

Package: ssdsims (via r-universe)

October 15, 2026

Title Simulation Analyses for Species Sensitivity Distributions

Version 0.0.0.9017

Description Runs reproducible simulation studies for species sensitivity distribution (SSD) models built on the 'ssdtools' package. Expands a declarative scenario into per-step task tables, draws data, fits distributions, and estimates hazard concentrations, with a 'targets'-based, Hive-partitioned shard pipeline for running studies in parallel or on a cluster.

License Apache License (== 2.0) | file LICENSE

URL <https://poissonconsulting.github.io/ssdsims/>

BugReports <https://github.com/poissonconsulting/ssdsims/issues>

Depends R (>= 4.3)

Imports chk, digest, dplyr, dqrng (>= 0.4.1), duckplyr, jsonlite, purrr, rlang, sessioninfo, ssdtools (>= 2.6.0), tibble, tidyr, withr

Suggests AzureRMR, AzureStor, callr, crew, crew.cluster, knitr, quarto, readr, roxyglobals, ssddata, tarchetypes, targets, testthat (>= 3.0.0), tidyselect

VignetteBuilder quarto

Remotes poissonconsulting/ssdtools

Config/testthat/edition 3

Config/roxyglobals/unique FALSE

Encoding UTF-8

Roxygen list(markdown = TRUE, roclets = c("`collate", "`namespace", "`rd", "`roxyglobals::global_roclet"))

Config/roxygen2/version 8.0.0.9000

LazyData true

Config/pak/sysreqs libicu-dev libx11-dev xz-utils

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

Date/Publication 2026-09-01 07:14:39 UTC

RemoteUrl <https://github.com/poissonconsulting/ssdsims>

RemoteRef HEAD

RemoteSha cd3d89a26c9e21a951c2927c373ae5a5ae4b4700

Contents

local_dqrng_backend	3
local_dqrng_state	4
scenario_dataset	5
scenario_distset	6
scenario_min_pmix	7
scenario_results_dir	8
ssd_analyse_cost	9
ssd_calibrate_cost	10
ssd_calibrate_cost_from_run	11
ssd_compare_cost	12
ssd_cost_calibration	13
ssd_cost_calibration_default	13
ssd_define_scenario	14
ssd_design	19
ssd_design_targets	20
ssd_distset	22
ssd_estimate_cost	23
ssd_gen	24
ssd_open_uploaded	26
ssd_pmix	27
ssd_run_scenario_baseline	28
ssd_run_scenario_shards	29
ssd_run_step	30
ssd_scenario_data	33
ssd_scenario_shards	34
ssd_scenario_targets	35
ssd_scenario_tasks	38
ssd_summarise	40
ssd_summarise_design	42
ssd_summarise_member	43
ssd_summarise_uploaded	44
ssd_test_upload	46
ssd_upload_azure	47
ssd_upload_shard	48
task_primer	50

Index

52

local_dqrng_backend	<i>Local dqrng pcg64 Backend</i>
---------------------	----------------------------------

Description

Activates the dqrng pcg64 RNG backend for the duration of the calling frame, then resets it when `.local_envir` exits. While active, base R's `runif()`, `rnorm()`, `rbinom()`, `rexp()`, `rgamma()`, `rpois()`, `sample.int()`, and `sample()` (and therefore `dplyr::slice_sample()` and `ssdtools::ssd_r*`()) draw from dqrng's pcg64, seeded via `dqrng::dqset.seed()`. pcg64 is forced explicitly because it accepts the length-2 stream argument the per-task primer design relies on; dqrng's own default (Xoroshiro128++) does not.

Usage

```
local_dqrng_backend(.local_envir = parent.frame())
```

Arguments

<code>.local_envir</code>	[environment]
---------------------------	---------------

The environment to use for scoping.

Details

Registering the backend is a process-global side effect that also advances base R's `.Random.seed`. `local_dqrng_backend()` follows the `withr` convention (compare `withr::local_seed()`): it pairs activation with deferred reset so the backend is always restored, including on error.

The helper is reentrant. `dqrng::register_methods()` / `dqrng::restore_methods()` keep a single global save-slot, so a nested reset would tear the backend down for the still-open outer scope. To avoid this, a `local_dqrng_backend()` call made while the backend is already active is a no-op: it does not re-activate the backend and schedules no further reset. Only the outermost call activates the backend on entry and resets it on exit, so the RNG stream is identical whether or not a nested call occurs.

Value

Invisibly returns `TRUE` if this call activated the backend (the outermost scope) or `FALSE` if the backend was already active and the call was a no-op.

See Also

`withr::local_seed()`, `dqrng::dqset.seed()`.

Examples

```
local_dqrng_backend()
dqrng::dqset.seed(42, stream = c(1L, 2L))
runif(3)
```

local_dqrng_state	<i>Local/With dqrng State</i>
-------------------	-------------------------------

Description

local_dqrng_state() installs a per-task (seed, primer) starting point as the running dqrng RNG state via `dqrng::dqset.seed()`, restoring the previous state when `.local_envir` exits. `with_dqrng_state()` evaluates code with that state installed, then restores the previous state. The primer argument is the per-task primer (the value handed to dqrng's stream argument, per TARGETS-DESIGN.md §2 and the GLOSSARY); the `_state` suffix marks that the wrapper installs that primer as the running RNG state.

Usage

```
local_dqrng_state(seed, primer, .local_envir = parent.frame())
```

```
with_dqrng_state(seed, primer, code)
```

Arguments

seed	[whole number] A scalar seed passed to <code>dqrng::dqset.seed()</code> .
primer	[integer(2)] A length-2 integer primer passed as the stream argument of <code>dqrng::dqset.seed()</code> . NA_integer_ is permitted (the reserved INT_MIN encoding of TARGETS-DESIGN.md §2).
.local_envir	[environment] The environment to use for scoping.
code	[any] Code to execute in the temporary environment

Details

They snapshot the RNG state on entry (via `dqrng::dqrng_get_state()`) and `withr::defer()` a restore (via `dqrng::dqrng_set_state()`), so a call leaves the surrounding RNG stream undisturbed, including on error.

Both require an active dqrng backend: they abort unless a `local_dqrng_backend()` scope is open. This fails fast rather than silently seeding base R's Mersenne-Twister.

Value

local_dqrng_state() invisibly returns primer; with_dqrng_state() returns the value of code.

See Also

`withr::local_seed()`, `local_dqrng_backend()`.

Examples

```

local_dqrng_backend()
local_dqrng_state(42, c(1L, 2L))
runif(3)

with_dqrng_state(42, c(1L, 2L), runif(3))

```

scenario_dataset

Isolate a Materialised Dataset from a Scenario by Name

Description

Returns the validated, materialised dataset tibble stored on scenario under name. The dataset was validated (a numeric Conc column) and materialised at construction by `ssd_define_scenario()`, so this accessor performs no registration, persistence, or re-validation - it just isolates the value a shard body fits. Aborts with an informative error when name is not one of the scenario's datasets.

Usage

```
scenario_dataset(scenario, name)
```

Arguments

scenario	An ssdsims_scenario from <code>ssd_define_scenario()</code> .
name	A scalar string naming one of the scenario's datasets.

Details

Names - not values - drive task hashing (TARGETS-DESIGN.md section 1.1): the task path carries the dataset *name* and this accessor resolves it back to the tibble at run time, so the tibble never enters a task identity.

Value

The materialised dataset tibble stored under name.

See Also

`scenario_min_pmix()` for the min_pmix counterpart.

Examples

```

data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 1L, seed = 42L)
scenario_dataset(scenario, "ccme_boron")

```

scenario_distset

Isolate a Distribution Set from a Scenario by Name

Description

Returns the member character vector of the distribution set stored on scenario under name (from scenario\$hc\$distsets). `ssd_define_scenario()` validates each set at construction (via `ssd_distset()`), so this accessor performs no registration, persistence, or re-validation - it just isolates the members the hc runner subsets the parent union fit by. Aborts with an informative error when name is not one of the scenario's distribution-set names.

Usage

```
scenario_distset(scenario, name)
```

Arguments

scenario	An ssdsims_scenario from <code>ssd_define_scenario()</code> .
name	A scalar string naming one of the scenario's distribution sets.

Details

Names - not members - drive task hashing (TARGETS-DESIGN.md section 1.1): the hc-task path carries the distset *name*, and this accessor resolves it back to the member vector at run time, so the members never enter a task identity (the same by-name pattern as `min_pmix` and `datasets`).

Value

The member character vector of the distribution set stored under name.

See Also

`ssd_distset()`, `scenario_min_pmix()`.

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 1L,
  seed = 42L,
  dists = ssd_distset(BCANZ = ssdtools::ssd_dists_bcanz())
)
scenario_distset(scenario, "BCANZ")
```

scenario_min_pmix	<i>Isolate a Materialised min_pmix Function from a Scenario by Name</i>
-------------------	---

Description

Returns the single-argument min_pmix function materialised on scenario under name. `ssd_define_scenario()` resolves each min_pmix reference to a function once, at construction (so a cluster worker needs no shared interactive environment), and stores it keyed by name; this accessor isolates it. Aborts with an informative error when name is not one of the scenario's min_pmix names.

Usage

```
scenario_min_pmix(scenario, name)
```

Arguments

scenario	An ssdsims_scenario from <code>ssd_define_scenario()</code> .
name	A scalar string naming one of the scenario's min_pmix entries.

Details

Names - not function values - drive task hashing (TARGETS-DESIGN.md section 1.1): the fit-task path carries the min_pmix *name*, and this accessor resolves it back to the function at run time, so the function value never enters a task identity (no byte-stability concern from byte-compilation or captured environments).

Value

The single-argument min_pmix function stored under name.

See Also

`scenario_dataset()` for the dataset counterpart.

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 1L, seed = 42L)
scenario_min_pmix(scenario, "ssd_min_pmix")
```

scenario_results_dir *Seed- and Layout-keyed Results Root for a Scenario*

Description

Returns `<root>/seed=<value>/layout=<hash>`, where the hash is derived from the scenario's `partition_by`. The leading `seed=<value>` level isolates each scenario's RNG streams (scenarios that differ only in seed share no draws, so they never mix shards) and - crucially - makes a single-scenario run and a design-of-one (`ssd_design_targets(ssd_design(scenario))`) address shards **identically**, so wrapping a scenario into a design reuses its existing shards rather than recomputing them. A step's Hive shard path depth and axes are a function of `partition_by/bundle`, so writing two different layouts into one root would leave shards of *different granularity* side by side - and the depth-agnostic glob the readers use (`<step>/**/part.parquet`) would then union stale and current shards, double-counting tasks. Keying the results root on the layout isolates each `partition_by` into its own subtree: re-running a scenario with a changed `partition_by/bundle` writes to a *fresh* root (never mixing granularities), while re-running the *same* layout reuses the root (idempotent and cache-friendly - the same shard paths are simply rewritten).

Usage

```
scenario_results_dir(scenario, root = "results")
```

Arguments

scenario	An <code>ssdsims_scenario</code> from <code>ssd_define_scenario()</code> .
root	The results root directory (default "results").

Details

The targets pipeline writes under this root (see the shipped `_targets.R` template). The single-core `ssd_run_scenario_shards()` takes the complementary approach: it *owns* and clears a fixed dir on each run.

Value

The seed- and layout-keyed path `file.path(root, paste0("seed=", <seed>), paste0("layout=", <hash>))`.

See Also

`ssd_run_scenario_shards()`, `ssd_summarise()`.

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 1L, seed = 42L)
scenario_results_dir(scenario)
```

ssd_analyse_cost

Analyse a Run's Observed Compute Cost

Description

Reads the per-task `.start/.end/.host` timings a completed run left in its `fit/hc` shard Parquets (the cost-analysis instrumentation), and attributes the **observed** compute to the scenario's `ci_method` x `nboot` axes - the measured counterpart to `ssd_estimate_cost()`'s prediction. It is strictly read-only: it reads result Parquets (and, optionally, a targets meta store), and never runs the pipeline, fits a distribution, draws random numbers, or writes a file. The observed **total** is serial-equivalent compute (the sum of per-task durations), distinct from elapsed wall time under parallel workers.

Usage

```
ssd_analyse_cost(scenario, root = scenario_results_dir(scenario), store = NULL)
```

Arguments

<code>scenario</code>	An <code>ssdsims_scenario</code> from <code>ssd_define_scenario()</code> .
<code>root</code>	The run's results root (the <code>fit/hc</code> shard trees live under it). Defaults to <code>scenario_results_dir()</code> for <code>scenario</code> .
<code>store</code>	Optional path to the run's targets data store, to add the per-shard envelope overhead. <code>NULL</code> (the default) reads the shards only.

Details

Given a targets store, it additionally reads each shard target's wall seconds from `targets::tar_meta()` and reports the per-shard **envelope overhead** (target seconds - sum(task durations): parent read, Parquet write, and dispatch), the number that informs `partition_by` tuning. The combined summary and `upload_<step>` targets are excluded; errored/unbuilt (NA-seconds) targets are dropped from totals; unmatched stored targets are reported, never silently dropped.

Value

An `ssdsims_cost_analysis`: a list with total and longest (both `difftime`), a breakdown tibble grouped by `ci_method` x `nboot`, `fit_seconds` (the measured fit addend), the hosts seen, a measured flag, the per-shard envelope (when a store is given), and provenance.

See Also

`ssd_estimate_cost()`, `ssd_compare_cost()`, `ssd_calibrate_cost_from_run()`.

ssd_calibrate_cost *Calibrate the Per-task Cost Model on the Current Machine*

Description

Runs a small, fixed benchmark sweep on the *current* architecture - tiny nboot values, all ssdtools::ssd_ci_methods(), and a couple of nrow values - times each ssdtools::ssd_hc() call, fits the per-ci_method cost model $\text{time} = (\text{base} + \text{slope} * \max(\text{nboot}, n0)) * \text{nrow_factor}(\text{nrow})$, and returns a versioned ssdsims_cost_calibration object carrying the fitted coefficients and provenance. Producing an architecture-specific estimator is nothing more than this single call plus passing the result to [ssd_estimate_cost\(\)](#).

Usage

```
ssd_calibrate_cost(
  nboot = c(20L, 50L, 100L, 200L),
  nrow = c(5L, 10L, 20L, 50L),
  data = NULL,
  seed = 42L
)
```

Arguments

nboot	An integer vector of tiny bootstrap sizes to sweep (the slope and floor are fit over these).
nrow	An integer vector of at least two sample sizes to fit one model per (the nrow_factor is derived from these).
data	A reference data frame with a numeric Conc column to resample (default ssddata::ccme_boron).
seed	A whole number seeding the resampling so the sweep is reproducible.

Details

The sweep is self-contained and dependency-light: it draws data by resampling a reference dataset (ssddata::ccme_boron by default) and times with [base::system.time\(\)](#). It takes minutes - not the hours a real scenario costs - because the nboot values are tiny (the slope and floor are estimable from small bootstraps). The one-time research that discovered the model's *form* (which axes are free, the $\max(\text{nboot}, n0)$ shape, the non-monotonic nrow factor) is illustrative and is not rerun here.

Value

An ssdsims_cost_calibration object with per-ci_method base/slope/n0 coefficients, a bounded nrow_factor, a fixed_addend, and provenance (cpu, R version, ssdtools version, date, sweep grid).

See Also

[ssd_cost_calibration\(\)](#) for the shipped default and [ssd_estimate_cost\(\)](#) to apply the result.

Examples

```
## Not run:
calibration <- ssd_calibrate_cost()
calibration

## End(Not run)
```

ssd_calibrate_cost_from_run

Recalibrate the Cost Model from an Observed Run

Description

Re-fits the per-task cost model from a run's **measured** hc task durations (the cost-analysis timings), returning an `ssdsims_cost_calibration` of the same shape [ssd_calibrate_cost\(\)](#) produces - so it drops straight into [ssd_estimate_cost\(\)](#) - but derived from real measurements rather than the synthetic micro-benchmark. The fixed addend comes from the measured fit task durations. Read-only: no pipeline, no RNG, no writes.

Usage

```
ssd_calibrate_cost_from_run(
  scenario,
  root = scenario_results_dir(scenario),
  host = NULL
)
```

Arguments

<code>scenario</code>	An <code>ssdsims_scenario</code> from ssd_define_scenario() .
<code>root</code>	The run's results root (the fit/hc shard trees live under it). Defaults to scenario_results_dir() for <code>scenario</code> .
<code>host</code>	Optional CPU description (a <code>.host</code> value) to select when the run spans more than one host. NULL (the default) requires a single-host run.

Details

Because the calibration is architecture-specific, timings from different `.host` values are never silently pooled: a run spanning more than one host requires an explicit `host`, or the function aborts listing the hosts found.

Value

An `ssdsims_cost_calibration` with run-derived provenance.

See Also

[ssd_calibrate_cost\(\)](#), [ssd_analyse_cost\(\)](#).

ssd_compare_cost

Compare Predicted Against Observed Compute Cost

Description

Places the [ssd_estimate_cost\(\)](#) prediction beside the [ssd_analyse_cost\(\)](#) observation for one scenario+run and reports the predicted and observed total compute, the predicted and observed longest task, and the predicted/observed ratio for each. Read-only.

Usage

```
ssd_compare_cost(
  scenario,
  root = scenario_results_dir(scenario),
  store = NULL,
  calibration = ssd_cost_calibration()
)
```

Arguments

scenario	An ssdsims_scenario from ssd_define_scenario() .
root	The run's results root (the fit/hc shard trees live under it). Defaults to scenario_results_dir() for scenario.
store	Optional path to the run's targets data store, to add the per-shard envelope overhead. NULL (the default) reads the shards only.
calibration	An ssdsims_cost_calibration for the prediction; defaults to the shipped ssd_cost_calibration() .

Value

An ssdsims_cost_comparison.

See Also

[ssd_estimate_cost\(\)](#), [ssd_analyse_cost\(\)](#).

`ssd_cost_calibration` *Default Cost Calibration*

Description

Returns the `ssdsims_cost_calibration` object shipped with the package - the calibration fitted during development (see [ssd_cost_calibration_default](#)) and used by [ssd_estimate_cost\(\)](#) when no calibration is supplied. Because the coefficients are architecture-specific, an estimate built on this default is a *ballpark* sized for the machine in its provenance; rerun [ssd_calibrate_cost\(\)](#) on your own machine for a trustworthy estimate.

Usage

```
ssd_cost_calibration()
```

Value

The shipped `ssdsims_cost_calibration` object.

See Also

[ssd_calibrate_cost\(\)](#) to re-fit on a target machine and [ssd_estimate_cost\(\)](#) to apply a calibration to a scenario.

Examples

```
ssd_cost_calibration()
```

`ssd_cost_calibration_default`
 Default Cost Calibration Object

Description

The `ssdsims_cost_calibration` object shipped with the package and returned by [ssd_cost_calibration\(\)](#). It carries the `per-ci_method` cost-model coefficients (base, slope, n_0), the bounded `nrow_factor`, a `fixed_addend` (sample + fit per-task overhead), and the provenance of the machine it was fitted on. [ssd_estimate_cost\(\)](#) uses it when no calibration is supplied.

Usage

```
ssd_cost_calibration_default
```

Format

An `ssdsims_cost_calibration` object: a list with coefficients (a tibble of `ci_method`, `base`, `slope`, `n0`), `nrow_factor` (a tibble of `nrow`, `factor`), `fixed_addend` (a scalar), and `provenance` (cpu, R version, `ssdtools` version, date, sweep grid).

Details

Because the coefficients are architecture-specific, this default yields a *ballpark* estimate sized for the machine in its provenance. Re-fit on your own machine with `ssd_calibrate_cost()` for a trustworthy estimate. It was produced by `data-raw/cost_calibration.R` (which simply runs `ssd_calibrate_cost()`).

Source

Fitted by `ssd_calibrate_cost()` during package development (Intel Xeon @ 2.10 GHz, R 4.5.3, `ssdtools` 2.6.0.9002).

See Also

`ssd_cost_calibration()`, `ssd_calibrate_cost()`, `ssd_estimate_cost()`.

<code>ssd_define_scenario</code>	<i>Define a Simulation Scenario</i>
----------------------------------	-------------------------------------

Description

Constructs a purely declarative `ssdsims_scenario` object: the root of the targets-based pipeline (see `TARGETS-DESIGN.md` section 1). The object stores only declarative fields - a scalar seed, the replicate count `nsim`, the sample sizes `nrow`, the dataset *names*, and the `fit` and `hc` argument grids. It performs **no** random-number generation, **no** task expansion, and has **no** dependency on targets.

Usage

```
ssd_define_scenario(
  data,
  nsim,
  seed,
  ...,
  nrow = 6L,
  replace = TRUE,
  rescale = FALSE,
  computable = FALSE,
  at_boundary_ok = TRUE,
  min_pmix = ssd_pmix(ssd_min_pmix = ssdtools::ssd_min_pmix),
  range_shape1 = list(c(0.05, 20)),
  range_shape2 = list(c(0.05, 20)),
  nrow_max = 1000L,
```

```

dists = ssd_distset(BCANZ = ssdtools::ssd_dists_bcanz()),
est_method = "multi",
proportion = 0.05,
ci = FALSE,
nboot = 1000,
ci_method = "weighted_samples",
parametric = TRUE,
samples = FALSE,
partition_by = NULL,
bundle = NULL
)

```

Arguments

data	An <code>ssd_scenario_data()</code> collection: a validated, named collection of Conc tibbles assembled from data frames and/or <code>ssd_gen()</code> generator datasets.
nsim	A count of the number of data sets to generate.
seed	A scalar whole number; the scenario's RNG root. Required - changing it fully re-roots the scenario's random-number draws.
...	Unused; must be empty.
nrow	A whole-number vector of sample sizes (the fit-step truncation axis), each between 5 (the fit floor) and <code>nrow_max</code> (the shared draw size, the universal ceiling). A value within <code>nrow_max</code> but above a dataset's row count is still valid: its <code>replace = TRUE</code> cell draws with replacement, while its <code>replace = FALSE</code> cell (which cannot exceed the dataset size) is silently discarded for that dataset.
replace	A logical vector (a cross-join axis of one or two values) specifying whether the shared sample draw resamples with replacement. Defaults to <code>TRUE</code> (the standard resampling model, drawing <code>nrow_max</code> rows, so <code>nrow</code> is not capped by the dataset size); <code>FALSE</code> draws a permutation, capping the effective draw - and so each <code>nrow</code> - at the dataset size.
rescale	A flag specifying whether to leave the values unchanged (<code>FALSE</code>) or to rescale concentration values by dividing by the geometric mean of the minimum and maximum positive finite values (<code>TRUE</code>) or a string specifying whether to leave the values unchanged ("no") or to rescale concentration values by dividing by the geometric mean of the minimum and maximum positive finite values ("geomean") or to logistically transform ("odds").
computable	A flag specifying whether to only return fits with numerically computable standard errors.
at_boundary_ok	A flag specifying whether a model with one or more parameters at the boundary should be considered to have converged (default = <code>TRUE</code>).
min_pmix	An <code>ssd_pmix()</code> collection of one or more single-argument <code>min_pmix</code> functions, referenced by name . This is the only accepted form; a bare function, a plain list, or a character vector of names is rejected (no string-to-function resolution). The collection's name (not the function value) is what the task path hashes; the function is materialised on the scenario (keyed by name) for execution and isolated via <code>scenario_min_pmix()</code> . Defaults to <code>ssd_pmix(ssd_min_pmix = ssdtools::ssd_min_pmix)</code> .

range_shape1	A list of numeric vectors of length two of the lower and upper bounds for the shape1 parameter.
range_shape2	A list of numeric vectors of length two of the lower and upper bounds for the shape2 parameter.
nrow_max	A whole number (default 1000L): the fixed size of the shared sample draw that every nrow value sub-truncates. A sample-level scenario setting, not a cross-join axis. The effective per-dataset draw is $\min(\text{nrow_max}, \text{nrow}(\text{data}))$ when <code>replace = FALSE</code> (the high default draws the full permutation) and <code>nrow_max</code> rows when <code>replace = TRUE</code> ; each <code>nrow</code> must not exceed the effective draw size.
dists	An <code>ssd_distset()</code> collection of one or more named distribution sets (pools model-averaged together to form one SSD). The fit step fits the union of every set's members once; the <code>hc</code> step subsets that union fit per set and re-averages, so the set name is an <code>hc</code> cross-join axis (" <code>distset</code> ") while individual distributions never fan out. A bare character vector or plain list aborts, pointing to <code>ssd_distset()</code> .
est_method	A string specifying whether to estimate directly from the model-averaged cumulative distribution function (<code>est_method = 'multi'</code>) or to take the arithmetic mean of the estimates from the individual cumulative distribution functions weighted by the AICc derived weights (<code>est_method = 'arithmetic'</code>) or to use the geometric mean instead (<code>est_method = 'geometric'</code>).
proportion	A numeric vector of proportion values to estimate hazard concentrations for.
ci	A flag specifying whether to estimate confidence intervals (by bootstrapping).
nboot	A count of the number of bootstrap samples to use to estimate the confidence limits. A value of 10,000 is recommended for official guidelines.
ci_method	A string specifying which method to use for estimating the standard error and confidence limits from the bootstrap samples. The default and recommended value is still <code>ci_method = "weighted_samples"</code> which takes bootstrap samples from each distribution proportional to its AICc based weights and calculates the confidence limits (and SE) from this single set. <code>ci_method = "multi_fixed"</code> and <code>ci_method = "multi_free"</code> generate the bootstrap samples using the model-averaged cumulative distribution function but differ in whether the model weights are fixed at the values for the original dataset or re-estimated for each bootstrap sample dataset. The value <code>ci_method = "MACL"</code> (was <code>ci_method = "weighted_arithmetic"</code>), which is only included for historical reasons, takes the weighted arithmetic mean of the confidence limits while <code>ci_method = "GMACL"</code> which takes the weighted geometric mean of the confidence limits was added for completeness but is also not recommended. Finally <code>ci_method = "arithmetic_samples"</code> and <code>ci_method = "geometric_samples"</code> take the weighted arithmetic or geometric mean of the values for each bootstrap iteration across all the distributions and then calculate the confidence limits (and SE) from the single set of samples.
parametric	A flag specifying whether to perform parametric bootstrapping as opposed to non-parametrically resampling the original data with replacement.
samples	A logical scalar (default FALSE): retain the bootstrap draws in the <code>hc</code> result's <code>samples</code> list-column (passed to <code>ssdtools::ssd_hc()</code>). This is output retention only - it does not change the estimates or the per-task RNG, so it is not a grid or task axis (a single TRUE is a superset of FALSE). Changing it re-runs the

	hc step (the discarded draws must be re-bootstrapped) but yields byte-identical estimates; retained samples can be large (nboot draws per dist per task), so it is off by default.
partition_by	An optional, possibly-partial named list keyed by step (sample/fit/hc) of character vectors naming the Hive path axes for that step (one shard per path cell; the inner complement rides as Parquet columns). Each entry must be unique, non-missing, and a subset of that step's axis vocabulary: sample = dataset, sim, replace; fit adds nrow, rescale, computable, at_boundary_ok, min_pmix, range_shape1, range_shape2; hc adds nboot, ci_method, parametric, distset (ci and est_method are hc scenario settings, not axes; dists is the fit-level scenario setting feeding the union, and distset - the set <i>name</i> - is the hc axis over the union's post-fit subsets). "nrow" is rejected only for sample (the shared draw carries no nrow axis; the fit step truncates it inline), and is a valid path axis for fit/hc. Steps partition independently - there is no cross-step constraint; a step may be finer or coarser than its neighbour on a shared axis (the m:n parent-shard relationship is resolved at the read layer). Steps left unnamed take their documented defaults (sample = c("dataset", "sim", "replace"), fit = c("dataset", "sim", "nrow", "rescale"), hc = c("dataset", "sim"); these supersede TARGETS-DESIGN.md section 5's pre-fold table). The split is orthogonal to the per-task RNG primer, so changing it shifts file paths only, never results.
bundle	An optional, possibly-partial named list keyed by step, the per-step complement of partition_by: it names the inner axes to keep together within a shard, and the stored path axes become setdiff(task_axes(step), bundle[[step]]). partition_by and bundle are complementary per-step entry points - at most one may name a given step (a step in both is an error), but they may be mixed across steps and either may be partial. Use partition_by when you want few path axes, bundle when you want fine sharding and only a few inner axes. Both normalise into the single stored partition_by path list.

Details

Input data arrives as an `ssd_scenario_data()` collection (already validated: a numeric Conc column is required) and is retained on the scenario (as `$data`) so a local run (`ssd_run_scenario_baseline()`) can sample it directly. The dataset *names* (`$datasets`) are what the targets/cluster path hashes; the validated tibbles ride on the scenario and are isolated by name via `scenario_dataset()`, so the hash need not carry the data frames.

Value

An S3 object of class `ssdsims_scenario`.

Dataset input

Dataset input is accepted **only** as an `ssd_scenario_data()` collection, which owns validation and naming. Assemble it first, then pass it in:

```
data <- ssd_scenario_data(boron = ccme_boron, cadmium = ccme_cadmium)
scenario <- ssd_define_scenario(data, ...)
```

Generator inputs (a `fitdists/tmbfit` object, a generator function, or a function-name string) are materialised - once, reproducibly - by `ssd_gen()` and composed into the same collection; the constructor itself performs **no** random-number generation.

nrow_max

`nrow_max` is the *sample-level scenario setting*: the fixed size of the shared sample draw that every `nrow` value sub-truncates (`head(sample, nrow)`, TARGETS-DESIGN.md section 5). The effective per-dataset draw is `min(nrow_max, nrow(data))` for `replace = FALSE` (the high default thus draws the full permutation) and `nrow_max` rows for `replace = TRUE`. Because the draw size is fixed - not derived from `max(nrow)` - adding `nrow` values (within the effective draw size) never changes the draw, so cached sample shards stay valid. Each `nrow` is validated at construction against the effective draw size. It is not *ci-gated* (the draw happens regardless of *ci*) and, like `dists` and `est_method`, it is absent from `task_axes("sample")`, so it never multiplies tasks or enters the per-task RNG primer.

ci

`ci` is a scalar flag (not a cross-join axis): the point estimate `est` is invariant to `ci` - it is computed analytically from the fit, independent of the bootstrap and RNG - so a single `ci = TRUE` run is a strict superset of `ci = FALSE` (same `est`, plus the `se/lcl/uc1` columns). The choice is scenario-wide either/or: `ci = FALSE` for cheap, bootstrap-free point estimates, or `ci = TRUE` for estimates plus confidence intervals. When `ci = FALSE`, the bootstrap-only scenario options `nboot`, `ci_method`, and `parametric` are meaningless; passing any of them in that case is an error, so set `ci = TRUE` to enable bootstrap, or omit the options.

dists and est_method

`dists` is an `ssd_distset()` collection: one or more **distribution sets** (pools of distributions model-averaged together to form one SSD), each named. The *fit* step fits the **union** of every set's members once - the single model-averaged superset every pool is a subset of - so `scenario$fit$dists` is that union and *individual* distributions still never fan out (an axis value is always a whole pool). The named sets ride on the *hc* grid (`scenario$hc$distsets`); the *hc* step subsets that one union fit down to each set's members (`subset(fit, set, strict = FALSE)`) and re-averages, so "distset" is an *hc* cross-join axis (`task_axes("hc")`) keyed by the set **name** - several pools reuse one fit rather than re-fitting. A bare character vector or plain list aborts loudly, pointing to `ssd_distset()`.

`est_method` is a **scenario setting**, not a cross-join axis - it is absent from `task_axes("hc")`, so it never multiplies tasks or enters the per-task RNG primer. It is an *hc*-level setting: every requested method is summarised from each *hc* task's **single** bootstrap sample set rather than re-bootstrapping per method (the CI is `est_method`-invariant and the point `est` is analytical), so a vector `est_method` yields one row per method within a task without fanning out into separate tasks.

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 100L, seed = 42L, nrow = c(5L, 10L))
scenario
```

Description

Collects one or more `ssd_define_scenario()` scenarios into a validated, named collection - the **design** (design-of-experiments sense: the set of conditions to run), the union of regular per-scenario grids into one possibly-non-regular design. The design is turned into a single targets pipeline by `ssd_design_targets()`.

Usage

```
ssd_design(...)
```

Arguments

... One or more `ssdsims_scenario` objects from `ssd_define_scenario()`, optionally named.

Details

Names are taken from the argument names where supplied, otherwise derived from the argument expression by symbol capture (e.g. a variable `base` becomes `"base"`), mirroring `ssd_scenario_data()`. Each name is a **scenario name** within the design, used only as the scenario identity column in the combined summary and the per-scenario summary target-name suffix - never in a shard path, a shard target name, the per-task primer, or any result value. Names must be unique, non-empty, and safe to serve as a target-name suffix (start with a letter; letters, digits, and underscore only). A design of **one** scenario is valid and uniformly shaped - the recommended starting point for a study that may grow (see the *migration* vignette).

Value

An `ssdsims_design` object: a named list of `ssdsims_scenario` objects.

Consistency contract

Because `ssd_design_targets()` addresses shards by cell and shares a cell across members (computing it once), the same **name** must denote the same value across members, or two members could disagree on a shared cell's bytes. `ssd_design()` therefore validates across members that the same dataset name binds identical data, the same `min_pmix` name binds an identical function, the same distset name binds identical members, and that `partition_by` is identical - aborting with an informative error otherwise. The seed may vary across members (it becomes a `seed=` results level); members sharing a seed share their coincident cells (common random numbers).

See Also

`ssd_define_scenario()`, `ssd_design_targets()`.

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
coarse <- ssd_define_scenario(data, nsim = 2L, seed = 42L, nrow = c(5L, 10L))
dense <- ssd_define_scenario(data, nsim = 2L, seed = 42L, nrow = c(6L, 7L, 8L))
ssd_design(coarse, dense)
```

ssd_design_targets	<i>Build the Targets Pipeline for a Design</i>
--------------------	--

Description

A **target factory** (the multi-scenario sibling of `ssd_scenario_targets()`): returns the list of targets objects that runs a `ssd_design()` - a named collection of scenarios - as one static-branching, Hive-sharded pipeline, so a whole `_targets.R` reduces to *build a design and call this*.

Usage

```
ssd_design_targets(design, ..., root = "results", upload = NULL, cue = NULL)
```

Arguments

design	An ssdsims_design from <code>ssd_design()</code> .
...	Unused; must be empty (forces root/upload/cue to be named).
root	The results root the shards and summaries are written under.
upload	An optional upload destination from <code>ssd_upload_azure()</code> or <code>ssd_upload_dryrun()</code> , or NULL (default) for no upload targets.
cue	An optional <code>targets::tar_cue()</code> applied to every shard target.

Details

A design is the **de-duplicated union** of its members' regular per-step task sets - the irregular (ragged) grid. Members are grouped by seed; within a group the union shard tables are computed (one target per cell, a cell shared by several members built **once**) and written under a legible `<root>/seed=<value>/layout=<hash>` tree, with the seed woven into the target names so cells never collide across seed groups. Each member then gets a `summary_<name>` target that filters the shared shards to its own task identities, and the top-level summary target unions those into `<root>/summary.parquet` with a scenario identity column (`ssd_summarise_design()`).

Value

A list of targets target objects, for `_targets.R` to return.

Migration from a single scenario

Growing a one-off `ssd_scenario_targets()` run into a study is a one-line switch: wrap the scenario with `ssd_design()` and call `ssd_design_targets()`. It is **cache-preserving** - a design of one addresses its shards identically to the standalone run (same seed=/layout= root via `scenario_results_dir()` and the same seed-woven target names), so re-running into the same root **reuses every existing shard** (no recompute); only the per-member and combined summary targets are new. Later members are added by extending the `ssd_design(...)` call; the cells they share (within a seed) stay cached.

Varying the seed

Members may use different seeds (e.g. repeating the exploration under several master seeds); they land under separate seed= trees and share nothing. Members sharing a seed share their coincident cells (common random numbers).

Per-overlap hc readout aggregation

Members of a seed group MAY differ in the four **non-axis** hc readout settings (proportion, `est_method`, `ci`, `samples`) and in their fit dists union; only the layout-shaping `nrow_max` and `partition_by` stay uniform-required. Differing readouts are reconciled **per shared hc cell, over only the members whose task set contains that cell** - proportion/`est_method` are union-ed and `ci`/`samples` reduced with `any()` - so the cell computes the maximal readout set in one shard and each member's summary filters its slice. A cell one member reaches keeps that member's (smaller) demand, so the expensive bootstrap runs only where a `ci = TRUE` member has tasks. The draw-shaping hc axes (`nboot`/`ci_method`/`parametric`/`distset`) are **not** aggregated - they stay cell axes (in the per-task primer), so byte-identity holds: a member's per-task results equal its standalone-run results.

Because a `ci = FALSE` task collapses `nboot`/`ci_method`/`parametric` to NA, its cell never coincides with a `ci = TRUE` task's. The point est is analytical and bootstrap-config-invariant, so a `ci = FALSE` cell is **served by a coincident `ci = TRUE` shard** at the same (fit, `distset`) when one exists (the computed hc shards are every `ci = TRUE` cell plus the `ci = FALSE` cells with no overlapping `ci = TRUE` shard); a `ci = TRUE` member's confidence interval still uses its own cell's (`nboot`, `ci_method`, `parametric`) primer. Differing fit dists unions are reconciled by fitting the **design-wide union** once per fit cell, each member subsetting via its `distset` axis (`distset-subset-invariance`), so members differing only in `distset` coverage share every sample/fit shard.

See Also

`ssd_design()`, `ssd_scenario_targets()`, `ssd_summarise_design()`.

Examples

```
## Not run:
# _targets.R
library(targets)
library(tarchetypes)
library(ssdsims)
data <- ssd_scenario_data(ssddata::ccme_boron)
coarse <- ssd_define_scenario(data, nsim = 2L, seed = 42L, nrow = c(5L, 10L))
```

```
dense <- ssd_define_scenario(data, nsim = 2L, seed = 42L, nrow = c(6L, 7L, 8L))
design <- ssd_design(coarse, dense)
ssd_design_targets(design)

## End(Not run)
```

ssd_distset

Assemble One or More Distribution Sets

Description

Constructs a validated, named ssdsims_distset collection - the single entry point ssd_define_scenario(dists = ...) accepts. A distribution **set** is the pool of distributions model-averaged together to form one SSD (one est); a **collection** is a named list of such sets, each supplied as a ... argument whose name labels the set.

Usage

```
ssd_distset(...)
```

Arguments

... One or more named character vectors, each a distribution set (its name labels the pool, its values are the distribution names averaged together).

Details

The fit step fits the **union** of every set's members once (the single model-averaged superset every pool is a subset of); the hc step then subsets that one union fit down to each set's members and re-averages, so several pools share one fit rather than re-fitting (distset is an hc-level axis - see [ssd_scenario_hc_tasks\(\)](#)). The set **name** is what hashes onto the hc task path (mirroring the by-name treatment of min_pmix and datasets); the member vectors are carried for execution only and never enter a task hash.

Set names are taken from the ... argument names and must be unique, non-missing, and filesystem-safe (each becomes a distset=<name> Hive path segment). Each set must be a non-empty, unique, non-NA character vector whose members are a subset of [ssdtools::ssd_dists_all\(\)](#).

Value

An ssdsims_distset object: a validated, named list of distribution-name character vectors.

See Also

[ssd_define_scenario\(\)](#), [scenario_distset\(\)](#).

Examples

```
ssd_distset(BCANZ = ssdtools::ssd_dists_bcanz())
ssd_distset(
  BCANZ = ssdtools::ssd_dists_bcanz(),
  Iwasaki = c("burrIII3", "gamma", "llogis", "lnorm", "weibull"),
  lnorm = "lnorm"
)
```

ssd_estimate_cost	<i>Estimate a Scenario's Compute Cost and Longest Task</i>
-------------------	--

Description

Predicts, *before* a scenario is launched, roughly how much compute it costs and how long its single longest task runs. It expands the scenario into its hc task table read-only (via [ssd_scenario_hc_tasks\(\)](#), without running any fit or bootstrap), applies the calibrated per-task cost model, and returns the ballpark serial **total** cost and the **duration of the longest single task**, plus a per-axis breakdown of which ci_method/nboot cells dominate.

Usage

```
ssd_estimate_cost(scenario, calibration = ssd_cost_calibration())
```

Arguments

scenario	An ssdsims_scenario from ssd_define_scenario() .
calibration	An ssdsims_cost_calibration; defaults to the shipped ssd_cost_calibration() . Pass the result of ssd_calibrate_cost() for an architecture-specific estimate.

Details

proportion and est_method are *free* axes: one bootstrap per nboot x ci_method x parametric cell serves every proportion/est_method, so adding values along those axes does not change the estimate. The estimator does **not** execute the scenario, draw random numbers, or alter any result.

The longest task is the irreducible wall-time floor under any amount of parallelism; wall-time under n workers is roughly max(longest_task, total / n), computed by the caller.

Value

An ssdsims_cost_estimate object: a list with total and longest (both difftime time quantities), a breakdown tibble grouped by ci_method x nboot, and the calibration's provenance.

See Also

[ssd_calibrate_cost\(\)](#) and [ssd_cost_calibration\(\)](#).

Examples

```
data <- ssd_scenario_data(ssdtools::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 10L,
  seed = 42L,
  replace = TRUE,
  nrow = c(5L, 10L, 20L, 50L),
  ci = TRUE,
  nboot = c(1000L, 5000L, 10000L, 50000L)
)
ssd_estimate_cost(scenario)
```

ssd_gen

Materialise Generator Datasets for a Simulation Scenario

Description

Accepts generator-style inputs and materialises each, once, to a validated tibble with a numeric Conc column of `.n` rows. The four generator kinds are:

Usage

```
ssd_gen(..., .n, .seed)
```

Arguments

- | | |
|--------------------|--|
| <code>...</code> | One or more generator inputs (a function, a function-name string, a <code>fitdists</code> object, or a <code>tmbfit</code> object), optionally named. |
| <code>.n</code> | A scalar whole number: the number of rows each generator materialises. Required. (This is the generated population size; the scenario's <code>nrow</code> controls the resample/truncation sizes.) |
| <code>.seed</code> | A scalar whole number: the base seed for generation. Required, and independent of the scenario's seed. |

Details

- a **function** taking the number of rows as its first argument (e.g. `ssdtools::ssd_rlnorm`);
- a **function-name string** (e.g. `"ssd_rlnorm"`), resolved as a bare name in the caller's environment and then the `ssdtools` namespace (never via parsing code); the string is also the dataset name;
- a `tmbfit` object (one distribution of a `ssdtools::ssd_fit_dists()` fit): drawn from the matching `ssd_r<dist>` function with the fit's estimates;
- a `fitdists` object: the top-weighted distribution is selected and drawn as for a `tmbfit`.

A data frame is **not** a generator - pass it directly to `ssd_scenario_data()`.

The result is an `ssdsims_gen` collection designed to compose with `ssd_scenario_data()` in two equivalent ways: as an unnamed argument (flattened in) or spliced with `!!!`:

```
ssd_scenario_data(boron = ccme_boron, ssd_gen(synth = ssd_rlnorm, .n = 30, .seed = 1L))
ssd_scenario_data(boron = ccme_boron, !!!ssd_gen(synth = ssd_rlnorm, .n = 30, .seed = 1L))
```

Value

An `ssdsims_gen` object: a named list of validated Conc tibbles of `.n` rows, for use within (or splicing into) `ssd_scenario_data()`.

Reproducibility: `.seed` and the name-keyed streams

`.n` and `.seed` are **required**: irreproducible or unsized generation is impossible by construction. They are dot-prefixed formals, so they are never absorbed into `...` and a generator named `seed =` or `n =` is never partial-matched onto them. Each generator draws under a scoped dqrng state seeded by `.seed` with its dataset **name** as the dqrng stream (`task_primer()` over `list(dataset = name)`), so a single `.seed` fans out across all generators in the call on independent, name-keyed streams. One `.n` applies per call; for differing sizes, splice several `ssd_gen()` calls together. The global `.Random.seed` is unchanged on return.

Names are taken from the argument names where supplied, otherwise the function-name string itself, otherwise derived from the argument expression by symbol capture (e.g. `ssdtools::ssd_rlnorm` becomes `"ssd_rlnorm"`); an input with no derivable name (e.g. an anonymous function literal) must be given an explicit name. Names must be unique across the call.

dqrng-backed generation

Generation draws under the dqrng pcg64 backend (`local_dqrng_backend()`), with each generator seeded through `local_dqrng_state()`, which brackets the draw with the per-task dqrng-integrity witness: it aborts if a generator escapes the backend (e.g. switches `RNGkind()` and draws from base R), since such draws are not reproducible under `.seed`. A generator that consumes no randomness passes.

See Also

`ssd_scenario_data()`, `ssd_define_scenario()`, `local_dqrng_backend()`, `task_primer()`, and the "Generating and Assembling Datasets" vignette (`vignette("generating-data", package = "ssdsims")`).

Examples

```
ssd_gen(synth = ssdtools::ssd_rlnorm, .n = 30, .seed = 42)

# one .seed fans out across generators on independent name-keyed streams
ssd_gen(a = ssdtools::ssd_rlnorm, b = ssdtools::ssd_rlnorm, .n = 30, .seed = 42)

# composes with data frames in ssd_scenario_data()
ssd_scenario_data(
  boron = ssdtools::ccme_boron,
  ssd_gen(synth = ssdtools::ssd_rlnorm, .n = 30, .seed = 42)
)
```

ssd_open_uploaded

*Open Uploaded Results for Querying, In Place***Description**

A generic, dispatched on the upload object's class, that opens the **uploaded** results so a user can read them back and confirm they landed right after an upload (TARGETS-DESIGN.md section 6.1). For an Azure destination it returns a **lazy** duckplyr/DuckDB table over the Hive glob <container>[/<prefix>]/<step>/**/part (honouring the destination's optional prefix subdirectory) - or, for the combined summaries, the single blob summary.parquet (step = "summary") / summary-samples.parquet (step = "summary_samples", shipped only when the scenario set samples = TRUE) - read **in place** via DuckDB's azure extension (predicate pushdown straight against blob storage - **no download**), composable with dplyr verbs so a one-line `ssd_open_uploaded(upload, step) |> dplyr::count()` is the immediate post-upload smoke test. It resolves the **same** front-end SSDSIMS_AZURE_* credentials as the write path and remaps them into a DuckDB azure secret for the backend read, aborting (naming the missing requirement) when the azure extension or a required credential is absent. For a dry-run destination it aborts: a dry run uploads nothing, so the local shards should be read directly.

Usage

```
ssd_open_uploaded(upload, step, ..., prudence = "stingy")
```

Arguments

upload	An upload destination from <code>ssd_upload_azure()</code> or <code>ssd_upload_dryrun()</code> .
step	One of "sample", "fit", "hc" (the step layer to read), "summary" (the uploaded compact summary), or "summary_samples" (the uploaded full summary retaining the dists/samples list-columns, shipped only when the scenario set samples = TRUE).
...	These dots are for future extensions and must be empty.
prudence	The duckplyr prudence of the returned table (default "stingy"): "stingy" keeps it lazy and composable but makes an implicit materialisation (e.g. <code>nrow()/\$</code>) against the remote glob error rather than triggering an unbounded download/scan; "lavish" restores automatic materialisation on first access. <code>dplyr::collect()</code> and <code>duckplyr::compute_parquet()</code> work under either.

Value

A lazy, dplyr-composable table over the uploaded results.

See Also

`ssd_upload_shard()`, `ssd_test_upload()`.

Examples

```
## Not run:
upload <- ssd_upload_azure("https://acct.blob.core.windows.net", "results")
ssd_open_uploaded(upload, "hc") |> dplyr::count()

## End(Not run)
```

ssd_pmix

Assemble and Validate min_pmix Functions for a Simulation Scenario

Description

Collects one or more min_pmix functions into a validated, named collection - the single entry point through which `ssd_define_scenario()` takes min_pmix input. Each entry must be a single-argument **function** (it inputs the number of rows of data and returns a proportion between 0 and 0.5); a name-string is **not** accepted, so the constructor performs no string-to-function resolution.

Usage

```
ssd_pmix(...)
```

Arguments

... One or more single-argument functions, optionally named.

Details

Names are taken from the argument names where supplied, otherwise derived from the argument expression by symbol capture (e.g. `ssdtools::ssd_min_pmix` becomes "ssd_min_pmix"), mirroring `ssd_scenario_data()`. Names must be unique across the collection. The name - not the function value - is what the task path hashes; the functions ride on the scenario for execution, isolated by name via `scenario_min_pmix()`.

Value

An `ssdsims_pmix` object: a named list of validated single-argument functions.

See Also

`ssd_define_scenario()`, `scenario_min_pmix()`.

Examples

```
ssd_pmix(ssdtools::ssd_min_pmix)
ssd_pmix(ssd_min_pmix = ssdtools::ssd_min_pmix, strict = function(n) 0.1)
```

ssd_run_scenario_baseline

Run a Scenario with the Baseline Loop Runner

Description

Executes the three task tables in dependency order - sample, fit, then hc - by looping over each table with `purrr::pmap()` and looking up each task's parent result by the parent's `<step>_id` foreign key. The fit step truncates its parent sample inline (`head(sample, nrow)`) before fitting. The runner does no task expansion of its own (it consumes `ssd_scenario_tasks()`); it just threads outputs forward and returns the collected per-step results.

Usage

```
ssd_run_scenario_baseline(scenario)
```

Arguments

scenario An `ssdsims_scenario` from `ssd_define_scenario()`.

Details

This is the no-frills baseline: it runs in-process, with **no** targets dependency, **no** shard grouping or partition_by, and **no** Parquet I/O.

It **is** reproducible without an external seed. The runner opens one `local_dqrng_backend()` scope and seeds each sample/fit/hc task exactly once through its `*_data_task_primer()` wrapper, with `seed = scenario$seed` and a per-task primer derived from the task's canonical identity (`task_primer()` over the `task_axes(step)` columns). Because each task's (seed, primer) pair fully determines its RNG starting point, two runs of a scenario with a fixed seed yield identical results, and a task's result is independent of the order in which tasks run. These same `*_data_task_primer()` wrappers are the per-task entry point a future targets shard body and the replay helper (TARGETS-DESIGN.md §7) reuse.

The scenario retains the data frames it was built from, so the runner reads them directly - no separate data argument. `min_pmix` names are resolved to their materialised functions off the scenario via `scenario_min_pmix()` (resolved once, at construction), not by a runtime `ssdtools/global-env` search.

Value

A named list with `sample`, `fit`, and `hc` elements: each the corresponding task table augmented with a list column of per-task results (sample draws, fits objects, and hc tibbles).

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
```

```

    nsim = 1L,
    nrow = 6L,
    seed = 42L,
    dists = ssd_distset(lnorm = "lnorm")
  )
  out <- ssd_run_scenario_baseline(scenario)
  out$hc

```

ssd_run_scenario_shards

Run a Scenario over Hive-partitioned Parquet Shards (single core)

Description

Executes a scenario's three task steps in dependency order - sample, then fit, then hc - materialising each step's results as **one Parquet per partition_by path cell** under a Hive-partitioned tree `<dir>/<step>/<axis=value>/.../part.parquet`, and linking steps by reading the parent step's shards back via duckplyr (predicate pushdown), rather than threading results in memory. This is the single-core, targets-free sibling of `ssd_run_scenario_baseline()` and the first consumer of partition-by's path/inner split (`scenario_dataset()`'s sibling `scenario_partition_axes()`).

Usage

```
ssd_run_scenario_shards(scenario, dir = tempfile("ssdsims-shards-"))
```

Arguments

scenario	An ssdsims_scenario from <code>ssd_define_scenario()</code> .
dir	A results root to write the Hive-partitioned shards under; created if absent. Defaults to a per-run session temp dir (the shards are left on disk for inspection and reuse). The runner owns the sample/fit/hc subtrees under dir and clears them on each run, so replaying a scenario with a changed partition_by/bundle never leaves stale-granularity shards beside the new ones. (The targets pipeline instead isolates each layout under its own <code>scenario_results_dir()</code> root.)

Details

It reuses the per-task seed-and-run wrappers, so for a fixed scenario\$seed it is reproducible and **order-independent**, and its per-task results are **byte-identical** to `ssd_run_scenario_baseline()` - partition_by is a free re-layout that moves only file paths, never results. The **m:n** parent-shard dependency (a child shard reading several parent shards, or a parent shard feeding several children, per the section 5 coarsening defaults) is resolved at read time: each task opens the parent shard at its `<parent>_id` identity projected onto the parent's path axes and filters to the rows it needs.

No targets, crew, manifest, or cloud upload - this is the plain-R storage loop only, de-risking hive-partitioning/task-tables.

Value

An ssdsims_shard_run object: a list with dir and the written sample, fit, and hc shard Parquet paths (one per shard).

See Also

[ssd_run_scenario_baseline\(\)](#) (the in-memory reference oracle), [ssd_scenario_sample_shards\(\)](#), [ssd_run_sample_step\(\)](#).

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 1L,
  nrow = 6L,
  seed = 42L,
  dists = ssd_distset(lnorm = "lnorm")
)
run <- ssd_run_scenario_shards(scenario)
run$hc
```

ssd_run_step

Run a Step Shard

Description

The per-shard step runners target the pipeline (and the single-core [ssd_run_scenario_shards\(\)](#) call - one target per shard, one runner per step. Each takes a shard's tasks (the tasks list-column of a row of the matching [ssd_scenario_sample_shards\(\)](#) family), runs the bundled tasks with the *same* per-task seed-and-run primitives the baseline runner uses (`*_data_task_primer()`) under one [local_dqrng_backend\(\)](#) scope, reads any upstream shard back from Parquet by partition path, and writes one Parquet at the shard's Hive partition path - returning that path (the format = "file" contract). Because a task's result is fully determined by its (seed, primer) and is order-independent, the per-task results are byte-identical to [ssd_run_scenario_baseline\(\)](#) regardless of how tasks bundle into shards.

Usage

```
ssd_run_sample_step(tasks, scenario, out_dir)
```

```
ssd_run_fit_step(tasks, scenario, sample_dir, out_dir)
```

```
ssd_run_hc_step(tasks, scenario, fit_dir, out_dir)
```

Arguments

tasks	A tibble of the shard's task rows (the tasks list-column of a row of the matching <code>ssd_scenario_*_shards()</code>), each carrying the step's axis values, its <code><step>_id</code> key, seed, and primer - and, for fit/hc, the parent step's path-axis values and <code><parent>_id</code> .
scenario	The <code>ssdsims_scenario</code> (a referenced global in <code>_targets.R</code>).
out_dir	The step's results root (e.g. "results/sample").
sample_dir	The sample results root the parent shards were written to (the fit step).
fit_dir	The fit results root the parent shards were written to (the hc step).

Value

The shard's Parquet path (the `format = "file"` contract).

Functions

- `ssd_run_sample_step()`: Run the sample tasks: read each task's dataset off the scenario via [scenario_dataset\(\)](#), draw the effective draw size - the scenario's `nrow_max` setting, capped at the dataset size for `replace = FALSE` - through `sample_data_task_primer()`, and tag each draw with its `sample_id` and a `.row` order index so a downstream fit shard can isolate and re-order it.
- `ssd_run_fit_step()`: Run the fit tasks: read the distinct set of parent sample shards the shard's tasks reference (each once - they may span several sample shards), isolate each task's draw by `sample_id` (restoring row order), truncate it inline (`head(sample, nrow)`, RNG-free, section 5), and fit with the per-task (`seed`, `primer`) through `fit_data_task_primer()` (resolving `min_pmix` off the scenario via [scenario_min_pmix\(\)](#)). The fitted `fitdists` object is serialised into a `fit_blob` string column keyed by `fit_id`, and one Parquet is written at the shard's partition path.
- `ssd_run_hc_step()`: Run the hc tasks: read the distinct set of parent fit shards the shard's tasks reference (each once - an hc shard typically spans several fit shards), decode each parent **union** fit once per `fit_id` (reused across every distset task that shares it), resolve each task's distset name to its members via [scenario_distset\(\)](#), subset the union fit to that pool (`strict = FALSE`), and estimate the hazard concentration with the per-task (`seed`, `primer`) through `hc_data_task_primer()` (the subset happens in that shared primitive). Each task's hc tibble (with the scalar `ci` applied uniformly and bootstrap-only scenario options NA when `ci = FALSE`) is tagged with its `hc_id`, parent `fit_id`, and distset name, stacked, and written as one Parquet at the shard's partition path. A set whose members all dropped from the union fit emits no rows for that cell (the survivor model).

The four non-axis hc readout settings (`proportion`, `est_method`, `ci`, `samples`) default to the scenario slice, so the single-scenario and standalone paths are byte-identical. When the shard's tasks carry per-task `proportion/est_method/ci/samples` columns (the design factory's per-overlap aggregated demand, [ssd_design_targets\(\)](#)), each task is summarised with its own cell's demand instead - the maximal readout set the per-member summary then filters.

See Also

[ssd_scenario_sample_shards\(\)](#) (the shard grouping these consume), [ssd_run_scenario_shards\(\)](#), [ssd_run_scenario_baseline\(\)](#).

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 1L, seed = 42L)
shards <- ssd_scenario_sample_shards(scenario)
dir <- tempfile()
ssd_run_sample_step(shards$tasks[[1L]], scenario, file.path(dir, "sample"))
```

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 1L,
  nrow = 6L,
  seed = 42L,
  dists = ssd_distset(lnorm = "lnorm")
)
dir <- tempfile()
ssd_run_sample_step(
  ssd_scenario_sample_shards(scenario)$tasks[[1L]],
  scenario,
  file.path(dir, "sample")
)
ssd_run_fit_step(
  ssd_scenario_fit_shards(scenario)$tasks[[1L]],
  scenario,
  file.path(dir, "sample"),
  file.path(dir, "fit")
)
```

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 1L,
  nrow = 6L,
  seed = 42L,
  dists = ssd_distset(lnorm = "lnorm")
)
dir <- tempfile()
ssd_run_sample_step(
  ssd_scenario_sample_shards(scenario)$tasks[[1L]],
  scenario,
  file.path(dir, "sample")
)
ssd_run_fit_step(
  ssd_scenario_fit_shards(scenario)$tasks[[1L]],
  scenario,
  file.path(dir, "sample"),
  file.path(dir, "fit")
)
ssd_run_hc_step(
  ssd_scenario_hc_shards(scenario)$tasks[[1L]],
  scenario,
```

```

    file.path(dir, "fit"),
    file.path(dir, "hc")
  )

```

ssd_scenario_data

Assemble and Validate Datasets for a Simulation Scenario

Description

Collects one or more datasets into a validated, named collection - the single entry point through which `ssd_define_scenario()` takes dataset input. Each dataset must carry a numeric Conc column (the species sensitivity distribution convention); additional columns are preserved.

Usage

```
ssd_scenario_data(...)
```

Arguments

... One or more data frames, optionally named, and/or `ssd_gen()` collections (unnamed, or spliced with `!!!`). Each data frame is validated for a numeric Conc column.

Details

Names are taken from the argument names where supplied, otherwise derived from the argument expression by symbol capture (e.g. `ssddata::ccme_boron` becomes `"ccme_boron"`). A literal with no derivable name (e.g. a bare `data.frame(...)` call) must be given an explicit name. Names must be unique across the collection.

Generator-style inputs (a `fitdists` or `tmbfit` object, a generator function, or a function-name string) enter the collection through `ssd_gen()`, which materialises each, once, to a reproducible Conc tibble. Its result composes with the data-frame inputs in two equivalent ways: passed as an unnamed argument, the collection is flattened in (each materialised tibble becomes a member under its own name); or spliced with `!!!` (`rlang::list2()` splicing), with identical results:

```
ssd_scenario_data(boron = ccme_boron, ssd_gen(synth = ssd_rlnorm, .n = 30, .seed = 1L))
ssd_scenario_data(boron = ccme_boron, !!!ssd_gen(synth = ssd_rlnorm, .n = 30, .seed = 1L))
```

A materialised generator dataset is an ordinary tibble in the collection, indistinguishable downstream from a data-frame dataset.

Value

An `ssdsims_data` object: a named list of validated tibbles.

See Also

[ssd_gen\(\)](#), [ssd_define_scenario\(\)](#), and the "Generating and Assembling Datasets" vignette ([vignette\("generating-data", package = "ssdsims"\)](#)).

Examples

```
ssd_scenario_data(ssddata::ccme_boron)
ssd_scenario_data(boron = ssddata::ccme_boron, cadmium = ssddata::ccme_cadmium)
```

ssd_scenario_shards *Group Tasks into Shards*

Description

Group a step's per-task table into a per-shard table: one row per partition_by **path** cell, carrying the path-axis columns (the tar_map target-name suffix and Hive path) and a tasks list-column of that cell's task rows. Each task row is decorated with seed = scenario\$seed and its per-task primer ([task_primer\(\)](#) over the step's task_axes()); the decoration is RNG-free (a pure hash, not a draw), so the bare task tables ([ssd_scenario_tasks\(\)](#)) keep their no-(seed, primer) contract. The result is the values a `tarchetypes::tar_map()` consumes to mint one target per shard.

Usage

```
ssd_scenario_sample_shards(scenario)

ssd_scenario_fit_shards(scenario)

ssd_scenario_hc_shards(scenario)
```

Arguments

scenario An `ssdsims_scenario` from [ssd_define_scenario\(\)](#).

Details

For fit/hc each task row in tasks also carries its parent step's path-axis values and <parent>_id, so the runner opens the matching parent shard by partition path.

Value

A tibble with one row per shard of the step: the path-axis columns and a tasks list-column. Suitable as `tarchetypes::tar_map(values = )`.

Functions

- `ssd_scenario_sample_shards()`: Group the sample tasks (`ssd_scenario_sample_tasks()`) by `partition_by$sample`.
- `ssd_scenario_fit_shards()`: Group the fit tasks (`ssd_scenario_fit_tasks()`) by `partition_by$fit`. Each task row in `tasks` carries its parent sample path-axis values and `sample_id`, so the runner opens the matching sample shard by partition path.
- `ssd_scenario_hc_shards()`: Group the hc tasks (`ssd_scenario_hc_tasks()`) by `partition_by$hc`. Each task row in `tasks` carries its parent fit path-axis values and `fit_id`, so the runner opens the matching fit shard by partition path.

See Also

`ssd_run_sample_step()` (the matching per-shard step runners).

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 2L, seed = 42L)
ssd_scenario_sample_shards(scenario)
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 2L,
  seed = 42L,
  rescale = c(FALSE, TRUE)
)
ssd_scenario_fit_shards(scenario)
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 2L,
  seed = 42L,
  ci = TRUE
)
ssd_scenario_hc_shards(scenario)
```

ssd_scenario_targets *Build the Targets Pipeline for a Scenario*

Description

A target factory: returns the list of targets objects that runs a scenario as a static-branching Hive-sharded pipeline (TARGETS-DESIGN.md section 6), so a whole `_targets.R` reduces to *build a scenario and call this*:

Usage

```
ssd_scenario_targets(
  scenario,
  ...,
  root = "results",
  upload = NULL,
  cue = NULL
)
```

Arguments

scenario	An ssdsims_scenario from ssd_define_scenario() .
...	Unused; must be empty. Its presence forces root, upload, and cue to be passed by name (rlang::check_dots_empty() aborts on a positional or misspelled argument), since root and upload are both path-shaped and easy to transpose.
root	The base results directory (default "results"). The shards and summary are written under the seed-/layout-keyed scenario_results_dir() (scenario, root), so a single-scenario run and a design-of-one address shards identically (a cache-free upgrade to ssd_design_targets()).
upload	An optional upload destination (the remote-destination sibling of root) from ssd_upload_azure() or ssd_upload_dryrun() , or NULL (default) for no upload targets. See the section above.
cue	An optional targets::tar_cue() applied to every shard target (e.g. targets::tar_cue(depend = FALSE) to pin trusted shards against code changes). NULL (default) uses targets' standard cue.

Details

```
library(targets)
library(tarchetypes)
library(ssdsims)
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 2L, seed = 42L)
ssd_scenario_targets(scenario)
```

The shard and summary targets carry error = "null" so a shard whose body fails entirely goes NULL (its error readable via tar_meta()) without aborting the run, and ssd_summarise() unions whatever landed (TARGETS-DESIGN.md section 6.2). The shipped _targets.R templates pair this with a pipeline-wide **keep-going** default (tar_option_set(error = "continue"), the make -k analogue) so an errored target skips only its dependents while the rest of the shards still build; fail-fast pre-flight checks (upload/cluster connectivity) belong in a separate script the user runs *before* tar_make(), not in this DAG.

For each step it tarchetypes::tar_map()s one named, format = "file", error = "null" target per partition_by path cell (the names are the step's path axes), and writes every shard and the summary under the per-layout [scenario_results_dir\(\)](#) root (so a changed partition_by/bundle never mixes shard granularities). Each step's command depends only on the **minimal scenario slice** its runner consumes (scenario_step_slice()) rather than the bare scenario global, so editing a field a step does not read leaves the other steps' shards cached. The sample slice is built **per shard**,

carrying only the dataset(s) that shard reads, so appending a dataset mints a new shard and leaves every existing shard cached.

Value

A list of targets target objects, for `_targets.R` to return.

Invalidation model

The shard targets use **content-hash invalidation over their** `format = "file"` **Parquet outputs** (TARGETS-DESIGN.md section 8), observable as **cache-by-existence**: a shard is up to date iff its Parquet exists *and* the inputs its body depends on - its task rows, the step's minimal scenario slice (`scenario_step_slice()`), and the parent shard target(s) it reads - are unchanged. A missing Parquet rebuilds; a recomputed shard whose bytes are byte-identical leaves its dependents skipped.

Instead of a coarse `sample -> fit -> hc tar_combine()` barrier (which marks the *whole* downstream step out of date when any one parent shard changes), each child shard target names **only the specific parent shard target(s) its tasks read** (the Option-3 per-child upstream edges of section 6), computed at sourcing time as `unique(path_key(tasks, partition_by[[parent]]))` - the same projection the runner uses to read them. So rewriting one parent shard re-runs only the child shards that read it. `summary` reads the whole `hc` directory, so it names every `hc` shard (it re-runs when any `hc` shard's bytes change, and unions the survivors of a partially-failed run).

Pinning trusted shards (cue)

Pass `cue = targets::tar_cue(depend = FALSE)` to **pin** the shard targets against upstream dependency/code changes (an edited per-task primitive, a bumped `ssdtools`), so trusted shards are not rebuilt by a code edit (TARGETS-DESIGN.md section 8.3). The carve-outs still hold: a shard rebuilds if its `format = "file"` Parquet is missing, if its task-table grouping changes (the grouping is part of the command, so path-axis and inner-axis growth still apply under the pin), or if it previously errored. Force a refresh of chosen shards with `targets::tar_invalidate()` (or by deleting their Parquet), overriding the pin (section 8.4). The default (NULL) is targets' standard cue.

Volatile fit/hc file hashes (cost-analysis timings)

The `fit/hc` shards carry per-task `.start/.end/.host` timing columns (the cost-analysis instrumentation), so a `fit/hc` shard's **file hash is no longer deterministic across recomputes**: a forced recompute that yields identical *results* still writes different bytes (a fresh wall-clock), so its dependent `hc/summary` targets re-run and any paired `upload_<step>` re-ships. This is scoped to `fit/hc`; `sample` shards carry no timing columns and stay byte-deterministic. Routine caching is unaffected (a cache hit is not a recompute, so a cached shard's bytes are unchanged); the cost lands only on a *forced* refresh (`tar_invalidate()`, a deleted Parquet) or a code-edit recompute - and the §8.3 `cue = tar_cue(depend = FALSE)` pin covers the latter. Per-task *results* remain byte-identical to the baseline oracle (the shard-runner contract narrows to the result columns, timing excluded).

The `head(sample, nrow)` truncation stays folded into the `fit` step (no materialised data shard): a `fit` shard is keyed by `fit_id`, which includes `nrow`, so extending `nrow` mints new `fit` shards and caches the rest. The shared draw is sized by the scenario's fixed `nrow_max` setting (carried on the sample slice), not `max(nrow)`, so extending `nrow` within the effective draw size leaves the sample shards cached too; changing `nrow_max` invalidates the sample slice and rebuilds the draw, propagating through the per-child edges - no stale short draw can arise.

To parallelise the shards, set a controller (e.g. a mirai-backed crew: :crew_controller_local()) with `targets::tar_option_set()` in `_targets.R` before calling this - the target set is unchanged.

Uploading shards to cloud storage (upload)

`upload` is the **remote-destination sibling of** `root` (default `NULL`). With `upload = NULL` the pipeline contains **no** upload targets - the clean default DAG for a non-uploader. With a non-`NULL` upload object the factory pairs each step shard with an `upload_<step>` target in the same `tar_map` (format = "file", error = "null"), so an unchanged shard is never re-uploaded (content-hash skip) and a per-shard upload failure isolates to its own branch; it also pairs the summary fan-in with a single `upload_summary` target (same format = "file", error = "null" contract) that ships the combined summary Parquet - and, when the scenario sets `samples = TRUE`, the full `summary-samples.parquet` alongside it - with the same content-hash skip, so the summary re-ships only when its bytes change. Pass `ssd_upload_dryrun()` for no-op upload targets that reach no network (exercising the DAG shape offline / in CI) or `ssd_upload_azure()` to ship to Azure. The factory performs **no** network I/O and never runs the `ssd_test_upload()` probe: it only assembles the target list, so sourcing `_targets.R` (which `targets` does on every `tar_make()`, `tar_manifest()`, `tar_visnetwork()`, and on each worker) stays side-effect free. Run `ssd_test_upload(upload)` yourself as a one-line preflight before `tar_make()` to confirm credentials and connectivity up front; a missing credential still fails loud per-shard at upload time as a backstop. The per-task results are byte-identical across all three upload modes; only the presence and behaviour of the upload targets differ.

See Also

`scenario_results_dir()`, `ssd_run_scenario_shards()` (the single-core, targets-free equivalent), `ssd_upload_azure()`.

Examples

```
## Not run:
# _targets.R
library(targets)
library(tarchetypes)
library(ssdsims)
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 2L, seed = 42L)
ssd_scenario_targets(scenario)

# Pair each shard with a (no-op) upload target, exercised offline:
ssd_scenario_targets(scenario, upload = ssd_upload_dryrun())

## End(Not run)
```

Description

The canonical expansion entry point (TARGETS-DESIGN.md section 1/section 2): `ssd_scenario_tasks()` derives the sample, fit, and hc task tables from a scenario in one call and bundles them into an `ssdsims_task_set`. The per-step derivations (`ssd_scenario_sample_tasks()`, `ssd_scenario_fit_tasks()`, `ssd_scenario_hc_tasks()`) remain available for callers that need a single table; each is equivalent to `ssd_scenario_tasks(scenario, step)` for the matching step.

All derivations are RNG-free: they perform no random-number generation and add no seed/primer/stream columns (those arrive in later roadmap steps; see TARGETS-DESIGN.md section 2). Each row carries a path-style `<step>_id` primary key (the Hive partition path) and, for non-root steps, its parent step's `<parent>_id` as a joinable foreign key, so a child task references its parent by a single column.

Usage

```
ssd_scenario_tasks(scenario, step = NULL)
```

```
ssd_scenario_sample_tasks(scenario)
```

```
ssd_scenario_fit_tasks(scenario)
```

```
ssd_scenario_hc_tasks(scenario)
```

Arguments

<code>scenario</code>	An <code>ssdsims_scenario</code> from <code>ssd_define_scenario()</code> .
<code>step</code>	Optional single step name ("sample", "fit", or "hc"). When supplied, returns just that step's <code>ssdsims_tasks</code> table (the same as the matching <code>ssd_scenario*_tasks()</code>); when NULL (default) returns the full <code>ssdsims_task_set</code> .

Value

An `ssdsims_task_set` object (a list with sample, fit, and hc elements, each an `ssdsims_tasks` table), or - when step is supplied - the single `ssdsims_tasks` table for that step. Each `ssdsims_tasks` table is a classed tibble recording one step, with one row per cell of that step's cross-join.

Functions

- `ssd_scenario_sample_tasks()`: Derive just the sample task table: one row per cell of the cross-join of the scenario's dataset names, replicate index (1:nsim), and replace values, keyed by `sample_id`. Each row is the single random draw that every `nrow` value sub-truncates (TARGETS-DESIGN.md section 5), so `nrow` is **not** a sample axis - the draw is shared. The draw size is the scenario's `nrow_max` setting, resolved by the runner against each dataset, not a row column: the table carries only the task identity.
- `ssd_scenario_fit_tasks()`: Derive just the fit task table: cross each sample-task identity (dataset, sim, replace) with the scenario's `nrow` values and each row of the scenario's fit argument grid (rescale, computable, at_boundary_ok, min_pmix name, range_shape1, range_shape2). `nrow` is a genuine fit cross-join axis: the fit step truncates its parent sample inline (`head(sample, nrow)`, RNG-free) before fitting, so the shared draw is sub-truncated

without a separate data step (TARGETS-DESIGN.md section 5). `min_pmix` is referenced by name, not by function value (TARGETS-DESIGN.md section 1.1). Each row carries a `fit_id` primary key and a `sample_id` foreign key referencing its parent sample task.

- `ssd_scenario_hc_tasks()`: Derive just the hc task table: cross each fit-task identity with each row of the scenario's hc argument grid (`nboot`, `ci_method`, `parametric`) **and with the scenario's declared distribution sets** (`distset`, the `set_names`). The scenario's scalar `ci` flag and the `est_method` setting are applied uniformly to every hc row - neither is a cross-join axis nor an emitted column; the runners read `ci` from the scenario and every requested `est_method` is summarised within each task from its single bootstrap sample set. When `ci = FALSE` the bootstrap-only scenario options (`nboot`, `ci_method`, `parametric`) are canonically NA, leaving `distset` as the only fan-out, so the grid is exactly D hc rows per fit task (one per set); when `ci = TRUE` the grid fans out across `distset` x `nboot` x `ci_method` x `parametric`. A single-set collection yields one `distset` value (one hc row per fit task when `ci = FALSE`). Each row carries an `hc_id` primary key, its `distset` name, and a `fit_id` foreign key referencing its parent (union) fit task.

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(data, nsim = 3L, seed = 42L)
tasks <- ssd_scenario_tasks(scenario)
tasks
tasks$hc
ssd_scenario_tasks(scenario, "hc")
ssd_scenario_sample_tasks(scenario)
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 3L,
  seed = 42L,
  rescale = c(FALSE, TRUE)
)
ssd_scenario_fit_tasks(scenario)
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 2L,
  seed = 42L,
  ci = TRUE,
  nboot = c(10L, 100L)
)
ssd_scenario_hc_tasks(scenario)
```

Description

Fans in the run's results without pulling shard target values back into R or recomputing anything: reads every hc shard Parquet under `dir_hc` (a Hive glob) with duckplyr - the analysis-ready per-task hazard-concentration estimates - unions them, and writes path. Because it reads the result directory (not the shard targets), it sees whatever shards landed, so it unions the survivors of a partially-failed run (`error = "null"`, section 6.2). `dir_sample` and `dir_fit` are accepted for signature symmetry with the three result layers; the sample draws and serialised fit objects are not summary material, so the combined summary is the hc layer.

Usage

```
ssd_summarise(
  dir_sample,
  dir_fit,
  dir_hc,
  path,
  path_with_samples = NULL,
  samples_row_group_bytes = "100MB"
)
```

Arguments

<code>dir_sample</code>	The sample results root.
<code>dir_fit</code>	The fit results root.
<code>dir_hc</code>	The hc results root.
<code>path</code>	The output Parquet path for the compact summary (dists/samples projected out).
<code>path_with_samples</code>	Optional output Parquet path for a full summary that retains the dists/samples list-columns. NULL (the default) writes only the compact summary.
<code>samples_row_group_bytes</code>	The Parquet row-group byte budget for the <code>path_with_samples</code> write (a string DuckDB's <code>ROW_GROUP_SIZE_BYTES</code> accepts, default "100MB"); see the <i>Memory and the full summary's row groups</i> section. Ignored when <code>path_with_samples</code> is NULL.

Details

The compact summary at `path` projects the dists/samples list-columns out at the DuckDB level, so the potentially-large retained bootstrap draws are never pulled into R. Supply `path_with_samples` to **also** write a full summary that retains those list-columns: that write reuses the same lazy DuckDB read, so the draws never materialise in R there either. The draws are populated only when the scenario set `samples = TRUE`, so the full summary is the analysis-ready estimates plus the per-row draws.

In a targets pipeline a directory read carries no dependency edge, so `ssd_scenario_targets()` orders summary after the shards by naming every hc shard target in its command (it re-runs when any hc shard's bytes change). Reading the directory - rather than the shard target values - is what lets it union whatever shards landed (the survivors of a partially-failed run, section 6.2).

Value

The summary Parquet path(s) (the format = "file" contract): path when path_with_samples is NULL, otherwise c(path, path_with_samples).

Memory and the full summary's row groups

The full summary is written in **byte-budgeted Parquet row groups** (samples_row_group_bytes, default "100MB"), so its memory requirement follows the per-group budget - about five times the budget - rather than the union's total row count, and the row-group row count adapts to the samples cell size (large groups for small draws, small groups for large ones). The engine accepts the byte budget because the pipeline configuration scope holds preserve_insertion_order = false (restored when ssd_summarise() returns); it is refused while preserving order, and only the global setting counts - the per-copy PRESERVE_ORDER option cannot substitute. The trade: the full summary's **row order is not contractual** - re-summarising the same shards yields the same rows (address them by hc_id/fit_id), but their order and the file's bytes may differ. Under the default single thread, writes were observed in input order and byte-identical across runs regardless.

Examples

```
data <- ssd_scenario_data(ssddata::ccme_boron)
scenario <- ssd_define_scenario(
  data,
  nsim = 1L,
  nrow = 6L,
  seed = 42L,
  dists = ssd_distset(lnorm = "lnorm")
)
# Materialise the shards single-core, then fan in the hc layer.
run <- ssd_run_scenario_shards(scenario)
ssd_summarise(
  file.path(run$dir, "sample"),
  file.path(run$dir, "fit"),
  file.path(run$dir, "hc"),
  file.path(run$dir, "summary.parquet")
)
```

ssd_summarise_design *Combine Per-scenario Summaries into One Design Summary*

Description

The design pipeline's fan-in: unions the named per-scenario compact summary Parquets into one combined summary, tagging each row with a scenario identity column equal to the member's name within the design. The union is performed at the DuckDB level (each file read lazily via duckplyr, tagged, and union_all-ed straight back out), so no per-scenario summary is collected into R. Per-scenario files that did not land are skipped, so the combined summary unions the surviving members (the keep-going property of `ssd_summarise()`).

Usage

```
ssd_summarise_design(summaries, path)
```

Arguments

summaries	A named character vector of per-scenario compact summary Parquet paths; the names are the scenario names within the design.
path	The output Parquet path for the combined design summary (the format = "file" contract).

Value

path (the format = "file" contract).

See Also

[ssd_design_targets\(\)](#), [ssd_summarise\(\)](#).

ssd_summarise_member	<i>Summarise One Design Member from the Shared hc Shards</i>
----------------------	--

Description

The per-scenario fan-in used by [ssd_design_targets\(\)](#): reads the (shared) hc shards under `dir_hc`, filters to the member's **serving** hc task identities (`hc_ids`) - its own cells, plus the coincident `ci = TRUE` cell serving each of its `ci = FALSE` cells under [ssd_design_targets\(\)](#)'s `ci` routing - projects out the `dists/samples` list-columns at the DuckDB level, and writes the member's compact summary at `path`. When the design aggregates differing readouts into a shared cell, `proportion/est_method` narrow the maximal computed set back to the member's requested read-out rows. The read, filter, projection, and write all happen inside DuckDB (never collecting into R), mirroring [ssd_summarise\(\)](#).

Usage

```
ssd_summarise_member(
  dir_hc,
  hc_ids,
  path,
  proportion = NULL,
  est_method = NULL
)
```

Arguments

<code>dir_hc</code>	The (shared) hc results root of the member's seed group.
<code>hc_ids</code>	The member's serving hc task identities (<code>hc_ids</code>) to keep.
<code>path</code>	The output Parquet path for the member's compact summary (the format = "file" contract).
<code>proportion</code>	Optional proportion values to keep (the member's requested proportions when the design aggregates a wider union into the shared cell); NULL (the default) keeps every proportion.
<code>est_method</code>	Optional <code>est_method</code> values to keep (the member's requested methods when the design aggregates a wider union); NULL (the default) keeps every method.

Value

`path`.

See Also

[ssd_design_targets\(\)](#), [ssd_summarise_design\(\)](#), [ssd_summarise\(\)](#).

`ssd_summarise_uploaded`

Summarise Uploaded Results, In Place (the cloud `ssd_summarise()`)

Description

The cloud counterpart of [ssd_summarise\(\)](#): a generic, dispatched on the upload object's class, that fans a step's **uploaded** shards into a single **lazy** duckplyr table read **in place** (no download). For an Azure destination it reads the `<container>[/<prefix>]/<step>/**/part.parquet` Hive glob - or, for the combined summaries, the single blob `summary.parquet` (`step = "summary"`) / `summary-samples.parquet` (`step = "summary_samples"`, shipped only when the scenario set `samples = TRUE`) - via DuckDB's azure extension - resolving the **same** front-end secret as the write path and remapping it (with the account derived from `url`) into a DuckDB azure secret - and returns the union as a lazy duckplyr tibble (not collected, so the read and projection stay in DuckDB). By default it projects away the heavy `dists/samples` list-columns (the analysis-ready summary, mirroring [ssd_summarise\(\)](#)); pass `drop_samples = FALSE` to keep them when the in-flight bootstrap samples are needed. Because the uploaded compact summary physically lacks those columns, `step = "summary"` with `drop_samples = FALSE` aborts pointing at `step = "summary_samples"` rather than silently returning a sample-less table. The default method (an unknown destination) and the dry-run method both abort.

Usage

```
ssd_summarise_uploaded(
  upload,
  step = "hc",
```

```

    ...,
    drop_samples = TRUE,
    prudence = "stingy"
  )

```

Arguments

upload	An upload destination from <code>ssd_upload_azure()</code> or <code>ssd_upload_dryrun()</code> .
step	One of "sample", "fit", "hc" (the step layer to read), "summary" (the uploaded compact summary), or "summary_samples" (the uploaded full summary retaining the dists/samples list-columns, shipped only when the scenario set <code>samples = TRUE</code>).
...	These dots are for future extensions and must be empty.
drop_samples	Flag (default TRUE): project away the heavy dists/samples list-columns for the analysis-ready summary. Pass FALSE to keep them (e.g. when the in-flight bootstrap samples are needed).
prudence	The duckplyr prudence of the returned table (default "stingy"): "stingy" keeps it lazy and composable but makes an implicit materialisation (e.g. <code>nrow()/\$</code>) against the remote glob error rather than triggering an unbounded download/scan; "lavish" restores automatic materialisation on first access. <code>dplyr::collect()</code> and <code>duckplyr::compute_parquet()</code> work under either.

Value

A **lazy** duckplyr/DuckDB tibble over the unioned, uploaded step layer (not collected), composable with dplyr verbs - `dplyr::collect()` it (or write it with `duckplyr::compute_parquet()`) when you need the rows in R.

See Also

`ssd_open_uploaded()`, `ssd_summarise()`, `ssd_upload_shard()`.

Examples

```

## Not run:
upload <- ssd_upload_azure("https://acct.blob.core.windows.net", "results")
ssd_summarise_uploaded(upload, "hc")
ssd_summarise_uploaded(upload, "hc", drop_samples = FALSE) # keep samples

## End(Not run)

```

ssd_test_upload*Probe an Upload Destination's Credentials and Connectivity*

Description

The front-door check, dispatched on the upload object's class, that confirms **before any compute** whether the destination is reachable and the credentials are in the right place (TARGETS-DESIGN.md section 6.1). Run it as a one-liner at the prompt before `tar_make()` - it is the user's explicit pre-flight. `ssd_scenario_targets()` deliberately does **not** call it (the factory does no network I/O, so sourcing `_targets.R` stays side-effect free); a missing credential still fails loud per-shard at `ssd_upload_shard()` time as a backstop.

Usage

```
ssd_test_upload(upload)
```

Arguments

`upload` An upload destination from `ssd_upload_azure()` or `ssd_upload_dryrun()`.

Details

For an Azure destination it resolves the credentials from the environment and, when a required variable is **absent**, aborts with a loud error **naming the missing variable** (rather than failing later on a worker); when they are present it lists the container and writes then deletes a small marker blob, returning invisibly on success and aborting with the backend's diagnostic on failure. For a dry-run destination it succeeds trivially without resolving credentials or reaching any network.

Value

NULL, invisibly (called for its side effect: the probe).

See Also

`ssd_upload_shard()`, `ssd_open_uploaded()`, `ssd_upload_azure()`.

Examples

```
ssd_test_upload(ssd_upload_dryrun())
```

ssd_upload_azure
Upload Destinations for a Scenario's Shards

Description

Typed, self-validating destination objects for the targets pipeline's per-shard upload (TARGETS-DESIGN.md section 6.1). Pass one to `ssd_scenario_targets()`'s upload argument (the remote-destination sibling of root) to pair each step shard with an upload_<step> target.

Usage

```

ssd_upload_azure(
    url,
    container,
    ...,
    prefix = NULL,
    domain = "blob.core.windows.net"
)

ssd_upload_dryrun()

```

Arguments

url	The Azure Blob Storage account endpoint, e.g. "https://<account>.blob.core.windows.net" (a non-empty string). The storage account name is derived from this endpoint's leading host label (so it need not be repeated in the environment); see domain.
container	The blob container name (a non-empty string).
...	Unused; must be empty. Its presence forces prefix/domain to be passed by name (rlang::check_dots_empty() aborts on a positional or misspelled argument).
prefix	An optional subdirectory (blob-name prefix) within the container under which the shards are written, e.g. "study-2026/run-3", or NULL (default) to write at the container root. Leading/trailing slashes are trimmed. With a prefix the shards land at <container>/<prefix>/<step>/<partition-path>/part.parquet and <code>ssd_open_uploaded()</code> reads them back from the same prefixed glob, so one container can hold several independent result sets.
domain	The storage endpoint domain suffix (default "blob.core.windows.net"). The storage account name is the part of url's host <i>before</i> .<domain> — so https://acct.blob.core.window yields account "acct". Override it for a sovereign/non-public cloud (e.g. "blob.core.usgovcloudapi.r url must end with .<domain> or construction aborts. The derived account is what the read-back path (<code>ssd_open_uploaded()</code> , <code>ssd_summarise_uploaded()</code>) hands to DuckDB's azure secret, so no account environment variable is needed.

Details

`ssd_upload_azure()` describes an Azure Blob Storage container; `ssd_upload_dryrun()` is a no-op destination that reaches no network, so the upload DAG shape can be exercised offline and in CI without credentials. Both return a plain, serialisable S3 object of class `c("ssdsims_upload_<backend>", "ssdsims_upload")` that carries **only the destination** - never credentials, open connections, or environments - so it travels unchanged to crew workers and through targets.

Credentials stay **external** to the object: the Azure methods (`ssd_test_upload()`, `ssd_upload_shard()`, `ssd_open_uploaded()`, `ssd_summarise_uploaded()`) resolve the **secret** from the environment at call time - one of `SSDSIMS_AZURE_STORAGE_KEY`, `SSDSIMS_AZURE_STORAGE_SAS`, or the service-principal trio `SSDSIMS_AZURE_TENANT_ID/SSDSIMS_AZURE_CLIENT_ID/SSDSIMS_AZURE_CLIENT_SECRET`

- and abort with a loud error naming the missing variable when none is present. The storage **account name** is derived from url (see domain), so it is **not** an environment variable.

Value

An S3 object of class `c("ssdsims_upload_azure_blob", "ssdsims_upload")` (for `ssd_upload_azure()`) or `c("ssdsims_upload_dryrun", "ssdsims_upload")` (for `ssd_upload_dryrun()`).

See Also

`ssd_test_upload()`, `ssd_upload_shard()`, `ssd_open_uploaded()`, `ssd_scenario_targets()`.

Examples

```
ssd_upload_azure("https://acct.blob.core.windows.net", "ssdsims-results")
ssd_upload_azure(
  "https://acct.blob.core.windows.net",
  "ssdsims-results",
  prefix = "study-2026/run-3"
)
ssd_upload_dryrun()
```

ssd_upload_shard

Ship Shard (or Summary) Parquet Files to an Upload Destination

Description

A generic, dispatched on the upload object's class, that ships the local Parquet file(s) at path to the destination and returns the **local** path unchanged (so the paired upload target stays format = "file"). The per-shard `upload_<step>` targets pass one path; the `upload_summary` target passes the summary Parquet path(s) - `summary.parquet` plus, when the scenario retains the bootstrap draws, `summary-samples.parquet`. For an Azure destination it resolves the credentials **once per call** and uploads each file to `<url>/<container>[/<prefix>]/<key>`, where the key is the file's path below the layout-keyed results root (a shard's `<step>/<partition-path>/part.parquet`, the summary's `summary.parquet / summary-samples.parquet`; the optional prefix subdirectory comes from `ssd_upload_azure()`; when the required credentials are **absent** it aborts with a

loud error - never a silent no-op - so intent to skip the network is only ever expressed by passing `ssd_upload_dryrun()`. For a dry-run destination it performs no network I/O, records a skip per file, and returns the local path.

Usage

```
ssd_upload_shard(path, upload)
```

Arguments

path	The local Parquet path(s) - a character vector of one or more files (a <step>_step target's single shard path, or the summary target's path(s)).
upload	An upload destination from <code>ssd_upload_azure()</code> or <code>ssd_upload_dryrun()</code> .

Value

The local path (a character vector), unchanged, so the paired upload target stays format = "file".

Adding a backend

The destination set is open and extended by a **constructor-plus-methods** contract - no edit to the existing methods. To add S3, GCS, or another backend:

1. Write a constructor returning an object of class `c("ssdsims_upload_<backend>", "ssdsims_upload")` that validates its destination at construction (as `ssd_upload_azure()` validates its url and container) and carries no credentials.
2. Implement the generic methods for that class: `ssd_upload_shard()` (ship one shard, return the local path), `ssd_test_upload()` (the credentials/connectivity probe, failing loud on a missing credential), `ssd_open_uploaded()` (read the uploaded results back in place), and `ssd_summarise_uploaded()` (the in-place fan-in summary).

The package ships only the Azure and dry-run backends; no speculative backends are added.

See Also

`ssd_test_upload()`, `ssd_open_uploaded()`, `ssd_scenario_targets()`.

Examples

```
path <- tempfile(fileext = ".parquet")
file.create(path)
ssd_upload_shard(path, ssd_upload_dryrun())
```

task_primer

Derive a Per-task Primer from its Parameters

Description

Derives the per-task **primer** – a length-2 integer vector – from `rlang::hash(params)`, suitable for the `stream` argument of `dqrng::dqset.seed()`. Together with the scenario seed, the primer fully specifies a task’s RNG starting point: `dqrng::dqset.seed(seed, stream = task_primer(params))`. It pairs with `local_dqrng_state()`, which installs the (seed, primer) pair under an active `local_dqrng_backend()` scope.

Usage

```
task_primer(params)
```

Arguments

`params` A plain named list of task parameters, or a single-row data frame (one task-table row).

Details

The primer packs 64 bits of the `rlang::hash()` digest (xxhash128) as `c(hi32, lo32)`. Each 32-bit half is encoded as a signed int32, with the reserved bit pattern `0x80000000` (INT_MIN, which R cannot represent as a non-NA integer) mapped to `NA_integer_`; `dqrng` accepts `NA_integer_` in `stream` and treats it as INT_MIN, so the encoding recovers the full 64 bits of stream entropy.

`params` may be a plain named list or a single-row data frame (one row of a {sample, fit, hc}_tasks table). A data-frame row is normalised to a canonical plain list – the inverse of `tibble::tibble_row()` – by dropping all attributes, unwrapping length-1 list-style columns to their element, and leaving df-style (nested data-frame) columns as data frames, before hashing. The primer is therefore identical whether derived from the row or from the equivalent plain list. Note that `rlang::hash()` is order-sensitive, so the plain list must use the **same name order** as the task-table columns to reproduce the row’s primer (assembling `params` in a canonical column order is part of the task-tables caller contract below).

`task_primer()` normalises **structure, not meaning**: it hashes whatever `params` it is given. The canonical, name-keyed representation is a caller contract assembled where `params` is built (task-tables, over the task-lists tables). Per the three-step model the RNG-consuming steps each take a primer over their task identity:

- **sample** – keyed (dataset, sim, replace) only. `nrow` is deliberately absent: every `nrow` shares one draw (sized by the scenario’s `nrow_max` setting) that the `fit` step truncates inline (`head(sample, nrow)`, RNG-free, no separate primer), so excluding `nrow` is load-bearing for the sub-truncation property (TARGETS-DESIGN.md §5).
- **fit** – the parent sample identity plus `nrow` and the fit-grid row (rescale, computable, at_boundary_ok, min_pmix name, range_shape1, range_shape2). `nrow` IS part of the fit primer: a fit on a different truncation is a genuinely different computation.

- **hc** – the parent `fit` identity plus the `hc-grid` row (`nboot`, `est_method`, `ci_method`, `parametric`). `ci` is a scalar `hc` setting applied uniformly and read from the scenario, not part of the task identity, so it is never a primer field (nor a task-row column).

Function-valued parameters (e.g. `min_pmix`) **MUST** be referenced **by name**, not by function value, so a recompile or JIT does not move a task's primer.

The primer is derived from `rlang::hash()`, a fast within-session identity hash rather than a portable checksum, so its digest is not *guaranteed* stable across `rlang` versions. Such changes are rare and not anticipated, but have happened, and would shift primers – and therefore simulation seeds. Reproducibility is therefore anchored by pinning the execution environment (including the `rlang` version) for a given simulation run.

Value

An integer vector of length 2 – the per-task primer – to pass as the `stream` argument of `dqrng::dqset.seed()` (via `local_dqrng_state()`).

See Also

`local_dqrng_state()`, `local_dqrng_backend()`.

Examples

```
task_primer(list(dataset = "boron", sim = 1L, replace = FALSE))
```

Index

*** datasets**
 ssd_cost_calibration_default, 13

dplyr::slice_sample(), 3
dqrng::dqset.seed(), 3, 4, 50, 51

local_dqrng_backend, 3
local_dqrng_backend(), 4, 25, 28, 30, 50, 51
local_dqrng_state, 4
local_dqrng_state(), 25, 50, 51

rlang::hash(), 51
rlang::list2(), 33

scenario_dataset, 5
scenario_dataset(), 7, 17, 29, 31
scenario_distset, 6
scenario_distset(), 22, 31
scenario_min_pmix, 7
scenario_min_pmix(), 5, 6, 15, 27, 28, 31
scenario_results_dir, 8
scenario_results_dir(), 9, 11, 12, 21, 29, 36, 38

ssd_analyse_cost, 9
ssd_analyse_cost(), 12
ssd_calibrate_cost, 10
ssd_calibrate_cost(), 11–14, 23
ssd_calibrate_cost_from_run, 11
ssd_calibrate_cost_from_run(), 9
ssd_compare_cost, 12
ssd_compare_cost(), 9
ssd_cost_calibration, 13
ssd_cost_calibration(), 11–14, 23
ssd_cost_calibration_default, 13, 13
ssd_define_scenario, 14
ssd_define_scenario(), 5–9, 11, 12, 19, 22, 23, 25, 27–29, 33, 34, 36, 39

ssd_design, 19
ssd_design(), 20, 21
ssd_design_targets, 20

ssd_design_targets(), 19, 31, 36, 43, 44
ssd_distset, 22
ssd_distset(), 6, 16, 18
ssd_estimate_cost, 23
ssd_estimate_cost(), 9–14
ssd_gen, 24
ssd_gen(), 15, 18, 33, 34
ssd_open_uploaded, 26
ssd_open_uploaded(), 45–49
ssd_pmix, 27
ssd_pmix(), 15
ssd_run_fit_step(ssd_run_step), 30
ssd_run_hc_step(ssd_run_step), 30
ssd_run_sample_step(ssd_run_step), 30
ssd_run_sample_step(), 30, 35
ssd_run_scenario_baseline, 28
ssd_run_scenario_baseline(), 17, 29–31
ssd_run_scenario_shards, 29
ssd_run_scenario_shards(), 8, 30, 31, 38
ssd_run_step, 30
ssd_scenario_data, 33
ssd_scenario_data(), 15, 17, 19, 24, 25, 27
ssd_scenario_fit_shards
 (ssd_scenario_shards), 34
ssd_scenario_fit_tasks
 (ssd_scenario_tasks), 38
ssd_scenario_fit_tasks(), 35, 39
ssd_scenario_hc_shards
 (ssd_scenario_shards), 34
ssd_scenario_hc_tasks
 (ssd_scenario_tasks), 38
ssd_scenario_hc_tasks(), 22, 23, 35, 39
ssd_scenario_sample_shards
 (ssd_scenario_shards), 34
ssd_scenario_sample_shards(), 30, 31
ssd_scenario_sample_tasks
 (ssd_scenario_tasks), 38
ssd_scenario_sample_tasks(), 35, 39
ssd_scenario_shards, 34

`ssd_scenario_targets`, 35
`ssd_scenario_targets()`, 20, 21, 41, 46–49
`ssd_scenario_tasks`, 38
`ssd_scenario_tasks()`, 28, 34, 39
`ssd_summarise`, 40
`ssd_summarise()`, 8, 42–45
`ssd_summarise_design`, 42
`ssd_summarise_design()`, 20, 21, 44
`ssd_summarise_member`, 43
`ssd_summarise_uploaded`, 44
`ssd_summarise_uploaded()`, 47–49
`ssd_test_upload`, 46
`ssd_test_upload()`, 26, 38, 48, 49
`ssd_upload_azure`, 47
`ssd_upload_azure()`, 20, 26, 36, 38, 45, 46, 48, 49
`ssd_upload_dryrun (ssd_upload_azure)`, 47
`ssd_upload_dryrun()`, 20, 26, 36, 38, 45, 46, 49
`ssd_upload_shard`, 48
`ssd_upload_shard()`, 26, 45, 46, 48
`ssdtools::ssd_dists_all()`, 22
`ssdtools::ssd_fit_dists()`, 24
`ssdtools::ssd_hc()`, 16
`system.time()`, 10

`task_primer`, 50
`task_primer()`, 25, 28, 34

`with_dqrng_state (local_dqrng_state)`, 4
`withr::local_seed()`, 3, 4