Electronic Communications of the EASST
Volume 83 Year 2025

deRSE24 - Selected Contributions of the 4th Conference for Research Software Engineering in Germany

Edited by: Jan Bernoth, Florian Goth, Anna-Lena Lamprecht and Jan Linxweiler

A concise guide to good practices for automated testing and documentation of Research Software

Jakob Fritz, Michele Mesiti, Jan Philipp Thiele

DOI: 10.14279/eceasst.v83.2622

A concise guide to good practices for automated testing and documentation of Research Software

Jakob Fritz[1], **Michele Mesiti**[2] **and Jan Philipp Thiele**[3]

[1] j.fritz@fz-juelich.de
Jülich Supercomputing Centre
Forschungszentrum Jülich, Jülich, Germany

[2] michele.mesiti@kit.edu
Scientific Computing Center (SCC),
Karlsruher Institut für Technologie (KIT), 76128, Karlsruhe, Germany

[3] thiele@wias-berlin.de
Weierstrass Institute Berlin (WIAS), 10117, Berlin, Germany

Abstract: This publication aims to highlight aspects of good practices from the areas of testing, documentation, and workflow management to improve existing code in Research.

Applying these practices helps to make the scientific results obtained with the software easier to reproduce, as the codebase is easier to understand (thanks to documentation) and it is ensured that the code works as expected (thanks to automated testing). As codebases grow larger, workflow management for automated testing is also needed in order to keep the development cycle fast.

Keywords: RSE, Continuous Integration, Testing, Documentation, Workflow

1 Introduction

The usage and importance of software has been noted in nearly all fields of science. Scientific software is also becoming more and more complex as the requirements become more demanding, and this underlines the importance of following good (Research) Software Engineering techniques to manage the increasing complexity. As Research Software Engineering has many different aspects, this publication puts a focus on three areas that help to deal with increasing complexity in software, where the authors think more emphasis should be put on in everyday development, namely how to write documentation, how to test (automatically) and how to design and manage automated testing workflows. These aspects complement each other well and should not be interpreted to exclude or contradict each other.

Although many of the aspects highlighted in this publication also apply to software in general, some are especially relevant to scientific and research software. Documentation of scientific code often represents a possible entry point to understand a scientific subject. High turnover rates in academia also require software to be well documented so that research projects depending on it can survive the departure of developers. Proper automatic testing is also known to be crucial in that regard, as it helps to ensure that the inevitable modifications needed to keep research software alive do not break features previously implemented. For an introduction to the peculiarities of Research Software Engineering compared to Software Engineering, please also have a look at [Lam23] and [CH24].

This publication does not aim to give detailed insights in the latest corner cases of the presented topics, but rather aims to emphasize the basics and how and where to start when including them in the practice of Research Software Engineering.

2 Documentation

In this Section we present a survey of the tooling and resources available to develop and distribute documentation, and a discussion of the challenges in maintaining it.

In particular, we will briefly discuss (or give references about) various aspects regarding documentation, i.e.:

- best practices in writing it;
- the file formats that are typically used;
- a selection of the tools that are typically used to produce it;
- the services available to host it online and their capabilities;
- problems and techniques in the automatic test of documentation.

2.1 Best practices in writing maintainable documentation

Documenting code efficiently and effectively is an important problem for which numerous tools have been developed and many guides have been written, also available online. A succinct yet useful introduction to best practices in writing documentation is maintained by the "Write the Docs" community [wri] and a practical tutorial is also part of the core offering of CodeRefinery [Cod]. As an example of the relevance of the topic in the deRSE community, Jessica Mitchell's talk [Mit] as part of the HiRSE_PS seminar series also describes challenges and best practices.

The consensus is that writing documentation once is not sufficient, and resources must be allocated during the lifetime of the software to also maintain the documentation, in addition to the software itself [wri]: this means that it is very important to adopt strategies that reduce the maintenance cost. To this aim, documentation should be automatically tested whenever possible (see Section 2.5), and excessive documentation should be considered a liability, not an asset (as is typically said about code[Ott]): giving descriptive names to entities in your code should be preferred to using code comments [MC09]. For a better traceability of the provenance of information (which can also be important for maintainability), links to existing materials should be always included when appropriate, and rewriting existing material should be avoided[1]. On the other hand, the minimalistic approach necessary to maximize maintainability might conflict with

[1] From the agile manifesto: "Simplicity - the art of maximizing the amount of work not done - is essential." (Accessed 2024-05-17)

didactic purposes (which typically require adapting and combining existing material to optimize learner performance) and, more in general, it is actually a good practice to tailor documentation to the intended audience [wri] (that can be either Research Software Engineers interested in extending the software or scientists interested in using it without modifications).

As a great example, the deal.II PDE/FE library [ABD$^+$21] includes a tutorial that can be viewed in multiple ways. The tutorial is composed of source code examples that are very clearly and didactically commented, and can be studied and modified by a practitioner in their editor. Two additional views of the documentation are also available: an HTML view for better readability in a browser, which can be appropriate when studying the underlying numerical methods, and a view where the source code has been automatically stripped of all the comments which can be appropriate for a user that already knows the numerical techniques and just needs to understand how to use the library. The HTML view is also available at the `dealii.org` website[2].

For users that need to either debug, extend or use the software beyond its originally intended applications *the test suite* represents a useful source of documentation that is by necessity more rigorous than usual documentation needs to be. A similar role is taken by a suite of examples that can be executed (e.g., see the CUDA samples repository[3]).

When developing software from scratch, an alternative approach named *Tutorial Driven Development*[Woo] has been proposed, where documentation is written before the code in the form of a tutorial that is used as a communication vehicle between researchers and Research Software Engineers developing the software, and acts as an integration test (this technique has been likened to user-centered design). The Tutorial Driven Development approach in particular can benefit from the use of specific tools which will be described later. In addition to the maintainability increase which is also common with Test Driven Development, this approach can lead to a software design which is a better fit to the real needs of the user.

In general, a structural match between the semantics of the content being conveyed and its textual representation is desirable. In addition to proper paragraph and file/chapter segmentation that applies also to non-technical writing, a practice that is often suggested is the use of Semantic Line Breaks ([Rho],[Mat]), a best practice that can be traced back to early Unix user guides [Ker].

2.2 Formats

Documentation can be written using many different formats. In the *Docs-as-Code* approach documentation is written using tools that can be well integrated into the software development workflow and can be easily processed by documentation generation tools.

For this reason, usually plain-text formats (as opposed to, e.g., `.doc` or `.odt`) are preferred.

The two mostly used formats for documentation are *MarkDown* [GS] and *reStructuredText* [Goo] (also abbreviated as ReST).

MarkDown is focused on direct writability and readability, making processing and structure a second class priority (from [GS]):

> The overriding design goal for Markdown's formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be pub-

[2] https://dealii.org/developer/doxygen/deal.II/Tutorial.html
[3] https://github.com/NVIDIA/cuda-samples

> lishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions.

The specification of the syntax is simple on purpose and relies on embedded HTML for more advanced features (from [GS]):

> HTML is a publishing format; Markdown is a writing format. Thus, Markdown's formatting syntax only addresses issues that can be conveyed in plain text.
>
> For any markup that is not covered by Markdown's syntax, you simply use HTML itself.

Due to this limitation, the format has been extended by many different flavors that handle different features more natively (e.g., tables, mathematical formulae, code examples), and this has led to a fragmentation of the use cases, which has anyway not hindered adoption[4]. Different flavors of Markdown are in use in different communities, depending on the tool chain used. There are efforts to create a standard version of Markdown, for example CommonMark [M+].

ReStructuredText instead trades writability for control, is properly standardized, includes a proper extension mechanism, and it has been widely used in the Python community (its reference implementation is a component of the Docutils framework).

It must be noted that both text formats can typically be used in code comments and docstrings that can be automatically processed by documentation generation tools to produce, for example, API references. This possibility is supposed to facilitate keeping documentation and code in sync, and to ease writing documentation in the first place as developers do not need to switch to a separate documentation file and to another tool chain.

Automatic conversion of documents from one format to another can be possible via Pandoc [Maca]: this makes it possible, e.g., to write documents in the most convenient form (typically Markdown) and have it converted to PDF (via *tex) or HTML.

There are some documentation formats that are tool-specific and will be briefly mentioned in the description of the corresponding tool.

Different Markdown flavors and ReST support mathematical formulas. In many Markdown flavors mathematical expressions can be rendered correctly when enclosed between dollar sign `$` delimiters (for inline expression) or in a code block marked with `math`. ReST supports mathematical formulas as well, either in `math::` blocks or inline with ``:math:`...'``.

2.3 Documentation systems

In this section, we will discuss documentation tools used for the documentation of scientific code. A similar discussion has been done to choose a documentation system for Hexatomic [DKLB23], and it is available as part of the project meta-documentation[5].

Typically, every community centered on a specific programming language has developed their own documentation generation tool.

[4] Even established journals such as the Journal of Open Source Software [JOS] - particularly relevant for the RSE community - uses it.

[5] https://hexatomic.github.io/about.html

Doxygen [vH] is the main documentation tool used in the C/C++ community. There is support for other languages as well (most notably FORTRAN), although with some limitations.

The syntax used by Doxygen can be described as roughly Markdown with custom additions. There is a variety of output formats available, including XML, which allows Doxygen to work as a front-end of a more complex tool chain (most notably, this allows Doxygen to interoperate with Sphinx through the Breathe plugin [J+], which also allows for ReST in the Doxygen markup).

One of the most useful features of Doxygen is the ability to generate API reference documentation that includes information extracted from the comments in the code and from the code itself. Doxygen can also render static call graphs and representations of class hierarchies. As an example of scientific code documentation that uses Doxygen, please refer to the already mentioned deal.II documentation at the `dealii.org` website[6].

Sphinx [B+] has typically been used in Python projects. It typically supports reStructuredText via Docutils[GGM+], but it has recently started supporting MarkDown as well (in the specific MyST flavor - supported by the Executable Books Project[7]). Sphinx can also render Jupyter notebooks and include them in the generated documentation (via the nbsphinx plugin[8], which is Pandoc [Maca] based, or MyST-NB[9]). As examples of this, see the documentation of the Parafields package [KKK+23] and of the py4dgeo package [pDCT23]. Like Doxygen, Sphinx is able to generate API reference documentation including information parsed from the code and the docstrings. This can be done natively for Python code, and with the appropriate plugins source code written in other languages can also be processed in a similar fashion. Thanks to Breathe, it can be used to produce documentation from sources originally parsed with Doxygen. Sphinx can produce output in a variety of formats, including LaTeX and HTML. For HTML there is a vast choice of themes that might improve the accessibility and readability of the produced documentation.

MkDocs [C+] is a static site generator that uses Markdown input. A popular framework built on top of MkDocs and its ecosystem is Material for MkDocs [Don]. The MkDocs ecosystem also includes plugins to generate API references from the code and the docstrings [Maz], supporting different languages with various degrees of maturity. MkDocs can also render Jupyter notebooks and include them in the generated documentation (thanks to the mkdocs-jupyter plugin[10]). There are efforts to integrate it with Doxygen (e.g., MkDoxy [And]).

Documenter.jl [Jul] is a Julia package geared towards the documentation of Julia code designed with the “batteries included” philosophy in mind, in the sense that it does not only take care of rendering the documentation (also generating API references by extracting information from the source code) but also provides functions for, e.g., automatic deployment to GitHub and GitLab pages.

In order to render the documentation for a R package into a static website, the *pkgdown* [WHS+24] utility can be used, designed as well with extreme convenience for the user in mind, as it even includes functions to automatically deploy static documentation to GitHub pages. At a lower level, *Rdconv* is used to produce documentation in various output formats starting from

[6] https://dealii.org/developer/doxygen/deal.II/Tutorial.html
[7] https://executablebooks.org/en/latest
[8] https://nbsphinx.readthedocs.io/en/latest/
[9] https://myst-nb.readthedocs.io/en/latest/
[10] https://mkdocs-jupyter.danielfrg.com/

`.Rd` files[11], which can also be generated automatically from comments in plain R source code by roxygen2 [WDCE24]. Long-form guides for a R packages are usually created in the form of a *vignette*. The functions for rendering and manipulating vignettes are part of a basic R installation.

The standard Rust distribution includes *Rustdoc*[12], which can be used both to generate documentation from standard MarkDown files (CommonMark) and API references extracting documentation from the source code. For higher-level documentation *mdbook*[13] can be used, also for non-rust projects: an example of this in the context of research code is the documentation of Hexatomic [DKLB23].

Code documentation can also be written using a literate programming-like [Knu92] paradigm, which offers another solution (alternative to in-code content) to the problem of documentation maintainability. With Rmarkdown and Jupyter Notebooks, such an approach can be implemented, and it makes Tutorial Driven Development [Woo] practical. This approach does not focus on extracting documentation from the source code, and can be seen as complementary to in-code documentation.

Rmarkdown [AX+] is a tool primarily used in the R community (although also other languages are supported) also in package vignettes. While the execution of code in a Rmarkdown document is delegated to the knitr package, the rendering of the document itself is performed by Pandoc [Maca]. For this reason, the input format is Pandoc-flavored Markdown. Rmarkdown files can also be used in the documentation of R packages, especially in vignettes.

Jupyter Notebooks [Pro] can also be used to document code in a literate programming fashion. A notable difference with the tools mentioned so far is that although Jupyter Notebooks can be still be modified with a simple text editor (being JSON documents), it is rarely practical to do so, and editing (and execution) is usually done from within a browser connected to a running Jupyter (or Jupyter Lab) server. Jupyter can be used for any programming language for which a proper Jupyter kernel exists (most notably, Python, Bash, C/C++ via xeus-cling[14] [MCRQ],[VCN+12], as well as R and Julia). As mentioned before, Jupyter notebooks can be included as part of the documentation generated by Sphinx or MkDocs.

The Julia community has a number of options when it comes to literate programming. In addition to Documenter.jl, *Pluto.jl* [P+] offers a way to develop notebooks which are also reactive (see *reactive programming*[15]). *Literate.jl* [E+] is also an implementation of literate programming for the Julia language, which can generate Markdown or Jupyter notebook files out of plain (commented) Julia code.

Hosting documentation can also be done in a so-called *"wiki"* connected to the code repository, which can be activated on GitHub[16] and GitLab[17], among others. The content in the wiki can typically be edited from a web interface, which makes them easy to use. In the GitHub and GitLab incarnations wikis consist of, under the hood, git repositories, and thanks to this many of the tools and formats that are used to write documentation bundled with the code can also

[11] https://rstudio.github.io/r-manuals/r-exts/Writing-R-documentation-files.html
[12] https://doc.rust-lang.org/rustdoc/index.html
[13] https://rust-lang.github.io/mdBook/
[14] https://github.com/jupyter-xeus/xeus-cling/tree/main
[15] https://en.wikipedia.org/wiki/Reactive_programming
[16] https://docs.github.com/en/communities/documenting-your-project-with-wikis/about-wikis
[17] https://docs.gitlab.com/ee/user/project/wiki/

Tool	Documentation Language(s)	Parsed languages (API Ref generation)	Search capabilities
Doxygen	• MarkDown (own flavour) • ReST (Breathe+Sphinx)	• C/C++ • Fortran (limited)	• Limited keyword based client-side • Full-text server side (custom library[18])
Sphinx	• ReST (Native) • MarkDown (MyST) • Jupyter Notebooks	• Python • C/C++ (with Doxygen+Breathe)	• Limited keyword based client-side (custom library) • Full text server side provided on readthedocs.io
MkDocs	• Python-MarkDown • Jupyter Notebooks	Python (via mkdocstrings [19])	search plugin[20] for client-side (`lunr.js`)
Documenter.jl	Julia-MarkDown	Julia	client-side (`minisearch.js`)
Pkgdown	• RMarkdown (Pkgdown) • LaTeX-like (Roxygen2 and Rdconv)	R (via Roxygen2 and Rdconv)	• web based client-side search (Pkgdown - using `fuse.js`) • (RStudio)
Rustdoc	MarkDown (CommonMark)	Rust	custom library
mdbook	MarkDown (CommonMark)		client-side (`elasticlunr.js`)

Table 1: Comparison of the HTML documentation generation tools mentioned in the text.

be used with wikis. But - crucially - wiki repositories are completely separate from the ones they are supposed to document, whereas having documentation and code in the same repository would make it much easier to keep both in sync using version control and automatic testing when possible. Moreover, not all wiki systems support the same formats, which means that choosing a particular one might result in a vendor lock-in.

[18] https://www.doxygen.nl/manual/searching.html
[19] Parser for different languages exist with different levels of maturity (https://mkdocstrings.github.io/)
[20] https://www.mkdocs.org/user-guide/configuration/#search

2.4 Making documentation accessible and convenient: hosting, searching and measuring

While it is possible to distribute documentation with the code and generate it as part of the code building process, it can be useful to make it accessible from the web. Most of the documentation systems mentioned above can produce static HTML websites that can be conveniently hosted on different platforms with little server-side computation and no server-side programming required.

One of the most know services to host documentation is ReadTheDocs[21]. This platform can interact with various Source Code Management Service (SCMS) via web hooks, building and deploying the documentation after each push to the SCMS. The configuration for these steps is contained in a YAML file called `.readthedocs.yaml`.

It is also possible to host the documentation as a static website directly on GitHub or on the GitLab instance where the code resides, using GitHub actions (optionally) or GitLab CI/CD respectively to build the documentation static web pages, and *GitHub pages* or *GitLab pages* respectively for deployment.

On github.com, the pages feature needs to be manually activated. A GitHub Action workflow needs to be defined for the generation of the documentation, unless Jekyll[22] (a general-purpose static site generator based on the Ruby stack) is used for this task.

On a GitLab server, the page feature needs to be activated by the administrator and a CI job needs to be configured (a guided setup is available).

A very important part of making documentation convenient to use is having a search option. The fact that typically documentation is deployed as a static website means that back end programming is then avoided. Either searching capabilities are somewhat limited because of this, or the search logic needs to be implemented on the client side. (however, readthedocs.io provides server side search[23] to static documentation). The benefit of client-side search is that it can also be performed while offline, if the documentation is generated locally. Notice though that good client side search requires the whole text content of the documentation to be made available on the client side, no matter what page is viewed, and for slow internet connections (or large documentation corpora), this can be a problem.

There are differences in the client-side search functionality provided by different tools. Some only provide a keyword based search, while others are bundling a Javascript search library. The capabilities might change in different releases of the tool, but are generally better then a pure keyword based search. For a full overview, including server-side capabilities (if provided), see table 1.

An alternative to providing search capabilities is to rely on the indexing by external search engines (as for example the Programmable Search Engine[24]).

Relevant to this topic are also recent advancements in Large Language Models in AI fields, primarily Retrieval Augmented Generation (RAG) [LPP+21], which can be used to create chat bots that can answer questions starting from a documentation corpus and a pre-trained Large Language Model (LLM). Fine tuning of the LLM is also possible, but it is much more computa-

[21] https://readthedocs.com/
[22] https://jekyllrb.com/
[23] https://docs.readthedocs.io/en/stable/server-side-search/index.html
[24] https://programmablesearchengine.google.com/about/

tionally expensive and retrieval-based memories present important advantages over fine-tuning (see [LPP+21] and references therein for a more complete view of the topic).

Hosting the documentation at a publicly accessible internet address can also permit the measurement of the user interest in each page and of the user interest in search terms. This information can help in the development and the maintenance not only of the documentation, but also of the software. Readthedocs.io has a search and traffic analytics feature[25]. As an alternative to using ReadTheDocs.io as a host, it is possible to activate external analytics services (e.g., Google Analytics et similia). A further alternative, if available, could be analyzing the web server logs to shed some light on the usage patterns.

2.5 Automated testing of documentation

Maintaining the documentation in sync with the software it is supposed to document is usually a challenge. Similar problems, with additional challenges, apply to the documentation of a facility (e.g, a HPC system). The parts that are usually easier to test in the documentation are the executable ones, i.e., the commands and the code blocks given as examples. A meaningful testing strategy needs to check that these run without errors, and when their expected output is reported in the documentation, that it reasonably matches the "real" one produced in the tests. This brings some challenges, typically (not all of these might always apply):

- define proper equivalence between expected and real output (this might require, e.g. use regex patterns or ellipsis);
- perform *setup and tear down* of the test cases (that is, perform those steps necessary to prepare the execution of the test case and to clean the state after its execution, when the description of these steps does not fit in the flow of the documentation);
- preserving state between code blocks;
- define the correct environment for the tests to run (e.g. machine, path, user, environment variables);
- extracting the code snippets from the documentation and test them (or associate the test cases with the right code snippets in the documentation)

Additionally, we would still like to detect issues with non-executable parts of the documentation (e.g., checking that all links are functional, checking spelling).

Over time, different communities have come up with different solutions for this problem.

The Python standard library includes *doctest* [Pyt], which can be both used to detect and test code blocks running the code and comparing results written in a documentation file as interactive session examples. It can equally detect and test such examples in the docstrings inside source code. It provides also limited ways to deal with the problem of proper equivalence of output.

The already mentioned `Documenter.jl` package [Jul] offers the same capabilities for Julia. Every example code block annotated with `jldoctest` will be tested. Labeling example code

[25] https://docs.readthedocs.io/en/stable/guides/content/index.html

blocks allows to keep state between them. A customizable filtering mechanism is available to make sure that only the relevant part of the output is checked.

In the case of compiled languages, Rustdoc and mdBook offers similar features for the Rust programming language, with a different set of limitations (for example, it is not possible to keep state between different code blocks). Documentation generated by mdBook allows the the user to execute code snippets embedded in it (which is performed on a remote server) with the press of a button, which can be convenient to manually test the correctness.

An alternative which requires more tooling and configuration is the combination of Jupyter notebooks and nbval [LCA$^+$], resorting again to literate programming. With this approach, the documentation can be written in Jupyter notebooks. Hidden cells can be used to set up and tear down test cases (cell hiding relies on nbconvert[26] and it is done via tags[27]). nbval [LCA$^+$] can then be used to execute the notebook and check that the output matches the expected one, and REGEX output sanitizing[28] can be used to address the problem of equivalence of output. The notebooks can be integrated into MkDocs-based[29] or Sphinx-based[30] documentation corpora. This approach can work for any language for which a Jupyter kernel implementation is available. Python examples can be found in the documentation of Parafields [KKK$^+$23] and py4dgeo [pDCT23].

A less customizable form of documentation test is available with the IPython Sphinx Directive[31], which allows to include IPython interactive sessions in Sphinx documentation. The extension can execute the code and compare the real output with the one saved in the document, and/or save the real output into the document.

Tooling is also available for automatic spellcheck (e.g., *Aspell* [A$^+$] and *Hunspell* [NHM$^+$]), which have been wrapped in Python packages and in GitHub actions.

3 Testing

Testing is an integral part of software development, as it ensures, that results are and stay correct over time and further development.

Automated testing is a crucial tool to ensure stability, which has been proven to be essential to guarantee a sustainable development pace in the software industry [FHK18].

In order to understand the field of testing better, tests can be categorized in two independent dimensions. One dimension is the scope of what is under test, the second dimension is how the test works, so the type of test used. Many combinations of these dimensions are possible, so that different scopes can be tested with different types of tests. A recommendation on where to put most emphasis on can be found at the end of this chapter.

Apart from the two dimensions of tests, it is important to execute the tests frequently. Therefore, it is advised to include them in Continuous Integration.

[26] https://nbconvert.readthedocs.io/en/latest/
[27] Tag management within the Jupyter web interface can be cumbersome in some version of Jupyter.
[28] https://nbval.readthedocs.io/en/latest/#REGEX-Output-sanitizing
[29] https://github.com/danielfrg/mkdocs-jupyter
[30] https://docs.readthedocs.io/en/stable/guides/jupyter.html
[31] https://ipython.readthedocs.io/en/stable/sphinxext.html

An important concept when dealing with tests is called *test coverage*. It reports the fraction of code that is executed when running the tests and where blind spots of the test suite are. There are multiple metrics for this concept. The most commonly used metric measures how many (and which) lines of code were run. When a line of code is executed at least once[32] during the run of the test suite, it is considered covered[33]. The ratio of the number of covered lines over the total number of lines in the code (excluding comments and empty lines) is the *line coverage*, but usually called test coverage.

Although test coverage is very convenient to measure, it is not perfect as a metric for testing completeness. In particular, while low test coverage definitely means that the code is not adequately tested, even 100% test coverage is not a guarantee of testing completeness, as only parts of a line need to be executed for the line to be counted as covered. An example is the use of a logical or, where the first part is true. This means, that the second part does not need to be evaluated anymore and is often skipped. Therefore, the line is covered, but not everything of that line was tested.

Nevertheless, this metric is helpful, as it helps to get an overview of how much code is tested in an easy, fast, and straight forward form. Although there are multiple other metrics available, the line coverage is still the one used in most cases.

3.1 Motivation for automated testing

The main motivation for writing tests is to avoid bugs (or unintended behavior of the code in general). The main reason for avoiding bugs is to be able to rely on the code returning correct results necessary for correct scientific conclusions.

Furthermore, since execution typically happens in a well-defined and documented environment and way (avoiding the inevitable fuzziness that is introduced by manual interaction), automated testing is important to ensure getting the same results for the same inputs, which is crucial when the code is developed further and extended or updated. Therefore, the tests should be carried out every time the updated code is pushed to the server. This is crucial for the reproducibility of scientific findings. Also, tests are important to find out if results depend on external parameters that can differ across installations (e.g. hardware or software versions). This also includes the check on if the code still works as expected with the most recent version of the dependencies used, as these are often the versions that are installed by default when someone else installs the code. Therefore, these tests are also crucial to ensure that other scientists are able to reproduce the claimed findings and can validate them.

The last large reason for automatic testing is to check if current changes in the code changed the behavior at another place in a way that was not intended. So checking examples from many (ideally all) places of the code is important to do repeatedly during development. It is important to detect bugs early in the development process, as it often is much more difficult to revert at a later stage in development [Pla02, Bec99].

[32] Some tools can measure coverage at a finer level, e.g. statements instead of lines, which is required for proper *branch coverage* assessment.

[33] These mechanisms can also be used to detect dead code that can be safely removed.

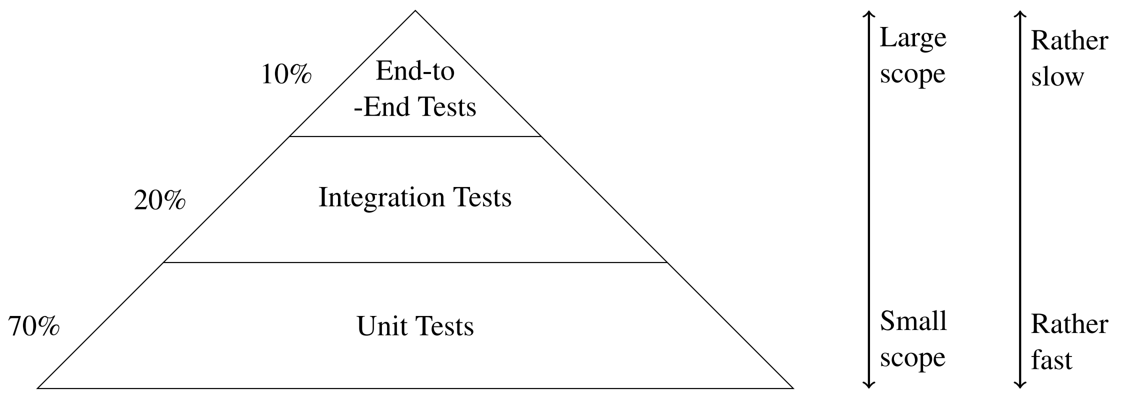

Figure 1: Testing pyramid differentiating by scopes of tests. Based on [Wac].

3.2 Scopes of testing

Tests can be categorized by the scope. The scope can range from checking the correctness of a single function to the whole codebase of a project. The following sections describe different scopes of tests and give recommendations from [Wac] on where to put emphasis on. This is also visualized in Figure 1.

It shows that the emphasis should be put on Unit tests, testing a single function. Integration tests check if individual units work together as expected, but check that for only few units at a time. End-to-End tests are designed to mimic the user and check the full system.

Those scopes of tests will be introduced more in the next paragraphs.

3.2.1 Unit tests

The Turing way [The] defines a unit test as

> A level of the software testing process where individual units of a software are tested. The purpose is to validate that each unit of the software performs as designed

A unit of code usually consists of one or multiple inputs and one output [The]. In most cases, a single function is in this scope.

As already stated by the Turing way, the goal of these tests is to check (and validate) that a single unit performs as expected.

The main reasons for these kinds of tests are the (normally) fast execution and little (ideally no) external dependencies. External dependencies in this context refers to dependencies at run time. This is not about installation of needed packages/modules, but rather about the use of network or web resources. Tests without external dependencies at run time are also called hermetic. As unit tests have little external dependencies, they are more stable and reproducible. And as the amount of covered code is smaller, it is easier to find the error, if the test fails.

Because of these advantages [Wac] recommends to focus on unit tests and states that around 70% of all tests should be unit tests. In order to achieve this, specific care might need to be applied in designing the code so that it is easy to test at the unit level. In particular, techniques that permit inversion of control might be needed to isolate the logic at the top of the function call hierarchy so that it is possible to test them with unit tests.

In *Test-First* development approaches unit tests become a part of the design process. Writing the tests first forces the developer to focus on the testability of the design right from the start, leading to higher coverage with unit test.

In particular, separation of concerns in its various aspects (e.g., the SOLID principles [Mar03]) can help in designing code that can be unit-tested effectively. For an introduction to proper software design practices in the context of Research Software Engineering please refer to the work of SURESOFT [BDF+22].

3.2.2 Integration tests

In Integration tests "two or more units are tested to make sure they interoperate correctly" [BBGT21]. Usually only a few units are combined for these tests. Most of the time the interaction between two units is tested.

As these tests combine multiple units, the tests are usually slower than unit tests and errors are harder to localize if no additional unit tests check the correctness of the individual units in question.

The recommendation by [Wac] is to have approximately 20% of all tests be integration tests.

Notice that often for this kind of tests real external resources are needed, making the tests more prone to flakiness and unexpected sources of error outside the own codebase (e.g., network issues). A way to limit the number of integration tests needed (testing the same functionality with a unit test instead) is to use *Mock-ups*. Mock-ups are dummy implementations that mimic the behavior of the real code used in production. See [MFC01, SABB17] for an empirical study of the use of mock-ups.

3.2.3 End-to-End tests

These tests are also often called *E2E* tests, *system* tests, or *application* tests. They test not a single function or API and not the interaction on a few units, but the system as a whole. These tests are much closer to how users will use the software in reality [Wac], making them an important part of a test suite of the codebase. However, as they test complex systems, it is hard to find errors if these tests fail and no other hints are given (e.g., by failing unit or integration tests). Furthermore, these tests are often more time-consuming than unit or integration tests. Because errors are harder to track, it is important to accompany end-to-end tests with any unit and some integration tests to point to a direction on where to look for the source of the error.

As they are not hermetic (nearly by definition) more sources of error outside the own code influence the result of the test more issues can be found but also not all the found issues can be reproduced or even solved.

Because these tests take most time and the errors are harder to track, the recommendation by [Wac] is to only have around 10% of all tests be end-to-end tests.

As some pieces of software expose a graphical user interface (GUI), this also needs to be tested. A popular package for testing web-based GUIs is *selenium*[34]. This can be used to check if steps on the website result in the expected reaction. However, it is still recommended checking most of the code with unit-tests that directly use the functions of the code instead of testing via the GUI. Then, the end-to-end tests via the GUI are a good extension.

3.3 Types of tests

Another way to differentiate tests is not by scope (as in the previous section), but by the type of test. From the different types, a subset of four of the most important ones shall be discussed here.

In the following paragraphs, single test cases will be discussed first, followed by property based testing and fuzzy tests. Finally, mutation tests will also be touched upon.

The order of the following corresponds to increasing complexity, and it is recommended to start with the earlier approaches before using the later ones.

3.3.1 Single test cases

These tests run the code for a specific value of input and compare the calculated result with a predefined output.

This approach can be followed for simple inputs (as it would be done in a unit test) as well as for testing cases with complex inputs (or complex input data sets). As the calculated result shall be identical to a predefined one, it is also comparatively easy to use this approach with result files. In the case of using result-files, special care should be taken that no frequently changing data (as e.g. time stamps) is included in the result files that are compared, as this leads to false positives in error detection.

This test type has many names that are used in different context. A notable example are *Golden master tests* which are the recommended technique to increase test coverage for a legacy code base [Fea04].

A drawback of this approach is that the tested scope is rather limited. This is because the correctness of the code is only known for a single input (i.e. the example used). So other edge or corner cases are not tested and may still be treated incorrectly.

3.3.2 Property based tests

In Property-based testing (or Property testing) *attributes* (or *properties*) of outputs and inputs are tested instead. Following a specification, a generator, or a "strategy" is used to create automatically test inputs that are fed to the code under test, and it is checked that the required properties hold for *all* outputs (or for all input-output pairs).

The results often cannot be specified as precise numbers but rather in terms of attributes (or *properties*) as well. These attributes are then checked by the tests for all created input data. The simple test cases can be kept as examples, so that the tests do not need to be duplicated. The strength of this approach is that typically the strategies for generating test cases prioritize edge

[34] https://www.selenium.dev/documentation/

and corner cases. Compared to single test cases, the advantage here is that many more inputs are tested and this often leads to raising (and therefore finding) additional errors in the code.

A downside of this approach is that most of the time it is more cumbersome to define the properties of created inputs than to create examples yourself. However, this process also often forces thinking about the specific required properties of input and output data on a more abstract level than just to think about the result for a single input. Another downside is that often it is not increasing line coverage, as it extends single test cases rather than replacing them. Therefore, it is more work and can give more confidence in the correctness of the code, but it rarely increases the test coverage.

More information on the idea and the concept of property based testing can be found in the documentation of a python package providing that functionality [Macb] as well as in an interview [Sch].

3.3.3 Fuzzy tests

Fuzzy testing focuses on discovering bugs (or vulnerabilities) that make the software crash. The approach of fuzzy testing is similar to property based testing (vast amounts of input are automatically generated) but uses a much more diverse input for the code and tests the output in a less strict way. The benefit is, that more user- or interaction errors can be found, but they are often found when and if the code fails. So the testing of results is more difficult or even completely skipped. Instead, it is tested if the code fails gracefully or just crashes for unexpected inputs. This approach is sometimes also called a smoke test, as it just checks if the code bursts into smoke when run.

A nice comparison between property based testing and fuzzy testing can be found in the blog post by the developer of a tool for property based testing [Macb].

3.3.4 Mutation tests

Mutation testing is an additional approach that can be used to assess the effectiveness of a test suite (alternative, or complementary, to measuring test coverage). A package for mutation testing [Kep] phrases it as follows: “Essentially, mutation testing is a test of the alarm system created by the unit tests.”

In the so-called mutation tests, the code is slightly modified, and the test suite is run against the *mutant* (the mutated code). It is expected that now at least one test fails, as otherwise it means that the tests are not sensitive enough to detect this change. If this change is undetected it may occur in the future, that this change is introduced by accident and the tests would not detect this wrong change.

Therefore, this approach tests how good the actual test coverage of the codebase is. It gives an even more specific picture than a report with line coverage, as an executed line may be modified without a test recognizing it. However, it only makes sense to start with mutation tests if the test coverage is high. This is because a change in an uncovered line will not be detected for sure, as that line was not even executed for the tests.

More information on the general idea of mutation tests can be found in early papers of the development of mutation testing from the 1980s [BLDS78, DLS78].

Table 2: An overview of commonly used frameworks or packages for multiple tasks related to testing. Where the cells of the table are empty, the authors are not aware of any popular or frequently used package.

	Single case	Mocking	Property based testing	Mutation testing
C++	Catch2[35] & GoogleTest[36]	GoogleTest	rapidcheck[37]	
Java	JUnit[38]	Mockito[39]	jqwik[40] & junit-quickcheck[41]	Pitest[42]
Julia	Test.jl[43] (builtin)	Mocking[44]		
Python	Pytest[45] & Unittest[46]	unittest.mock[47]	Hypothesis[48]	Mutatest[49]
R	testthat [50]	testthat [51]		
Rust	Cargo tests (builtin)[52]	Mockall[53]	Quickcheck[54] & proptest[55]	Cargo-mutants[56]

3.4 Popular frameworks for testing

There are commonly used frameworks for testing for nearly all programming languages, that are frequently used in Research Software Engineering. Some frameworks shall be highlighted here together with some recommendations for what they can be used well.

Table 2 shall give some hints on frameworks to use for different aspects of testing depending on the programming language used in the project at hand. For a few combinations of types and programming languages, no popular frameworks could be found. Those combinations are left empty. Also, for some combinations, there are two popular frameworks. In those cases, it is recommended to have a look at both to determine which one suits the own needs better. Often, there are comparisons of the two frameworks in the docs of each of them. Because of the multitude of frameworks, the individual entries of the table are only named and links are provided for further reading and usage. However, the frameworks are not discussed or compared here, as this is beyond the scope of this publication. Additionally, a comparison is of little benefit when comparing frameworks for different tasks or in different ecosystems.

The differentiation is done by types of tests. Often, the different scopes of tests (Unit tests, integration tests, End-to-End tests) can be done with the same packages if they are conducted as the same type of tests.

An exception to this is the column of mocking, as this is no separate type of test, but rather an additional tool mostly used when running integration tests.

3.5 Summary

In summary, it proved useful to focus the testing on unit tests, with some integration tests and even fewer system tests. This provides a good combination between speed and accuracy of pinpointing bugs (via the unit tests) and find errors that occur from the interaction between multiple parts of the code (via integration and system tests).

A simple way to start the test suite is to use end-to-end tests that often already exist. Although sometimes these tests are performed by manually doing multiple actions after each other. These steps can be put into scripts to increase the level of automation and reproducibility. However, as these end-to-end tests normally do not show where or why code failed, adding unit tests is important.

The recommended approach is to start testing with single test cases that test the code for a few specific examples. This can then be extended to also include property based testing to find unexpected behavior for edge and corner cases. Fuzzy testing helps to find severe errors (making the code crash) when confronted with unexpected input. It normally does not, however, check for correctness. When the test suite is in place, the test suite itself can be tested looking at the coverage metric and with mutation tests. These will highlight where changes in the code stay undetected, i.e., where more tests should be added.

4 Workflow management

When a code base becomes large, so should its test suite. Consequently, the full test suite takes more time and resources than desirable for running on every change.

This resource use can be mitigated by managing multiple pipelines and workflow layers that run at different points in time. In the following, we will take a more detailed look at the typical

[35] https://github.com/catchorg/Catch2
[36] https://google.github.io/googletest/
[37] https://github.com/emil-e/rapidcheck
[38] https://junit.org/junit5/
[39] https://site.mockito.org/
[40] https://jqwik.net/
[41] https://github.com/pholser/junit-quickcheck
[42] https://pitest.org/
[43] https://docs.julialang.org/en/v1/stdlib/Test/
[44] https://juliahub.com/ui/Packages/General/Mocking
[45] https://docs.pytest.org/en/stable/
[46] https://docs.python.org/3/library/unittest.html
[47] https://docs.python.org/3/library/unittest.mock.html
[48] https://hypothesis.readthedocs.io/en/latest/
[49] https://mutatest.readthedocs.io
[50] https://testthat.r-lib.org/
[51] https://testthat.r-lib.org/reference/local_mocked_bindings.html
[52] https://doc.rust-lang.org/book/ch11-01-writing-tests.html
[53] https://docs.rs/mockall/
[54] https://github.com/BurntSushi/quickcheck
[55] https://github.com/proptest-rs/proptest
[56] https://mutants.rs/

problems with testing large code bases, look at the analytical side of reducing complexity and finally at the technical side of splitting up the test suite.

4.1 The problems in more detail

Most software does not run in isolation, but depends on the operating system, other software libraries and often a compiler or interpreter.

It is often advised to capture the computational environment in some way. To this aim, there are different tools available, ranging from language-specific package managers to containerization solutions (e.g., Docker/OCI and Apptainer) [The].

This is an important approach to ensure the reproducibility of specific scientific findings and a convenient way to distribute software even in some HPC contexts[57], but a software package that can be built and run correctly in *all* relevant environments (thus placing fewer requirements on its users) can benefit from a larger adoption, can gain performance from better hardware and libraries, can interoperate with more software, especially with more recent (and likely, better) software.

Thus, even if we could, we should not “ship our workstation” to the users of our software. Consequently, we should also test for the relevant combinations of all the aforementioned options. Typically, this means at least one test run for each operating system.

Let us look at the following example and how many combinations we get. We want to be able to build on Linux, MacOS and Windows (3 combinations), since system libraries may behave differently [Gal19]. On each we want to test with different versions of our compiler or interpreter. Let us take the three most recent ones, which means 9 combinations to test and build one version of the software with one version of each used library. Additionally, we might want to test against multiple versions of important libraries, with different build systems (e.g. Make, Ninja), with different compiler families (e.g. GCC, Intel, Clang) or interpreters (Python, PyPy). We might also have different combinations of libraries to use or build with. A common example would be different types of parallelism, e.g. sequential or parallel with shared memory (e.g. OpenMP) or distributed memory (e.g. MPI).

The ideal case that gets taught in most introductions is to run the full test suite on each and every pull request (or merge request on some platforms). This avoids relying on developers to run the test suite before integrating their changes and catches many bugs before code gets integrated into the main branch. It also helps reviewers to decide whether or not to integrate the change. The possible downside of this approach is that, typically, every small change during review automatically triggers the test suite to run completely. It might be possible to prevent this behavior, but that might lead to untested code being merged, so it should be done with caution.

From the point of view of computing resource usage this means that we hit the limit of either our payment plan on a public hosting service (in terms of concurrent jobs or minutes), or our own hardware for self hosted services, e.g. GitLab, Jenkins CI. Additionally, we might not want to allow every pull request (PR) to automatically run on our own hardware to prevent misuse.

[57] https://catalog.ngc.nvidia.com/containers

4.2 Reducing complexity

By surveying the needs and customs of the user community we can address the problem of combinatorial explosion. For example, a large part of the community might use a specific Linux distribution and the software versions of libraries provided by its package manager making it a natural candidate for regular tests.

Of course this requires interacting with the user community to know their needs sufficiently well. When it comes to research software we can also help our chances in discovering the users and use cases by making the software citable, for example with a DOI by publishing on Zenodo and/or writing a software publication. Additionally, a CFF file[58] can help users with knowing how to cite the software.

When it comes to the complexity of the test suite itself, we can use common tools around testing. Profilers allow us to see which functions are called regularly and how long their execution takes, giving us an idea about the importance of functions as well as the time it would take to test them. Code coverage tools can not only tell us the total coverage of all tests, but also the coverage of a subset. While this metric does not guarantee that all parts of each line are tested, as discussed in 3.3.4, it can still inform us when constructing a subset that is covering enough of the code base at minimal run time.

Proper design (or refactoring) of the software can also help in reducing the number of tests needed, if the software can be factored reliably into components with sufficiently low coupling. Abstraction techniques could be limited by performance requirement in some applications.

4.3 Layering test suites

Now that we have some idea about what to test regularly on which configuration we can look at how we actually divide the work into multiple layers.

The most important layer for core functionality is a test run automatically on every single pull request. Apart from functional tests this is usually a good place for examining code formatting, linting and documentation as well.

A second layer can be again at the PR stage, but only after a first review. This prevents test from being run unseen on our own hardware and there are multiple ways to achieve this. A job could be set to manual, but that would require manual restart after every iteration of the PR. Alternatively, we can use conditional expressions in the job configuration to steer this, e.g. with labels and a corresponding statement of the form
`if: ${{ github.event.label.name == 'ready to test'}}.`
After human inspection this label can be set on the PR and trigger multiple jobs to run.

The final layer can be a dedicated *regression testing* server. The consequence of having multiple layers (that is, not running all the tests every time) is that errors can be incorporated into the code base without immediate detection and only noticed when the tests in further layers are run. Regression testing is about tracking the status and history of the all tests in the test suite. Therefore it helps with knowing if a change introduced a test failure, did not change the results or fixed a previously failing test. Dedicated dashboard tools like CDash can be hosted to display regression results in a useful manner. Test servers will run tests on a fixed schedule, independent

[58] https://book.the-turing-way.org/communication/citable/citable-cff.html

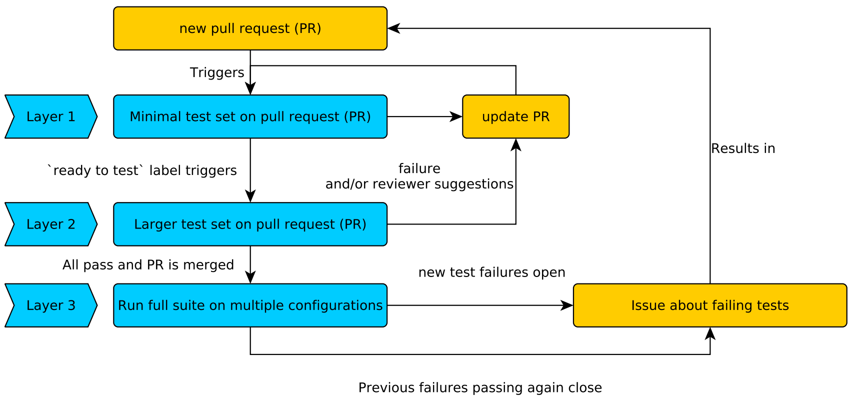
new pull request (PR)
Triggers
Layer 1
Minimal test set on pull request (PR)
update PR
Results in
`ready to test` label triggers
failure
and/or reviewer suggestions
Layer 2
Larger test set on pull request (PR)
All pass and PR is merged
new test failures open
Layer 3
Run full suite on multiple configurations
Issue about failing tests
Previous failures passing again close

Figure 2: Example workflow for a layered approach.

of individual pull requests, and communicate the results to the dashboard instance. Often the tests are set to run only on the newest available commit. When a test fails, the test run history allows the dashboard server to see if this is a new problem. A test server can then be notified to search for the actual commit where the error was introduced, by doing a targeted run of only the failing tests. Additionally, this allows the server to see if a previous error was fixed in the newest commit. All this information can then be bundled and used to notify developers of failing tests or resolved issues on configurations that have not been tested in the earlier stages. To sum up this layered approach, Figure 2 shows an example workflow starting from a new pull request.

4.4 A concrete example: deal.II

An open source project that uses all the previously described layers is deal.II [ABD+21]. When you open a pull request, multiple GitHub actions start:

- Static checks: `clang-tidy`, `clang-format`, `Doxygen` documentation.
- Linux: GCC, GCC+MPI, Intel compiler, CUDA, CUDA+clang
- MacOS: GCC, GCC+MPI
- Windows: MSVC

After review one of the principal developers sets a ‘ready to test’ label and a server running Jenkins starts full test suite runs for Linux with GCC, GCC+MPI and CUDA as well as a test on MacOS.

Finally, CTest servers run the full test suite on each of the 23 different configurations and put their results on the CDash server[59]. If a test fails, a pinned issue is automatically opened on GitHub with the following information:

[59] https://cdash.dealii.org/index.php?project=deal.II

- Current and previous revision (run+commit hash) with link to CDash results.
- Fails and warnings with respective configuration
- Pull requests that might be responsible, pinging the accounts responsible for creating and merging the request.

When a new revision fixes all errors noted in the issue it will be closed automatically and similarly reopened when it was closed and an issue still persists.

5 Summary and outlook

This paper gave an insight on approaches to deal with the increasing importance and increasing complexity of research software, by applying different techniques from Research Software Engineering.

As good documentation is vital to understand the intention, the structure and the implementation of the code, it is important for it to be well readable and up-to-date. The approach of in-code documentation helps to keep it in sync with the code. It was shown how to include documentation in the codebase and multiple frameworks have been introduced. All these frameworks have their unique combination of features, so the one can be chosen that matches the own requirements best. These requirements can be the programming language that is mainly used, but also features or plugins that provide additional benefit and are available for some frameworks.

In order to increase reproducibility and to find errors early in development, testing is important when writing or changing research software. These tests should be run frequently (ideally automated) to find changes early, as this makes fixing mistakes easier, than if they go unnoticed for long time. The focus should be put on fast and more specific tests (unit tests) compared to tests that mimic the users better, but take more time and give less specific information on where a found error originated from (as is the case for system tests). Tests (in particular, end-to-end and integration tests) can also be valid examples of use of the software, and represent an important form of documentation which is more rigorous and easier to maintain.

For larger codebases the complexity of the test suite itself or the number of environments to test the software in can become quite large. This complexity has to be managed and approached both analytically and practically. The analytical side focuses on figuring out the integral parts of the software that have to be tested on every change to the codebase as well as choosing the most important environments to do these integral tests in. Since this may prevent errors from being caught right away, the practical side focuses on frequently running the full test suite on more environments.

Proper software design practices can also ease the testability of the code, reducing the reliance on end-to-end tests, the complexity of the test suite and cost of running it (although performance requirements can pose challenging constraints on the available techniques).

With the introduced techniques the authors aim to assist scientists to increase the usage of good practices when developing scientific code. The tools mentioned are (by far) not a complete list, but shall show the diversity of tools available and shall give hints on what category of tools to look for if the mentioned tools are not suited or not sufficient for the own codebase.

Acknowledgements: Jan Philipp Thiele wants to thank the principal developers of deal.II for not only providing excellent documentation on the software itself, but also on almost every other aspect of its development.

Jakob Fritz and Michele Mesiti thank the HiRSE_PS project, ("Helmholtz Platform for Research Software Engineering - Preparatory Study"), funded by the Innovation Fund of the Helmholtz Association.

The authors would like to acknowledge the attendees of the session "Automatic Testing of Documentation" that happened as part of the undeRSE23 event for contributing, making them aware of relevant additional tools and techniques, as well as the attendees of the "Intermediate CI" session at deRSE24, for their questions and remarks that helped shaping the choice of topics in the current paper.

Bibliography

[A$^+$] K. Atkinson et al. aspell.
http://aspell.net/ (Accessed: 2024-05-17)

[ABD$^+$21] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells. The deal.II finite element library: Design, features, and insights. *Computers & Mathematics with Applications* 81:407–422, 2021. Development and Application of Open-source Software for Problems with Numerical PDEs.
doi:https://doi.org/10.1016/j.camwa.2020.02.022
https://www.sciencedirect.com/science/article/pii/S0898122120300894

[And] J. Andrysek. MkDoxy.
https://mkdoxy.kubaandrysek.cz/ (Accessed: 2024-05-15)

[AX$^+$] J. Allaire, Y. Xie et al. rmarkdown: Dynamic Documents for R.
https://cloud.r-project.org/web/packages/rmarkdown/index.html (Accessed: 2024-05-15)

[B$^+$] G. Brandl et al.
https://www.sphinx-doc.org/en/master/ (Accessed: 2024-05-16)

[BBGT21] R. Bierig, S. Brown, E. Galván, J. Timoney. *Essentials of Software Testing*. Cambridge University Press, 1 edition, 2021.
doi:10.1017/9781108974073
https://www.cambridge.org/highereducation/product/9781108974073/book (Accessed: 2024-03-26)

[BDF$^+$22] C. Blech, N. Dreyer, M. Friebel, C. Jacob, M. Shamil Jassim, L. Jehl, R. Kapitza, M. Krafczyk, T. Kürner, S. C. Langer, J. Linxweiler, M. Mahhouk, S. Marcus, I. Messadi, S. Peters, J.-M. Pilawa, H. K. Sreekumar, R. Strötgen, K. Stump, A. Vogel, M. Wolter. SURESOFT: Towards Sustainable Research Software. 2022.

doi:10.24355/dbbs.084-202210121528-0
https://leopard.tu-braunschweig.de/receive/dbbs_mods_00071451

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Publishing Company, 1999.

[BLDS78] T. A. Budd, R. J. Lipton, R. DeMillo, F. Sayward. The Design of a Prototype Mutation System for Program Testing. In *Managing Requirements Knowledge, International Workshop On*. Pp. 623–623. IEEE Computer Society, 1978.

[C$^+$] T. Christie et al. Mkdocs - Project documentation with Markdown.
https://www.mkdocs.org/ (Accessed: 2024-05-15)

[CH24] N. Chue Hong. Is research software engineering coming of age? June 2024.
doi:10.5281/zenodo.11580329

[Cod] CodeRefinery. How to document your research software.
https://coderefinery.github.io/documentation/ (Accessed: 2024-05-14)

[pDCT23] py4dgeo Development Core Team. py4dgeo: library for change analysis in 4D point clouds. 12 2023.
https://github.com/3dgeo-heidelberg/py4dgeo (Accessed: 2024-08-26)

[DKLB23] S. Druskat, T. Krause, C. Lachenmaier, B. Bunzeck. Hexatomic. 10 2023.
doi:10.5281/zenodo.6900689
https://hexatomic.github.io

[DLS78] R. DeMillo, R. Lipton, F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11(4):34–41, 1978.
doi:10.1109/C-M.1978.218136
http://ieeexplore.ieee.org/document/1646911/ (Accessed: 2024-05-15)

[Don] M. Donath. Material for MkDocs.
https://squidfunk.github.io/mkdocs-material/ (Accessed: 2024-05-15)

[E$^+$] F. Ekre et al. Literate.jl.
https://fredrikekre.github.io/Literate.jl/v2/ (Accessed: 2024-05-15)

[Fea04] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, USA, 2004.

[FHK18] N. Forsgren, J. Humble, G. Kim. *Accelerate*. IT Revolution, Portland, OR, 2018.

[Gal19] S. Gallagher. Researchers find bug in Python script may have affected hundreds of studies. *ars technica*, 2019.
https://arstechnica.com/information-technology/2019/10/chemists-discover-cross-platform-python-scripts-not-so-cross-platform/ (Accessed: 2024-05-22)

[GGM+] D. Goodger, E. Gruber, G. Milde, T. J. Ibbs, L. Wiemann. Docutils: Documentation Utilities - Written in Python, for General- and Special-Purpose use. https://docutils.sourceforge.io/ (Accessed: 2024-05-15)

[Goo] D. Goodger. reStructuredText - Markup Syntax and Parser Component of Docutils. https://docutils.sourceforge.io/rst.html (Accessed: 2024-05-14)

[GS] J. Gruber, A. Schwartz. Markdown. https://daringfireball.net/projects/markdown/ (Accessed: 2024-05-15)

[vH] D. van Heesch. Doxygen. https://www.doxygen.nl/ (Accessed: 2024-05-15)

[J+] M. Jones et al. Breathe. hhttps://www.breathe-doc.org/ (Accessed: 2024-05-15)

[JOS] Submitting a paper to JOSS. https://github.com/openjournals/joss/blob/d4d3046b0662af11c9e21e88643711154970f74f/docs/submitting.md

[Jul] JuliaDocs. Documenter.jl. https://documenter.juliadocs.org/stable/ (Accessed: 2024-05-16)

[Kep] E. Kepner. Mutatest 3.1.0 Documentation. https://mutatest.readthedocs.io/en/latest/install.html#mutation-trial-process (Accessed: 2024-05-15)

[Ker] B. W. Kernighan. UNIX for Beginners. https://web.archive.org/web/20130108163017if_/http://miffy.tom-yam.or.jp:80/2238/ref/beg.pdf (Accessed: 2024-05-15)

[KKK+23] D. Kempf, O. Klein, R. Kutri, R. Scheichl, P. Bastian. parafields: A generator for distributed, stationary Gaussian processes. *Journal of Open Source Software* 8(92):5735, 2023. doi:10.21105/joss.05735

[Knu92] D. E. Knuth. *Literate Programming*. Stanford, California: Center for the Study of Language and Information, 1992.

[Lam23] A.-L. Lamprecht. Research Software, Software Research, and more. Sept. 2023. doi:10.5281/zenodo.8355984

[LCA+] O. W. Laslett, D. Cortes-Ortuno, M. Albert, O. Hovorka, H. Fangohr. Py.test plugin for validating Jupyter notebooks. https://nbval.readthedocs.io/en/latest/ (Accessed: 2024-05-16)

[LPP+21] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, D. Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. 2021.

[M+] J. MacFarlane et al. CommonMark - A strongly defined, highly compatible specification of Markdown.
https://commonmark.org/ (Accessed: 2024-05-15)

[Maca] J. MacFarlane. Pandoc.
https://pandoc.org/ (Accessed: 2024-05-15)

[Macb] D. R. Maclever. What Is Property Based Testing?
https://hypothesis.works/articles/what-is-property-based-testing/ (Accessed: 2024-05-15)

[Mar03] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, USA, 2003.

[Mat] Mattt. Semantic Line Breaks.
https://sembr.org/ (Accessed: 2024-05-15)

[Maz] T. Mazzucotelli. mkdocstrings - Automatic documentation from sources, for MkDocs.
https://mkdocstrings.github.io/ (Accessed: 2024-05-15)

[MC09] R. C. Martin, J. O. Coplien. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ [etc.], 2009.

[MCRQ] J. Mabille, S. Corlay, M. Renou, QuantStack. xeus.
https://github.com/jupyter-xeus/xeus (Accessed: 2024-05-15)

[MFC01] T. Mackinnon, S. Freeman, P. Craig. *Endo-testing: unit testing with mock objects*. Pp. 287–301. Addison-Wesley Longman Publishing Co., Inc., USA, 2001.

[Mit] J. Mitchell. Docs as code: How to write documentation with developers.
doi:10.5281/zenodo.7260347
https://www.youtube.com/watch?v=A8IzzZibs3c (Accessed: 2024-05-14)

[NHM+] L. Németh, K. Hendricks, C. McNamara, B. Jacke et al. hunspell.
http://hunspell.github.io/ (Accessed: 2024-05-17)

[Ott] T. Ottinger. Code is a Liability.
http://web.archive.org/web/20070420113817/http://blog.objectmentor.com/articles/2007/04/16/code-is-a-liability (Accessed: 2024-05-17)

[P+] F. van der Plas et al. Pluto - Simple reactive notebooks for Julia.
https://plutojl.org/ (Accessed: 2024-05-17)

[Pla02] S. Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* 1, 2002.

[Pro] Project Jupyter. Jupyter.
https://jupyter.org/about (Accessed: 2024-05-16)

[Pyt] Python Software Foundation. doctest - Test interactive Python examples.
https://docs.python.org/3/library/doctest.html (Accessed: 2024-05-16)

[Rho] B. Rhodes. Semantic Linefeeds.
https://rhodesmill.org/brandon/2012/one-sentence-per-line/ (Accessed: 2024-05-15)

[SABB17] D. Spadini, M. Aniche, M. Bruntink, A. Bacchelli. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Pp. 402–412. IEEE, Buenos Aires, Argentina, May 2017.
doi:10.1109/MSR.2017.61
http://ieeexplore.ieee.org/document/7962389/ (Accessed: 2024-08-06)

[Sch] P. Schmidt. ByteSized RSE: Property Based Testing - Duncan McGregor and Nicholas Del Grosso.
https://www.buzzsprout.com/1326658/14897745 (Accessed: 2024-05-15)

[The] The Turing Way Community. The Turing Way: A Handbook for Reproducible, Ethical and Collaborative Research.
doi:10.5281/ZENODO.7625728

[VCN$^+$12] V. Vassilev, P. Canal, A. Naumann, L. Moneta, P. Russo. Cling – The New Interactive Interpreter for ROOT 6. *Journal of Physics: Conference Series* 396(5):052071, dec 2012.
doi:10.1088/1742-6596/396/5/052071

[Wac] M. Wacker. Just Say No to More End-to-End Tests.
https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (Accessed: 2023-10-11)

[WDCE24] H. Wickham, P. Danenberg, G. Csárdi, M. Eugster. roxygen2: In-Line Documentation for R. 2024. R package version 7.3.2, https://github.com/r-lib/roxygen2.
https://roxygen2.r-lib.org/ (Accessed: 2024-08-20)

[WHS$^+$24] H. Wickham, J. Hesselberth, M. Salmon, O. Roy, S. Brüggemann. pkgdown: Make Static HTML Documentation for a Package. 2024. R package version 2.1.0, https://github.com/r-lib/pkgdown.
https://pkgdown.r-lib.org/ (Accessed: 2024-08-20)

[Woo] C. Woods. Tutorial Driven Development: What is it, how it works, and why it is great!
https://drive.google.com/file/d/1EBguPSwwlDNp9qQC9f_dyUcTchZ3XrML/view (Accessed: 2024-05-14)

[wri] writethedocs.org.
https://www.writethedocs.org (Accessed: 2024-05-14)