Package ‘maestro’

February 18, 2026

Type Package

Title Orchestration of Data Pipelines

Version 1.0.1

Maintainer Will Hipson <will.e.hipson@gmail.com>

Description Framework for creating and orchestrating data pipelines. Organize, orchestrate, and monitor multiple pipelines in a single project. Use tags to decorate functions with scheduling parameters and configuration.

License MIT + file LICENSE

URL https://github.com/whipson/maestro,
https://whipson.github.io/maestro/

BugReports https://github.com/whipson/maestro/issues

Encoding UTF-8

LazyData true

Imports cli (>= 3.3.0), dplyr (>= 1.1.0), glue, lifecycle, logger, lubridate (>= 1.9.1), purrr (>= 1.0.0), R.utils, R6, rlang (>= 1.0.0), roxygen2, tictoc, timechange, utils

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Suggests asciicast, DiagrammeR, furrr, future, knitr, quarto, rmarkdown, rstudioapi, testthat (>= 3.0.0), withr

Config/testthat/edition 3

VignetteBuilder knitr, quarto

NeedsCompilation no

Author Will Hipson [cre, aut, cph] (ORCID: <https://orcid.org/0000-0002-3931-2189>), Ryan Garnett [aut, ctb, cph]

Repository CRAN

Date/Publication 2026-02-18 14:40:02 UTC

Contents

build_schedule . . . 2
create_maestro . . . 3
create_orchestrator . . . 4
create_pipeline . . . 5
get_artifacts . . . 6
get_flags . . . 7
get_run_sequence . . . 8
get_schedule . . . 9
get_slot_usage . . . 9
get_status . . . 11
invoke . . . 11
last_build_errors . . . 13
last_run_errors . . . 13
last_run_messages . . . 14
last_run_warnings . . . 14
MaestroSchedule . . . 15
maestro_tags . . . 17
run_schedule . . . 21
show_network . . . 23
suggest_orch_frequency . . . 24

Index **26**

`build_schedule` *Build a schedule table*

Description

Builds a schedule data.frame for scheduling pipelines in `run_schedule()`.

Usage

```
build_schedule(pipeline_dir = "./pipelines", quiet = FALSE)
```

Arguments

`pipeline_dir`	path to directory containing the pipeline scripts
`quiet`	silence metrics to the console (default = `FALSE`)

Details

This function parses the maestro tags of functions located in `pipeline_dir` which is conventionally called 'pipelines'. An orchestrator requires a schedule table to determine which pipelines are to run and when. Each row in a schedule table is a pipeline name and its scheduling parameters such as its frequency.

The schedule table is mostly intended to be used by `run_schedule()` immediately. In other words, it is not recommended to make changes to it.

It is recommended to build the schedule from scratch on each run of the orchestrator script rather than reusing or caching the schedule object. This is because the schedule object precomputes and stores a limited set of future run times for each pipeline.

Value

MaestroSchedule

Examples

```
# Creating a temporary directory for demo purposes! In practice, just
# create a 'pipelines' directory at the project level.

if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  build_schedule(pipeline_dir = pipeline_dir)
}
```

`create_maestro` *Creates a new maestro project*

Description

Creates a new maestro project

Usage

```
create_maestro(path, type = "R", overwrite = FALSE, quiet = FALSE, ...)
```

Arguments

`path`	file path for the orchestrator script
`type`	file type for the orchestrator (supports R, Quarto, and RMarkdown)
`overwrite`	whether to overwrite an existing orchestrator or maestro project
`quiet`	whether to silence messages in the console (default = `FALSE`)
`...`	unused

Value

invisible

Examples

```
# Creates a new maestro project with an R orchestrator
if (interactive()) {
  new_proj_dir <- tempdir()
  create_maestro(new_proj_dir, type = "R", overwrite = TRUE)

  create_maestro(new_proj_dir, type = "Quarto", overwrite = TRUE)
}
```

`create_orchestrator` *Create a new orchestrator*

Description

Create a new orchestrator

Usage

```
create_orchestrator(
  path,
  type = c("R", "Quarto", "RMarkdown"),
  open = interactive(),
  quiet = FALSE,
  overwrite = FALSE
)
```

Arguments

`path`	file path for the orchestrator script
`type`	file type for the orchestrator (supports R, Quarto, and RMarkdown)
`open`	whether or not to open the script upon creation
`quiet`	whether to silence messages in the console (default = `FALSE`)
`overwrite`	whether to overwrite an existing orchestrator or maestro project

Value

invisible

`create_pipeline` *Create a new pipeline in a pipelines directory*

Description

Allows the creation of new pipelines (R scripts) and fills in the maestro tags as specified.

Usage

```
create_pipeline(
  pipe_name,
  pipeline_dir = "pipelines",
  frequency = "1 day",
  start_time = Sys.Date(),
  tz = "UTC",
  log_level = "INFO",
  quiet = FALSE,
  open = interactive(),
  overwrite = FALSE,
  skip = FALSE,
  inputs = NULL,
  outputs = NULL,
  priority = NULL
)
```

Arguments

`pipe_name`	name of the pipeline and function
`pipeline_dir`	directory containing the pipeline scripts
`frequency`	how often the pipeline should run (e.g., 1 day, daily, 3 hours, 4 months). Fills in maestroFrequency tag
`start_time`	start time of the pipeline schedule. Fills in maestroStartTime tag
`tz`	timezone that pipeline will be scheduled in. Fills in maestroTz tag
`log_level`	log level for the pipeline (e.g., INFO, WARN, ERROR). Fills in maestroLogLevel tag
`quiet`	whether to silence messages in the console (default = `FALSE`)
`open`	whether or not to open the script upon creation
`overwrite`	whether or not to overwrite an existing pipeline of the same name and location.
`skip`	whether to skip the pipeline when running in the orchestrator (default = `FALSE`)
`inputs`	vector of names of pipelines that input into this pipeline (default = `NULL` for no inputs)
`outputs`	vector of names of pipelines that receive output from this pipeline (default = `NULL` for no outputs)
`priority`	a single positive integer corresponding to the order in which this pipeline will be invoked in the presence of other simultaneously invoked pipelines.

Value

invisible

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline(
    "extract_data",
    pipeline_dir = pipeline_dir,
    frequency = "1 hour",
    open = FALSE,
    quiet = TRUE,
    overwrite = TRUE
  )

  create_pipeline(
    "new_job",
    pipeline_dir = pipeline_dir,
    frequency = "20 minutes",
    start_time = as.POSIXct("2024-06-21 12:20:00"),
    log_level = "ERROR",
    open = FALSE,
    quiet = TRUE,
    overwrite = TRUE
  )
}
```

`get_artifacts` *Get the artifacts (return values) of the pipelines in a MaestroSchedule object.*

Description

Artifacts are return values from pipelines. They are accessible as a named list where the names correspond to the names of the pipeline.

Usage

```
get_artifacts(schedule)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`

Value

named list

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  schedule <- run_schedule(
    schedule,
    orch_frequency = "1 day",
    quiet = TRUE
  )

  get_artifacts(schedule)

  # Alternatively, use the underlying R6 method
  schedule$get_artifacts()
}
```

`get_flags` *Get the flags of pipelines in a MaestroSchedule object*

Description

Creates a long data.frame where each row is a flag for each pipeline.

Usage

```
get_flags(schedule)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`

Value

data.frame

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  get_flags(schedule)

  # Alternatively, use the underlying R6 method
  schedule$get_flags()
}
```

`get_run_sequence` *Get the run sequence of a schedule*

Description

Retrieves the scheduled run times for a given schedule, with optional filtering by number of runs and datetime range.

Usage

```
get_run_sequence(
  schedule,
  n = NULL,
  min_datetime = NULL,
  max_datetime = NULL,
  include_only_primary = FALSE
)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`
`n`	Optional positive integer. If specified, returns only the first `n` runs for each pipeline.
`min_datetime`	Optional minimum datetime filter. Can be a lubridate::Date or lubridate::POSIXct object. If specified, only returns runs scheduled at or after this datetime.
`max_datetime`	Optional maximum datetime filter. Can be a lubridate::Date or lubridate::POSIXct object. If specified, only returns runs scheduled at or before this datetime.
`include_only_primary`	only primary pipelines are included (this are pipelines that are scheduled and not downstream nodes in a DAG)

Value

A vector of datetime values representing the scheduled run times.

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  get_run_sequence(schedule)

  # Alternatively, use the underlying R6 method
  schedule$get_run_sequence()
}
```

`get_schedule` *Get the schedule from a MaestroSchedule object*

Description

A schedule is represented as a table where each row is a pipeline and the columns contain scheduling parameters such as the frequency and start time.

Usage

```
get_schedule(schedule)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`

Details

The schedule table is used internally in a MaestroSchedule object but can be accessed using this function or accessing the R6 method of the MaestroSchedule object.

Value

data.frame

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  get_schedule(schedule)

  # Alternatively, use the underlying R6 method
  schedule$get_schedule()
}
```

`get_slot_usage` *Get time slot usage of a schedule*

Description

Get the number of pipelines scheduled to run for each time slot at a particular slot interval. Time slots are times that the orchestrator runs and the slot interval determines the level of granularity to consider.

Usage

```
get_slot_usage(schedule, orch_frequency, slot_interval = "hour")
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`
`orch_frequency`	of the orchestrator, a single string formatted like "1 day", "2 weeks", "hourly", etc.
`slot_interval`	a time unit indicating the interval of time to consider between slots (e.g., 'hour', 'day')

Details

This function is particularly useful when you have multiple pipelines in a project and you want to see what recurring time intervals may be available or underused for new pipelines.

Note that this function is intended for use in an interactive session while developing a maestro project. It is not intended for use in the orchestrator.

As an example, consider we have four pipelines running at various frequencies and the orchestrator running every hour. Then let's say there's to be a new pipeline that runs every day. One might ask 'what hour should I schedule this new pipeline to run on?'. By using `get_slot_usage(schedule, orch_frequency = '1 hour', slot_interval = 'hour')` on the existing schedule, you could identify for each hour how many pipelines are already scheduled to run and choose the ones with the lowest usage.

Value

data.frame

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  get_slot_usage(
    schedule,
    orch_frequency = "1 hour",
    slot_interval = "hour"
  )
}
```

`get_status` *Get the statuses of the pipelines in a MaestroSchedule object*

Description

A status data.frame contains the names and locations of the pipelines as well as information around whether they were invoked, the status (error, warning, etc.), and the run time.

Usage

```
get_status(schedule)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`

Value

data.frame

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  schedule <- run_schedule(
    schedule,
    orch_frequency = "1 day",
    quiet = TRUE
  )

  get_status(schedule)

  # Alternatively, use the underlying R6 method
  schedule$get_status()
}
```

`invoke` *Manually run a pipeline regardless of schedule*

Description

Instantly run a single pipeline from the schedule. This is useful for testing purposes or if you want to just run something one-off.

Usage

```
invoke(
  schedule,
  pipe_name,
  resources = list(),
  ...,
  quiet = TRUE,
  log_to_console = FALSE
)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`
`pipe_name`	name of a single pipe name from the schedule
`resources`	named list of shared resources made available to pipelines as needed
`...`	other arguments passed to `run_schedule()`
`quiet`	silence metrics to the console (default = `FALSE`). Note this does not affect messages generated from pipelines when `log_to_console` = `TRUE`.
`log_to_console`	whether or not to include pipeline messages, warnings, errors to the console (default = `FALSE`) (see Logging & Console Output section)

Details

Scheduling parameters such as the frequency, start time, and specifiers are ignored. The pipeline will be run even if `maestroSkip` is present. If the pipeline is a DAG pipeline, `invoke` will attempt to execute the full DAG.

Value

invisible

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  invoke(schedule, "my_new_pipeline")
}
```

`last_build_errors` *Retrieve latest maestro build errors*

Description

Gets the latest schedule build errors following use of `build_schedule()`. If the build succeeded or `build_schedule()` has not been run it will be `NULL`.

Usage

```
last_build_errors()
```

Value

error messages

Examples

```
last_build_errors()
```

`last_run_errors` *Retrieve latest maestro pipeline errors*

Description

Gets the latest pipeline errors following use of `run_schedule()`. If the all runs succeeded or `run_schedule()` has not been run it will be `NULL`.

Usage

```
last_run_errors()
```

Value

error messages

Examples

```
last_run_errors()
```

`last_run_messages` *Retrieve latest maestro pipeline messages*

Description

Gets the latest pipeline messages following use of `run_schedule()`. If there are no messages or `run_schedule()` has not been run it will be `NULL`.

Usage

```
last_run_messages()
```

Details

Note that setting `maestroLogLevel` to something greater than `INFO` will ignore messages.

Value

messages

Examples

```
last_run_messages()
```

`last_run_warnings` *Retrieve latest maestro pipeline warnings*

Description

Gets the latest pipeline warnings following use of `run_schedule()`. If there are no warnings or `run_schedule()` has not been run it will be `NULL`.

Usage

```
last_run_warnings()
```

Details

Note that setting `maestroLogLevel` to something greater than `WARN` will ignore warnings.

Value

warning messages

Examples

```
last_run_warnings()
```

`MaestroSchedule` *Class for a schedule of pipelines*

Description

Class for a schedule of pipelines

Class for a schedule of pipelines

Public fields

`PipelineList` object of type MaestroPipelineList

Methods

Public methods:

- `MaestroSchedule$new()`
- `MaestroSchedule$print()`
- `MaestroSchedule$run()`
- `MaestroSchedule$get_schedule()`
- `MaestroSchedule$get_status()`
- `MaestroSchedule$get_artifacts()`
- `MaestroSchedule$get_network()`
- `MaestroSchedule$get_flags()`
- `MaestroSchedule$show_network()`
- `MaestroSchedule$get_run_sequence()`
- `MaestroSchedule$clone()`

Method `new()`: Create a MaestroSchedule object

Usage:

```
MaestroSchedule$new(Pipelines = NULL)
```

Arguments:

`Pipelines` list of MaestroPipelines

Returns: MaestroSchedule

Method `print()`: Print the schedule object

Usage:

```
MaestroSchedule$print()
```

Returns: print

Method `run()`: Run a MaestroSchedule

Usage:

```
MaestroSchedule$run(..., quiet = FALSE, run_all = FALSE, n_show_next = 5)
```

Arguments:

`...` arguments passed to MaestroPipelineList$run

`quiet` whether or not to silence console messages

`run_all` run all pipelines regardless of the schedule (default is `FALSE`) - useful for testing.

`n_show_next` show the next n scheduled pipes

Returns: invisible

Method `get_schedule()`**:** Get the schedule as a data.frame

Usage:

```
MaestroSchedule$get_schedule()
```

Returns: data.frame

Method `get_status()`**:** Get status of the pipelines as a data.frame

Usage:

```
MaestroSchedule$get_status()
```

Returns: data.frame

Method `get_artifacts()`**:** Get artifacts (return values) from the pipelines

Usage:

```
MaestroSchedule$get_artifacts()
```

Returns: list

Method `get_network()`**:** Get the network structure of the pipelines as an edge list (will be empty if there are no DAG pipelines)

Usage:

```
MaestroSchedule$get_network()
```

Returns: data.frame

Method `get_flags()`**:** Get all pipeline flags as a long data.frame

Usage:

```
MaestroSchedule$get_flags()
```

Returns: data.frame

Method `show_network()`**:** Visualize the DAG relationships between pipelines in the schedule

Usage:

```
MaestroSchedule$show_network()
```

Returns: interactive visualization

Method `get_run_sequence()`**:** Get full sequence of scheduled executions for all pipelines

Usage:

```
MaestroSchedule$get_run_sequence(
  n = NULL,
  min_datetime = NULL,
  max_datetime = NULL,
  include_only_primary = FALSE
)
```

Arguments:

`n` optional sequence limit

`min_datetime` optional minimum datetime

`max_datetime` optional maximum datetime

`include_only_primary` only primary pipelines are included (this are pipelines that are scheduled and not downstream nodes in a DAG)

Returns: data.frame

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MaestroSchedule$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)
}
```

`maestro_tags` *Maestro Tags*

Description

maestro tags are roxygen2 comments for configuring the scheduling and execution of pipelines.

Details

maestro tags follow the format: `#' @maestroTagName value`

Some tags may not take a value.

maestro tags must be written above the function that is to be included as a pipeline. A typical pipeline with tags could look like this:

```
#' @maestroFrequency 1 hour
#' @maestroStartTime 12:30:00
#' @maestroLogLevel WARN
my_pipeline <- function() {

   # Pipeline code
}
```

Below are descriptions of all the tags currently available in maestro along with examples.

maestroFrequency

How often to run the pipeline. This tag takes a time unit indicating how long to wait between subsequent runs of the pipeline. Acceptable values include an integer value followed by one of minute(s), hour(s), day(s), week(s), month(s), and year(s). Note that some combinations of integer + unit may be invalid. Adverbs like 'hourly', 'daily', 'weekly', etc. are also valid.

Default: `1 day`

Examples:

- `#' @maestroFrequency 1 hour`
- `#' @maestroFrequency daily`
- `#' @maestroFrequency 2 weeks`
- `#' @maestroFrequency 3 months`

maestroStartTime

Timestamp, date, or time corresponding to the start time of the pipeline. This also sets the cadence of the pipeline in some cases. For instance, if the start time is `2025-03-18 03:00:00` and the frequency is daily, the pipeline will run at 03:00 every day. A value in the future prevents the pipeline from running until that time has been reached.

Default: `2024-01-01 00:00:00`

Examples:

- `#' @maestroStartTime 2025-02-05 12:00:00`
- `#' @maestroStartTime 2025-01-01`
- `#' @maestroStartTime 08:00:00`

maestroTz

Timezone in which the maestroStartTime is to be considered. Takes any valid timezone string that can be found in `OlsonNames()`.

Default: `UTC`

Examples:

- `#' @maestroTz Europe/Paris`
- `#' @maestroTz America/Halifax`
- `#' @maestroTz US/Pacific`

maestroLogLevel

Minimum logging threshold for messages, warnings, and errors that come from the pipeline. These levels correspond to those in `logger:::log_levels_supported` but most commonly used are ERROR, WARN, INFO. For example, if you use `WARN` then any messages of lower urgency (i.e., INFO) will be suppressed, but errors will be logged.

Default: `INFO`

Examples:

- `#' @maestroLogLevel ERROR`
- `#' @maestroLogLevel WARN`

maestroHours

Specific hours of the day in which to run the pipeline. This only applies for pipelines that run at an hourly or minutely frequency. Acceptable values are integers from 0-23 separated by spaces. If empty, pipeline runs at all hours. This tag uses the timezone specified by `maestroTz` (will be UTC if empty).

Default:

Examples:

- `#' @maestroHours 1 4 7 10`
- `#' @maestroHours 0 5 20`

maestroDays

Specific days of week or days of month on which to run the pipeline. This only applies for pipelines that run at a minutely, hourly, or daily frequency. Acceptable values are either integers from 1-31 or day of week strings like Mon, Tue, Wed, etc. These two options cannot be used in combination.

Default:

Examples

- `#' @maestroDays 1 11 21 31`
- `#' @maestroDays Mon Tue Wed Thu Fri`

maestroMonths

Specific months of the year on which to run the pipeline. This only applies for pipelines that do run at least monthly. Acceptable values are integers (1-12) corresponding to the month of the year (e.g., 1 = January, 2 = February, etc.).

Default:

Examples:

- `#' @maestroMonths 3 8 12`
- `#' @maestroMonths 10`

maestroInputs

For a DAG style pipeline, the names of pipelines that input into the pipeline. These names must match the function name defining the inputting pipeline. Multiple pipelines can be used as inputs and the input value is used in the target pipeline via the required `.input` parameter. Note that this tag could be redundant if the inputting pipeline uses `maestroOutputs`.

Default:

Examples:

- `#' @maestroInputs extract verify`

maestroOutputs

For a DAG style pipeline, the names of pipelines that receive the return value of this pipeline as input. These names must match the function name defining the outputting pipeline. Multiple pipelines can be outputted into. The return value of the pipeline will be passed into the receiving pipeline. Note that this tag could be redundant if pipeline to be inputted into uses `maestroInputs`.

Default:

Examples:

- `#' @maestroOutputs transform`

maestroSkip

Flags a pipeline to never be executed even if it is scheduled to run. This can be useful when developing or testing a pipeline. This tag takes no value, instead the presence of the tag indicates whether it is skipped. This tag is ignored when `run_schedule(..., run_all = TRUE)` or when using `invoke()`.

Default:

Examples:

- `#' @maestroSkip`

maestroPriority

Determines the order in which pipelines that run at the same scheduled instance are executed. Values are positive integers from 1 to N. Order is determined in descending order such that 1 indicates the highest priority. Pipelines with the same priority run in the order in which `build_schedule()` parses the pipeline (usually alphabetical according to file name and then line number within file). By default, all pipelines are given equal priority.

Default:

Examples:

- `#' @maestroPriority 1`
- `#' @maestroPriority 3`

### maestroFlags

Arbitrary labeling tags which are then made accessible via `get_flags()`. A pipeline can have multiple tags separated by spaces.

Default:

Examples:

- `#' @maestroFlags critical etl cloud`
- `#' @maestroFlags aviation`

### maestro

Generic tag for identifying a maestro pipeline with all the defaults. Useful when you want a pipeline to be scheduled via maestro that accepts all default tag values. Only use this tag if you have no other maestro tags. The tag takes no value.

Default:

Examples:

- `#' @maestro`

`run_schedule` *Run a schedule*

Description

Given a schedule in a `maestro` project, runs the pipelines that are scheduled to execute based on the current time.

Usage

```
run_schedule(
  schedule,
  orch_frequency = "1 day",
  check_datetime = lubridate::now(tzone = "UTC"),
  resources = list(),
  run_all = FALSE,
  n_show_next = 5,
  cores = 1,
  log_file_max_bytes = 1e+06,
  quiet = FALSE,
  log_to_console = FALSE,
  log_to_file = FALSE
)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`
`orch_frequency`	of the orchestrator, a single string formatted like "1 day", "2 weeks", "hourly", etc.
`check_datetime`	datetime against which to check the running of pipelines (default is current system time in UTC)
`resources`	named list of shared resources made available to pipelines as needed
`run_all`	run all pipelines regardless of the schedule (default is `FALSE`) - useful for testing. Does not apply to pipes with a `maestroSkip` tag. Conditional pipelines using `maestroRunIf` still behave according to their condition.
`n_show_next`	show the next n scheduled pipes
`cores`	number of cpu cores to run if running in parallel. If > 1, `furrr` is used and a multisession plan must be executed in the orchestrator (see details)
`log_file_max_bytes`	numeric specifying the maximum number of bytes allowed in the log file before purging the log (within a margin of error)
`quiet`	silence metrics to the console (default = `FALSE`). Note this does not affect messages generated from pipelines when `log_to_console = TRUE`.
`log_to_console`	whether or not to include pipeline messages, warnings, errors to the console (default = `FALSE`) (see Logging & Console Output section)
`log_to_file`	either a boolean to indicate whether to create and append to a `maestro.log` or a character path to a specific log file. If `FALSE` or `NULL` it will not log to a file.

Details

Pipeline schedule logic:

The function `run_schedule()` examines each pipeline in the schedule table and determines whether it is scheduled to run at the current time using some simple time arithmetic. We assume `run_schedule(schedule, check_datetime = Sys.time())`, but this need not be the case.

Output:

`run_schedule()` returns the same MaestroSchedule object with modified attributes. Use `get_status()` to examine the status of each pipeline and use `get_artifacts()` to get any return values from the pipelines as a list.

Pipelines with arguments (resources):

If a pipeline takes an argument that doesn't include a default value, these can be supplied in the orchestrator via `run_schedule(resources = list(arg1 = val))`. The name of the argument used by the pipeline must match the name of the argument in the list. Currently, each named resource must refer to a single object. In other words, you can't have two pipes using the same argument but requiring different values.

Running in parallel:

Pipelines can be run in parallel using the `cores` argument. First, you must run `future::plan(future::multisession)` in the orchestrator. Then, supply the desired number of cores to the `cores` argument. Note that console output appears different in multicore mode.

Logging & Console Output:

By default, maestro suppresses pipeline messages, warnings, and errors from appearing in the console, but messages coming from print() and other console logging packages like cli and logger are not suppressed and will be interwoven into the output generated from run_schedule(). Messages from cat() and related functions are always suppressed due to the nature of how those functions operate with standard output.

Users are advised to make use of R's message(), warning(), and stop() functions in their pipelines for managing conditions. Use log_to_console = TRUE to print these to the console.

Note that log_to_console does not work when running multicore. In the case of multicore, it is recommended to use log_to_file.

Maestro can generate a log file that is appended to each time the orchestrator is run. Use log_to_file = TRUE or log_to_file = '[path-to-file]' and maestro will create/append to a file in the project directory. This log file will be appended to until it exceeds the byte size defined in log_file_max_bytes argument after which the log file is deleted.

Value

MaestroSchedule object

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  # Runs the schedule every 1 day
  run_schedule(
    schedule,
    orch_frequency = "1 day",
    quiet = TRUE
  )

  # Runs the schedule every 15 minutes
  run_schedule(
    schedule,
    orch_frequency = "15 minutes",
    quiet = TRUE
  )
}
```

---

show_network *Visualize the schedule as a DAG*

Description

Create an interactive network visualization to show the dependency structure of pipelines in the schedule. This is only useful if there are pipelines in the schedule that take inputs/outputs from other pipelines.

Usage

```
show_network(schedule)
```

Arguments

`schedule`	object of type MaestroSchedule created using `build_schedule()`

Details

Note that running this function on a schedule with all independent pipelines will produce a network visual with no connections.

This function requires the installation of `DiagrammeR` which is not automatically installed with `maestro`.

Value

DiagrammeR visualization

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)

  schedule <- run_schedule(
    schedule,
    orch_frequency = "1 day",
    quiet = TRUE
  )

  show_network(schedule)
}
```

`suggest_orch_frequency` *Suggest orchestrator frequency based on a schedule*

Description

Suggests a frequency to run the orchestrator based on the frequencies of the pipelines in a schedule.

Usage

```
suggest_orch_frequency(
  schedule,
  check_datetime = lubridate::now(tzone = "UTC")
)
```

Arguments

`schedule`	MaestroSchedule object created by `build_schedule()`
`check_datetime`	datetime against which to check the running of pipelines (default is current system time in UTC)

Details

This function attempts to find the smallest interval of time between all pipelines. If the smallest interval is less than 15 minutes, it just uses the smallest interval.

Note this function is intended to be used interactively when deciding how often to schedule the orchestrator. Programmatic use is not recommended.

Value

frequency string

Examples

```
if (interactive()) {
  pipeline_dir <- tempdir()
  create_pipeline("my_new_pipeline", pipeline_dir, open = FALSE)
  schedule <- build_schedule(pipeline_dir = pipeline_dir)
  suggest_orch_frequency(schedule)
}
```

Index

build_schedule, 2

create_maestro, 3
create_orchestrator, 4
create_pipeline, 5

get_artifacts, 6
get_flags, 7
get_run_sequence, 8
get_schedule, 9
get_slot_usage, 9
get_status, 11

invoke, 11

last_build_errors, 13
last_run_errors, 13
last_run_messages, 14
last_run_warnings, 14
lubridate::Date, *8*
lubridate::POSIXct, *8*

maestro_tags, 17
MaestroSchedule, 15

run_schedule, 21

show_network, 23
suggest_orch_frequency, 24