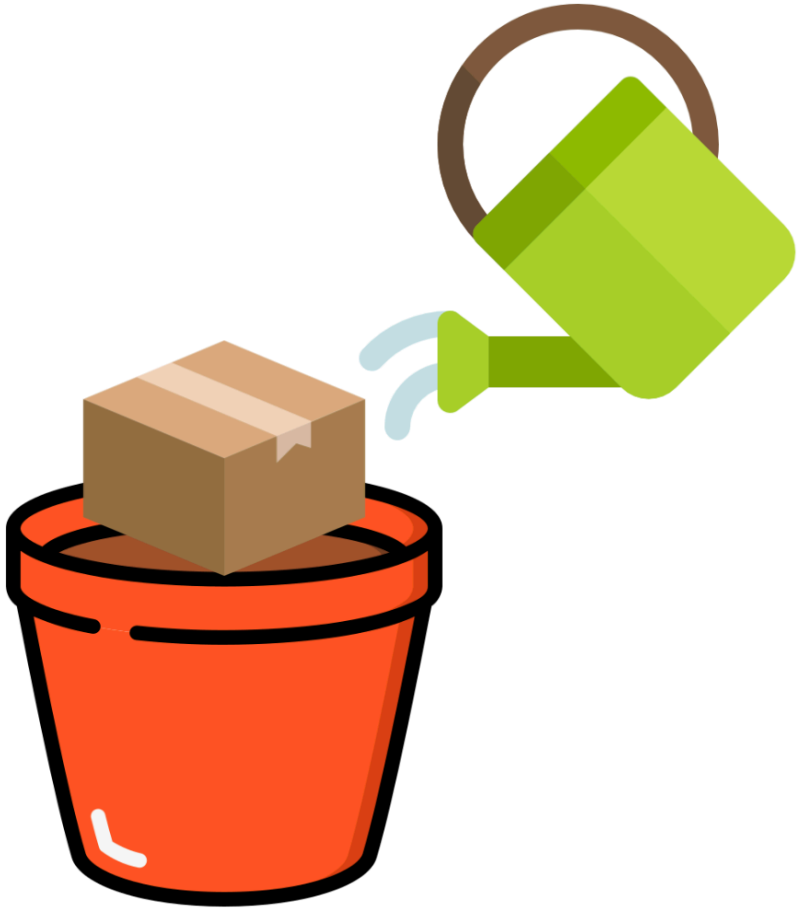
Preventive Care for R Packages

Indrajeet Patil



"Software engineering ought to produce sustainability."

- Mark Seemann (*Code That Fits in Your Head*)

Target audience

As an R package developer, if you ever lay awake in the bed wondering:

- if the users are having a bad experience while using the package,
- if you will receive the dreaded CRAN email about archival, and
- if you will be able to update the package in time,

then this presentation is for you! 😊

Before we begin

Don’t miss the forest for the trees.

It's not about the tools

I will rely heavily on GitHub as the hosting platform and GitHub Actions as the CI/CD framework. Even if you use neither, the broader takeaways should still be relevant. You can implement the necessary checks with preferred tech stack.

It's not *just* about CRAN

Following the recommended practices will make packages more robust to CRAN checks, but that benefit is *incidental*. You can follow these practices even if you never plan to submit to CRAN. The goal here is to improve user experience and reduce maintenance workload.

It's not even about R

The practices outlined here are just as relevant to software development in any other programming language (just replace the package with module/library/etc.).

Iconography

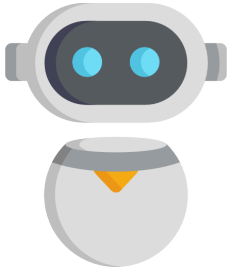
Overarching goal

Problem to solve

Bad user experience

Maintenance headaches

Tools



Automation

Digging the pit of success

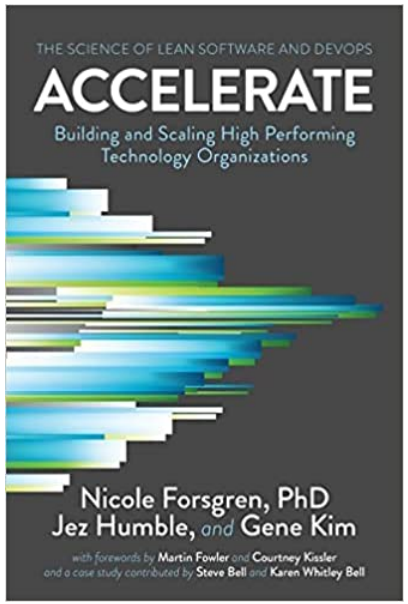
How can software engineering improve sustainability

Be always release-ready

Based on research, *Accelerate* (Forsgren, Humble, & Kim, 2018) argues that the key difference between high-performing vs low-performing software teams is *the ability to make a release at the drop of a hat*.

For R packages, this translates to making sure that every commit on the `main`-branch is **release-ready**.

That is, if you were asked to make a new release soon, you can be confident that the latest commit doesn't have any documentation issues, code quality issues, performance regressions, etc.



How can software engineering help to achieve this goal?

Fighting software entropy

The biggest reason why a software project becomes unsustainable is the unchecked accumulation of complexity.

Software engineering is the active and conscious process of preventing complexity from growing.

It provides the methodology to make sure that the software works as intended and to ensure that it stays that way.

Software development is an inherently complex process. To make it more manageable, we break it down into checklists of best practices—each designed to stave off complexity—so that we don't forget about them.

Following each item on a checklist is a small improvement, but constantly keeping an eye on internal quality prevents software entropy from growing. Although software engineering is more than about automating this process, automation is undoubtedly an important part of it.

“The only way to go fast, is to go well.”

- Robert C. Martin

R package development

Using automation to tick checklists for various aspects of package development.

Plan

First, you will see the checklists to tick, and then the details on how to build infrastructure to ensure that none of the checklist items are forgotten.

- Documentation
- Exception handling
- Portability
- Code quality
- Dependency management

Checklist for documentation

For a good user experience, make sure that the docs are plentiful, valid, and up-to-date.

Item
Make sure there are enough examples in the documentation.
Make sure all README examples are working.
Make sure all examples in help pages are working.
Make sure examples in vignettes are working.
Make sure all URLs are valid.
Make sure there are no spelling mistakes.
Make sure all HTML in the help pages is valid.

Checklist for exception handling

To reduce maintenance headaches, make sure that warnings are easily detected for further scrutiny and forthright dealt with.

Item
Make sure examples in README produce no warnings.
Make sure examples in help pages produce no warnings.
Make sure examples in vignettes produce no warnings.
Make sure tests produce no extrinsic warnings.

Checklist for portability

For a good user experience, make sure that package would work as expected across diverse settings.

Item
Make sure package passes checks on commonly used OS.
Make sure package passes checks on all supported R versions.
...

The items on this list will vary significantly from package to package (e.g., does it work in different locales, with different compilers, etc.). Extend the list for your workflow!

Checklist for code quality

To reduce maintenance headaches, make sure that the code is readable, maintainable, and follows agreed conventions.

Item
Make sure the code follows a style guide.
Make sure there are no known code quality issues.
Make sure there are no performance regressions.

Checklist for dependency management

To reduce maintenance headaches, make sure that the package is robust to availability of soft dependencies and breaking changes in hard dependencies.

Item
Make sure examples in help pages are run conditionally.
Make sure vignettes (or code chunks therein) are executed conditionally.
Make sure tests are run conditionally.
Make sure excluded vignettes (or code chunks therein) are executed conditionally.
Anticipate possible breaking changes coming from dependencies and act on it.

Documentation

Preventive care to make sure the docs are up-to-date.

“Incorrect documentation is often worse than no documentation.”

- Bertrand Meyer

For a good user experience, make sure that the docs are plentiful, valid, and up-to-date.

Make sure there are enough examples in the documentation.

An example is worth a thousand words

Access to abundant examples in help pages and vignettes provides a natural starting point for users to explore and experiment with the available functionality.

Good examples are difficult to write, but any examples are better than none.

Without enough expository examples, users are left to fumble their way into discovering the available functionality.

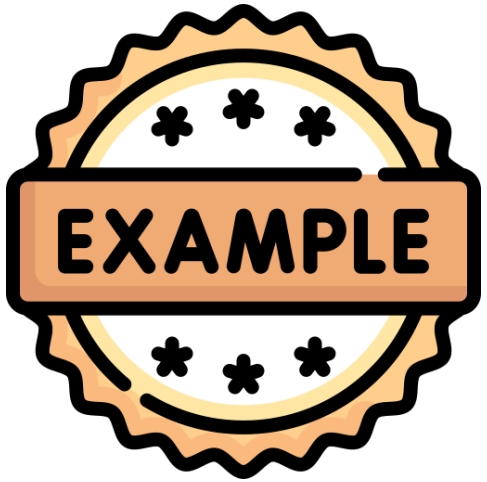
How to ensure that there are *enough* number of examples?

Sources of documentation

There are three types of documents that constitute package documentation.

README.md

Help pages



Vignettes

Therefore, we want to make sure that, when combined across these sources, there are enough examples to cover a significant proportion of available functionality.

Maintaining example code coverage

Use `{covr}` to compute example code coverage (i.e. proportion of the source code that is executed when running examples in help pages and vignettes), and to ensure that it is above a certain threshold.

```
package_coverage(type = c("examples", "vignettes"), commentDonttest = FALSE, commentDontrun = FALSE)
```

Make sure all README examples are working.

README documentation

`README.md` provides a quick overview of the package API and can feature examples of key functions.

Although breaking changes might be infrequent, when they do occur, the code in README may become defunct.

`README.md` is probably the first and the most-visited document in a project and any broken examples therein are bound to confuse many users.

How to insure against broken code in README?

Detecting broken README examples

Use `{rmarkdown}` to dynamically generate `README.md` from `README.Rmd`. If there is broken code in README, it will fail to render.

```
rmarkdown::render("README.Rmd", output_format = rmarkdown::github_document())
```

Make sure *all* examples in help pages are working.

Broken examples in help pages

Types of examples

Help pages for exported functions should contain examples illustrating their usage. But you can skip executing some examples (e.g. because they are too time-consuming) using any of the following tags:

Tag	Run by `example()`?	Run by `R CMD check`?
`\dontrun{}`	❌	❌
`\donttest{}`	✅	❌
`\dontshow{}`	✅	✅

Thus, broken `\dontrun{}` and `\donttest{}` examples may not be flagged by `R CMD check`. Users will still see these examples, and wonder why they aren't working for them.

CRAN's additional "donttest" check does run `\donttest{}` examples.

How to catch examples that don't run successfully?

Checking all examples

Use `{devtools}` to run all examples, and catch and fix the broken ones.

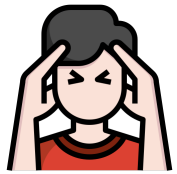
```
devtools::run_examples(run_dontrun = TRUE, run_donttest = TRUE)
```

Make sure examples in *all* vignettes are working.

Vignette examples

Types of vignettes

Vignettes included in the package will fail if examples are broken. But you can choose to exclude some vignettes to reduce package size or to reduce check time. You can do so by placing them in a `vignettes/` subdirectory or by adding them to `.Rbuildignore`.



Thus, the `R CMD check` won't catch broken examples in excluded vignettes. Although users might not see them on CRAN, you shouldn't retain defunct examples anywhere in the documentation. This also makes excluded vignettes future-proof; if you decide to include them in the package, there won't be any issues.

How to detect broken examples in excluded vignettes?

Building all vignettes

Although excluded vignettes may not be checked by `R CMD check`, they are still built by `{pkgdown}` to generate a static website. Thus, building a website would detect any broken examples in excluded vignettes.

```
pkgdown::build_site()
```

Make sure *all* URLs are valid.

What is link rot?

Package documentation often includes plenty of hyperlinks to external resources. But some of them may become invalid over time.

Link rot happens because web pages move to new addresses or become permanently unavailable.

Such dangling references can be frustrating for the users trying to access the resources that the dead links were previously pointing to.

How to prevent link rot from accumulating in the documentation?

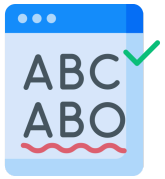
Detecting link rot

You can use `{urlchecker}` to detect dead web references and their locations in the documentation. Fixing these links is then straightforward.

```
urlchecker::url_check()
```

Make sure there are no spelling mistakes.

Spelling mistakes



Spelling mistakes are inevitable and, if left unchecked, they can accumulate rapidly with the increase in the documentation.

Spelling mistakes obvious to native speakers may not be so for non-native speakers, who will be frustrated that they can't find the meaning of the misspelt word.

Additionally, misspelling technical words (e.g. *innode* vs. *inode*) can lead users down the wrong path and waste their time.

How to prevent spelling mistakes from accumulating in the documentation?

Creating a list of allowed misspellings

There exist multiple English spelling standards (e.g. in British English: *anaemia*, but in American English: *anemia*). You can specify your preferred standard in `DESCRIPTION`.

E.g. for British English

```
Language: en-GB
```

Detecting spelling mistakes

Use `{spelling}` to detect misspelt words and their location in the docs.

```
spelling::spell_check_package()
```

Make sure *all* HTML in the help pages is valid.

HTML5 check

HTML has few syntactic rules, and browsers may not even enforce the rules it does have. E.g., browsers will tolerate mismatched tags, missing end tags, misnested tags, etc. HTML5 standard aims to improve this situation, and R package help pages use HTML5 standard since `R 4.2`.

You will thus need to make sure that your package documentation neither generates invalid HTML nor contains raw invalid HTML.

How to make sure help pages have valid HTML?

Detecting invalid HTML

CRAN uses HTML tidy to detect markup errors in HTML. But, if you use `{roxygen2}`, it will ensure against producing any invalid HTML.

```
roxygen2::roxygenise()
```

Checklist for documentation

For a good user experience, make sure that the docs are plentiful, valid, and up-to-date.

Item
Make sure there are enough examples in the documentation.
Make sure all README examples are working.
Make sure all examples in help pages are working.
Make sure examples in vignettes are working.
Make sure all URLs are valid.
Make sure there are no spelling mistakes.
Make sure all HTML in the help pages is valid.

Exception handling

Preventive care to make sure that you don’t miss out on important warnings.

“There is a problem with warnings. No one reads them.”

- Patrick Burns

To reduce maintenance headaches, make sure that warnings are easily detected for further scrutiny and forthright dealt with.

Sending signals

Types of conditions/exceptions

A function can use conditions to signal that something unexpected has happened with varying severity.

Condition	Severity	Meaning
`error`	high	execution stopped because there was no way to continue
`warning`	medium	execution encountered some problem but recovered
`message`	low	execution was successful and here are some extra details

Out of these, warnings are the most nebulous!

- Errors bring functions to a halt and you *must* attend to them.
- Messages are innocuous and you *can* safely ignore them.
- But warnings are harbingers of problems that you will need to fix at some point. They *need* to be dealt with, *pronto*, and yet it is easy to ignore them.

A needle in the haystack

Types of warnings

There are two kinds of warnings that you, as a developer, will need to deal with:

Intrinsic warnings are warnings produced by functions in *your* package.

E.g. a warning from a function to winsorize data.

```
winsorize(x, threshold = 2)
#> Warning message:
#> `threshold` for winsorization must be a scalar between 0 and 0.5.
```

Extrinsic warnings are warnings stemming from your package *dependencies*.

E.g. a possible warning if your package relies on `{ggside}`.

```
ggplot(mpg, aes(hwy, class)) + geom_xsidedensity()
#> Warning: Using the `size` aesthetic in this geom was deprecated in ggplot2 3.4.0.
#> i Please use `linewidth` in the `default_aes` field and elsewhere instead.
```

Suppressing intrinsic warnings

There is almost never a need to explicitly highlight warnings intrinsic to your package.

- Warnings are generated in contexts where functions in *your* package were used unexpectedly by the users. But such contexts shouldn't be deliberately highlighted in the documentation. Users should always see happy path examples in help pages, README, or vignettes.
- While testing functions, you should use expectations (e.g. `testthat::expect_warning()`) to check that expected warnings are triggered. You shouldn't print the warnings, since they can completely overwhelm the test log and make it difficult to catch important warnings.
- If it's a complex package function that can encounter a large number of slippery situations, it makes sense to provide an argument to turn off a few warnings (e.g. `verbose = FALSE`) and use it to omit warnings in the docs.

Suppressing extrinsic warnings

Warnings from dependencies can be critical and should be dealt with ASAP.

- If dependencies are emitting warnings because your functions are using imported code in unexpected ways, rewrite functions to remove warnings.
- If the warnings are about deprecated functions or arguments, switch to using the suggested alternatives. Don't wait until they are removed.
- Some warnings are unavoidable and not as important. They can be suppressed using `suppressWarnings()`. That said, avoid using it in examples in help pages, lest users think that this is part of *your* package API.
- If the warnings are coming from somewhere upstream (e.g. `{ggplot2}` → `{ggside}` → your package), you have little control over them. You can inform the upstream maintainer and ignore such warnings using `suppressWarnings()`. Don't forget to remove suppress calls once the warnings are fixed upstream.

Detecting warnings

Convert warnings into errors during checks to detect warnings.

```
options(warn = 2L)
```

Checklist for exception handling

To reduce maintenance headaches, make sure that warnings are easily detected for further scrutiny and forthright dealt with.

Item
Make sure examples in README produce no warnings.
Make sure examples in help pages produce no warnings.
Make sure examples in vignettes produce no warnings.
Make sure tests produce no extrinsic warnings.

Portability

Preventive care to make sure that your package works across a variety of settings.

“Each new user of a new system uncovers a new class of bugs.”

- Brian Kernighan

For a good user experience, make sure that package works as expected across diverse settings.

The unbearable diversity of contexts

You (the developer) may be developing the package in a certain setting: with a certain version of R, on a particular OS, in a certain locale, etc. Even if all tests pass and all examples run successfully for you locally, you can't assume that your users will use the package in similar settings.

If you restrict your checks only to these specific settings, you may not catch problems experienced by users in other settings (e.g., the graphics device that works on `windows` may not work on `macOS`, code that works with `R 4.1` may not work with `R 3.6`, etc.).

The key assumption here is that your package claims to support these settings. If your package docs clearly state that the package will work only on (e.g.) Windows, you don't need to worry about other OS.

How to make sure that your package is working as expected across various settings?

Checking across multiple settings

Use `{rcmdcheck}` to run `R CMD check` from R.

```
rcmdcheck::rcmdcheck()
```

Checklist for portability

For a good user experience, make sure that package would work as expected across diverse settings.

Item
Make sure package passes checks on commonly used OS.
Make sure package passes checks on all supported R versions.
...

The items on this list will vary significantly from package to package (e.g., does it work in different locales, with different compilers, etc.). Extend it for your workflow!

Code quality

Preventive care to detect code quality issues and performance regressions.

“Don’t comment bad code —rewrite it.”

- Brian W. Kernighan

To reduce maintenance headaches, make sure that the code is readable, maintainable, and follows agreed conventions.

Make sure the code follows a style guide.

Code formatting

The physical layout of the program can assist the reader in understanding the underlying logic. Additionally, understanding a large codebase is easier when all the code has consistent formatting. Style guides outline conventions to enforce a uniform formatting schema across the codebase.

Style guides can be highly opinionated and arbitrary. So, more important than *which* style guide you follow is the fact that you follow *a* style guide.

In larger projects or teams, multiple contributors may have different formatting preferences. This can lead to contributors undoing each others' changes, leading to unnecessarily large git diffs and even unnecessary unpleasantness.

How to make sure that codebase follows a consistent style guide?

Following style guide

Use `{styler}` to enforce the tidyverse style guide throughout the package (including source code, test files, vignettes, etc.).

```
styler::style_pkg()
```

Make sure there are no known code quality issues.

Code quality assessment

Code smells (aka lints) are patterns that are known to be problematic for readability, efficiency, consistency, style, etc. Catching such issues early on can help prevent bugs and other issues from creeping into code, which can save time and effort when it comes to debugging and testing.

```
# code with a lint
lint(text = "x = 1")
```

```
# code without a lint
lint(text = "x <- 1")
```

Detecting lints

Use `{lintr}` to carry out static code analysis to detect code quality issues.

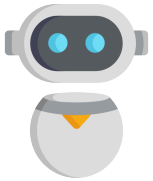
```
lintr::lint_package()
```

Detecting issues with non-R code

An R package contains a number of configuration files written in languages other than R (e.g., YAML, JSON, DCF, etc.), and you may also wish to make sure that none of them are malformed.

To detect multi-language code issues, you can use pre-commit, which is a framework for managing and maintaining git hooks. This framework can be accessed in R using `{precommit}`.

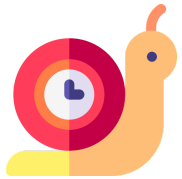
The framework offers hundreds of hooks to choose from and you can choose ones relevant to your config files (e.g. to lint and format JSON and YAML files).



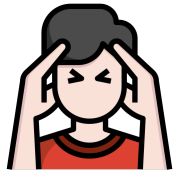
Use GHA workflow to detect any problems with non-R code on each commit. You can specify the hooks relevant to you in a pre-commit config file.

Make sure there are no performance regressions.

Don’t slow it down



If your package is mature and stable enough that you have started to invest in improving its efficiency, it is important that you have some metric by which you can benchmark if a new Pull Request improves or degrades performance in comparison with the latest commit on `main`.

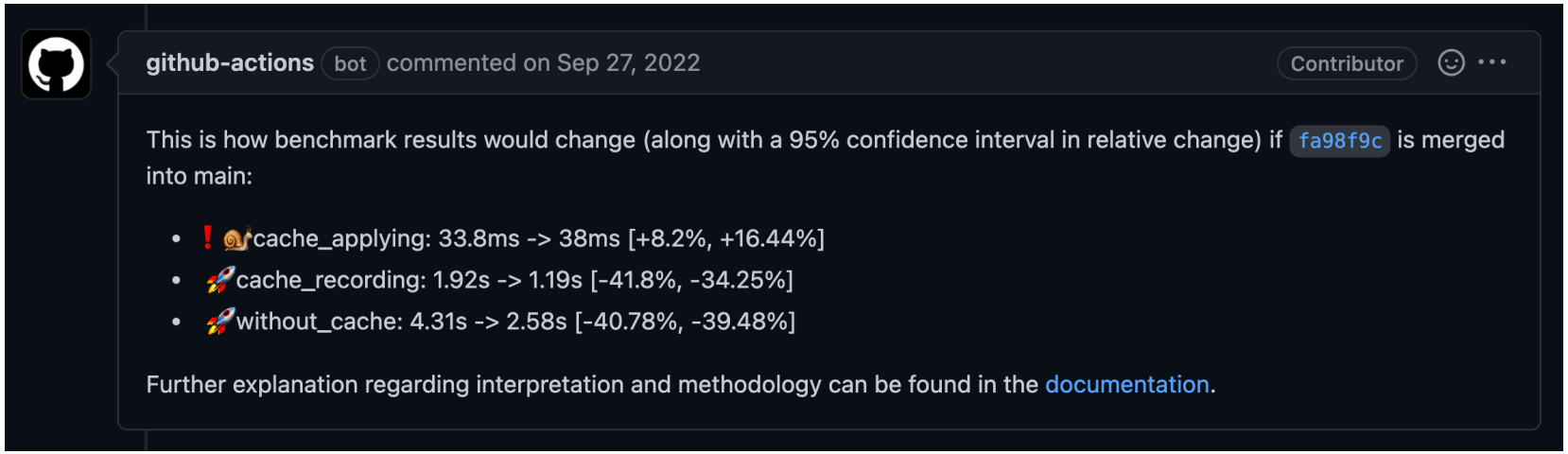


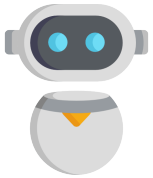
Checking for performance regressions just before or even after the release is not ideal, since it might be difficult to revert back to the state before regression took place.

How to reliably benchmark Pull Requests for performance regressions?

Detecting performance regressions

`{touchstone}` offers a continuous benchmarking tool for reliable relative measurement of performance in the Pull Request versus `main`-branch. The results are directly reported as a comment in GitHub Pull Requests.





Use GHA workflow for benchmarking PRs and detecting any performance regressions.

Checklist for code quality

To reduce maintenance headaches, make sure that the code is readable, maintainable, and follows agreed conventions.

Item
Make sure the code follows a style guide.
Make sure there are no known code quality issues.
Make sure there are no performance regressions.

Dependency management

Preventive care to make sure that all suggested (aka weak or soft) dependencies are used conditionally.

“Dependencies are invitations for other people to break your package.”

- Joshua Ulrich

To reduce maintenance headaches, make sure that the package is robust to availability of soft dependencies and breaking changes in hard dependencies.

Types of Dependencies

Dependencies (code that your source code relies on) are an inevitable part of package development.

Not all dependencies are created equal!

Hard dependencies have a broader scope because they are needed at runtime; i.e. your package won't work without them, while soft dependencies have a narrow scope; e.g. because they are needed only for testing or for examples.

Dependency	In `DESCRIPTION`	Scope
Hard	`Depends`/`Imports`	Needed for your package to work as expected.
Soft	`Suggests`	Nice to have but your package will work regardless.

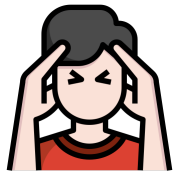
Dependencies bring risk

If a critical dependency becomes unavailable (e.g. because its author decides to archive it), bad luck. You **must** either refactor to remove dependency or look for an alternative. Otherwise, your package is no longer going to work. But this shouldn't be the case for soft dependencies since they are not critical for your package to work.

Make sure soft dependencies are used conditionally.

Soft dependency hygiene

In a high-level, user-facing package, there can be a substantial number of soft dependencies. The package then risks breakage for *any* unavailable soft dependency, if it is not used conditionally.



It can be quite taxing for maintainers to remove examples or tests related to a soft dependency if it gets archived, and restore them once the dependency is unarchived. To avoid this, soft dependencies should always be used conditionally.

CRAN runs the additional "noSuggests" check to look for possible breakages.

How to make sure *all* soft dependencies are being conditionally used?

Conditional dependency usage

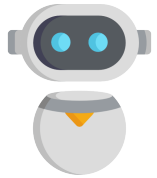
Soft dependencies should be used conditionally across all contexts. Let's say `{lme4}` is such a dependency.

In `DESCRIPTION`

```
Suggests:
    lme4
```

Checking conditional usage

The trick here is to run `R CMD check` twice: once when *all* soft dependencies are available, and once when *none* are available. If the package passes the `R CMD check` in the first but not the second context, you can be sure that some soft dependencies are not being used conditionally.



Use GHA workflows to install *all* dependencies or *only* hard dependencies and check if the package passes `R CMD check` on each commit.

Additional tips

- This trick won't work for excluded vignettes (from `vignettes/` subdirectory or `.Rbuildignore`-ed). To ensure that excluded vignettes are using soft dependencies conditionally, build package website in "noSuggests" mode.
- For `R CMD check` with all dependencies installed, use strict workflow that fails on any `NOTE`. To avoid failing on `NOTE`s accepted by CRAN, include this in `DESCRIPTION`:

```
Config/rcmdcheck/ignore-inconsequential-notes: true
```

Special care for examples

Prefer using `\donttest()` over `\dontrun()` for skipping examples.

This is because the latter will be skipped during R CMD Check and `{pkgdown}` also will not execute these examples. Therefore, you will miss out on an example that should be run conditionally. This can become an issue if you do decide to run this example in the future by removing `\dontrun`.

E.g., assuming `{lme4}` is a soft dependency and is not available:

Won't fail in "noSuggests" workflow

```
#' @examples
#' \dontrun{
#'   lme4::lmer(...)
#' }
```

Will fail in "noSuggests" workflow

```
#' @examples
#' \donttest{
#'   lme4::lmer(...)
#' }
```

Anticipating breakages

It is important to anticipate changes in dependencies that may break your packages. Although some maintainers (of your dependencies) will be kind enough to let you know of upcoming breaking changes, such communication can't be assumed or might not even be possible.

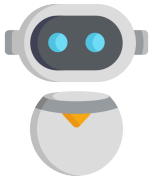
If your package is not on CRAN, there is no way for the maintainers to detect breakages in reverse dependencies and inform maintainers. Similarly, if a reverse dependency is skipping tests on CRAN, possible breakages can go undetected.

You may figure out that something is broken *after* a dependency is updated, the package stops working for the users, and they inform you. This is especially true if your package is not under active development, and so CI/CD won't detect that your package is broken.

How to detect upcoming breaking changes in dependencies?

Detecting breakages early

In order to detect breakages earlier, you can run `R CMD check` by installing development versions of dependencies.



Use scheduled GHA workflow to automate checking breakages.

How frequently you should run this check (e.g., once a month, once every six months, etc.) and which dependencies you should include (e.g., only hard, only soft, a few of each, etc.) depends on how actively both your own package and your dependencies are being developed.

What should you do if you *do* detect a breakage?

- If the root cause of breakage turns out to be a regression in the dependency, discuss with the maintainer.
- If the breakage is legitimate, make a PR to your repo with a fix but merge only when either the maintainer informs you of an upcoming release or the package has a new release. **Don't** push the fix to the `main`-branch because the breaking change could be reverted before release.

Checklist for dependency management

To reduce maintenance headaches, make sure that the package is robust to availability of soft dependencies and breaking changes in hard dependencies.

Item
Make sure examples in help pages are run conditionally.
Make sure vignettes (or code chunks therein) are executed conditionally.
Make sure tests are run conditionally.
Make sure excluded vignettes (or code chunks therein) are executed conditionally.
Anticipate possible breaking changes coming from dependencies and act on it.

Eco-friendly workflows

Do you *really* need to run all workflows on each commit?!

Skipping workflows

Although all these workflows can be run for free (thanks Microsoft!), they still expend energy and it might bother you that you are wasting energy even on minor changes.

Example of an insignificant change.

```
# In `NEWS.md`
- witdh
+ width
```

Caveats

No good practice is dogma. There always exist exceptions.

Using workflows flexibly

- You can skip some workflows or create new workflows depending on the project-specific demands (e.g. compiled code, database connections, API access, etc.).
- Sometimes it might not even be possible to run all workflows successfully because a few of them conflict with each other.

If you have a data visualization package (e.g.), the more examples you have in the documentation, the bigger the package size would be, which might leave a `NOTE` in `R CMD check` and strict workflow will fail. Which of these workflows is more important is a subjective decision.

Failure is the only option

It is necessary that builds fail for any existing or newly found issues.

“Later equals never.”

- LeBlanc’s Law

Managing technical debt

Tomorrow never comes

It might seem excessive that workflows fail if *any* issue is found (*"Really?! You want builds to fail even if there is a single code quality issue?!"*).

But this is the only way to sidestep procrastination (*"We can fix the broken link later!"*) that can lead to the accumulation of technical debt. With time, this debt can compound and every new feature requires longer to implement.

Failed builds also act as organizational quality control mechanisms. You will no longer need to justify carving out time to address technical debt if a new release can't be made unless all checks are green.

Once green, always green

The initial work you will put into achieving green checks will pay dividends in the long run. And, once a check is green, you only need to make sure that it stays that way for each new commit to the `main`-branch.

Easing into it

It might not be feasible to implement all workflows in one go, especially when they will be marked as failed until *all* relevant issues have been dealt with.

This is especially true for organizations that have a policy to always keep the default branch “green”.

I’d highly recommend that you adopt such a policy, both for your organization and private repositories.

You can adopt an **incremental approach** of adding one workflow per PR.

This PR can be merged when the new workflow runs successfully.

Conclusion

It is possible to build robust automation infrastructure for R package development that can improve user experience and make long-term development more reliable and sustainable.

Further reading

For more extensive discussions on best practices in software development.

R-specific

- Writing R Extensions
- Wickham, H., and Bryan, J. (2023). *R Packages* (2nd edition). O'Reilly.

Language-agnostic

- McConnell, S. (2004). *Code Complete* (2nd edition). Microsoft Press.
- Martin, R.C. (2017) *Clean Architecture*. Addison-Wesley.
- Ousterhout, J. K. (2018). *A Philosophy of Software Design*. Palo Alto: Yaknyam Press.
- Seemann, M. (2021) *Code That Fits in Your Head*. Addison-Wesley.

Source code for these slides can be found on GitHub.

Star the repo and share with others if you liked the slides!

😊

For more

If you are interested in good programming and software development practices, check out my other slide decks.

Acknowledgements

Thanks to all creators, maintainers, and contributors for the tools mentioned throughout the presentation. Without them, it wouldn't be so easy to create robust package development architecture in R! 🙏

All images used in these slides have been taken from *Flaticon* (www.flaticon.com) by *freepik* (www.freepik.com). Huge thanks to them for making such fantastic resource freely available.

Find me at...

- Twitter
- LikedIn
- GitHub
- Website
- E-mail

Thank You

And Happy Package Care! 👼

Session information

```
sessioninfo::session_info(include_base = TRUE)
```