Package: genproc (via r-universe)

May 21, 2026

Title Robust, Logged and Reproducible Iteration at Organizational Scale

Version 0.2.0

Description Turns one-off iterative R procedures (such as for loops, lapply() or pmap() from 'purrr') into production-grade workflows by wrapping them with orthogonal, composable execution layers. Two layers are always active: structured logging with real traceback and per-case timing; and reproducibility capture, which records the R version, loaded package versions, execution environment, the exact iteration mask, and a stat-based fingerprint of every input file referenced in the mask (with a diff_inputs() helper to detect silent drift between runs). Parallel execution (built on the 'future' framework, Bengtsson (2021) <doi:10.32614/RJ-2021-048>), non-blocking background jobs, and opt-in progress reporting (via 'progressr') are implemented as optional, composable layers. Further layers (error replay, content-hash input fingerprinting, content-based case identifiers) are planned and will remain composable with the default layers.

License MIT + file LICENSE

URL https://danielrak.github.io/genproc/, https://github.com/danielrak/genproc

BugReports https://github.com/danielrak/genproc/issues

Imports future, future.apply, parallel, progressr, stats, utils

Suggests dplyr, knitr, magrittr, purrr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Repository https://danielrak.r-universe.dev

Date/Publication 2026-05-21 17:18:35 UTC

RemoteUrl https://github.com/danielrak/genproc

RemoteRef HEAD

RemoteSha 28324a58c8e76bb7d7db90abfd21c42a572a32b4

Contents

add_trycatch_logrow 2
await 3
diff_inputs 4
errors 5
from_example_to_function 6
from_function_to_mask 8
genproc 9
nonblocking_spec 13
parallel_spec 15
print.genproc_result 16
print.genproc_result_summary 17
rename_function_params 18
rerun_affected 19
rerun_failed 20
status 22
summary.genproc_result 23

Index **25**

`add_trycatch_logrow` *Wrap a function with error handling and a structured log row*

Description

Takes a function and returns a modified version that:

- always returns a one-row data.frame (the "log row")
- captures errors without stopping, recording the error message and the **real** traceback (call stack at the point of failure)
- records wall-clock execution time

Usage

```
add_trycatch_logrow(f)
```

Arguments

f	A function to wrap.

Details

Why not plain tryCatch?:
`tryCatch()` unwinds the call stack before entering the error handler. This means `sys.calls()` inside a `tryCatch` error handler returns the handler's own stack, not the stack that led to the error. This function uses `withCallingHandlers()` (which fires while the original stack is still intact) to capture the traceback, then lets the error propagate to an outer `tryCatch()` for control flow.

Log row structure:
The returned data.frame always has one row. Columns:

- One column per formal argument of `f`, with the value passed
- `success`: `TRUE` if the body executed without error
- `error_message`: the error's `conditionMessage()`, or `NA`
- `traceback`: the formatted call stack at the error, or `NA`
- `duration_secs`: wall-clock seconds elapsed

Value

A function with the same formals as `f`, whose return value is a one-row data.frame with columns: all original arguments, `success` (logical), `error_message` (character or `NA`), `traceback` (character or `NA`), `duration_secs` (numeric).

Examples

```
safe_sqrt <- add_trycatch_logrow(function(x) sqrt(x))

# Happy path: one-row data.frame, success = TRUE.
safe_sqrt(4)[, c("x", "success", "error_message", "duration_secs")]

# Failing call: the run does not stop. The row carries the error
# message and a filtered traceback instead of throwing.
bad <- safe_sqrt("a")
bad[, c("x", "success", "error_message")]
```

---

await *Block until a non-blocking genproc run has resolved*

Description

`await()` blocks until the background future of a non-blocking `genproc()` run has resolved, then returns a `genproc_result` with `log`, `n_success`, `n_error`, `duration_total_secs`, and `status` populated. If the wrapper future itself crashed (a rare case — user errors inside individual cases are caught by the logging layer and are *not* wrapper crashes), the returned object has `status = "error"` and a populated `error_message`.

Usage

```
await(x, ...)

## S3 method for class 'genproc_result'
await(x, ...)
```

Arguments

x	An object. Methods exist for genproc_result.
...	Unused, for future extensions.

Details

await() is idempotent: calling it on a result that has already been materialized (or was synchronous to begin with) returns it unchanged.

Value

A genproc_result with status != "running".

See Also

status(), nonblocking_spec()

---

diff_inputs *Compare input file fingerprints between two genproc runs*

Description

Takes two genproc_result objects produced by genproc() (the same function over the same mask, run at two different times) and reports which referenced input files have changed since the first run.

Usage

```
diff_inputs(r0, r1)

## S3 method for class 'genproc_input_diff'
print(x, ...)
```

Arguments

r0, r1	Two genproc_result objects. By convention, r0 is the earlier run and r1 the later one, but the function is symmetric with respect to changed / unchanged. The labels removed (present in r0, absent in r1) and added (the opposite) follow the asymmetric convention.
x	A genproc_input_diff object.
...	Ignored (present for S3 method consistency).

Details

Files are matched by canonical absolute path. The `method` field must agree between the two runs.

Value

An object of class `genproc_input_diff` (a named list) with components:

method Character, e.g. `"stat"`.

changed A data.frame with columns `path`, `size_before`, `size_after`, `mtime_before`, `mtime_after`. One row per file whose size or mtime differs.

unchanged Character vector of paths whose size and mtime are identical in both runs.

removed Character vector of paths present in `r0`'s snapshot but absent in `r1`'s.

added Character vector of paths present in `r1`'s snapshot but absent in `r0`'s.

cases_affected A data.frame with columns `case_id`, `path`, `column`, `change_type` (one of `"changed"`, `"removed"`, `"added"`). One row per (case, input column) impacted by the diff. Pass to `rerun_affected()` to re-run only the impacted cases.

Examples

```
# Two runs of the same procedure, with one input file rewritten
# in between. `diff_inputs()` reports the drift.
src <- file.path(tempdir(), "diff-inputs-demo")
dir.create(src, showWarnings = FALSE, recursive = TRUE)
write.csv(head(iris), file.path(src, "a.csv"), row.names = FALSE)

mask <- data.frame(
  path = file.path(src, "a.csv"),
  stringsAsFactors = FALSE
)
read_one <- function(path) nrow(read.csv(path))

r0 <- genproc(read_one, mask)

# Rewrite the file with strictly more rows: size changes.
write.csv(iris, file.path(src, "a.csv"), row.names = FALSE)

r1 <- genproc(read_one, mask)
diff_inputs(r0, r1)
```

`errors` *Subset a genproc result to its failed cases*

Description

Returns the rows of `result$log` corresponding to cases where `success == FALSE`. The columns are exactly those of `result$log` (case_id, mask parameters, success, error_message, traceback, duration_secs).

Usage

```
errors(x, ...)

## S3 method for class 'genproc_result'
errors(x, ...)
```

Arguments

x	A genproc_result produced by genproc().
...	Unused, for future extensions.

Value

A data.frame with one row per failed case. Empty data.frame (with the same columns) if there are no failures. Returns NULL (with a message) if the run is non-blocking and has not been materialized yet.

See Also

rerun_failed(), summary.genproc_result()

Examples

```
result <- genproc(
  f = function(x) if (x %% 2 == 0) x / 0 else x,
  mask = data.frame(x = 1:6)
)
errors(result)[, c("case_id", "x", "error_message")]
```

---

from_example_to_function

Transform an example expression into a parameterized function

---

Description

Takes a concrete R expression (e.g. a data processing script that works on one specific case) and returns a function where every external value (strings, variables from the environment that are not functions) has been replaced by a named parameter with the original value as default.

Usage

```
from_example_to_function(expr, env = parent.frame())
```

Arguments

expr	An expression of length 1, typically created with `expression()` or `quote()` wrapped in `as.expression()`.
env	The environment in which to look up symbols. Symbols found in this environment that are **not** functions will be turned into parameters. Defaults to the caller's environment.

Details

This is the first step in the genproc workflow: the user writes a working example, and `from_example_to_function()` extracts a reusable, parameterized version of it.

What gets parameterized:

- **String literals**: every string in the expression becomes a parameter (e.g. `"output.csv"` -> `param_1` with default `"output.csv"`).
- **Non-function symbols**: if a symbol exists in `env` and its value is not a function, it becomes a parameter.

What is left unchanged:

- **Locally bound symbols**: variables created by assignments inside the expression (e.g. `result <- ...`) are never parameterized.
- **Function names**: the head of a call (e.g. `read.csv` in `read.csv(path)`) is never parameterized.
- **Functions in the environment**: symbols whose value is a function are assumed to be part of the program structure, not data.
- **Numeric, logical, NULL, and other non-character atomic values**.

Deduplication:

The same value produces the same parameter. If `"output.csv"` appears twice, both occurrences map to the same `param_N`.

Value

A function whose body is the rewritten expression and whose formals are the detected parameters with their default values. The function's environment is set to `env`.

Examples

```
# --- Basic usage ---
# `input_path` exists in this environment; "output.csv" is a
# string literal. Both become parameters of the resulting function,
# with their original values as defaults.
input_path <- "/data/input.csv"

expr <- expression({
  df <- read.csv(input_path)
  write.csv(df, "output.csv")
})
```

```
fn <- from_example_to_function(expr)
formals(fn)

# --- Local bindings are protected ---
# `x` is assigned inside the block, so it is NOT parameterized
# even though x = 42 exists in the calling environment.
x <- 42
expr2 <- expression({
  x <- 1
  y <- x + 1
})
fn2 <- from_example_to_function(expr2)
formals(fn2)
```

`from_function_to_mask` *Derive an iteration mask template from a function's signature*

Description

Takes a function (typically produced by `from_example_to_function()`) and returns a one-row data.frame where each column corresponds to a parameter, with the default value as the cell value. This "template mask" is the starting point the user expands into a multi-row mask that defines all iteration cases.

Usage

```
from_function_to_mask(f)
```

Arguments

`f`	A function whose formals define the mask columns.

Details

What is a mask?:

In genproc, a **mask** is a data.frame where each row is an iteration case and each column is a parameter. The function `genproc()` will call the user's function once per row, passing column values as arguments.

`from_function_to_mask()` produces a one-row template. The user then builds the full mask by adding rows (e.g. via `rbind()`, `dplyr::bind_rows()`, or by constructing a multi-row data.frame directly).

Current limitations (v0.1):

Only scalar atomic defaults are supported (character, numeric, integer, logical). Non-scalar defaults (vectors, lists, data.frames) will be supported in a future version via list-columns. This extension will preserve backwards compatibility: any mask that works today will continue to work unchanged.

Metadata (case_id, hashes, etc.):

The mask returned here is a **pure data.frame of parameter values**. Metadata such as `case_id`, input file hashes, or seeds are managed separately by `genproc()` at execution time — they are not stored as columns or attributes of the mask. This design ensures that standard data.frame operations (`dplyr::filter()`, `[`, `rbind()`) never accidentally strip metadata.

When the mask is later generalized to a dedicated class (`genproc_mask`), existing code passing a plain data.frame will continue to work (backwards compatibility is a hard constraint).

Value

A one-row data.frame with one column per parameter. Parameters with default values get those values; parameters without defaults get `NA`.

Examples

```
fn <- function(input_path = "data.csv", n_rows = 100) {
  head(read.csv(input_path), n_rows)
}
mask <- from_function_to_mask(fn)
mask
#   input_path n_rows
# 1   data.csv    100

# Expand to multiple cases:
full_mask <- data.frame(
  input_path = c("a.csv", "b.csv", "c.csv"),
  n_rows = c(10, 50, 100)
)
```

`genproc` *Run a function over a mask with mandatory logging and reproducibility*

Description

This is the central function of the genproc package. It takes a function and an iteration mask (data.frame), calls the function once per row of the mask, and returns a structured result with:

- a log data.frame (one row per case, with success/error/traceback/timing)
- reproducibility information (R version, packages, environment, parallel spec)
- the exact mask used
- stable case IDs linking log rows to mask rows

Usage

```
genproc(
  f,
  mask,
  f_mapping = NULL,
  parallel = NULL,
  nonblocking = NULL,
  track_inputs = TRUE,
  input_cols = NULL,
  skip_input_cols = NULL
)
```

Arguments

f	A function to apply to each row of the mask. Each formal of f should correspond to a column in mask (or have a default value). Can be produced by from_example_to_function() or written by hand.
mask	A data.frame where each row is an iteration case and each column is a parameter value. Can be produced by from_function_to_mask() and expanded by the user.
f_mapping	Optional named character vector to rename f's parameters before execution. Passed to rename_function_params(). Names are current parameter names, values are new names matching mask columns.
parallel	NULL (default, sequential execution) or a genproc_parallel_spec object produced by parallel_spec(). When supplied, cases are dispatched to workers via future.apply::future_lapply().
nonblocking	NULL (default, synchronous call) or a genproc_nonblocking_spec object produced by nonblocking_spec(). When supplied, genproc() returns immediately with a genproc_result of status "running", and the run continues in a background future. Use status() to poll the state and await() to block until resolution. Can be combined with parallel — the non-blocking wrapper envelops the parallel dispatch.
track_inputs	Logical. When TRUE (default), genproc detects columns of mask that reference input files and records their size + mtime in result$reproducibility$inputs. Use diff_inputs() to compare two runs and detect silent input drift. Set to FALSE to skip input tracking entirely.
input_cols	NULL (default) or a character vector of mask column names. When supplied, the heuristic detection is bypassed and exactly these columns are tracked. Paths that do not exist at capture time are recorded with NA size/mtime and a warning is emitted. Mutually exclusive with skip_input_cols.
skip_input_cols	NULL (default) or a character vector of mask column names to exclude from heuristic detection. Useful when a label column happens to match an existing file. Mutually exclusive with input_cols.

Details

The *logged* and *reproducibility* layers are always active and cannot be disabled. Three optional layers compose on top: *parallel* (pass a `parallel_spec()` to `parallel`), *non-blocking* (pass a `nonblocking_spec()` to `nonblocking`), and *progress monitoring* (wrap the call in `progressr::with_progress()`).

Execution model:

Cases are executed **sequentially** in row order by default. Supply `parallel = parallel_spec(...)` to dispatch them in parallel via the **future** ecosystem. The logging and reproducibility layers remain active in both modes; the parallel layer is strictly additive.

Parallel execution preserves the mask row order in the resulting `log` data.frame, regardless of the order in which workers return.

Parallel execution requires **genproc** to be installed (not only loaded via `devtools::load_all()`) on each worker, because the logging layer serializes closures whose environments reference genproc internals. The only exception is `parallel_spec(strategy = "sequential")`, which runs in the current process and needs nothing extra — this is the recommended mode for deterministic testing.

Error handling:

Errors in individual cases do **not** stop the run. Each case is wrapped with `add_trycatch_logrow()`, which captures the error message and the real traceback (via `withCallingHandlers`). The run continues with the next case. This holds identically in sequential and parallel mode.

Progress monitoring:

`genproc()` emits one `progressr` signal per completed case in sequential and parallel modes. The signals are no-op unless the calling code is wrapped in `progressr::with_progress(...)`, in which case the user sees a progress bar (or any other handler chosen via `progressr::handlers()`):

```
library(progressr)
with_progress(
  result <- genproc(my_fn, mask, parallel = parallel_spec(workers = 4))
)
```

Without `with_progress()`, there is zero overhead and zero visible change: the integration is a hook, not a default behaviour. `progressr` is declared in `Suggests`; the integration is conditional on its installation. The non-blocking path does not yet emit signals — live monitoring of background runs is on the roadmap.

Composing parallel and non-blocking:

When both `parallel` and `nonblocking` are supplied, the non-blocking wrapper envelops the parallel dispatch (one outer future submits the run, inner workers process the cases). On platforms where the wrapper subprocess R inherits a restrictive default for `getOption("mc.cores")` (typically 1 on Windows and in some RStudio configurations), `parallelly` would otherwise refuse to spawn the inner workers. `genproc()` works around this with two surgical adjustments inside the wrapper subprocess, applied *only* in the composed case and *only* when the user has not set their own values:

1. Set `R_PARALLELLY_AVAILABLECORES_METHODS = "system"` so that `availableCores()` ignores the legacy `mc.cores` option and reports the true detected core count (lifts the hard-limit refusal).

2. Raise `options(mc.cores)` from 1 to the system core count, so that `parallelly`'s soft-limit warning no longer fires with a misleading "only 1 CPU cores available" message.

The calling session is never modified by either adjustment.

Case IDs:

Each row of the mask receives a `case_id` (currently index-based: `case_0001`, `case_0002`, ...). This ID appears in the log and can be used for replay, monitoring, and cross-referencing.

Parameter matching:

The mask does not need to contain a column for every parameter of `f`. Parameters not present in the mask will use their default values. However, parameters without defaults that are also missing from the mask will cause an error before execution starts.

Extra columns in the mask (not matching any parameter) are silently ignored.

Value

An object of class `genproc_result` (a named list) with components:

log A data.frame with one row per case. Contains all parameter values, plus `case_id`, `success`, `error_message`, `traceback`, and `duration_secs`.

reproducibility A list of environment information captured at run start (R version, packages, OS, locale, timezone, mask snapshot, parallel and non-blocking specs if any, and `inputs` — a stat-based fingerprint of every input file referenced by the mask). See `capture_reproducibility()`.

n_success Integer, number of successful cases.

n_error Integer, number of failed cases.

duration_total_secs Numeric, total wall-clock time for the entire run.

status Character. `"done"` for a synchronous run that has completed; `"running"` for a non-blocking run whose future has not resolved yet; `"done (not collected)"` for a non-blocking run whose future has resolved but whose result has not been collected via `await()` yet; `"error"` when the background run errored out.

The `genproc_result` class is designed for forward compatibility. Existing fields (`log`, `reproducibility`, `n_success`, `n_error`, `duration_total_secs`) are guaranteed stable. Future versions may add new fields (e.g. `worker_id` in the log for parallel runs, or `collect()`/`poll()` methods for non-blocking execution) but will never remove or rename existing ones.

See Also

Optional execution layers: `parallel_spec()`, `nonblocking_spec()`, `status()`, `await()`. Inspecting a result: `errors()`, `summary.genproc_result()`, `rerun_failed()`. Reproducibility tooling: `diff_inputs()`, `rerun_affected()`.

Examples

```
# Sequential run (the default). Returns immediately when done.
result <- genproc(
  f = function(x, y) x + y,
  mask = data.frame(x = c(1, 2, 3), y = c(10, 20, 30))
```

```
)
result$log

# One-off parallel call: genproc installs a temporary multisession
# plan and restores the previous one on exit. Capped at 2 workers
# to comply with the CRAN policy on parallelism in examples.

  result <- genproc(
    f = function(x) x * 2,
    mask = data.frame(x = 1:4),
    parallel = parallel_spec(workers = 2)
  )
  result$log


# Non-blocking + parallel composed: launch in the background,
# keep the console, collect later with await().

  job <- genproc(
    f = function(x) x * 2,
    mask = data.frame(x = 1:4),
    parallel    = parallel_spec(workers = 2),
    nonblocking = nonblocking_spec()
  )
  status(job)         # "running" until the future resolves
  job <- await(job)   # blocks; idempotent on already-resolved jobs
  job$log
```

`nonblocking_spec` *Specify a non-blocking execution strategy for genproc()*

Description

Returns a configuration object to pass as the `nonblocking` argument of `genproc()`. When supplied, `genproc()` returns immediately with a `genproc_result` of status `"running"` while the actual work continues in a background future. Use `status()` to poll the state and `await()` to block until completion.

Usage

```
nonblocking_spec(strategy = "multisession", packages = NULL, globals = TRUE)
```

Arguments

`strategy`	Character, or `NULL`. One of `"sequential"`, `"multisession"`, `"multicore"`, `"cluster"`. Default `"multisession"`. Unlike `parallel_spec()`, the default is not `NULL`: a function named "non-blocking" must not silently block because the current `future::plan()` is sequential. Pass `strategy = NULL` explicitly to

	defer to the caller's plan. `"sequential"` is accepted mainly for deterministic testing — it exercises the code path but does *not* actually free the console.
`packages`	Character vector, or `NULL`. Extra packages to attach on the background worker before running. **genproc** itself is attached automatically for every strategy other than `"sequential"`.
`globals`	Logical or character. Forwarded to `future::future()`'s `globals` argument. Default `TRUE` enables automatic detection.

Value

A list of class `"genproc_nonblocking_spec"` with the validated, normalized fields.

Composition with parallel_spec()

`nonblocking_spec()` and `parallel_spec()` are orthogonal and can be combined. The nonblocking layer launches *one* outer future; inside it, the parallel layer dispatches cases via **future.apply**. With both strategies set to `"multisession"`, **future** resolves the inner layer as `"sequential"` by default (see `future::plan()` nesting rules) unless the caller installs an explicit nested plan via `future::plan(list(...))`.

On Windows and in some RStudio configurations, the wrapper subprocess inherits `getOption("mc.cores")` set to 1, which would lead `parallelly` to refuse the inner workers ("only 1 CPU core available for this R process"), and to also emit a misleading soft-limit warning even after the refusal is lifted. `genproc()` works around both issues transparently in the composed case (only when the user has not set their own values). See ?genproc for details.

See Also

`parallel_spec()`, `status()`, `await()`

Examples

```
# Launch in the background, keep the console.

  spec <- nonblocking_spec()
  job <- genproc(
    f = function(x) x * 2,
    mask = data.frame(x = 1:4),
    nonblocking = spec
  )
  status(job)           # "running"
  job <- await(job)     # blocks until done
  job$log


# Deterministic test: exercise the code path without real async
spec <- nonblocking_spec(strategy = "sequential")
```

parallel_spec *Specify a parallel execution strategy for genproc()*

Description

Returns a configuration object to pass as the `parallel` argument of `genproc()`. The object describes *how* to parallelize; the actual execution is carried out by `future.apply::future_lapply()` on top of the backend selected by `future::plan()`.

Usage

```
parallel_spec(
  workers = NULL,
  strategy = NULL,
  chunk_size = NULL,
  seed = TRUE,
  packages = NULL,
  globals = TRUE
)
```

Arguments

`workers`	Integer >= 1, or `NULL`. Number of workers to use. Ignored when `strategy = "sequential"`. If `NULL`, the current `future::plan()` decides.
`strategy`	Character, or `NULL`. One of `"sequential"`, `"multisession"`, `"multicore"`, `"cluster"`. If `NULL` (default), the current `future::plan()` is used unchanged. If specified, genproc temporarily sets the corresponding plan for the run and restores the previous plan on exit.
`chunk_size`	Integer >= 1, or `NULL`. Number of iteration cases bundled per future. Larger values reduce scheduling overhead at the cost of load-balance granularity. `NULL` delegates to `future.apply`'s default heuristic.
`seed`	Controls reproducible random-number generation across workers. Passed to `future.apply::future_lapply()`'s `future.seed` argument. Default `TRUE` derives independent L'Ecuyer-CMRG streams from a random master seed. A single integer fixes the master seed. `FALSE` disables reproducible RNG and is not recommended unless the user function is known to be RNG-free.
`packages`	Character vector, or `NULL`. Extra packages to attach on each worker before running the user function. **genproc** itself is attached automatically for every strategy other than `"sequential"`.
`globals`	Logical or character. Forwarded to `future.apply::future_lapply()`'s `future.globals`. Default `TRUE` enables automatic detection, which is correct in almost all cases. Set to a character vector only to override detection.

Value

A list of class `"genproc_parallel_spec"` with the validated, normalized fields.

Choosing a strategy

- `"sequential"`: runs in the current process, no workers. Exercises the parallel code path without the overhead; useful for deterministic testing.
- `"multisession"`: portable (works on Windows), launches R subprocesses via **parallelly**. The default recommendation for most workloads.
- `"multicore"`: forks the current process (Unix/macOS only, **not available on Windows** and not reliable inside RStudio). Faster startup than multisession but loses portability.
- `"cluster"`: explicit cluster of workers, possibly on other machines. For large-scale batch execution.

For most users, leaving `strategy = NULL` and calling `future::plan()` once at the top of the session is the cleanest setup.

RNG reproducibility

With `seed = TRUE`, each case receives an independent L'Ecuyer-CMRG stream derived from a random master seed. Same master seed -> identical results regardless of worker count or chunking. To pin the master seed, pass an integer (`seed = 42L`).

Examples

```
# Use whatever plan the caller has set
spec <- parallel_spec()

# One-off parallel call with 4 workers, reproducible RNG
spec <- parallel_spec(workers = 4, strategy = "multisession",
                      seed = 42L)

# Exercise the parallel code path deterministically in a test
spec <- parallel_spec(strategy = "sequential")
```

`print.genproc_result` *Print a genproc result*

Description

Displays a structured summary of the run: status, timestamp, execution mode, case counts, total duration, and an actionable hint when relevant.

Usage

```
## S3 method for class 'genproc_result'
print(x, ...)
```

Arguments

`x`	A `genproc_result` object.
`...`	Ignored (present for S3 method consistency).

Details

For non-blocking results, the status is queried *live* from the attached future via `status()` rather than read from the stored field. Repeated `print(x)` calls therefore reflect the actual progress of the background run. `status()` distinguishes `"done"` (the future resolved successfully) from `"error"` (the wrapper future itself crashed). Numeric fields stay `(pending)` until `await()` is called to materialize the result.

When the parallel layer was used and startup overhead clearly dominated the run, the print method emits a `Note` hinting at the issue — a pattern that often surprises users on small workloads. Two metrics depending on whether `workers` is known: parallel efficiency (`(cumulative / workers) / wall`) below 50% when `workers` is supplied, or `wall > cumulative * 1.2` in power-user mode (workers unknown). Both require `wall > 0.5s` to avoid noise on very short runs.

Value

`x`, invisibly.

`print.genproc_result_summary` *Print method for genproc_result_summary*

Description

Print method for genproc_result_summary

Usage

```
## S3 method for class 'genproc_result_summary'
print(x, ...)
```

Arguments

`x`	A `genproc_result_summary` produced by `summary.genproc_result()`.
`...`	Unused, for S3 method consistency.

Value

`x`, invisibly.

`rename_function_params` *Rename the parameters of a function*

Description

Takes a function and a name mapping, and returns a new function where both the formals and all symbol references in the body have been renamed according to the mapping. This is typically used after `from_example_to_function()` to replace generated names like `param_1`, `param_2` with meaningful names.

Usage

```
rename_function_params(f, mapping)
```

Arguments

`f`	A function whose parameters should be renamed.
`mapping`	A named character vector. Names are the **current** parameter names, values are the **new** names. Example: `c(param_1 = "input_path", param_2 = "output_path")`.

Details

Validation:

The function checks that:

- All names in `mapping` actually exist as formals of `f`
- New names are unique (no duplicates)
- New names do not collide with parameters not being renamed

Limitation:

If the body contains a nested function definition whose formals shadow a parameter being renamed, the shadowed references in that inner body will still be renamed. This is unlikely in practice (parameters from `from_example_to_function()` are named `param_N`) but is noted here for completeness.

Value

A function with renamed formals and body.

Examples

```
fn <- function(param_1 = "in.csv", param_2 = "out.csv") {
  df <- read.csv(param_1)
  write.csv(df, param_2)
}

fn2 <- rename_function_params(fn, c(
```

```
  param_1 = "input_path",
  param_2 = "output_path"
))

# Formals were renamed:
formals(fn2)

# And the body too — references to `param_1` and `param_2` are
# updated in place, the function source is not edited.
body(fn2)
```

`rerun_affected` *Re-run only the cases impacted by an input diff*

Description

Filters the original mask of `r0` down to the cases that referenced inputs reported as changed, removed, or added by `diff_inputs()`, and re-runs `genproc()` on that subset.

Usage

```
rerun_affected(
  r0,
  diff,
  f,
  parallel = NULL,
  nonblocking = NULL,
  track_inputs = TRUE,
  input_cols = NULL,
  skip_input_cols = NULL
)
```

Arguments

`r0`	A `genproc_result` produced by `genproc()`. Its `$reproducibility$mask_snapshot` provides the original mask; it must contain `track_inputs = TRUE` (the default).
`diff`	A `genproc_input_diff` produced by `diff_inputs()`.
`f`	A function. Typically the same function passed to the original `genproc()` call. The result object does not store `f`, so it must be supplied here.
`parallel, nonblocking, track_inputs, input_cols, skip_input_cols`	Forwarded to `genproc()` for the re-run. By default, these inherit a sensible behaviour: `track_inputs = TRUE` (so the re-run is itself comparable), the other arguments default to `NULL` (sequential, blocking, automatic input tracking).

Details

This is the actionable end of the reproducibility layer: when an upstream file silently drifts, you do not need to re-run the whole mask. `rerun_affected()` produces a smaller run that refreshes only the impacted outputs.

Value

A new `genproc_result` covering only the affected cases. Its `case_id`s are local to the subset (re-numbered starting at `case_0001`); the link back to the original `r0` is via the matching rows of `r0$reproducibility$mask_snapshot`. If `diff` reports no affected cases, the function returns `NULL` with a message — there is nothing to re-run.

See Also

`diff_inputs()`, `genproc()`

Examples

```
# Set up a tiny workspace with one tracked input file.
csv <- tempfile(fileext = ".csv")
write.csv(iris, csv, row.names = FALSE)

count_rows <- function(p) nrow(read.csv(p))

r0 <- genproc(count_rows, data.frame(p = csv))

# ... time passes, the upstream file is silently rewritten ...
write.csv(head(iris), csv, row.names = FALSE)

r1 <- genproc(count_rows, data.frame(p = csv))
d <- diff_inputs(r0, r1)
# d$cases_affected lists the case_ids whose inputs drifted.

refreshed <- rerun_affected(r0, d, f = count_rows)
refreshed$log
```

---

rerun_failed *Re-run only the cases that failed*

Description

Filters the original mask of `r0` down to the cases for which `success == FALSE` and re-runs `genproc()` on that subset. Useful when a transient external problem caused some cases to fail and the user has fixed the cause: rather than re-running the whole mask, only the failed cases are refreshed.

Usage

```
rerun_failed(
  r0,
  f,
  parallel = NULL,
  nonblocking = NULL,
  track_inputs = TRUE,
  input_cols = NULL,
  skip_input_cols = NULL
)
```

Arguments

r0	A genproc_result produced by genproc(). Its $reproducibility$mask_snapshot provides the original mask.
f	A function. Typically the same function passed to the original genproc() call. The result object does not store f, so it must be supplied here. If the previous failures were caused by a bug in f, pass the corrected version.
parallel, nonblocking, track_inputs, input_cols, skip_input_cols	Forwarded to genproc() for the re-run.

Value

A new genproc_result covering only the failed cases. Its case_ids are local to the subset (re-numbered starting at case_0001); the link back to the original r0 is via the matching rows of r0$reproducibility$mask_snapshot. If r0 has zero failed cases, the function returns NULL with a message — there is nothing to re-run.

See Also

rerun_affected(), errors(), summary.genproc_result()

Examples

```
r0 <- genproc(
  f = function(x) if (x %% 2 == 0) stop("even") else x,
  mask = data.frame(x = 1:6)
)
# 3 cases failed (the even ones). After fixing f, retry only those:

  r1 <- rerun_failed(r0, f = function(x) abs(x))
  r1$log
```

`status` *Query the status of a genproc run without blocking*

Description

`status()` is a non-blocking S3 generic. On a `genproc_result`, it returns one of:

- `"running"` — the underlying future is not yet resolved.
- `"done"` — the future has resolved successfully (the result is ready to be collected via `await()`).
- `"error"` — the wrapper future itself crashed. Call `await()` to retrieve the error message.

Usage

```
status(x, ...)

## S3 method for class 'genproc_result'
status(x, ...)
```

Arguments

`x`	An object. Methods exist for `genproc_result`.
`...`	Unused, for future extensions.

Details

For a synchronous (non-nonblocking) result, `status()` simply returns `result$status` (`"done"` or `"error"`).

`status()` peeks at the resolved future via `future::value()` inside a `tryCatch`. Because `value()` consumes the future, the peek result is cached in a shared environment so that a subsequent `await()` does not re-materialize it.

Value

A single character string: `"running"`, `"done"`, or `"error"`.

See Also

`await()`, `nonblocking_spec()`

summary.genproc_result *Summarise a genproc result*

Description

Produces a compact digest of the run: status, success rate, duration stats, and the top recurring error messages. Useful on runs with a lot of cases where the raw log is too noisy to eyeball.

Usage

```
## S3 method for class 'genproc_result'
summary(object, top_errors = 10L, ...)
```

Arguments

`object`	A `genproc_result` produced by `genproc()`.
`top_errors`	Integer. Maximum number of distinct error messages to include in the summary, ranked by occurrence. Default 10.
`...`	Unused, for future extensions.

Value

An object of class `genproc_result_summary` (a list) with components:

materialized Logical. `FALSE` if the run is non-blocking and has not been collected via `await()`. In that case the other fields are `NA`.

status Character, mirrors `result$status`.

n_cases Integer.

n_success, n_error Integers.

success_rate Numeric in 0..1.

duration_total_secs Numeric, wall-clock total.

duration_stats List with `total`, `mean`, `max`, and `slowest_case_id`. `NULL` if no per-case durations.

top_errors data.frame with columns `error_message` and `count`, sorted by count descending. Trimmed to `top_errors` rows.

See Also

`errors()`, `rerun_failed()`

Examples

```
result <- genproc(
  f = function(x) {
    if (x %% 2 == 0) stop("even")
    if (x %% 3 == 0) stop("multiple of three")
    x
  },
  mask = data.frame(x = 1:12)
)
summary(result)
```

Index

add_trycatch_logrow, 2
add_trycatch_logrow(), *11*
as.expression(), *7*
await, 3
await(), *10*, *12–14*, *17*, *22*, *23*

diff_inputs, 4
diff_inputs(), *10*, *12*, *19*, *20*

errors, 5
errors(), *12*, *21*, *23*
expression(), *7*

from_example_to_function, 6
from_example_to_function(), *8*, *10*, *18*
from_function_to_mask, 8
from_function_to_mask(), *10*
future.apply::future_lapply(), *10*, *15*
future::plan(), *15*

genproc, 9
genproc(), *3*, *4*, *6*, *8*, *9*, *13–15*, *19–21*, *23*
genproc_result, *4*

nonblocking_spec, 13
nonblocking_spec(), *4*, *10–12*, *22*

parallel_spec, 15
parallel_spec(), *10–14*
print.genproc_input_diff *(*diff_inputs*)*, 4
print.genproc_result, 16
print.genproc_result_summary, 17

quote(), *7*

rename_function_params, 18
rename_function_params(), *10*
rerun_affected, 19
rerun_affected(), *5*, *12*, *21*
rerun_failed, 20
rerun_failed(), *6*, *12*, *23*

status, 22
status(), *4*, *10*, *12–14*, *17*
summary.genproc_result, 23
summary.genproc_result(), *6*, *12*, *17*, *21*