_l(), *144*, *146*, *150*, *152*
tar_cas_u(), *146*, *150*, *152*
tar_completed, *16*, 16, *42*, *48*, *128*, *131*, *133*, *134*, *187*, *212*, *213*, *215*
tar_condition, *8*, 17, *67*, *197*
tar_config_get, 18, *21*, *25*, *27*, *29*, *47*, *108*, *109*, *117*
tar_config_get(), *10*, *12*, *15*, *16*, *29*, *34*, *38*, *40*, *42*, *43*, *48–52*, *60*, *65*, *75*, *79*, *82*, *83*, *85*, *88*, *89*, *91*, *93–98*, *102*, *103*, *105*, *107*, *120*, *127–129*, *131*, *132*, *134*, *136*, *138*, *140*, *142*, *151*, *185*, *186*, *200*, *202*, *203*, *205*, *208*, *216*, *218*
tar_config_projects, *20*, 20, *25*, *27*, *29*, *47*, *108*, *109*, *117*
tar_config_set, *20*, *21*, 21, *27*, *29*, *47*, *108*, *109*, *117*
tar_config_set(), *10*, *12*, *15*, *16*, *29*, *34*, *38*, *40*, *42*, *43*, *47–52*, *60*, *65*, *75*, *79*, *82*, *83*, *85*, *88*, *89*, *91*, *93–98*, *102*, *103*, *105*, *107*, *120*, *127–129*, *131*, *132*, *134*, *136*, *138*, *140*, *142*, *151*, *185*, *186*, *200*, *202*, *203*, *205*, *208*, *216*, *218*
tar_config_unset, *20*, *21*, *25*, 26, *29*, *47*, *108*, *109*, *117*
tar_config_unset(), *22–24*, *71*
tar_config_yaml, *20*, *21*, *25*, *27*, 28, *47*, *108*, *109*, *117*
tar_crew, 29, *127*, *130*

tar_cue, 30, *195*
tar_cue(), *85*, *118*, *183–185*
tar_definition, *5*, *9*, *14, 15*, 32, *38*, *46*, *57*, *62*, *99*, *121–124*, *188*, *201*
tar_delete, 34, *41*, *66*, *137*, *139*, *204*
tar_delete(), *35*, *40, 41*, *69*, *136*, *140*, *203*
tar_deparse_language *(*tar_language*)*, 66
tar_deparse_safe *(*tar_language*)*, 66
tar_deps, 36, *86*, *102*, *120*, *186*, *205*
tar_deps(), *32*, *36*
tar_deps_raw *(*tar_deps*)*, 36
tar_deps_raw(), *36*
tar_described_as, *5*, *9*, *14, 15*, *33*, 37, *46*, *57*, *62*, *99*, *121–124*, *188*, *201*
tar_described_as(), *15, 16*, *34*, *38*, *48*, *59*, *65*, *68*, *73*, *77*, *81*, *84*, *87*, *101*, *103*, *105*, *107*, *119*, *131, 132*, *183*, *186*, *203*, *206*, *217*
tar_destroy, *35*, 39, *66*, *137*, *139*, *204*
tar_destroy(), *35*, *41*, *69*, *92*, *136, 137*, *140*, *203, 204*
tar_dispatched, *16, 17*, 42, *48*, *128*, *131*, *133, 134*, *187*, *212, 213*, *215*
tar_edit, 43, *58*, *64*, *143*, *178*
tar_engine_knitr, 43, *65*, *105*, *199*
tar_envir, *5*, *9*, *14, 15*, *33*, *38*, 45, *57*, *62*, *99*, *121–124*, *188*, *201*
tar_envvars, *20, 21*, *25*, *27*, *29*, 46, *108, 109*, *117*
tar_envvars(), *47*
tar_error *(*tar_condition*)*, 17
tar_errored, *16, 17*, *42*, 48, *128*, *131*, *133*, *134*, *187*, *212, 213*, *215*
tar_exist_meta, 49, *50–53*
tar_exist_objects, *49*, 50, *51–53*
tar_exist_process, *49, 50*, 51, *52, 53*
tar_exist_progress, *49–51*, 51, *53*
tar_exist_script, *49–52*, 52
tar_format, 53, *69, 70*, *106*, *141*
tar_format(), *55*, *116, 117*, *155*, *162, 163*, *174*, *195*
tar_format_get, *5*, *9*, *14, 15*, *33*, *38*, *46*, 56, *62*, *99*, *121–124*, *188*, *201*
tar_format_get(), *57*, *144*
tar_github_actions, *43*, 57, *64*, *143*, *178*
tar_github_actions(), *47*
tar_glimpse, 58, *89*, *208*
tar_glimpse(), *22*, *84*
tar_group, *5*, *9*, *14, 15*, *33*, *38*, *46*, *57*, 61, *99*, *121–124*, *188*, *201*
tar_group(), *112*, *191*
tar_helper, *43*, *58*, 63, *143*, *178*
tar_helper(), *63*
tar_helper_raw *(*tar_helper*)*, 63
tar_helper_raw(), *63*
tar_interactive, *45*, 64, *105*, *199*
tar_invalidate, *35*, *41*, 65, *137*, *139*, *204*
tar_invalidate(), *35*, *40*, *92*, *106*
tar_language, *8*, *18*, 66, *197*
tar_load, *55*, 67, *70*, *106*, *141*
tar_load(), *35*, *41*, *65*, *67–69*, *136*, *140*, *203*
tar_load_everything, *55*, *69*, 70, *106*, *141*
tar_load_everything(), *67*
tar_load_globals, 71, *200*, *217, 218*
tar_load_raw *(*tar_load*)*, 67
tar_load_raw(), *67, 68*
tar_make, 73, *80*, *83*
tar_make(), *9*, *13*, *19–29*, *33*, *38–40*, *45*, *57*, *60*, *65*, *71*, *75*, *77*, *79, 80*, *82, 83*, *85*, *88*, *92*, *99*, *102*, *111*, *114–116*, *118*, *119*, *124*, *127*, *129, 130*, *135, 136*, *138*, *142, 143*, *145*, *149*, *151, 152*, *178*, *180*, *183–185*, *193, 194*, *201*, *204*, *208*, *210, 211*, *213*, *216*
tar_make_clustermq, *76*, 77, *83*
tar_make_clustermq(), *9*, *23, 24*, *40*, *47*, *92*, *111*, *114*, *136*
tar_make_future, *76*, *80*, 80
tar_make_future(), *9*, *23*, *40*, *47*, *55*, *85*, *92*, *111*, *114*, *136*, *193*
tar_manifest, *36*, 84, *102*, *120*, *186*, *205*
tar_manifest(), *38*, *114*, *193*
tar_mermaid, *61*, 86, *208*
tar_message *(*tar_condition*)*, 17
tar_message_run *(*tar_condition*)*, 17
tar_message_validate *(*tar_condition*)*, 17
tar_meta, 90, *94, 95*, *97, 98*
tar_meta(), *68*, *70*, *91, 92*, *115*, *140*, *184*
tar_meta_delete, *92*, 93, *95*, *97, 98*
tar_meta_delete(), *92*
tar_meta_download, *92*, *94*, 94, *97, 98*
tar_meta_download(), *35*, *41*, *69*, *92*, *137*, *140*, *203*
tar_meta_sync, *92*, *94, 95*, 96, *98*
tar_meta_sync(), *35*, *41*, *69*, *92*, *97*, *137*, *140*, *203*

tar_meta_upload, 92, 94, 95, 97, 97
tar_meta_upload(), 35, 41, 69, 92, 137, 141, 204
tar_name, 5, 9, 14, 15, 33, 38, 46, 57, 62, 99, 121–124, 188, 201
tar_network, 36, 86, 100, 120, 186, 205
tar_network(), 36
tar_newer, 103, 107, 198
tar_noninteractive, 45, 65, 104, 199
tar_objects, 55, 69, 70, 105, 141
tar_older, 104, 106, 198
tar_option_get, 20, 21, 25, 27, 29, 47, 108, 109, 117
tar_option_get(), 26, 110
tar_option_reset, 20, 21, 25, 27, 29, 47, 108, 109, 117
tar_option_reset(), 115
tar_option_set, 20, 21, 25, 27, 29, 47, 108, 109, 110
tar_option_set(), 22, 31, 53, 54, 75, 79, 83, 84, 92, 94, 97, 108, 109, 116, 117, 142, 149, 154, 156, 158, 160, 161, 163–165, 167, 169–171, 173, 174, 176, 178, 195, 216
tar_outdated, 36, 86, 102, 118, 186, 205
tar_outdated(), 23, 92
tar_path, 5, 9, 14, 15, 33, 38, 46, 57, 62, 99, 121–124, 188, 201
tar_path_script, 5, 9, 14, 15, 33, 38, 46, 57, 62, 99, 121, 122–124, 188, 201
tar_path_script_support, 5, 9, 14, 15, 33, 38, 46, 57, 62, 99, 121, 122, 123, 124, 188, 201
tar_path_store, 5, 9, 14, 15, 33, 38, 46, 57, 62, 99, 121, 122, 123, 124, 188, 201
tar_path_target, 5, 9, 14, 15, 33, 38, 46, 57, 62, 99, 121–123, 123, 188, 201
tar_path_target(), 65
tar_pattern, 11–13, 125
tar_pid, 30, 127, 130
tar_poll, 16, 17, 42, 48, 128, 131, 133, 134, 187, 212, 213, 215
tar_print (tar_condition), 17
tar_process, 30, 127, 129
tar_progress, 16, 17, 42, 48, 128, 130, 133, 134, 187, 212, 213, 215
tar_progress(), 91
tar_progress_branches, 16, 17, 42, 48, 128, 131, 132, 134, 187, 212, 213, 215
tar_progress_summary, 16, 17, 42, 48, 128, 131, 133, 133, 187, 212, 213, 215
tar_progress_summary(), 128
tar_prune, 35, 41, 66, 135, 139, 204
tar_prune(), 40, 137–139
tar_prune_list, 35, 41, 66, 137, 137, 204
tar_prune_list(), 136
tar_read, 55, 69, 70, 106, 139
tar_read(), 35, 41, 65, 69, 73, 136, 137, 139, 140, 203, 204
tar_read_raw (tar_read), 139
tar_read_raw(), 139, 140
tar_renv, 43, 58, 64, 141, 178
tar_repository_cas, 143, 150, 152
tar_repository_cas(), 112, 146, 149, 150, 152, 155, 174, 191
tar_repository_cas_local, 147, 148, 152
tar_repository_cas_local(), 144, 146, 150, 152
tar_repository_cas_local_gc, 147, 150, 150
tar_reprex, 5, 153, 219, 220
tar_resources, 154, 159, 160, 162, 163, 165–167, 169, 170, 172, 173, 175, 176
tar_resources_aws, 156, 156, 160, 162, 163, 165–167, 169, 170, 172, 173, 175, 176
tar_resources_aws(), 112, 191
tar_resources_clustermq, 156, 159, 159, 162, 163, 165–167, 169, 170, 172, 173, 175, 176
tar_resources_crew, 156, 159, 160, 161, 163, 165–167, 169, 170, 172, 173, 175, 176
tar_resources_crew(), 155
tar_resources_custom_format, 156, 159, 160, 162, 162, 165–167, 169, 170, 172, 173, 175, 176
tar_resources_custom_format(), 54, 155
tar_resources_feather, 156, 159, 160, 162, 163, 164, 166, 167, 169, 170, 172, 173, 175, 176
tar_resources_feather(), 155
tar_resources_fst, 156, 159, 160, 162, 163, 165, 165, 167, 169, 170, 172, 173, 175, 176

tar_resources_future, *156*, *159, 160*, *162, 163*, *165, 166*, 166, *169, 170*, *172, 173*, *175, 176*
tar_resources_gcp, *156*, *159, 160*, *162, 163*, *165–167*, 168, *170*, *172, 173*, *175, 176*
tar_resources_network, *156*, *159, 160*, *162, 163*, *165–167*, *169*, 169, *172, 173*, *175, 176*
tar_resources_network(), *145*, *148*, *176*
tar_resources_parquet, *156*, *159, 160*, *162, 163*, *165–167*, *169, 170*, 171, *173*, *175, 176*
tar_resources_qs, *156*, *159, 160*, *162, 163*, *165–167*, *169, 170*, *172*, 172, *175, 176*
tar_resources_repository_cas, *156*, *159, 160*, *162, 163*, *165–167*, *169, 170*, *172, 173*, 174, *176*
tar_resources_repository_cas(), *145*, *147*, *155*
tar_resources_url, *156*, *159, 160*, *162, 163*, *165–167*, *169, 170*, *172, 173*, *175*, 175
tar_script, *43*, *58*, *64*, *143*, 177
tar_script(), *10*, *38*, *43*, *47*, *52*, *60*, *64*, *75*, *79*, *82*, *85*, *88*, *93*, *95, 96*, *98*, *102*, *120*, *136*, *138*, *142*, *185*, *202*, *205*, *208*, *216*
tar_seed(), *115*
tar_seed_create, 178, *181*, *183*
tar_seed_create(), *179*, *181*
tar_seed_get, *180*, 180, *183*
tar_seed_set, *180, 181*, 181
tar_seed_set(), *91*, *179–182*, *190*, *216*
tar_sitrep, *36*, *86*, *102*, *120*, 183, *205*
tar_skipped, *16, 17*, *42*, *48*, *128*, *131*, *133, 134*, 186, *212, 213*, *215*
tar_source, *5*, *9*, *14, 15*, *33*, *38*, *46*, *57*, *62*, *99*, *121–124*, 187, *201*
tar_store, *5*, *9*, *14, 15*, *33*, *38*, *46*, *57*, *62*, *99*, *121–124*, *188*, *201*
tar_target, *32*, 189
tar_target(), *10*, *22*, *31, 32*, *37*, *50*, *53, 54*, *75*, *79*, *83, 84*, *91*, *104*, *107–110*, *112, 113*, *116, 117*, *125, 126*, *142*, *149*, *154–156*, *158*, *160, 161*, *163–165*, *167–176*, *178*, *189–192*, *195*
tar_target_raw (tar_target), 189
tar_target_raw(), *84*, *108–110*, *189, 190*
tar_test, *8*, *18*, *67*, 196
tar_throw_file (tar_condition), 17
tar_throw_run (tar_condition), 17
tar_throw_validate (tar_condition), 17
tar_tidy_eval (tar_language), 66
tar_tidyselect_eval (tar_language), 66
tar_timestamp, *104*, *107*, 197
tar_timestamp(), *197, 198*
tar_timestamp_raw (tar_timestamp), 197
tar_timestamp_raw(), *197, 198*
tar_toggle, *45*, *65*, *105*, 199
tar_traceback, *72*, 200, *217, 218*
tar_unblock_process, *5*, *9*, *14, 15*, *33*, *38*, *46*, *57*, *62*, *99*, *121–124*, *188*, 201
tar_unblock_process(), *201*
tar_unscript, 202
tar_unscript(), *202*
tar_unversion, *35*, *41*, *66*, *137*, *139*, 203
tar_unversion(), *35*, *41*, *69*, *137*, *140*, *203*
tar_validate, *36*, *86*, *102*, *120*, *186*, 204
tar_visnetwork, *61*, *89*, 205
tar_visnetwork(), *13*, *22*, *58*, *84*, *92*, *111*, *114*, *193*, *209, 210*, *212*, *215*
tar_warn_deprecate (tar_condition), 17
tar_warn_run (tar_condition), 17
tar_warn_validate (tar_condition), 17
tar_warning (tar_condition), 17
tar_watch, *16, 17*, *42*, *48*, *128*, *131*, *133, 134*, *187*, 209, *213*, *215*
tar_watch(), *212*, *214*
tar_watch_server, *16, 17*, *42*, *48*, *128*, *131*, *133, 134*, *187*, *212*, 212, *215*
tar_watch_server(), *209*, *214*
tar_watch_ui, *16, 17*, *42*, *48*, *128*, *131*, *133, 134*, *187*, *212, 213*, 214
tar_watch_ui(), *209*, *212*
tar_workspace, *72*, *200*, 215, *218*
tar_workspace(), *40*, *115*, *179, 180*, *182*, *200*, *217, 218*
tar_workspaces, *72*, *200*, *217*, 217
targets-package, 5

use_targets, *5*, *153*, 218, *220*
use_targets_rmd, *5*, *153*, *219*, 219