Research Software Best Practices

Rafael Mudafort and Garrett Barter

National Renewable Energy Laboratory

This is a temporary cover page. A final cover page will be generated by comms with current style and formatting.

v2.0.4, November 2022

Executive Summary

Wind energy researchers typically share one key characteristic: a passion for increasing wind energy in the global energy mix. The U.S. Department of Energy (DOE) supports this mission in several ways, including allocating funding directly to various aspects of wind energy research through the Office of Energy Efficiency and Renewable Energy (EERE) via the Wind Energy Technologies Office (WETO). Though the traditional output of research is academic publication, software development efforts are increasingly a major focus. Software tools in the research environment allow researchers to describe an idea and quickly increase the scope and scale as they study it further. As a product of research, these tools represent a direct pipeline from researcher to industry practitioners since they are the implementation of ideas described in academic publications. Given this vital role in wind energy research and commercial development, the broad research software portfolio supported by WETO must maintain a minimum level of quality to support the wind energy field in the growing transition to renewable energy. **This report outlines a series of best practices to be adopted by all WETO-supported software projects as well as expectations that the communities interacting with these projects should have of the developers and tools themselves.**

Wind energy research software has a unique standing in the field of scientific software. The stakeholders are varied with a subset being the following:

- DOE EERE leadership
- DOE WETO leadership and program managers
- National laboratory leadership
- Associated project principal investigators
- Research software engineers
- Wind energy researchers in academia (including graduate students, postdocs, and national lab staff)
- Industry researchers and practitioners
- Commercial software developers
- The general public interested in wind energy

These software tools are typically the end user of other generic software libraries, so the funding cycles are often tied to applied research rather than the development of the software itself. Because the developers are also wind energy researchers, these tools are typically designed in a way that closely resembles the application in which they're used. In addition, the expertise and incentives for the developers have a high variability, and often neither is aligned with software engineering or computer science.

Given the unique environment in which wind energy research software is produced and consumed, it is critical for model owners to understand the context of their software. A framework for developing this understanding is to answer the following questions of a given software project:

- What is its purpose?
- What is its role in the field of wind energy?
- What is the profile of the expected users?
- How long will it be relevant?
- What is the expected impact?

These questions allow model owners to identify the appropriate methods for the design, development, and long-term maintenance of their software. In addition, the answers provide context for future planners to understand why particular decisions were made and discern the consequences of changing course.

The information is aggregated from experience within WETO-supported software development groups as well as external organizations and efforts to define the craft of research software engineering. These best practices aim to make the collaborative development process efficient and effective while improving the model understanding across stakeholders. In addition, the general adoption of a common framework for software quality ensures the end users of WETO software can trust these tools and accurately understand the risks to workflow integration.

Table of Contents

List of Figures

1 Summary of Best Practices

Accessibility

- Barriers: Determine the barriers to entry for expected users and address accessibility accordingly. Automate accessibility methods and processes so it is implicit in the software development process.
- Prerequisite knowledge: Identify target user profiles and anticipate their levels of understanding. Accurately understand the complexity of the systems used to access the software, and evaluate whether this matches the expected skills in target users. Note technical solutions can be augmented with documentation to address gaps in prerequisite knowledge.
- Distribution: Provide a streamlined method of installation using common software distribution systems.

Usability

- User interface (UI): UIs should be predictable and adopt existing conventions for the contexts in which they exist.
- Command line interface (CLI): If a CLI exists, it should be meaningful, predictable, and well documented. Refer to contextual guidelines and conventions for flags, syntax, and functionality. At a minimum, provide documentation via the help flag; extended documentation alongside examples and tutorials is helpful.
- Input and output files: Use a common file structure relevant to the type of data produced from a software, and leverage the existing ecosystem of tools to pre- and post-process input and output files.
- Error messages: Identify an error messaging system that enables communicating to users without encumbering the development process. Provide useful errors that include data, provide guidance for moving forward, and help maintainers identify potential bugs.
- Metadata: Providing metadata to users requires minimal effort for developers, and it enables users to more effectively share and compare data and get help. At a minimum, display version numbers, critical settings, and dependency info.

Extendability

- Ease of development: How easily a project can be extended is critical to its viability as a long-term DOE-funded project. Prioritize simplicity in architecture, dependencies, and toolchains. Create a development environment balancing modern needs with stability.
- Code style: Strive to write code that external developers can easily read and comprehend with minimal preexisting context.
- Architecture and design: Adopt an explicit design process where the major ideas are chosen before any code is written.
- Software design process - Create a parti diagram and list performance requirements for each level of fidelity in the software. Establish methods to validate the design and implementation given knowledge of how a software is ultimately used.
- Design patterns: Study existing design patterns, and adopt a few, as needed. Refer to existing materials especially relevant to research software architecture.
- Version control: Craft a version control history that communicates the evolution of changes of the software to future developers, including the author of current changes. Evolve the software in a logical, linear process with digestible, easily reviewable changes.
- Collaborative workflows with GitHub: Treat GitHub as the home page of a software project, and develop the planning and coordination activities as a first-order communication, signaling, and organizational mechanism for the community of users.

- Pull requests (PRs): All components of a PR should be considered documentation for future reference and an aspect of version control. PR reviews should be verbose, thorough, positive, and referential to guiding documents.

- Continuous integration: automating tests, compliance, and delivery: Codify software quality by establishing automated systems to check and provide feedback within the development process. Offload as many manual processes as possible and practical to the continuous integration system.

2 Accessibility

Accessibility is concerned with how practitioners are expected to obtain and integrate software into their processes. The product that is to be obtained is the executable version of the software. In the case of compiled programming languages, this is a binary executable or library file, whereas interpreted languages typically require distributing the source code directly.

For guidance on developer accessibility, see Extendability.

The technical approaches to address accessibility depend on the targeted users. To identify methods for improving accessibility, first identify the expected users and anticipate their barriers to entry. Then, create processes and technical solutions to minimize these barriers. Finally, automate the processes so accessibility is implicit to the process rather than dependent on developers remembering to meet these needs.

2.1 Prerequisite Knowledge

Using a computer in a scientific context is a learned skill and requires years of practice to become proficient. Tools such as a "terminal," "shell," or "command prompt" are not universally intuitive, and that these three terms are used interchangeably can lead to further confusion. This is an example of a barrier to entry often encountered by early-career researchers and experienced practitioners alike. To improve accessibility, it is important to understand the experience of users and design software to meet their needs.

Following are some examples of common barriers to entry:

- Navigating a "terminal"
- Knowledge of acronyms, jargon, or interchangeable phrases
  - Command line interface (CLI), application programming interface (API), integrated development environment (IDE), and so on
  - Compile, clone, check out
  - Terminal vs. shell vs. command prompt
- Extensions: .exe, .so, . dll , . dylib
- Installation
  - Navigating package managers
  - Downloading executable files
  - Configuring an environment

Identify target user profiles including their levels of experience of understanding in computing environments. Then, design the research software so it matches the expected level of expertise in users. Note this is often an iterative process, and technical solutions are not always needed to address barriers to entry. Explanatory documentation is a major resource in addressing ambiguity or inexperience in a particular technology. Leverage existing tutorials where necessary; for example, a high-level overview of methods to use a terminal in the context of a specific software project along with an accompanying link to a deep dive into terminal training can be helpful.

2.2 Distribution

Research computing software often depend on third-party libraries, and many of these dependencies are research software themselves. Therefore, the installation and environment configuration for this type of software can easily become complex. Mature package managers are a great resource because they have a distribution system already in place and manage dependencies between software tools. The ecosystem of open source software package managers has coalesced around a few primary tools:

- Python Package Index (PyPI)

    - Source and binary distribution package manager for Python software
    - Platform: any
- Conda
    - Package, dependency, and environment management for any language
    - Platform: any
- Conda-forge
    - A community-led collection of recipes, build infrastructure, and distributions for the conda package manager
    - Platform: any
- Homebrew (brew)
    - The Missing Package Manager for macOS (or Linux)
    - Platform: ubiquitous for macOS but also available for Linux
- Spack
    - Package manager for supercomputers supporting any language and distributable product
    - Platform: ubiquitous for Linux-based supercomputers; available for macOS and Linux
- APT
    - A user interface that works with core libraries to handle the installation and removal of software on Debian and Debian-based Linux distributions
    - Platform: ubiquitous for Linux for system-level or generic packages
- Fortran package manager (FPM)
    - Fortran-specific executable and library package manager.

The process for including a package in a package management system varies, but all are designed to integrate with automated systems to prepare and distribute the package automatically upon a given event. The practice of releasing a software package after a tagged release (see Version Control) or requisite set of changes is called "continuous distribution," a component of "continuous integration." See Continuous Integration: Automating Tests, Compliance, and Delivery for details. Tools for this level of automation are ubiquitous, and a practical choice is GitHub Actions (see Collaborative Workflows with GitHub).

3 Usability

Usability is concerned with how practitioners are expected to execute the software including creating inputs and managing outputs. Though the content and promise of a particular software will bring users to it in the first place, the ease of usability is responsible for keeping them engaged with the software. In this context, consider any user interfaces including messaging back to the user through errors as the "touch points" that should be optimized. Developers should recall their own experience in using software including outside of the research environment. Contemporary software consumers have short attention spans and will generally choose the path of least resistance to accomplish a task even at the cost of access to a more advanced feature.

3.1 User Interface

The UI is any mechanism through which users interact with the software, typically by providing inputs and receiving outputs. Examples of UIs include the following:

- Graphical user interface (GUI)
- Web-based front ends
- Input and output files
- Command line interface
- Library APIs

WETO software UIs should be well defined and predictable. They should adopt the conventions that already exist in the environments and contexts in which they're used. Most importantly, all user interfaces should be well documented.

3.1.1 Command Line Interface

The command line interface, or CLI, is one type of front end for software. It is the method by which a software is executed via a computer's terminal. WETO software should in general adhere to the following conventions and principles for CLIs. However, these are guidelines and can be skipped when context is clear or another option improves usability.

- Adopt command line syntax requirements from https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html
  - Guideline 1: Utility names should be between two and nine characters, inclusive.
  - Guideline 2: Utility names should include lowercase letters (the lower character classification) and digits only from the portable character set.
  - Guideline 3: Each option name should be a single alphanumeric character (the alnum character classification) from the portable character set. The –W (capital-W) option shall be reserved for vendor options. Multi-digit options should not be allowed.
  - Guideline 4: All options should be preceded by the "-" delimiter character.
  - Guideline 5: One or more options without option-arguments, followed by at most one option that takes an option-argument, should be accepted when grouped behind one – delimiter.
  - Guideline 6: Each option and option-argument should be a separate argument, except as noted in Utility Argument Syntax, item (2).
  - Guideline 7: Option-arguments should not be optional.
  - Guideline 8: When multiple option-arguments are specified to follow a single option, they should be presented as a single argument, using <comma> characters within that argument or <blank> characters within that argument to separate them.

- Guideline 9: All options should precede operands on the command line.
- Guideline 10: The first −− argument that is not an option-argument should be accepted as a delimiter indicating the end of options. Any following arguments should be treated as operands, even if they begin with the − character.
- Guideline 11: The order of different options relative to one another should not matter, unless the options are documented as mutually-exclusive and such an option is documented to override any incompatible options preceding it. If an option that has option-arguments is repeated, the option and option-argument combinations should be interpreted in the order specified on the command line.
- Guideline 12: The order of operands may matter and position-related interpretations should be determined on a utility-specific basis.
- Guideline 13: For utilities that use operands to represent files to be opened for either reading or writing, the − operand should be used to mean only standard input (or standard output when it is clear from context that an output file is being specified) or a file named "-".
- Guideline 14: If an argument can be identified according to Guidelines 3 through 10 as an option, or as a group of options without option-arguments behind one − delimiter, then it should be treated as such.

- Adopt these minimum GNU conventions
  - A short version with one dash and a long version with two dashes
  - −v / −−version to show version information
  - −h / −−help to display help information
  - −i / −−input for input file specification
  - −o / −−output for input file specification
  - −V / −−verbose to include additional output in terminal
  - −q / −−quiet to suppress terminal output
- Use context-specific switches
  - Unix: − or −−
  - Python: − or −−
  - Windows command prompt: /

Command line interfaces should include documentation via the −−help / −h flag. For Python software, using the standard argparse library creates a help prompt automatically. Extended CLI documentation alongside tutorials and explanations of the software is helpful to attach meaning to the functionality available via the CLI.

3.1.2 Input and Output Files

The ecosystem of tools for processing data files is vast and mature. Therefore, input and output files should adopt a common file type and syntax relevant to the field and context of the software itself. For example, large datasets generated by computational fluid dynamics software are often exported in HDF5 format because robust libraries are available to export the data and load them into post-processing tools. Similarly, input files should retain a ubiquitous human-readable format such as YAML because this allows users to generate input files programmatically using standard libraries. Input and output files required by WETO software should adhere to the following conventions and principles:

- Simple, clear, and predictable structure
- Expressive and concise

- Easy to produce and consume using ubiquitous software tools
- Minimal data consumption
  - For large datasets, option to split into smaller files or binary format
- Typical and predictable data types

Following are file types with common libraries for popular language ecosystems:

- JSON - JavaScript Object Notation; a common data structure used throughout the web and in various computing environments
- YAML - YAML Aint Markup Language; not entirely but basically a human-readable version of JSON
- CSV - Comma separated values
- VTK - Visualization Tool Kit; a variety of file types and readers for different types of data
- HDF5 - Hierarchical Data Format; used for large, complex, heterogeneous datasets; HDF includes libraries for reading and writing HDF files
- Plot3D - Data type for 3D structured grid data
- CGNS - CFD General Notation System
- Markdown - A markup language for text documents
- reStructured Text - A markup language for text documents

3.2 Error Messages

Messaging to practitioners from within a software can be immensely helpful. At the same time, the infrastructure for communicating messages can be burdensome to put into place. It is important to find a balance of appropriate levels of messaging while also ensuring the messages themselves are up to date with the software features and implementations. Too much messaging results in information overload, and critical messages can be lost in noise. In addition, messaging is another developer responsibility and can be overlooked among the other responsibilities during the development cycle.

Useful error messages have the following characteristics:

- Expect the reader does not have the context of the author
  - Include a stack trace in all messages
  - At minimum, include the calling function name
- Anticipate the needs of the reader
  - What will they be thinking about when this error pops up?
  - What will they need to do next?
- Include information that will help project maintainers understand the context of the problem
  - Include metadata where relevant; see Metadata
  - Include the value of data that are found invalid

3.3 Metadata

Tracking metadata in software projects is a simple way to provide clarity to all users. This greatly improves usability and has the added effect of improving the debugging process. This information can be provided to the user in any structured output from the software. For example, output data files, reports, images, and so on can all include a snapshot of the metadata. The objective is to communicate information on the state of the software (version and runtime), the state of the computing environment, and any user decisions.

The following fields are minimum metadata to include:

- Version number in semantic versioning format (MAJOR.MINOR.BUGFIX, i.e. v3.2.1)
- Execution time
- Compile info, if applicable
  - Compiler vendor
  - Compile time
  - Compiler settings
- System information such as operating system (OS) and relevant hardware (i.e., accelerators) vendor
- Relevant settings enabled

4 Extendability

Extendability is concerned with how improvements such as new features, bug fixes, and general maintenance are added to an existing software project. This covers both the technical aspects and the management of multiple developers and development efforts happening concurrently.

The life cycle of WETO software projects typically follows a pattern of funding, development, and release, resulting in a recurring development workflow depicted in Figure 1. The "Maintenance" tasks are usually optional and implicitly embedded in future development efforts. Therefore, it is critical to the life of all WETO software to prioritize extendability so future funding opportunities are attractive to stakeholders and general maintenance and infrastructure upgrades can be introduced with minimal overhead.

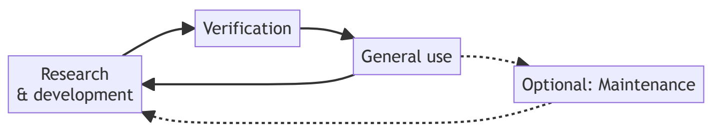

Figure 1. A representation of the typical life cycle of software extension tasks within the research environment

This topic is closely tied to the need for communicating elements of design, and the objective is to ensure developers can easily approach the project with minimal overhead required to align their computing environment, scope the work, implement the changes, and verify the results.

A guiding principle on extendability is to use ubiquitous infrastructure. Mature and ubiquitous tools and libraries come with formal and community-based documentation, ecosystems of tools such as IDE extensions, and institutional or cultural knowledge of their use and nuances that can be difficult and time consuming to create for specialized infrastructure. Common build systems such as CMake with the GNU or LLVM toolchains should be used instead of the newest projects. Popular programming languages (Python, C++, Fortran) are more approachable than specialized languages (Rust, Julia, Elixir) and enable a wider developer base. Software project managers should strive to create a development environment balancing the need for modern tooling, modern developer expectations, and stability.

4.1 Code Style

In software development, the word "grok" is often used (see usage in Hacker News, Lobsters, StackOverflow) to communicate about degrees of understanding. This word is described by its creator as follows: (Source: Wikipedia).

> *Grok* means "to understand", of course, but Dr. Mahmoud, who might be termed the leading Terran expert on Martians, explains that it also means, "to drink" and "a hundred other English words, words which we think of as antithetical concepts. 'Grok' means *all* of these. It means 'fear', it means 'love', it means 'hate' – proper hate, for by the Martian 'map' you cannot hate anything unless you grok it, understand it so thoroughly that you merge with it and it merges with you – then you can hate it. By hating yourself. But this implies that you love it, too, and cherish it and would not have it otherwise. Then you can *hate* – and (I think) Martian hate is an emotion so black that the nearest human equivalent could only be called mild distaste."

That such a word exists and is widely used in software development illustrates the high value of clear and understandable code. WETO software should avoid complexity where possible and favor readability over writability. Strive to create software that can be easily grokked by developers who do not have the current context, and remember that often these developers are domain experts rather than computer scientists.

The designers of the Python programming language consider readability a primary priority, and the most famous of the many Python language-development documents is PEP 8 which proposes a style guide for Python code. PEP 8 is summarized into 19 aphorisms (20 including one that's implied) and is referred to as "The Zen of Python". Much of the WETO software portfolio is Python-based, so these guiding principles directly apply. However, these principles are programming language agnostic and eloquently describe the paradigm for developing extendable software.

4.1.1 The Zen of Python

In an interactive Python interpreter (REPL, or run-execute-print-loop), typing

```python
import this
```

prints the Zen of Python:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one -- obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

4.2 Architecture and Design

> If you think good architecture is expensive, try bad architecture.
>
> – Brian Foote and Joseph Yoder, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*

In the development of any complex system, the design and its implementation are either explicit or implicit. Explicit design involves identifying relationships between modules, composition of data structures, and flow of data prior to writing code, whereas an implicit design evolves during the process of writing new code. In open source software, an explicit design process is critical to allowing the project to grow beyond a single developer, and the consequence of an implicit design process is the common case of technical debt.

4.2.1 Software Design Process

Primarily, an explicit design process involves identifying the fundamental principles of a particular design — how it is expected to function in various aspects. This process should result in two statements:

1. The parti, a description of the fundamental, driving design intent as a brief text (one or two sentences) or a simple diagram
2. A list of requirements that the parti and its implementation should satisfy

The *parti* is the abstract objective, and the list of requirements are the criteria to verify the implementation satisfies the parti. In other words, these are the tests for the design. Upon establishing this information, it should be codified into a design document and style guide that are made publicly available to all developers such as in online documentation.

There are various levels of fidelity to consider when designing a software system:

- Level 0: Syntax and code style
- Level 1: Function scope, function signatures
- Level 2: Module composition
- Level 3: System composition

Each should be addressed individually but referring to one another. For example, having a major design driver to limit complexity at Level 3 can be negated if complexity is allowed at Level 0. However, the definitions of complexity at these levels are entirely different and should be directly defined.

> Architecture is a hypothesis that needs to be proven by implementation and measurement
>
> – Tom Gilb, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*

Though having an explicit design process is important, it is not required to adhere to a chosen design at all cost. Throughout the development of a software, the architecture and design should be regularly revisited and reevaluated given the new knowledge acquired during implementation. How a software is ultimately used and the problems faced cannot be known at design time, so developing a process for design validation is required.

4.2.2 Design Patterns

The software engineering community has created a wide range of design patterns to address specific design problems. These are often used as a reference for creating a specific architectural design, and they often focus on fidelity levels 1 and 2. Multiple design patterns can even be pieced together to create a high-level monolithic architecture. The benefits of adopting an existing design pattern are as follows:

- The methods to describe the design pattern to new developers are already established.
- Teams can work with the architecture in the abstract to develop their concrete customized implementation.
- Ecosystems of third-party tools exist to leverage some of the common design patterns.
- Some patterns can be easily replaced by others *in situ*.

Though software architecture and software design patterns are entire fields of knowledge, many resources exist to teach common methods. Following are a few in-depth references specifically relevant to WETO-supported research software:

- Uncle Bob's Clean Architecture: A Craftsman's Guide to Software Structure and Design
- IDEAS-ECP HPC Best Practices Webinar: Software Design Patterns in Research Software with Examples from OpenFOAM
- Architecture of Open Source Applications Volume 1 and Volume 2

4.3 Version Control

Version control, typically with git, is a tool for tracking the evolution of a project change by change establishing a history of changes. Each change, called a "commit", is itself a version of the software, and, collectively, the changes provide a snapshot of thought processes and progression of work.

Version control with git can seem like simply a mechanism to "save" the state of a document, and it is easy to relegate this process to a secondary concern in the development process. However, it carries far more meaning in the context of software extendability. Because the git system is decentralized, it allows multiple developers to make changes to a project concurrently. Git also provides a mechanism for resolving differences so multiple changes can be merged together easily.

In addition to the content of changes themselves, the connectivity between changes over the lifetime of a project is meaningful. The connectivity between commits is structured as a directed acyclical graph (DAG). Each commit has a parent, and each parent can have multiple children. This provides a mechanism for easily and accurately rolling back to the state of the project at any time in history.

To best leverage the power of git to enable extendability, consider the following guidelines:

- It is reasonable to spend time crafting each commit and a sequence of commits.
- Each commit should optimize for readability in both the content of the changes and the message:
  - Keep changes within an easily communicated scope.
  - Avoid the temptation to mix formatting changes with algorithmic changes.
  - More smaller commits are generally better than fewer large commits.
- Practice editing a series of commits to ensure the progress of work is captured accurately.
- Consider whether the commit history is concise and readable to people who are not the authors.
- Become familiar with the following actions:
  - Interactive rebase,
  - Cherry-pick,
  - Squash,
  - Edit a commit message.
- Commit messages should be short, and it is a convention to limit them to fewer than 50 characters.
- An additional line can be included as a longer description of the commit beneath the 50-character line. The second line is typically limited to 70 characters, but it is considered reasonable to use as much space as needed.

4.4 Collaborative Workflows with GitHub

The processes through which developers interact with a software and other developers is an essential component of extendability. These processes should generally strive for efficiency while minimizing overhead. Automated processes are better than manual processes, and objective is better than subjective. The majority of collaborative software development processes occur on the GitHub platform, and automated processes leverage GitHub's free cloud-based resources.

GitHub and git (see Version Control) are tightly connected, but they are different systems and serve different purposes in the development process. Git is a version control system for tracking and merging changes to a software. GitHub is a platform for orchestrating and coordinating the various processes that happen around the development cycle. GitHub activities add context on top of the individual changes captured in commits. Whereas commits often capture low-level information, GitHub activities can map the low-level details to high-level efforts. GitHub provides extensive training material for git as well as GitHub features.

The primary GitHub features are described next, and a typical sequence of events across these features is diagrammed in Figure 2.

- Actions: This is a full-featured cloud computing environment that is typically used for automating software quality processes such as running tests, compiling software for release and distribution, and compiling and deploying online documentation.
- Discussions: This is typically the starting point for any collaboration. Create a discussion topic and engage with other model stakeholders to define the idea and develop a proposed implementation.
- Issues: Document the proposed solution to a problem or implementation of a new feature as outlined in the corresponding Discussion. Finalize the description and outline test cases to verify the idea.

- Projects: Collect Issues, Pull Requests, and generic cards to establish a relationship across all ongoing works in progress. This is typically most useful for large development efforts and prioritizing work for upcoming releases.

- Pull Requests: Pull Requests are a request to accept a change into a branch. This typically happens across forks of a repository, but it can also happen between branches of the same fork. During the implementation of an Issue, open a pull request to communicate work is ongoing. This is also the venue for code reviews.

- Releases: Several accepted pull requests can be aggregated to comprise one release, and this is listed in a project's GitHub Releases page along with release notes to describe the changes and communicate relevant details.

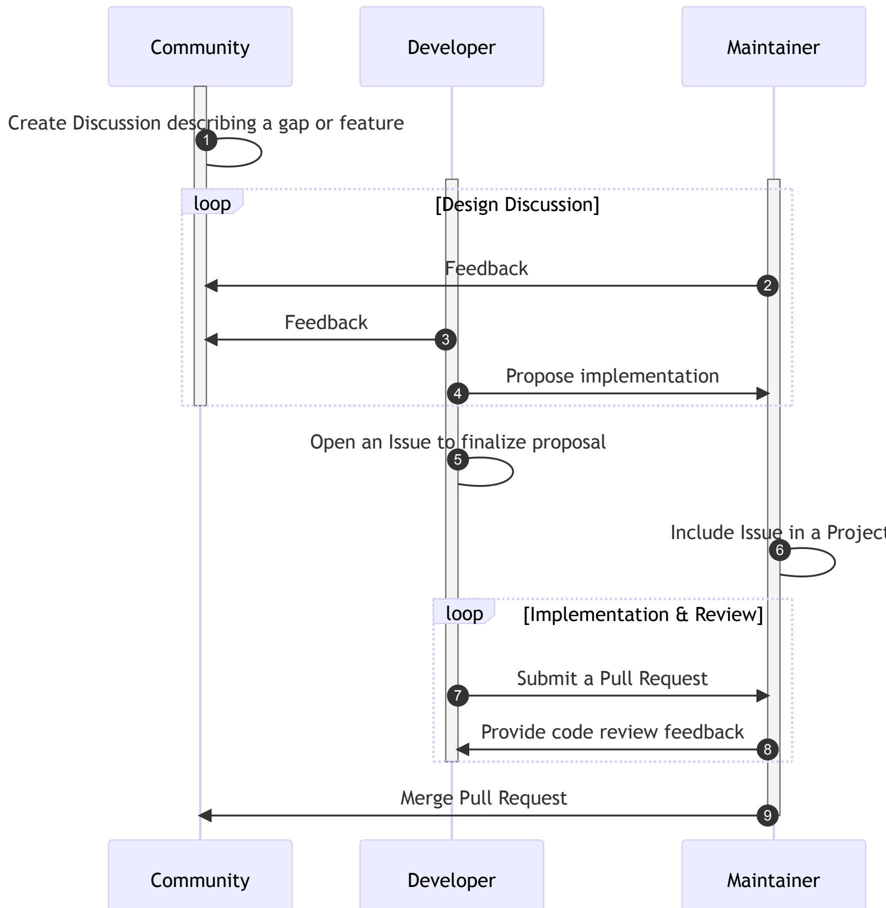

Figure 2. A representative workflow among all actors in a software development workflow leveraging GitHub features

The combination of git and GitHub provides a powerful mechanism to capture design intent, factors that lead to particular decisions, and the evolution of a project for future reference. It is important to carefully craft the messages to avoid washing out information with noise. Consider the following guidelines when engaging on GitHub:

- Descriptions of any activity should be well scoped and easily understandable.

- Pictures really are worth 1,000 words. Include a diagram, plot, screenshot, or picture when it will add clarity.

- Prefer actual text over screenshots of text. GitHub is searchable, so text provides more searchable content whereas screenshots do not. In addition, text-based code snippets can be copied easily by other users.

- Establish a practice of assigning responsibility to a core team member for each Issue and Pull Request to avoid ambiguity about how these will be addressed.

4.5 Pull Requests

A pull request, or "PR", is a request to merge a particular set of code changes into another instance of the software, typically an agreed upon "main" version. Pull request descriptions should include contextual information regarding the code change. The objective is to convince reviewers and maintainers the new code is in a good state and that its inclusion would be a benefit to the project. This typically involves a contextual description of the change, an explanation of why the change is valid, and an overview of the tests added to the test suite to demonstrate and exercise the new code.

The size and scope of a pull request should be chosen so it is both easy to explain and easy to review. It is common to create many pull requests in the development of a single feature because this process enables periodically syncing forks or branches and supports milestones or periodic check-ins throughout development. The primary objective is to optimize for readability in the pull request description as well as the code changes themselves.

Consider pull request titles and descriptions as documentation that will be relevant to future developers. When a pull request is merged, it can either be combined into one commit (squash and merge) in the destination branch or included through a merge-commit. The former does not maintain the commit history of the working branch whereas the latter does. The squash-and-merge approach is often preferred by project maintainers because of its simplicity, and in this case the title of the pull request becomes the commit message. Because merging the pull request directly affects the commit history of the destination branch, the review and merge process should also follow the Version Control guidelines. Finally, the release process through GitHub Releases can automatically construct release notes from the title of all pull requests merged since the previous release.

Though the details of workflows around defining, designing, and implementing new development efforts should be identified explicitly following the guidance in Collaborative Workflows with GitHub, pull requests, in practice, are often a good place to iterate collaboratively on the design and implementation details. Pull request reviews should have the following characteristics:

- Be very verbose with efficient but specific and complete feedback
- Be constructive rather than destructive; blame (negative) is nearly always irrelevant, and credit (positive) is nearly always appreciated
- Call out good ideas as well as bad ideas
- Include snippets of code to exercise portions of the changes
- Include plots or graphics showing the impact of the changes
- Refer to precedent or contextual conventions
- Refer to design documentation and style guides.

The GitHub Pull Request Review documentation provides a detailed guide on using the various features to suggest and integrate code review feedback.

4.6 Continuous Integration: Automating Tests, Compliance, and Delivery

The term "continuous integration", or "CI", is often used to refer to any of the many automated systems that support software quality. In its essence, continuous integration refers to the practice of deploying a change in the code directly into the production or released version. This practice is enabled by constructing a system of quality check infrastructure that gives maintainers the confidence to accept a change and release immediately. The "continuous" aspect refers to the automated nature of the quality check systems. Ideally, full continuous integration requires all characteristics and potential impacts of a code change are tested and validated automatically and without human input, such as the following:

- Requiring new code is covered by unit tests, integration tests, and regression tests

- Checking impacts to computational cost (speed) are within a threshold
- Checking memory impacts are within a threshold
- Validating documentation changes and functionality
- Linting for code syntax.

It can be helpful to break the topic of CI into three general areas:

- Continuous testing (CT)
- Continuous compliance (CC)
- Continuous delivery (CD).

Continuous testing is established by adopting a testing framework and ensuring all new code is well tested. Though automatically testing the *quality* of tests may be impractical, it is simple and helpful to automatically check the *quantity* of tests to ensure new code is covered. For the sake of a user-friendly CT pipeline, consider grouping tests into categories that can be run in parallel by the automated system. Also, minimize the time required to execute the test suite so developers get the automated feedback as soon as possible.

Continuous compliance is related to automatically checking for code style, complexity, existence of docstrings or other types of documentation, and any other requirements that describe aspects of the code itself. A common method is to use a linter for the programming language used. Most linters are highly configurable and so can be tailored to the needs and style of the development team. This step typically happens very quickly, so execution time is usually not a concern.

Continuous delivery handles how the software is exported to users for consumption. For web-based software, this involves deploying to a server, whereas modeling and analysis libraries are typically delivered via package managers or compiled binaries. The "continuous" aspect of CD refers to the practice of automatically pushing the "released" product upon any change to the primary branch or via a periodic semi-automated release process.

All aspects of CI contribute to the quality of a software project, and a full ecosystem of open source, freely available infrastructure is available to address them all. Ultimately, though, the true beneficiaries of CI pipelines are the developers and maintainers because major portions of quality enforcement and distribution are automated. Without this infrastructure, code reviews can be prohibitively time consuming and error prone, and the release process can take hours or days. By committing to the initial investment and regular maintenance, computers handle these detailed and repetitive tasks.

Given the inherent challenges in managing groups of people with various software development styles and opinions, establishing the automated systems described here can help align expectations around minimum standards for acceptance of code while reducing the burden on a project's "benevolent dictator" or gatekeeper. It is recommended to establish these processes at the onset of a software project and continuously adjust as needed.

For reference, a typical CI pipeline for a Python package is shown in Figure 3 where the square components are GitHub Actions steps.

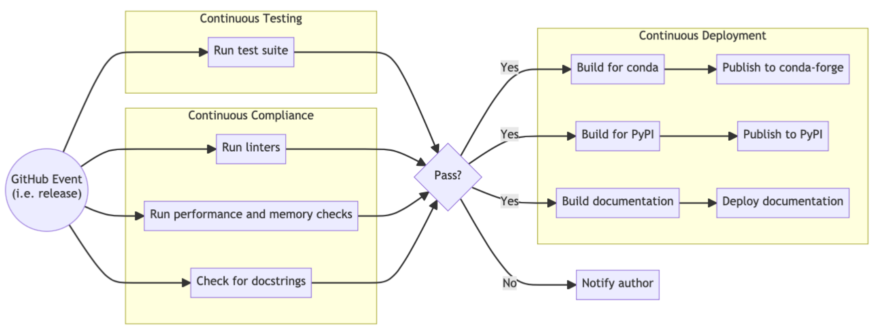

Figure 3. A typical continuous integration pipeline using GitHub features including distinct steps for testing, compliance checking, and deployment

Appendix A. RSEs: The Engineers Behind Research Software

Research software exists in a unique environment where the majority of users and developers share expertise within a specific field, and funding mechanisms are often tied to results from using the software rather than to the software itself. Because of these nuances of the research software environment, the incentives to create high-quality software are often misaligned with the career incentives for the engineers creating the software. Without the appropriate incentives, the best practices listed in this report will never gain adoption, and Wind Energy Technologies Office (WETO) software will suffer in all of the areas listed. For the sake of the WETO software portfolio and the researchers working in these groups, it is important to directly consider the needs and expectations of the people responsible for designing and implementing research software projects.

The term research software engineer (RSE) is defined by the UK-RSE Society as follows:

> A Research Software Engineer (RSE) combines professional software engineering expertise with an intimate understanding of research.

While all modern research typically involves using research software, it is common for researchers to focus skill development on either the research domain or the computational considerations involved in implementing the research in software. The research environments in academia and government labs are often structured to incentivize academic publication, so the resulting teams are commonly made up of mostly domain researchers and a minority of research software engineers. The domain researchers inform the needs of the research software and are the primary users. The RSEs design and develop the software systems as well as manage various information technology (IT) responsibilities for the group such as creating computer-based workflows, managing data, constructing web-based research artifacts, and training colleagues on best practices in research computing.

In this context, note the difference between computer science and software engineering (both descriptions taken from Wikipedia):

- Computer science is the study of computation, information, and automation. Computer science spans theoretical disciplines (such as algorithms, theory of computation, and information theory) to applied disciplines (including the design and implementation of hardware and software).
- Software engineering is an engineering-based approach to software development. A software engineer is a person who applies the engineering design process to design, develop, maintain, test, and evaluate computer software. The term programmer is sometimes used as a synonym, but may emphasize software implementation over design and can also lack connotations of engineering education or skills. Engineering techniques are used to inform the software development process, which involves the definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle process itself.

A.1 RSE Value Recognition

Writing code and designing software systems are entirely different things, and the latter must be recognized relative to the value it adds to the research process. Software design and software architecture are complicated topics covered in text books, courses at top universities, and academic publications. The process of creating a software system given various requirements is a *design process*. It involves stating requirements, iterative design, and validation and verification of the design. It can take years to fully accomplish a design objective, and at the same time the landscape of computers and software development is constantly changing. In addition, software is rarely created by one person, so RSEs must manage multiple contributors making changes simultaneously while also striving to meet the needs of the project. Therefore, consider it a best practice within the context of WETO software to recognize the contributions and value of RSEs within the labs. Avoid trivializing software design as "programming", and consider many RSEs have engineering or science degrees and treat their work as engineering or science.

A.2 Career Growth and Trajectory

In addition to acknowledgment of work and value added, it is important to provide meaningful career guidance to RSEs to both serve their personal goals and ensure the projects have well-rounded contributors. RSEs should have some level of domain experience; that is, they should *use* as well as *develop* their software. RSEs should know the context in which their software exists. They should be experts in the implementation and very good in the usage. A characteristic career trajectory within the national lab environment may take the following path:

- Year 1: Implement models; develop tests and documentation
- Year 2: Co-author analyses, improve modeling, inform work plans
- Year 3: Lead author analyses, guide future development efforts, write work plans
- Year 4: Propose new work; seek funding to expand the software project
- Year 5: Inform centerwide software culture and practices.

In general, the amount of code written by an RSE should peak around Year 2 or 3 and then taper off. The responsibility for creating software should not be entirely removed, but the majority of involvement in software development should be code review, design, and project planning. As in any career advising, the details should be a discussion between RSE, their direct manager, and the center or lab leadership.