

Package: epwshiftr (via r-universe)

October 12, 2026

Title Create Future 'EnergyPlus' Weather Files using 'CMIP6' Data

Version 0.1.4.9001

Description Query, download climate change projection data from the 'CMIP6' (Coupled Model Intercomparison Project Phase 6) project <<https://pcmdi.llnl.gov/CMIP6/>> in the 'ESGF' (Earth System Grid Federation) platform <<https://esgf.llnl.gov>>, and create future 'EnergyPlus' <<https://energyplus.net>> Weather ('EPW') files adjusted from climate changes using data from Global Climate Models ('GCM').

Imports checkmate (>= 2.0.0), cli (>= 3.4.0), curl, data.table (>= 1.12.4), duckdb, jsonlite, mirai (>= 2.7.3), R6, RNetCDF, S7

Suggests testthat (>= 3.0.0), webfakes, withr

License MIT + file LICENSE

Encoding UTF-8

URL <https://ideas-lab-nus.github.io/epwshiftr/>,
<https://github.com/ideas-lab-nus/epwshiftr>

BugReports <https://github.com/ideas-lab-nus/epwshiftr/issues>

Roxygen list(markdown = TRUE)

Collate 'weather-input.R' 'weather-component.R' 'weather-signal.R'
'signal-adjustment.R' 'component-temperature-epw.R'
'adapter-daily-adjusted-epw.R' 'weather-sequence.R'
'adapter-direct-model-calendar.R' 'weather-solar.R'
'epw-physics.R' 'epw-file.R' 'epw-morph-context.R'
'adapter-direct-model-epw.R' 'utils.R' 'source-cds.R'
'source-reanalysis.R' 'source-era5.R' 'weather-pipeline.R'
'weather-recipe.R' 'calendar-subdaily.R'
'method-solar-radiation-interpolation.R'
'method-temporal-interpolation.R'
'component-hourly-weather-interpolation.R'
'component-hourly-kqdm-input.R' 'backend-hourly-kernel-qdm.R'
'backend-registry.R' 'epw-morph-recipe.R' 'weather-transform.R'
'backend-original-morphing.R' 'store.R' 'epw-morpher.R'

'query-result.R' 'solr-date.R' 'query-param.R' 'query.R'
 'shift-stage.R' 'adapter-era-cf.R' 'method-bws.R'
 'method-btws.R' 'weather-temperature.R' 'backend-bws-btws.R'
 'backend-daily-temperature.R' 'backend-ek-daily-temperature.R'
 'backend-quantile-mapping-morphing.R' 'backend-sobie-curry.R'
 'calendar-daily.R' 'cli-doctor.R' 'cli-download.R'
 'cli-extract.R' 'cli-help.R' 'cli-launcher.R' 'cli-morph.R'
 'cli-parse.R' 'cli-query.R' 'cli-render.R' 'cli-retry.R'
 'cli-shift-config.R' 'cli-shift-watch.R' 'cli-shift.R'
 'cli-storage.R' 'cli.R' 'cmip6-availability.R'
 'cmip6-eligibility.R' 'component-btws.R' 'component-bws-btws.R'
 'component-hourly-calendar-grouping.R' 'data-node.R'
 'dataset-regions.R' 'dataset.R' 'dict-check.R' 'dict-fetch.R'
 'dict-parse-cv.R' 'dict-parse-dreq.R' 'dict-print.R'
 'dict-store.R' 'dict.R' 'distribution-parametric.R'
 'distribution-quantile.R' 'epw-output.R' 'epwshiftr-package.R'
 'era5-epw-source.R' 'era5-epw.R' 'geo.R' 'gh.R'
 'import-standalone-schema.R' 'method-cdf-transform.R'
 'method-daily-temperature.R' 'method-delta-change.R'
 'method-equidistant-cdf-matching.R' 'method-isimip3basd.R'
 'method-kernel-qdm.R' 'method-linear-scaling.R'
 'method-quantile-delta-mapping.R' 'method-quantile-mapping.R'
 'method-scaled-distribution-mapping.R' 'netcdf.R'
 'print-truncation.R' 'query-result-access.R'
 'query-result-plan.R' 'query-result-replicas.R' 'schema.R'
 'shift-batch-execution.R' 'shift-batch-plan.R'
 'shift-batch-read.R' 'shift-batch-ui.R' 'shift-batch-window.R'
 'shift-batch.R' 'shift-climate.R' 'shift-execution.R'
 'shift-export.R' 'shift-inspect.R' 'shift-job.R' 'shift-path.R'
 'shift-persist.R' 'shift-print.R' 'shift-reference.R'
 'shift-ui.R' 'shift-reporter.R' 'shift-resolve.R' 'shift-run.R'
 'shift-selection.R' 'shift-spec.R' 'shift-tui.R'
 'shift-ui-state.R' 'shift-ui-view.R' 'source-read.R'
 'standalone-cache.R' 'standalone-downloader.R'
 'weather-method.R' 'zzz.R'

Config/testthat/edition 3

Config/Needs/website r-lib/asciicast, r-lib/pkgload

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libnetcdf-dev libssl-dev libudunits2-dev xz-utils

Repository <https://hongyuanjia.r-universe.dev>

Date/Publication 2026-10-12 15:17:11 UTC

RemoteUrl <https://github.com/ideas-lab-nus/epwshiftr>

RemoteRef HEAD

RemoteSha baab459fb4d1452485aaa1b4d2a2c1879b4dd5b6

Contents

epwshiftr-package	4
[.EsgResult	5
data_node_status	5
Downloader	6
epw_morph_backend	20
epw_morph_backends	20
epw_morph_periods	21
epw_morph_register_backend	21
epw_morph_result	22
epw_morpher	22
EpwMorphBackend	23
EpwMorpher	26
epwshiftr_cli	31
esg_result	32
EsgDataset	33
EsgDict	44
esgdict_check	56
esgdict_option	57
EsgQuery	58
EsgStore	75
install_cli	92
is.solr_date	92
monthly_transform	93
shift_batch_get	94
shift_cmip6_avail	94
shift_collect	96
shift_epw_reanalysis	106
shift_era5	108
shift_era6	109
shift_history	110
shift_selection	111
shift_summary	112
solr_date	112
store_dir	114
store_reset	115
uninstall_cli	115
weather_transforms	116

epwshiftr-package

epwshiftr: Create future EnergyPlus Weather files using CMIP6 data

Description

Query, download climate change projection data from the **CMIP6 (Coupled Model Intercomparison Project Phase 6) project** in the **ESGF (Earth System Grid Federation) platform**, and create future **EnergyPlus Weather (EPW)** files adjusted from climate changes using data from Global Climate Models (GCM).

Package options

- `epwshiftr.verbose`: If TRUE, more detailed message will be printed. Default: FALSE.
- `epwshiftr.progress`: If TRUE, progress bars are shown for long-running operations that support them. Default: `interactive()`.
- `epwshiftr.threshold_alpha`: the threshold of the absolute value for alpha, i.e. monthly-mean fractional change, when performing morphing operations. The default value is set to 3. If the morphing methods are set "stretch" or "combined", and the absolute alpha exceeds the threshold value, warnings are issued and the morphing method fallbacks to "shift" to avoid unrealistic morphed values.
- `epwshiftr.dir_store`: The persistent store directory for query snapshots, dictionaries, source mirrors, downloads, extraction results, outputs, and the store manifest. If not set, `tools::R_user_dir()` with type "data" will be used.
- `epwshiftr.cache`: Controls caching behavior. TRUE enables normal caching (default), FALSE disables caching entirely, and "offline" enables offline mode where only cached data is used and no network requests are made. Default: TRUE
- `epwshiftr.dir_cache`: The directory for disposable cache entries. Deleting this directory can require re-fetching or re-parsing data, but should not invalidate a persistent store.
- `epwshiftr.mirai_workers`: Maximum concurrent source readers, default 4. Set to 1 for serial reads. Single-task and batch workflows share this limit.
- `epwshiftr.ui_height`: Optional terminal display height; unset by default.
- `epwshiftr.cache_max_size`: Disposable cache capacity in bytes, default 1 GiB.
- `epwshiftr.cache_max_age`: Disposable cache lifetime in seconds, default 1800.
- `epwshiftr.cache_max_n`: Maximum cache entries, default Inf. Cache limits are applied when the session's cache is first initialized.
- `epwshiftr.query.timeout`: Catalog request timeout in seconds, default 300.
- `epwshiftr.query.connect_timeout`: Catalog connection timeout in seconds, default 30.

Execution settings are captured when a workflow attempt starts and forwarded to detached processes and source workers. They do not change scientific plan identities. Development and test dependency substitutions are not options.

Author(s)

Hongyuan Jia

See Also

Useful links:

- <https://ideas-lab-nus.github.io/epwshiftr/>
- <https://github.com/ideas-lab-nus/epwshiftr>
- Report bugs at <https://github.com/ideas-lab-nus/epwshiftr/issues>

[.EsgResult

*Subset an ESGF query result***Description**

[is a one-dimensional shortcut for `x$slice(i)`.

Usage

```
## S3 method for class 'EsgResult'
x[i, j, ..., drop = FALSE]
```

Arguments

<code>x</code>	An EsgResult object.
<code>i</code>	A row selector accepted by <code>x\$slice(i)</code> .
<code>j, ..., drop</code>	Unsupported.

Value

A new result object of the same type, or `x` for `x[]`.

data_node_status

*Get status of ESGF data nodes***Description**

`data_node_status()` is the user-facing replacement for the legacy data-node helper name used in earlier releases.

Usage

```
data_node_status(
  speed_test = FALSE,
  timeout = 3,
  index_node = INDEX_NODES[["ORNL"]]
)
```

Arguments

speed_test	If TRUE, perform a lightweight HTTP probe on each UP data node. A probe_ms column is appended in returned data.table which stores elapsed request time in milliseconds. Default: FALSE.
timeout	Timeout for each HTTP probe in seconds. Default: 3.
index_node	The index node to query for data-node status. Default: INDEX_NODES[["ORNL"]].

Value

A `data.table::data.table()` of 2 or 3 (when speed_test is TRUE) columns:

Column	Type	Description
data_node	character	Web address of data node
status	character	Status of data node. "UP" means OK and "DOWN" means currently not available
probe_ms	double	HTTP probe elapsed time in milliseconds for UP data nodes

Examples

```
## Not run:
data_node_status()

## End(Not run)
```

 Downloader

General Purpose File Downloader

Description

The implementation in `R/standalone-downloader.R` can be copied and sourced on its own, with its qualified package dependencies installed. Detached jobs retain the copied source path, so keep that file available until the jobs finish. Integration with a host's cache policy and store is optional and supplied by the host package; ordinary downloads do not require `epwshiftr` to be loaded.

Downloader provides a general purpose file download system with:

- File status management (missing, downloading, downloaded, verified)
- Incremental checksum verification during download
- Resume capability for interrupted downloads
- Async and parallel download using `mirai`
- Progress tracking
- Error handling and retry logic

Active bindings

`data_dir` The final data directory
`tmp_dir` The temporary files directory
`max_retries` Maximum number of retry attempts
`timeout` Download timeout in seconds
`network_policy` Network options passed to libcurl.
`node_policy` Data-node cooldown and ranking policy.
`transfer_policy` Curl transfer policy.
`resource_policy` Local resource and scheduling policy.
`n_workers` Number of parallel workers
`manifest` Persistent download manifest path, or NULL.
`config` Current downloader configuration as a named list. When manifest is set, the configuration is stored in the manifest's `download_config` table.

Methods**Public methods:**

- `Downloader$new()`
- `Downloader$download()`
- `Downloader$enqueue()`
- `Downloader$preflight()`
- `Downloader$run()`
- `Downloader$start()`
- `Downloader$jobs()`
- `Downloader$job_status()`
- `Downloader$job_logs()`
- `Downloader$stop_job()`
- `Downloader$daemon_start()`
- `Downloader$daemon_status()`
- `Downloader$daemon_stop()`
- `Downloader$sessions()`
- `Downloader$tasks()`
- `Downloader$status()`
- `Downloader$events()`
- `Downloader$on()`
- `Downloader$off()`
- `Downloader$data_nodes()`
- `Downloader$reset_data_nodes()`
- `Downloader$record_probes()`
- `Downloader$retry()`
- `Downloader$cancel()`

- `Downloader$resume()`
- `Downloader$verify()`
- `Downloader$cleanup_tmp()`
- `Downloader$get_tasks()`
- `Downloader$get_task_status()`
- `Downloader$wait_for_tasks()`
- `Downloader$cancel_task()`
- `Downloader$list_incomplete()`
- `Downloader$verify_checksum()`
- `Downloader$print()`
- `Downloader$clone()`

Method `new()`: Create a new Downloader object

Usage:

```
Downloader$new(
  dest = NULL,
  temp = NULL,
  retries = 3L,
  timeout = 3600L,
  ssl_verifypeer = TRUE,
  proxy = NULL,
  connect_timeout = NULL,
  useragent = NULL,
  cleanup = TRUE,
  n_workers = 4L,
  node_policy = NULL,
  transfer_policy = NULL,
  resource_policy = NULL,
  manifest = NULL
)
```

Arguments:

- `dest` A string specifying the directory for final downloaded files. If NULL, uses a temporary directory. Default: NULL.
- `temp` A string specifying the directory for temporary files (.part, .done). If NULL, uses `dest/.tmp`. Should ideally be on the same filesystem as `dest` for atomic rename operations. Default: NULL.
- `retries` A positive integer specifying the maximum number of retry attempts for failed downloads. Default: 3L.
- `timeout` A positive integer specifying the timeout in seconds for each download. Default: 3600L (1 hour).
- `ssl_verifypeer` Whether to verify HTTPS certificates. Default: TRUE.
- `proxy` Optional proxy URL passed to libcurl. Default: NULL.
- `connect_timeout` Optional connection timeout in seconds passed to libcurl. Default: NULL.
- `useragent` Optional HTTP user agent passed to libcurl. Default: NULL.

cleanup A logical value specifying whether to automatically clean up failed temporary files. Default: TRUE.

n_workers A non-negative integer specifying the number of parallel workers for async downloads. If 0, async downloads will fallback to synchronous mode. Default: 4L.

node_policy A list controlling historical data-node cooldown and ranking. Missing fields use conservative defaults.

transfer_policy A list controlling curl transfer options and optional experimental Range-piece downloads. Supported curl fields are `chunk_size`, `bandwidth_limit`, `low_speed_limit`, and `low_speed_time`. Range fields are `range_mode` ("off", "single", "multi", or "auto"), `piece_size`, `piece_concurrency`, `max_sources`, `require_checksum_for_multisource`, and `range_probe_timeout`. The default `range_mode = "off"` keeps the existing streaming download behavior.

resource_policy A list controlling local resource checks and scheduling. Supported fields are `host_concurrency`, `disk_preflight`, and `min_free_space`.

manifest Optional DuckDB manifest path for persistent sessions, tasks, candidate URLs, and events. If NULL, only the single-file shortcut API is available. Default: NULL.

Returns: An Downloader object.

Examples:

```
\dontrun{
dl <- Downloader$new()
dl <- Downloader$new(dest = "~/data")
dl <- Downloader$new(
  dest = "~/data",
  temp = "~/data/.tmp",
  n_workers = 8
)
}
```

Method `download()`: Download a single file with state management and resume support

Usage:

```
Downloader$download(
  url,
  filename = NULL,
  subdir = NULL,
  progress = TRUE,
  overwrite = FALSE,
  checksum = NULL,
  checksum_type = "sha256",
  resume = TRUE,
  block = TRUE,
  .tmp_id = NULL
)
```

Arguments:

url A string specifying the URL to download from.

filename A string specifying the filename for the downloaded file. If NULL, uses filename from URL. Default: NULL.

subdir A string specifying the subdirectory within **dest** to save file. Default: **NULL** (save directly in **dest**).

progress A logical value specifying whether to show progress bar. Default: **TRUE**.

overwrite A logical value specifying whether to overwrite existing file. Default: **FALSE**.

checksum A string specifying the expected checksum for verification. If provided, enables incremental checksum calculation. Default: **NULL**.

checksum_type A string specifying the checksum type ("sha256" or "md5"). Default: "sha256".

resume A logical value specifying whether to resume interrupted downloads. Default: **TRUE**.

block A logical value specifying whether to block until download completes. If **FALSE**, downloads asynchronously in background. Default: **TRUE**.

.tmp_id Internal temporary file ID used by persistent download tasks. Default: **NULL**.

Returns: If **block** = **TRUE**, returns the path to the downloaded file. If **block** = **FALSE**, returns a task ID for tracking the download.

Examples:

```
\dontrun{
# Blocking download
path <- dl$download(url = "https://example.com/data.nc")

# Async download
task_id <- dl$download(
  url = "https://example.com/data.nc",
  block = FALSE
)
dl$wait_for_tasks(task_id)

# Multiple files (async batch)
urls <- c("https://example.com/file1.nc", "https://example.com/file2.nc")
task_ids <- sapply(urls, function(url) {
  dl$download(url = url, block = FALSE)
})
results <- dl$wait_for_tasks(task_ids)
}
```

Method `enqueue()`: Add a download plan to the persistent manifest.

Usage:

```
Downloader$enqueue(plan, session_label = NULL)
```

Arguments:

plan A data frame with at least **logical_file_id**, **filename**, and **url** columns.

session_label Optional label for this download session.

Returns: The created session ID.

Method `preflight()`: Check local resource requirements before downloading.

Usage:

```
Downloader$preflight(
  plan = NULL,
  session_id = NULL,
  task_id = NULL,
  overwrite = FALSE
)
```

Arguments:

plan Optional download plan. If supplied, preflight is calculated without writing to the persistent manifest.

session_id Optional persistent session ID.

task_id Optional persistent task ID vector.

overwrite Whether existing final files would be overwritten. Default: FALSE.

Returns: A one-row data frame with byte and disk-space summary.

Method `run()`: Run queued persistent download tasks.

Usage:

```
Downloader$run(
  session_id = NULL,
  task_id = NULL,
  block = TRUE,
  progress = TRUE,
  overwrite = FALSE,
  resume = TRUE
)
```

Arguments:

session_id Optional session ID.

task_id Optional task ID vector.

block Whether to block until completion. If FALSE, creates a detached background job via `$start()`.

progress Whether to show per-file progress.

overwrite Whether to overwrite existing final files.

resume Whether to resume .part files.

Returns: If *block* = TRUE, a data frame of selected task records after the run. If *block* = FALSE, a one-row background job record.

Method `start()`: Start a persistent download session in the background.

Usage:

```
Downloader$start(
  session_id = NULL,
  task_id = NULL,
  overwrite = FALSE,
  resume = TRUE,
  mode = c("process", "daemon"),
  store_path = NULL
)
```

Arguments:

session_id Optional session ID.
task_id Optional task ID vector.
overwrite Whether to overwrite existing final files.
resume Whether to resume .part files.
mode Background execution mode. "process" starts a detached Rscript; "daemon" submits the job to a running downloader daemon.
store_path Optional [EsgStore](#) path to sync after completion.

Returns: A one-row data frame describing the background job.

Method `jobs()`: List downloader background jobs.

Usage:

`Downloader$jobs(status = NULL)`

Arguments:

status Optional job status filter.

Method `job_status()`: Return downloader background job status.

Usage:

`Downloader$job_status(job_id = NULL)`

Arguments:

job_id Optional job ID filter.

Method `job_logs()`: Return downloader background job log lines.

Usage:

`Downloader$job_logs(job_id, tail = 100L)`

Arguments:

job_id Job ID.

tail Number of trailing lines to return.

Method `stop_job()`: Request cancellation of a background downloader job.

Usage:

`Downloader$stop_job(job_id, force = FALSE)`

Arguments:

job_id Job ID.

force Whether to kill the recorded process immediately.

Method `daemon_start()`: Start a persistent downloader daemon.

Usage:

`Downloader$daemon_start(port = NULL, heartbeat_interval = 5)`

Arguments:

port Optional localhost TCP port. If NULL, a random high port is chosen.

heartbeat_interval Seconds between daemon heartbeat checks.

Returns: A one-row data frame describing the daemon.

Method `daemon_status()`: Return downloader daemon status records.

Usage:

```
Downloader$daemon_status()
```

Method `daemon_stop()`: Request the running downloader daemon to stop.

Usage:

```
Downloader$daemon_stop(force = FALSE)
```

Arguments:

`force` Whether to kill the daemon process immediately.

Method `sessions()`: List persistent download sessions.

Usage:

```
Downloader$sessions()
```

Method `tasks()`: List persistent download tasks.

Usage:

```
Downloader$tasks(session_id = NULL, job_id = NULL, status = NULL)
```

Arguments:

`session_id` Optional session ID.

`job_id` Optional background job ID.

`status` Optional task status filter.

Method `status()`: Return persistent download task status.

Usage:

```
Downloader$status(session_id = NULL, job_id = NULL, task_id = NULL)
```

Arguments:

`session_id` Optional session ID.

`job_id` Optional background job ID.

`task_id` Optional task ID vector.

Returns: A data frame of matching task records.

Method `events()`: Return persistent downloader event logs.

Usage:

```
Downloader$events(session_id = NULL, job_id = NULL, task_id = NULL)
```

Arguments:

`session_id` Optional session ID.

`job_id` Optional background job ID.

`task_id` Optional task ID vector.

Returns: A data frame of event records.

Method `on()`: Register an in-session downloader event callback.

Usage:

Downloader\$on(event, fun)

Arguments:

event Event name.

fun Callback function called with (event, downloader).

Returns: A callback token for \$off().

Method off(): Remove a downloader event callback.

Usage:

Downloader\$off(token)

Arguments:

token Callback token returned by \$on().

Returns: TRUE when a callback was removed.

Method data_nodes(): Return historical data node download performance.

Usage:

Downloader\$data_nodes(service = NULL)

Arguments:

service Optional ESGF service filter.

Returns: A data frame of data node performance records.

Method reset_data_nodes(): Reset historical data-node health records.

Usage:

Downloader\$reset_data_nodes(data_node = NULL, service = NULL)

Arguments:

data_node Optional data-node host to reset.

service Optional ESGF service filter.

Returns: The remaining data-node records.

Method record_probes(): Record URL probe outcomes from a download plan into node history.

Usage:

Downloader\$record_probes(plan, probed = TRUE)

Arguments:

plan A download plan returned by \$download_plan().

probed Whether probe = TRUE was used to create the plan.

Returns: The current data-node records.

Method retry(): Requeue failed or cancelled persistent tasks.

Usage:

```
Downloader$retry(
  session_id = NULL,
  task_id = NULL,
  status = c("error", "cancelled")
)
```

Arguments:

`session_id` Optional session ID.

`task_id` Optional task ID vector.

`status` Task statuses to requeue. Default: `c("error", "cancelled")`.

Returns: A data frame of requeued task records.

Method `cancel()`: Cancel queued or in-progress persistent download tasks.

Usage:

```
Downloader$cancel(
  session_id = NULL,
  task_id = NULL,
  status = c("queued", "downloading")
)
```

Arguments:

`session_id` Optional session ID.

`task_id` Optional task ID vector.

`status` Task statuses to cancel. Default: `c("queued", "downloading")`.

Returns: A data frame of cancelled task records.

Method `resume()`: Resume queued or interrupted persistent tasks.

Usage:

```
Downloader$resume(session_id = NULL, task_id = NULL, ...)
```

Arguments:

`session_id` Optional session ID.

`task_id` Optional task ID vector.

`...` Additional arguments passed to `$run()`.

Returns: A data frame of selected task records after the run.

Method `verify()`: Verify checksums for completed persistent tasks.

Usage:

```
Downloader$verify(session_id = NULL, task_id = NULL)
```

Arguments:

`session_id` Optional session ID.

`task_id` Optional task ID vector.

Returns: A data frame of completed task records with a `checksum_ok` column.

Method `cleanup_tmp()`: Clean up temporary files (.part and .done files)

Usage:

```
Downloader$cleanup_tmp(all = FALSE)
```

Arguments:

`all` If TRUE, removes all temporary files. If FALSE, only removes orphaned files (no corresponding final file). Default: FALSE.

Returns: Number of files removed.

Examples:

```
\dontrun{
n_removed <- downloader$cleanup_tmp()
n_removed <- downloader$cleanup_tmp(all = TRUE)
}
```

Method `get_tasks()`: Get all async download tasks

Usage:

```
Downloader$get_tasks()
```

Returns: A list of DownloadTask objects.

Examples:

```
\dontrun{
tasks <- downloader$get_tasks()
}
```

Method `get_task_status()`: Get status of an async download task

Usage:

```
Downloader$get_task_status(task_id)
```

Arguments:

`task_id` Task ID returned by `download(block = FALSE)`.

Returns: A list with task information including status, progress, etc.

Examples:

```
\dontrun{
task_id <- dl$download(url, block = FALSE)
status <- dl$get_task_status(task_id)
}
```

Method `wait_for_tasks()`: Wait for all async download tasks to complete

Usage:

```
Downloader$wait_for_tasks(task_ids = NULL, progress = TRUE)
```

Arguments:

`task_ids` Optional vector of task IDs to wait for. If NULL, waits for all tasks. Default: NULL.

`progress` Whether to show progress. Default: TRUE.

Returns: A list of completed task statuses.

Examples:

```
\dontrun{
task1 <- dl$download(url1, block = FALSE)
task2 <- dl$download(url2, block = FALSE)
results <- dl$wait_for_tasks()
}
```

Method `cancel_task()`: Cancel an async download task

Usage:

```
Downloader$cancel_task(task_id)
```

Arguments:

`task_id` Task ID returned by `download(block = FALSE)`.

Returns: Logical TRUE if cancellation was successful, FALSE otherwise.

Examples:

```
\dontrun{
task_id <- dl$download(url, block = FALSE)
# Cancel if needed
dl$cancel_task(task_id)
}
```

Method `list_incomplete()`: List incomplete downloads

Usage:

```
Downloader$list_incomplete()
```

Returns: A data.frame with information about incomplete downloads.

Examples:

```
\dontrun{
incomplete <- downloader$list_incomplete()
}
```

Method `verify_checksum()`: Verify file checksum

Usage:

```
Downloader$verify_checksum(file, expected, type = "sha256")
```

Arguments:

`file` Path to file to verify.

`expected` Expected checksum value.

`type` Checksum type ("md5" or "sha256"). Default: "sha256".

Returns: TRUE if checksum matches, FALSE otherwise.

Examples:

```
\dontrun{
valid <- downloader$verify_checksum("data.nc", "abc123", "sha256")
}
```

Method `print()`: Print downloader summary

Usage:

```
Downloader$print()
```

Returns: The Downloader object itself, invisibly.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Hongyuan Jia

Examples

```
## -----
## Method `Downloader$new`
## -----

## Not run:
dl <- Downloader$new()
dl <- Downloader$new(dest = "~/data")
dl <- Downloader$new(
  dest = "~/data",
  temp = "~/data/.tmp",
  n_workers = 8
)

## End(Not run)

## -----
## Method `Downloader$download`
## -----

## Not run:
# Blocking download
path <- dl$download(url = "https://example.com/data.nc")

# Async download
task_id <- dl$download(
  url = "https://example.com/data.nc",
  block = FALSE
)
dl$wait_for_tasks(task_id)

# Multiple files (async batch)
urls <- c("https://example.com/file1.nc", "https://example.com/file2.nc")
task_ids <- sapply(urls, function(url) {
  dl$download(url = url, block = FALSE)
})
results <- dl$wait_for_tasks(task_ids)
```

```

## End(Not run)

## -----
## Method `Downloader$cleanup_tmp`
## -----

## Not run:
n_removed <- downloader$cleanup_tmp()
n_removed <- downloader$cleanup_tmp(all = TRUE)

## End(Not run)

## -----
## Method `Downloader$get_tasks`
## -----

## Not run:
tasks <- downloader$get_tasks()

## End(Not run)

## -----
## Method `Downloader$get_task_status`
## -----

## Not run:
task_id <- dl$download(url, block = FALSE)
status <- dl$get_task_status(task_id)

## End(Not run)

## -----
## Method `Downloader$wait_for_tasks`
## -----

## Not run:
task1 <- dl$download(url1, block = FALSE)
task2 <- dl$download(url2, block = FALSE)
results <- dl$wait_for_tasks()

## End(Not run)

## -----
## Method `Downloader$cancel_task`
## -----

## Not run:
task_id <- dl$download(url, block = FALSE)
# Cancel if needed
dl$cancel_task(task_id)

## End(Not run)

```

```
## -----
## Method `Downloader$list_incomplete`
## -----

## Not run:
incomplete <- downloader$list_incomplete()

## End(Not run)

## -----
## Method `Downloader$verify_checksum`
## -----

## Not run:
valid <- downloader$verify_checksum("data.nc", "abc123", "sha256")

## End(Not run)
```

epw_morph_backend	<i>Get an EPW morphing backend</i>
-------------------	------------------------------------

Description

Get an EPW morphing backend

Usage

```
epw_morph_backend(name = "original_morphing")
```

Arguments

name	Backend name.
------	---------------

Value

An [EpwMorphBackend](#) object.

epw_morph_backends	<i>EPW morphing backends</i>
--------------------	------------------------------

Description

EPW morphing backends

Usage

```
epw_morph_backends()
```

Value

A character vector of registered backend names.

epw_morph_periods	<i>EPW morphing periods</i>
-------------------	-----------------------------

Description

EPW morphing periods

Usage

```
epw_morph_periods(...)
```

Arguments

... Named integer year vectors.

Value

A data.table with columns period and year.

epw_morph_register_backend	<i>Register an EPW morphing backend</i>
----------------------------	---

Description

Register an EPW morphing backend

Usage

```
epw_morph_register_backend(name, backend, overwrite = FALSE)
```

Arguments

name	Backend name.
backend	An EpwMorphBackend object.
overwrite	Whether to replace an existing backend.

Value

The backend object, invisibly.

epw_morph_result	Create an EPW morphing backend result
------------------	---------------------------------------

Description

Backend runner functions return epw_morph_result objects. Use epw_morph_result() in custom backends after producing complete hourly EPW weather data.

Usage

```
epw_morph_result(  
  context,  
  epw = context$epw,  
  data,  
  parts = list(),  
  diagnostics = morpher__empty_diagnostics(),  
  factors = NULL  
)
```

Arguments

context	Canonical EPW morphing context supplied to the backend runner.
epw	EPW object associated with the result.
data	Complete hourly EPW weather data ready for Parquet output or EPW writing.
parts	Optional named list of intermediate backend result tables.
diagnostics	Optional backend diagnostic rows.
factors	Optional backend factor rows.

Value

An epw_morph_result object.

epw_morpher	Create an EPW morpher
-------------	-----------------------

Description

Create an EPW morpher

Usage

```
epw_morpher(  
  store,  
  epw,  
  site_id = NULL,  
  transform = monthly_transform("original_morphing"),  
  label = NULL  
)
```

Arguments

store	An EsgStore object.
epw	Baseline EPW path, internal EpwFile, or an external object inheriting from "Epw".
site_id	Optional site identifier.
transform	A reusable weather transformation created by monthly_transform() , daily_transform() , or hourly_transform() .
label	Optional source label.

Value

An [EpwMorpher](#) object.

EpwMorphBackend	<i>EPW morphing backend</i>
-----------------	-----------------------------

Description

EpwMorphBackend defines a low-level execution backend used by registered weather transformations and executed by [EpwMorpher](#).

Public fields

- name Backend name.
- label Human-readable backend label.
- requires_reference Whether external reference climate data are mandatory for this backend.
- accepts_reference Whether the backend can consume external reference climate data. A backend may accept a reference without requiring one.

Methods

Public methods:

- [EpwMorphBackend\\$new\(\)](#)
- [EpwMorphBackend\\$methods\(\)](#)
- [EpwMorphBackend\\$method_choices\(\)](#)
- [EpwMorphBackend\\$rules\(\)](#)
- [EpwMorphBackend\\$component_pipeline\(\)](#)
- [EpwMorphBackend\\$required_variables\(\)](#)
- [EpwMorphBackend\\$validate_methods\(\)](#)
- [EpwMorphBackend\\$rules_with_methods\(\)](#)
- [EpwMorphBackend\\$run\(\)](#)
- [EpwMorphBackend\\$clone\(\)](#)

Method `new()`: Create an EPW morphing backend.

Usage:

```
EpwMorphBackend$new(
  name,
  label = NULL,
  methods = NULL,
  method_choices = NULL,
  rules,
  requires_reference = FALSE,
  accepts_reference = requires_reference,
  pipeline = NULL,
  runner = NULL
)
```

Arguments:

`name` Backend name.

`label` Human-readable backend label.

`methods` Named default method vector.

`method_choices` Allowed method values.

`rules` Backend rule table.

`requires_reference` Whether external reference climate data are mandatory. This can be TRUE only when `accepts_reference` is also TRUE.

`accepts_reference` Whether external reference climate data can be consumed. TRUE with `requires_reference = FALSE` defines an optional-reference backend such as original morphing.

`pipeline` Optional internal component pipeline specification.

`runner` Optional function taking (context, backend) and returning an `epw_morph_result`. Exactly one of `pipeline` and `runner` must be supplied.

Method `methods()`: Return default backend methods.

Usage:

```
EpwMorphBackend$methods()
```

Method `method_choices()`: Return allowed backend method values.

Usage:

`EpwMorphBackend$method_choices()`

Method `rules()`: Return backend rules.

Usage:

`EpwMorphBackend$rules()`

Method `component_pipeline()`: Return the optional component pipeline used by this backend.

Usage:

`EpwMorphBackend$component_pipeline()`

Method `required_variables()`: Return required CMIP variable IDs.

Usage:

`EpwMorphBackend$required_variables()`

Method `validate_methods()`: Validate and complete method overrides.

Usage:

`EpwMorphBackend$validate_methods(methods = NULL)`

Arguments:

`methods` Optional named method override vector.

Method `rules_with_methods()`: Return backend rules with methods applied.

Usage:

`EpwMorphBackend$rules_with_methods(methods = NULL)`

Arguments:

`methods` Optional named method override vector.

Method `run()`: Run this backend on a canonical EPW morphing context.

Usage:

`EpwMorphBackend$run(context)`

Arguments:

`context` Canonical EPW morphing context.

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

Usage:

`EpwMorphBackend$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

EpwMorpher

*Store-native EPW morpher***Description**

EpwMorpher consumes completed [EsgStore](#) extraction outputs and creates future EPW files through a store-backed morphing workflow.

Methods**Public methods:**

- [EpwMorpher\\$new\(\)](#)
- [EpwMorpher\\$required_variables\(\)](#)
- [EpwMorpher\\$preflight\(\)](#)
- [EpwMorpher\\$summarise_climate\(\)](#)
- [EpwMorpher\\$summarise_baseline\(\)](#)
- [EpwMorpher\\$plan\(\)](#)
- [EpwMorpher\\$preview_plan\(\)](#)
- [EpwMorpher\\$diagnose\(\)](#)
- [EpwMorpher\\$check\(\)](#)
- [EpwMorpher\\$run\(\)](#)
- [EpwMorpher\\$write_epw\(\)](#)
- [EpwMorpher\\$workflow\(\)](#)
- [EpwMorpher\\$status\(\)](#)
- [EpwMorpher\\$process_data\(\)](#)
- [EpwMorpher\\$outputs\(\)](#)
- [EpwMorpher\\$clone\(\)](#)

`EpwMorpher$new()`: Create an EPW morpher.

Usage:

```
EpwMorpher$new(
  store,
  epw,
  site_id = NULL,
  transform = monthly_transform("original_morphing"),
  label = NULL
)
```

Arguments:

`store` An [EsgStore](#) object.

`epw` Baseline EPW path, internal `EpwFile`, or an external object inheriting from "Epw".

`site_id` Optional site identifier.

`transform` A reusable weather transformation created by a scale-specific transform constructor.

label Optional source label.

EpwMorpher\$required_variables(): Return transform-required CMIP variable IDs.

Usage:

```
EpwMorpher$required_variables()
```

EpwMorpher\$preflight(): Preflight EPW morphing inputs without writing store state.

Usage:

```
EpwMorpher$preflight(
  plan_id = NULL,
  periods = NULL,
  reference_plan_id = NULL,
  reference_periods = NULL,
  summary_id = NULL,
  reference_summary_id = NULL,
  observed_plan_id = NULL,
  observed_periods = NULL,
  observed_summary_id = NULL,
  baseline_id = NULL,
  by = c("source_id", "experiment_id", "variant_label", "period"),
  strict = TRUE
)
```

Arguments:

plan_id Optional extraction plan IDs.

periods Optional period table from [epw_morph_periods\(\)](#).

reference_plan_id Optional reference extraction plan IDs for change-factor backends.

reference_periods Optional reference period table from [epw_morph_periods\(\)](#).

summary_id Optional climate summary ID.

reference_summary_id Optional reference climate summary ID for change-factor backends.

observed_plan_id Optional observed-weather extraction plan IDs.

observed_periods Optional observed-weather period table.

observed_summary_id Optional observed-weather summary ID.

baseline_id Optional baseline summary ID.

by Climate grouping columns.

strict Whether required-data issues are errors.

EpwMorpher\$summarise_climate(): Summarise extracted climate data by period and month.

Usage:

```
EpwMorpher$summarise_climate(
  plan_id,
  periods,
  strict = TRUE,
  overwrite = FALSE,
  input_role = "model_future"
)
```

Arguments:

plan_id Extraction plan IDs.
 periods Period table from `epw_morph_periods()`.
 strict Whether incomplete extraction coverage is an error.
 overwrite Whether to replace existing rows for this summary.
 input_role Source role used to validate raw variables and frequencies, one of `model_future`, `model_historical`, or `observed_reference`. Workflow calls select this automatically.

`EpwMorpher$summarise_baseline()`: Summarise baseline EPW weather by month.

Usage:

```
EpwMorpher$summarise_baseline(overwrite = FALSE)
```

Arguments:

overwrite Whether to replace existing rows.

`EpwMorpher$plan()`: Create a morphing plan and monthly factors.

Usage:

```
EpwMorpher$plan(
  summary_id,
  reference_summary_id = NULL,
  observed_summary_id = NULL,
  baseline_id = NULL,
  by = c("source_id", "experiment_id", "variant_label", "period"),
  strict = TRUE,
  overwrite = FALSE
)
```

Arguments:

summary_id Climate summary ID.
 reference_summary_id Optional reference climate summary ID for change-factor backends.
 observed_summary_id Optional observed-weather summary ID.
 baseline_id Baseline summary ID. If NULL, baseline summary is created.
 by Climate grouping columns.
 strict Whether missing required variables are blocking errors.
 overwrite Whether to replace an existing plan.

`EpwMorpher$preview_plan()`: Preview a morphing plan and monthly factors without writing store state.

Usage:

```
EpwMorpher$preview_plan(
  summary_id,
  reference_summary_id = NULL,
  observed_summary_id = NULL,
  baseline_id = NULL,
  by = c("source_id", "experiment_id", "variant_label", "period"),
  strict = TRUE
)
```

Arguments:

summary_id Climate summary ID.
 reference_summary_id Optional reference climate summary ID for change-factor backends.
 observed_summary_id Optional observed-weather summary ID.
 baseline_id Baseline summary ID. If NULL, baseline summary is created.
 by Climate grouping columns.
 strict Whether missing required variables are blocking errors.

EpwMorpher\$diagnose(): Diagnose a morphing plan.

Usage:

```
EpwMorpher$diagnose(morph_id)
```

Arguments:

morph_id Morphing plan ID.

EpwMorpher\$check(): Abort if a morphing plan has blocking diagnostics.

Usage:

```
EpwMorpher$check(morph_id)
```

Arguments:

morph_id Morphing plan ID.

EpwMorpher\$run(): Execute a morphing plan and write one hourly Parquet file per result member and weather year.

Usage:

```
EpwMorpher$run(morph_id, overwrite = FALSE, resume = TRUE, reporter = NULL)
```

Arguments:

morph_id Morphing plan ID.
 overwrite Whether to overwrite existing result files.
 resume Whether to reuse complete existing results.
 reporter Optional workflow reporter used by task-level runs.

EpwMorpher\$write_epw(): Write one future EPW file for every persisted result member.

Usage:

```
EpwMorpher$write_epw(  
  morph_id,  
  dir,  
  separate = TRUE,  
  overwrite = FALSE,  
  resume = TRUE,  
  reporter = NULL  
)
```

Arguments:

morph_id Morphing plan ID.

`dir` Output directory inside the store root. Relative paths are resolved under the store root. If `NULL`, the workflow stops after writing morph result Parquet files and does not write EPW outputs.

`separate` Whether to create case subdirectories.

`overwrite` Whether to overwrite existing EPW files.

`resume` Whether to reuse complete existing EPW outputs.

`reporter` Optional workflow reporter used by task-level runs.

`EpwMorpher$workflow()`: Run the store-native EPW morphing workflow.

Usage:

```
EpwMorpher$workflow(
  plan_id,
  periods,
  reference_plan_id = NULL,
  reference_periods = NULL,
  observed_plan_id = NULL,
  observed_periods = NULL,
  by = c("source_id", "experiment_id", "variant_label", "period"),
  strict = TRUE,
  dir = "outputs/future-epw",
  separate = TRUE,
  overwrite = FALSE,
  resume = TRUE,
  reporter = NULL
)
```

Arguments:

`plan_id` Extraction plan IDs.

`periods` Period table from `epw_morph_periods()`.

`reference_plan_id` Optional reference extraction plan IDs for change-factor backends.

`reference_periods` Optional reference period table from `epw_morph_periods()`.

`observed_plan_id` Optional observed-weather extraction plan IDs.

`observed_periods` Optional observed-weather period table.

`by` Climate grouping columns.

`strict` Whether blocking diagnostics should abort the workflow.

`dir` Output directory. Relative paths are resolved under the store root.

`separate` Whether to create case subdirectories.

`overwrite` Whether to overwrite existing plan, result, and EPW outputs.

`resume` Whether to reuse complete existing result and EPW outputs.

`reporter` Optional workflow reporter used by task-level runs.

`EpwMorpher$status()`: Return morphing plan status rows.

Usage:

```
EpwMorpher$status(morph_id = NULL)
```

Arguments:

morph_id Optional morphing plan IDs.

`EpwMorpher$process_data()`: Read saved execution products for one morphing plan. The result is a list named by case ID. Each entry contains the saved plan and case, runtime options, backend factors, method-specific parts, runtime diagnostics and provenance, plus the associated result manifest rows in results. These are execution products, not preview factors.

Factor tables may summarize hourly values. Use method-specific parts for hourly application parameters where supplied by the backend. Empty or NULL products mean that the backend did not supply them. This interface does not reconstruct unrecorded pipeline stages. Multi-year members share one case payload. Reading checks its SHA256. Older runs without saved process data require an explicit rerun with `overwrite = TRUE`. No climate data are fetched by this method.

Usage:

```
EpwMorpher$process_data(morph_id, case_id = NULL)
```

Arguments:

morph_id Morphing plan ID.

case_id Optional case ID to read.

`EpwMorpher$outputs()`: Return future EPW output rows.

Usage:

```
EpwMorpher$outputs(morph_id = NULL)
```

Arguments:

morph_id Optional morphing plan IDs.

`EpwMorpher$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

deep Whether to make a deep clone.

Author(s)

Hongyuan Jia

epwshiftr_cli

Run the epwshiftr command line interface

Description

`epwshiftr_cli()` is the package-level entry point used by the optional `epwshiftr` launcher. It exposes ESGF store operations, weather-method discovery, and persistent single-run or batch future-weather workflows, and returns status metadata when `exit = FALSE`, which makes it testable from R.

Usage

```
epwshiftr_cli(args = commandArgs(trailingOnly = TRUE), exit = FALSE)
```

Arguments

<code>args</code>	Command line arguments. Defaults to <code>commandArgs(trailingOnly = TRUE)</code> .
<code>exit</code>	Whether to terminate the current R process with the command status. Default: <code>FALSE</code> .

Value

Invisibly, a list with status, result, and error.

<code>esg_result</code>	<i>Create empty query result object</i>
-------------------------	---

Description

`esg_result()` creates an empty query result object of input type, so that you can load the saved JSON file via `EsgResult$load()`.

Usage

```
esg_result(type = c("dataset", "file", "aggregation"))
```

Arguments

<code>type</code>	A string indicating what type of ESGF query result should be created. Should be one of "dataset", "file" or "aggregation".
-------------------	--

Value

An empty [EsgResult](#) object of given type.

EsgDataset

*Remote NetCDF Dataset Access via OPeNDAP***Description**

EsgDataset provides a unified interface for accessing NetCDF data remotely via OPeNDAP protocol. It wraps RNetCDF functions and provides convenient methods for subsetting, slicing, and reading data without downloading entire files.

The class supports three levels of interfaces:

- **Basic layer:** Direct wrappers around RNetCDF functions
- **Middle layer:** Convenient methods for subsetting by time/space
- **High layer:** Data manipulation and format conversion

It also supports aggregating multiple files into a single logical dataset, automatically handling time dimension concatenation.

Active bindings

`url` The OPeNDAP URL(s)

`is_open` Whether the connection is open

`is_aggregated` Whether the dataset contains multiple files

`file_count` Number of files in the dataset

`time_filter` A result-level time filter recorded by `EsgResultFile$filter_time()` or `EsgResultAggregation$filter_time()` or NULL.

Methods**Public methods:**

- `EsgDataset$new()`
- `EsgDataset$open()`
- `EsgDataset$close()`
- `EsgDataset$slice()`
- `EsgDataset$reachable()`
- `EsgDataset$file_inq()`
- `EsgDataset$var_inq()`
- `EsgDataset$dim_inq()`
- `EsgDataset$att_get()`
- `EsgDataset$var_get()`
- `EsgDataset$get_variables()`
- `EsgDataset$get_dimensions()`
- `EsgDataset$get_time_axis()`
- `EsgDataset$get_spatial_grid()`

- `EsgDataset$read_array()`
- `EsgDataset$read_data_table()`
- `EsgDataset$read_region()`
- `EsgDataset$selection()`
- `EsgDataset$print()`
- `EsgDataset$clone()`

Method `new()`: Create a new `EsgDataset` object

Usage:

```
EsgDataset$new(urls)
```

Arguments:

`urls` A character vector of OPeNDAP URLs. Can be a single URL or multiple URLs for a multi-file dataset.

Returns: An `EsgDataset` object.

Examples:

```
\dontrun{
# Single file
ds <- EsgDataset$new("https://example.com/data.nc")

# Multiple files
ds <- EsgDataset$new(c("url1.nc", "url2.nc"))
}
```

Method `open()`: Open OPeNDAP connection(s)

Usage:

```
EsgDataset$open(
  async = FALSE,
  timeout = NULL,
  progress = getOption("epwshiftr.progress", interactive())
)
```

Arguments:

`async` If TRUE, first validates opening in a one-shot worker, then re-opens caller-owned handles before returning so the dataset remains opened after `open()` returns. The caller still receives the final `EsgDataset` object itself rather than a `Mirai/Future`-like handle. Default: FALSE.

`timeout` Optional positive number of seconds for the `async` worker pre-open phase. Only supported when `async = TRUE`. It does not limit the final caller-owned reopen that makes the returned dataset stay opened.

`progress` Whether to show a progress bar while opening NetCDF/OPeNDAP handles. By default the package option `epwshiftr.progress` is used, falling back to `interactive()`.

Returns: The `EsgDataset` object itself, invisibly.

Examples:

```
\dontrun{
ds$open()
# Returns the opened dataset directly; no Mirai/Future to collect.
ds$open(async = TRUE, timeout = 10)
}
```

Method `close()`: Close OPeNDAP connection(s)

Usage:

```
EsgDataset$close()
```

Returns: The EsgDataset object itself, invisibly.

Examples:

```
\dontrun{
ds$close()
}
```

Method `slice()`: Select files from this dataset by file position.

`$slice()` creates a new EsgDataset with a subset of the current dataset URLs. This is a file/URL-level operation; NetCDF variable, dimension, time, and spatial slicing is still performed by `$var_get()`, `$read_array()`, `$read_data_table()`, and `$read_region()`.

Usage:

```
EsgDataset$slice(i, reopen = FALSE)
```

Arguments:

`i` A positive or negative integer vector, or a logical vector with one value per file.

`reopen` Whether to open the returned dataset when the current dataset is already open. If FALSE (default), slicing an open dataset raises an error because RNetCDF handles cannot be safely shared between dataset objects.

Returns: A new EsgDataset object.

Method `reachable()`: Probe whether this dataset's current files or URLs are reachable.

`$reachable()` checks the actual URLs or local paths stored in the dataset. It does not reuse reachability checks from an EsgResult; opened datasets, fallback downloads, and manually created datasets are always evaluated from their current `url` values.

Usage:

```
EsgDataset$reachable(level = c("data_node", "url"), probe = NULL)
```

Arguments:

`level` Probe level. "data_node" probes the root URL of each remote data node; "url" probes the actual dataset URL. Default: "data_node".

`probe` Optional named list of probe settings. Supported fields are `timeout`, `concurrency`, `network_policy`, `cache_seconds`, and `cache_failures_seconds`.

Returns: A [data.table](#) with columns `file_index`, `source_index`, `data_node`, `service`, `url`, `reachable`, `latency_ms`, `error`, `probe_level`, `probe_url`, and `probe_cached`.

Method `file_inq()`: Get file information

Usage:

```
EsgDataset$file_inq(index = 1L)
```

Arguments:

`index` File index for multi-file datasets. Default: 1L.

Returns: A list with file information.

Examples:

```
\dontrun{
info <- ds$file_inq()
}
```

Method `var_inq()`: Get variable information

Usage:

```
EsgDataset$var_inq(var, index = 1L)
```

Arguments:

`var` Variable name or ID.

`index` File index for multi-file datasets. Default: 1L.

Returns: A list with variable information.

Examples:

```
\dontrun{
var_info <- ds$var_inq("tas")
}
```

Method `dim_inq()`: Get dimension information

Usage:

```
EsgDataset$dim_inq(dim, index = 1L)
```

Arguments:

`dim` Dimension name or ID.

`index` File index for multi-file datasets. Default: 1L.

Returns: A list with dimension information.

Examples:

```
\dontrun{
dim_info <- ds$dim_inq("time")
}
```

Method `att_get()`: Get attribute value

Usage:

```
EsgDataset$att_get(var, att, index = 1L)
```

Arguments:

`var` Variable name or ID, or "NC_GLOBAL" for global attributes.

`att` Attribute name.

`index` File index for multi-file datasets. Default: 1L.

Returns: The attribute value.

Examples:

```
\dontrun{
units <- ds$att_get("tas", "units")
}
```

Method `var_get()`: Read variable data

Usage:

```
EsgDataset$var_get(
  var,
  start = NULL,
  count = NULL,
  index = 1L,
  collapse = FALSE,
  async = FALSE,
  timeout = NULL
)
```

Arguments:

var Variable name or ID.

start Starting indices (1-based). If NULL, starts from beginning.

count Number of values to read. If NULL, reads all.

index File index for multi-file datasets. Default: 1L.

collapse Whether to collapse result. Default: FALSE.

async If TRUE, perform the variable read in a one-shot worker and return the final array directly once complete. No Mirai/Future object is exposed. Default: FALSE.

timeout Optional positive number of seconds for the async worker phase. Only supported when **async** = TRUE.

Returns: An array with variable data.

Examples:

```
\dontrun{
data <- ds$var_get("tas")
data_subset <- ds$var_get("tas", start = c(1, 1, 1), count = c(10, 10, 1))
# Returns the final array directly; no Mirai/Future handling required.
data_async <- ds$var_get("tas", async = TRUE, timeout = 10)
}
```

Method `get_variables()`: List all variables in the dataset

Usage:

```
EsgDataset$get_variables(index = 1L)
```

Arguments:

index File index for multi-file datasets. Default: 1L.

Returns: A character vector of variable names.

Examples:

```
\dontrun{
vars <- ds$get_variables()
}
```

Method `get_dimensions()`: List all dimensions in the dataset

Usage:

```
EsgDataset$get_dimensions(index = 1L)
```

Arguments:

index File index for multi-file datasets. Default: 1L.

Returns: A character vector of dimension names.

Examples:

```
\dontrun{
dims <- ds$get_dimensions()
}
```

Method `get_time_axis()`: Get time axis information

Usage:

```
EsgDataset$get_time_axis(index = 1L)
```

Arguments:

index File index for multi-file datasets. Default: 1L.

Returns: A list containing time values, units, calendar, canonical CF coordinates, optional CF interval bounds, and axis length.

Examples:

```
\dontrun{
time_info <- ds$get_time_axis()
}
```

Method `get_spatial_grid()`: Get spatial grid information (latitude and longitude)

Usage:

```
EsgDataset$get_spatial_grid(index = 1L)
```

Arguments:

index File index for multi-file datasets. Default: 1L.

Returns: A list containing latitude and longitude values.

Examples:

```
\dontrun{
grid <- ds$get_spatial_grid()
}
```

Method `read_array()`: Read variable data as a list of arrays (one per file)

Usage:

```
EsgDataset$read_array(
  variable,
  start = NULL,
  count = NULL,
  collapse = FALSE,
  async = FALSE,
  timeout = NULL
)
```

Arguments:

variable Variable name.

start Starting indices. If NULL, starts from beginning.
 count Number of values to read. If NULL, reads all.
 collapse Whether to collapse result. Default: FALSE.
 async If TRUE, read array values in a one-shot worker and return the final list directly once complete. No Mirai/Future object is exposed. Default: FALSE.
 timeout Optional positive number of seconds for the async worker phase. Only supported when async = TRUE.

Returns: A list of arrays with variable data. Each element corresponds to a file in the dataset.

Examples:

```
\dontrun{
data_list <- ds$read_array("tas")
data <- data_list[[1]]
# Returns the final list directly; no Mirai/Future handling required.
data_list_async <- ds$read_array("tas", async = TRUE, timeout = 10)
}
```

Method read_data_table(): Read variable data as a list of data.table (one per file)

Usage:

```
EsgDataset$read_data_table(
  variable,
  start = NULL,
  count = NULL,
  rbind = FALSE,
  async = FALSE,
  timeout = NULL
)
```

Arguments:

variable Variable name.
 start Starting indices. If NULL, starts from beginning.
 count Number of values to read. If NULL, reads all.
 rbind If TRUE, return a single data.table by row-binding the per-file results with data.table::rbindlist(..., idcol = "file_index"). Default: FALSE.
 async If TRUE, offload the array read phase to a one-shot worker and still return the final data.table result directly. No Mirai/Future object is exposed. Default: FALSE.
 timeout Optional positive number of seconds for the async worker phase. Only supported when async = TRUE.

Returns: If rbind = FALSE, a list of data.table (one per file). If rbind = TRUE, a single data.table with an extra file_index column.

Examples:

```
\dontrun{
dt_list <- ds$read_data_table("tas")
dt <- dt_list[[1]]
dt_all <- ds$read_data_table("tas", rbind = TRUE)
# Returns the final data.table directly; no Mirai/Future handling required.
dt_async <- ds$read_data_table("tas", async = TRUE, timeout = 10)
}
```

Method `read_region()`: Read variable values near a target coordinate and optional time range

Usage:

```
EsgDataset$read_region(
  variable,
  lon,
  lat,
  time = "auto",
  method = "nearest",
  rbind = TRUE,
  async = FALSE,
  timeout = NULL
)
```

Arguments:

`variable` Character vector of variable names.

`lon` Target longitude.

`lat` Target latitude.

`time` Time range to read. Use "auto" to reuse the time range recorded by `EsgResultFile$filter_time()` or `EsgResultAggregation$filter_time()` when available; if no recorded range exists, all times are read. Use NULL to always read the full time axis. A length-2 character, Date, or POSIXt range is parsed in UTC and used explicitly. Default: "auto".

`method` Grid extraction method. One of "nearest", "idw", "bilinear", or "mean". Default: "nearest".

`rbind` If TRUE, return one data.table. If FALSE, return a list of per-file, per-variable data.tables. Default: TRUE.

`async` If TRUE, offload each NetCDF variable read to a one-shot worker. Default: FALSE.

`timeout` Optional positive number of seconds for each async read. Only supported when `async = TRUE`.

Returns: A data.table or list of data.tables with columns including `file_index`, `variable`, `time`, optional `time_bound_start` and `time_bound_end`, canonical `cf_*` calendar coordinates including `cf_second_of_day` (seconds since native-calendar midnight), `annual_phase`, `lon`, `lat`, `method`, and `value`. The "grid_sources" attribute records contributing grid coordinates and weights.

Examples:

```
\dontrun{
dt <- ds$read_region(
  variable = c("tas", "hurs"),
  lon = 103.98,
  lat = 1.37,
  time = c("2050-01-01", "2050-12-31")
)
}
```

Method `selection()`: Return file selection provenance for this dataset.

`$selection()` maps the current dataset file positions back to the result rows that produced the dataset when that information is available. It does not record intermediate filter steps.

Usage:

```
EsgDataset$selection()
```

Returns: A list with source_count, source_num_found, and source_indices.

Method print(): Print dataset summary

Usage:

```
EsgDataset$print()
```

Returns: The EsgDataset object itself, invisibly.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Author(s)

Hongyuan Jia

Examples

```
## -----
## Method `EsgDataset$new`
## -----

## Not run:
# Single file
ds <- EsgDataset$new("https://example.com/data.nc")

# Multiple files
ds <- EsgDataset$new(c("url1.nc", "url2.nc"))

## End(Not run)

## -----
## Method `EsgDataset$open`
## -----

## Not run:
ds$open()
# Returns the opened dataset directly; no Mirai/Future to collect.
ds$open(async = TRUE, timeout = 10)

## End(Not run)

## -----
## Method `EsgDataset$close`
## -----

## Not run:
```

```

ds$close()

## End(Not run)

## -----
## Method `EsgDataset$file_inq`
## -----

## Not run:
info <- ds$file_inq()

## End(Not run)

## -----
## Method `EsgDataset$var_inq`
## -----

## Not run:
var_info <- ds$var_inq("tas")

## End(Not run)

## -----
## Method `EsgDataset$dim_inq`
## -----

## Not run:
dim_info <- ds$dim_inq("time")

## End(Not run)

## -----
## Method `EsgDataset$att_get`
## -----

## Not run:
units <- ds$att_get("tas", "units")

## End(Not run)

## -----
## Method `EsgDataset$var_get`
## -----

## Not run:
data <- ds$var_get("tas")
data_subset <- ds$var_get("tas", start = c(1, 1, 1), count = c(10, 10, 1))
# Returns the final array directly; no Mirai/Future handling required.
data_async <- ds$var_get("tas", async = TRUE, timeout = 10)

## End(Not run)

## -----

```

```

## Method `EsgDataset$get_variables`
## -----

## Not run:
vars <- ds$get_variables()

## End(Not run)

## -----
## Method `EsgDataset$get_dimensions`
## -----

## Not run:
dims <- ds$get_dimensions()

## End(Not run)

## -----
## Method `EsgDataset$get_time_axis`
## -----

## Not run:
time_info <- ds$get_time_axis()

## End(Not run)

## -----
## Method `EsgDataset$get_spatial_grid`
## -----

## Not run:
grid <- ds$get_spatial_grid()

## End(Not run)

## -----
## Method `EsgDataset$read_array`
## -----

## Not run:
data_list <- ds$read_array("tas")
data <- data_list[[1]]
# Returns the final list directly; no Mirai/Future handling required.
data_list_async <- ds$read_array("tas", async = TRUE, timeout = 10)

## End(Not run)

## -----
## Method `EsgDataset$read_data_table`
## -----

## Not run:
dt_list <- ds$read_data_table("tas")

```

```

dt <- dt_list[[1]]
dt_all <- ds$read_data_table("tas", rbind = TRUE)
# Returns the final data.table directly; no Mirai/Future handling required.
dt_async <- ds$read_data_table("tas", async = TRUE, timeout = 10)

## End(Not run)

## -----
## Method `EsgDataset$read_region`
## -----

## Not run:
dt <- ds$read_region(
  variable = c("tas", "hurs"),
  lon = 103.98,
  lat = 1.37,
  time = c("2050-01-01", "2050-12-31")
)

## End(Not run)

```

EsgDict

ESG Project Dictionary

Description

EsgDict is an [R6](#) class for project-specific ESG controlled vocabulary data. It stores vocabulary tables, optional request tables, normalized query indices, and source metadata used by local option discovery and legality checks.

`esgdict()` is a small constructor around `EsgDict$new()`.

Usage

```

esgdict(project = "CMIP6")

esgdict_set_default(dict)

esgdict_get_default(project = "CMIP6")

```

Arguments

<code>project</code>	ESG project identifier, such as "CMIP6" or "CMIP6PLUS".
<code>dict</code>	An EsgDict object used as the package-level default dictionary for its project.

Value

`esgdict()` returns a new [EsgDict](#) object. `esgdict_set_default()` returns `dict`, invisibly. `esgdict_get_default()` returns the current package-level default dictionary for `project`, or `NULL`.

Supported projects

The dictionary currently supports "CMIP6", "CMIP6PLUS", "INPUT4MIP", "OBS4REF", "CORDEX-CMIP6", "CMIP7", and "EMD". CMIP6 dictionaries include both controlled vocabularies and CMOR request-table data. Other projects use vocabulary data only until a project-specific request source is registered.

Source downloads in examples

Building a dictionary may download upstream vocabulary/request sources when the parsed dictionary cache and raw source cache are missing. Most examples load a small installed CMIP6 example dictionary and run without network access. The example that calls `$build()` is wrapped in `\dontrun{}` so package checks do not depend on GitHub or upstream CV availability.

Methods

Public methods:

- `EsgDict$new()`
- `EsgDict$project()`
- `EsgDict$profile()`
- `EsgDict$version()`
- `EsgDict$sources()`
- `EsgDict$timestamp()`
- `EsgDict$built_time()`
- `EsgDict$status()`
- `EsgDict$has_data()`
- `EsgDict$is_empty()`
- `EsgDict$build()`
- `EsgDict$get()`
- `EsgDict$capabilities()`
- `EsgDict$relation_fields()`
- `EsgDict$fields()`
- `EsgDict$indices()`
- `EsgDict$options()`
- `EsgDict$check()`
- `EsgDict$save()`
- `EsgDict$load()`
- `EsgDict$print()`

Method `new()`: Create a new ESG project dictionary.

The new dictionary is empty. Use `$build()` to fetch and parse upstream sources, or `$load()` to restore a saved dictionary JSON file.

Usage:

```
EsgDict$new(project = "CMIP6")
```

Arguments:

project ESG project identifier, such as "CMIP6" or "CMIP6PLUS".

Returns: An EsgDict object.

Examples:

```
dict <- EsgDict$new(project = "CMIP6")
dict$status()
```

Method project(): Return the normalized ESG project identifier.

Usage:

```
EsgDict$project()
```

Returns: A single string.

Examples:

```
dict <- EsgDict$new(project = "CMIP6PLUS")
dict$project()
```

Method profile(): Return the internal dictionary profile.

The profile determines how project-specific vocabulary sources are parsed and normalized.

Usage:

```
EsgDict$profile()
```

Returns: A single string.

Examples:

```
dict <- EsgDict$new(project = "CMIP6")
dict$profile()
```

Method version(): Return vocabulary and request-source versions.

Empty or partially loaded dictionaries return NULL.

Usage:

```
EsgDict$version()
```

Returns: A named list with vocab and request elements, or NULL.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$version()
```

Method sources(): Return upstream source metadata.

Source metadata records repository, tag/ref, commit, and local source directory information for the data used to build the dictionary.

Usage:

```
EsgDict$sources()
```

Returns: A named list, or NULL for an empty dictionary.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$sources()
```

Method `timestamp()`: Return source vocabulary timestamps.

Timestamps are extracted from source vocabulary metadata when available.

Usage:

```
EsgDict$timestamp()
```

Returns: A named list of timestamps, or NULL.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$timestamp()
```

Method `built_time()`: Return the time when this dictionary was built.

Usage:

```
EsgDict$built_time()
```

Returns: A POSIXct value, or NULL.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$built_time()
```

Method `status()`: Return the dictionary lifecycle status.

Status values are:

- "empty": no vocabulary/request payload is loaded.
- "partial": some required payload is missing.
- "built": the dictionary was built in this R session.
- "loaded": the dictionary was restored from disk.

Usage:

```
EsgDict$status()
```

Returns: A single string.

Examples:

```
dict <- EsgDict$new(project = "CMIP6")
dict$status()
```

Method `has_data()`: Check whether the dictionary contains usable data.

A dictionary has usable data after a complete `$build()` or `$load()`.

Usage:

```
EsgDict$has_data()
```

Returns: TRUE or FALSE.

Examples:

```
dict <- EsgDict$new(project = "CMIP6")
dict$has_data()
```

Method `is_empty()`: Check whether the dictionary is empty.

Usage:

```
EsgDict$is_empty()
```

Returns: TRUE or FALSE.

Examples:

```
dict <- EsgDict$new(project = "CMIP6")
dict$is_empty()
```

Method `build()`: Build the dictionary from upstream source data.

`$build()` resolves the configured project vocabulary source, downloads or reuses raw source files as needed, parses them into normalized tables, and builds query indices for option discovery and validation. If the dictionary already has data and `force = FALSE`, the object is returned unchanged.

Usage:

```
EsgDict$build(
  token = NULL,
  force = FALSE,
  cv_tag = NULL,
  request_tag = NULL,
  dreq_tag = NULL,
  use_cache = TRUE,
  source_dir = dict__source_dir(project = private$m_project)
)
```

Arguments:

`token` Optional GitHub token used for source resolution and downloads.

`force` If TRUE, rebuild even when the dictionary already has data and bypass the parsed dictionary cache.

`cv_tag` Optional vocabulary source tag or ref. When NULL, the project default ref or latest tagged source is used.

`request_tag` Optional request-table source tag. Used by projects that define a request source, currently CMIP6.

`dreq_tag` Deprecated alias for `request_tag`.

`use_cache` If TRUE, use the parsed dictionary cache when available. Raw source files may still be reused from `source_dir`.

`source_dir` Directory used to read and write raw source files. The default is the package store source directory for this project.

Returns: The modified EsgDict object itself.

Examples:

```
\dontrun{
  dict <- EsgDict$new(project = "CMIP6")
  dict$build()
  dict$has_data()
}
```

Method `get()`: Return raw dictionary payload data.

`$get("vocab")` returns the full vocabulary payload list. `$get("request")` and `$get("dreq")` return the request table when available. Any other value is interpreted as a vocabulary field name, such as "experiment_id" or "source_id".

Usage:

```
EsgDict$get(type)
```

Arguments:

type Data type to retrieve. Use "vocab", "request", "dreq", or a project vocabulary field name.

Returns: A copy of the requested data.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$get("experiment_id")
dict$get("request")
```

Method `capabilities()`: Return available dictionary capabilities.

Capabilities describe whether vocabulary data, request data, and relation indices are currently available.

Usage:

```
EsgDict$capabilities()
```

Returns: A named list with vocab, request, and relations.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$capabilities()
```

Method `relation_fields()`: Return supported relation-index fields.

Relation fields describe which field combinations can be used for constrained option discovery and cross-field legality checks.

Usage:

```
EsgDict$relation_fields()
```

Returns: A named list of character vectors.

Examples:

```
dict <- EsgDict$new(project = "CMIP6")
dict$relation_fields()
```

Method `fields()`: Return normalized dictionary field names.

Empty dictionaries return `character()`.

Usage:

```
EsgDict$fields()
```

Returns: A character vector.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$fields()
```

Method `indices()`: Return normalized dictionary indices.

`$indices()` returns all available indices. `$indices(type)` returns a single index table, such as "values", "variable", "activity_experiment", or "activity_source".

Usage:

```
EsgDict$indices(type = NULL)
```

Arguments:

`type` Optional index name.

Returns: A named list of indices, or a `data.table::data.table()` when `type` is supplied.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
names(dict$indices())
dict$indices("values")
```

Method `options()`: Discover valid values for a dictionary field.

Constraints supplied through `...` are used when a matching relation index exists. For example, CMIP6 `experiment_id` options can be constrained by `activity_id`.

Usage:

```
EsgDict$options(field, ...)
```

Arguments:

`field` ESG dictionary field name or supported alias.

`...` Optional field constraints.

Returns: A `data.table::data.table()` with available values and metadata. The `ignored_constraints` attribute records constraints that could not be applied.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$options("experiment_id", activity_id = "CMIP")
```

Method `check()`: Check dictionary values and relationships.

`$check()` validates supplied values against dictionary value indices and, when possible, validates cross-field combinations using relation indices.

Usage:

```
EsgDict$check(
  ...,
  error = FALSE,
  suggest = TRUE,
  n_suggestions = 5L,
  relationship = c("any", "all_pairs")
)
```

Arguments:

... ESG dictionary field values.

error If TRUE, throw an error when invalid values or relationships are found.

suggest If TRUE, include near-match suggestions for invalid values.

n_suggestions Maximum number of suggestions for each invalid value.

relationship Relationship validation mode. "any" validates ESGF-query style OR semantics. "all_pairs" requires every supplied combination inside each relation index to exist.

Returns: An `esgdict_check_result` [data.table::data.table\(\)](#).

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$check(activity_id = "CMIP", experiment_id = "historical")
dict$check(variable_id = "tas", table_id = "Amon")
```

Method `save()`: Save the dictionary to JSON.

If `path = NULL`, the dictionary is saved in the package store and registered in the store manifest. If `path` is supplied, only that JSON file is written.

Usage:

```
EsgDict$save(path = NULL, allow_empty = FALSE)
```

Arguments:

path Optional JSON file path. If NULL, use the package store.

allow_empty If TRUE, allow saving an empty dictionary.

Returns: The normalized output path.

Examples:

```
dict_path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(dict_path))

path <- tempfile(fileext = ".json")
dict$save(path)
file.exists(path)
```

Method `load()`: Load a dictionary from JSON.

If `path = NULL`, the latest stored dictionary for this project is located through the package store manifest.

Usage:

```
EsgDict$load(path = NULL)
```

Arguments:

`path` Optional JSON file path. If `NULL`, load the latest stored dictionary for this project.

Returns: The modified EsgDict object itself.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
restored <- EsgDict$new(project = "CMIP6")
suppressMessages(restored$load(path))
restored$has_data()
```

Method `print()`: Print a dictionary summary.

Usage:

```
EsgDict$print()
```

Returns: The EsgDict object itself, invisibly.

Examples:

```
path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$print()
```

Author(s)

Hongyuan Jia

See Also

[esgdict_option\(\)](#) and [esgdict_check\(\)](#) for user-facing discovery and validation helpers.

Examples

```
example_path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- esgdict(project = "CMIP6")
suppressMessages(dict$load(example_path))
dict$project()

esgdict_set_default(dict)
identical(esgdict_get_default("CMIP6"), dict)
esgdict_option("experiment_id", activity_id = "CMIP")
esgdict_check(activity = "CMIP", experiment = "historical")

## -----
## Method `EsgDict$new`
## -----
```

```

dict <- EsgDict$new(project = "CMIP6")
dict$status()

## -----
## Method `EsgDict$project`
## -----

dict <- EsgDict$new(project = "CMIP6PLUS")
dict$project()

## -----
## Method `EsgDict$profile`
## -----

dict <- EsgDict$new(project = "CMIP6")
dict$profile()

## -----
## Method `EsgDict$version`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$version()

## -----
## Method `EsgDict$sources`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$sources()

## -----
## Method `EsgDict$timestamp`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$timestamp()

## -----
## Method `EsgDict$built_time`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$built_time()

```

```

## -----
## Method `EsgDict$status`
## -----

dict <- EsgDict$new(project = "CMIP6")
dict$status()

## -----
## Method `EsgDict$has_data`
## -----

dict <- EsgDict$new(project = "CMIP6")
dict$has_data()

## -----
## Method `EsgDict$is_empty`
## -----

dict <- EsgDict$new(project = "CMIP6")
dict$is_empty()

## -----
## Method `EsgDict$build`
## -----

## Not run:
  dict <- EsgDict$new(project = "CMIP6")
  dict$build()
  dict$has_data()

## End(Not run)

## -----
## Method `EsgDict$get`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$get("experiment_id")
dict$get("request")

## -----
## Method `EsgDict$capabilities`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$capabilities()

## -----

```

```

## Method `EsgDict$relation_fields`
## -----

dict <- EsgDict$new(project = "CMIP6")
dict$relation_fields()

## -----
## Method `EsgDict$fields`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$fields()

## -----
## Method `EsgDict$indices`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
names(dict$indices())
dict$indices("values")

## -----
## Method `EsgDict$options`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$options("experiment_id", activity_id = "CMIP")

## -----
## Method `EsgDict$check`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$check(activity_id = "CMIP", experiment_id = "historical")
dict$check(variable_id = "tas", table_id = "Amon")

## -----
## Method `EsgDict$save`
## -----

dict_path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(dict_path))

path <- tempfile(fileext = ".json")

```

```

dict$save(path)
file.exists(path)

## -----
## Method `EsgDict$load`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
restored <- EsgDict$new(project = "CMIP6")
suppressMessages(restored$load(path))
restored$has_data()

## -----
## Method `EsgDict$print`
## -----

path <- system.file("extdata", "examples", "cmip6-dict.json", package = "epwshiftr")
dict <- EsgDict$new(project = "CMIP6")
suppressMessages(dict$load(path))
dict$print()

```

esgdict_check

Check ESG Dictionary Parameter Values

Description

esgdict_check() validates project parameter values against the local ESG dictionary. It checks individual field values and cross-field relationships represented in the dictionary's normalized query indices.

Usage

```

esgdict_check(
  ...,
  project = NULL,
  dict = NULL,
  error = FALSE,
  suggest = TRUE,
  n_suggestions = 5L,
  relationship = c("any", "all_pairs")
)

```

Arguments

...	ESG dictionary field values.
project	ESG project identifier. If NULL, the project is inferred from dict when supplied, otherwise "CMIP6" is used.

dict	Optional EsgDict object. If NULL, the package-level default dictionary for project is used when available; otherwise the project default dictionary is loaded from the persistent store manifest.
error	If TRUE, throw an error when invalid values or relationships are found.
suggest	If TRUE, include near-match suggestions for invalid values.
n_suggestions	Maximum number of suggestions to keep for each invalid value.
relationship	Relationship validation mode. "any" validates ESGF-query style OR semantics. "all_pairs" requires every supplied combination inside each relation index to exist.

Value

An esgdict_check_result [data.table::data.table\(\)](#).

esgdict_option	<i>Discover Valid ESG Dictionary Options</i>
----------------	--

Description

esgdict_option() returns valid values for an ESG dictionary field, optionally constrained by other supplied fields.

Usage

```
esgdict_option(field, ..., project = NULL, dict = NULL, warn_ignored = TRUE)
```

Arguments

field	An ESG dictionary field name or supported alias.
...	Optional field constraints, such as activity_id = "CMIP" or table_id = "day" for CMIP6.
project	ESG project identifier. If NULL, the project is inferred from dict when supplied, otherwise "CMIP6" is used.
dict	Optional EsgDict object. If NULL, the package-level default dictionary for project is used when available; otherwise the project default dictionary is loaded from the persistent store manifest.
warn_ignored	If TRUE, warn when supplied constraints cannot be used by any available relation index.

Value

A [data.table::data.table\(\)](#) with at least field, value, and description columns. The ignored_constraints attribute records constraints that were not used.

Description

The Earth System Grid Federation (ESGF) is an international collaboration for the software that powers most global climate change research, notably assessments by the Intergovernmental Panel on Climate Change (IPCC).

The ESGF search service exposes RESTful APIs that can be used by clients to query the contents of the underlying search index, and return results matching the given constraints. The documentation of the APIs can be found using this [link](#).

EsgQuery is the workhorse for dealing with ESGF search services. Start with `esg_query()` / `EsgQuery` for new workflow code. The legacy data.table-oriented API is available from the legacy branch or v0.1.4.

Usage

```
esg_query(index_node = "https://esgf-node.ornl.gov")
```

Arguments

- | | |
|-------------------------|---|
| <code>index_node</code> | <p>The URL to the ESGF Index Node. Default is to use the ORNL (Oak Ridge National Laboratory) Index Node. Current possible values could be:</p> <ul style="list-style-type: none"> • ORNL (Oak Ridge National Laboratory), USA: <code>https://esgf-node.ornl.gov</code>. The default value. • LLNL (Lawrence Livermore National Laboratory), USA: <code>https://esgf-node.llnl.gov</code> • NCI (National Computational Infrastructure), Australia: <code>https://esgf.nci.org.au</code> • IPSL (Institut Pierre-Simon Laplace), France: <code>https://esgf-node.ipsl.upmc.fr</code> • DKRZ (Deutsches Klimarechenzentrum), Germany: <code>https://esgf-data.dkrz.de</code> • LIU (National Academic Infrastructure for Supercomputing), Sweden: <code>https://esg-dn1.nsc.liu</code> • CEDA (Centre for Environmental Data Analysis), UK: <code>https://esgf.ceda.ac.uk</code> |
|-------------------------|---|

EsgQuery object

`esg_query()` returns an `EsgQuery` object, which is an [R6](#) object with quite a few methods that can be classified into 3 categories:

- **Value listing:** methods to list all possible values of facets, fields, shards, and values.
- **Parameter getter & setter:** methods to get the query parameter values or set them before sending the actual query to the ESGF search services.
- **Query responses:** methods to collect results for the query response.

Value listing:

`EsgQuery` object provides the following value-listing methods to query available facets, fields, shards, and values from the ESGF index node:

- `EsgQuery$list_facets()`: List all available facet names. When called, a **facet listing query** is sent to the index node to get all available facets for the current project (default: CMIP6).
- `EsgQuery$list_fields()`: List all available field names. This is useful for bridge index nodes where facet listing is not available.
- `EsgQuery$list_shards()`: List all available shards (ESGF index nodes) that can be queried in distributed searches.
- `EsgQuery$list_values()`: List all available values of specific facets.

Parameter getter & setter:

The ESGF search services support a lot of parameters. The EsgQuery contains dedicated methods to set values for most of them, including:

- Most common keywords: `facets`, `offset`, `limit`, `fields`, `replica`, `latest`, `distrib` and `shards`.
- Most common facets: `project`, `activity_id`, `experiment_id`, `source_id`, `variable_id`, `frequency`, `variant_label`, `nominal_resolution`, `datetime_range`, `timestamp_range`, `version_range` and `data_node`.

All methods act in a similar way:

- If input is given, the corresponding parameter is set and the updated EsgQuery object is returned.
 - This makes it possible to chain different parameter setters, e.g. `EsgQuery$project("CMIP6")$frequency("day")` sets the parameter project, frequency and limit sequentially.
 - For parameters that want character inputs, you can put a preceding ! to negate the constraints, e.g. `EsgQuery$project(!"CMIP6")` searches for all projects except for CMIP6.
- If no input is given, the current parameter value is returned. For example, directly calling `EsgQuery$project()` returns the current value of the project parameter. The returned value can be two types:
 - NULL, i.e. there is no constraint on the corresponding parameter
 - A QueryParam object. Use `query_param__value()` and `query_param__negate()` to inspect it.

Despite methods for specific keywords and facets, you can specify arbitrary query parameters using `EsgQuery$params()` method. For details on the usage, please see the [documentation](#).

Query responses:

The query is not sent unless related methods are called:

- `EsgQuery$count()`: Count the total number of records that match the query.
 - You can return only the total number of matched record by calling `EsgQuery$count(facets = FALSE)`
 - You can also count the matched records for specified facets, e.g. `EsgQuery$count(facets = c("source_id", "activity_id"))`
- `EsgQuery$collect()`: Collect the query results and format it into an [EsgResultDataset](#) object.

Bridge Index Nodes:

Some ESGF index nodes are "bridge" nodes that have certain limitations compared to standard index nodes. When using a bridge index node (e.g., <https://esgf-node.ornl.gov/esgf-1-5-bridge>), the following restrictions apply:

- The `fields` parameter is not supported. All available fields are always returned.
- Only Dataset and File queries are supported. Aggregation queries should use a standard ESGF search index node, such as `https://esgf-data.dkrz.de` or `https://esgf.ceda.ac.uk`.
- The `retracted` parameter is not supported and will be ignored.
- Wget script generation is not supported. Calling `$url(wget = TRUE)` will result in an error.
- Facet listing is not available. `$list_facets()` will return a predefined set of common facets instead. Use `$list_fields()` to get all available fields.

Other helpers:

EsgQuery object also provides several other helper functions:

- **Query URL generation:**
 - `EsgQuery$url()`: Returns the actual query URL or the wget script URL which can be used to download all files matching the given constraints.
- **State persistence:**
 - `EsgQuery$save()`: Save the query state to a JSON file for later use.
 - `EsgQuery$load()`: Restore the query state from a JSON file created by `$save()`.
- **Display:**
 - `EsgQuery$print()`: Print a summary of the current EsgQuery object including the index node URL and all query parameters.

Methods

Public methods:

- `EsgQuery$new()`
- `EsgQuery$index_node()`
- `EsgQuery$list_facets()`
- `EsgQuery$list_fields()`
- `EsgQuery$list_shards()`
- `EsgQuery$list_values()`
- `EsgQuery$project()`
- `EsgQuery$activity_id()`
- `EsgQuery$experiment_id()`
- `EsgQuery$source_id()`
- `EsgQuery$variable_id()`
- `EsgQuery$frequency()`
- `EsgQuery$variant_label()`
- `EsgQuery$nominal_resolution()`
- `EsgQuery$data_node()`
- `EsgQuery$facets()`
- `EsgQuery$fields()`
- `EsgQuery$shards()`
- `EsgQuery$datetime_range()`
- `EsgQuery$timestamp_range()`

- `EsgQuery$version_range()`
- `EsgQuery$replica()`
- `EsgQuery$latest()`
- `EsgQuery$limit()`
- `EsgQuery$offset()`
- `EsgQuery$distrib()`
- `EsgQuery$params()`
- `EsgQuery$url()`
- `EsgQuery$count()`
- `EsgQuery$collect()`
- `EsgQuery$state()`
- `EsgQuery$reset()`
- `EsgQuery$save()`
- `EsgQuery$load()`
- `EsgQuery$print()`
- `EsgQuery$clone()`

Method `new()`: Create a new EsgQuery object

Usage:

```
EsgQuery$new(index_node = "https://esgf-node.ornl.gov")
```

Arguments:

`index_node` The URL to the ESGF Index Node. Default is to use the **ORNL (Oak Ridge National Laboratory)** Index Node. Current possible values could be:

- ORNL (Oak Ridge National Laboratory), USA: `https://esgf-node.ornl.gov`. The default value.
- LLNL (Lawrence Livermore National Laboratory), USA: `https://esgf-node.llnl.gov`
- NCI (National Computational Infrastructure), Australia: `https://esgf.nci.org.au`
- IPSL (Institut Pierre-Simon Laplace), France: `https://esgf-node.ipsl.upmc.fr`
- DKRZ (Deutsches Klimarechenzentrum), Germany: `https://esgf-data.dkrz.de`
- LIU (National Academic Infrastructure for Supercomputing), Sweden: `https://esg-dn1.nsc.liu.se`
- CEDA (Centre for Environmental Data Analysis), UK: `https://esgf.ceda.ac.uk`

Returns: An EsgQuery object.

Examples:

```
\dontrun{
q <- EsgQuery$new(index_node = "https://esgf-node.ornl.gov")
q
}
```

Method `index_node()`: Get or set the ESGF index node.

`$index_node()` returns the current normalized index node URL. `$index_node(value)` updates the index node after applying the same normalization used by `EsgQuery$new()`. Existing query parameters are kept unchanged.

Usage:

```
EsgQuery$index_node(value)
```

Arguments:

value A string giving the new index node URL. If omitted, the current index node is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a string.

Examples:

```
\dontrun{
q$index_node()
q$index_node("https://esgf.ceda.ac.uk")
}
```

Method list_facets(): List all available facet names

Usage:

```
EsgQuery$list_facets(force = FALSE)
```

Arguments:

force By default, every facet listing query is cached and reused when possible. If TRUE, the previous cache is abandoned and a new query is re-sent and cached. Default: FALSE.

Returns: A character vector.

Examples:

```
\dontrun{
q$list_facets()
}
```

Method list_fields(): List all available field names

Usage:

```
EsgQuery$list_fields(force = FALSE)
```

Arguments:

force By default, every field listing query is cached and reused when possible. If TRUE, the previous cache is abandoned and a new query is re-sent and cached. Default: FALSE.

Returns: A character vector or NULL if no facet listing is found.

Examples:

```
\dontrun{
q$list_fields()
}
```

Method list_shards(): List all available shards.

Usage:

```
EsgQuery$list_shards(force = FALSE)
```

Arguments:

force By default, every shard listing query is cached and reused when possible. If TRUE, the previous cache is abandoned and a new query is re-sent and cached. Default: FALSE.

Returns: A character vector or NULL if no shard listing is found.

Examples:

```
\dontrun{
q$list_shards()
}
```

Method `list_values()`: List all available values of specific facets.

Usage:

```
EsgQuery$list_values(facets, force = FALSE)
```

Arguments:

`facets` A character vector giving the facet names.

`force` By default, every value listing query is cached and reused when possible. If TRUE, the previous cache is abandoned and a new query is re-sent and cached. Default: FALSE.

Returns: If `length(facets) == 1`, a named integer vector giving the facet value counts. Otherwise, a list of named integer vectors of the same length as `facets`.

Examples:

```
\dontrun{
q$list_values(c("activity_id", "experiment_id"))
}
```

Method `project()`: Get or set the project facet parameter.

Usage:

```
EsgQuery$project(value = "CMIP6")
```

Arguments:

`value` A character vector, NULL, or a negated character expression such as `!"CMIP6"`. If omitted, the current value is returned. Default when setting without an explicit value: `"CMIP6"`.

Returns: If `value` is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `activity_id()`: Get or set the activity_id facet parameter.

Usage:

```
EsgQuery$activity_id(value)
```

Arguments:

`value` A character vector, NULL, or a negated character expression such as `!c("CFMIP", "ScenarioMIP")`. If omitted, the current value is returned.

Returns: If `value` is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `experiment_id()`: Get or set the experiment_id facet parameter.

Usage:

```
EsgQuery$experiment_id(value)
```

Arguments:

`value` A character vector, NULL, or a negated character expression such as `!c("ssp126", "ssp585")`. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `source_id()`: Get or set the `source_id` facet parameter.

Usage:

`EsgQuery$source_id(value)`

Arguments:

value A character vector, NULL, or a negated character expression. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `variable_id()`: Get or set the `variable_id` facet parameter.

Usage:

`EsgQuery$variable_id(value)`

Arguments:

value A character vector, NULL, or a negated character expression. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `frequency()`: Get or set the `frequency` facet parameter.

Usage:

`EsgQuery$frequency(value)`

Arguments:

value A character vector, NULL, or a negated character expression. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `variant_label()`: Get or set the `variant_label` facet parameter.

Usage:

`EsgQuery$variant_label(value)`

Arguments:

value A character vector, NULL, or a negated character expression. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `nominal_resolution()`: Get or set the `nominal_resolution` facet parameter.

Usage:

`EsgQuery$nominal_resolution(value)`

Arguments:

value A character vector, NULL, or a negated character expression. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method data_node(): Get or set the data_node facet parameter.

Usage:

```
EsgQuery$data_node(value)
```

Arguments:

value A character vector, NULL, or a negated character expression. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method facets(): Get or set the facets parameter used by \$count().

Usage:

```
EsgQuery$facets(value)
```

Arguments:

value A character vector, "*", or NULL. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method fields(): Get or set the fields parameter.

Usage:

```
EsgQuery$fields(value = "*")
```

Arguments:

value A character vector, "*", or NULL. If omitted, the current value is returned. Default when setting without an explicit value: "*".

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method shards(): Get or set the shards parameter for distributed searches.

Usage:

```
EsgQuery$shards(value)
```

Arguments:

value A character vector or NULL. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method datetime_range(): Get or set temporal coverage overlap constraints.

Usage:

```
EsgQuery$datetime_range(start, stop)
```

Arguments:

start, stop Temporal boundary strings accepted by `solr_date()`, complete Solr range expressions, "*", or NULL. If both are omitted, the current range state is returned. The helper renders Solr constraints for the ESGF REST start/end temporal coverage keyword semantics.

Returns: If either boundary is supplied, the modified EsgQuery object. Otherwise, a list with start and stop elements.

Method `timestamp_range()`: Get or set Solr index timestamp range constraints.

Usage:

```
EsgQuery$timestamp_range(from, to)
```

Arguments:

from, to Timestamp boundary strings accepted by `solr_date()`, "*", or NULL. Complete Solr range expressions are not accepted here. If both are omitted, the current range state is returned.

Returns: If either boundary is supplied, the modified EsgQuery object. Otherwise, a list with from and to elements.

Method `version_range()`: Get or set version range constraints.

Usage:

```
EsgQuery$version_range(min, max)
```

Arguments:

min, max Version boundaries such as 20200101, "20200101", simplified dates, "*", or NULL. ESGF version is queried as a numeric field; simplified date inputs are normalized to comparable YYYYMMDD integer boundaries before rendering. Solr Date Math and complete range expressions are not accepted here. If both are omitted, the current range state is returned.

Returns: If either boundary is supplied, the modified EsgQuery object. Otherwise, a list with min and max elements.

Method `replica()`: Get or set the replica parameter.

Usage:

```
EsgQuery$replica(value)
```

Arguments:

value A flag or NULL. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `latest()`: Get or set the latest parameter.

Usage:

```
EsgQuery$latest(value = NULL)
```

Arguments:

value A flag, or NULL to remove the latest constraint. If omitted, the current value is returned.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `limit()`: Get or set the limit parameter.

Usage:

```
EsgQuery$limit(value = 10L)
```

Arguments:

value A positive integer. If omitted, the current value is returned. Default when setting without an explicit value: 10L.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `offset()`: Get or set the offset parameter.

Usage:

```
EsgQuery$offset(value = 0L)
```

Arguments:

value A non-negative integer. If omitted, the current value is returned. Default when setting without an explicit value: 0L.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `distrib()`: Get or set the distrib parameter.

Usage:

```
EsgQuery$distrib(value = TRUE)
```

Arguments:

value A flag. If omitted, the current value is returned. Default when setting without an explicit value: TRUE.

Returns: If value is supplied, the modified EsgQuery object. Otherwise, a QueryParam object or NULL.

Method `params()`: Get or set ad hoc query parameters.

`$params()` handles parameters without dedicated methods and can also update supported dedicated parameters by name. The type and format control parameters cannot be changed here: EsgQuery always performs Dataset queries and always parses JSON responses. Use `EsgResultDataset$collect()` to collect File or Aggregation records from Dataset results.

Usage:

```
EsgQuery$params(...)
```

Arguments:

... Named parameter values. If omitted, existing ad hoc parameters are returned. If a single unnamed NULL is supplied, all ad hoc parameters are removed.

Returns: If parameters are supplied, the modified EsgQuery object. Otherwise, a named list of QueryParam objects.

Method `url()`: Get the URL of actual query or wget script

The wget script URL can be used to download a bash script that contains wget commands for downloading all files matching the query constraints. This is useful for batch downloading large amounts of data.

Usage:

```
EsgQuery$url(wget = FALSE)
```

Arguments:

`wget` Whether to return the URL of the wget script that can be used to download all files matching the given constraints. Default: FALSE.

Returns: A single string.

Examples:

```
\dontrun{
q$url()

# get the wget script URL
q$url(wget = TRUE)

# You can download the wget script using the URL directly. For
# example, the code below downloads the script and save it as
# 'wget.sh' in R's temporary folder:
download.file(q$url(TRUE), file.path(tempdir(), "wget.sh"), mode = "wb")

}
```

Method `count()`: Send a query of facet counting and fetch the results

Usage:

```
EsgQuery$count(facets = TRUE)
```

Arguments:

`facets` NULL, a flag or a character vector. There are three options:

- If NULL or FALSE, only the total number of matched records is returned.
- If TRUE, the value of `$facets()` is used to limit the facets. If `$facets()` returns NULL, only the total count is returned. This is the default value.
- If a character vector, it is used to limit the facets.

Returns:

- If `facets` equals NULL or FALSE, or `$facets()` returns NULL, an integer.
- Otherwise, a named list with the first element always being `total` which is the total number of matched records. Other elements have the same length as input `facets` and are all named integer vectors.

Examples:

```
\dontrun{
# get the total number of matched records
q$count(NULL) # or q$count(facets = FALSE)

# count records for specific facets
```

```
q$facets(c("activity_id", "source_id"))$count()

# same as above
q$count(facets = c("activity_id", "source_id"))
}
```

Method `collect()`: Send the actual query and fetch the results

`$collect()` sends the actual query to the ESGF search services. By default it collects `type=Dataset` results and returns an [EsgResultDataset](#) object. If `type` is "File" or "Aggregation", it first collects matching Dataset results and then collects child File or Aggregation results for those datasets. The fields included depend on `fields` parameter. However, the following fields are always included in the results: `access`, `data_node`, `id`, `index_node`, `instance_id`, `latest`, `master_id`, `number_of_aggregations`, `number_of_files`, `replica`, `size`, `url`, `version`. When a local [EsgDict](#) is available for the query project, `$collect()` also performs a warning-only dictionary check before sending the query. Missing local dictionaries are ignored and never downloaded.

Usage:

```
EsgQuery$collect(
  all = FALSE,
  limit = TRUE,
  params = TRUE,
  type = "Dataset",
  fields = NULL,
  progress = getOption("epwshiftr.progress", interactive()),
  ...
)
```

Arguments:

`all` Whether to collect all results despite of the value of `offset`. Default: `FALSE`.

`limit` If `all = FALSE`, the maximum number of records to collect in this request. If `all = TRUE`, the page size used for each paginated request, not a total cap. When `all = TRUE` and `limit = TRUE`, the current query `limit` value is used; if `limit = FALSE`, the allowed maximum limit number 10000 is used. It can also be a positive integer used as a temporary page size. Default: `TRUE`.

`params` Whether to include facet fields that have parameter constraints explicitly set using `EsgQuery$project()`, `EsgQuery$activity_id()`, `EsgQuery$params()` and etc. in the returned fields. For example, if you set `$experiment_id("ssp585")`, the `experiment_id` field will be included in the results when `params = TRUE`. Default: `TRUE`.

`type` Result type to collect. One of "Dataset", "File", or "Aggregation". Default: "Dataset".

`fields` Optional fields used only when `type` is "File" or "Aggregation". Dataset fields should be configured with `$fields()` before collecting.

`progress` Whether to show a progress bar while collecting ESGF JSON search pages. By default, the value of option `epwshiftr.progress` is used, falling back to [interactive\(\)](#).

`...` Arguments passed to [EsgResultDataset](#) child collection when `type` is "File" or "Aggregation", including the `data_node` scope filter and child-query controls. File/Aggregation collection does not use ESGF datetime search parameters; use `$filter_time()` on the returned result for time filtering.

Returns: An [EsgResultDataset](#), [EsgResultFile](#), or [EsgResultAggregation](#) object.

Examples:

```
\dontrun{
# by default, all fields with constraints are included in the results
query <- esg_query()$experiment_id("ssp585")$frequency("1hr")$fields("source_id")
res1 <- query$collect()
res1$fields

# set `params` to `FALSE` to exclude them
query$collect(params = FALSE)$fields

# collect all matched records with `query$limit()` records per query
res2 <- query$collect(all = TRUE, limit = TRUE)
identical(query$count(), res2$count())

# same as above, but collect all matched records with max allowed
# record limit per query
res3 <- query$collect(all = TRUE, limit = FALSE)
identical(res2$count(), res3$count())

# same as above, but collect all matched records with specified limit
# per query
res4 <- query$collect(all = TRUE, limit = 30)
identical(res2$count(), res4$count())
}
```

Method `state()`: Get the current query state.

`$state()` returns a read-only snapshot containing the current index node and the current parameter state.

Usage:

```
EsgQuery$state(name = NULL, null = FALSE)
```

Arguments:

`name` A character vector of parameter names to include, or `NULL` to include all parameters.

`null` If `TRUE`, include parameters whose current value is `NULL`. Otherwise, omit unset parameters.

Returns: A named list with elements `index_node` and `parameter`.

Examples:

```
\dontrun{
q$state()
q$state(null = TRUE)
}
```

Method `reset()`: Reset query parameters to their defaults.

`$reset()` clears the current parameter store and restores the default query parameters. The current index node is kept unchanged.

Usage:

```
EsgQuery$reset()
```

Returns: The modified EsgQuery object itself.

Examples:

```
\dontrun{
q$experiment_id("ssp585")$reset()
}
```

Method `save()`: Save the query into a JSON file
`$save()` puts main data of an EsgQuery object into a JSON file which can be loaded to restore the current state of query using [EsgQuery\\$load\(\)](#).

Usage:

```
EsgQuery$save(file = "query.json", pretty = TRUE)
```

Arguments:

`file` A string indicating the JSON file path to save the data to.

`pretty` Whether to add indentation whitespace to JSON output. For details, please see [jsonlite::toJSON\(\)](#).

Default: TRUE.

Returns: The full path of the output JSON file.

Examples:

```
\dontrun{
q$save(tempfile(fileext = ".json"))
}
```

Method `load()`: Restore the query state from an JSON file
`$load()` reads data of an EsgQuery object from a JSON file created using [EsgQuery\\$save\(\)](#).

Usage:

```
EsgQuery$load(file)
```

Arguments:

`file` A string indicating the JSON file path to read the data from.

Returns: The modified EsgQuery object itself.

Examples:

```
\dontrun{
f <- tempfile(fileext = "json")

q <- esg_query()
json <- q$save(f)
q$load(f)
}
```

Method `print()`: Print a summary of the current EsgQuery object
`$print()` gives the summary of current EsgQuery object including the index node URL and all query parameters.

Usage:

```
EsgQuery$print()
```

Returns: The EsgQuery object itself, invisibly.

Examples:

```
\dontrun{
q$print()
}
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Note

For bridge index nodes, only predefined common facets are returned. `$list_fields()` can be used to get all available fields, including facets.

Author(s)

Hongyuan Jia

Examples

```
## -----
## Method `EsgQuery$new`
## -----

## Not run:
q <- EsgQuery$new(index_node = "https://esgf-node.ornl.gov")
q

## End(Not run)

## -----
## Method `EsgQuery$index_node`
## -----

## Not run:
q$index_node()
q$index_node("https://esgf.ceda.ac.uk")

## End(Not run)

## -----
## Method `EsgQuery$list_facets`
## -----

## Not run:
q$list_facets()

## End(Not run)
```

```

## -----
## Method `EsgQuery$list_fields`
## -----

## Not run:
q$list_fields()

## End(Not run)

## -----
## Method `EsgQuery$list_shards`
## -----

## Not run:
q$list_shards()

## End(Not run)

## -----
## Method `EsgQuery$list_values`
## -----

## Not run:
q$list_values(c("activity_id", "experiment_id"))

## End(Not run)

## -----
## Method `EsgQuery$url`
## -----

## Not run:
q$url()

# get the wget script URL
q$url(wget = TRUE)

# You can download the wget script using the URL directly. For
# example, the code below downloads the script and save it as
# 'wget.sh' in R's temporary folder:
download.file(q$url(TRUE), file.path(tempdir(), "wget.sh"), mode = "wb")

## End(Not run)

## -----
## Method `EsgQuery$count`
## -----

## Not run:
# get the total number of matched records
q$count(NULL) # or q$count(facets = FALSE)

```

```

# count records for specific facets
q$facets(c("activity_id", "source_id"))$count()

# same as above
q$count(facets = c("activity_id", "source_id"))

## End(Not run)

## -----
## Method `EsgQuery$collect`
## -----

## Not run:
# by default, all fields with constrains are included in the results
query <- esg_query()$experiment_id("ssp585")$frequency("1hr")$fields("source_id")
res1 <- query$collect()
res1$fields

# set `params` to `FALSE` to exclude them
query$collect(params = FALSE)$fields

# collect all matched records with `query$limit()` records per query
res2 <- query$collect(all = TRUE, limit = TRUE)
identical(query$count(), res2$count())

# same as above, but collect all matched records with max allowed
# record limit per query
res3 <- query$collect(all = TRUE, limit = FALSE)
identical(res2$count(), res3$count())

# same as above, but collect all matched records with specified limit
# per query
res4 <- query$collect(all = TRUE, limit = 30)
identical(res2$count(), res4$count())

## End(Not run)

## -----
## Method `EsgQuery$state`
## -----

## Not run:
q$state()
q$state(null = TRUE)

## End(Not run)

## -----
## Method `EsgQuery$reset`
## -----

## Not run:

```

```

q$experiment_id("ssp585")$reset()

## End(Not run)

## -----
## Method `EsgQuery$save`
## -----

## Not run:
q$save(tempfile(fileext = ".json"))

## End(Not run)

## -----
## Method `EsgQuery$load`
## -----

## Not run:
f <- tempfile(fileext = ".json")

q <- esg_query()
json <- q$save(f)
q$load(f)

## End(Not run)

## -----
## Method `EsgQuery$print`
## -----

## Not run:
q$print()

## End(Not run)

```

EsgStore

Local ESGF Store

Description

EsgStore manages a local DuckDB manifest and a fixed directory layout for query result snapshots, dictionaries, source files, downloaded NetCDF files, Parquet regional extracts, and generated outputs.

Active bindings

path Store directory.
manifest DuckDB manifest path.
is_open Whether the manifest connection is open.

Methods

Public methods:

- `EsgStore$new()`
- `EsgStore$close()`
- `EsgStore$get_meta()`
- `EsgStore$set_meta()`
- `EsgStore$download_layout()`
- `EsgStore$set_download_layout()`
- `EsgStore$register_artifact()`
- `EsgStore$artifact_path()`
- `EsgStore$validate()`
- `EsgStore$add_query()`
- `EsgStore$track_query()`
- `EsgStore$untrack_query()`
- `EsgStore$tag_query()`
- `EsgStore$untag_query()`
- `EsgStore$query_tags()`
- `EsgStore$require_query()`
- `EsgStore$unrequire_query()`
- `EsgStore$query_graph()`
- `EsgStore$queries()`
- `EsgStore$query_files()`
- `EsgStore$preview_update_queries()`
- `EsgStore$update_queries()`
- `EsgStore$download_preflight()`
- `EsgStore$download_query()`
- `EsgStore$download_status()`
- `EsgStore$query_status()`
- `EsgStore$query_updates()`
- `EsgStore$query_changes()`
- `EsgStore$workflow_status()`
- `EsgStore$workflow_report()`
- `EsgStore$remove_query()`
- `EsgStore$remove_files()`
- `EsgStore$prune_orphans()`
- `EsgStore$storage_report()`
- `EsgStore$validate_files()`
- `EsgStore$repair_files()`
- `EsgStore$cleanup_downloads()`
- `EsgStore$retry_downloads()`
- `EsgStore$add_files()`
- `EsgStore$downloader()`

- `EsgStore$download_files()`
- `EsgStore$sync_downloads()`
- `EsgStore$plan_region()`
- `EsgStore$extract()`
- `EsgStore$query()`
- `EsgStore$summarise()`
- `EsgStore$coverage()`
- `EsgStore$assert_complete()`
- `EsgStore$clone()`

Method `new()`: Create or open a local store.

Usage:

```
EsgStore$new(path = NULL, create = TRUE, overwrite = FALSE)
```

Arguments:

`path` Store directory. Default: `store_dir()`.

`create` If TRUE, create the store directory when it does not exist. Default: TRUE.

`overwrite` If TRUE, remove an existing store directory before creating a new store. Default: FALSE.

Returns: An EsgStore object.

Method `close()`: Close the DuckDB connection.

Usage:

```
EsgStore$close()
```

Returns: The store object itself, invisibly.

Method `get_meta()`: Return a store metadata value.

Usage:

```
EsgStore$get_meta(key, default = NULL)
```

Arguments:

`key` Metadata key.

`default` Value returned when key is not set.

Returns: A single string, or default.

Method `set_meta()`: Set a store metadata value.

Usage:

```
EsgStore$set_meta(key, value)
```

Arguments:

`key` Metadata key.

`value` Metadata value. NULL is stored as NA.

Returns: The store object, invisibly.

Method `download_layout()`: Return the store download layout policy.

Usage:

```
EsgStore$download_layout()
```

Returns: A named list describing how store downloads are placed under downloads/.

Method `set_download_layout()`: Configure how store-managed ESGF downloads are placed under downloads/.

Usage:

```
EsgStore$set_download_layout(
  layout = c("flat", "dataset", "drs", "template"),
  template = NULL,
  include_version = TRUE,
  collision = c("error", "checksum", "suffix"),
  missing = c("fallback", "error")
)
```

Arguments:

`layout` Download layout. "flat" stores files directly under downloads/; "dataset" groups by dataset; "drs" uses a CMIP6-style DRS path; "template" uses template.

`template` Optional subdirectory template for layout = "template", using placeholders such as {source_id}.

`include_version` Whether DRS paths include the ESGF version. Default: TRUE.

`collision` How to handle different logical files that map to the same local path. Default: "error".

`missing` How to handle missing layout fields. Default: "fallback".

Returns: The store object, invisibly.

Method `register_artifact()`: Register a file artifact in the store manifest.

Usage:

```
EsgStore$register_artifact(
  kind,
  path,
  role = NULL,
  project = NULL,
  status = "available",
  checksum = NULL,
  checksum_type = "sha256",
  size = NULL,
  query_id = NULL,
  file_key = NULL,
  dict_id = NULL,
  source_url = NULL,
  source_repo = NULL,
  source_tag = NULL,
  source_commit = NULL,
  metadata = list()
)
```

Arguments:

kind Artifact kind.
 path Artifact path. Absolute paths must be inside the store root.
 role Artifact role. If NULL, a role is inferred from kind.
 project Optional ESGF project.
 status Artifact status. Default: "available".
 checksum Expected checksum. If NULL and path exists, it is calculated with checksum_type.
 checksum_type Checksum algorithm. Default: "sha256".
 size Artifact size in bytes. If NULL and path exists, it is read from the file.
 query_id, file_key, dict_id Optional manifest links.
 source_url, source_repo, source_tag, source_commit Optional source provenance.
 metadata Optional metadata list encoded as JSON.

Returns: The artifact ID.

Method `artifact_path()`: Return an artifact path from the manifest.

Usage:

```
EsgStore$artifact_path(artifact_id)
```

Arguments:

artifact_id Artifact ID.

Returns: Absolute artifact path.

Method `validate()`: Validate registered artifact files against the manifest.

Usage:

```
EsgStore$validate()
```

Returns: A data.table with validation results.

Method `add_query()`: Add an ESGF query to the long-lived store query registry.

Usage:

```
EsgStore$add_query(query, label = NULL, track = FALSE)
```

Arguments:

query An [EsgQuery](#) object.

label Optional label.

track Whether to mark the query as tracked. Default: FALSE.

Returns: The stable query ID.

Method `track_query()`: Mark a stored ESGF query as tracked.

Usage:

```
EsgStore$track_query(query_id)
```

Arguments:

query_id Query ID returned by `$add_query()`.

Returns: The store object, invisibly.

Method `untrack_query()`: Mark a stored ESGF query as untracked.

Usage:

EsgStore\$untrack_query(query_id)

Arguments:

query_id Query ID returned by \$add_query().

Returns: The store object, invisibly.

Method tag_query(): Add tags to a stored ESGF query.

Usage:

EsgStore\$tag_query(query_id, tag, replace = FALSE)

Arguments:

query_id Query ID returned by \$add_query().

tag Character vector of tags.

replace Whether to replace existing tags for the query.

Returns: A data.table of tags for the query.

Method untag_query(): Remove tags from a stored ESGF query.

Usage:

EsgStore\$untag_query(query_id, tag = NULL)

Arguments:

query_id Query ID returned by \$add_query().

tag Optional tags. If NULL, all tags are removed.

Returns: A data.table of remaining tags for the query.

Method query_tags(): List stored ESGF query tags.

Usage:

EsgStore\$query_tags(query_id = NULL)

Arguments:

query_id Optional query ID filter.

Returns: A data.table of query tags.

Method require_query(): Record that one stored query depends on another stored query.

Usage:

EsgStore\$require_query(query_id, parent_query_id)

Arguments:

query_id Child query ID.

parent_query_id Required parent query ID.

Returns: A data.table of query dependency edges.

Method unrequire_query(): Remove query dependency edges.

Usage:

EsgStore\$unrequire_query(query_id, parent_query_id = NULL)

Arguments:

`query_id` Child query ID.

`parent_query_id` Optional parent query ID. If NULL, all parents for `query_id` are removed.

Returns: A data.table of remaining dependency edges for the query.

Method `query_graph()`: List stored query dependency edges.

Usage:

```
EsgStore$query_graph(
  query_id = NULL,
  direction = c("children", "parents", "both"),
  recursive = TRUE
)
```

Arguments:

`query_id` Optional query ID anchor.

`direction` Which edge direction to return for an anchor.

`recursive` Whether to include transitive edges.

Returns: A data.table of dependency edges.

Method `queries()`: List stored ESGF queries.

Usage:

```
EsgStore$queries(tracked = NULL)
```

Arguments:

`tracked` Optional tracked-state filter.

Returns: A data.table of stored query records.

Method `query_files()`: List files linked to a stored ESGF query.

Usage:

```
EsgStore$query_files(query_id, status = NULL)
```

Arguments:

`query_id` Query ID returned by `$add_query()`.

`status` Optional query-file status filter.

Returns: A data.table of linked file records.

Method `preview_update_queries()`: Preview tracked ESGF query updates without changing the store.

Usage:

```
EsgStore$preview_update_queries(
  query_id = NULL,
  tracked = TRUE,
  tag = NULL,
  children = FALSE,
  detail = FALSE,
  all = TRUE,
```

```

    limit = FALSE,
    fields = "*",
    ...
)

```

Arguments:

`query_id` Optional query ID. If NULL, tracked queries are previewed by default.
`tracked` Tracked-state filter used when `query_id` is NULL.
`tag` Optional query tag filter used when `query_id` is NULL.
`children` Whether to include dependency children of selected queries.
`detail` Whether to return per-file changes together with the summary. Default: FALSE.
`all, limit, fields` Arguments passed to `EsgQuery$collect()`.
`...` Additional File query filters passed to `EsgQuery$collect()`.

Returns: A data.table summary, or a list with summary and changes when `detail = TRUE`.

Method `update_queries()`: Refresh stored ESGF queries and link their current File records.

Usage:

```

EsgStore$update_queries(
  query_id = NULL,
  tracked = TRUE,
  tag = NULL,
  children = FALSE,
  enqueue = FALSE,
  downloader = NULL,
  replica = "auto",
  session_label = NULL,
  service = "HTTPServer",
  probe = TRUE,
  probe_concurrency = NULL,
  probe_cache_seconds = 3600L,
  strategy = c("fastest", "first", "stable"),
  all = TRUE,
  limit = FALSE,
  fields = "*",
  ...
)

```

Arguments:

`query_id` Optional query ID. If NULL, tracked queries are updated by default.
`tracked` Tracked-state filter used when `query_id` is NULL.
`tag` Optional query tag filter used when `query_id` is NULL.
`children` Whether to include dependency children of selected queries.
`enqueue` Whether to enqueue current files after updating. Default: FALSE.
`downloader` Optional [Downloader](#) used when `enqueue = TRUE`.
`replica` Replica policy passed to `$download_plan()` when enqueueing.
`session_label` Optional download session label.
`service` ESGF URL service used for the download plan.

probe Whether to probe candidate URLs before ranking.
 probe_concurrency Maximum concurrent URL probes when probe = TRUE. Default comes from the downloader worker count when enqueue = TRUE.
 probe_cache_seconds Seconds to reuse fresh data-node probe history before probing a URL again. Default: 3600.
 strategy Candidate ranking strategy.
 all, limit, fields Arguments passed to EsgQuery\$collect().
 ... Additional File query filters passed to EsgQuery\$collect().
Returns: A data.table of query-file links touched by the update.

Method download_preflight(): Preview a tracked query download without changing the store.

Usage:

```

EsgStore$download_preflight(
  query_id,
  downloader = NULL,
  replica = "auto",
  service = "HTTPServer",
  probe = TRUE,
  probe_concurrency = NULL,
  probe_cache_seconds = 3600L,
  strategy = c("fastest", "first", "stable"),
  all = TRUE,
  limit = FALSE,
  fields = "*",
  ...
)

```

Arguments:

query_id Query ID returned by \$add_query().
 downloader Optional [Downloader](#) used only for node history, network policy, and cooldown policy.
 replica Replica policy passed to \$download_plan().
 service, probe, strategy Download plan arguments.
 probe_concurrency Maximum concurrent URL probes when probe = TRUE. Default comes from downloader when supplied.
 probe_cache_seconds Seconds to reuse fresh data-node probe history before probing a URL again. Default: 3600.
 all, limit, fields Arguments passed to EsgQuery\$collect().
 ... Additional File query filters passed to EsgQuery\$collect().

Returns: A list with summary, changes, files, and candidates.

Method download_query(): Refresh, enqueue, and optionally run downloads for a stored ESGF query.

Usage:

```

EsgStore$download_query(
  query_id,
  downloader = NULL,
  replica = "auto",
  dry_run = FALSE,
  run = TRUE,
  background = FALSE,
  mode = c("process", "daemon"),
  session_label = NULL,
  service = "HTTPServer",
  probe = TRUE,
  probe_concurrency = NULL,
  probe_cache_seconds = 3600L,
  strategy = c("fastest", "first", "stable"),
  progress = TRUE,
  overwrite = FALSE,
  resume = TRUE,
  all = TRUE,
  limit = FALSE,
  fields = "*",
  ...
)

```

Arguments:

`query_id` Query ID returned by `$add_query()`.

`downloader` Optional [Downloader](#). Default: `$downloader()`.

`replica` Replica policy passed to `$download_plan()`.

`dry_run` Whether to return a download preflight without changing the store, enqueueing, or downloading. Default: FALSE.

`run` Whether to run the queued session immediately. Default: TRUE.

`background` Whether to run the queued session in the background. Default: FALSE.

`mode` Background execution mode. "process" starts a detached Rscript; "daemon" submits the job to a running downloader daemon.

`session_label` Optional download session label.

`service, probe, strategy` Download plan arguments.

`probe_concurrency` Maximum concurrent URL probes when `probe = TRUE`. Default comes from the downloader worker count.

`probe_cache_seconds` Seconds to reuse fresh data-node probe history before probing a URL again. Default: 3600.

`progress, overwrite, resume` Run arguments.

`all, limit, fields` Arguments passed to `EsgQuery$collect()`.

`...` Additional File query filters passed to `EsgQuery$collect()`.

Returns: The created downloader session ID, `NA_character_` when there is no pending file to download, or a one-row background job record when `run = TRUE` and `background = TRUE`.

Method `download_status()`: Return downloader tasks linked to stored query files.

Usage:

```
EsgStore$download_status(query_id = NULL, session_id = NULL, downloader = NULL)
```

Arguments:

query_id Optional stored query ID.

session_id Optional downloader session ID.

downloader Optional [Downloader](#). Default: \$downloader().

Returns: A data.table of downloader task rows.

Method query_status(): Summarise tracked ESGF query file and download status.

Usage:

```
EsgStore$query_status(query_id = NULL, downloader = NULL)
```

Arguments:

query_id Optional stored query ID vector. If NULL, all stored ESGF queries are summarised.

downloader Optional [Downloader](#). Default: \$downloader().

Returns: A data.table with one row per stored query.

Method query_updates(): List tracked query update runs.

Usage:

```
EsgStore$query_updates(query_id = NULL, latest = FALSE)
```

Arguments:

query_id Optional stored query ID filter.

latest Whether to return only the latest update per query.

Returns: A data.table of update run summaries.

Method query_changes(): List per-file changes recorded by tracked query updates.

Usage:

```
EsgStore$query_changes(update_id = NULL, query_id = NULL, change_type = NULL)
```

Arguments:

update_id Optional update run ID filter.

query_id Optional stored query ID filter.

change_type Optional change type filter.

Returns: A data.table of per-file query update changes.

Method workflow_status(): Summarise query, download, local, and extraction status together.

Usage:

```
EsgStore$workflow_status(query_id = NULL, downloader = NULL)
```

Arguments:

query_id Optional stored query ID filter.

downloader Optional [Downloader](#). Default: \$downloader().

Returns: A data.table with one row per stored query.

Method workflow_report(): Return a compact ESGF query workflow health report.

Usage:

```
EsgStore$workflow_report(query_id = NULL, downloader = NULL)
```

Arguments:

`query_id` Optional stored query ID filter.

`downloader` Optional [Downloader](#). Default: `$downloader()`.

Returns: A list with summary, updates, changes, downloads, and nodes.

Method `remove_query()`: Remove stored ESGF queries and optionally delete orphaned local files.

Usage:

```
EsgStore$remove_query(query_id, delete = c("none", "orphaned"))
```

Arguments:

`query_id` Stored query ID vector.

`delete` Whether to leave local files untouched ("none") or delete files orphaned by the removal ("orphaned").

Returns: A data.table describing removed queries.

Method `remove_files()`: Remove ESGF file records and optionally delete local artifacts.

Usage:

```
EsgStore$remove_files(file_key, delete_local = FALSE, force = FALSE)
```

Arguments:

`file_key` File key vector.

`delete_local` Whether to delete local NetCDF files. Default: FALSE.

`force` Whether to remove files still linked to queries. Default: FALSE.

Returns: A data.table describing removed file records.

Method `prune_orphans()`: Report or remove file records no longer linked to any query.

Usage:

```
EsgStore$prune_orphans(delete_local = FALSE)
```

Arguments:

`delete_local` Whether to delete local NetCDF files and remove orphaned registry records. Default: FALSE.

Returns: A data.table of orphaned file records.

Method `storage_report()`: Summarise store download storage, registered local assets, temporary files, and cleanup candidates.

Usage:

```
EsgStore$storage_report(detail = FALSE)
```

Arguments:

`detail` Whether to return detailed file tables. Default: FALSE.

Returns: A summary data.table, or a list when `detail = TRUE`.

Method `validate_files()`: Validate store-managed NetCDF downloads against the manifest.

Usage:

```
EsgStore$validate_files(query_id = NULL, checksum = FALSE, layout = TRUE)
```

Arguments:

`query_id` Optional stored query IDs to validate. When NULL, all known downloaded ESGF files are checked.

`checksum` Whether to compute file checksums. Default: FALSE.

`layout` Whether to compare registered files with the current download layout policy. Default: TRUE.

Returns: A list with summary, files, artifacts, untracked, and actions data.tables. The method is read-only.

Method `repair_files()`: Repair safe store download inconsistencies reported by `$validate_files()`.

Usage:

```
EsgStore$repair_files(actions = NULL, dry_run = TRUE)
```

Arguments:

`actions` Optional action table from `$validate_files()`\$actions. When NULL, actions are generated from `$validate_files()`.

`dry_run` Whether to only report planned repairs. Default: TRUE.

Returns: A data.table describing attempted repairs.

Method `cleanup_downloads()`: Report or remove download cleanup candidates.

Usage:

```
EsgStore$cleanup_downloads(
  scope = c("tmp", "orphan_records", "untracked_files", "missing_records"),
  dry_run = TRUE,
  older_than = NULL
)
```

Arguments:

`scope` Cleanup scopes. Supported values are "tmp", "orphan_records", "untracked_files", and "missing_records".

`dry_run` Whether to only report cleanup candidates. Default: TRUE.

`older_than` Optional age filter for file scopes. A numeric value is interpreted as seconds before now; a POSIXct value is used as an absolute mtime cutoff.

Returns: A data.table describing cleanup candidates or removals.

Method `retry_downloads()`: Requeue retryable downloader tasks linked to stored query files.

Usage:

```
EsgStore$retry_downloads(
  query_id = NULL,
  session_id = NULL,
  downloader = NULL,
  status = c("error", "cancelled"),
  run = TRUE,
  ...
)
```

Arguments:

query_id Optional stored query ID.
 session_id Optional downloader session ID.
 downloader Optional [Downloader](#). Default: `$downloader()`.
 status Retryable statuses. Default: `c("error", "cancelled")`.
 run Whether to run requested tasks immediately. Default: `TRUE`.
 ... Additional arguments passed to `Downloader$run()`.

Returns: A `data.table` of matching task rows after retry handling.

Method `add_files()`: Add File or Aggregation query results to the local file catalog.

Usage:

```
EsgStore$add_files(files, label = NULL)
```

Arguments:

files An `EsgResultFile` or `EsgResultAggregation` object.
 label Optional label for this query run.

Returns: The created or updated query ID.

Method `downloader()`: Return a `Downloader` bound to this store.

Usage:

```
EsgStore$downloader(...)
```

Arguments:

... Additional arguments passed to `Downloader$new()`.

Returns: A [Downloader](#) object.

Method `download_files()`: Enqueue and optionally download ESGF file records through the store downloader.

Usage:

```

EsgStore$download_files(
  files = NULL,
  query_id = NULL,
  replica = "auto",
  downloader = NULL,
  run = TRUE,
  background = FALSE,
  mode = c("process", "daemon"),
  session_label = NULL,
  service = "HTTPServer",
  probe = TRUE,
  probe_concurrency = NULL,
  probe_cache_seconds = 3600L,
  strategy = c("fastest", "first", "stable"),
  progress = TRUE,
  overwrite = FALSE,
  resume = TRUE,
  ...
)
```

Arguments:

files Optional [EsgResultFile](#) or [EsgResultAggregation](#) object. If supplied, it is cataloged before the download plan is created.

query_id Optional file collection query IDs to enqueue when files is NULL. If NULL, all cataloged files missing local paths are considered.

replica Replica policy passed to `$download_plan()`.

downloader Optional [Downloader](#). Default: `$downloader()`.

run Whether to run the queued session immediately. Default: TRUE.

background Whether to run the queued session in the background. Default: FALSE.

mode Background execution mode. "process" starts a detached Rscript; "daemon" submits the job to a running downloader daemon.

session_label Optional download session label.

service ESGF URL service to download from. Default: "HTTPServer".

probe Whether to lightly probe URLs before ranking them.

probe_concurrency Maximum concurrent URL probes when probe = TRUE. Default comes from the downloader worker count.

probe_cache_seconds Seconds to reuse fresh data-node probe history before probing a URL again. Default: 3600.

strategy Candidate ranking strategy.

progress Whether to show per-file download progress.

overwrite Whether to overwrite existing final files.

resume Whether to resume interrupted .part files.

... Additional arguments passed to `$download_plan()` and `Downloader$run()`.

Returns: The created downloader session ID, or a one-row background job record when run = TRUE and background = TRUE.

Method `sync_downloads()`: Register completed downloader tasks as local store artifacts.

Usage:

```
EsgStore$sync_downloads(downloader = NULL)
```

Arguments:

downloader Optional [Downloader](#). Default: `$downloader()`.

Returns: A data.table of completed tasks.

Method `plan_region()`: Plan regional extraction jobs from cataloged files.

Usage:

```
EsgStore$plan_region(
  query_id,
  lon,
  lat,
  time,
  site_id = "site-1",
  variable_id = NULL,
  filters = list(),
  method = "nearest"
)
```

Arguments:

`query_id` Query ID returned by `$add_files()`.
`lon, lat` Target longitude and latitude.
`time` Length-2 time range.
`site_id` Site identifier. Default: "site-1".
`variable_id` Optional variable IDs. If NULL, all cataloged variables in the query are used.
`filters` Named list of exact-match file catalog filters.
`method` Grid extraction method. One of "nearest", "idw", "bilinear", or "mean". Default: "nearest".

Returns: A data.table of extraction plan rows.

EsgStore\$extract(): Execute pending or failed regional extraction plans. Independent remote reads run concurrently, with at most `getOption("epwshiftr.mirai_workers", 4L)` source tasks per call. Set `options(epwshiftr.mirai_workers = 1L)` for serial reads. The same limit applies to shared batch acquisition; separate R sessions have separate limits. The calling process owns manifest and Parquet writes. With shared caching enabled, pending plans for one physical file reuse its connection. With caching disabled, workers return one plan at a time to bound memory. Complete ordinary extraction caches are consumed directly without starting workers. Speedup depends on the number of independent tasks and the remote service.

Usage:

```

EsgStore$extract(
  plan_id = NULL,
  status = c("pending", "failed"),
  fallback = c("auto", "error"),
  overwrite = FALSE,
  resume = TRUE,
  reporter = NULL
)

```

Arguments:

`plan_id` Optional plan IDs to run.
`status` Plan statuses to run when `plan_id` is NULL. Default: `c("pending", "failed")`.
`fallback` What to do when OPeNDAP open, metadata, or read access fails. "auto" downloads through HTTPServer and retries once when possible; "error" marks the plan failed without downloading. Default: "auto".
`overwrite` If TRUE, overwrite existing Parquet outputs. Default: FALSE.
`resume` Whether to reuse complete existing extraction outputs. Default: TRUE.
`reporter` Optional workflow reporter used by task-level runs.

Returns: A data.table of processed extraction plan rows.

Method query(): Run a DuckDB SQL query against the extraction manifest.

Usage:

```
EsgStore$query(sql)
```

Arguments:

`sql` SQL query.

Returns: A data.table.

Method summarise(): Summarise extracted Parquet outputs by manifest columns.

Usage:

```
EsgStore$summarise(  
  by = c("source_id", "experiment_id", "variant_label", "frequency", "variable_id",  
        "site_id", "year")  
)
```

Arguments:

by Character vector of grouping columns. Default groups by source, experiment, variant, frequency, variable, site and year.

Returns: A data.table.

Method coverage(): Check extraction coverage for planned jobs.

Usage:

```
EsgStore$coverage(plan_id = NULL)
```

Arguments:

plan_id Optional plan IDs to check.

Returns: A data.table with one row per plan.

Method assert_complete(): Assert that selected extraction plans are complete.

Usage:

```
EsgStore$assert_complete(plan_id = NULL)
```

Arguments:

plan_id Optional plan IDs to check.

Returns: The store object itself, invisibly.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Author(s)

Hongyuan Jia

install_cli	<i>Install the epwshiftr command launcher</i>
-------------	---

Description

install_cli() writes a small platform launcher that runs the current Rscript with epwshiftr::epwshiftr_cli(exit = TRUE). It does not modify shell profiles or PATH.

Usage

```
install_cli(bin_dir = NULL, name = "epwshiftr", overwrite = FALSE)
```

Arguments

bin_dir	Directory where the launcher should be written. Defaults to ~/.local/bin on macOS/Linux and %LOCALAPPDATA%/epwshiftr/bin on Windows.
name	Launcher command name. Default: "epwshiftr".
overwrite	Whether to replace an existing launcher. Default: FALSE.

Value

A data.table describing the installed launcher.

is.solr_date	<i>Check whether an object is a parsed Solr date</i>
--------------	--

Description

is.solr_date() returns TRUE when x is a SolrDate object created by solr_date() or returned unchanged from it.

Usage

```
is.solr_date(x)
```

Arguments

x	An object to test.
---	--------------------

Value

A single logical value.

See Also

[solr_date\(\)](#)

Examples

```
is.solar_date(solar_date("2025"))
is.solar_date("2025")
```

monthly_transform	<i>Configure a built-in future-weather transformation</i>
-------------------	---

Description

monthly_transform(), daily_transform(), and hourly_transform() select a scientific method on its transformation scale. The resulting WeatherTransformSpec contains no climate data, EPW template, period, site, or output path and can therefore be reused across runs.

Usage

```
monthly_transform(method, reconstruction = NULL, ...)
```

```
daily_transform(method, reconstruction = NULL, ...)
```

```
hourly_transform(method, reconstruction = NULL, ...)
```

Arguments

method	A method key returned by weather_transforms() .
reconstruction	An hourly reconstruction choice. Supply this only for methods that list more than one permitted reconstruction.
...	Method-specific scientific settings. Unknown settings are errors. For hourly kernel_qdm, include_longwave = TRUE requires and corrects rlds in all three climate roles using an experimental package extension. Its interval-mean reconstruction conserves source energy without solar weighting. model_utc_offset_hours (default 0) shifts reconstructed UTC model series to the site's fixed standard clock before year selection; observed-reference data must already use that clock. Both options are persisted in transform specifications and cache identities.

Value

A reusable WeatherTransformSpec.

See Also

[weather_transforms\(\)](#), [shift_epw_future\(\)](#)

shift_batch_get	<i>Read a persisted future-weather batch</i>
-----------------	--

Description

Reopen a batch and its independent child plans or runs without querying climate catalogs. Dry-run batches retain their child plans; executed batches retain their durable child run identifiers.

Usage

```
shift_batch_get(batch_id, store = NULL)
```

Arguments

- batch_id Batch identifier returned by `shift_ids()`.
- store Root store used for `shift_epw_future()`, or the batch directory.

Value

A ShiftBatch accepted by the ordinary shift_*() inspectors, `shift_run()`, `shift_resume()`, `shift_cancel()`, and `shift_watch()`.

shift_cmip6_avail	<i>Query CMIP6 availability by variables or weather methods</i>
-------------------	---

Description

Query shared CMIP6 Dataset metadata and identify model/member/grid identities that satisfy requested variables or registered weather-method inputs. ESGF treats multiple variables as OR alternatives; required AND combinations are evaluated locally with `data.table`.

Usage

```
shift_cmip6_avail(  
  variables = NULL,  
  scenarios = c("ssp245", "ssp585"),  
  include_historical = TRUE,  
  source = NULL,  
  member = NULL,  
  grid = NULL,  
  frequency = "day",  
  table = NULL,  
  activity = "ScenarioMIP",  
  historical_activity = "CMIP",  
  index_node = NULL,
```

```

    data_node = NULL,
    filters = list(),
    store = NULL,
    ui = NULL,
    methods = NULL,
    transform = NULL,
    common = FALSE,
    include_optional_historical = FALSE
)

```

Arguments

variables	CMIP6 variable IDs that must all be present. Supply these, methods, or transform.
scenarios	Future CMIP6 experiment IDs.
include_historical	For variable queries, require the same variables for the "historical" experiment. For method queries, historical inputs come from the method contract; leave this argument unspecified.
source	Optional CMIP6 source/model IDs. NULL discovers all models.
member	Optional CMIP6 variant labels. NULL discovers all members.
grid	Optional single CMIP6 grid label.
frequency	For variable queries, a scalar frequency or named vector assigning a frequency to each variable. Method queries infer frequencies from their contracts; leave this argument unspecified.
table	For variable queries, an optional scalar or named per-variable table selection. NULL discovers tables. Method queries discover tables at the frequencies declared by their contracts; leave this unspecified.
activity	Future CMIP6 activity ID.
historical_activity	Historical CMIP6 activity ID.
index_node	ESGF index-node name or URL. NULL uses DKRZ; "ORNL" and "LLNL" use the ORNL ESGF 1.5 Bridge endpoint.
data_node	Optional ESGF data-node filter.
filters	Additional named ESGF filters. Core availability arguments take precedence when names overlap.
store	Optional EsgStore or store path forwarded to shift_datasets() .
ui	Optional shift UI configuration forwarded to shift_datasets() .
methods	Unique weather-method keys, such as c("qdm", "sobie_curry"). Ambiguous keys require an explicit transform.
transform	One WeatherTransformSpec or a list of them, created by monthly_transform() , daily_transform() , or hourly_transform() . Use this to specify method options, such as the variables adjusted by morphing.
common	For method queries, a logical flag. FALSE (default) selects candidates separately for each method; TRUE requires eligibility for every selected method.

include_optional_historical

For method queries, include historical model inputs that the method declares optional. Mandatory inputs are always required regardless of this flag.

Details

All selected methods share one logical Dataset query for the union of their required variables, frequencies, and experiments. The existing query layer handles pagination and response caching; one logical query may require multiple HTTP requests. No File catalog or NetCDF values are downloaded. Dataset availability does not establish requested-year coverage or scientific quality. Incomplete identities found in the shared catalog remain visible. Entirely absent identities cannot be inferred from an empty catalog. Future and historical model inputs must share a variable combination, matching the requirements of the execution resolver.

Value

A data.table, including when there are no matching identities. Variable queries return one row per model/member/grid identity, retaining the existing complete, coverage counts, missing, frequency_spec, and table columns. Method queries return one row per identity/method/scenario: catalog_eligible describes that scenario including required history; method_eligible requires a single input path across all requested scenarios; common_eligible requires all methods; selected applies common. missing explains rejections. List columns variables, frequency_spec, and table describe the chosen future input path. transform_key distinguishes method configurations. period_coverage, readability, and quality remain "not_checked".

Examples

```
## Not run:
models <- shift_cmip6_avail(
  methods = c("qdm", "sobie_curry"),
  scenarios = c("ssp245", "ssp585")
)
models[selected == TRUE]

temperature <- shift_cmip6_avail(
  variables = "tas", scenarios = "ssp245", frequency = "day"
)
temperature[complete == TRUE]

## End(Not run)
```

Description

shift_*() functions provide a stage-oriented workflow facade over [EsgQuery](#), [EsgStore](#), [Downloader](#), and [EpwMorpher](#). Each step returns a small S7 stage object that can be printed, inspected, saved, and passed to the next step without manually passing manifest IDs. Method/model batches can be reopened with [shift_batch_get\(\)](#) and inspected with the same status, case, output, diagnostic, and explanation functions. [shift_watch\(\)](#) follows all active children; [shift_resume\(\)](#) starts saved child plans or resumes interrupted runs independently.

Usage

```
shift_collect(  
  x,  
  store = NULL,  
  fields = "*",  
  all = TRUE,  
  limit = FALSE,  
  label = NULL,  
  ui = NULL,  
  ...  
)  
  
shift_download(  
  x,  
  downloader = NULL,  
  run = TRUE,  
  background = FALSE,  
  resume = TRUE,  
  overwrite = FALSE,  
  session_label = NULL,  
  ui = NULL,  
  ...  
)  
  
shift_extract(  
  x,  
  site = NULL,  
  periods = NULL,  
  variables = NULL,  
  time = NULL,  
  filters = list(),  
  method = "nearest",  
  fallback = c("auto", "error"),  
  overwrite = FALSE,  
  resume = TRUE,  
  ui = NULL  
)  
  
shift_morph(  
  x,  
  store = NULL,  
  fields = "*",  
  all = TRUE,  
  limit = FALSE,  
  label = NULL,  
  ui = NULL,  
  ...  
)
```

```
x,
baseline = NULL,
transform,
reference = NULL,
observed_reference = NULL,
strict = TRUE,
complete_only = TRUE,
by = c("source_id", "experiment_id", "variant_label", "period"),
overwrite = FALSE,
resume = TRUE,
ui = NULL
)

shift_epw_write(
  x,
  dir = NULL,
  separate = TRUE,
  export_dir = NULL,
  overwrite = FALSE,
  resume = TRUE,
  ui = NULL
)

shift_check(x, strict = FALSE, network = FALSE, ...)

shift_epw_export(
  x,
  dir,
  separate = TRUE,
  overwrite = FALSE,
  resume = TRUE,
  ui = NULL
)

shift_explain(x, ...)

shift_refresh(x)

shift_ids(x, refresh = TRUE)

shift_cases(x, refresh = TRUE)

shift_missing(x)

shift_runs(store = NULL)

shift_run_get(run_id, store = NULL)
```

```
shift_result(x, store = NULL)

shift_logs(x, store = NULL, tail = 100L)

shift_files(x)

shift_data(
  x,
  n = 100L,
  variables = NULL,
  case_id = NULL,
  columns = NULL,
  refresh = TRUE
)

shift_diagnostics(x, severity = NULL, refresh = TRUE)

shift_store(x, create = FALSE)

shift_target(x)

shift_coverage(x)

shift_outputs(x, refresh = TRUE)

shift_artifacts(x)

shift_status(x, refresh = TRUE)

shift_resume(x, store = NULL, background = FALSE, ui = shift_ui())

shift_watch(
  x,
  store = NULL,
  follow = TRUE,
  interval = 1,
  events = 10L,
  ui = shift_ui()
)

shift_cancel(x, store = NULL, force = FALSE)

shift_reference_from_plan(
  plan_id,
  periods,
  role = c("model_historical", "observed_reference")
)
```

```
historical_reference(years = 1995:2014, period = "reference", ...)

shift_reference_historical(
  periods,
  experiment = "historical",
  activity = "CMIP",
  match = c("source_id", "variant_label", "frequency", "table_id", "grid_label"),
  filters = list(),
  options = list(),
  collect = list(),
  extract = list(fallback = "auto")
)

shift_ui(
  progress = c("auto", "dynamic", "log", "none"),
  detail = c("normal", "detail", "debug"),
  motion = c("auto", "full", "reduced", "none"),
  refresh = 0.12,
  heartbeat = 10
)

shift_complete(x)

shift_run(x, background = FALSE, ui = shift_ui(), ...)

shift_datasets(x, all = TRUE, limit = FALSE, store = NULL, ui = NULL)

shift_request(
  provider = "esgf",
  project = NULL,
  source = NULL,
  experiment = NULL,
  variant = NULL,
  variables = NULL,
  frequency = NULL,
  time = NULL,
  filters = list(),
  options = list(),
  ...
)

shift_site(
  id = NULL,
  lon = NULL,
  lat = NULL,
  label = NULL,
  epw = NULL,
  metadata = list()
```

```
)

shift_cmip6(
  model = 3L,
  scenarios,
  member = NULL,
  grid = NULL,
  frequency = NULL,
  table = NULL,
  activity = "ScenarioMIP",
  index_nodes = NULL,
  data_node = NULL,
  filters = list(),
  common = TRUE
)

shift_control(
  strict = TRUE,
  allow_partial = FALSE,
  download = c("auto", "always", "never"),
  resume = TRUE,
  overwrite = FALSE,
  refresh = FALSE,
  extraction_method = "nearest",
  output_layout = c("nested", "flat")
)

shift_cmip6_scenario(
  source,
  scenario,
  member = NULL,
  years = NULL,
  variables = "recommended",
  frequency = "mon",
  activity = "ScenarioMIP",
  table_id = NULL,
  grid_label = NULL,
  data_node = NULL,
  index_node = NULL,
  filters = list(),
  options = list()
)

shift_plan(
  request,
  site,
  periods,
  store,
```

```

    transform,
    reference = NULL,
    observed_reference = NULL,
    control = shift_control(),
    collect = list(),
    download = list(),
    extract = list(),
    morph = list(),
    epw = list()
)

shift_epw_future(
  sites,
  climate,
  periods,
  transform = NULL,
  dir,
  reference = NULL,
  observed_reference = NULL,
  control = shift_control(),
  ui = shift_ui(),
  store = NULL,
  dry_run = FALSE,
  background = FALSE,
  methods = NULL,
  calibration = NULL
)

```

Arguments

<code>x</code>	A shift stage object.
<code>store</code>	An EsgStore , store path, or NULL.
<code>fields</code>	File fields collected from Dataset records. The default requests all fields and lets the result/store layers preserve and validate provider response metadata.
<code>all, limit</code>	Collection controls passed to EsgQuery / EsgResultDataset . If a numeric <code>limit</code> is supplied without explicitly setting <code>all</code> , it caps the Dataset result count. With <code>all = TRUE</code> , a numeric <code>limit</code> retains the low-level meaning of pagination page size.
<code>label</code>	Optional label recorded with collected File records.
<code>ui</code>	Runtime presentation options from shift_ui() .
<code>...</code>	Additional provider-specific filters or workflow options.
<code>downloader</code>	Optional Downloader instance.
<code>run</code>	Whether to run queued downloads immediately. Downloading full NetCDF files is optional for the normal workflow because shift_extract() can use OPeN-DAP first and only download as a fallback when requested.

background	For shift_download() , whether to run queued downloads in a background job. For task-level run/resume functions, whether to launch a detached Rscript worker. Single plans return a queued ShiftRun; batches return a queued ShiftBatch whose single coordinator shares source reads and starts child workflows. Both modes honor the same source-worker limit from <code>options(epwshiftr.mirai_workers = 4L)</code> . Use shift_refresh() or shift_watch() to follow source reading and newly registered child runs.
resume	Whether to reuse complete existing downloads, extraction outputs, morphing results, or EPW outputs.
overwrite	Whether to overwrite existing downloads, extraction outputs, morphing results, or EPW outputs.
session_label	Optional download session label.
site	A shift_site() object.
periods	A period table, usually from epw_morph_periods() . shift_epw_future() and shift_plan() also accept a numeric vector of target years; each year becomes an independently named output period.
variables	Provider-neutral request alias in shift_request() , optional extraction variables in shift_extract() , or optional variables to read in shift_data() .
time	Optional request or extraction time filter. Numeric years such as 2060L are expanded to the full UTC year; otherwise supply one or two date-time values accepted by the provider/store.
filters	Provider-specific query filters in shift_request() , or extraction filters in shift_extract() .
method	Grid extraction method used by shift_extract() .
fallback	Extraction fallback policy.
baseline	Optional baseline EPW path or shift_site() object containing epw.
transform	A reusable specification from monthly_transform() , daily_transform() , or hourly_transform() .
reference	Optional ShiftReferenceSpec or ShiftClimate stage containing historical model climate when required by transform.
observed_reference	Optional plan-backed ShiftReferenceSpec or ShiftClimate stage containing observed weather when required by transform.
strict	If TRUE, abort when diagnostics contain errors.
complete_only	Whether shift_morph() should morph only complete extraction plans when a climate stage also contains failed or incomplete plans.
by	Grouping columns used to create morphing cases.
dir	In shift_epw_future() , the user-facing delivery directory. In shift_epw_write() , an output directory inside the store root; relative paths are resolved under the store root.
separate	Whether to create separate output directories per morphing case.
export_dir	Optional directory outside or inside the store where EPW files should also be copied for user-facing delivery.

network	For a reanalysis source, whether to verify the configured token against the provider. The default performs local configuration checks only and never submits a data request.
refresh	In <code>shift_control()</code> , whether to refresh remote catalogs and service addresses instead of resuming persisted inputs. In <code>shift_ui()</code> , minimum seconds between visual animation frames. In ShiftRun inspectors, whether to reload persisted state first.
run_id	Persisted workflow run ID.
tail	Maximum number of trailing execution log lines to return.
n	Maximum number of data rows to read. Use Inf to read all rows.
case_id	Optional morphing case IDs to read from morphed or EPW output stages.
columns	Optional data columns to keep.
severity	Optional diagnostic severities to keep.
create	Whether to create a store when x is a path.
follow	Whether to continue watching until the run reaches a terminal status.
interval	Polling interval in seconds.
events	Number of recent events to display or return.
force	If FALSE, request cancellation at the next safe workflow boundary. If TRUE, persist the request and then terminate the recorded background worker process immediately. For coordinated batches, send an interrupt to the owner so it can close its source workers; cooperative cancellation remains requested if the platform cannot deliver the interrupt.
plan_id	Store extraction plan IDs for manually selected reference climate data.
role	Semantic role of the plan-backed climate. Use "observed_reference" only for an observational extraction plan.
years	Optional years used to constrain the future request time window.
period	Reference period name used when constructing periods from years.
activity	CMIP6 activity ID. <code>shift_cmip6_scenario()</code> defaults to "ScenarioMIP" and <code>shift_reference_historical()</code> defaults to "CMIP".
match	File metadata fields copied from the future climate stage when resolving an automatic historical reference.
options	Provider-specific request options. For ESGF, <code>index_node</code> and <code>time_filter_method</code> are recognized.
collect	Named collection options. Historical reference collection may use <code>fields</code> , <code>all</code> , <code>limit</code> , <code>label</code> , and <code>time</code> ; <code>shift_plan()</code> applies the same strict field validation to its collection stage.
extract	Named extraction options. Historical reference extraction may use <code>variables</code> , <code>time</code> , <code>filters</code> , <code>method</code> , and <code>fallback</code> ; <code>shift_plan()</code> applies the same strict field validation to its extraction stage.
progress	In <code>shift_ui()</code> , workflow presentation mode: "auto", "dynamic", "log", or "none".

detail	Presentation detail level. "normal" shows task progress, "detail" adds selection, reuse, and fallback decisions, and "debug" also shows full URLs, paths, and low-level transfer context.
motion	Dynamic-terminal animation policy: "auto", "full", "reduced", or "none". This never changes log, JSON, or workflow data.
heartbeat	Minimum seconds between job-liveness updates. Durable writes are always throttled to at least one second. Use progress = "log" for screen readers, reduced-motion use, redirected output, and stable captured logs.
provider	Climate data provider. The first implementation supports "esgf".
project	Optional provider project, for example "CMIP6".
source, experiment, variant, frequency	Provider-neutral request fields. Values must use the selected provider's controlled vocabulary. In <code>shift_cmip6()</code> , an unnamed scalar frequency applies to every transform input, while a named character vector assigns one frequency to every source variable so 3hrPt, 3hr, and day data can be collected together. In <code>shift_reference_historical()</code> , experiment is the historical reference experiment filter. Values are not translated; for ESGF, use exact facet values such as project = "CMIP6" and frequency = "mon".
id	Optional site identifier. If id is an EPW file path and epw is NULL, it is treated as epw.
lon, lat	Optional site longitude and latitude. Missing values are read from the EPW LOCATION header when epw is supplied.
epw	A baseline EPW path, internal <code>EpwFile</code> , or external object inheriting from "Epw" in site and task APIs; in <code>shift_plan()</code> , a named EPW export option list.
metadata	Optional site metadata.
model	CMIP6 model selection. A positive whole number selects that many compatible models after complete File-year coverage is verified, preferring candidates that require fewer source files and then sorting by model/member/grid identity; a character vector selects explicit source/model IDs; NULL selects every compatible model. The default selects three models.
scenarios	CMIP6 future scenario experiment IDs.
member	CMIP6 variant label, for example "r1i1p1f1".
grid	Optional single CMIP6 grid label.
table	Optional CMIP6 table selection. NULL automatically maps each transform input to its native table (including snd to LImon); an unnamed scalar pins every variable to one table; a named character vector overrides individual variables.
index_nodes	Ordered ESGF index nodes used for failover.
data_node	Optional ESGF data node filter.
common	A logical flag for batch candidate selection. TRUE (default) uses the same model/member/grid identities across methods. FALSE selects independently: a numeric model requests that many models per method, NULL keeps all compatible models, and explicit IDs limit the eligible pool. Every explicit model must qualify for at least one method. Different pools are reported in batch diagnostics; their results confound method differences with model selection. File-year coverage is checked before either selection policy is applied.

allow_partial	Whether a task-level run may complete with missing cases.
download	Source-data policy in <code>shift_control()</code> ("auto", "always", or "never"), or a named download-stage option list in <code>shift_plan()</code> .
extraction_method	Grid extraction method.
output_layout	Output directory layout.
scenario	CMIP6 scenario experiment IDs, for example "ssp126" or "ssp585".
table_id	One or more CMIP6 table IDs. If NULL, the native table is inferred for each variable's frequency.
grid_label	Optional CMIP6 grid label.
index_node	Optional ESGF index node.
request	A <code>shift_request()</code> object, commonly from <code>shift_cmip6_scenario()</code> .
control	Workflow controls from <code>shift_control()</code> .
morph	Named morph-stage options. Stage option lists are validated and cannot override task-level controls or the transform/reference inputs.
sites	A <code>shift_site()</code> object or a non-empty list of them, each with a baseline EPW. The constructor derives omitted coordinates and labels from the EPW header. Time zone and elevation remain those of the baseline. Always returns a ShiftBatch, ordered by site ID, with separate output and store directories per location, method, and model. Candidate discovery and bounded native reads are shared; each child owns its persisted outputs. Use declarative historical/reanalysis references. For previously extracted site-specific references, use <code>shift_plan()</code> with the matching site and store.
climate	A complete future-climate specification from <code>shift_cmip6()</code> .
dry_run	If TRUE, discover eligible datasets and return a planned batch without extracting climate values or generating EPWs.
methods	One or more unambiguous method keys from <code>weather_transforms()</code> . This high-level form creates a ShiftBatch across every selected method and model.
calibration	Alias for observed_reference in high-level multi-method workflows. It is routed only to methods that accept observational input.

Value

A shift stage object.

shift_epw_reanalysis *Generate annual reference EPW files from ERA5*

Description

Create independent annual weather files without a baseline EPW or GCM. This first implementation supports hourly ERA5 single-level data and whole-hour standard timezone offsets, without daylight saving time.

Usage

```
shift_epw_reanalysis(
  source,
  sites,
  dir,
  data = NULL,
  leap_day = c("drop", "keep"),
  start_day = "calendar",
  precipitation = c("missing", "total_water_equivalent"),
  resume = TRUE
)
```

Arguments

source	A shift_era5() specification for product = "single_levels". Leave variables and frequency unset. Its years select local calendar years.
sites	One shift_site() or a list of sites. Each site's metadata must provide timezone (standard UTC offset in hours) and elevation (metres). Optional state and country appear in the EPW header.
dir	Directory for source caches, immutable conversion attempts and receipts.
data	Optional local NetCDF path(s), an explicit data bundle, or a named list of those inputs keyed by site ID. Local input never contacts CDS. A bundle contains data (a data.frame), units (a named character vector), grid (latitude/longitude), interval_seconds = 3600, and optional provenance. Its columns are utc_time (ordered POSIXct validity times), tas, tdps, ps, uas, vas, rsds, fdir, rlds, clt, and pr. NetCDF input must already use the hourly ERA5 single-level validity-time convention; forecast-step/ensemble dimensions must be resolved before calling. No implicit accumulation reset is inferred. With NULL, the existing CDS transport retrieves small monthly requests; this is not a bulk-download accelerator.
leap_day	Keep February 29 (8784 hours in leap years), or drop it for a 8760-hour calendar. The omitted source hours remain in an artifact.
start_day	"calendar" uses the actual January 1 weekday. Alternatively supply an English weekday, e.g. "Monday", for a fixed simulation schedule.
precipitation	Default "missing" leaves EPW liquid precipitation missing because ERA5 total precipitation does not resolve its phase. "total_water_equivalent" explicitly writes that total into the liquid precipitation field, with a provenance warning.
resume	Reuse successful results only when inputs, implementation, options and all output hashes match.

Details

Instantaneous variables are assigned at hour end. Radiation and precipitation cover the preceding hour: accumulated radiation in J/m2 is divided by 3600; hourly mean W/m2 and hourly Wh/m2 have the same numeric EPW value. FDIR is direct *horizontal* radiation. Diffuse radiation is SSRD minus FDIR; DNI uses the shared hourly solar projection at the target site's coordinates. Negative

energy, inconsistent fields, missing hours and impossible DNI fail that annual job without clipping or substituting weather.

Relative humidity uses the ECMWF liquid-water saturation relation, including below freezing. Original dew point is retained. EnergyPlus interprets RH using a different saturation relation below freezing; the accompanying hourly diagnostics quantify the resulting humidity-ratio difference. Thus a complete conversion does not establish moisture equivalence for building simulation. Grid values are not elevation-adjusted to the target site.

Unsupported weather fields use documented EPW missing sentinels. Ground temperatures, design conditions, holidays and typical/extreme periods are not fabricated. These and precipitation/snow assumptions must be configured as appropriate in the simulation model. This creates reanalysis annual weather, not observations or a typical meteorological year. Leap-day removal changes calendar continuity; original timestamps and omitted hours remain available. Solar conversion uses each source record's original calendar date.

Value

A data.table with one row per site/year, status, reuse flag, EPW and receipt paths, and any error. Partial failures are retained and warned. complete means conversion and EPW readback passed, not building validation.

Examples

```
site <- shift_site("example", lon = 113.3, lat = 23.2,
  metadata = list(timezone = 8, elevation = 10))
## Not run:
shift_epw_reanalysis(shift_era5(1995), site, "reference-weather",
  data = c("instantaneous.nc", "accumulated.nc"))

## End(Not run)
```

shift_era5

Use ERA5 as an observed weather reference

Description

shift_era5() describes an ERA5 calibration source. Variables, temporal frequency, and the target coordinate are normally inferred from the weather method and baseline EPW when [shift_epw_future\(\)](#) builds the run. Constructing the source neither reads credentials nor contacts CDS. Use shift_check(source) to validate local configuration and shift_check(source, network = TRUE) to authenticate the configured token without submitting a data request. Dataset-specific terms are checked by CDS when retrieval begins and are never accepted by the package.

Usage

```
shift_era5(
  years,
  product = c("single_levels", "land"),
```

```
variables = NULL,  
frequency = NULL,  
access = c("auto", "arco", "cds"),  
...  
)
```

Arguments

years	Calendar years used as the observed reference period.
product	ERA5 single-level product. "single_levels" uses global ERA5; "land" uses ERA5-Land.
variables	Optional CF variable IDs. NULL infers them from the selected weather method.
frequency	Optional "hour", "day", or "mon" frequency. A named vector can assign frequencies by CF variable. NULL infers the contract.
access	Data access route. "auto" prefers the official ARCO-backed time-series service for supported point requests and otherwise uses the complete CDS subset service.
...	Named provider options retained in the source specification.

Details

rsdsdiff is derived from matched hourly SSRD minus FDIR accumulations (direct horizontal solar radiation, not DNI). It requires the full CDS single-level product; automatic access falls back from ARCO to CDS.

Value

A reusable ShiftReanalysisSpec containing no credentials.

References

[Copernicus Climate Data Store ERA5 hourly time-series data](#) and the [CDS API setup guide](#).

See Also

[shift_epw_future\(\)](#), [shift_check\(\)](#)

shift_era6	<i>Describe the future ERA6 source boundary</i>
------------	---

Description

ERA6 production has started, but a stable public dataset endpoint is not yet available. This constructor fails explicitly so ERA5 is never substituted or spliced into a request labelled ERA6.

Usage

```
shift_era6(
  years,
  product = "single_levels",
  variables = NULL,
  frequency = NULL,
  access = c("auto", "arco", "cds"),
  ...
)
```

Arguments

years	Calendar years used as the observed reference period.
product	ERA5 single-level product. "single_levels" uses global ERA5; "land" uses ERA5-Land.
variables	Optional CF variable IDs. NULL infers them from the selected weather method.
frequency	Optional "hour", "day", or "mon" frequency. A named vector can assign frequencies by CF variable. NULL infers the contract.
access	Data access route. "auto" prefers the official ARCO-backed time-series service for supported point requests and otherwise uses the complete CDS subset service.
...	Named provider options retained in the source specification.

Value

This function currently raises an unavailable-data error.

References

ECMWF, ERA6 reanalysis production.

shift_history	List saved workflow runs and batches
---------------	--------------------------------------

Description

Read local run records and batch receipts without contacting climate services or creating stores. Child runs include their batch ID and exact store path. Unreadable records are returned with status unavailable and an error message.

Usage

```
shift_history(store = NULL, type = c("all", "run", "batch"), status = NULL)
```

Arguments

store	Root workflow store, batch directory, or an open EsgStore.
type	Include all records, only runs, or only batches.
status	Optional character vector of states to retain.

Value

A data.table with type, full ID, status, parent batch, method, model, update time, store path, and inspection error, sorted newest first.

shift_selection	<i>Inspect Recorded Climate Input Selection</i>
-----------------	---

Description

Read the original workflow configuration, pinned selection, and candidate checks saved during File-catalog resolution. No remote queries or candidate recalculation are performed. Missing records in older runs remain explicitly unavailable.

Usage

shift_selection(x)

Arguments

x	A ShiftPlan, finished ShiftRun, a stage associated with a run, or a ShiftBatch. Running jobs must finish before durable inspection.
---	---

Details

The scope is the File-level candidates evaluated for requested models. It does not include every model in ESGF, candidates excluded by upstream Dataset discovery or server filters, or a history of arbitrary user filters. eligible_alternative_not_selected describes execution selection, not a scientific ranking. future_coverage_incomplete and no_complete_matching_reference describe coverage or input pairing. user_member_constraint and user_grid_constraint identify explicit limits when those alternatives occur in the evaluated table. An empty check list means failure occurred before candidates could be evaluated. An attempt may fail after selecting candidates, for example during service access.

Plans return state = "planned". Finished runs without captured candidates return state = "not_recorded", retaining any saved configuration and final selection. Captured attempts return state = "recorded", which does not imply successful weather generation. Inspect the attempt outcome and cases.

Value

A list with state, run_id, requested, resolved, attempts, cases, and diagnostics. Batches return a named list per child. Each attempt identifies its index node and outcome. Its checks retain candidate tables before and after service resolution, query IDs, required variables and explicit constraints. Future candidate rows carry a reason.

shift_summary	<i>Summarize future-weather outputs for comparison</i>
---------------	--

Description

Compare method/model/scenario/period groups without conflating cases with physical EPW files. Optional weather statistics read existing local EPWs and report means and valid-hour counts; they do not rank methods or imply that different output types or calendar periods are scientifically equivalent. For standalone staged runs, case counts describe distinct cases with saved outputs; the method is recovered from the result's morphing stage.

Usage

```
shift_summary(
  x,
  store = NULL,
  refresh = TRUE,
  weather = FALSE,
  ui = shift_ui()
)
```

Arguments

x	A ShiftBatch, ShiftRun, or ShiftPlan; a run ID is also accepted.
store	Store used when x is a run ID.
refresh	Whether to reload saved workflow state before inspection.
weather	Whether to read local EPWs and include hourly weather means. Missing files and read errors are counted explicitly. Missing EPW sentinel codes are excluded, and every mean includes its valid-hour denominator.
ui	Presentation options from shift_ui() for optional EPW reads.

Value

A data.table with one row per site (when supplied), method, model, scenario and period (and member/grid where applicable), counts, field roles and optional means.

solr_date	<i>Parse a Solr date, Date Math expression, or range</i>
-----------	--

Description

solr_date() parses a scalar input into an internal S7 SolrDate object. The resulting object can represent a single instant, a Date Math expression, an unbounded boundary (*), or a Solr range.

Usage

```
solr_date(x)
```

Arguments

- x** A scalar input to parse. Supported inputs are:
- an existing SolrDate object, which is returned unchanged;
 - a scalar Date or POSIXt object;
 - a scalar numeric value, which is first converted to character;
 - a scalar character string representing either a single boundary or a complete Solr range expression.
- POSIXt inputs must use the "UTC" timezone.

Details

Character inputs support the following forms:

- Simplified dates such as "2025", "2025-02", "2025-02-03", and "20250203".
- Datetimes accepted by the internal parser, including ISO-like forms such as "2025-01-15T12:30:45Z", timezone offsets like "+08:00", and common separators such as "/" and ".".
- Solr Date Math expressions rooted at NOW, e.g. "NOW", "NOW-1YEAR", or "NOW/DAY-1YEAR+6MONTHS".
- Fixed-base Date Math expressions of the form "<datetime>Z<math>", e.g. "2025-01-01T00:00:00Z+1MONTH".
- Solr range expressions using the exact separator " TO " and boundary brackets \[\] or \{ \}, e.g. "[2000 TO 2010]", "{2000 TO 2010}", or "[* TO *]".

Supported Date Math operators are +, -, and /. Supported units are YEAR, YEARS, MONTH, MONTHS, DAY, DAYS, DATE, HOUR, HOURS, MINUTE, MINUTES, SECOND, SECONDS, MILLI, MILLIS, MILLISECOND, and MILLISECONDS.

Use format() or as.character() to render a parsed value. format() supports as = "iso" and as = "num". as.POSIXct() can be used on instants; for ranges it returns the start boundary with a warning, and for unbounded or Date Math values it errors because no single concrete instant is available.

Value

An internal S7 object inheriting from SolrDate. The exact subclass is an implementation detail and may represent a single instant, a Date Math expression, an unbounded boundary, or a range.

See Also

[is.solr_date\(\)](#)

Examples

```

solr_date("2025")
solr_date("2025-02")
solr_date("20250203")
solr_date("2025-01-15T12:30:45Z")

solr_date("NOW")
solr_date("NOW/DAY-1YEAR+6MONTHS")
solr_date("2025-01-01T00:00:00Z+1MONTH")

solr_date("[2000 TO 2010]")
solr_date("{2000 TO 2010}")
solr_date("[* TO *]")

x <- solr_date("2025-01-15T12:30:45Z")
format(x)
format(x, as = "num")
as.character(x)
as.POSIXct(x)
is.solr_date(x)
print(x)

```

store_dir	<i>Get the epwshiftr store directory</i>
-----------	--

Description

store_dir() returns the root directory used for persistent epwshiftr store artifacts, including query snapshots, dictionaries, sources, downloads, extracted data, generated outputs, and the DuckDB manifest.

Usage

```
store_dir(init = TRUE)
```

Arguments

init If TRUE, create the directory when it does not exist.

Value

A single string indicating the directory location.

store_reset	<i>Recreate an epwshiftr store</i>
-------------	------------------------------------

Description

store_reset() creates a current-schema store at path. Existing stores are moved to a timestamped sibling by default, preserving query snapshots, downloads, extracted climate data, generated EPWs, and run history for manual recovery. Permanent removal requires both backup = FALSE and force = TRUE.

Usage

```
store_reset(path = store_dir(init = FALSE), backup = TRUE, force = FALSE)
```

Arguments

path	Store directory to recreate. Defaults to store_dir() .
backup	Whether to move an existing store to a timestamped sibling before creating the replacement. Default: TRUE.
force	Required when backup = FALSE would permanently remove an existing populated store. Default: FALSE.

Details

This function does not migrate older manifests. Close open [EsgStore](#) objects and stop active foreground or background jobs before resetting.

Value

Invisibly, a list containing the new path, optional backup_path, previous and current schema versions, and the performed action.

uninstall_cli	<i>Uninstall the epwshiftr command launcher</i>
---------------	---

Description

Remove a launcher generated by install_cli().

Usage

```
uninstall_cli(bin_dir = NULL, name = "epwshiftr")
```

Arguments

- `bin_dir` Directory where the launcher should be written. Defaults to `~/ .local/bin` on macOS/Linux and `%LOCALAPPDATA%/epwshiftr/bin` on Windows.
- `name` Launcher command name. Default: `"epwshiftr"`.

Value

A `data.table` describing the uninstall result.

<code>weather_transforms</code>	<i>List built-in future-weather transformations</i>
---------------------------------	---

Description

`weather_transforms()` lists the same public scale, method, reconstruction, source-frequency, input-role, evidence, and output contracts used by the three transform constructors.

Usage

`weather_transforms()`

Value

A data table with one row per selectable configuration.

See Also

[monthly_transform\(\)](#)

Index

[.EsgResult, 5

daily_transform(monthly_transform), 93
daily_transform(), 23, 95, 103
data.table, 35
data.table::data.table(), 6, 50, 51, 57
data_node_status, 5
Downloader, 6, 82–86, 88, 89, 97, 102

epw_morph_backend, 20
epw_morph_backends, 20
epw_morph_periods, 21
epw_morph_periods(), 27, 28, 30, 103
epw_morph_register_backend, 21
epw_morph_result, 22
epw_morpher, 22
EpwMorphBackend, 20, 21, 23
EpwMorpher, 23, 26, 97
epwshiftr(epwshiftr-package), 4
epwshiftr-package, 4
epwshiftr_cli, 31
esg_query(EsgQuery), 58
esg_result, 32
EsgDataset, 33
EsgDict, 44, 44, 57, 69
esgdict(EsgDict), 44
esgdict_check, 56
esgdict_check(), 52
esgdict_get_default(EsgDict), 44
esgdict_option, 57
esgdict_option(), 52
esgdict_set_default(EsgDict), 44
EsgQuery, 58, 79, 97, 102
EsgResult, 5, 32
EsgResultAggregation, 69, 89
EsgResultDataset, 59, 69, 102
EsgResultFile, 69, 89
EsgStore, 12, 23, 26, 75, 95, 97, 102, 115

hourly_transform(monthly_transform), 93
hourly_transform(), 23, 95, 103

install_cli, 92
interactive(), 4, 34, 69
is.solr_date, 92
is.solr_date(), 113

jsonlite::toJSON(), 71

monthly_transform, 93
monthly_transform(), 23, 95, 103, 116

R6, 44, 58

shift_api(shift_collect), 96
shift_artifacts(shift_collect), 96
shift_batch_get, 94
shift_batch_get(), 97
shift_cancel(shift_collect), 96
shift_cancel(), 94
shift_cases(shift_collect), 96
shift_check(shift_collect), 96
shift_check(), 109
shift_cmip6(shift_collect), 96
shift_cmip6(), 106
shift_cmip6_avail, 94
shift_cmip6_scenario(shift_collect), 96
shift_collect, 96
shift_complete(shift_collect), 96
shift_control(shift_collect), 96
shift_control(), 104, 106
shift_coverage(shift_collect), 96
shift_data(shift_collect), 96
shift_datasets(shift_collect), 96
shift_datasets(), 95
shift_diagnostics(shift_collect), 96
shift_download(shift_collect), 96
shift_download(), 103
shift_epw_export(shift_collect), 96
shift_epw_future(shift_collect), 96

shift_epw_future(), [93](#), [94](#), [103](#), [108](#), [109](#)
shift_epw_reanalysis, [106](#)
shift_epw_write(shift_collect), [96](#)
shift_epw_write(), [103](#)
shift_era5, [108](#)
shift_era5(), [107](#)
shift_era6, [109](#)
shift_explain(shift_collect), [96](#)
shift_extract(shift_collect), [96](#)
shift_extract(), [102](#), [103](#)
shift_files(shift_collect), [96](#)
shift_history, [110](#)
shift_ids(shift_collect), [96](#)
shift_ids(), [94](#)
shift_logs(shift_collect), [96](#)
shift_missing(shift_collect), [96](#)
shift_morph(shift_collect), [96](#)
shift_morph(), [103](#)
shift_outputs(shift_collect), [96](#)
shift_plan(shift_collect), [96](#)
shift_plan(), [103–106](#)
shift_reference_from_plan
 (shift_collect), [96](#)
shift_reference_historical
 (shift_collect), [96](#)
shift_refresh(shift_collect), [96](#)
shift_refresh(), [103](#)
shift_request(shift_collect), [96](#)
shift_request(), [103](#), [106](#)
shift_result(shift_collect), [96](#)
shift_resume(shift_collect), [96](#)
shift_resume(), [94](#), [97](#)
shift_run(shift_collect), [96](#)
shift_run(), [94](#)
shift_run_get(shift_collect), [96](#)
shift_runs(shift_collect), [96](#)
shift_selection, [111](#)
shift_site(shift_collect), [96](#)
shift_site(), [106](#), [107](#)
shift_status(shift_collect), [96](#)
shift_store(shift_collect), [96](#)
shift_summary, [112](#)
shift_target(shift_collect), [96](#)
shift_ui(shift_collect), [96](#)
shift_ui(), [102](#), [104](#), [112](#)
shift_watch(shift_collect), [96](#)
shift_watch(), [94](#), [97](#), [103](#)
solr_date, [112](#)
solr_date(), [92](#)
store_dir, [114](#)
store_dir(), [77](#), [115](#)
store_reset, [115](#)

tools::R_user_dir(), [4](#)

uninstall_cli, [115](#)

weather_transforms, [116](#)
weather_transforms(), [93](#), [106](#)