ESIP Software Guidelines

Editor: Soren Scott

IN DRAFT Oct 2016

- Usable
  - Software has a clear, understandable interface.
    - For software with a visual interface, such as a desktop GUI, plugin/extension or web/mobile application, the interface provides a clean and clear layout.
    - For web applications or any software accepting user inputs, Unicode is well-supported throughout the system.
    - For web applications, the application implements responsive design, i.e. support for multiple screen sizes.
    - For web applications, the application adheres to progressive enhancement practices.
    - For web applications, the interface follows established guidelines for web accessibility.
    - For web applications, support current browser versions.
    - For a web service, the API is RESTful or follows a known community standard or specification.
    - For collaboration tools, ensure that the interface design does not encourage sensitive information leaks.
  - Software is performant and stable.
    - Documentation describes the current development status.
    - For web applications or web services, the documentation or project website includes service level qualities.
    - For web applications and web services, systems are monitored.
    - For high performance computing, GPU or other cluster-based software, the project provides basic benchmarking statistics.
- Documented
  - Source code is documented.
    - Source code comments use a standard comment style, related to the selected style guide or language or related to a document generation tool.
    - API documentation is automatically generated.
    - The documentation describes how to automatically generate the documentation.
    - API Documentation is versioned.
  - The codebase includes documentation to support adoption and reuse.
    - The source code repository is documented using plain text file formats.
    - The build procedure or installation process is described.
    - The documentation describes how to run simple examples.
    - The documentation describes how to execute the test suite.
    - The versioning scheme is described.
    - Release notes or changelogs are provided for each release.
  - Software is documented.
    - Any Documentation
    - User documentation

- Code or software requirements are documented.
- Project is documented.

- Secure
  - Source code, including automation configurations and scripts, have been sanitized for public release.
  - For web applications and services, software follows industry recommendations for secure practices.
  - For any code or software system, follow the recommended security practices for each system component.
  - Keep systems and dependencies up-to-date.
  - Collaborative platforms include tests for permissions and integrations.
  - Containers follow recommended practices for security.
- Sharable
  - Source code is licensed.
  - Source code includes configurations for automated systems.
  - Project name is discoverable.
- Governed
  - Contribution policies are provided.
    - For any project, describe the workflow used by the project for contributions to the source code (or other versioned items).
    - For any project, describe commit message, pull request, and issue preferences.
    - For any project, define the code review process.
    - For open source projects, provide a code of conduct or state clear expectations of behavior and communication.
    - For open source projects, provide guidelines about what contributions will or will not be accepted.
  - Development activities are transparent.
    - Development activities are managed through an issue tracker or similar software.
    - Project provides support mechanism(s).
    - Project provides a development roadmap
- Code as Research Products
  - Publication and Citation
    - Software, as binaries and source code, are published to a sustainable third-party repository.
    - Documentation includes citation details.
  - Preservation/Archiving
  - Credit
  - Provenance
  - Reproducibility/Replicability

- Progression, Sustainability and Reusability/Adoption

Code, software, and other such products are often developed in the course of publicly-funded research. “Code-wise,” there is nothing special about these research products in themselves; however, unlike commercially-derived code and software, research products bear the additional burdens of reproducibility, publication, and preservation, because of the nature of the funding that produced them.

The Federation for Earth Science Information Partners (ESIP) recognizes the role code plays in contemporary research practices as well as the complex issues community members face when approaching technology-dependent research. With the cross-disciplinary nature of the ESIP membership, we feel well positioned to develop guidance and recommended practices for the Earth observation and geosciences communities in a way that recognizes the domain expertise within these communities and that supports the communities’ desire for improved practices around research code and software.

Current discussions around code and software practices in the broader research community often do not include representations from either the Earth observation or geosciences communities. And while we agree that code is code in any domain, public or private, we believe that it is important that these discussions include a broad range of researchers, including faculty, research staff, developers, graduate students and postdoctoral fellows. Such inclusion is necessary to ensure that the practices evolving from these discussions have the broadest possible usage and application. Our goal is to develop meaningful and practical assessment procedures that address the concerns of all researchers, and to develop training and support networks for early career and future researchers.

Motivation

Our motivation for developing these guidelines is twofold. First, we recognize, both within our research communities and within funding agencies, that code is vital to many of our research

activities. It is, therefore, crucial to support researchers, at all levels, in developing high quality code to meet the needs of these activities, and to continue to do so as capabilities and requirements grow in complexity and scale. We also have the additional burden to understand and support our research community practices. For researchers and research groups, integrating these guidelines can save time, improve trust in the code product and its outputs, and reduce the amount of effort needed to meet community expectations for publication and reproducibility, among other benefits.

Second, with the increased focus on return on investment for grant-funded cyberinfrastructure and, perhaps more crucially, a new focus by funding agencies on “time to science” (where some cyberinfrastructure, or software product, reduces the time or effort needed to address the primary research goal), we see renewed interest in evaluating the adoption and impact of cyberinfrastructure solutions. We stress the need to develop evaluation processes in ways that reflect current code outputs and that encourage adoption of development and management practices to support improved research outcomes and impact.

This document will provide a broad overview of current development practices and provide guidelines for improving those practices. We will then discuss the relationship of these guidelines to the larger research development ecosystem as well as their limitations. We hope this discussion leads to equitable evaluation practices, and, more critically, to new approaches to supporting high quality, sustainable research development.

Background

This work uses the criteria for research code developed by the Software Sustainability Institute (SSI, Jackson et al. 2011) and extends their use through a larger evaluation framework developed by ESIP in collaboration with NASA AIST to verify Technology Readiness Levels (TRL, NASA 2012). The evaluation framework was used in a pilot effort (Burgess 2016, Graybeal 2016) over the course of autumn 2015.

This effort results from both community discussions around these criteria and from ESIP’s larger interest in technology evaluation as a mentoring activity rather than as a verification activity. The guidelines outlined in this document will serve as the foundation for future assessment and training implementations.

Developing Software Guidelines: A Community-driven Process

We began developing these guidelines by adapting the SSI’s criteria so that they would better meet the diverse needs of the interdisciplinary ESIP community. During this process, we made a conscious effort to solicit input from active developers. Our aim was to have roughly even representation of developers on one hand and researchers, managers and/or principal investigators on the other.

Formal revision of the initial draft was undertaken through three different activities. The first involved a remote sprint where ESIP members volunteered to review different sections of the original criteria based on their domain expertise and interest. The remote sprint ended with a half day workshop during the ESIP Summer Meeting, July 2016. The second activity involved community discussions with members of the Boulder Earth and Space Science Informatics Group (BESSIG). The last activity was the Joint EarthCube and ESIP Workshop on Software Assessment (Boulder, CO; June 2016). Two achievements that resulted from this workshop were, first, that the existing guidelines were revised and new guidelines for interoperability were established, and second, that EarthCube and ESIP's roles were established for addressing the larger EO and geoscience needs for improving both research code and software practices.

What Is Research Code?

For our purposes, research code and research software are any code outputs developed through grant-funded activities. This understanding is broader and more inclusive than that found in more traditional “science code” definitions, and it is a necessary expansion for supporting the current needs of our research communities. Research code and research software are critical to many contemporary research activities (Hettrick et al. 2014).

Here, we also make a distinction between research code and research software. Research code can be scripts, data processing or analysis workflows, or other codes developed to generate data, perform analyses, or perform other project-specific tasks leading to artifacts published through journal articles or otherwise. This kind of code is usually not intended for community adoption through open sourcing or public release, nor is it included as an explicit deliverable on a grant. The potential for long term support for these codes is low, but these codes are nevertheless important to the community for provenance and reproducibility, replicability and publication.

For its part, research software is any code product, whether as scripts or as some packaged object, that is intended to be released for community adoption and use. Intention of release is not a complete description of what we mean by research software as we also include intentional design or systems understanding for a defined purpose. This software is most likely an explicit deliverable for a grant. These outputs are valuable for provenance and other community practices, and have the additional potential of longer term support through both ongoing maintenance by the research group or other contributors and through continued funding.

The distinctions here are related to the purpose of the release and the prospect for ongoing maintenance. In practice, these factors usually do not affect the guidance provided, and for most of this text, the terms research code and research software will be simplified to “codes”. However, there are certain circumstances in which the differences matter, and these situations will be noted.

Community and Stakeholders

Our focus on assessment for verification and guided educational activities encompasses a wide range of stakeholders. Table 1 outlines the main stakeholder communities and related use cases.

Table 1. Stakeholder use cases. Note that different stakeholders may have overlap in desired outcomes.

Stakeholder	Use Case	Desired Outcome
Funder	As a funding agency, we're interested in evaluating the software projects we fund.	A functional evaluation system based on accepted metrics.
Project Manager, Principal Investigator (manager in practice)	As a manager, I'm interested in using the rubric/progression as a learning tool to help improve the development practices in my research group.	A checklist or other informal assessment to help the research group meet funder's expectations and to determine the next steps for training or related activities in the research group.
Principal Investigator	As a PI, I would like a tool to assess our progress and to ensure we're meeting our funder's expectations for a software project based on the readiness level stated in the original proposal and as defined by the funder.	A checklist or other informal assessment to help the research group meet funder's expectations, and to determine the next steps for training or related activities in the research group. This informal assessment would also provide aid for formal reviews.
Science Software Developer, Researcher who codes	As a science software developer, I'm interested in using the recommended practices to improve my own workflow and skillsets.	A checklist or mentoring activity to help guide me towards training options to meet my research and skillset goals.
Developer	As a developer, I would like community-supported guidelines to support requests to change our current dev team practices.	A checklist or informal assessment to encourage my manager or PI to allow the development team to adopt appropriate practices.
Grad Student, Post-Doc,	I've taken the introductory courses and want to continue to improve my	A checklist or mentoring activity to help guide me towards training

Researcher interested in continuing code education	skills but don't know what steps to take next, and I'd like guidance based on my skillset.	options to meet my research and skillset goals.
Research Community	We want to provide educational materials or other support for community members to meet their goals regarding research software implementation and career growth.	A set of guidelines for technology assessment, and the framework for using those guidelines as educational tools.

We must understand that research software projects are undertaken by a variety of research groups that have varying degrees of resources available. These groups range from well-supported institutions with dedicated developer teams to individual PIs with only part-time developer support, or with small teams of graduate students with varying levels of experience in software development. With that understanding in mind, we have structured this guidance to demonstrate worthwhile and concrete actions that can be taken to improve development practices within a given research group's current resource limitations.

Over time, some individuals will take on different stakeholder roles at different stages of their academic careers, and their educational goals will take on a different slant. They may find, for instance, that the skills that were adequate to manage code at the graduate student or post-doc level are no longer adequate when they become principal investigators of more complex technical projects. Likewise, in transitioning to a program officer role at a funding agency, one may find that one's technical experience is not adequate for developing functional and realistic metrics for project success. While these issues are not the explicit focus of this document, we hope that the discussions here bring some awareness to these different needs.

Use cases are more fully explained in the Scenarios section below to provide more concrete examples and to address the assessment implications for maturity and for education in real world situations.

Research Software Projects

Scenarios

In this section, we will describe a number of common scenarios where research produces some code or software, and where the expectations for assessment, development process and project maturity, researcher education, and code sustainability will play out differently.

Scenario 1. An individual developing data processing scripts.

I am a grad student or post-doc without any computer science or software development training and I need to develop some data processing scripts for my analysis. I know these scripts will be made public so I'd like try to follow good code practices but don't know where to start.

Assessment: The developed scripts are secondary research products where the desired products are the data and analyses. The code is unlikely to undergo formal review but is expected to meet basic code standards and publication requirements.

Maturity: Few expectations for high levels of process maturity although this is dependent on the nature of the scripting. However, since it has data production or analysis as its primary goal, correctness, and demonstrations of that correctness, are expected.

Education: Good code practices for clean code and tested code. Accepted publication practices for the community.

Sustainability: No ongoing support from the grad student, post-doc or research group is expected if the code is published to an external archive.

Scenario 2. A new system based on a single grant.

Our research group is starting a new project to develop an analysis platform as a web application. It will use a well-known JavaScript framework and will be developed as open source. We plan on developing our community to support active contributions and to support the application's adoption in areas where the analysis is useful.

Assessment: Verification that the new system meets the research goals and meets expectations to support reuse and adoption.

Maturity: Processes must be mature enough to allow external developers to set up a development environment and contribute features or patches. The group considers it beta software so stability is not expected.

Education: Guidance on developing and supporting community contributions. Identification of next steps for an operational system.

Sustainability: Demonstrations of community activity and successful engagement to show adoption potential. Could consider outreach for early adopters in a specific research community.

Scenario 3. An existing system with a single grant to add new features.

Our research group has an existing project, a geospatial processing library. Our grant includes development time for several new features that we'll release as a new major version.

Assessment: An initial assessment of the existing system is necessary to establish a baseline maturity level before evaluating the current grant work. Assess the new features against this baseline with the expectation that the implementation meets the baseline maturity and raises concerns if lower. Include evaluation of the new features against the stated research goals.

Maturity: As noted, the maturity of the new features are expected to be at or above the level of the existing system. Process improvements may also occur during the grant cycle.

Education: Identification of next steps for an operational system.

Sustainability: We could assume this is managed as an open source project where improvements are made with a research goal in mind. Steps towards gaining contributions of funding or effort from other sources have been undertaken.

Scenario 4. An existing system with multiple grants, one of which adds a new standalone component.

We have an existing platform for data access that is currently receiving support from multiple grants. One of the grants is funding an extension to the platform that will be developed as a standalone application (middleware). Our platform is based on an open source project developed by a different group not supported by this funding (we are reusing the OS system as a base for our own efforts).

Assessment: Evaluate the standalone component for completeness and functionality at the maturity levels laid out in the grant. The integration of the standalone component with the existing system may also be evaluated.

Maturity: Started at known levels at the start of the grant with no expectations of improvement.

Education: Project manager would like information on the assessment criteria based on the stated project goals.

Sustainability: Project has reached a stable state, continues to look for adoption at other institutions and funding opportunities.

Scenario 5. An existing system with multiple grants, one of which supports a new data product.

We currently maintain a data portal with legacy code. In our new grant, we are collaborating with other groups, but we are not responsible for the main development tasks. Part of our requirements are to add new data products in support of the larger

collaboration. We need to write processing code for those products; the code won't be integrated into our legacy platform and won't be needed long-term.

Assessment: Evaluation of the larger collaborative project where the successful generation of the data product is relevant to demonstrate interoperability/integration, but the code itself is not evaluated.

Maturity: Has reached some stable maturity levels, but the state of the legacy platform is not relevant to the assessment.

Education: Research group would like guidance on automation practices to better support data pipelines in the future.

Sustainability: None. Code will be published as an archive for provenance.

Scenario 6. A single grant with multiple collaborators providing outputs/functionality through their independent systems.

We are demonstrating a new analysis platform that relies on distributed data sources. One collaboration group is developing the new analysis platform, one is implementing new features in an existing platform and the third developing a data pipeline process. The success of the project depends on the integration of the three systems and the groups are not contributing to one codebase.

Assessment: Assessment at the project level, focusing on the successful integration of the three activities. If the analysis platform is intended for potential reuse, additional assessment may be considered for that component only.

Maturity: The new development, seen in the analysis platform, is expected to meet maturity expectations for a prototype web application while the two data providing systems should work within the maturity level of the existing systems.

Education: The collaboration groups providing data may want information on the interoperable service implementations. The project collaborators are interested in understanding the assessment process.

Sustainability: Sustainability is likely relevant only to the analysis platform as the component intended for reuse/adoption, certainly from a funder's perspective. The two data systems ideally would continue to support the new functionality, as early adopters of the system or to support other downstream activities.

Scenario 7. An existing system with a grant specifically to provide maintenance.

We are the maintainers of a successful visualization tool that has reached a point in its lifecycle where major refactoring is required to continue to support future activities. The

grant doesn't include any new functionality, but may include upgrades to the language version, bug fixes and refactoring to manage technical debt.

Assessment: Assessment requirements unlikely unless revisions are extensive.

Maturity: Expected to remain at the maturity level at the start of the maintenance effort or improve based on the stated goals.

Education: No explicit education requirements.

Sustainability: Project is expected to continue community engagement and support activities (assuming major upgrade) and continue pursuing additional funding, whether through additional maintenance grants or for extending the functionality.

Hopefully these scenarios, while clearly not a comprehensive list of all possibilities, provide some insight into how different project types and phases might be assessed.

Development Process and Project Maturity

Development process maturity and project maturity, identified either through longevity or by some other defined criteria, are not linear, nor do they necessarily coincide. A project can start with a high level of development process maturity in an initial proof of concept phase. For its part, project maturity, often presented as governance patterns similar to traditional open source projects, can begin the same way. Improvements in both can occur within the lifespan of a single grant, or they can happen through a series of grants. Here we can see the potential mismatch between the way that funding agencies often assess and categorize a project and the way that software develops in reality.

Additionally, a mismatch between assessment and maturity can occur when, for example, a codebase does not meet the desired level of code maturity or project maturity, but is nevertheless successfully serving a communal need. Maturity markers are indicators of project health, and we strongly encourage their use, but not to the detriment of the larger goals of providing a product that meets a research community's needs.

Within the diversity of research software projects, two overarching categories can be distinguished that bear on the assessment of maturity: first, code/software intentionally developed for reusability and adoption, and second, code/software developed for highly specific project needs. Neither category can neglect the recent requirements for openness, notably the U.S. Office of Science and Technology Office's policies on Open Science (Holdren, 2013) and the increasing number of academic journals requiring data and code publication; however, the methods used for achieving those requirements, and expectations around development process or project maturity, are different in each category. For projects developing code or software not intended for reuse and adoption, the focus for assessment is on development process maturity,

documentation and preservation or publication. (We should note that “not intended” reflects stated project milestones, sustainability and community engagement activities as per the funded proposal.) For projects developing code or software that is intended for adoption, the focus for assessment is on project maturity related to governance and documentation as well as development process maturity.

We are focused here on code/software developed by a research group. For the purposes of assessment, we exclude from discussion of maturity any software products reused by a research group,while noting the value in assessing project maturity. More specifically, we support, and encourage, a research group’s decision to reuse a current, outcome-appropriate technology over reimplementing a similar technology. We do not, however, consider the reused technology to be an assessment target itself. A simple example is developing modules for an existing web publication framework instead of developing an entirely new framework.

On a related issue, in cases where a project employs an existing technology to support the publication or release of a research product that is not itself code or software, the pre-existing technology again would not be considered during the assessment of the research software or code. A very broad example of this would be developing datasets and providing them publicly through ESRI’s ArcOnline platform. In this case, a project assessment would consider the datasets, but not ArcOnline. (Guidance for quality and maturity for other research product types, such as datasets or ontologies, is beyond the scope of this document.)

Analytics and altmetrics are included here to highlight two aspects of analytics for assessment. First, projects at different stages of their lifecycles are characterized by different outcomes, and those outcomes may not occur within a specific funding period. Second, our guidance focuses on those activities that a research group can usefully act upon to meet its own maturity goals. The success of these activities may, at least in part, be demonstrated through analytics and altmetrics. Understanding how those metrics relate to funding cycle timelines is necessary before including them in an assessment process focused on activities within a cycle. We suggest caution, then, in interpreting what is being measured, and in being aware of when in the cycle it is being measured.

Systems

As exemplified by the scenarios, when we talk about maturity and/or assessment, we are often discussing existing systems maintained by a research group or a larger collaboration. From the funder’s perspective, a formal evaluation of existing products is not cost effective; however, there are instances when assessment requires an understanding of the existing system’s baseline maturities. We discuss this requirement further in Progression**,** where the context and research goals of the grant play a role in determining the assessment need. Regardless of assessment requirements, understanding the current state of a project is necessary if a research group wants to evaluate whether process changes are leading to improvements or to regressions.

We are not starting with a tabula rasa for research cyberinfrastructure in general or when beginning new grant-funded activities. Nor do we often start cyberinfrastructure projects without taking advantage of existing platforms, frameworks or libraries. There is a need to assess the selection of these dependencies as they relate to the overall project goals, even if we are not evaluating the maturity of the dependencies themselves.

One final note regarding these systems: technology changes rapidly. Our cyberinfrastructure and research development challenges often require complex solutions, and guiding system architectures is beyond the scope of this document. There is often no single solution for many of our research activities. It is the responsibility of the researcher or research group to investigate current technologies that may address the specific project needs and select those technologies accordingly. Researchers must consider the project requirements, previous efforts, and the pros and cons of available platforms and solutions.

Goals of this Document

1. Provide resources and guidance to different kinds of research groups and other stakeholders to support research code/software development within our research communities.
2. Provide a progression model and related discussion around sustainability, adoption and reuse, and assessment that reflects the variation in research development products.

Guidelines

Research code, in itself, is not special as code; however, as a product of research funding, it comes with the burdens of reproducibility, publication and preservation. The guidelines presented here are intended to help researchers solve problems they encounter while addressing these requirements. We will structure our discussion around the reasons for the requirements, and wherever possible, we will refer to current industry and/or research community practices and recommendations.

Some aspects of development process or project maturity may not relate to every type of software or code activity. Table 2 describes the software types we considered when developing these guidelines. Where necessary, additional guidance is provided for a specific type if the general guidance neglects some aspect important for that software type.

Table 2. Code and Software Categories.

Code/Software Type	**Description**
Processing scripts	Code developed for one-off processing or other project-specific activities, not generally intended for

	adoption and reuse.
Notebooks	Hybrid documents containing code, documentation, and analyses.
Module/library/command line interface (CLI)	Packaged or buildable codebase, usually serving a specific purpose, often intended for adoption and reuse.
Plugin/extension	Packaged or installable software provided as a component of another piece of software.
Desktop Application (GUI)	Installable software developed for use on a desktop with a visual interface.
Web Application	Software deployed to a web server with functionality provided through a visual interface.
Web Service	Software deployed to a web server, providing data access or other functionality through a Web API.
Platforms/Systems Software*	Operating systems, utilities, servers, database engines or other systems developed to serve or support application software.

* Programming languages, such as Julia or R, are outside the scope of this document; however, the guidelines provided here support development activities within the specific language ecosystems.

The guidance provided below strives to be language- and platform-agnostic. Wherever possible, the examples provided are not an endorsement of a language or product and are intended solely as examples of the larger concept. When choices must be made by a research group on how to implement a concept, we suggest that in almost all cases there are many acceptable paths. Choose one, support it and move on to the project implementation.

Code notebooks, supported in platforms like Jupyter, R Notebook, Apache Zeppelin, are hybrid products containing code, documentation and analysis outputs. They can serve as processing scripts and workflows for generating data and analyses or as documentation and tutorials for a code or software project. When approaching the guidelines below, consider the different elements within a notebook, as code or as documentation, when considering the guidelines.

Finally, while we strongly support open science and the principles of accessibility, reproducibility and preservation of research software and code, we acknowledge that there are resource limitations, differing levels of experience and differing needs, both within and among communities. Choice of language, choice of platform, choice of tools—these are driven by the specific research and community needs rather than any dogmatic adherence to a particular philosophy.

Principles

The Guidelines section is organized into eight principles to define the characteristics we support for research code and software activities, regardless of the specific products.

> **Sustainable Code:** The code itself is well-structured and readable and is developed to support understandability and maintenance over time.
> **Interoperable:** The code or software uses, generates or serves data using recognized community standards or formal standards.
> **Usable:** The software is developed using recommended practices for the given software type to create functional, performant and stable interfaces in order to promote adoption and improve research practices.
> **Documented:** The project includes documentation to support understanding at different levels throughout (code, reuse and adoption, project description).
> **Secure:** The code/software is developed using recommended practices for securing deployments, systems and user interactions.
> **Sharable:** Project incorporates practices to ensure the code/software products are open to reuse and adoption for future research activities as well as for preservation.
> **Governed:** The project uses established practices to support community engagement, growth and transparency to encourage adoption and reuse, especially for those projects released as open source.
> **Code as Research Products:** The project supports community practices to encourage and support the practice of research, including citation, preservation, provenance and reproducibility/replicability.

From these principles and our understanding of the range of research product development activities, we can provide a core model to reflect the minimum set of criteria for any research development product to meet:

1. Sustainable Code
    a. Clean, standardized code
    b. Versioned code
    c. Tested (strongly recommended)
2. Documented
    a. Source code is documented.
    b. The codebase includes documentation to support adoption and reuse.
3. Sharable
    a. Code is licensed.
4. Code as Research Products
    a. Publication and Citation

These cover the basic requirements to support open science activities. For graduate students and those new to development projects, we encourage you to start with these core guidelines.

Sustainable Code

Sustainable code deals with the characteristics that make source code maintainable as time passes and contributors change.

Clean, standardized code

The codebase is well-structured and consistently structured.

The text of the code is formatted in a clean and consistent manner throughout the codebase. Function (method) names and parameter (variable) names follow a standard structure such as CamelCase.

Code follows the language's recommended style guide or the research group's style guide.

A code style guide describes the formatting conventions preferred within a language community or within a research group. Other organizations, such as Google or Mozilla, also publish style guides for different languages. The key is in selecting a style guide and consistently applying it.

Conformance to the style is verified using a linter.

A linter is a small utility used to check for style discrepancies and other potential issues in source code. These can be integrated into a preferred development environment, such as the vi or emacs text editors or into a GUI-based IDE like Eclipse. Using a linter catches syntax or formatting errors quickly and conveniently.

These are language dependent, for example, Python's Pylint, Javascript's JSLint or Oracle's Java guide.

Linting is included during automated testing.

Once an automated testing system is in use for a project, ensure that linting is one of the tests included. Often, this is included in continuous integration processes. See Tested below.

Regardless of whether a codebase is open sourced, i.e. accepting outside contributions, having a code style guide for your research group or for a project is encouraged. Emphasizing clean code practices aids in maintenance during active development or during long-term support. Code is the interface for developers so maintaining clean code over the lifespan of a project and as contributors change is encouraged.

Errors and exception handling follows recommends practices for the language used.

Use language and module-defined exceptions in most cases. Catch specific errors and handle those appropriately for the code/software. For web applications and services, do not publicly expose full stack traces in any operational environment.

Versioned code

Software (or code) versioning is important for open sourced code and for research activities related to citation, reproducibility (Stodden and Miguez 2014) and provenance.

Source code is maintained in a version control system (VCS).

Any code should be versioned in a version control system of your choice, i.e. git, Apache Subversion, or Mercurial. It is, for individual researchers or local teams, not critical which system you choose so long as it's being used. For code or software intended to be publicly released and released for adoption, it is important to note that, at the time of this document, git-based systems are widely available and widely used.

For code notebooks, include the exported code (for easier comparison of modifications) and the exported HTML (for visual comparisons of generated outputs). The generation of these files can be automated depending on notebook platform.

These repositories do not need to be public to start a project or during active development. VCS's can be run locally, i.e. installing git on your laptop, on a private system for local team development, using a third party VCS system such as BitBucket or GitHub that offers private repositories or through larger institutional systems.

When considering a VCS, consider the funder's delivery and assessment requirements. Code repositories can be migrated from one VCS to another, they can be forked or cloned and, in the process, the repository can become decoupled from issue tracking systems or other related structure that are part of an assessment. See Governed and Code as Research Products for information on related guidelines. We note this issue, and it is unresolved for distributed VCS.

Note that versioning, through remote VCS systems, is not a replacement for backups.

Source code uses a versioning scheme.

For code that is intended for adoption and ongoing maintenance, implement a versioning scheme. A versioning scheme, such as Semantic Versioning or the GNU preference, here refers to the release version used in packaging and generally offers major and minor numbers, for example, "version 1.2" has a major version number of 1 and a minor number 2. This process gives adopters, using packaging tools or other automation tools for managing dependencies, a

way to more effectively manage those dependencies and a way to manage those dependencies within their development cycles.

See Documented for related activities.

Source code uses tagged releases.

Depending on the version control system in use, use tagged releases. A release is basically a specific commit in the code repository that meets some milestone conditions (defined by the project) and is stable (or not yet stable but released for beta testing). This provides adopters both an indication of when to upgrade to a stable version and, when combined with release notes (see Documented), some understanding of the effort needed to incorporate that new version into their own codebase or workflows.

Redistributable

The method for delivering the software depends on the project requirements. For software projects implementing desktop GUIs, for example, pre-built installers may be made available. For other projects, the preferred method may be providing scripts for automated build tools and installation instructions.

Source code includes build scripts to automate builds.

Indicate what system is used to perform the build and provide configuration files or related information. For some languages, builds are described as package installs.

For dependencies that aren't installed as part of the build, provide information about those requirements and procedures. Otherwise, use the dependency management of your selected build tool to ensure any required dependencies are installed during the build process.

If your software has optional dependencies, indicate that and provide installation information. Note what features or capabilities are supported by each optional dependency.

In some languages, packaging tools can provide binaries that serve effectively as installers. Consider providing this option if it meets project goals and community practices for that language or your target audience.

Scripts or configurations are provided for creating binary installers.

For software delivered as an installer, provide the language-specific configurations and tool information to generate a new installer. (This assumes the code is released as open source.)

For desktop GUIs, provide an installer.

Provide an installer, tied to a versioned release, for any desktop GUI. Indicate which operating systems are supported and provide an installer for each.

Whenever possible, include an uninstaller or provide documentation to allow someone to perform a clean uninstall of the software.

Note that systems can be delivered as runnable containers, Unix/Linux distributions or other options. These prepackaged options (not to be confused with code packages) should follow recommended security and creation guidelines as provided by the platform used.

If the product is a code notebook, redistribution is often publication to a hosting platform like Jupyter's NBViewer or another public platform that can render the notebooks like GitHub.

Tested

Testing provides numerous benefits throughout the code or software lifecycle like basic quality control, verification of builds or installation, and effective and efficient means of assessing contributions to the code base from internal team members or external contributors. In a well-designed test, you are setting the expectation of success and writing your code to meet that expectation so it is documenting intent and providing a means of verifying that intent as the code changes.

We promote code/software testing regardless of method. The style of testing during the active development cycle, whether test first (often using Test Driven Development (Beck 2003; Ambler 2013) or Behavior Driven Development) or test last (where testing is an integral part of the development process without adhering to the TDD methodology), is not as critical as encouraging and maintaining a strong testing culture in the research group. Integrate testing early in development (where possible), select a testing workflow that fits the group's development workflow (and be open to adjusting the testing workflow to support continued, good quality testing rather than strict adherence to any particular methodology) and focus on developing high quality tests—these efforts provide direct benefit to a research group in their development efforts (as executable documentation, early bug detection, workflow improvements with automation and promoting overall trust in the product) and longer term benefits for the community as a whole (promoting trustworthy products, as code/software or the data/analyses generated from it, and supporting reproducibility, replicability, and preservation goals).

Tests, like documentation, are only usable if kept up to date. A reasonable criteria for adoption is that a code base does not include failing tests and certainly not failing tests that have been in the code for a long period of time. Code coverage can promote unsustainable practices; we recommend, if nothing else, testing key aspects and integrations. This includes testing for success states as well as exceptions or exceptional conditions.

Source code includes unit and integration tests.

Unit and integration tests (and, further, functional tests) help ensure that the code is meeting expectations at different levels of abstraction in the system: unit tests at a narrow component

level and integration tests to confirm the behavior of some small set of components. Testing frameworks (or harnesses) are available for most programming languages, for example Tape (Javascript), MOxUnit (Matlab and GNU Octave), or RUnit (R) as well as those provided in the core packages of a language (unittest (Python) or testing (Go)).

If the project provides specifications, ensure that tests cover the functional requirements within that document. Otherwise, consider ensuring coverage for those areas most likely to affect the user, your research or science goal.

For code notebooks, testing is often considered at the notebook level rather than the code block level. Some platforms may have third party modules to provide support for unit testing within the code blocks, but this is platform and language dependent. If the task benefits from testing at the code block level, take advantage of these modules; if not, consider the notebook the unit and the successful execution of all code blocks in order as the test.

For code and software projects requiring a higher degree of correctness, include assertions to help verify the code and diagnose bugs (Regehr 2014; Kudrjavets et al. 2006). Assertions can tell us if assumptions we've made about the code at some point in the execution are correct as we execute the code and without becoming disconnected from the code, as might happen with providing those assumptions in a comment block or specification. Assertions don't replace testing as they each serve different and valuable functions in the codebase.

Tests limit dependence on external services where possible.

Where possible, limit the use of external resources as inputs for the tests. Use mocking (Fowler 2007) or include test infrastructure to build database dependencies (Narla and Salas 2012) or similar tasks. Tests should be as realistic as possible with regard to the system and with an eye towards efficient testing. Related, mocking allows tests to be developed for conditions that may be difficult or impossible to repeat from an external system.

Testing is automated through a continuous integration system.

Continuous integration is a development process that allows developers to commit small, discrete changes (bug patches or a new feature), into a shared codebase in a way that doesn't "break the build" for other members of the team. Continuous integration is a way to enforce code quality, (code isn't merged into the main branch if tests fail), to avoid large merge issues (conflicts in the code submitted by two developers), to catch integration bugs early in the development process, and to provide a process for acceptance testing in reproducible environments. Common tools include Jenkins, Circle CI and Travis CI which can be integrated into a GitHub workflow or, depending on the tool, run in a locally-hosted environment.

Code notebooks can also be tested in continuous integration systems. This automation is encouraged for notebooks considered research products or documentation.

Note that continuous integration relies on several of the concepts outlined in this document before it can be implemented, such as version control and versioning, a test suite, and automated builds. Nor is it limited to simple unit testing.

For web applications, include automated GUI testing.

Automated GUI testing extends code tests to the interface, giving developers a means of ensuring that the user interface remains functional. This includes repeatable web form interactions, consistent interactions across browsers, and visual tests to ensure web pages render consistently and as expected. Tools such as Selenium or Watir provide options for automated testing against web interfaces and across browsers.

Interoperable

These guidelines are intended for any code or software that provides web services or data for consumption by other platforms; however, we are mostly focusing on the web application/web service projects.

Interoperable services fall under the same guidelines for performance and stability. See Usable for information.

For data services

Use community standards or conventions for data and metadata formats as well as the web services delivering those data.

Ensure that the data is usable by validating the data format during testing, for example, ensuring that an ISO 19115 file is valid XML.

Ensure that the web service is valid. Often this means validating a capabilities document against the schema provided by the standards body or through a web validator (CWICSmart for OpenSearch, the OGC Web Validator). At a lower level, ensure that the web service returns valid HTTP status codes and uses them for their intended purposes. Verify that content headers, especially those related to language and character encodings, are present and correct. Include service validation as part of the test suite for any generated service.

Ensure that the web service is complete with regard to the goals and use cases of the project.

Indicate, in the web API documentation for a service, that the service has been validated and the validation method, and that it is valid.

We note that often the development of systems supporting interoperability include external frameworks or platforms that are integrated into the project system. Given that these guidelines

are intended only for code and software developed by the project team, some of the above may not be feasible.

See Documented for API documentation discussions.

Applications, desktop or web-based, supports community data standards for inputs and outputs.

Whenever possible, use open standards for data inputs and outputs.

For Semantic Services

Data dictionaries for each appropriate data output is provided in the documentation.

If implementing a standard that supports it, provide codelists or controlled vocabularies.

Provide ontology or vocabulary services. Every collection and concept/term published online should have a persistent, dereferencable URI and a definition.

If providing RDF or JSON-LD formats (or other Linked Open Data formats), provide the ontology. Follow recommended practices for LOD services such as including a license, ensuring forward and back-linked URIs are available, providing provenance metadata and providing performant services (Zaveri et al. 2015).

Publish the ontology in a stable, persistent community repository.

Usable

Usable software meets the needs of the communities it has been developed for; ideally, meeting those needs also matches the goals of the project.

See Sustainable Code for guidelines regarding the usability of code.

Software has a clear, understandable interface.

For software with a visual interface, such as a desktop GUI, plugin/extension or web/mobile application, the interface provides a clean and clear layout.

For desktop GUIs, this often means compliance with recommended design patterns for an operating system (Apple OS X Human Interface Guidelines) or for the interface toolkit of a programming language (Tkinter for Python or Swing for Java).

For plugins or extensions, the interface components, such as forms and menus, conform to the design of the parent GUI. If the parent platform provides guidelines for extensions or plugins, the extension follows those guidelines (QGIS: Developing Python Plugins).

For web applications, the interface design depends on the goals of the project. Web interface design is not as constrained as desktop GUIs, nor do we want to impose visual design preferences here. However, for research groups with limited resources, consider using an existing framework, whether an extensible content delivery framework (ie, CMS or portal) or a more generic front-end framework (Bootstrap or Foundation).

When assessing web applications, visual design is important only as it helps or hinders the functionality of the site. Unless a stated research goal involves developing a new interaction method, use common interface designs.

Provide clear labels and easily identifiable means of accessing help. Apply styles consistently across the application. Provide a style guide for visual styles, for example the USDS 18F Web Design Standards (advanced option and more for those projects developed for adoption).

For web applications or any software accepting user inputs, Unicode is well-supported throughout the system.

We include this as an example of a what may appear to be a fairly specific detail but one that can, in web applications and services, indicate issues in the development process and, more importantly, in the understanding of the research group to understand and meet the needs of their intended audience. It is, in effect, an implementation canary for potential adopters and reusers.

For web applications, the application implements responsive design, i.e. support for multiple screen sizes.

Responsive design adjusts an application's layout based on the screen size (mobile, tablet, desktop) without requiring support for multiple application versions.

For web applications, the application adheres to progressive enhancement practices.

Progressive enhancement adapts the supported features of the application based on the detected device. In practice, this means adjusting functionality based on slow connections, prevalence of feature phones (instead of smartphones) in the community being addressed, or other concerns related to the execution of Javascript code.

This depends on the stated goals of the project and requires an understanding of the community needs before implementation.

For web applications, the interface follows established guidelines for web accessibility.

Web accessibility guidelines provide information to help develop web sites and applications that are open and inclusive for people with a range of abilities. Visit the W3C's Web Accessibility Initiative (WCAG, WAI-ARIA) and the U.S. GSA's Section 508 guidelines.

These guidelines discuss structural concerns, for example semantic and well-structured HTML to support screen readers, and navigation or interaction requirements, such as keyboard bindings. We consider semantic and well-structured HTML in a similar context to that described in the Sustainable Code section, namely that it is beneficial to meet guidelines related to clean, well-structured code and is not considered an onerous or additional burden.

For web applications supporting dynamic mapping functions, try to use mapping libraries that include support for accessibility whenever possible. We note that support for colorblind users, represented only in data color schemes only, does not meet the minimum expectations for supporting web accessibility.

Project websites should also follow web accessibility guidelines. This may not be in the research group's control, given institutional requirements or existing cyberinfrastructure; however, we encourage research groups to consider web accessibility support as an important adoption criterion should the choice of framework be available.

For web applications, support current browser versions.

Front-end frameworks and libraries have greatly improved cross-browser support, but it's still important to ensure that the web application functions as intended across any and all browsers supported by the project. Any modern Javascript framework, whether the more generic JQuery or interface frameworks like React, Backbone, or Ember, will provide cross-browser support.

Sometimes, project goals and community needs may require supporting older browser versions. In this case, state the supported versions and take steps to meet those requirements safely (noting vulnerabilities in older browser versions, etc). Progressive enhancement principles can be considered to provide equivalent functionality for users limited to older browser versions and users using current versions.

The opposite may also hold true, where a research goal is to develop a proof of concept or prototype application using cutting edge functionality that may only be supported in certain browsers. In this case, progressive enhancement is not necessary but the project documentation and web application should indicate that limited support clearly.

For a web service, the API is RESTful or follows a known community standard or specification.

API design is a usability concern. As with other areas, an API should be clear and legible, with meaningful names and using the RESTful design pattern (Fielding 2000) or HTTP API pattern. See Interoperable for guidance on implementing community standards, understanding that those standards may not be RESTful or conform to the conceptual models found in Web API documentation platforms. For the purposes of assessment, correct implementation of the community standard outweighs any RESTful recommendations.

For collaboration tools, ensure that the interface design does not encourage sensitive information leaks.

Carefully consider interface design choices related to an individual's permissions and access selections. These characteristics, related to layout, label text and other details, are included in general web interface guidelines but they are important to highlight here due to privacy concerns. Permissions and access options should not result in unexpected behavior from the viewpoint of the individual using the platform.

Software is performant and stable.

Documentation describes the current development status.

For open or public codebases, this gives potential adopters an understanding of where the project is in its lifecycle. This can be as simple as adding a note stating that the source code is under very active development (high rate of commits, known breaking bugs remain, or rough interface), the source code is not being maintained or is in a reasonably stable in-between state.

For code or software that will not be maintained or has reached end-of-life, update the README and other documentation to indicate this status. This can be text or with a badge (see @potch No Maintenance Intended).

Please refer to Governed for managing contributions.

For web applications or web services, the documentation or project website includes service level qualities.

Simply put, provide potential adopters of the public-facing system information about the reliability and performance of that system.

For web services, include limitations for requests (rate limiting) and other policies related to automated access. This gives adopters guidelines for developing clients safely, i.e. they understand the system limitations and can work within them, and gives adopters of the system some indication of system limitations when taking the deployment requirements into

consideration. It also gives you, as the maintainer of a system, a clear process for dealing with violations of these limits and an understanding of the system performance for planning future work.

For web applications and services, use an appropriate method for managing human and automated access. Consider the service level qualities and project goals with the understanding that potential adopters may be approaching your services with very different, but valid, use cases. This is an inherent feature of publishing to the open web! Use robots.txt, rate limiting via server configuration, access token policies or other connection configuration options.

For web applications and services, provide information regarding uptime, methods for receiving outage notifications, expected response times for responses to support requests.

These service level qualities are dependent on effective monitoring. Providing service level quality information is not, in itself, enough to build and maintain trust in a system. See Monitoring below.

For web applications and web services, systems are monitored.

Monitor your systems for security and infrastructure (anomalous activity), for service level quality metrics and performance (uptime, service response times), for sustainability metrics (growth, retention, service usage). Note that these are different concepts and use different platforms, for example, Nagios for infrastructure, the Elastic Stack (ElasticSearch, Logstash and Kibana) for log analytics, and Google Analytics for application events. You may also be able to take advantage of solutions provided by your cloud resource provider.

The level of monitoring depends on where in the project lifecycle you are and what expectations you have for adoption and use. For production (beta) or operational systems, monitoring is not considered “nice to have” as access issues, performance issues or other issues affecting the use of the system has a negative impact on the trust of the system—instability or persistent performance problems (Brutlag, 2009) drives people and potential adopters away.

For high performance computing, GPU or other cluster-based software, the project provides basic benchmarking statistics.

For algorithm development or similar code, provide benchmarks with comparisons to similar algorithms. Describe the benchmarking process. We note that this activity can be the topic of a research paper in certain domains. References to the paper and related research products can be provided instead of a detailed write-up in the documentation. Cite it appropriately (see Publication and Citation).

Documented

Documentation is important across project areas and audiences. Research code and software has a higher documentation burden. This is generally seen in two areas: the temporary and variable nature of the workforce and the expectations of reproducibility and contributions to the larger research community. Both can be mitigated by improved source code and project documentation where an effective knowledge base protects the project against unexpected churn and aids in onboarding new participants. The ideal of self-documenting code must be considered against the realities of the research development workforce and conditions.

When in doubt, a good rule of thumb is to document things that can't be found in any dependency or framework documentation and that might appear counter-intuitive to another developer, even if it's you in six months.

Source code is documented.

Comment blocks are included in the source code. Minimum documentation describes methods and the input and output parameters. In addition, we recommend code comments to describe sections that may not be clear, for example, bug patches that appear counter-intuitive or business decisions that shouldn't be modified without clear external reasons.

For code notebooks as a research product, include text blocks to describe the workflow and any outputs.

Source code comments use a standard comment style, related to the selected style guide or language or related to a document generation tool.

As with the code itself, the comment style should follow the chosen style guide or language conventions.

API documentation is automatically generated.

In this case, the API can refer to the methods and properties of the codebase or a web API. Document generators, such as Sphinx (Python) or Codox (Clojure), expect explicitly formatted comment blocks within the code or as standalone files.

For web APIs, especially RESTful APIs, use a documentation specification geared to that such as Swagger (or Open API Initiative), RAML or API Blueprint. In some cases, document generation tools may support the API specification so check that documentation for possible integrations. Keep in mind that public web API documentation does not replace source code documentation as one serves adoption for client development and reuse and the other supports further contributions.

The documentation describes how to automatically generate the documentation.

When using a documentation generator, describe the tools and process for generating the documents.

For traditionally open sourced projects, this description is related to other contribution guidelines and offers contribution options beyond code.

API Documentation is versioned.

Like the code, the generated API documentation is versioned and maintained in a VCS. API documentation can be hosted under the project website or an external documentation host such as Read the Docs. If possible, link the documentation version with a code release (for reproducibility, preservation and community support).

The codebase includes documentation to support adoption and reuse.

This documentation is related to those information items that describe how to deploy the code or software. Reuse and adoption here refers to the developer community but it can also benefit preservation needs.

The source code repository is documented using plain text file formats.

Current practice is to provide certain documentation through Markdown or reStructuredText. Expected files include a project-level README, LICENSE, and INSTALL and may include a code of conduct, CONTRIBUTING, AUTHOR or other files common to the GNU recommended file structure.

The build procedure or installation process is described.

Describe the build procedure if building from the source code. Describe the installation process if providing binaries. Be sure to include any dependency requirements, including version information. Also include system requirements, including version information.

Often, build/installation details are provided in the source code's README or INSTALL files.

The documentation describes how to run simple examples.

Select a representative task and provide configuration and parameters or input data to execute the required code. Include a brief description of the task. Provide the expected outcome or where to locate any generated outputs as appropriate. This documentations can also be provided as code notebooks.

See Software is documented > Any Documentation for information regarding similar documentation with quick start guides.

The documentation describes how to execute the test suite.

The documentation describes the tool used for testing and how to execute tests. Often this is included as a post-install or post-build step to demonstrate that that process was successful. For automated deployments where the configurations are provided, safely, with the code base, indicate that tests are automatically run.

The versioning scheme is described.

For source code in a VCS and with a defined versioning scheme, provide the versioning scheme used in the documentation. This is most likely included in the CONTRIBUTING file.

Release notes or changelogs are provided for each release.

Clear and complete release notes provide adopters some understanding of how a major or minor release may affect their own systems and workflows.

For preservation and archiving concerns, provide these items in the code repository or other documentation. Consider a future scenario in which a potential issue with a generated dataset comes up and, while the code is accessible (locatable and viewable), it may not be possible to run. Release notes in these situations provide avenues for understanding when and where the issue may have occurred.

Software is documented.

Here we are discussing documentation guidelines for higher level guides for user or developer documentation, such as how-to guides, quick start guides or tutorials.

Certain code or software types, like APIs and libraries/modules, lend themselves to integration into code notebooks to provide interactive examples and tutorials. Notebooks can be a valuable publication option for some of the guidelines below.

Any Documentation

Any restrictions or constraints for a particular method, request, or process is included in that item’s documentation. This includes file size limits or rate limiting for web APIs.

Quick start guides are provided to provide any user an introduction to the code or software. The selected example should reflect a normal interaction, provide a description of common possible errors and how to respond to each, and a successful outcome. If warranted, provide a simple dataset for this demonstration.

Developer Documentation
Note that developer documentation can simply be API documentation for modules/libraries, command line interfaces, or web APIs.

For command line tools and web APIs, all input and output parameters for each command or request are defined, including default values. Every command or request is likewise described.

User documentation

This documentation is aimed at users of research software, whether a desktop GUI, plugin/extension, or web application.

The software documentation is complete. It describes every possible action with annotated screenshots of interfaces and step-by-step processes. This includes possible errors and recovery options.

The images and method or parameter names are up-to-date, i.e. they match what the user sees in the software.

Provide this documentation in a manner that is searchable through a web search engine (preferred) for both desktop GUIs and web applications. For desktop GUIs, provide documentation in a manner supported by the application or the operating system.

Code or software requirements are documented.

Formal specifications are not always required but, for those projects that have a clear need for formal correctness (algorithm implementations, data processing, etc), provide the specifications used for implementation. If not provided as a published document, ensure that the expectations are described in the appropriate tests (see Tested for related).

Project is documented.

This is the highest level documentation and often the first information made public. It can, in some ways, be considered pre-registration. This documentation is maintained on a project website separate from the code.

This website should include the grant award numbers and agencies, initial project milestones, a description of the project and the expected software or code deliverables and a listing of project team members.

As the project continues, include reports, articles, conference presentations or other materials on the site. If possible, include testimonials from satisfied users.

Include links to the project source code or software installers and documentation.

Provide copyright information, commonly in the footer of the project site. Consider licensing the project web site content through Creative Commons or under the license of the department or organization.

Secure

Any code or software should be developed following the current recommended practices for the systems being used. Security must be considered across functional areas, from system administration and backend development to frontend development and user access. Consider using tools such as Coverity, SourceClear or QuickCheck variants.

The principles outlined below are not inclusive and are meant to highlight certain areas of concern and point to resources to provide more detailed information. Security through obscurity is not a viable method.

Source code, including automation configurations and scripts, have been sanitized for public release.

Whether you release source code at the end of the funded project for preservation or use a version control platform during development, it is important to ensure that sensitive information, such as access credentials for cloud services or database logins, are not ever publicly available. This starts before the first commit to a repository — the information remains in the history without additional effort. Take steps to ensure that you are not leaking credentials through your version control histories, through backups, through any automation configuration, or any other potential avenue.

For web applications and services, software follows industry recommendations for secure practices.

It is beyond the scope of this document to cover all aspects of web application or web service security. However, we do strongly recommend following those put forth by the Center for Trustworthy Scientific Cyberinfrastructure and The Open Web Application Security Project (OWASP) as well as taking advantage of the Vendor Security Assessment Questionnaire (Google VSAQ).

Some considerations:

- Enable HTTPS on every web application or service (note, Google uses HTTPS for search results ranking (Google Security Blog 2015) and, as such, is recommended beyond security requirements).
- Sanitize web form inputs to prevent cross-site scripting (XSS) or SQL injection attacks.
- Protect against Cross-site Request Forgery (CSRF) attacks.
- Sanitize responses related to errors and exceptions, i.e. do not return stack traces or similar from any public-facing platform.

For HTTPS, check with your home institution or department for certificate providers in use (InCommon is a common university option). Small research groups may also want to consider an open certificate authority such as Let's Encrypt.

For any code or software system, follow the recommended security practices for each system component.

For any system or framework or similar dependency used, follow the recommended security practices for that system for deployment and for ongoing management. Consider databases, web frameworks, authentication systems (such as OAuth2), web servers, cloud infrastructure access.

Keep systems and dependencies up-to-date.

Security vulnerabilities can occur in any part of the implementation stack. For software or code currently being maintained under a grant, take note of vulnerability notifications and update dependencies and code accordingly. Indicate to adopters when a version contains a security patch. Indicate in the release notes which dependencies require updating—do not force a potentially breaking update without some warning.

See Governed for information regarding sunsetting web applications or services. If resources are not available to upgrade the application to manage vulnerabilities in the dependencies, including the programming language version, operating system or web server version, we suggest archiving the system.

If the project is releasing runnable containers, we encourage you to provide upgrade paths for the container or the application it runs. If nothing else, refrain from presenting a runnable container as one-time task and provide estimates for system maintenance for potential adopters, particularly if presenting the product as a solution for those without dedicated technical resources.

Collaborative platforms include tests for permissions and integrations.

Include extensive tests for permissions conflicts in the test suite, particularly those that might make public content the person or group marked as private. Consider the interactions of all the content privacy settings. Consider carefully any new setting that might be added to the system, as potential conflicts and errors grow accordingly.

Indicate in the documentation, at a very high level, that this kind of testing is performed. This is not dissimilar to a service level quality in that the statement alone is without merit. It must be backed up with valid tests of the system.

Include additional testing for integrations, or the use of third party APIs such as Google Docs or GitHub. Only ask for permissions from the third party platforms that are required for use in the collaboration platform. Follow any recommended practices for securely connecting to those services.

Containers follow recommended practices for security.

Container systems, such as Docker (Docker 2015) or the Linux Container project, are not secure by default. For any public-facing system or any system deployed to a cloud provider, follow the recommended security practices for that container platform. This applies to containers deployed and maintained by the project team and those built as a project deliverable or with the intent to share ("Project in a Box" situations).

See Sharable for additional guidelines on automation.

Sharable

The guidelines promote practices for adoption and reusability of the code or software's codebase whether it was born-open or publicly released at a later time. These are intended for code released for publication and reproducibility as well as code released as an active open source project.

Source code is licensed.

Any code or software product, whether released for publication/preservation only or for ongoing development activities, should include a license. ESIP cannot provide recommendations for which license to apply but we encourage you to refer to your institution's intellectual property policies with consideration for your funding agency's policies or any policies put forth in the solicitation of the award. Unlicensed code is unusable code—you haven't provided a potential adopter any information regarding permissions or limitations on its use so the assumption is that an adopter is not permitted to use or modify that source code. Unusable code is unsustainable code in the context of a funder's return on investment.

License selection is affected by the intended use of the software and the licenses of the dependencies you use. For this reason, we encourage you to consider your license options early in the development process and to take that choice into consideration throughout implementation.

For projects allowing outside contributions, you may also want to consider Contributor License Agreements. A CLA states that the contributor agrees to contribute and that the contributor grants rights to the project so that the project can use the contribution in distributions and that the contributor can't revoke that right.

For a description of open source licenses, please visit the Open Source Initiative (OSI) or Choose a License. We strongly encourage the adoption of an OSI-approved license whenever possible.

For source code published for preservation, at a minimum, the selected license should allow someone to rerun your code for the purposes of reproducibility or replication.

One final note on software licenses in academic workplaces—intellectual property restrictions are related to your job status (faculty, student, staff). Again, ESIP cannot provide legal advice; however, we encourage research groups to familiarize themselves with these issues and agree to internal procedures for handling intellectual property issues (licensing or copyright). For a related discussion about research data, please refer to the RDA-CODATA Legal Interoperability for Research Data report (Uhlir and Clement, eds, 2016).

See Documented for related.

Source code includes configurations for automated systems.

Automation can include test tools, build tools or packagers, provisioning and/or orchestration tools (Chef, Vagrant, Puppet), or continuous integration systems.

For tests, this includes providing verified inputs and outputs as necessary.

Follow the tool's security practices for preventing credential leaks or other vulnerability concerns.

Project name is discoverable.

When discussing project names, ensure that the name or acronym is unique within a domain, at a minimum, and that it is searchable as is or with some additional context. Common words or highly similar acronyms/names can make a project difficult or impossible to find through a search engine.

Governed

Governance guidelines refer to the processes around active development and maintenance as well as communications related to those.

Project governance guidelines can be applied without requiring a project to be traditionally open sourced. Indeed, many of the guidelines are simply explicit statements of process. We also note that it is not uncommon to consider a project "internal open source," i.e. it's managed in a way similar to a public OS project without being public. As with Documented, these guidelines are also meant to mitigate some of the issues related to the nature of our staff resources. These,

then, can be applied to internal teams, larger collaborative groups and to traditionally open sourced projects.

Contribution policies are provided.

This example template for contribution policies covers many of the recommendations and provides examples of existing documents.

For any project, describe the workflow used by the project for contributions to the source code (or other versioned items).

This assumes that the source code is maintained in a VCS. Choose a workflow for contributions to the code repository. For git-based systems, this most likely involves a branch-based workflow such as GitHub Flow.

For local teams (or even single contributors), select a workflow that is workable for all of the team. For public processes, select a commonly used workflow for your VCS of choice and your review process.

Part of this workflow definition should include expectations about testing and documentation related to the contribution.

For any project, describe commit message, pull request, and issue preferences.

If your workflow or automation processes support it (or for legibility), define commit message conventions and branch naming conventions. Set expectations for issue content. Define what is expected for bug reports or new features.

See Documented for related.

For any project, define the code review process.

This describes the process for accepting a contribution into the repository. It applies to any development team and sets the roles and responsibilities of the reviewers and an expected timeframe for response.

Note that this response time is not dissimilar to that described for service level qualities as outstanding pull requests can be a warning flag for potential contributors and for sustainability concerns.

For open source projects, provide a code of conduct or state clear expectations of behavior and communication.

This is a sign of openness to potential contributors. A common structure is the Contributor Covenant. Consider the statements made in any code of conduct you develop or reuse and ensure that you agree with and will enforce the statements in the document.

For open source projects, provide guidelines about what contributions will or will not be accepted.

This includes contributions not related to source code, contributions that don't fit within stated milestones or near-term feature implementation goals, or contributions that don't comply with other stated expectations. A common example of what won't be accepted are contributions based solely on style guide modifications (tabs changed to spaces, for example).

Development activities are transparent.

The section involves those activities related to active development and the communication processes in places for development and support.

Development activities are managed through an issue tracker or similar software.

For internal teams, this can be the tracker provided through an external or self-hosted VCS (GitHub, BitBucket, GitLab), through a project management tool (Pivotal Tracker) or a TODO manager (Trello).

For traditionally open sourced projects, the issue tracker provided through the external VCS is preferred, although any public tracker is acceptable. This allows adopters to submit bug reports or external contributors to submit pull requests for patches and new features.

Project provides support mechanism(s).

For project support beyond a specific code base (a project can support multiple codebases), provide a dedicated support email or contact form.

For security concerns, a support contact option is recommended over the public issue tracker. This provides a way for someone to describe a potential vulnerability privately, giving the project team time to respond. Include this contact information and preference regarding vulnerability concerns in your CONTRIBUTING documentation or other appropriate documentation location. We strongly encourage this for collaborative platforms.

Other communication methods include Twitter, listservs or mailing lists and IRC.

Any public-facing support mechanism should be actively used by project team members. This does not mean enforced chatter; it means that support requests are addressed in a timely manner.

In the event that a web application or service reaches the end of its lifespan, project members use these support mechanisms and other public-facing avenues (the website and/or code repository README, for example) to notify the community of adopters of service sunsetting and expectations for the cessation of services. Include information about what will happen with

crowdsourced data, account information, site access and other relevant details based on the nature of the project.

Related: service level qualities.

Project provides a development roadmap

A roadmap document describes the philosophical underpinnings of the project and its future technical path. This gives potential contributors and other adopters insight into how well the project may fit their needs longer term and the types of features or enhancements that are likely to be accepted. This is higher level than the contribution guidelines.

Code as Research Products

These guidelines discuss activities important to the research community for preservation, reproducibility, provenance and credit. When discussing accessibility in this section, we are referring to the ability of an individual to locate a project and, more importantly, its software or code products.

The Geoscience Paper of the Future (Gil et. al 2016) outlines recommended practices and the relationships of research code and software within the larger scholarly ecosystem. Academic journals may also provide guidance on their expectations for research code and software related to accepted papers. Support journals that are signatories of or have implemented the Transparency and Openness Promotion Guidelines (Alter et al. 2016).

The ongoing shift towards considering code and software as research products in their own right is a rapidly changing environment. The guidance below starts with currently actionable steps, defaulting to capturing the information in the code repository at least.

Publication and Citation

Software, as binaries and source code, are published to a sustainable third-party repository.

This criteria can be met for projects hosting source code in an external VCS platform during development and simply leaving those publicly accessible at the end of the project.

Some research communities have developed domain-specific software and code registries for publication.

Other options include depositing software/code and related documentation with your institutional repository.

Documentation includes citation details.

As noted earlier regarding other metrics for assessing the sustainability (or viability) of a research software package includes impact related to citations. Provide a suggested citation in the README of the source code and the project website.

If publishing a software paper to a dedicated research software journal such as SoftwareX, Journal of Open Research Software, or The Journal of Open Source Software, provide the citation in the README and on the project website.

We encourage the adoption of the Software Citation Principles (Smith et al. 2016) outlined by the Force11 Software Citation Working Group, noting that currently we do not have a normalized citation structure for code. In some areas, community practices exist; for those areas without, the F11 document provides references for currently used options. Providing citation details is only one part of fully realizing the value of software citations. Citing others' code and software you use for your research activities, for datasets or analyses, is part of being a good research citizen.

Preservation/Archiving

As noted in Publication and Citation, depositing a stable or final version in an external repository is the minimum option. Meeting most, if not all, of the documentation guidelines and ensuring that the documentation is deposited with the source code provides a more useful archive. If using certain version control systems, such as GitHub, export and archive all of the metadata related to the repository (wikis, issues, etc).

Containerization is being explored as another archive option (DASPOS). Here, the software is provided in a runnable container, preserving the system and dependencies. In some cases, this kind of preservation is not possible due to license constraints related to redistribution (for proprietary software, for example).

Realistically, black box archiving or a container providing only installed or built software is not as future-proof as providing both the source code and documentation archive and a containerized version. Runnable archives address reproducibility and replicability concerns; they do not address other aspects of code that are of value. Code, maintained in its basic text format, is valuable for education and for other research activities as data.

Credit

Authorship and credit for software is overlooked in traditional scholarly spaces. Citation is one method for addressing that. Projects such as depsy, CASRAI's CRediT, OntoSoft or Dagstuhl's Software Credit Ontology offer new opportunities for understanding contributions and transitive credit. Often these options are limited to certain programming languages, VCS platforms or

package indexes and may require implementation through the software publication platforms. To ensure that the authorship information is available regardless of these current limitations, include an AUTHOR file in the code repository. Indicate major contributions, important contributions to code and other components.

Provenance

Several of the guidelines outlined throughout this section support provenance efforts (consider publication and versioning, for example). Rather than reiterate those details, we discuss two higher level guidelines here.

For code/software creating or serving data, integrate provenance trace generation into the process. This might mean developing the product in an existing workflow management system (VisTrails, Taverna or Kepler), provenance capture integrated into code notebooks as workflows (Ma *et al*. 2017, ECO-OP 2015), or integrating a provenance capture package into the codebase (RDataTracker or recipy). Different domains and research activities may implement traces at different abstraction levels. Whenever possible, we encourage generating original traces with a high level of detail to better support integration across activities as provenance activities mature.

Once provenance traces are implemented within a system, publish those traces through a system supported by the research group, in a domain-specific provenance store, or in a public provenance store (ProvStore). Traces, regardless of publication method, should be provided using the W3C PROV specification with an ontology reflecting the abstraction model used for generation.

Reproducibility/Replicability

Rather than providing any additional guidelines specific to reproducibility and replicability, we simply note that following guidelines for clean code, automation, documentation and publication will provide a solid working baseline for meeting the requirements for a specific publication or community practices.

Progression, Sustainability and Reusability/Adoption

In this section, we will discuss how we can best place the concepts outlined in the guidelines into three larger areas of concern for research code and software, namely progression, sustainability, and reusability. We will mostly be discussing these areas in terms of publicly released research software as such products are often the main area of concern, but this discussion should not be taken as discounting the importance of research code.

Before we discuss each topic in detail, we will pause to reflect on some of the gaps in our collective knowledge of the larger geoscience and research development communities. We do

not have an understanding of the full breadth of research code/software activities, nor of how those activities have been sustained over time, but we do have a wealth of anecdotal evidence based on our own project activities, our interactions with collaborators in other research groups, and our participation at conferences and meetings. From these local viewpoints, we know that we can't form a realistic picture of the overall landscape and, without that, it will be difficult to provide meaningful measures for sustainability and reuse. We risk artificially limiting the discussion to a subset of situations, or providing recommendations based on survivorship bias and false equivalencies.

As we have approached this process of developing guidelines, we were struck by the ways in which the limited representations of code and software in research prevented many from seeing how their activities are situated in this ecosystem. Again, it is worthwhile to consider, even with largely imperfect knowledge, how our approaches to sustainability, progression and reusability can promote equitable assessment processes and actively support those researchers who are developing and managing code.

With that in mind, we offer a number of outstanding questions regarding our research code/software activities.

- How often are projects developing systems from scratch?
    - How do we define "systems from scratch"?
    - Are there material differences in how we assess and adopt systems?
    - How do we support incremental improvements in cyberinfrastructure to sustain research activities as well as supporting innovation?
- What are the distributions for the types of research code/software projects currently active?
- How can we evaluate adoption and reuse for…
    - Library modules/packages?
    - Standalone but integratable platforms/frameworks?
- How do we identify and support projects that…
    - have successfully supported a research activity within their community?
    - have transitioned into a successful general product to support research activities across different domains and support activities in private sector efforts?
- Do our funding structures support alternative means of sustaining valuable but niche research software?
    - Can we apply specific crowdfunding support to other projects in our budgets, as one time payments or subscription fees?
    - Can our organizations accept funds from crowdfunding services, as one time payments or subscription fees?

It is clear that there is no one-size-fits-all solution to any sustainability or assessment question given the variations in projects and in research goals. Our discussion here is a renewed call to explore viable and equitable solutions for both concerns.

Progression

As described earlier in this document, the initial evaluation efforts relied on NASA's Technology Readiness Levels to describe progression and maturity. In reframing the effort to focus more on education, we determined that the TRL structure did not lend itself well to this approach. Born out of a particular management style, the TRL's rigidity and linearity are out of sync with common development management practices today, and with practices modified for groups with limited resources.

We therefore suggest a structure that is different from the TRL approach, one that will better address each of the following three main concerns:

1. The need to align guidelines and development activities within the scope of a grant, that partially or fully funds the activities;
2. The need to align long-term projects with continued grant support;
3. The need to apply context-driven models of maturity appropriate to different project stages.

The first concern acknowledges that many of the concepts outlined in the guidelines will be implemented during the active development cycle of a grant, and that these guidelines may be applied unevenly. For example, a codebase may be configured for continuous integration early on in the development process but never provide high-level documentation at any point during the cycle.

The second concern acknowledges the complexities of project lifecycles, and that different stages in that lifecycle may map to different solicitation requirements (or expectations). Transitions between different phases may also require different expectations.

The third concern acknowledges the importance of context-driven assessments of maturity levels, which has been noted previously in this document in relation to different kinds of code/software project types. Here, we are highlighting the need to address maturity expectations at different project phases and with respect to the different products from research code and software. We will be describing these phases in terms of four characteristics: intent, utility, correctness and criticality.

Table 3. Definition of progression characteristics.

Characteristic	**Description**	**Values**
Intent	Describes the nature of the code or software as internally or externally focused. Internally focused describes much that we have called research code, but is not limited to that definition.	Internal: product is very specific to the project, heavily integrated with a legacy system or otherwise not intended to be released as

		reusable code but can be published.
		External: product is a stated deliverable or is intended for adoption/reuse outside of the research group.
Utility	Describes the function of the code or software in relation to the project's outputs. Is the code itself a stated deliverable (primary) or does it generate a deliverable (secondary)?	Primary: product is one of the project's stated deliverables or goals.
		Secondary: product is used to generate or support an output of the project but is not, itself, a stated deliverable or goal.
Correctness	Describes the importance of the correctness of the code's output, particularly as it relates to other project outputs. Correctness is not necessarily an indication of bug-free code, more that the code meets the provided specifications.	Low: limited need for formal specifications.This does not preclude the inclusion of testing.
		Medium: formal specifications and testing, while not required for research goals, are useful for infusion or community goals.
		High: research goal requires demonstration of correct implementation through formal specifications and testing.
Criticality	Describes the level of operational support, in part related to project goals and type. Criticality pertains to expectations for support as well as actual support, depending on the project.	Low: limited or no need for ongoing support or maintenance of the code product or availability (if hosted web application or service).
		Medium: product may receive support at varying intervals but little ongoing support.
		High: product is expected to be available for internal or

		external users according to the service level qualities or release cycles, with continued maintenance and support. (This is the equivalent of “operational” in many respects.)

We encounter difficulties in defining progressions when there is a tension between more traditional understandings of science code and the code that is more likely to come out of research grants in the broader ecosystem. Realistically, production code is anything that supports a public outcome, whether that outcome is a generated data product, a tool or framework to be adopted, or a web application. The distinction between internal and external support, categorized above as “Intent,” matters to the development of training and assessment tools more than it does to any functional differences with industry versus research software.

Table 4 outlines a four phase progression model for capturing common patterns in research code and software using our four characteristics. The model is not meant to be comprehensive and, indeed, we can imagine other situations where the assigned values may be quite different. Instead, the table is meant to provide common ground for building relationships between process and project maturity as indicated by the guidelines, with the kinds of activities pursued as research development, and as understood from grant solicitations.

Table 4. Context-driven progression.

Phase	**Intent**	**Utility**	**Correctness**	**Criticality**
Proof of Concept	Internal/External	Primary	Low	Low
Prototype	External	Primary	> Medium	> Medium
Research Production	Internal	Secondary	High	Low*
	Internal	Primary	High	High
Production/Operations	External	Primary	> High	> High

*The generated output may be critical but the code, and the need to continue to support that code, may not be.

Proofs of concept may be released publicly but with limited expectations of community support. One can consider the public activity to be demonstrations or collaborator integration. As such, active operational support is not expected. Correctness here may be limited to core functionality.

Prototypes have higher expectations than proofs of concept in terms of support (criticality) and correctness, as these projects are likely to have higher expectations for adoption. Prototypes are not, however, considered operational products.

Research Production encompasses those projects or codes that are internally focused, i.e. not intended for adoption or reuse on their own, but that have generated primary research products. Criticality here depends on whether the need (and support) for those products is ongoing. Correctness of the code is important for generating high quality primary products, whether those are data or analyses.

Production or operational systems are public-facing products with priority placed on both criticality and correctness. These systems have been adopted as reusable components, standalone software or web applications within one or more communities.

This progression structure more explicitly captures the realities of research development activities. Using this structure, we can consider innovative solutions with the understanding that additional resources will be required to operationalize those early-stage products showing promise. We can also more usefully consider a long-running software and its need for dedicated, ongoing maintenance to ensure continued successful research outcomes for those who rely on it. And we can consider the utility of other codes that do not become part of any dedicated cyberinfrastructure, but that instead produce data and analyses.

When we describe the different progression phases as context-driven, we are creating a framework that gives us a flexible and functional method of applying the guidelines in practice, for both training and assessment activities. For training and education, we can now design practical guides for meeting community expectations for divergent goals (with the understanding that code is code, regardless). And for assessment, we can set expectations for our research products early in the funding cycle, with a common understanding to negotiate that space according to the requirements of the solicitation and the relationship of the new research products to existing systems.

If nothing else, removing language specific to development process or project maturity from the progression phases leaves us free to adapt code practices to meet new technical requirements and new methodologies while providing funders or other researchers a means of assessing improvements across research communities and research development activities.

Development process maturity and project maturity is integrated in this progression model based on the specific assessment requirements of an implementation. That being said, the characteristics do assume certain levels of maturity for both aspects where we expect production systems with high criticality and correctness values to reflect higher maturity levels.

While the structure of the guidelines implies certain paths to more mature practices within the sections, we have not developed an explicit model here. We will discuss the application of these guidelines and the progression model further in Using this guidance for assessment.

Sustainability

The medium- to long-term sustainability of research software projects is often tied to a traditional open source model, and the guidance provided here draws heavily on those kinds of governance and contribution models. That being said, research software and code have additional expectations as part of the larger scholarly and research community. Traditional open source might be a big hammer for research projects that may not actually use nails. It is of course appropriate and necessary to understand the context in which open source activities and research are each undertaken, and adapt these processes and potential funding avenues to reflect different needs.

To support the different expectations and development activities, we propose a three-tiered approach:

- **Public Source:** meets the publication and provenance requirements. Code artifacts are public with no maintenance expectations.
- **Research Open Source:** similar to internal open source in that open source practices are used but with no expectations of outside contributions due to community structure,research goals, or research group governance. Critical to support community adoption of the product.
- **Open Source:** traditional open source with expectations of external contributors.

Public Source does not exclude Research Open Source or Open Source and it is possible to meet the Public Source requirements through either Research Open Source or Open Source governance. The key difference is the publication requirement.

Within these three categories, we immediately note the difference between Public Source and the two Open Source categories. Sustainability for Public Source products is most often the responsibility of some external platform, like a domain-specific software repository, a hosted VCS or an institutional archive. The sustainability of the projects developing such products is certainly of concern, but largely out of scope of this discussion.

We can, based on our stakeholder expectations, speak to sustainability for Research Open Source and Open Source activities. We can suggest activities to support the adoption and growth of a piece of research software, as would be required for its longer term viability, but there are no guarantees of success. This is as true of the larger Open Source community (Eghbal 2016) as it is for research software. Realistically, most projects are unlikely to draw from a large pool of potential contributors, which limits their ability to diversify the contributor base beyond the original research group. This limitation does not mean that the research goals will

not be met, nor that the project is not valuable to the audience it is aimed at; it simply means that we cannot expect high levels of external contributions to sustain future development efforts.

We are faced again with a situation that does not provide a single solution. Furthermore, without a comprehensive understanding of the research development landscape, it is irresponsible to suggest that there is one. But we can consider sustainability in three broad areas, with an eye towards a healthy research development culture and broader impacts.

Sustainability based on potential

For projects in early phases, e.g. proofs of concept or prototypes, sustainability may be more accurately called opportunity, i.e. whether the project has met its research goals and whether it is being managed in such a way to indicate future success. As we described in Progression, one of the goals of these projects is to demonstrate the viability of an implementation. Resources might not be dedicated to supporting activities related to adoption; or, in other cases, adoption is a goal but whether any resources can be used to support adoption will depend on timing.

Sustainability here can mean either additional funding to operationalize the software, or to provide time to allow for community adoption and reuse.

Sustainability of broadly applicable cyberinfrastructure

It is telling that, in conversations about research code or software, we speak of certain open source projects as models for adoption. The kinds of projects we talk about, however, are often not good comparisons. We point to data formats, database systems, visualization libraries or analysis packages, these more general products, as exemplars for any research code/software open source project. These are products that can be adopted outside the research community.

Discussions around sustainability and sustainability plans are also where open source expectations around community management and engagement play a larger role. Outreach for a project requires resources and effort. The guidelines describe the technologies and platform needs to support community activities; the mere existence of those technologies won't grow a viable community.

Alternative funding models in the open source community include crowdfunding (one time payments or subscription), foundation funding, or corporate patronage (Eghbal 2016). Of course, pursuing funding outside of federal research grants is also resource-intensive.

Sustainability of research development

What about development that cannot be broadly adopted across research communities or beyond? Here, we are talking about those projects that develop domain-specific products such as models, those that develop and maintain citizen science applications for ongoing data collection, or those that provide infrastructure for data or analysis.

In some cases, a high priority goal is to have the product integrated into an agency's operational systems. Reaching this goal however does not guarantee that there will be ongoing maintenance support.

The alternative funding models listed for open source projects may be applicable here as well. Anecdotal stories from the open source community, particularly stories involving patronage (monies from a foundation, whether corporate or otherwise), describe limitations on that funding, suggesting that these models may not be effective long-term solutions (Eghbal 2016). If the patronage model revolves around funding specific features that are required by the patron, one must consider the larger research goals of the project, both in terms of the available resources for the maintenance of those features, and in terms of the philosophical underpinnings of the project.

It is, then, a question for our research communities as to how we develop sustainable funding models for projects that cannot make the transition from research product to a more widely applicable product, projects that are nonetheless successful in supporting valuable science and research activities.

Finally, concepts recommended for sustainability related to financial support are also valuable within a research group. That is, for groups that rely heavily on potentially transient contributors (like graduate students and post-docs) for development activities, internal project governance and solid documentation can provide some stability for projects and improve the transitions to new research staff.

Adoption and Reuse

This discussion centers on research software as that set of products aimed for use beyond the original research group. This may be presented as infusion; here we will consider two aspects that fall under that larger concept: adoption (by consumers) and reuse (by developers). For funders, the distinction is not important; for researchers and research groups, the distinction can help clarify what monitoring and other metrics they may need to collect based on the project's goals for infusion. We also consider the timing of the infusion signals in the metrics, both in term of the software phase and of the grant cycle.

We don't have research to describe growth rates or patterns in adoption of open source research software as applications or as reusable modules. This obviously makes it difficult to provide timeframes for assessing these metrics for a project. In some cases, it may be possible to evaluate some infusion metrics by project grant's end, for instance, if the release occurred well within the grant. For projects that release software products late in the funding cycle, there is simply no time to grow. This is as true for analytics as it is for altmetrics.

To provide some flexibility in assessing infusion that is aware of these timing concerns, we offer three general phases (durations are estimated and refer to the time since public release):

1. Early phase (<6 months): analytics provides little information, perhaps capturing early adopters or known project collaborators.
2. Middle phase (6 months < 18 months): analytics show active growth (adoption and/or reuse) and some sustained use. Potential altmetrics activity.
3. Advanced phase (> 18 months): analytics show continued growth, if limited, but sustained active user base (adoption and/or reuse). Altmetrics activity expected.

We do not provide any expectations of scale in these phases as these must be considered in terms of the research goals and the communities being served. For example, the adoption of a hydrology model will display different characteristics than would a citizen science data collection web application. These considerations become one more aspect of of our larger context-driven approach, tied to progression, to product type, to research goals, that need to be addressed to support equitable and realistic assessment efforts when addressing “time to science” and ROI from the funder’s perspective.

Returning to research code specifically, we want to consider adoption and reuse not necessarily of the code as the primary concern but of the dataset generated or analysis performed using that code. Here again, the research code is a secondary output but one that is still very key to the research goals. The code’s impact is tied to the impact of the dataset and it’s this situation where the value of strong cultural practices around software and data citation as well as use and quality analyses using provenance traces (Car 2016) comes to the forefront. How we value those measures for code as secondary outputs is an open question.

Using this guidance for assessment

We have touched upon assessment throughout this document as we describe some of the ways that development intersects with our research environments and practices. We have touched on the various stakeholder expectations, development progression and maturity, sustainability and infusion, and the guidelines themselves. These provide a foundation for implementing an assessment instrument when taking into account the different stakeholder needs and different community needs, although providing an explicit model to address each is outside the scope of this document.

We err here in favor of externally verifiable indicators for any assessment task, whether that assessment is done in-house or through a more formal process. In some cases, these indicators will support other sustainability and adoption goals. For most assessment, such as those done by a funding agency, sustainability and adoption/reuse are often the only metrics of interest for evaluating a piece of research software.

There are also recommended practices, invisible to this kind of external review, for improving code quality. Code reviews, pair programming, Agile or Lean management practices—these activities can be difficult to discern from the project artifacts that we consider in the guidance above. While there are practices described there that are commonly used in Agile management, for example, we are not advocating any one management style. We are, rather, noting that there are processes involved in the development of these products that are difficult or impossible to assess at a project's end or at all.

For those wishing to use these guidelines for assessing research products for adoption or reuse, we note that there is no one measure for fitness of use provided by these guidelines. Each researcher or research group must weigh their own project's goals and resources (every dependency is a maintenance cost) when evaluating these products for reuse. We do not recommend the use of these guidelines alone when evaluating projects for adoption as development process maturity is not the most relevant measure of whether a research product is a worthwhile tool to add to your research workflow. Each project, and project phase, requires its own analysis of potential reuse/adoption, an analysis that can be aided by the discussion here and by metrics such as those found in the Reuse Readiness Levels (RRLs) put forth by NASA's Earth Science Data Systems' Software Reuse Working Group (2010).

Returning to our scenarios, we would like to address concerns around just what is likely to be assessed. We are not speaking for any funders in this; however, we all recognize that it is simply not feasible to review every code/software product and that no one assessment effort will address the different stakeholder needs and expectations. From the ESIP community, we place efforts to improve the technical capabilities of our researchers and research groups as key to approaching the underlying concerns found in the most frequently discussed needs for assessment: "rigor" in code-dependent research activities (from the research community) and return on investment or "time to science" (from the funding agencies).

Conclusions

These, or any code/software, guidelines alone reflect only one aspect of a research project involving code or software development. We never want to lose sight of the research goals in favor of a purely technical answer. That is, from the viewpoint of a developer, the fastest way to reduce "time to science" and one that disregards the necessities of our research communities. Our research activities rely heavily on code and, while code is code and these guidelines reflect that understanding, we do have additional expectations of this code because it is written for research activities. We expect rigor, inspection, trust. To promote that, we need to support our communities and provide the education and training to produce, manage and understand the code/software throughout the research lifecycle.

We will end with one final note. This document is as much about creating a shared understanding around what it takes to produce research code/software. We are not advocating

that every researcher must also be a trained software engineer. It is more important that our researchers have the technical literacy and knowledge to understand how code supports their research, the limitations of code and the ways in which it can fail. Those that continue to code and continue to learn have our support. But those that don't continue on that path may still have active roles in shaping the technologies that we create. That may include product owners (for those working in an Agile system), project managers, domain researchers, principal investigators, or federal program officers. When we consider the ways in which producing code and software has grown more complex, in terms of technologies and in the number of skillsets required, we cannot ignore the very real need to better support interdisciplinary teams and more collaborative approaches to research development. We start by building these shared understandings of our research development activities so that we can better support our research communities.

Contributors

ESIP
Valerie Toner (NOAA)
James Gallagher (OPeNDAP)
Robert Downs (CIESIN)
Anne Wilson (LASP)
Kim Kokkonen (LASP)
Joshua Elliott (LASP)
Chris Pankratz (LASP)
Ruth Duerr (Ronin Institute)
Ryan Bowe (Geospatial Metadata)
Chris Kotfila (Kitware)
Leslie Hsu (USGS)
Sam Silva (MIT)
Bon Simons (NOAA)
Ben Barton (University of Alaska Fairbanks)
Greg Janée (UC Santa Barbara)

EarthCube/ESIP Software Assessment Workshop, June 1-3, Boulder, CO
Anne Wilson (LASP)
Emily Law (JPL NASA)
Ken Keiser (UAL-H)
Aleksandar Jelenak (The HDF Group)
Nathan Hook (NCAR-CISL)
Ward Fisher (UCAR/Unidata)
Bob Arko (UNOLS/R2R)
Adam Shephard (WHOI)
Matt Tricomi (Xentity)

Lindsay Powers (The HDF Group)
Arika Virapongse (University of Florida)

Community
Neil P. Chue Hong, Software Sustainability Institute, University of Edinburgh

Acknowledgements

Special thanks to Jennifer Ast for editorial support and Anne Wilson for her patience while rubberducking.