Building tidy R packages for the OMOP common data model

Edward Burn, Martí Català

2025-12-01

Table of contents

Preface

This is an **opinionated** book on how we develop packages working in the Tidy R OMOP CDM space.

Our book, although covers some parts of basic R package development assumes the user knows the very basis. If you are new to R package development we would recommend you the following book: R Packages; Learn how to create a package, the fundamental unit of shareable, reusable, and reproducible R code.

The book is divided in four parts:

- **(I)** Getting started: covers how to start with an empty package, and helps you think about which is the scope of your package or guide you on how to contribute to existing packages.
- **(II)** Principles: this is the most important part of the book as it covers which packages to use, the different naming conventions, classes to use, how to validate the inputs and more.
- **(III)** Documentation: covers how to create the documentation of the packages, such as populate readme, documenting functions, documenting code, vignettes and more.
- **(IV)** Testing: covers how to write tests for the packages, what to test and how to do it efficiently.
- **(V)** GitHub: covers how to set up automatic GitHub testing and documentation using GitHub actions.

- **(VI)** Maintaining a package: once a package is ready for a first release you will have to maintain it, this packages covers how to submit to cran for first time and maintain the package for the future.

Part I.

Getting started

In this first part it is covered how to ...

1. Scope

1.1. Is this package needed?

Before deciding to develop a package you must ask yourself many questions starting with: *Is this package needed?* A package must cover need and be refined.

2. Ecosystem

2.1. How the package fits in the ecosystem

3. Contributing to a package

3.1. GitHub issues

3.1.1. Opening GitHub issues

The first step to contributing to an existing package is by opening issues on its GitHub repository. These issues could be about bugs you encountered when using the package, requests for additional functionality (that you might even want to add yourself), or clarification questions on package documentation.

3.1.2. What makes a good issue

If reporting a bug, then a reprex makes the maintainers life much, much easier (and in turn increases the likelihood of a quick fix being introduced much higher!). This reprex will allow the developer to quickly reproduce the issue. This is already halfway to solving it, given the old adage > "A problem well stated is half solved."

Often an issue may arise when running code against a database with real patient-level data which can make it challenging to quickly create a reprex (as the patient data for which the bug was seen cannot be shared). In this situation it may be possible to reproduce the problem on an available synthetic dataset like Eunomia, in which case the package developer will be able to reproduce the bug with the synthetic data. However, it may be that you only encounter the issue for some particular set of patients

with specific characteristics which are not seen in the synthetic data. In this case you can, for example, use the omock package to create a set of synthetic patients with these characteristics so that the developer can use this same data to reproduce the problem.

Although of course it would be ideal to include a reprex, this may not always be possible. In such cases please do make sure to provide a bug report, with as much information as possible, as this will still be extremely useful and appreciated.

If opening an issue to ask for clarifications on documentation or with requests for additional functionality, the more precise and specific the issue the better. If you have a long laundry list of feature requests, it would generally be better to open each as separate issues so they can be addressed incrementally.

Note, even if you are not the maintainer of package but want to open a pull request to change documentation or add functionality then try to always open an issue to discuss this first. Even some seemingly trivial changes might be out of scope for the package (or they might have already been added in development branches of the repo). However well-intentioned, "drive-by pull requests" are often challenging for package maintainers to deal with.

3.2. Responding to GitHub issues

On the other side of the equation if you are the maintainer of package, receiving issues is one the main ways you will receive feedback on the package. Get used to starting with somewhat vague issues that will need some back and forth to get to the route of the problem or request for additional functionality.

When a bug is fixed in the package as a rule you should also add a corresponding test to make sure that it will stay fixed into the future. One

benefit of being given a reprex is that not only will it help you to fix the problem but the reprex itself can be used as the test for the package. By receiving and acting on issues raised, which will often be somewhat exotic edge cases or people using the package in some way that you hadn't expected, the package will become hardened over time.

When it comes to requests for adding functionality, or even offers to add it, this is a tricky balancing act. Remember that you will be the one maintaining the package in the long-term. It is often the case that the quick addition of something seemingly useful can make a package more difficult to maintain or extend in the future. Typically it is worth waiting a day or two at least before agreeing to add something to a package so that you don't rush into making an addition or change that you will later come to regret.

Another point to keep in mind when considering issues with requests for functionality is whether to add more general functionality than is being asked for. A well-intentioned user may be asking for something very specific to be added, but often underlying this request will be a broader need for more general functionality. This my tie in to other requests received, or may even need more discussion with users to better understand whether there is a broader need at play. general solutions

Remember that sadly few people be opening issues with praise if they've used your package successfully. The majority of users will use the code you've written without a problem and likely without even realising how much work has gone into it. And the people that do open issues are providing an extemely valuable service that will lead to the software getting better and better over time. So try to keep this in mind when tackling the seemingly never-ending list of issues you'll get if your software is being used.

3.3. Contributing code

Contribute documentation

Contributing documentation is one of the best ways to improve a package. Fixing typos, adding clarifications, and even writing whole vignettes will almost always be highly welcomed by the maintainer and improve the experience of other users. You can find issues related to documentation searching for the label *documentation*. Also you can open an issue if you think that somethings is not documented properly and propose the maintainer to fix it yourself.

You can read more details on how the documentation of the packages is done in the relevant chapters.

Fixing a bug

Contributing code is a more involved task. Before starting, make sure you've at the very least interacted with the maintainer so that they will be in favour of the changes or additions you are planning. You can do that replying to the issue you want to fix. Apart from fixing the bug, make sure to add tests alongside code changes to show that it is truly fixed. When fixing a bug existing tests should typically be unchanged (unless there is an issue with the test itself). If you are having to change existing tests to fix a bug this is normally a sign that there might be some more profound issue and will likely need back and forth between you and the maintainer for this to be addressed.

Adding new functionality

If adding functionality this again should be accompanied with tests to show that it adds the desired behaviour. Moreover, updates to package docu-

mentation, including changes or additions to vignettes might be needed. Expect back and forth with the package maintainer, as they might well have feedback on the implementation you propose.

3.4. Opening pull requests

Changes to packages should come via pull requests. Create a branch or a fork of the code, make your change, and open your pull request. If opening a pull request, more than anything try to create a pull request that will be easy for the package maintainer to review and merge. A pull request will be better for everyone involved when it addresses just one specific issue and affects less than 250 lines of code. One quick way to create complexity for the maintainer is to open a pull request that addresses multiple issues at the same time. Although this could be more efficient for the person opening the pull request, it often slows down the review by increasing the complexity of the review for the maintainer.

Before starting to contribute any code, first make sure the package tests are all passing. To do so after cloning run the following code:

```
devtools::check()
```

The output should be:

```
0 errors v | 0 warnings v | 0 notes v
```

If not raise an issue before going any further (although please first make sure you have all the packages from imports and suggests installed).

Now you are ready to do your code contribution. Add the relevant code and when you are happy with the changes that you have made, please follow these steps to open a pull request:

3.4.1. Initial checks

Run the below to update and check package documentation:

```
devtools::document()
devtools::check_man()
```

Test that the examples work:

```
devtools::run_examples()
```

Test that the readme and vignettes work and update the results if the output of any function has changed:

```
devtools::build_readme()
devtools::build_vignettes()
```

3.4.2. Run tests

As you then contribute code, make sure that all the current tests and any you add continue to pass. All package tests can be run together with:

```
devtools::test()
```

Code to add new functionality should be accompanied by tests. Code coverage can be checked using:

```
# note, you may first have to detach the package
# detach("package:IncidencePrevalence", unload=TRUE)
devtools::test_coverage()
```

3.4.3. Adhere to code style

Please adhere to the code style when adding any new code. Do not though restyle any code unrelated to your pull request as this will make code review more difficult.

```
lintr::lint_package(linters = lintr::linters_with_defaults(
  lintr::object_name_linter(styles = "camelCase")
))
```

3.4.4. Run check() before opening a pull request

Before opening any pull request please make sure to run:

```
devtools::check()
```

Please make sure that the output is the expected:

```
0 errors v | 0 warnings v | 0 notes v
```

Any error or warning will make your pull request actions fail. Although notes can pass the github action tests we encourage to fix all the notes as this is a requirement to keep the package on cran.

3.4.5. Opening the pull request

Once you've made sure that checks are passing and you are happy with your code additions you can open the pull request. When opening the pull request you must take into account the following:

- Write a meaningful title, titles of pull requests are later used to document the changes done. Make sure that the title of the pull request describes the issue that is fixed, therefore the documentation of the changes will be easier.
- Link the issue that your pull request is closing (we strongly encourage to fix issues one by one and do not include multiple issues in the same pull request). You can link a pull request and an issue using any of the gitub closing words or you can do it manually in the *development* section (bottom of the right sidebar).
- Describe any potential issue you want to remark. Help the maintainer review your pull request indicating if there is anything you are not sure about or that you want some feedback on. Don't hesitate to ask any question if needed.

3.5. Reviewing pull requests

Reviewing pull requests is a very important step that the maintainers have to do. When reviewing a pull request you have to find a good equilibrium between being kind (we want to encourage people to contribute to our packages) and being strict (you -as a maintainer- will be the ultimate responsible for that code; so review it carefully and only accept if you are happy about it and you feel comfortable maintaining that code).

4. Set up your environment

To build an R package you will need R installed and an IDE: preferably R Studio or Positron.

You will need to install devtools and usethis as they are key to follow the different steps to get started.

```
# use base R
install.packages(c("usethis", "devtools"))

# or pak (recommended)
library(pak)
pkg_install(c("usethis", "devtools"))
```

To work with the Tidy R OMOP ecosystem, you will also need to install the omopgenerics package. Alongside with other packages that may be useful:

- PatientProfiles for data manipulation.
- CodelistDiagnostics to query the vocabularies.
- visOmopResults for data visualisation.

You can see the list of Tidy R packages in our website.

4.1. Create an empty package

To create an empty package you can do it with *usethis*:

```
library(usethis)

create_package(path = "path/to/folder")
```

> **Package name**
>
> The package will be named by the name of the folder.

4.2. Add a license

Adding licenses is very important so you are protected and users know how they can use your product. In general we release using the Apache 2.0 license.

You can add a license easily with usethis:

```
use_apl2_license()
```

This will add the `LICENSE.md` file and populate the DESCRIPTION file with the relevant license field.

> **License agreements**
>
> License agreements supported by `usethis` are:
>
> - More permissive:
>   - MIT: simple and permissive.

  - Apache 2.0: MIT + provides patent protection.

- Copyleft:
  - GPL v2: requires sharing of improvements.
  - GPL v3: requires sharing of improvements.
  - AGPL v3: requires sharing of improvements.
  - LGPL v2.1: requires sharing of improvements.
  - LGPL v3: requires sharing of improvements.

- Creative commons licenses appropriate for data packages:
  - CC0: dedicated to public domain.
  - CC-BY: Free to share and adapt, must give appropriate credit.

```
use_mit_license(copyright_holder = NULL)

use_gpl_license(version = 3, include_future = TRUE)

use_agpl_license(version = 3, include_future = TRUE)

use_lgpl_license(version = 3, include_future = TRUE)

use_apache_license(version = 2, include_future = TRUE)

use_cc0_license()

use_ccby_license()

use_proprietary_license(copyright_holder = "...")
```

4.3. Create a first mock function

The `R` folder contains the relevant files for your package

4.4. Check the package

4.5. Create a simple test

We will see

After adding the test we should check the test works properly. You can run the tests of the active file using:

```
test_active_file()
```

Or you can run all the tests of the package:

```
test()
```

4.6. Create `README.Rmd`

4.7. Create website

4.8. Set up GitHub

GitHub actions are very useful to ensure the continuous integration workflow. The fifth part of the book is dedicated to GitHub and GitHub Actions at that stage you will learn how to use github for:

- Automatic checking your package
- Automatic deployment of a documentation website for your package
- Automatic assessment of test coverage
- Automatic testing across multiple sources

4.9. Package template

Although the first time it is quite nice to do all the steps one by one there is a template with all those steps done, you can create a new package repo using this link.

> **Template link**
>
> The GitHub template link can be found: https://github.com/oxford-pharmacoepi/EmptyPackageTemplate.

Part II.

Principles

5. Core dependencies

The *Tidy R OMOP CDM* packages rely on dplyr and the tidyverse packages to manipulate the data from the `cdm` object. The `cdm` object is an object that contains all the tables available and is central for the data manipulation. This `cdm` object is defined in omopgenerics.

5.1. omopgenerics

5.1.1. Classes

5.1.2. Methods

5.1.3. Input validation functions

5.1.4. Manipulate a `cdm_reference` object

5.1.5. Manipulate a `summarised_result` object

5.2. Other useful packages

The *omopverse* has some core packages that will be useful to simplify the scope of your package:

- PatientProfiles: this package is very useful for data manipulation, you can find extract basic demographic information, intersections of cohorts and summarise data in a standard format. Take a look at the list of functions in PatientProfiles to prevent you from duplicating code.
- visOmopResults: this package is very useful to create data visualisations (tables and plots) from standardised data. There are many different supported types of tables (`r visOmopResults::tableType()`) and with little customisation you can get quite pretty tables. For plots the support is a bit more limited but it creates some simple ggplot2 visualisations with very little effort. Take a look at the list of functions in visOmopResults to prevent you from duplicating code.
- CodelistGenerator: this package can be useful to query the vocabularies. Take a look at the list of functions in CodelistGenerator to prevent you from duplicating code.
- omock: this package can be very useful for testing. With very few lines of code you can create a mock CDM object with your desired specifications, specially useful can be the function `mockCdmFromTables()` that will create a viable CDM object from the supplied information. Take a look at the list of functions in omock to prevent you from duplicating code. This package is quite useful, but usually listed in Suggests as you will only need this package for testing purposes.

6. Function interfaces

`generate...CohortSet`, `...Cohort`, `add...`, `summarise...`, `plot...`, `table...`,

7. Conventions

Our code adheres to some style conventions as decribed in this chapter.

7.1. Casing

In general we use **camelCase** naming for arguments and functions.

7.2. Argument naming

8. Error messages and input validation

Giving users good and informative error messages is key for a good user experience. To do so it is important that we perform an input validation at the beginning of each function. On the other hand we do not want to spend more time checking the input than executing the function, so don't overdo it. In general we would check that any input function has the desired type and length and that the desired evaluation looks feasible with the current input parameters.

8.1. omopgenerics validatin functions

Some arguments that are consistent across different functions and packages have their own *validate* functions in the **omopgenerics** package:

- `validatevalidateAgeGroupArgument()` is used to validate the `ageGroup` argument. The output `ageGroup` will always be formatted as a named list (name of the age group) and each age group will be defined by named intervals.
- `validateCdmArgument()` is used to validate the `cdm` argument. By default only the class is validated as this can take time, but specific checks can be triggered if needed.
- `validateCohortArgument()` is used to validate `cohort` argument, very used in many packages, this validates that the input is a properly formatted cohort. Extra checks can be triggered if needed.

- `validateCohortIdArgument()` is used to validate `cohortId` argument
- `validateConceptSetArgument()`
- `validateNameArgument()`
- `validateNameStyle()`
- `validateResultArgument()`
- `validateStrataArgument()`
- `validateWindowArgument()`

It is important that we assign the output to the variable as the object might change during the validation process to ensure different allowed inputs but as the output of the validation process will always be the same this simplifys the code as you do not have to think about the different allowed inputs.

the `ageGroup` argument can be a good example of this behavior:

```
library(omopgenerics, warn.conflicts = FALSE)
ageGroup <- validateAgeGroupArgument(ageGroup = list(c(0, 1), c(10, 20)))
ageGroup
```

```
$age_group
$age_group$`0 to 1`
[1] 0 1

$age_group$`10 to 20`
[1] 10 20
```

```
ageGroup <- validateAgeGroupArgument(ageGroup = list(
  my_column = list("young" = c(0, 19), 20, c(21, Inf)),
  list(c(0, 9), c(10, 19), c(20, 29), c(30, Inf))
))
ageGroup
```

```
$my_column
$my_column$young
[1]  0 19

$my_column$`20 to 20`
[1] 20 20

$my_column$`21 or above`
[1]  21 Inf


$age_group_2
$age_group_2$`0 to 9`
[1] 0 9

$age_group_2$`10 to 19`
[1] 10 19

$age_group_2$`20 to 29`
[1] 20 29

$age_group_2$`30 or above`
[1]  30 Inf
```

As you can see the output is always a named list that contains named intervals the function itself will also throw explanatory errors if they are not properly formatted:

```r
validateAgeGroupArgument(
  ageGroup = list(age_group1 = list(c(0, 19), c(20, Inf)), age_group2 = list(c(0, In
  multipleAgeGroup = FALSE
)
```

```
Error:
```

```
! Multiple age group are not allowed
```

```
validateAgeGroupArgument(
  ageGroup = list(age_group1 = list(c(-5, 19), c(20, Inf)))
)
```

```
Error in `purrr::map()`:
i In index: 1.
i With name: age_group1.
Caused by error:
! Elements of `ageGroup` argument must be greater or equal to "0".
```

```
validateAgeGroupArgument(
  ageGroup = NULL, null = FALSE
)
```

```
Error:
! `ageGroup` argument can not be NULL.
```

8.2. omopgenerics assert functions

The omopgenerics package contains some functions for simple validation steps this can be useful helpers to validate an input with a single line of code, they also contain arguments to check if they have

8.3. Examples

Let's say we have a function with four arguments (`cohort`, `cohortId`, `window` and `overlap`), we could easily validate the input arguments of the function with 4 lines of code:

```
myFunction <- function(cohort, cohortId = NULL, window = c(0, Inf), overlap = FALSE)
  # input check
  cohort <- omopgenerics::validateCohortArgument(cohort = cohort)
  cohortId <- omopgenerics::validateCohortIdArgument(cohortId = {{cohortId}}, cohort
  window <- omopgenerics::validateWindowArgument(window = window)
  omopgenerics::assertLogical(overlap, length = 1)

  # code ...

}
```

Note the {{ symbols are needed to be able to use tidyselect verbs such as `starts_with()` or `contains()`.

A second example that needs some custom extra code can be:

```
myFunction <- function(cdm, conceptSet, days = 180L, startDate = NULL, overlap = TRU
  # input check
  cdm <- omopgenerics::validateCdmArgument(cdm = cdm)
  conceptSet <- omopgenerics::validateCdmArgument(conceptSet = conceptSet)
  omopgenerics::assertNumeric(days, integerish = TRUE, min = 0, length = 1)
  omopgenerics::assertDate(startDate, length = 1, null = TRUE)
  if (overlap & days > 365) {
    cli::cli_abort(c(x = "{.var days} is can not be >= 365 if {.var overlap} is TRUE
  }

  # code ...

}
```

You can throw custom error and warning messages using the **cli** package.

8.4. Conclusions

Validating arguments is a very important step to give user a good experience and prevent running undesired code in big datasets. The **omopgenerics** provides you with some functionality to keep the validation step short and consistent with other packages.

9. Function outputs

9.1. The summarised_result object

The summarised_result object is an S3 class defined by omopgenerics that aims to standardise the result outputs that contain estimates.

9.1.1. The summarised_result basis

The summarised result is a tibble wi

- **Main table**
- **Settings**. Compulsory settings are:
  - *result_type*
  - *package_name*
  - *package_version* Other columns that are always created in settings:
  - *group*, *strata*, *additional*
  - *min_cell_count* this column will always exists (it will be "0" by default).

The *tidy* format, a summarised result can easily be converted to a simpler table, you can use internally this format

9.1.2. How to create a summarised result

Please include only one result_type this make it easy

- way to identify result_type
- unique result_type
- transform to summarise result
- tidy

9.1.3. Suppression

- make sure that we include a number subjects if relevant
- use the word count in estimates

9.1.4. Register the result_type

9.2. Further reading

- The summarised result object omopgenerics.

10. Visualisations

11. Styling your code

12. Reexporting from other packages

Part III.

Contributing documentation

13. Documenting code

Documenting code it is also important, after some weeks we will probably remember why you added some lines of code or what was needed next. Adding notes for yourself or the rest of developers it is always a good idea.

13.1. When to document?

In general not all the lines of code need to be documented, a very simple example can be:

```
y <- x + 1
```

You do not need a comment to know that the value of `y` is going to be `x` plus one unit. But maybe you need to document why you did that for example when we calculate duration it is `end - start + 1` as we want to calculate the number of days (start and end are included) and not the difference (`end - start`). This might not be obvious at first glance and giving a clue to you in the future or any other developer will save you time:

```
# end and start are included in duration, we want number of days not difference
duration <- end - start + 1
```

14. Documenting functions

We use the roxygen2 package for documenting our functions. Using this package will help us to write the .Rd files in the man directory which is what will be rendered when users call `?my_function` or `help("my_function")` and peruse a package website.

Above each exported function we'll create a roxygen **block**, with lines starting with `#'`. The general structure is shown below and we'll go through each in turn.

```
#' Title
#'
#' @description
#'
#' @param x
#' @param y
#'
#' @returns
#' @export
#'
#' @examples
```

14.1. Title

We want to provide a concise yet informative title for our function. The title should typically be written in sentence case but without any full stop

at the end.

If we run `help(package = "mypackage")` we can see all the functions and their titles for a package. From this we can quickly see how informative our titles are.

14.2. Description

Every function should have a desription. This could be a sentence or two for a simple function, or multiple paragraphs for a more complex function. The description should summarise the purpose of the function and explain any internal assumptions or decisions made to implement it.

When writing the description we can use markdown commands, for example to add bullet points, italics, etc. This will often help make longer desriptions more readable.

14.2.1. lifecycle badges

14.3. Parameters

After our description of the function as a whole, we will then describe each of the inputs to the function. In general we should be able to describe an argument in a few sentences. We should use them to tell the user what is allowed as an input and then what the input will be used for.

If we're working on a package where we have multiple arguments with the same argument we can document arguments just once. We can create a helper like below for parameter x.

```
#' Helper for consistent documentation of `x`.
#'
#' @param x
#'
#' @name xDoc
#' @keywords internal
```

We can then reuse this documentation across the package by using `@inheritParams`

```
#' Title
#'
#' @description
#'
#' @inheritParams xDoc
#' @param y
#'
#' @returns
#' @export
#'
#' @examples
```

14.4. Returns

We must also document what the function will return. For example, we can tell a user that our function will return a tibble with a certain set of columns. If our function has side-effects then we chould also document these.

14.5. Export

By including the `@export` tag our documentation will be generated as an `.Rd` file and users will see it in the package website, etc. If instead we want to document an internal function then we can replace this with `@noRd`.

14.6. Examples

Last but by no means least are a set of examples where we provide R code that shows how the function can be used in practice. These examples are quite possible the most important piece of documentation, as it is many users will look at these before reading any of the above.

Typically we should try to give a number of informative examples that show users how the function can be used and provides some intuition on the impact of different arguments.

15. README

The `README.md` is a very important part of an R package. It is probably the first thing the users will read once they decide to try out our package. The `README` must contain the goal of the package and clear examples so users can start using the package. Now we will so how to create it and how to populate it.

15.1. Create the `.Rmd` file

In general we will populate our `README.md` using a rmarkdown file (`README.Rmd`) if you don't have it created you can create it with:

```
library(usethis)
use_readme_rmd()
```

this will create the `README.Rmd` file that generates the `README.md`. To later build the `README.md` you can use the following command:

```
library(devtools)
build_readme()
```

If you are new to *rmakdown* probable this introduction will be useful: Introduction to R Markdown.

> **README.md and README.Rmd**
>
> Note that for cran/website/github only the `README.md` exists, so any change that you do to the `README.Rmd` file wont affect the 'real' README.md file till you build it. Make sure to build the `.md` file every time that you modify the `.Rmd`.

15.2. Content

A good `README.md` file must contain the following sections:

15.2.1. Badges

Badges are quite useful to display at the top of your README, they contain information of the package in a synthetic and consistent way. The badges are generated from `usethis` functions. You will have to manually copy the badge that will be printed in the console in the desired README section, usually at the top:

```
<!-- badges: start -->
PASTE BADGES HERE
<!-- badges: end -->
```

lifecycle status The development status badge can provide the user the development status of the package. We would usually use "experimental" for packages still under development (usually version $< 1.0.0$) and "stable" for stable packages, note if you add the stable badge this sets some expectations about the package like backwards compatibility. A lifecycle badge is highly recommended.

```
use_lifecycle_badge("experimental")
use_lifecycle_badge("stable")
```

cran version The cran badge informs about the current version in cran (if submitted to cran). This badge is also highly recommended (even before submitting to cran).

```
use_cran_badge()
```

build status The badge indicates if the last checks ran in your package passed or not. This badge should also be green otherwise this means the continuous integration is broken. If at some point actions break in main, fix it as soon as possible. This badge is also highly recommended. The badge is generated when GitHub Actions are set.

```
[![Build Status](https://github.com/{user/organisation}/{package_name}/workflows/R-
CMD-check/badge.svg)](https://github.com/{user/organisation}/{package_name}/actions?
CMD-check)
```

test coverage

cran downloads

15.2.2. Tested sources

If your packages is designed to work with different cdm sources (e.g. duckdb, local, postgres, sql server, …) we would encourage you to include the different sources you actively test against. We will later see the Testing multiple DBMS chapter where we explain how to do this automatically using GitHub actions. A good display to show the tested sources would be:

```
| Source | Driver | CDM reference | Status |
|------------------|------------------|------------------|------------------|
| Local R dataframe | N/A | `omopgenerics::cdmFromTables()` | [![](https://github.co
| In-memory duckdb datatabase | duckdb | `CDMConnector::cdmFromCon()` | [![](https:/
| Postgres database | RPostgres | `CDMConnector::cdmFromCon()` |  |
| Postgres database | DatabaseConnector | `CDMConnector::cdmFromCon()` |  |
| SQL Server database | odbc | `CDMConnector::cdmFromCon()` |  |
| SQL Server database | DatabaseConnector | `CDMConnector::cdmFromCon()` |  |
```

15.2.3. Goal of your package

Clearly state the goal of your package, do not extend much, usually 2/3 sentences are enough to describe what your package does.

15.2.4. How to install

Make sure to include information on how to install the package the current release from cran and the development version from GitHub. this section would usually look like:

```
# Installation

{pakage_name} can be installed from CRAN:

  ```r
  install.packages("{pakage_name}")
```

or

```
library(pak)
pkg_install("{pakage_name}")
```
```

If you prefere to use `pak`.

The development version can be installed from GitHub:

```
library(pak)
pkg_install("{user/organisation}/{package_name}")
```

````
### Simple examples

This is the most important section. Catch user attention, show how your package can

We also recommend to include diagrams on how some functions relate or how your packa

## Example

Here we created a simple example of how a good `README` would look like:


`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImRvY3VtZW50YXRpb24ifQ== -->`{=html}

```{=html}
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImRvY3VtZW50YXRpb24iLCJib29rSXRlbVR5c
````
````

16. Vignettes

17. Documentation website

Part IV.

Testing

Testing is a crucial part of R package development. Well-designed tests ensure that each function behaves as intended, protect the package from unexpected breakage when new features are added (new contributions should not cause existing tests to fail), and provide confidence when refactoring code (improving internal logic for efficiency should not alter the expected outputs). A comprehensive test suite also lowers the barrier for contributors: by making expected behaviour explicit, it becomes easier for others to understand how functions should work and to spot unintended changes.

This part of the book introduces good testing practices in the context of OMOP-based package development. It is organised into three chapters:

- Introduction to testing provides a high-level overview of testing in R using modern tools and conventions. This chapter summarises the key concepts presented in the Testing section of R Packages by Wickham and Bryan (https://r-pkgs.org/testing-basics.html).
- Testing in OMOP describes approaches for testing packages that interact with the OMOP CDM instances. In particular, it introduces the use of the omock R package [CITE], which enables the creation of synthetic OMOP data to validate package functionality without relying on a live database.
- Testing against multiple DBMS explains how to configure your package to run tests across multiple database management systems that host an OMOP CDM, ensuring that your code behaves consistently and reliably across different environments.

18. Introduction to testing

Testing is a fundamental part of developing reliable, maintainable R packages. A good test suite provides confidence that your functions behave as expected, prevents regressions when the code evolves, and acts as a form of living documentation of your package's intended behaviour. This chapter introduces the core ideas of testing in R, following tidyverse practices. It focuses on practical techniques you will use in every package: setting up tests, writing expectations, organising helpers and fixtures, deciding what to test, and documenting your tests clearly.

18.1. The Basics

R's modern testing ecosystem centres around the **testthat** package, supported by **usethis** for scaffolding test infrastructure and **devtools** for running tests during development. Together, these tools provide a simple and consistent workflow.

18.1.1. Setting up test infrastructure

The first step is to enable testing for your package:

```
library(usethis)
use_testthat()
```

This command:

- adds *testthat* to your package's `Suggests`
- creates a top-level file `tests/testthat.R`
- creates the directory `tests/testthat/` where test files will live

> **Parallel testing**
>
> Tests are run sequentially in alphabetical order of the files, but you can run those tests in parallel if you use that option:
>
> ```
> use_testthat(parallel = TRUE)
> ```
>
> Note that by default tests will be ran in 2 cores, but you can increase that number setting an environment option `TESTTHAT_CPUS`, see: https://testthat.r-lib.org/articles/parallel.html.

To create a new test file, use:

```
use_test("my_function")
```

This creates `tests/testthat/test-my_function.R`, a template ready for adding tests.

18.1.2. Writing test blocks

Tests are written inside `test_that()` blocks:

```
test_that("my_function adds numbers correctly", {
  expect_equal(my_function(1, 2), 3)
})
```

A good test block:

- has a clear, descriptive name

- focuses on one coherent behaviour
- keeps heavy computation or complex setup outside the block

18.1.3. Running tests

During development, run your tests using:

```
library(devtools)
test()
```

All tests also run automatically during `R CMD check` (`check()`).

Note tests will be run against the current loaded functions, so make sure that you run `load_all()` to load the current/latest version of your package.

18.2. Expectations

Expectations are the core of every test. They express the conditions that must be true for a function to behave correctly. If an expectation fails, testthat produces an informative message.

18.2.1. Common expectations

Frequently used expectations include:

- `expect_equal(x, y)` — checks that objects are equal (with tolerance)
- `expect_identical(x, y)` — strict equality including type and attributes
- `expect_true(x)` / `expect_false(x)` — checks logical conditions

- `expect_error(expr)` / `expect_warning(expr)` / `expect_message(expr)` — checks for specific conditions produced by code
- `expect_s3_class(x, "class")` — checks object class
- `expect_type(x, "double")` — checks base type

18.2.2. Why expectations matter

Expectations define the expected behaviour of your functions. They make assumptions explicit, illustrate expected output, and help maintainers understand the code. They also provide protection against regressions when internal implementations change.

A single `test_that()` block may contain several expectations, as long as they relate to the same behaviour.

18.3. Setup and Helper Functions

As your test suite grows, repeated patterns often emerge: constructing similar inputs, generating mock data, or creating temporary environments. To keep tests clear and maintainable, testthat provides a structured approach for managing shared setup code.

18.3.1. Helper files

Reusable functions should be placed in helper files. Any file named:

```
tests/testthat/helper-*.R
```

is executed before the tests run. These files are ideal for:

- mock datasets

- utility functions
- shared configuration
- frequently used objects

Example helper:

```
# tests/testthat/helper-data.R
make_mock_patient <- function(id = 1, gender = "M") {
  tibble::tibble(person_id = id, gender_concept_id = gender)
}
```

Tests can then use:

```
patient <- make_mock_patient()
```

18.3.2. Setup and teardown

For more complex requirements, testthat supports setup and teardown files:

- `setup-*.R` — executed before tests
- `teardown-*.R` — executed after tests

This pattern is useful for:

- temporary directories
- external resources
- database connections

We will see how to use those files in the Testing against multiple DBMS to tests your package against multiple database magement systems.

18.3.3. Avoid top-level code

You should avoid running code at the top level of test files. Instead, place code inside helpers or `test_that()` blocks. This makes tests more predictable and prevents shared state from leaking between tests.

18.4. What to Test

Good tests focus on what your package promises to do. Tests should be scoped to the behaviour that your package is responsible for—not on implementation details or on the behaviour of external code.

18.4.1. General guidelines

You should test:

- **Normal cases** — typical inputs your users will provide
- **Edge cases** — empty inputs, missing values, boundary conditions
- **Error behaviour** — invalid inputs should produce informative errors
- **Output structure** — class, column names, attributes, lengths
- **Regression tests** — when a bug is fixed, add a test preventing its return, this is a very important step in test development so we do not encounter the same error once again.

> **i** Scope tests to your own code
>
> Avoid testing behaviour of other packages, we want to check that our package not the other packages.

18.5. Documenting Tests

Tests should be readable and easy to understand. A well-written test suite functions as documentation for your package's expected behaviour.

18.5.1. Descriptive test names

The description passed to `test_that()` should be clear:

```
test_that("addAge correctly add age to a table", {
  ...
})
```

This helps identify failures and understand the intention behind the test.

18.5.2. Commenting test logic

Use comments to clarify:

- why specific inputs were chosen
- why an output is expected
- what bug a test prevents from reappearing (e.g. you can refer the issue where the problem was reported)
- key assumptions behind the test

Example:

```
# Missing values should be ignored; this mirrors the documented default.
expect_equal(mean_custom(c(1, 2, NA)), 1.5)
```

18.5.3. Keeping tests readable

To maintain readability:

- keep tests small and focused
- avoid deep nesting
- use helper functions for repeated patterns
- keep example datasets small and simple (we will see how to create mock OMOP CDM datasets in the Testing in the OMOP CDM chapter)

Readable tests make collaboration easier and reduce maintenance effort.

18.6. Test Coverage

Test coverage measures the proportion of your package's code that is executed during testing. It is expressed as the percentage of lines inside your `R/` folder that are run by the tests in `tests/testthat/`.

You can compute test coverage using:

```
test_coverage()
```

This generates an interactive report in the *Viewer* panel showing how many times each line of code was executed during testing, which lines are covered by tests, and, therefore, which lines remain untested.

High test coverage increases confidence that your functions behave as expected across typical and edge-case scenarios. While achieving **100% coverage** is ideal, it is not always practical; deprecated code paths or some edge-case checks may be difficult to test directly.

As a rule of thumb **aim for at least 90%-95% coverage**, in general, try to test all core logic, leaving only truly unavoidable lines uncovered.

Coverage is not a substitute for thoughtful testing, but it is a valuable tool for identifying weak spots in your test suite and ensuring your package remains reliable as it evolves. At the end having a 100% test does not ensures that your package does what you want, it just ensures that it does not break. So think carefully your tests and test the core functionality.

18.7. Testing on CRAN

When submitting a package to CRAN the package is checked (like `check()` function does), all tests, examples, vignettes, and documentation must finish within **10 minutes** of CPU time. Test can consume a substantial portion of this time limit, and it may exceed CRAN's limits.

To keep your CRAN checks within the allowed time while still maintaining a comprehensive local tests, you can selectively skip the most expensive tests on CRAN using `skip_on_cran()`:

test_that("Test mean_custom behaviour", { skip_on_cran() # Missing values should be ignored; this mirrors the documented default. expect_equal(mean_custom(c(1, 2, NA)), 1.5) }) ""

This approach allows you to run **full tests locally** and in continuous integration (e.g., GitHub Actions), but omit heavy or slow tests during CRAN checks, ensuring that only **core functionality** is tested on CRAN, while edge cases, performance tests, and large data scenarios are tested elsewhere.

> **i** Skip functions
>
> The *testthat* package provides several functions for conditionally skipping tests.
> Common examples include:
>
> - `skip_on_cran()`: skip tests on CRAN only.

- `skip_if()`: skip based on a custom condition (e.g. `skip_if(dbToTest == "Postgres")`).
- `skip_if_not_installed()`: skip if a required package is missing (e.g. `skip_if_not_installed("dplyr")`).

See the full list of skip helpers at: https://testthat.r-lib.org/reference/skip.html.

18.8. Further reading

- **Wickham, H., & Bryan, J. *R Packages (2nd ed.): Testing basics (Chapter 13).*** Available online at: https://r-pkgs.org/testing-basics.html
- **Wickham, H., & Bryan, J. *R Packages (2nd ed.): Designing your test suite (Chapter 14).*** Available online at: https://r-pkgs.org/testing-design.html
- **Wickham, H., & Bryan, J. *R Packages (2nd ed.): Advanced testing techniques (Chapter 15).*** Available online at: https://r-pkgs.org/testing-advanced.html

19. Testing in the OMOP CDM

20. Testing against multiple DBMS

Part V.

GitHub

21. Set up GitHub

22. Automatic R-CDM checks

23. Website deployment

24. Test coverage

25. Testing multiple DBMS

Part VI.

Maintaing a package

26. Submit to CRAN

- What version to release (major, minor or patch)
- Forced releases by cran
- news
- reverse dependency