Package: toolero (via r-universe)

October 9, 2026

Title A Toolkit for Research Workflows

Version 0.6.0

Description Provides utility functions to help researchers implement best practices for their coding projects. Includes tools for reading and cleaning data files, initializing R projects with a standard folder structure and optional YAML configuration, recording the resolved structure in a project configuration file that downstream tools can read, recording the outputs an analysis produces and the commit that produced them, creating 'Quarto' documents from reproducible templates with optional sample data and custom styling, detecting the execution context across interactive, 'Quarto', and script-based workflows, splitting data frames into group-level output files, applying analysis functions to each group with optional parallel execution, and rendering syntactic tree diagrams as standalone PNG images via 'Typst'.

License MIT + file LICENSE

Depends R (>= 4.2.0)

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

Suggests dplyr, furrr, future, knitr, pdftools, rmarkdown, spelling, quarto, testthat (>= 3.2.0)

Config/testthat/edition 3

Imports cli, fs, glue, here (>= 1.0.0), janitor, jsonlite, lifecycle, parallelly, purrr, readr, renv (>= 1.0.0), rlang, rprojroot (>= 2.0.0), rvest, tibble, tidyr, usethis, utils, withr, xml2, yaml

URL https://github.com/erwinlares/toolero, https://erwinlares.github.io/toolero/

BugReports https://github.com/erwinlares/toolero/issues

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

Config/pak/sysreqs
cmake git make libgit2-dev libicu-dev libuv1-dev libxml2-dev libssl-dev libx11-dev

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

Date/Publication 2026-10-09 18:03:56 UTC

RemoteUrl https://github.com/erwinlares/toolero

RemoteRef HEAD

RemoteSha 36d604d95a8047cb8a4806b96f7c15450b22ece6

Contents

arborize 2
check_project 6
create_qmd 9
detect_execution_context 15
generate_citation 16
generate_data_doc 18
generate_kb_xml 20
generate_license 22
generate_manifest 24
generate_profile 28
generate_project_config 30
init_project 32
qmd_to_r 38
read_clean_csv 40
read_output_records 42
resolve_input_path 45
run_by_group 48
save_output 54
write_by_group 57
write_clean_csv 61

Index **64**

`arborize` *Render a syntactic tree as a PNG image*

Description

Takes a syntactic tree and renders it using Quarto's Typst engine, exporting the result as a PNG image. Supports two rendering backends controlled by `tree_notation`:

Usage

```
arborize(
  tree,
  output = "syntactic-tree.png",
  dpi = 300,
  tree_notation = c("simple", "structured"),
  papersize = "a5",
  margin = "0.5cm",
  provenance = TRUE,
  overwrite = FALSE
)
```

Arguments

tree	A character string. For `tree_notation = "simple"`, a bracket notation string e.g. `"[S [NP] [VP]]"`. For `tree_notation = "structured"`, a lingotree `tree()` call string.
output	A character string. Path to the output PNG file. A relative path is written under the project root (see the Paths section). Defaults to `"syntactic-tree.png"` at the project root.
dpi	A numeric value. Resolution of the output PNG in dots per inch. Defaults to `300`. Use `600` for print-quality output.
tree_notation	A character string. One of `"simple"` (default) or `"structured"`. Controls which Typst rendering backend is used. See Details.
papersize	A character string. Typst paper size for the intermediate PDF. Defaults to `"a5"`. Increase to `"a4"` for very wide trees.
margin	A character string. Page margin for the intermediate PDF. Defaults to `"0.5cm"`. Reduce for tighter crops around the tree.
provenance	A logical. Whether to write a companion `.yml` file recording the tree string and all rendering arguments alongside the PNG. Defaults to `TRUE`. The provenance file has the same name as the PNG but with a `.yml` extension and lives in the same directory. Pass `FALSE` to suppress it. (toolero 0.5.x and earlier used `.yaml`.)
overwrite	A logical. Whether to overwrite existing output files. When `TRUE`, overwrites both the PNG and the provenance file if they exist. Defaults to `FALSE`.

Details

- `"simple"` uses `@preview/syntree` and accepts a bracket notation string, e.g. `"[S [NP [Det the] [N cat]] [VP [V sat]]]"`. This is the most compact input format and suits basic linguistic trees.
- `"structured"` uses `@preview/lingotree` and accepts a nested `tree()` call string. This backend supports per-node styling, movement arrows, and multi-dominant trees.

The function is useful for producing standalone tree figures that can be embedded in any document format – LaTeX, Word, HTML, or presentations – without requiring a full LaTeX installation.

`arborize()` performs the following steps:

1. Validates inputs and resolves the Typst package from `tree_notation`.
2. Builds a minimal `.qmd` document via `.build_arborize_qmd()`.
3. Writes the document and renders it inside a self-cleaning temporary directory managed by `withr::with_tempdir()`.
4. Calls `quarto::quarto_render()` to produce an intermediate PDF via Typst.
5. Converts the PDF to PNG using `pdftools::pdf_convert()`.
6. Reads the PNG bytes into memory before the temporary directory is deleted, then writes them to the specified output path.
7. If `provenance = TRUE`, writes a companion `.yml` file recording the tree string and all rendering arguments.

On first use, Typst will download the required package from the Typst package registry. This requires an internet connection. Subsequent renders use the locally cached package.

Requires Quarto 1.4 or later with Typst support, and the `pdftools` package for PDF-to-PNG conversion. Install `pdftools` with `install.packages("pdftools")`.

Value

Invisibly returns the path to the output PNG file.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.

- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

References

syntree Typst package (v0.2.1): https://typst.app/universe/package/syntree

lingotree Typst package (v1.0.0): https://typst.app/universe/package/lingotree

Examples

```
## Not run:
# Simple bracket notation (default) -- also writes tree-1.yml
arborize("[NP [Det the] [N cat]]", output = "my-trees/tree-1.png")

# Suppress provenance file
arborize("[NP [Det the] [N cat]]", provenance = FALSE)

# Wider tree with print-quality output
arborize(
  paste0(
    "[Aspectual Classes ",
    "[Statives [States]] ",
    "[Dynamic ",
    "[Atelic [Activities]] ",
    "[Telic ",
    "[Instantaneous [Achievements]] ",
    "[Durative [Accomplishments]]]]]"
  ),
  output    = "aspectual-classes.png",
  dpi       = 600,
  papersize = "a4"
)

# Structured notation using lingotree
arborize(
  "tree(
    tag: [VP],
    tree(
      tag: [DP],
      [every],
      [farmer]
    ),
    [smiled]
```

```
  )",
  tree_notation = "structured",
  output        = "vp-tree.png"
)

## End(Not run)
```

`check_project` *Check a project for toolero conventions*

Description

`check_project()` audits a project directory and reports whether it follows the structure and conventions that `init_project()` creates. It is useful both for projects initialized with `init_project()` and for existing projects that were created independently.

Usage

```
check_project(path = ".", config = NULL, error = TRUE)
```

Arguments

`path`	Character. Path to the project directory. A relative path is read from the project root (see the Paths section). Defaults to `"."`, the project root, so calling it from a subfolder audits the project.
`config`	Character or `NULL`. Path to a YAML configuration file produced by `generate_project_config()`. When supplied, the `folders:` list in the file replaces the standard toolero folder set for the folder checks. When `NULL` (the default) and the project carries a `_toolero.yml`, that file is used instead – there is no need to hand `check_project()` the same config on every call. Non-folder hygiene checks always run regardless.
`error`	**[Deprecated]** Logical. Previously controlled whether the function printed a cli report (`TRUE`) or returned a tibble visibly without printing (`FALSE`). Deprecated in v0.5.0 – the cli report now always prints and the tibble is always returned invisibly. Assign the result to access it programmatically: `out <- check_project()`.

Details

Each check records one of four status values: `"pass"` (the expected artifact was found), `"fail"` (a required artifact is missing), `"warn"` (a recommended artifact is missing or a problematic file was found), or `"info"` (a file was found that warrants attention but is not necessarily a problem).

README detection is case-insensitive and extension-agnostic: any file whose stem matches `readme` (in any capitalization) counts, regardless of extension or the absence of one. `init_project()` uses the same detection when deciding whether it would overwrite an existing README.

Value

A tibble with columns `check`, `status`, and `message`, returned invisibly. Assign the result to use it programmatically.

Where the folder set comes from

Three sources, in order of precedence.

An explicit `config` argument wins. Folders it declares and the project lacks are reported as `"fail"`: the caller named a file and that file states what the project should look like.

Failing that, a `_toolero.yml` at the project root is used. `init_project()` writes one recording the structure it actually created, so a folder listed there and missing from disk means something removed it. That is also a `"fail"`.

Failing both, the built-in standard set is used – `data-raw/`, `data/`, `R/`, `scripts/`, `output/figures/`, `output/tables/`, and `reports/`. Missing folders here are `"warn"`, not `"fail"`: nobody declared anything, so the standard set is a suggestion rather than a contract.

A `_toolero.yml` that exists but cannot be parsed is reported as a failing check and the audit continues against the built-in set. A `config` that cannot be parsed is an error, since the caller asked for that file specifically.

The renv checks

Beyond the presence of `renv.lock`, two checks guard the failure mode that costs the most to discover late: a lockfile that does not describe the analysis, which produces a container image that builds cleanly and then cannot run.

A `.renvignore` excluding `.qmd` files is reported, and the advice is to remove the entry. It stops `renv` from seeing the `library()` calls in a project whose Quarto document is the source of truth. That the document will eventually be purled to a `.R` file does not make up for it: the snapshot you containerize from may be taken before the purl, and the `.qmd` is the file being maintained either way. Versions of `init_project()` before v0.5.0 wrote one; projects created by those versions still carry it.

A `renv.lock` recording no packages is reported only when the project also has `.R` or `.qmd` source files. A newly scaffolded project legitimately has an empty lockfile – `renv::scaffold()` does no dependency discovery, because there is nothing yet to discover – so the pairing is what makes the observation worth printing.

Stale purled scripts

Every `.qmd` under `path` whose header declares `purl: true` (see `create_qmd()`'s `use_purl` argument) gets its own row, comparing it against the `.R` script `R/purl.R` is expected to have derived from it. In an ordinary project that script is flat, `reports/analysis.qmd` to `R/analysis.R`. In a Quarto website or book (`project: type:` in `_quarto.yml`) it mirrors the document's path, `posts/one/index.qmd` to `R/posts/one/index.R`. A script found at the other layout's path, which happens when `R/purl.R` is an older copy, is reported with the path it was found at and how to refresh the hook. Missing entirely, or older than the `.qmd` it was purled from, is reported as `"warn"`: the `.qmd` is the source of truth, so an `R/` script older than the document it came from means an edit was made and not yet re-rendered, and a container or cluster job that bakes in the `.R` file would run the old analysis without any error to say so. Documents never opted into purl produce no row – nothing to check.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`init_project()`, `generate_project_config()`

Examples

```
# Audit the current working directory

check_project()
```

```
# Audit a specific project directory

project_dir <- withr::local_tempdir()
check_project(path = project_dir)


# Audit against a custom folder structure

project_dir <- withr::local_tempdir()
config_path <- file.path(tempdir(), "my-config.yml")
generate_project_config("my-config.yml", path = tempdir())
check_project(path = project_dir, config = config_path)


# Access results programmatically

project_dir <- withr::local_tempdir()
out <- check_project(path = project_dir)
```

`create_qmd` *Create a new Quarto document from a template*

Description

Creates a new Quarto document in the specified directory. Optionally copies a sample dataset and a worked analysis example, wires up custom branding assets from a directory of standardized files, and scaffolds a post-render purl hook for extracting R code.

Usage

```
create_qmd(
  filename = NULL,
  path = ".",
  header_defaults = NULL,
  overwrite = FALSE,
  use_purl = FALSE,
  include_examples = TRUE,
  use_style = FALSE,
  yaml_data = lifecycle::deprecated()
)
```

Arguments

`filename`	A string or `NULL`. Name of the generated `.qmd` file, relative to `path`. Must be supplied explicitly, e.g. `"analysis.qmd"` or `"reports/analysis.qmd"`. A folder named in it is created if it does not exist. See the section on documents in subfolders below.

`path` A string. Path to the directory where the document will be created. A relative path is read from the project root (see the Paths section). Defaults to `"."`, the project root.

`header_defaults`

A string or `NULL`. Path to a YAML file (a relative path is read from the project root) supplying values to pre-populate the document header – typically a profile written by `generate_profile()`, but any YAML file following the same shape works. If `NULL` (the default), the template is copied as-is with placeholder prompts intact. Every key in the file, at any depth, replaces the template's key of the same name; keys the file does not mention are left exactly as the template wrote them. A key whose value is itself a mapping (`format: html: ...`) is descended into and merged key by key, so a sibling the file doesn't mention – `css:` from `use_style`, say – survives; a key whose value is a sequence (`author:`, `categories:`) is replaced as a whole, not merged element by element. Named `yaml_data` before v0.5.1.9000; that name still works but is deprecated (see below).

`overwrite` A logical. Whether to overwrite existing files. Defaults to `FALSE`. Note two exceptions: `assets/logo.png` is never overwritten, since an existing logo is assumed to be deliberate branding rather than a stale copy of the placeholder; and `_quarto.yml` is never governed by `overwrite` at all – when it is touched, it is merged rather than replaced, and in some cases (see `use_purl` below) it is left untouched entirely regardless of `overwrite`, on purpose.

`use_purl` Logical. Defaults to `FALSE`. When `TRUE`:

- Stamps the document's own YAML header with `purl: true`.
- Ensures `R/purl.R` exists in `path` (subject to `overwrite`, like any other scaffolded file – an existing `R/purl.R` is left in place unless `overwrite = TRUE`).
- Ensures `path/_quarto.yml` has a `project: post-render:` entry pointing at `R/purl.R` – *unless* `_quarto.yml` already exists and declares `project: type:` as `website`, `book`, or `manuscript`, in which case the hook is deliberately **not** wired automatically. A `cli_warn()` explains why, reports whether `R/purl.R` was created or was already present, and names the `post-render:` entry to add by hand. This guard exists because the failure mode it's protecting against is deciding *whether* to opt a multi-document project in at all, since a website or book renders many documents on every full build and the person scaffolding one `.qmd` may not be thinking about the other twenty. If `_quarto.yml` does not yet exist at all, the package template is copied in as usual (nothing to guard against yet – a fresh `_quarto.yml` with no `type:` is not a multi- document project). Outside the guarded types, an existing `_quarto.yml` gets the hook merged into its existing `project:` block rather than overwritten, so `type`, `website`, and any other project options are left untouched. This merge (when it happens) is unaffected by `overwrite`, since appending one line to `post-render` is non-destructive.

When `use_purl = FALSE`, the document's header is still stamped, with `purl: false`, so `R/purl.R` (in a project where some other document has `use_purl = TRUE`) can positively confirm this document should be skipped rather than merely lacking an opinion.

R/purl.R itself only purls documents whose own header carries purl: true, so turning this on for one document inside a larger project – a Quarto website, a book – does not cause every other .qmd in that project to be purled whenever the project renders in full.

Where the script lands under R/ depends on the project. In an ordinary project it is flat: reports/analysis.qmd purls to R/analysis.R. In a Quarto website or book (project: type: in _quarto.yml) the script mirrors the document's path relative to the project root, so two documents that happen to share a filename in different directories (e.g. a directory-per-post convention using index.qmd) do not overwrite each other's output. Every script starts with a line naming the document it was generated from. In the flat layout the hook leaves alone an existing R/<name>.R that does not carry that line for this document, and warns instead, so a hand-written script, or one generated from a different document with the same filename, is never overwritten.

include_examples

Logical. If TRUE (the default), copies a sample dataset (sample.csv) into data-raw/ and uses a template .qmd pre-populated with a worked analysis example. The YAML header includes a params block referencing the sample data. If FALSE, creates a blank .qmd with only the YAML header and no example content, and skips copying the sample dataset.

A placeholder logo (generic-logo.png, copied as logo.png) is also copied into assets/, but only when branding is actually part of the project: if path carries a _toolero.yml (as written by init_project()) whose folders: list does not include assets (i.e. the project was scaffolded with branding = "none"), the logo is skipped along with it, so a project that declared no branding does not end up with an undeclared assets/logo.png anyway. A .qmd created outside any toolero-scaffolded project (no _toolero.yml at path) always gets the logo, since there is no project-level branding decision to defer to. If assets/logo.png already exists (e.g. from a prior init_project() call with branding set), it is always left untouched – an existing logo takes precedence over the generic placeholder even when overwrite = TRUE.

use_style

Logical or character. Controls whether custom branding assets are wired into the YAML.

- FALSE (the default): no custom styling. The YAML format: html: block contains only standard Quarto options.
- TRUE: shorthand for "assets/". Looks in path/assets/ for styles.css, header.html, and footer.html by name, and wires up whichever of these are present.
- A directory path (e.g. "my-branding/", read from the project root when relative): looks in the given directory for the same three standardized filenames. The caller is responsible for ensuring the directory contains the files it needs under these exact names; create_qmd() does not rename or infer from other file names.

styles.css is added as css:, header.html as include-before-body:, and footer.html as include-after-body:. Any subset may be present; only files that exist are wired into the YAML. If none of the three are found, a warning is issued and style injection is skipped. Note that favicon.png, though shipped

with the branding asset set, is not wired into the document YAML – favicons are a Quarto website-project option rather than an HTML format option, so set it in `_quarto.yml` if you need one.

`yaml_data` **[Deprecated]** A string or `NULL`. Renamed to `header_defaults` in v0.5.1.9000 – the argument still works, and its value is used when `header_defaults` is not also supplied, but new code should use `header_defaults`.

Details

`create_qmd()` performs the following steps:

1. Validates that `filename` is supplied and `path` exists.
2. If `include_examples = TRUE`: creates `data-raw/` under `path` and copies `sample.csv` there. Creates `assets/` if needed and copies the generic placeholder logo as `logo.png`, unless a logo already exists there. Uses the example template for the `.qmd`.
3. If `include_examples = FALSE`: uses the skeleton template for the `.qmd`. No sample data or logo is copied.
4. If `use_style` is `TRUE` or a directory path: looks for `styles.css`, `header.html`, and `footer.html` by name and injects whichever are present into the YAML header.
5. Stamps `purl: true` or `purl: false` into the document's own YAML header, reflecting `use_purl`.
6. If `header_defaults` (or the deprecated `yaml_data`) is provided, reads the YAML file and substitutes its values into the document header, descending into nested mappings so a sibling key it doesn't mention survives. This runs after style injection and the purl stamp, so `header_defaults` can override any auto-generated YAML key, including `purl` itself.
7. If `use_purl = TRUE`, ensures `R/purl.R` exists. Then, unless `_quarto.yml` already exists and declares `project: type:` as `website`, `book`, or `manuscript` (in which case wiring is skipped with a warning explaining why), ensures `_quarto.yml` has the post-render hook – creating `_quarto.yml` from the package template if absent, or merging the hook into the existing file's `project:` block if present.
8. The sample dataset bundled with the template, `data-raw/sample.csv`, is a subset of the Palmer Archipelago penguin data, taken from an earlier version of the `palmerpenguins` package than the one on CRAN today. Treat it as teaching material rather than as a citable copy of the data.

   Since R 4.5.0 the same data ships with base R, so `?datasets::penguins` is the most convenient reference, with `datasets::penguins_raw` carrying the uncleaned form. One difference matters when reading the two side by side: base R shortened four of the column names, so `bill_length_mm`, `bill_depth_mm`, `flipper_length_mm` and `body_mass_g` here are `bill_len`, `bill_dep`, `flipper_len` and `body_mass` there. `species`, `island`, `sex` and `year` are spelled the same in both. This template keeps the longer names, which carry their units.

   Original data: Gorman KB, Williams TD, Fraser WR (2014). Ecological sexual dimorphism and environmental variability within a community of Antarctic penguins (genus Pygoscelis). PLoS ONE 9(3): e90081. doi:10.1371/journal.pone.0090081. R package: Horst AM, Hill AP, Gorman KB (2020). palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. doi:10.5281/zenodo.3960218. Collected by Palmer Station Antarctica LTER, a member of the Long Term Ecological Research Network.

Value

Invisibly returns `path`.

Documents in subfolders

`path` is the project root; `filename` may place the document below it, as in `create_qmd("reports/analysis.qmd")`. The sample data, the placeholder logo, `R/purl.R`, and `_quarto.yml` still go to the project root. Paths in the document follow two rules:

- **Code paths start at the project root.** The example template writes its results to `"output/..."` through toolero functions, and its `params: input_file: "data-raw/sample.csv"` is read from the root by `resolve_input_path()`, so the same code works at the root, in `reports/`, and on an HTCondor execute node.
- **Header paths are relative to the document**, because that is how Quarto resolves them. For a document in `reports/`, `use_style` writes `css: ../assets/styles.css`, and likewise for the two `include-*` entries. When a header include is wired in, the document's `resource-path` also lists the project root, so the logo `header.html` refers to as `assets/logo.png` is found.

Every edit to the document's YAML header is made line by line rather than by parsing the header and writing it back out. Keys the edit does not touch keep the template's own quoting, indentation, comments, and ordering, so the document a reader opens is the template we shipped plus the keys they asked for.

Note: `filename` has no default value and must always be supplied explicitly. Use `tempdir()` for temporary output during testing or exploration.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

Examples

```
# Minimal blank document -- no examples, no styling, no purl
create_qmd(path = tempdir(), filename = "analysis.qmd",
           include_examples = FALSE)

# Full worked example with sample data and placeholder logo
create_qmd(path = tempdir(), filename = "analysis.qmd",
           overwrite = TRUE)

# Opt this document into purl: stamps purl: true and wires up
# R/purl.R + the _quarto.yml post-render hook (merged if the file
# already exists, e.g. inside a larger Quarto website project)
create_qmd(path = tempdir(), filename = "analysis.qmd",
           overwrite = TRUE, use_purl = TRUE)

# Blank document wired to branding assets (assumes assets/ exists,
# e.g. from init_project(branding = "uw-madison"))
create_qmd(path = tempdir(), filename = "report.qmd",
           include_examples = FALSE, use_style = TRUE,
           overwrite = TRUE)

# Blank document with custom branding from a different directory
create_qmd(path = tempdir(), filename = "report.qmd",
           include_examples = FALSE, use_style = "my-branding/",
           overwrite = TRUE)

# Pre-populated YAML header, typically from generate_profile()
profile_file <- tempfile(fileext = ".yml")
writeLines("author:\n  - name: 'Your Name'", profile_file)
create_qmd(path = tempdir(), filename = "analysis.qmd",
           header_defaults = profile_file, overwrite = TRUE)
```

`detect_execution_context` *Detect the current execution context*

Description

Identifies which of three execution environments the code is currently running in: an interactive R session, a `quarto render` call, or a plain `Rscript` invocation. This is useful for writing code that behaves correctly across all three contexts, such as choosing figure dimensions or deciding whether to report progress.

Usage

```
detect_execution_context(interactive_fn = interactive)
```

Arguments

`interactive_fn`	A function. Used to detect whether the session is interactive. Defaults to `base::interactive`. Override in tests to simulate different execution environments.

Details

To resolve an input data path across the same three contexts, use `resolve_input_path()`, which calls this function and then validates what the chosen branch produced.

Detection follows a priority order:

1. If `interactive()` is `TRUE`, returns `"interactive"`.
2. If the environment variable `QUARTO_DOCUMENT_PATH` is set and non-empty, returns `"quarto"`.
3. Otherwise, returns `"rscript"`.

The order is unobservable in practice, and that is worth recording so nobody has to re-derive it. `QUARTO_DOCUMENT_PATH` is set only by Quarto rendering a document, and every path that renders one – `quarto render`, `quarto preview`, `quarto::quarto_render()`, the RStudio Render button – runs the R code in a spawned process that is not interactive. Running chunks inline in RStudio is the reverse case: the session is interactive and the variable is unset (checked in RStudio, September 2026). The two tests therefore never fire together, so there is no case in which the priority arbitrates anything.

Value

A character string, one of `"interactive"`, `"quarto"`, or `"rscript"`.

See Also

`resolve_input_path()` for the input-path case specifically.

Examples

```
# Behave differently depending on where the code is running. Progress
# output is useful at a console and only clutters a cluster job log.
context <- detect_execution_context()

if (context == "rscript") {
  options(cli.progress_show_after = Inf)
}
```

`generate_citation` *Generate a CITATION.cff file*

Description

Writes a Citation File Format (`CITATION.cff`) skeleton, optionally pre-filled with author information from a profile written by `generate_profile()`, so a project's citation metadata doesn't mean retyping the same name, affiliation, and ORCID a third time.

Usage

```
generate_citation(
  filename = "CITATION.cff",
  path = ".",
  profile = NULL,
  overwrite = FALSE
)
```

Arguments

`filename`	Character. Name of the file to write. Defaults to `"CITATION.cff"`, the name tools that read this format expect; changing it means those tools will not find the file automatically.
`path`	Character. Directory to write the file into. Defaults to `"."`, the project root. A relative path is read from the project root (see the Paths section).
`profile`	Character or `NULL`. Path to a profile file (a relative path is read from the project root), typically one written by `generate_profile()`. When supplied, every entry in the profile's `author:` block becomes an author entry in the generated file. When `NULL` (the default), a single generic placeholder author is written instead.
`overwrite`	Logical. When `FALSE` (default), an existing file at the destination is an error rather than being replaced.

Details

`title`, `version`, `repository-code`, `url`, and `license` are project facts a personal profile has no way to know, so they are left as placeholders (some commented out) regardless of `profile`. `date-released` is filled in with today's date, since that much is always knowable at generation time; edit it later if the actual release date differs.

The given-names/family-names split needed by the Citation File Format is done by splitting a profile's single `name` field on its last space, which is right for the ordinary case and wrong for some real names. Review the generated file's `given-names`/`family-names` fields before relying on them, especially for a multi-word family name.

Value

The path to the written file, invisibly.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`generate_profile()`, the function that writes the file this one can read from.

Examples

```
generate_citation(path = tempdir())

profile_file <- tempfile(fileext = ".yml")
writeLines(
  paste(
    'author:',
    '  - name: "Erwin Lares"',
    '    affiliation: "RCI, UW-Madison"',
    '    orcid: "0000-0002-3284-828X"',
    sep = "\n"
  ),
  profile_file
)
generate_citation(path = tempdir(), profile = profile_file, overwrite = TRUE)
```

---

generate_data_doc *Generate a data documentation stub*

Description

Writes a Markdown file documenting a single dataset, with the standard sections a data-management plan asks for – source, date obtained, license and usage terms, collection method, variables, and known issues – pre-filled where they can be, and left as placeholders where they can't.

Usage

```
generate_data_doc(dataset, path = "data", overwrite = FALSE)
```

Arguments

`dataset`	A character string. The file name of the dataset this doc describes, e.g. `"survey_responses.csv"`. Used to name the output file (the extension is replaced with `.md`) and to fill in the doc's title; must be supplied, since there is no reasonable default.
`path`	A character string. Directory in which to write the file. A relative path is read from the project root (see the Paths section). Defaults to `"data"`, the project's `data/` folder.

`overwrite` Logical. If `TRUE`, overwrites an existing doc at the same location. Defaults to `FALSE`.

Details

Recall that a research compendium's `data/` and `data-raw/` folders hold the data itself, but not where it came from or what its columns mean. `generate_data_doc()` fills that gap the same way `generate_citation()` fills in for a missing `CITATION.cff`: a skeleton with a few fields already filled in, ready for you to complete by hand.

Value

Invisibly returns the full path to the written file.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node ~ and $HOME cannot be trusted ($HOME can be the submit node's path), so prefer relative paths in job scripts.

See Also

init_project(), generate_license(), generate_citation()

Examples

```
## Not run:
generate_data_doc("survey_responses.csv")

generate_data_doc("weather_2024.csv", path = "data-raw")

## End(Not run)
```

---

generate_kb_xml *Generate a KB-importable XML file from a Quarto document*

Description

Takes a Quarto document and produces an XML file that is directly importable into a UW-Madison Knowledge Base (KB) article. The function re-renders the .qmd with embed-resources: true so all visual assets are self-contained, extracts the HTML body, and wraps it in the KB XML structure along with metadata drawn from the document's YAML header.

Usage

```
generate_kb_xml(html_path, qmd_path = NULL, output_dir = NULL)
```

Arguments

html_path	A string. Path to the rendered HTML file; a relative path is read from the project root. Used to infer the output filename and, if qmd_path is NULL, the location of the source .qmd.
qmd_path	A string or NULL. Path to the source .qmd file (a relative path is read from the project root). If NULL (the default), inferred by replacing the .html extension of html_path with .qmd.
output_dir	A string or NULL. Directory where the .xml file will be written; a relative path is read from the project root. If NULL (the default), written to the same directory as html_path.

Details

`generate_kb_xml()` performs the following steps:

1. Validates that `html_path` exists.
2. Infers `qmd_path` from `html_path` if not supplied, then validates it.
3. Extracts `title`, `description`, and `categories` from the `.qmd` YAML header and maps them to `kb_title`, `kb_summary`, and `kb_keywords`.
4. Re-renders the `.qmd` in an isolated temporary directory with `embed-resources: true` so all CSS, images, and JS are self-contained. The `data/` and `assets/` folders are copied alongside the `.qmd` to ensure the render succeeds.
5. Extracts the <body> from the embedded HTML.
6. Escapes HTML entities in the body for XML compatibility, as required by the UW-Madison KB import format.
7. Builds the XML structure with `kb_title`, `kb_keywords`, `kb_summary`, and `kb_body` nodes.
8. Writes the `.xml` file to `output_dir`.

Temporary files are managed via `withr::local_tempdir()` and are automatically cleaned up when the function exits, even on error.

When importing the resulting XML into the KB, check the *Decode HTML entity in body content* option.

Value

Invisibly returns the path to the written `.xml` file.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

Examples

```
# Infer qmd_path automatically, write XML alongside the HTML
# generate_kb_xml(html_path = "docs/analysis.html")

# Supply qmd_path explicitly and write to a specific output directory
# generate_kb_xml(
#   html_path  = "docs/analysis.html",
#   qmd_path   = "analysis.qmd",
#   output_dir = "exports"
# )
```

`generate_license` *Generate a LICENSE file*

Description

Writes a plain-text `LICENSE` file at the project root, using one of a small set of common license templates with the copyright holder and year filled in.

Usage

```
generate_license(
  license = "MIT",
  holder = NULL,
  year = format(Sys.Date(), "%Y"),
  path = ".",
  overwrite = FALSE
)
```

Arguments

license	A character string. One of "MIT", "CC0", or "GPL-3". Only this small, common set is supported; write a LICENSE file by hand for anything else. Defaults to "MIT".
holder	A character string. The copyright holder – a person or an institution. Must be supplied explicitly; there is no default, since guessing it wrong is worse than asking.
year	A character string. The copyright year. Defaults to the current year.
path	A character string. Directory in which to write the file. A relative path is read from the project root (see the Paths section). Defaults to ".", the project root.
overwrite	Logical. If TRUE, overwrites an existing LICENSE file at the same location. Defaults to FALSE.

Details

Note that this is different from how an R package normally carries a license. A package's DESCRIPTION declares License: MIT + file LICENSE and pairs a two-line CRAN stub (LICENSE) with the full text in a separate LICENSE.md. The projects init_project() scaffolds are research compendia, not packages – there is no DESCRIPTION to reference a license from – so generate_license() instead writes one self-contained file carrying the license text directly, the way a plain GitHub repository does.

Value

Invisibly returns the full path to the written file.

GPL-3

The GNU General Public License's full text runs to several hundred lines. Rather than risk an inaccurate transcription of it, license = "GPL-3" writes the license notice the Free Software Foundation itself recommends attaching to a program, together with a link to the canonical full text at https://www.gnu.org/licenses/gpl-3.0.txt. "MIT" and "CC0" are both short enough to reproduce here in full, and are written that way.

Paths

A relative path is read from the project root, not from the working directory. So save_output(fit, "output/fit.rds", saveRDS) writes into the project's own output/ folder whether you call it from the project root, from a document under reports/, or from a job on an HTCondor execute node. You never need to wrap a path in here::here(). Doing so still works, because here::here() returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when _CONDOR_SCRATCH_DIR names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the .here file init_project() writes, an .Rproj file, a package DESCRIPTION, or a version control folder such as .git.

3. The working directory, when neither of the above applies.

Absolute paths (including those starting with ~, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node ~ and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`init_project()`, `generate_data_doc()`

Examples

```
## Not run:
generate_license(license = "MIT", holder = "Jane Researcher")

generate_license(license = "CC0", holder = "Example Lab",
                 path = "~/Documents/my-project")

## End(Not run)
```

`generate_manifest` *Write the output record*

Description

`generate_manifest()` reads the accumulator written by `save_output()` over the course of an analysis, collapses it to one row per output file, and writes the output record, `project-manifest.json`, describing every artifact the project produced.

Usage

```
generate_manifest(
  output_dir = NULL,
  filename = "project-manifest.json",
  overwrite = FALSE,
  config = NULL,
  git_root = "."
)
```

Arguments

output_dir	Character or NULL. Directory containing accumulator.csv and receiving the output record. An explicit relative value is read from the project root (see the Paths section); an absolute one is used as given. If NULL (the default), resolved from config's output_dir convention when config is supplied, or "output" otherwise, exactly as save_output() resolves it, so the two always meet at the same accumulator.
filename	Character. Name of the output record file. Defaults to "project-manifest.json".
overwrite	Logical. When FALSE (default), an existing output record at that path is an error rather than being replaced.
config	Character or NULL. Path to a project configuration file (typically a project's own _toolero.yml, as written by init_project()). Only consulted when output_dir is not supplied; an explicit output_dir always wins. Defaults to NULL.
git_root	Character. Directory to check for a git commit to record in the output record (see the Provenance section below). A relative path is read from the project root. Defaults to ".", the project root.

Details

The output record opens with schema_version, then holds execution_context, generated_at, and commit once at the top level, followed by an artifacts array with one entry per output file, ordered chronologically. Context, generation time, and commit are facts about the run as a whole rather than about any individual artifact, so they are not repeated per entry. Package and R versions are deliberately absent: that is renv's job, and duplicating it here would create a second record to keep in sync.

Field names are toolero's own rather than RO-Crate vocabulary. The translation to @id, dateCreated, and the rest belongs in encapsulr::describe() as a thin mapping layer, so that toolero's public interface does not inherit a downstream package's data model.

Checksums are likewise excluded. RO-Crate defers fixity to BagIt and OCFL, and rocrateR::bag_rocrate() computes manifest-sha512.txt automatically at bagging time.

A missing accumulator is an error: no save was ever recorded, and an empty output record would present that as a finished result. An accumulator holding no rows is different – the file exists, so the machinery was wired up – and produces an empty output record with a warning.

Value

The path to the output record, invisibly.

Provenance

The output record also holds `commit`: the git commit checked out in `git_root` at the moment the record was written, or `null` when the project is not a git repository, has no commits yet, or `git` is not installed. This is deliberately the one piece of "which version of the code produced this" that package versions cannot supply – `renv.lock` already answers which package versions were in play, but nothing else records which revision of the analysis script itself ran. Like `execution_context` and `generated_at`, it describes the run as a whole and is not repeated per artifact.

`config` is entirely opt-in and affects `output_dir` only, not `commit`. Without it, `output_dir` defaults to `output/` under the project root. When `config` is supplied but cannot be read, this aborts with the same message `init_project()` gives for a bad `config`, rather than silently falling back to `"output"`.

Format

The output record is a single JSON object with these keys, in this order:

- `schema_version` – integer, currently `1`.
- `execution_context` – `"interactive"`, `"quarto"`, or `"rscript"`, as returned by `detect_execution_context()`.
- `generated_at` – when the record was written, in UTC with millisecond precision (`"2026-09-29T18:04:12.345Z"`).
- `commit` – a 40-character git commit SHA, or `null`.
- `artifacts` – an array, empty rather than absent when nothing was saved. Each entry carries the seven accumulator fields: `file_path` (relative to the project root when the file is inside the project, as given to `save_output()` otherwise), `r_class` (the object's classes joined with `"|"`), `timestamp` (same format as `generated_at`), `function_used`, `status` (`"success"` or `"failure"`), `error_message`, and `note`. A field with no value is written as `null`, never as an empty string.

`schema_version` increments only when an existing key is removed, renamed, or changes meaning or type. A record with no `schema_version` was written by toolero 0.5.x and has the version 1 shape without the key. The full specification, including the rules for readers, is in the family's `CONVENTIONS.md`.

The output record is toolero's own format. Read it with `read_output_records()`; other packages should treat the file as opaque rather than parse it.

The output record and the job manifest

This is the *output record*: a record of outputs from a computation that has already happened. It is distinct from the *job manifest* produced by `write_by_group()` and consumed by `submitr::htc_gen_submit()`, which lists inputs to a computation about to happen. The file name, `project-manifest.json`, and this function's name predate the family's vocabulary and are kept for compatibility; the file defaults to `project-manifest.json` rather than `manifest.json` so the two documents cannot be confused on disk.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`save_output()`, which feeds the accumulator, and `read_output_records()`, which reads the output record back.

Examples

```
output_dir <- withr::local_tempdir()

save_output(
  object = mtcars,
```

```
  file_path = fs::path(output_dir, "mtcars.rds"),
  .f = saveRDS,
  output_dir = output_dir
)

generate_manifest(output_dir = output_dir)
```

`generate_profile` *Generate a personal defaults file for create_qmd()*

Description

Writes a YAML skeleton, pre-filled with placeholders and explanatory comments, covering the author information and formatting preferences that tend to be identical across every document you create with `create_qmd()`. Edit the result with your own values, then pass its path to `create_qmd(header_defaults = )` instead of retyping the same author block and format options on every call.

Usage

```
generate_profile(filename, path = fs::path_home(), overwrite = FALSE)
```

Arguments

`filename`	Character. Name of the file to write, e.g. `"profile.yml"` or a personal name like `"erwinlares.yml"`. Must be supplied explicitly – there is no default, so that keeping more than one profile (a personal one and a work one, say) under different filenames is a normal thing to do, not a workaround.
`path`	Character. Directory to write the file into. Defaults to the user's home directory (`fs::path_home()`), not `"."` – unlike `generate_project_config()`, this file's whole purpose is being reusable across every project rather than tied to one, so it belongs somewhere that outlives any single project directory. Pass `path = "."` if you'd rather keep a copy at the root of a specific project instead. A relative path is read from the project root (see the Paths section).
`overwrite`	Logical. When `FALSE` (default), an existing file at the destination is an error rather than being replaced.

Details

The written file has two sections. Personal information becomes the document's `author:` block: name, affiliation, ORCID, email, and a website URL, the fields Quarto's default HTML title block already knows how to render. Document settings covers the recurring, non-personal choices `create_qmd()` doesn't otherwise remember for you: `date`/ `date-modified`, `categories`, `lang`, `execute` options for quiet and reproducible rendering, and a `format: html:` block of layout preferences. Neither section touches `css`, `include-before-body`, or `include-after-body` – those are `create_qmd()`'s `use_style` argument's job, reading from a project's own `assets/` folder, and a profile that also tried to set them would collide with a specific project's branding rather than complementing it.

The file is a plain copy of the packaged template, not a form filled in programmatically – this function does not prompt for values. Open the written file, replace the placeholder values with your own, and delete any key you don't want pre-filled; `create_qmd()` leaves a deleted key's template placeholder untouched.

A phone number and mailing address are deliberately not among the placeholders. Documents built from this file are the kind that tend to get rendered to HTML and published, and neither belongs in something public.

Value

The path to the written file, invisibly.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

create_qmd(), whose header_defaults argument reads the file this writes; generate_citation(), which can also draw on it.

Examples

```
generate_profile("my-profile.yml", path = tempdir())
```

---

generate_project_config

Generate a project configuration file

Description

Writes a YAML configuration file pre-filled with the standard toolero folder structure and naming conventions. Edit the file to define a custom project layout, then pass its path to init_project() via the config argument.

Usage

```
generate_project_config(filename, path = ".", overwrite = FALSE)
```

Arguments

filename	A character string. Name of the YAML file to create (e.g., "linguistics-project.yml"). Must be supplied explicitly.
path	A character string. Directory in which to write the file. A relative path is read from the project root (see the Paths section). Defaults to ".", the project root. Consider using "~" (your home directory) so the file is easy to reference in future init_project() calls regardless of which project is active.
overwrite	Logical. If TRUE, overwrites an existing file at the same location. Defaults to FALSE.

Details

The file uses the same schema as the _toolero.yml that init_project() writes into a project, so a config you author by hand and the project config a project carries are the same kind of document. The only difference is who wrote it.

Value

Invisibly returns the full path to the written file.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`init_project()`, `check_project()`

Examples

```
## Not run:
# Write to the current working directory
generate_project_config("my-project.yml")

# Write to home directory for easy reuse across projects
```

```
generate_project_config("linguistics-project.yml", path = "~")

# Overwrite an existing config
generate_project_config("my-project.yml", overwrite = TRUE)

## End(Not run)
```

`init_project` *Initialize a new R project with a standard folder structure*

Description

`init_project()` creates a new R project at the given path with an opinionated folder structure suited for research workflows. It optionally initializes `renv` for package management and git for version control, and records the structure it resolved in a project config, `_toolero.yml`, at the project root.

Usage

```
init_project(
  path,
  use_renv = TRUE,
  use_git = TRUE,
  custom_folders = NULL,
  config = NULL,
  open = FALSE,
  branding = "none",
  uw_branding = deprecated(),
  use_readme = TRUE,
  use_rprofile = FALSE,
  scaffold_fn = renv::scaffold
)
```

Arguments

`path`	A character string with the path and name of the new project (e.g., `"~/Documents/my-project"`).
`use_renv`	Logical. If `TRUE`, initializes `renv` in the new project. Defaults to `TRUE`.
`use_git`	Logical. If `TRUE`, initializes a git repository in the new project. Defaults to `TRUE`.
`custom_folders`	A character vector of folder names to add to or remove from the project structure after the base set is resolved. Bare names (e.g., `"models"`) add a folder. Names prefixed with `"-"` (e.g., `"-output/figures"`) suppress creation of that folder. When removing from the built-in default set, only the named leaf is suppressed – parent directories are preserved, so `"-output/figures"` still leaves an `output/` folder behind. When removing from a set supplied via `config`, parents are not preserved: a config is an explicit and complete statement of the structure, so nothing is added back that the author did not ask for. Duplicates of existing folders generate a message and are skipped. References to non-existent folders

via "-" generate a warning. "-R" is the one removal that does not take effect on disk: see config below. Defaults to NULL.

config — A character string. Path to a YAML project config file produced by generate_project_config(). When supplied, the folder list in the config replaces the built-in standard structure entirely, and any conventions: it declares override the defaults key by key. custom_folders is still applied on top of the config-derived set. Defaults to NULL.

One folder cannot be suppressed, by a config or by custom_folders: R/. usethis::create_project() creates it unconditionally, so it is present in every project init_project() makes. A structure that leaves it out is honored everywhere else – R/ is absent from the project config, gets no .gitkeep, and is not audited by check_project() – but the directory itself is there.

open — Logical. If TRUE, opens the new project in RStudio after creation. Defaults to FALSE.

branding — Character or logical. Controls whether an assets/ folder is created and populated. TRUE populates it with generic placeholder branding files (logo.png, favicon.png, header.html, footer.html, styles.css). "uw-madison" populates it with UW-Madison RCI branding files under the same standardized names. "none" or FALSE creates no assets/ folder. Defaults to "none". When branding is enabled, assets/ joins the project's folder set and is recorded in the project config alongside every other folder, so downstream packages can find the branding files without being told about them separately. Note that favicon.png is included in the asset set but is not automatically wired into Quarto output – favicons are a website-project option set in _quarto.yml rather than a per-document HTML option.

uw_branding — **[Deprecated]** Use branding instead. uw_branding = TRUE now maps to branding = "uw-madison"; uw_branding = FALSE maps to branding = "none".

use_readme — Logical or character. Controls whether a README file is created at the project root. TRUE creates README.md from the generalist toolero template. FALSE creates no README file. "plain" creates README.txt with the same generalist content as README.md – only the extension differs, not the content. Defaults to TRUE.

use_rprofile — Logical or character. Controls whether the project's .Rprofile also sources a personal one, so a project under renv does not silently shadow the user's own startup customizations. TRUE appends a guarded block that sources ~/.Rprofile if it exists. A character string names a specific file to source instead – useful for a personal profile kept somewhere other than the platform default, such as one tracked in a dotfiles repository (e.g. use_rprofile = "~/dotfiles/rprofile"). FALSE adds nothing. If the named file does not exist yet, a warning is issued when init_project() runs, but the guard is still written – see the "Personal .Rprofile and renv" section below. Defaults to FALSE.

scaffold_fn — A function. Called as scaffold_fn(project = path) when use_renv = TRUE, in place of calling renv::scaffold() directly. Defaults to renv::scaffold. Exists for the same reason detect_execution_context()'s interactive_fn does: so tests can substitute a fake and exercise use_renv = TRUE without loading renv's namespace into the test process. Overriding it outside of tests is unsupported.

Value

Called for its side effects. Invisibly returns `path`.

The project config

[Experimental]

`init_project()` writes `_toolero.yml` to the project root, recording the folder set it resolved and the naming conventions in force. The file records the *resolved* structure, never the inputs that produced it, so a project built from a `config`, one built with `custom_folders`, and one built from the defaults all produce the same shape of file and a reader never has to replay anything to learn what the project looks like.

It exists because the structure is configurable. `check_project()` can audit a customized project without being handed the same config again, and `containr` and `submitr` can resolve where code, data, and outputs live rather than assuming. Commit the file: it describes the project, not the machine it was created on.

The format is experimental and may gain keys before it settles. The `schema_version` field exists so that a reader can tell whether it understands what it is holding.

The `.here` marker

`init_project()` also writes an empty `.here` file at the project root. It marks the project root for toolero's own functions, which read a relative path from there (see the Paths section), and for `here::here()` in any code of yours that still calls it. An `.Rproj` file or a `.git` folder would serve too, but a project created outside RStudio and without git has neither, so `.here` guarantees a marker. Commit it. It is not needed on an execute node, where toolero takes the job's scratch directory as the root. An existing `.here` is left as it is.

Empty folders and git

Each folder `init_project()` creates that is still empty when the call finishes receives a zero-byte `.gitkeep`.

git tracks files rather than directories, so without this a scaffolded structure survives nothing: the opening commit contains the files at the project root and none of the layout, and a collaborator cloning the repository gets a project with no folders in it. The placeholders are written whether or not `use_git = TRUE`, since a project can be git-initialized at any point afterwards.

Folders that already have content are left alone – `assets/` holds branding files by then and is tracked on the strength of those.

The active project

`init_project()` makes the new project the active `usethis` project for the duration of the call, and restores whichever project was active before when it returns. Nothing is left pointing somewhere the caller did not ask for.

This matters more than it sounds. `usethis::create_project()` sets the active project only for its own duration – it uses `usethis::local_project()` internally and restores the caller's project on exit when `open = FALSE`. Any step that resolves paths through the active project therefore has to set it again explicitly. In v0.5.0 and earlier `init_project()` did not, so with `use_git = TRUE` the

git initialization and its opening commit ran against whatever project happened to be active in the calling session rather than the project just created.

Dependency discovery and renv

When use_renv = TRUE, init_project() calls renv::scaffold(), which creates renv/library, renv/activate.R, renv/.gitignore, an .Rprofile that activates the project in future sessions, and an initial renv.lock.

The call is made through the injectable scaffold_fn argument rather than by name, purely so tests can substitute a stand-in and never load renv's namespace into the test process. Loading it there is what broke covr::package_coverage(): renv's load hook takes over .libPaths(), and every package a test reaches with :: afterwards (rather than one already loaded) becomes unresolvable. Two environment variables papered over the symptom before this argument existed; see covr-renv-incident.md for the incident this closes out.

Three things changed here in v0.5.0, and they are worth understanding together.

renv::scaffold() replaces renv::init(). init() loads the new project into the *calling* session, repointing .libPaths() at a library that is empty apart from renv itself – so every package the caller had available vanishes until they restart R. Its restart argument suppresses the restart, not the activation. That is reasonable behavior for someone adopting renv in the project they are sitting in, and the wrong behavior for a function whose job is to scaffold a project somewhere else. scaffold() builds the same infrastructure and leaves the caller's session untouched.

The .renvignore containing *.qmd is gone. It excluded Quarto documents from renv's dependency discovery, which meant that a project whose library() calls live in its .qmd source – the arrangement this package recommends – could snapshot a lockfile with none of the analysis packages in it, and containr::generate_dockerfile() would then build an image that could not run the analysis. At the point the file was written the project contained no .qmd files at all, so it never affected the snapshot taken at creation time; its only effect was on every snapshot the user took afterwards.

The snapshot at creation time is gone too, for the same underlying reason the .renvignore was pointless there: a project that has just been created has no code in it, so there is nothing to discover and nothing worth recording. Take a snapshot yourself once the project has code, and before containerizing:

```
renv::snapshot()
```

Personal .Rprofile and renv

R reads exactly one .Rprofile per session: the project's own if the working directory has one, a personal one only if it does not. When use_renv = TRUE, renv::scaffold() writes a project .Rprofile containing source("renv/activate.R"), and from that point on a user's own startup customizations – aliases, options, personal helper functions – are shadowed for every session opened in this project. Nothing warns about this; it simply stops loading.

use_rprofile = TRUE appends a guarded block to the project's .Rprofile, after renv's own activation line, that sources ~/.Rprofile if it exists. Pass a character string instead of TRUE to source a different file – for instance a profile tracked in a dotfiles repository rather than kept at the platform default location. Either way, the existence check happens inside the written block, at every session start, rather than being baked in once, so a file written or edited after the project is created is still

picked up. If the named file does not exist yet when `init_project()` itself runs, a warning says so – most often a sign of a typo – but the guard is written regardless, since the file may simply not exist yet.

R has no mechanism for merging profiles, so the two files stack rather than combine: the personal file runs in the same session, top to bottom, at the point where the block appears. Settings that do not overlap coexist without trouble. If both files set the same option (`repos`, for example), the later line wins, and because the block is appended at the end of the project `.Rprofile`, the personal value overrides anything set above it, including values renv established during activation. A personal profile that must not touch such options inside a project can guard them, for instance with `if (!file.exists("renv.lock"))`.

The personal file runs under renv's restricted library paths. Packages it attaches or relies on – `devtools`, or a clipboard helper that uses `clipr` – are found only if they are installed in the project library, so a startup line such as `require(devtools)` may warn, and a helper that depends on an optional package should fall back gracefully when that package is absent.

Defaults to `FALSE` because it cuts against renv's own isolation goal: a project that automatically re-sources a personal environment is no longer fully isolated from it. The `TRUE` mode is also written generically – the block sources whichever file sits at `~/.Rprofile` for whoever opens the project, not a specific person's file – so a collaborator who clones the project gets the same behavior the project's original author chose, rather than one tied to a specific person's home directory. A character path is written into the guard exactly as given, so the same portability holds as long as the path itself is written portably (~-relative, say, rather than an absolute path tied to one machine).

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`check_project()`, `generate_project_config()`

Examples

```
## Not run:
init_project(path = file.path(tempdir(), "project1"),
             use_renv = FALSE, use_git = FALSE)

# Generic placeholder branding
init_project(path = file.path(tempdir(), "project2"),
             branding = TRUE, use_renv = FALSE, use_git = FALSE)

# UW-Madison RCI branding
init_project(path = file.path(tempdir(), "project2b"),
             branding = "uw-madison", use_renv = FALSE, use_git = FALSE)

# Add a folder and suppress one from the standard set
init_project(path = file.path(tempdir(), "project3"),
             custom_folders = c("models", "-output/figures"),
             use_renv = FALSE, use_git = FALSE)

# Drive structure entirely from a config file
init_project(path = file.path(tempdir(), "project4"),
             config = "~/linguistics-project.yml",
             use_renv = FALSE, use_git = FALSE)

# Plain-text README instead of Markdown (same content, README.txt)
init_project(path = file.path(tempdir(), "project5"),
             use_readme = "plain", use_renv = FALSE, use_git = FALSE)

# Skip the README entirely
init_project(path = file.path(tempdir(), "project6"),
             use_readme = FALSE, use_renv = FALSE, use_git = FALSE)

# Keep loading your own ~/.Rprofile customizations under renv (the
# scenario this argument exists for -- renv's own generated .Rprofile
```

```
# would otherwise shadow ~/.Rprofile entirely)
init_project(path = file.path(tempdir(), "project7"),
             use_rprofile = TRUE, use_git = FALSE)

# Source a personal profile kept somewhere other than ~/.Rprofile
init_project(path = file.path(tempdir(), "project8"),
             use_rprofile = "~/dotfiles/rprofile", use_git = FALSE)

## End(Not run)
```

---

qmd_to_r *Extract R code from a Quarto document*

Description

`qmd_to_r()` extracts R code chunks from a `.qmd` file and writes them to a standalone `.R` script using `knitr::purl()`. It works on any `.qmd` file regardless of whether it was created with `create_qmd()`.

Usage

```
qmd_to_r(input, output = NULL, documentation = 1L, quiet = TRUE)
```

Arguments

`input`	A character string with the path to the `.qmd` file. A relative path is read from the project root (see the Paths section).
`output`	A character string with the path to the output `.R` file; a relative path is written under the project root. If `NULL` (the default), the output file is written to the same directory as `input` with the `.qmd` extension replaced by `.R`.
`documentation`	An integer controlling how much documentation is included in the extracted script. Passed to `knitr::purl()`: `0` strips all documentation; `1` (the default) includes chunk labels as comments; `2` includes full roxygen blocks.
`quiet`	Logical. If `TRUE` (the default), suppresses knitr's own output. toolero provides its own cli feedback instead.

Details

The parent directory of `output` is created if it does not already exist, and the creation is reported. `knitr::purl()` does not do this itself, so writing a derived script into `R/` from a project that was not created by `init_project()` would otherwise fail.

Value

Invisibly returns the path to the output `.R` file.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

Examples

```
# Extract R code from a qmd file
qmd <- tempfile(fileext = ".qmd")
writeLines(c(
  "---",
  "title: Analysis",
  "---",
  "",
  "```{r}",
  "x <- 1 + 1",
```

```
  "```"
), qmd)

# Default output path: same directory, .R extension
qmd_to_r(input = qmd)

# Explicit output path. R/ is where toolero expects derived scripts;
# the directory is created if it does not exist yet.
out <- tempfile(fileext = ".R")
qmd_to_r(input = qmd, output = out)

# Strip all documentation
qmd_to_r(input = qmd, output = out, documentation = 0L)
```

`read_clean_csv` *Read and clean a CSV file*

Description

`read_clean_csv()` reads a CSV file, standardizes column names, optionally handles missing values, and optionally prints an ingest summary. It combines `readr::read_csv()`, `janitor::clean_names()`, and `tidyr::drop_na()` into a single, reproducibility-friendly step.

Usage

```
read_clean_csv(
  path,
  na = c("", "NA"),
  drop_na = FALSE,
  summary = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`path`	A character string with the path to the CSV file. A relative path is read from the project root (see the Paths section).
`na`	A character vector of strings to treat as missing values. Passed directly to `readr::read_csv()`. Defaults to `c("", "NA")`, which matches `readr`'s own default behavior.
`drop_na`	Logical or character vector. If `FALSE` (the default), no rows are dropped. If `TRUE`, drops all rows containing any missing value. If a character vector of column names, drops only rows with missing values in those columns. Always emits a cli message reporting how many rows were dropped and how many remain.

summary	Logical. If TRUE, prints a brief ingest summary after reading and cleaning: row and column counts, number of column names cleaned, and missing value totals. Reflects the final state of the tibble after any drop_na action. Defaults to FALSE.
verbose	Logical. If TRUE, displays column type messages from readr::read_csv(). Defaults to FALSE.
...	Additional arguments passed to readr::read_csv(), such as col_types, skip, or locale.

Value

A tibble with cleaned column names.

Paths

A relative path is read from the project root, not from the working directory. So save_output(fit, "output/fit.rds", saveRDS) writes into the project's own output/ folder whether you call it from the project root, from a document under reports/, or from a job on an HTCondor execute node. You never need to wrap a path in here::here(). Doing so still works, because here::here() returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when _CONDOR_SCRATCH_DIR names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the .here file init_project() writes, an .Rproj file, a package DESCRIPTION, or a version control folder such as .git.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with ~, or a drive letter on Windows) and URLs such as https:// or s3:// are used as given. .. and ./ are not interpreted, so "../data/x.csv" means the folder above the root; write an absolute path to leave the project. A bare "." means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- init_project() reads path, config and use_rprofile from the working directory, because the project does not exist yet and so has no root.
- Under Rscript, resolve_input_path() reads the command line argument as given. Run Rscript from the project root or pass an absolute path.
- Arguments passed on through ... to another function reach it untouched.
- Where a default is "beside the input" (qmd_to_r() and generate_kb_xml()), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node ~ and $HOME cannot be trusted ($HOME can be the submit node's path), so prefer relative paths in job scripts.

Examples

```
sample_path <- system.file("templates", "sample.csv", package = "toolero")

# Basic usage
data <- read_clean_csv(sample_path)

# Explicit missing-value codes
data <- read_clean_csv(sample_path, na = c("", "NA", "N/A", ".", "-999"))

# Drop rows missing in any column
data <- read_clean_csv(sample_path, drop_na = TRUE)

# Drop rows missing in specific columns
data <- read_clean_csv(sample_path, drop_na = c("bill_length_mm", "sex"))

# Print ingest summary
data <- read_clean_csv(sample_path, summary = TRUE)

# Combine arguments
data <- read_clean_csv(
  sample_path,
  na      = c("", "NA", "N/A", "."),
  drop_na = TRUE,
  summary = TRUE
)
```

`read_output_records` *Read one or more output records*

Description

[Experimental]

Reads the output record, `project-manifest.json`, that `generate_manifest()` writes at the end of an analysis, and returns what it describes as a table with one row per artifact. Given several output folders, it reads each and stacks the results, so that the outputs of a multi-job run (one folder per job, as `submitr::htc_collect()` leaves them) can be inspected as one table.

`read_output_records()` is the reader that goes with `generate_manifest()`. The output record is toolero's own format, and this is the one place that interprets it, so other code (including other packages) can read the table this returns rather than parse the file.

Usage

```
read_output_records(
  path = NULL,
  filename = "project-manifest.json",
  config = NULL
)
```

Arguments

path	Character vector or NULL. Output folders, each holding a project-manifest.json, an accumulator.csv, or both; or paths to output record files themselves; relative paths are read from the project root (see the Paths section). A named vector labels each folder's rows in the source column with its name; an unnamed one (or an element with an empty name) uses the path as you typed it. If NULL (the default), reads the project's own output folder, resolved exactly as save_output() and generate_manifest() resolve it.
filename	Character. Name of the output record within each folder. Defaults to "project-manifest.json". Ignored for elements of path that are files rather than folders.
config	Character or NULL. Path to a project configuration file (typically a project's own _toolero.yml). Only consulted when path is NULL, to find the project's output folder from its output_dir convention. Defaults to NULL.

Value

A tibble with one row per artifact and these columns, in order:

- source – the name given to the folder in path, or the path itself.
- file_path, r_class, timestamp, function_used, status, error_message, note – the seven artifact fields, exactly as recorded (see generate_manifest()). All character; timestamps are kept as the strings written, and a null in the record becomes NA.
- read_from – "output record", or "accumulator" when the rows came from the fallback described below.
- schema_version – integer; NA for rows read from an accumulator.
- execution_context, generated_at, commit – facts about the run as a whole, repeated on each of its rows; NA for rows read from an accumulator, which does not record them.

An output record with no artifacts contributes no rows. When nothing at all can be read, the result is a tibble with these columns and no rows.

Versions

A record written by toolero 0.5.x carries no schema_version and is read as version 1, which is the shape it has. A record with a version this toolero does not know is read as far as possible, with a warning: the fields listed above are taken where they exist and have a single value, and anything else is ignored. The rules are the family's, set out in its CONVENTIONS.md.

When there is no usable output record

An analysis that stopped before generate_manifest() ran, or whose record cannot be parsed, may still have left its accumulator behind. For each folder, read_output_records() reads the output record when it can; otherwise it reads accumulator.csv from the same folder, keeping the latest row per file_path exactly as generate_manifest() would, and warns that it did so. A folder with neither contributes no rows, also with a warning. One incomplete folder therefore never stops the others from being read. A path that does not exist at all is an error, since that is a mistake in the call rather than a fact about a run.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`generate_manifest()`, which writes the output record, and `save_output()`, which feeds it.

Examples

```
output_dir <- withr::local_tempdir()

save_output(
  object = mtcars,
  file_path = fs::path(output_dir, "mtcars.rds"),
```

```
  .f = saveRDS,
  output_dir = output_dir
)
generate_manifest(output_dir = output_dir, git_root = output_dir)

read_output_records(output_dir)

## Not run:
# Several jobs brought back by submitr: label each job's rows by group.
jobs <- submitr::htc_collect()
jobs <- jobs[!is.na(jobs$output_dir), ]
records <- read_output_records(stats::setNames(jobs$output_dir, jobs$group_id))

## End(Not run)
```

`resolve_input_path` *Resolve the input data path for the current execution context*

Description

[Experimental]

Picks an input file path according to where the code is running, then checks that the result is usable and explains what went wrong when it is not. It is the pattern this family of packages recommends everywhere, packaged so that it lives in one place rather than being retyped into every document.

Usage

```
resolve_input_path(
  interactive = NULL,
  quarto = NULL,
  rscript = NULL,
  must_exist = TRUE,
  context = detect_execution_context()
)
```

Arguments

`interactive`	The path to use in an interactive session. Typically the local development copy of the data. If not supplied, falls back to the document's own `params$input_file`, when the document declares one.
`quarto`	The path to use while Quarto renders the document. Normally `params$input_file`, which is also what is used when this argument is not supplied.
`rscript`	The path to use under `Rscript`, normally the first command line argument. Defaults to exactly that.
`must_exist`	Logical. Whether to check that the resolved path exists on disk. Defaults to `TRUE`. Set it to `FALSE` when the path is a URL, a database handle, or anything else `fs::file_exists()` cannot see.

`context` Character. The execution context to resolve for, one of `"interactive"`, `"quarto"`, or `"rscript"`. Defaults to `detect_execution_context()`. Supply it directly in tests, or when the caller has already computed it and does not want a second call.

Details

Each of the three arguments is an ordinary R argument and therefore a promise, so only the branch matching `context` is ever evaluated. Under `Rscript`, where `params` does not exist, passing `quarto = params$input_file` is safe because that expression is never forced.

The reason this is a function rather than a documented `switch()` follows from the same mechanism. When a branch *is* selected and its expression fails, the failure happens inside this function, where it can be caught and explained. A document that declares no `params:` block raises `object 'params' not found`, which says nothing about YAML headers; a hand-written `switch()` in the document evaluates that expression in the document's own frame, where nothing is in a position to intercept it.

The three ways a branch produces something unusable, and what this function says about each:

- Under `Rscript` with no argument passed, `commandArgs(trailingOnly = TRUE)[1]` is `NA_character_`. Reading that produces an error about `NA` rather than about a missing argument.
- Under Quarto, `params` exists only if the YAML header declares it, and `params$input_file` is `NULL` if the block exists without that key.
- In any context, the resolved path may simply not be there, which is most often a working directory that is not what the author assumed.

Value

A single character string: the resolved path. In the `interactive` and `quarto` contexts, a relative path is returned joined to the project root (see the section below).

Relative paths start at the project root

In the `interactive` and `quarto` contexts, a relative path such as `"data-raw/sample.csv"` is read from the project root, not from the working directory. So a document under `reports/` can declare `input_file: data-raw/sample.csv` in its header and find the file whether it is rendered (Quarto runs the code in `reports/`) or run chunk by chunk in RStudio. How the root is found, and the fallback for documents written before this rule, are described in the Paths section below.

The `rscript` context is left alone. On an HTCondor execute node the path arrives as a command line argument that already points at the file: an absolute path to data baked into the image, or a bare subset file name in the job's scratch directory.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`detect_execution_context()`, which decides the branch.

Examples

```
# Resolving for a named context. must_exist = FALSE because there is no
# such file here; in a real document you want the default.
resolve_input_path(
  rscript    = "data-raw/sample.csv",
  context    = "rscript",
  must_exist = FALSE
)

## Not run:
# A path per context, which is what a scaffolded document shows. Only
# the branch matching the current context is evaluated, so the params
# reference is safe under Rscript, where params does not exist.
```

```
input_file <- resolve_input_path(
  interactive = "data-raw/sample.csv",
  quarto      = params$input_file,
  rscript     = commandArgs(trailingOnly = TRUE)[1]
)

# The defaults cover the mechanical branch and fall back to the
# document's own params for the other two, so a document whose YAML
# header declares input_file needs no arguments at all.
input_file <- resolve_input_path()

## End(Not run)
```

`run_by_group` *Apply a function to each group in a job manifest or named list*

Description

`run_by_group()` applies a function to each subset of a dataset and collects the results. Subsets can be supplied in two ways: as files listed in a job manifest produced by `write_by_group()`, or as a named list of data frames already in memory. When the function returns tabular output (a data frame or tibble), the results are automatically unnested into a flat tibble with a group-id column. When the function returns non-tabular output (a model, a plot, a file path), the results are returned as a nested tibble with a group-id column and a `results` list-column.

Usage

```
run_by_group(
  manifest = NULL,
  .f,
  ...,
  groups = NULL,
  .id = "group_id",
  .read_fn = read_clean_csv,
  workers = 1L,
  seed = NULL,
  verbose = FALSE
)
```

Arguments

`manifest`	A character string, data frame, or `NULL`. If a string, the path to a job manifest (`manifest.csv`) produced by `write_by_group(manifest = TRUE)`; a relative path is read from the project root (see the Paths section). Must contain a `group_value` and a `file_path` column, and a relative `file_path` is read from the project root too. If a data frame, used directly. If `groups` is supplied, `manifest` is ignored with a warning and may be omitted entirely.

.f	A function to apply to each subset. Must accept a data frame as its first argument. Additional arguments can be passed via ....
...	Additional arguments passed to .f on every call. Forwarded unevaluated when running sequentially, so .f receives them exactly as it would from a direct call and tidy evaluation works. Materialized when workers > 1, which parallel execution requires. See the section on passing arguments below.
groups	A named list of data frames, or NULL (the default). When supplied, manifest is ignored and .f is applied directly to each list element. All elements must be data frames with identical column names and column types – consistent with subsets produced by write_by_group(). If the list is unnamed, groups are assigned fallback names group_1, group_2, etc. with a warning.
.id	A character string. Name of the column that identifies each group in the output. Defaults to "group_id".
.read_fn	A function used to read each subset file when manifest is used. Defaults to read_clean_csv(). Ignored when groups is supplied.
workers	A positive integer, or NULL. Number of parallel R sessions to use. When 1L (the default) or NULL, subsets are processed sequentially with purrr::map(); NULL is accepted as a way of saying "do not parallelize" and is treated as 1L. When greater than 1, subsets are processed in parallel with furrr::future_map(). Requires the furrr and future packages. The maximum allowed value is max(1L, parallelly::availableCores() - 1L) to reserve one core for the main R session. A good starting value is the number of groups or that core ceiling, whichever is smaller.
seed	An integer or NULL. Random seed for reproducible parallel execution. Only relevant when workers > 1 and .f involves randomness (e.g. simulations, bootstrapping). When NULL (the default), no seed management is applied. Ignored when workers = 1L.
verbose	Logical. If TRUE, prints a progress message before processing each group. When workers > 1, per-group progress is replaced by a single summary message showing the worker count. Defaults to FALSE.

Value

A tibble. If .f returns tabular output, the tibble is flat with a .id column prepended. If .f returns non-tabular output, the tibble has two columns: .id and results (a list-column).

The split-apply pattern

run_by_group() is the apply half of the split-apply workflow in toolero. The split half is write_by_group(), which partitions a data frame by a grouping column and writes one file per group along with a job manifest.

```
# Split to disk
write_by_group(penguins, group_col = "species",
               output_dir = "data/jobs", manifest = TRUE)

# Apply from disk via the job manifest
```

```
results <- run_by_group(
  manifest = "data/jobs/manifest.csv",
  .f       = my_analysis
)

# Apply from memory via named list
subsets <- penguins |>
  dplyr::group_split(species) |>
  setNames(c("Adelie", "Chinstrap", "Gentoo"))

results <- run_by_group(
  groups = subsets,
  .f     = my_analysis
)
```

The split is done once. The apply step can be run many times as you iterate on the analysis function.

What .f receives and returns

`.f` receives a single data frame as its first argument. It can return anything, but the return type must be consistent across all groups. Consistency is evaluated by bucket: either all groups return a data frame (tabular) or none do (non-tabular). Mixed returns cause an error identifying which groups returned unexpected types.

Common return types and their output shape:

- A one-row tibble of summary statistics – unnested into a flat table
- A multi-row tibble (e.g. model coefficients) – unnested with the group ID repeated per row
- A model object – returned as a list-column
- A ggplot object – returned as a list-column
- A file path – returned as a list-column

Passing arguments through to `.f`

Most arguments need nothing special. A number, a string, a logical, a file path, a function: these are ordinary values, and they reach `.f` in both modes exactly as you would expect. `run_by_group()` imposes no requirement on how `.f` is written.

```
scale_group <- function(data, multiplier) {
  dplyr::summarise(data, total = sum(x) * multiplier)
}

run_by_group(groups = subsets, .f = scale_group, multiplier = 10)
```

One case is different, and only one: an argument that is a bare *column name*. A symbol like `flipper_length_mm` has no meaning until it meets the data, so a function that accepts one has to capture it rather than evaluate it, which is what `{{ }}` does. That is a property of how `.f` is written rather than anything `run_by_group()` asks for – calling such a function directly has the same requirement.

```
plot_group <- function(data, x, y) {
  ggplot2::ggplot(data, ggplot2::aes(x = {{ x }}, y = {{ y }})) +
    ggplot2::geom_point()
}

run_by_group(
  groups = subsets,
  .f     = plot_group,
  x      = flipper_length_mm,
  y      = body_mass_g
)
```

Run sequentially, which is the default, `...` is forwarded *unevaluated*, so `flipper_length_mm` stays a symbol until `{{ }}` captures it inside `plot_group()` and evaluates it against the group's data. Versions before 0.5.0 evaluated `...` in `run_by_group()`'s own frame, where that symbol means nothing, so the call above failed with `object 'flipper_length_mm' not found` before `plot_group()` was reached.

If you would rather not write `.f` that way, pass the column name as a string and index with it. A string is an ordinary value, so it needs no tidy evaluation and works in both modes:

```
plot_group <- function(data, x, y) {
  ggplot2::ggplot(data, ggplot2::aes(x = .data[[x]], y = .data[[y]])) +
    ggplot2::geom_point()
}

run_by_group(
  groups = subsets,
  .f     = plot_group,
  x      = "flipper_length_mm",
  y      = "body_mass_g"
)
```

Bare column names do not survive `workers > 1`**.** Parallel execution sends the work to separate R sessions, which means every argument has to be materialized and serialized first. An argument whose value exists only inside the data mask `.f` builds has nothing to serialize, so it cannot make the trip. `run_by_group()` reports this directly rather than letting it surface from inside `future`'s globals inspection. Ordinary values, strings included, are unaffected.

The portable form for a bare column name moves it inside `.f`, which works in both modes:

```
run_by_group(
  groups  = subsets,
  .f      = \(d) plot_group(d, x = flipper_length_mm, y = body_mass_g),
  workers = 4
)
```

Two smaller consequences of forwarding rather than forcing, both matching what a direct call to `.f` does, and both sequential-only: an argument with a side effect is evaluated at most once for the whole call rather than once per group, and an argument `.f` never touches is never evaluated at all.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

Examples

```
sample_path <- system.file("templates", "sample.csv", package = "toolero")
penguins <- read_clean_csv(sample_path)

# Split the data to disk
tmp <- tempdir()
write_by_group(penguins, group_col = "species",
               output_dir = tmp, manifest = TRUE)

# Define an analysis function
```

```
summarise_species <- function(data) {
  dplyr::summarise(data,
    n            = dplyr::n(),
    mean_mass    = mean(body_mass_g, na.rm = TRUE),
    mean_flipper = mean(flipper_length_mm, na.rm = TRUE)
  )
}

# Apply via the job manifest -- returns a flat tibble
results <- run_by_group(
  manifest = file.path(tmp, "manifest.csv"),
  .f       = summarise_species
)

# Apply via named list in memory
subsets <- penguins |>
  dplyr::group_split(species) |>
  setNames(c("Adelie", "Chinstrap", "Gentoo"))

results <- run_by_group(
  groups = subsets,
  .f     = summarise_species
)

# Apply a function that returns a model -- returns a nested tibble
fit_model <- function(data) {
  lm(body_mass_g ~ flipper_length_mm, data = data)
}

models <- run_by_group(
  manifest = file.path(tmp, "manifest.csv"),
  .f       = fit_model
)

# Parallel execution using available cores
workers <- max(1L, parallelly::availableCores() - 1L)

results <- run_by_group(
  manifest = file.path(tmp, "manifest.csv"),
  .f       = summarise_species,
  workers  = workers
)

# Reproducible parallel execution with a fixed seed
random_summary <- function(data) {
  tibble::tibble(val = sample(seq_len(nrow(data)), 1))
}

results <- run_by_group(
  manifest = file.path(tmp, "manifest.csv"),
  .f       = random_summary,
  workers  = workers,
  seed     = 1234
```

```
)
```

`save_output` *Save an object and record it in the project accumulator*

Description

`save_output()` writes `object` to `file_path` via `.f(object, file_path, ...)`, then – when `manifest = TRUE` (the default) – appends a row to the project-level accumulator at `output_dir/accumulator.csv` recording what was saved, how, and whether the write succeeded. The accumulator is the working file later consumed by `generate_manifest()`, which deduplicates it and reshapes it into the output record, `project-manifest.json`.

Usage

```
save_output(
  object,
  file_path,
  .f,
  ...,
  manifest = TRUE,
  note = NULL,
  output_dir = NULL,
  config = NULL
)
```

Arguments

`object`	The object to save.
`file_path`	Character. A single destination path for `object`. A relative path is written under the project root (see the Paths section), so `"output/fit.rds"` lands in the project's own `output/` folder wherever the code runs. Its parent directory is created if it does not already exist, and the creation is reported. It is recorded in the accumulator relative to the project root (`"output/fit.rds"`) when it lies inside the project, and as the absolute path otherwise.
`.f`	A function used to perform the save, called as `.f(object, file_path, ...)`. Supply the function itself (for example `saveRDS`, `ggplot2::ggsave`), not a call and not a string. Avoid reassignment indirection (`my_fn <- ggsave; .f = my_fn`) – the accumulator records the name exactly as written at the call site, so this records `"my_fn"` rather than `"ggsave"`. Anonymous functions are recorded as `"anonymous function: ..."` with the body collapsed to a single truncated line.
`...`	Additional arguments passed to `.f`.
`manifest`	Logical. When `TRUE` (default), append a row to the accumulator. When `FALSE`, `object` is still saved via `.f`, but nothing is recorded.
`note`	Character or `NULL`. An optional free-text note recorded alongside this row.

`output_dir` Character or `NULL`. Directory containing (or to contain) the accumulator. An explicit relative value is read from the project root (see the Paths section); an absolute one is used as given. If `NULL` (the default), resolved from `config`'s `output_dir` convention when `config` is supplied, or `"output"` otherwise, and in either case taken from the project root: relative to the folder holding `config`, or to the project root.

`config` Character or `NULL`. Path to a project configuration file (typically a project's own `_toolero.yml`, as written by `init_project()`). Only consulted when `output_dir` is not supplied; an explicit `output_dir` always wins. Defaults to `NULL`, which leaves pre-0.5.1 behavior unchanged.

Details

The call to `.f` is wrapped in a narrowly-scoped `tryCatch()` – only the `.f(object, file_path, ...)` call itself, not the rest of `save_output()`'s body. On failure, a row is still appended recording `status = "failure"` and the caught message, after which the original condition is rethrown unmodified. Its class, message, and call are preserved as caught, so downstream handlers behave as though `.f()` had been called directly. This is the one place in the package where the `cli` convention is deliberately not followed: `cli::cli_abort()` would construct a new condition and discard the original class.

`r_class` records `class(object)` as a single pipe-separated field, captured before the write, since class cannot be reliably recovered from the file afterward. Timestamps are recorded in UTC with millisecond precision so that they sort lexicographically – `generate_manifest()` relies on this when keeping the latest row per `file_path`.

Value

`object`, invisibly. Called for its side effects.

Project conventions

The accumulator for a document under `reports/` lands in the project's own `output/` rather than in `reports/output/`, because paths start at the project root. `config` is opt-in: without it, `output_dir` defaults to `output/` under the project root. When `config` is supplied but cannot be read, this aborts with the same message `init_project()` gives for a bad `config`, rather than silently falling back to `"output"`.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.

2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`generate_manifest()`

Examples

```
output_dir <- withr::local_tempdir()

save_output(
  object = mtcars,
  file_path = fs::path(output_dir, "mtcars.rds"),
  .f = saveRDS,
  note = "Unmodified example data.",
  output_dir = output_dir
)
```

write_by_group *Split a data frame by one or more grouping columns and write each group to a CSV file*

Description

Splits a data frame by one or more grouping columns and writes each group to a separate CSV file. Optionally writes a job manifest listing the output files, their group values, and row counts.

Usage

```
write_by_group(
  data,
  group_col,
  output_dir = NULL,
  manifest = FALSE,
  drop_na = TRUE,
  prefix = NULL,
  config = NULL
)
```

Arguments

data	A data frame or tibble to split and save.
group_col	A character vector. The name(s) of the column(s) to group by. When more than one column is supplied, groups are formed from the combinations of values actually present in the data (not the full cross-product of possible values).
output_dir	A string or NULL. Path to the directory where output files will be written. A relative path is written under the project root (see the Paths section). Created if it does not exist. If NULL, resolved from config's split_dir convention when config is supplied; otherwise the user must supply a path explicitly.
manifest	A logical. Whether to write a manifest.csv file to output_dir listing the output files, group values, and row counts. Defaults to FALSE.
drop_na	A logical. If TRUE (default), rows with a missing value in any grouping column are dropped before splitting, and a message reports how many rows were dropped and from which column(s). If FALSE, missing values are treated as their own group instead of being dropped.
prefix	A string or NULL. An optional namespace prepended to every output filename, sanitized the same way group values are and joined with a single -. prefix = "data" grouping on one column turns a.csv into data-a.csv; grouping on two turns a--female.csv into data-a--female.csv. Defaults to NULL, which leaves filenames unchanged.
config	A string or NULL. Path to a project configuration file (typically a project's own _toolero.yml, as written by init_project()). When supplied, and output_dir is not, output_dir defaults to the config's split_dir convention. An explicit

output_dir always wins over config; config only fills in what you didn't supply. Defaults to NULL, which leaves today's behavior unchanged: output_dir must be supplied directly, config or no config. Placed last in the signature, along with prefix, so that adding either does not shift any existing positional argument.

Details

Output filenames are derived from the group values of group_col. Each value is sanitized independently: converted to lowercase, runs of non-alphanumeric characters replaced with a single -, and leading/trailing dashes stripped. When group_col has more than one element, the sanitized values are joined with -- in the order supplied, so group_col = c("species", "sex") on an Adelie male produces adelie--male.csv. A prefix, if supplied, is sanitized the same way and joined to the front with a single -.

Value

Invisibly returns output_dir.

Why the separators differ

Because a run of non-alphanumeric characters collapses to exactly one dash, a sanitized value can contain a single - but never two in a row. That is what makes -- safe between columns: it can only ever appear where this function put it.

The alternative would lose data rather than merely look untidy. Joined with a single dash, the groups ("a-b", "c") and ("a", "b-c") both produce the key a-b-c, and since the split is performed on that key the two groups would be merged into one file and reported as one manifest row. Joined with -- they are a-b--c and a--b-c, and stay distinct.

prefix is joined with a single - instead, because it is constant across every file in a call and so cannot create a collision: prepending the same string to two keys leaves them exactly as distinct as they were. It is a namespace for the whole split rather than another field of the group, and reads better as one.

The job manifest

If manifest = TRUE, a manifest.csv is written to output_dir. This is the *job manifest*: a list of inputs to a computation that has not happened yet, and the file consumed by run_by_group() and by submitr::htc_gen_submit() in multiple-job mode. It is a different document from the *output record* that generate_manifest() writes, which records outputs from a computation that already has.

The schema is one column per grouping variable, holding the raw unsanitized value, followed by group_value, n_rows, and file_path. file_path is relative to the project root when the file is inside the project ("data/jobs/adelie.csv"), and the absolute path otherwise, so the manifest means the same thing wherever it is read from. Grouping on one column therefore produces a job manifest whose first column repeats group_value exactly. That redundancy is deliberate: one schema with a varying column count is easier to read, validate and rely on than two schemas selected by how many columns you happened to group on.

group_value is a human-readable composite of the raw values joined by " | ", so "Adelie | male" for two columns and simply "Adelie" for one.

Row order

Groups are written, and job manifest rows recorded, in order of first appearance in `data`. This matters downstream: `submitr` writes its `subdatasets.csv` in job manifest order, HTCondor assigns `ProcId` in that order, and log filenames are reconstructed from position, so job manifest row order is the mapping from a job number back to a group.

Missing values

With `drop_na = TRUE` (the default), rows with a missing value in any grouping column are removed before splitting and a message reports how many.

With `drop_na = FALSE`, missing values are coerced to the string `"NA"` so that they form their own group rather than being dropped silently by `split()`. A column that also contains a literal `"NA"` value – North America, Not Applicable, a country code – would then have two semantically different groups collapse into one file. Rather than merge them, `write_by_group()` aborts and names the column.

Note: `output_dir` has no default value of its own. Supply it directly, or supply `config` and let it resolve from the project's `split_dir` convention. Use `tempdir()` for temporary output during testing or exploration.

Project conventions

`config` is entirely opt-in. A project never scaffolded by `init_project()` behaves exactly as before: pass `output_dir` and nothing about `config` changes. When `config` is supplied but the file cannot be read, this aborts with the same message `init_project()` gives for a bad `config`, rather than silently falling back to a built-in default – a config you asked for and didn't get should never look identical to not asking.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

See Also

`run_by_group()`, the apply half of this pair.

Examples

```
# Split a small data frame by group and write to a temp directory
data <- data.frame(
  species = c("Adelie", "Adelie", "Gentoo"),
  mass    = c(3750, 3800, 5000)
)
write_by_group(data, group_col = "species", output_dir = tempdir())

# Same but also write a job manifest
write_by_group(data, group_col = "species",
               output_dir = tempdir(), manifest = TRUE)

# Namespace the filenames -- adelie.csv becomes penguins-adelie.csv
write_by_group(data, group_col = "species", prefix = "penguins",
               output_dir = tempdir())

# Group by more than one column
data2 <- data.frame(
  species = c("Adelie", "Adelie", "Gentoo"),
  sex     = c("male", "female", "male"),
  mass    = c(3750, 3550, 5000)
)
write_by_group(data2, group_col = c("species", "sex"),
               output_dir = tempdir(), manifest = TRUE)

# Let a project's own _toolero.yml supply output_dir via split_dir.
# generate_project_config() writes the default conventions, including
# split_dir = "data/jobs" -- a relative path meant to be resolved against
```

```
# a project's own root, so this example runs from config_dir via
# withr::with_dir() rather than writing into the working directory.
config_dir <- tempfile()
dir.create(config_dir)
generate_project_config("_toolero.yml", path = config_dir)
withr::with_dir(config_dir, {
  write_by_group(
    data,
    group_col = "species",
    config    = file.path(config_dir, "_toolero.yml")
  )
})
```

`write_clean_csv` *Write a cleaned data frame to a CSV file*

Description

`write_clean_csv()` writes a data frame to a CSV file using `readr::write_csv()` and emits a cli confirmation message reporting the number of rows and columns written. It is the natural counterpart to `read_clean_csv()`, reinforcing the convention that `data-raw/` holds original inputs and `data/` holds cleaned, analysis-ready outputs.

Usage

```
write_clean_csv(data, path, overwrite = FALSE, ...)
```

Arguments

`data`	A data frame or tibble to write.
`path`	A character string with the path to the output CSV file. A relative path is written under the project root (see the Paths section), and its folder is created if it does not exist.
`overwrite`	Logical. If `FALSE` (the default), errors if the file already exists. Set to `TRUE` to overwrite an existing file.
`...`	Additional arguments passed to `readr::write_csv()`, such as `append`, `col_names`, or `quote`.

Details

If column names are not already clean, `write_clean_csv()` applies `janitor::clean_names()` before writing and emits a warning listing the affected columns.

Value

Invisibly returns `path`, resolved from the project root when it was relative.

Paths

A relative path is read from the project root, not from the working directory. So `save_output(fit, "output/fit.rds", saveRDS)` writes into the project's own `output/` folder whether you call it from the project root, from a document under `reports/`, or from a job on an HTCondor execute node. You never need to wrap a path in `here::here()`. Doing so still works, because `here::here()` returns an absolute path and absolute paths are used as given.

The project root is found afresh on every call, in this order:

1. The job's scratch directory, when `_CONDOR_SCRATCH_DIR` names one. HTCondor sets that variable itself, inside every job.
2. The nearest folder above the working directory that carries a project marker: the `.here` file `init_project()` writes, an `.Rproj` file, a package `DESCRIPTION`, or a version control folder such as `.git`.
3. The working directory, when neither of the above applies.

Absolute paths (including those starting with `~`, or a drive letter on Windows) and URLs such as `https://` or `s3://` are used as given. `..` and `./` are not interpreted, so `"../data/x.csv"` means the folder above the root; write an absolute path to leave the project. A bare `"."` means the root itself.

Functions that read are forgiving about older code. A relative path that does not exist at the project root but does exist relative to the working directory is read from there, and a message says so. Functions that write never guess: a relative path is always written under the root.

A few arguments are deliberately left alone:

- `init_project()` reads `path`, `config` and `use_rprofile` from the working directory, because the project does not exist yet and so has no root.
- Under `Rscript`, `resolve_input_path()` reads the command line argument as given. Run `Rscript` from the project root or pass an absolute path.
- Arguments passed on through `...` to another function reach it untouched.
- Where a default is "beside the input" (`qmd_to_r()` and `generate_kb_xml()`), the output follows wherever the input was read from.

Two limits are worth knowing. A project nested inside another project is not supported: the nearest marker wins, which can surprise you when you work from the outer one. And on an execute node `~` and `$HOME` cannot be trusted (`$HOME` can be the submit node's path), so prefer relative paths in job scripts.

Examples

```
sample_path <- system.file("templates", "sample.csv", package = "toolero")
data <- read_clean_csv(sample_path)

# Write to a temp file
out <- tempfile(fileext = ".csv")
write_clean_csv(data, out)

# Overwrite an existing file
write_clean_csv(data, out, overwrite = TRUE)
```

```
# Dirty names are cleaned automatically with a warning
dirty <- data.frame("First Name" = "Jane", "Last Name" = "Doe",
                    check.names = FALSE)
write_clean_csv(dirty, tempfile(fileext = ".csv"))
```

Index

arborize, 2

check_project, 6
check_project(), *31*, *33, 34*, *37*
create_qmd, 9
create_qmd(), *7*, *28*, *30*

detect_execution_context, 15
detect_execution_context(), *26*, *33*, *46, 47*

fs::file_exists(), *45*

generate_citation, 16
generate_citation(), *19, 20*, *30*
generate_data_doc, 18
generate_data_doc(), *24*
generate_kb_xml, 20
generate_kb_xml(), *5*, *8*, *14*, *17*, *19*, *22*, *24*, *27*, *29*, *31*, *37*, *39*, *41*, *44*, *47*, *52*, *56*, *60*, *62*
generate_license, 22
generate_license(), *20*
generate_manifest, 24
generate_manifest(), *42–44*, *54–56*, *58*
generate_profile, 28
generate_profile(), *10*, *16*, *18*
generate_project_config, 30
generate_project_config(), *6*, *8*, *28*, *33*, *37*

init_project, 32
init_project(), *4*, *6–8*, *11*, *13, 14*, *17*, *19–27*, *29–31*, *36–39*, *41*, *44*, *47*, *52*, *55–57*, *59, 60*, *62*

qmd_to_r, 38
qmd_to_r(), *5*, *8*, *14*, *17*, *19*, *22*, *24*, *27*, *29*, *31*, *37*, *39*, *41*, *44*, *47*, *52*, *56*, *60*, *62*
quarto::quarto_render(), *15*

read_clean_csv, 40
read_clean_csv(), *49*
read_output_records, 42
read_output_records(), *26, 27*
renv::init(), *35*
renv::scaffold(), *7*, *33*, *35*
resolve_input_path, 45
resolve_input_path(), *5*, *8*, *13–15*, *17*, *19*, *22*, *24*, *27*, *29*, *31*, *37*, *39*, *41*, *44*, *47*, *52*, *56*, *60*, *62*
run_by_group, 48
run_by_group(), *58*, *60*

save_output, 54
save_output(), *24–27*, *43, 44*

usethis::create_project(), *33, 34*
usethis::local_project(), *34*

write_by_group, 57
write_by_group(), *26*, *48, 49*
write_clean_csv, 61