# Software Sustainability Institut: SSI Software Management Plan - Minimal Software Management Plan

## What software will you write?

What software will you write?

*Recommandations*:

**Questions to consider:**

- What will your software do?
- Will your software have a name? Do you have one in mind? Is this name unique and meaningful and not in violation of any existing trademarks?

**Guidance:**

When you come to choose a name for your software, see the Software Sustainability Institute's "Choosing project and product names".

## Who are the intended users of your software?

Who are the intended users of your software?

*Recommandations*:

**Questions to consider:**

- Is there just one type of user or are there many?
- Is your software for those new to your research field, for experts in your field, or for both?
- What, if any, software installation and configuration skills, knowledge and expertise will your users need? Will they need to be familiar with building and installing software via the command-line?
- What software development skills, knowledge and expertise do your users need? Will they need to develop their own code to be able to use your software?

**Guidance:**

Your intended users could include yourself, your research group, your department or institution, your project, your research community, other research communities, or the general public.

If your software is a framework that allows users to develop their own plug-ins, or is a library that users can use within their own research software, your users will need some experience of software development to get the most from your software.

## How will you make your software available to your users?

How will you make your software available to your users?

*Recommandations*:

**Questions to consider:**

- Will you release binaries, libraries or packages? How will users access these?
- Will your software be accessed solely as an online service or via a web portal?
- Will you release your source code? Do your funders, or other stakeholders, require you to release your source code? How will users access your source code?
- Will users have to register to access or use your software?
- Will you charge users a fee to access or use your software? What will the revenue generated by these fees be used for?

**Guidance:**

There are many ways in which you can release your software. These include: a binary executable that can be run directly; bundled in an installer program; an archive (.zip or .tar.gz) of binary executables or libraries, or as Python or R packages; an archive of source code; via a download link on a web site; via e-mail from you; via access to a source code repository hosted at your institution or on a third-party site such as GitHub, GitLab, BitBucket, LaunchPad or Assembla.

Building or compiling software can be complicated and time-consuming. If you can, provide your software in a form that can be deployed and used without requiring your users to build it. This saves your users both time and effort, and can be especially valuable if your users are not software developers.

Even if your funders or other stakeholders do not require you to release your source code, give strong consideration to releasing it anyway. See OSS Watch's "Benefits of Open Source Code".

If you don't need to monitor access to your software, or to restrict access to authorised users only, consider allowing anonymous access to and use of your software.

If you want, or need, users to pay a fee before users can access or use your software, you need to tell some users why they need to pay this fee and what they get in return. Your fee might be needed to help fund your time to develop your software, to fund your time to provide support for your software, or to help pay for any infrastructure or third-party dependencies you use.

Providing free and anonymous access to your software gives users immediate access to it to "give it a go". This can also

help make your software more appealing than competing software that has similar functionality, but which requires registration or charges a fee for its use.

## How will your software contribute to research?

How will your software contribute to research?

*Recommandations*:

**Questions to consider:**

- Will it help to produce results more rapidly?
- Will it help to produce results to a higher degree of accuracy or a finer level of detail?
- Will it help to conduct analyses cannot be conducted at present?
- Will it help users to exploit the power of modern super-computers?
- Will it, in some form, implement a novel solution to a research problem?
- What are the limitations of similar research software that already exists? How will your software be better?
- What are the benefits for each type of user?

**Guidance:**

When developing research software, it is good to have some idea as to how it will contribute to research, whether this is research done by you or by others.

## How will you measure its contribution to research?

How will you measure its contribution to research?

*Recommandations*:

**Questions to consider:**

- What evidence do your funders, or other stakeholders, expect you to present to show that your software has contributed to research?
- Will you measure who has downloaded your software?
- Will you measure who has used your software?
- Will you gather information on publications that describe research which your software has helped to enable?
- Will you have a recommended reference or citation for your software, or a related paper?
- Will you contact users via e-mail or questionnaires, or at conferences or workshops, to ask them how they used, and benefited from, your software?
- Will you encourage users to write blog posts on how they have used your software and how it helped them?

**Guidance:**

There are many ways to quantify interest in your software, including how many people have shown interest in your software, how many have used your software, and what they have used it for. These include: number of downloads; number of forks, if hosted on GitHub or BitBucket; number of pull requests or code contributions such as bug fixes, enhancements or extensions; number of e-mail list or forum members; number of support requests such as e-mails, bug reports, feature requests or open issues; papers you, and/or your users, have published to which the use of your software contributed; citations of these papers; blog posts by others about how they used your software; number of attendees at conference or workshop talks, demonstrations, poster sessions or tutorials.

Asking users to cite your software, directly or via a related paper, and providing a recommended citation, means you can search for these citations. Consider adding a citation requirement to your software's licence, so it becomes a condition of its use. See the Software Sustainability Institute's "How to cite and describe software" and "Oh research software, how shalt I cite thee?", which has examples of recommended citations for various software packages.

# Software Sustainability Institut: SSI Software Management Plan - Full Software Management Plan

## About your software

What software will you write?

*Recommandations*:

**Questions to consider:**

- What will your software do?
- Will your software have a name? Do you have one in mind? Is this name unique and meaningful and not in violation of any existing trademarks?

**Guidance:**

When you come to choose a name for your software, see the Software Sustainability Institute's "Choosing project and product names".

Who are the intended users of your software?

*Recommandations*:

**Questions to consider:**

- Is there just one type of user or are there many?
- Is your software for those new to your research field, for experts in your field, or for both?
- What, if any, software installation and configuration skills, knowledge and expertise will your users need? Will they need to be familiar with building and installing software via the command-line?
- What software development skills, knowledge and expertise do your users need? Will they need to develop their own code to be able to use your software?

**Guidance:**

Your intended users could include yourself, your research group, your department or institution, your project, your research community, other research communities, or the general public.

If your software is a framework that allows users to develop their own plug-ins, or is a library that users can use within their own research software, your users will need some experience of software development to get the most from your software.

How will you make your software available to your users?

*Recommandations*:

**Questions to consider:**

- Will you release binaries, libraries or packages? How will users access these?
- Will your software be accessed solely as an online service or via a web portal?
- Will you release your source code? Do your funders, or other stakeholders, require you to release your source code? How will users access your source code?
- Will users have to register to access or use your software?
- Will you charge users a fee to access or use your software? What will the revenue generated by these fees be used for?

**Guidance:**

There are many ways in which you can release your software. These include: a binary executable that can be run directly; bundled in an installer program; an archive (.zip or .tar.gz) of binary executables or libraries, or as Python or R packages; an archive of source code; via a download link on a web site; via e-mail from you; via access to a source code repository hosted at your institution or on a third-party site such as GitHub, GitLab, BitBucket, LaunchPad or Assembla.

Building or compiling software can be complicated and time-consuming. If you can, provide your software in a form that can be deployed and used without requiring your users to build it. This saves your users both time and effort, and can be especially valuable if your users are not software developers.

Even if your funders or other stakeholders do not require you to release your source code, give strong consideration to releasing it anyway. See OSS Watch's "Benefits of Open Source Code".

If you don't need to monitor access to your software, or to restrict access to authorised users only, consider allowing anonymous access to and use of your software.

If you want, or need, users to pay a fee before users can access or use your software, you need to tell some users why they need to pay this fee and what they get in return. Your fee might be needed to help fund your time to develop your software, to fund your time to provide support for your software, or to help pay for any infrastructure or third-party dependencies you use.

Providing free and anonymous access to your software gives users immediate access to it to "give it a go". This can also help make your software more appealing than competing software that has similar functionality, but which requires registration or charges a fee for its use.

How will your software contribute to research?

*Recommandations*:

**Questions to consider:**

- Will it help to produce results more rapidly?
- Will it help to produce results to a higher degree of accuracy or a finer level of detail?
- Will it help to conduct analyses cannot be conducted at present?
- Will it help users to exploit the power of modern super-computers?
- Will it, in some form, implement a novel solution to a research problem?
- What are the limitations of similar research software that already exists? How will your software be better?
- What are the benefits for each type of user?

**Guidance:**

When developing research software, it is good to have some idea as to how it will contribute to research, whether this is research done by you or by others.

How will you measure its contribution to research?

*Recommandations*:

**Questions to consider:**

- What evidence do your funders, or other stakeholders, expect you to present to show that your software has contributed to research?
- Will you measure who has downloaded your software?
- Will you measure who has used your software?
- Will you gather information on publications that describe research which your software has helped to enable?
- Will you have a recommended reference or citation for your software, or a related paper?
- Will you contact users via e-mail or questionnaires, or at conferences or workshops, to ask them how they used, and benefited from, your software?
- Will you encourage users to write blog posts on how they have used your software and how it helped them?

**Guidance:**

There are many ways to quantify interest in your software, including how many people have shown interest in your software, how many have used your software, and what they have used it for. These include: number of downloads; number of forks, if hosted on GitHub or BitBucket; number of pull requests or code contributions such as bug fixes, enhancements or extensions; number of e-mail list or forum members; number of support requests such as e-mails, bug reports, feature requests or open issues; papers you, and/or your users, have published to which the use of your software contributed; citations of these papers; blog posts by others about how they used your software; number of attendees at conference or workshop talks, demonstrations, poster sessions or tutorials.

Asking users to cite your software, directly or via a related paper, and providing a recommended citation, means you can search for these citations. Consider adding a citation requirement to your software's licence, so it becomes a condition of its use. See the Software Sustainability Institute's "How to cite and describe software" and "Oh research software, how shalt I cite thee?", which has examples of recommended citations for various software packages.

## Your software development infrastructure

What infrastructure will you need?

*Recommandations*:

**Questions to consider:**

- What infrastructure will you need, now and in the future?
- Who needs access to this infrastructure, and for what?
- Does it provide the features you need now?
- Does it provide the features you will need in the future?
- What infrastructure needs to be kept private, and what can be publicly visible?

**Guidance:**

Source code should be held under revision control (version control). With revision control you can retrieve any version of your software, or any file within those versions, from any point in time. Revision control allows multiple developers to work on the same software, at the same time, sharing their changes without the risk of a file being overwritten and its previous contents being lost forever. Revision control automatically records who changed what, and when, and allows you to record why the changes were made. It provides a complete audit trail of the evolution of your software for you and, in an open source world, for your users and developers. See "Make Incremental Changes" in "Best Practices for Scientific Computing" and the Software Sustainability Institute's "Top tips on version control". Popular revision control tools include Git, Mercurial, and Subversion.

Other infrastructure you might need can include: a web site, a wiki, a ticketing system (for managing queries, bug reports, feature requests and any other task that needs done), mailing lists or forums with publicly searchable archives, chat rooms, blogs, test servers, continuous integration servers (for automatically run tests whenever changes are committed to a source code repository), collaborative document editing tools, and project management tools.

See the Software Sustainability Institute's "Infrastructure for unselfish software development", which suggests

infrastructure to start developing research software, infrastructure for closer collaboration, and infrastructure to strengthen community engagement and deliver reliable software.

Where will your infrastructure be hosted?

*Recommandations*:

**Questions to consider:**

- Will you host the infrastructure yourself?
- Is there departmental or institutional infrastructure you can use?
- Does your funder recommend infrastructure that you can use?
- Will everyone who needs to access the infrastructure be able to access it?
- Is the host's quality of service for the infrastructure acceptable for your needs?
- Is the infrastructure free or do you have to pay a fee to the host? If you have to pay a fee, can you afford the payments for the lifetime of your project?
- Does the infrastructure, and host, look like they will be around for as long as you need them?
- Is it easy to back up, or export, all your content?
- Are there any alternatives that are also suitable, should you need to migrate in future?

**Guidance:**

Ask your IT support staff if there is departmental or institutional infrastructure you can use. Likewise, ask your funders if they recommend, or require, a specific host to use.

Popular third-party hosts for source code repositories include as GitHub, GitLab, BitBucket, LaunchPad and Assembla. These hosts also provide other project infrastructure including web sites, ticketing systems, wikis and notifications. Third-party hosts can differ in their pricing and levels of service. Some charge for private repositories. Some offer discounts or free hosting for academics.

Ticketing systems that can be deployed locally include JIRA, Bugzilla and Trac. Trac also provides a wiki. JIRA and Trac can also be used for project management. A third-party project management tool is Trello.

Continuous integration servers include Jenkins, which can be deployed locally, and Travis CI, a hosted continuous integration service that automatically runs tests whenever changes are committed to a source code repository on GitHub.

Third-party mailing lists include Google Groups and MailChimp.

Collaborative document editing tools include EtherPad, an open source, collaborative document editing service that can be deployed locally, or used via a public instance, or third-party tools such as GoogleDocs, and HackPad.

See the Software Sustainability Institute's "Choosing a repository for your software project" and "Top tips for choosing a computing infrastructure".

## Developing good software

How you will deliver code that can be understood?

*Recommandations*:

**Questions to consider:**

- Do your implementation languages have coding standards?
- Will you define a coding standard to which your code will be expected to conform?
- Will you use structured comments from which documentation can be automatically generated?

**Guidance:**

Readable code is useful not only for the developer who writes the code but also for other developers who may need to modify or extend the code. This can include new researchers within your department or project, your collaborators, or, for open source software, your users. It can even include you, six months or a year from now, trying to understand what you wrote and why you wrote it that way.

A coding standard is a specification of how your code should look, including naming, formatting and use of whitespace. Many projects and organisations have coding standards including Google's "Java style", GNU's "Coding standards", and the UK Met Office's "Fortran 90 Standards". Some languages have generic coding standards including Java – Oracle's "Code Conventions for the Java Programming Language" – and the Python community's "PEP 0008 – Style Guide for Python Code".

Many languages support structured comments including JavaDoc for Java, Doxygen for C, C++, Fortran or Python, and Docstrings or Sphinx for Python. From these comments, documentation can be automatically generated in HTML, LaTeX or other formats. This documentation can help developers to understand your code without having to look at your source code.

See "Write Programs for People, Not Computers" in "Best Practices for Scientific Computing" and the Software Sustainability Institute's "Writing readable source code".

How you will deliver good quality code?

*Recommandations*:

**Questions to consider:**

- Will you organise regular code reviews?

- Will you encourage pair programming?

**Guidance:**

Code reviews, where one developer reviews another's code, provide an assessment of the quality of code, including: how understandable the code is, and how this can be improved; possible bugs; and, suggestions as to how the code can be made more robust, extensible, or efficient. Code reviews also provide a means by which knowledge of the code can be transferred between developers, thereby increasing the software's "bus factor".

Pair programming, where two developers write code using one keyboard, can serve as a form of continuous code review and knowledge transfer.

See "Collaborate" in "Best Practices for Scientific Computing".

How will you choose your test cases?

*Recommandations*:

**Questions to consider:**

- How will you assess whether your software is behaving as intended, that is, producing scientifically valid results?
- How will you assess whether your software is not behaving as intended?
- Will your intended users help you choose your test cases?

**Guidance:**

You need some way of deciding whether your software is working correctly, producing results that are scientifically valid as determined by you or the researchers who will use your software. It is also useful to have some way of determining if your software is not working correctly. You don't want your software to output valid data when given nonsensical input data, but to exit with some suitable error message, so you want to test that it does exit in such cases.

Test cases allow you to check that your software behaves as expected, when given both valid and invalid inputs, and continues to do so as it is extended, refactored, fixed, optimised or parallelised.

See the Software Sustainability Institute's "Testing your software".

How will you make it easy to write and run tests?

*Recommandations*:

**Questions to consider:**

- Will you use automated tests?
- Will you use a unit test framework?
- Will you use test coverage tools?

**Guidance:**

After modifying your software, you and your developers will want to check that your changes have not broken anything. Automated tests, code that tests code, can be written once and then run many times, for example every day, or, even, every time you change your software. What is a repetitive manual process becomes an automated process, which makes it easier for you and your developers to run the tests. See the Software Sustainability Institute's "Testing your software".

Unit test frameworks are available in many languages to help you write, discover and run tests, and report test results. See "Plan for Mistakes" in "Best Practices for Scientific Computing" and Wikipedia's "List of unit testing frameworks".

Code coverage tools are a useful complement to automated tests. They analyse your code as your tests are run and report on what parts of your code are, or are not, executed. They can help you to identify whether your tests are causing the critical parts of your software to be executed. See Cunningham and Cunningham's "Code Coverage Tools".

How will you let users know about the tests you do?

*Recommandations*:

**Questions to consider:**

- Will you publish information on the platforms and environments your software has been tested under?
- Will you publish your test cases?
- Will you publish your test results?
- Will you publish your test coverage results?

**Guidance:**

Publishing information on your test cases, tested platforms and environments and test results can give your users confidence in the quality of your software. You can automatically publish test results on a website or have test results mailed to a mailing list. Continuous integration tools can also be configured to publish test results.

How will you ensure that your software is tested regularly?

*Recommandations*:

**Questions to consider:**

- Will you set up an automated process to run your tests regularly?
- Will you use continuous integration?

- Will you use a hosted continuous integration service?

**Guidance:**

Automating the process of running tests allows you to run your tests on the most recent version of your software at regular intervals. This means you, and others if you publish the test results more widely, can quickly see the impact of changes you make to your software. At its simplest, this can be done by deploying a server that regularly executes a job to run your tests.

A more powerful solution is to use continuous integration, automatically running tests whenever changes are committed to your source code repository. Hosted continuous integration services can relieve you, and your organisation, of the cost and effort of deploying and hosting a continuous integration server yourself.

See the Software Sustainability Institute's "How continuous integration can help you regularly test and release your software" and "Hosted continuous integration".

How will you help developers to understand, modify, extend and test your software?

*Recommandations*:

**Questions to consider:**

- Will you document what developers need to know before they can start developing your software?
- Will you document how to build and deploy your software?
- Will you document how to test your software? Will you document lists of manual steps for test cases which cannot be run automatically but which need to be tested?
- Will you document how to release your software?
- Will you document its design?
- Will you document its application programming interfaces?
- Will you document the data formats it uses for input and output?
- How often will you review your documentation to ensure it is up-to-date and captures all the knowledge about how to develop your software?
- How will you ensure knowledge is captured and exchanged so that nothing is lost when people leave?

**Guidance:**

Documenting everything about your software gets information out of the heads of you and your developers and into a persistent, shared resource such as a collection of documentation files in your source code repository, on a wiki, or within another resource that can host documents. This helps to increase your software's "bus factor". This documentation is also useful for other developers who wish to modify, extend, fix or test your code. This can include new researchers within your department or project, your collaborators, or, for open source software, your community.

See the Software Sustainability Institute's
"Developing maintainable software".

Will your software run under multiple environments?

*Recommandations*:

**Questions to consider:**

- What are the popular operating systems, web browsers, compilers or language versions used by your intended users?
- Would supporting additional operating system, web browsers, compilers or language versions encourage new users to adopt your software?

**Guidance:**

If the environments popular with your intended users are not those you would usually prefer, then target the environments of your users. Do not expect your intended users to migrate to your preferred environment as this puts a barrier between them and your software. For example, if Windows is the prevalent operating system of your intended users then it is unreasonable to expect them to migrate to Linux to use your software. As another example, if most of your intended users use Python 2.7 then don't use Python 3-only features in your code.

Code that compiles or runs under two or more of Windows, Unix/Linux and Mac OS X will have a larger potential user community than code that is restricted to a single operating system. Likewise, for web-based applications, those which can run within two or more of Internet Explorer, Chrome, Firefox and Safari have a larger potential community that those that restrict users to a single browser.

For C, C++ and Fortran code, it can be advantageous to support different compilers, even for the same operating system. For example, if you want your code to run on a Linux desktop and a super-computing service, you may want to consider supporting the GNU, Intel and Cray compilers. Or, for example, for C++ you might want to support compilation using the GNU C++ compiler on Linux and Visual C++ Express on Windows. Tools such as Automake and CMake can assist you in managing support for multiple compilers under different operating systems.

Another option for supporting multiple operating systems is to consider some form of virtualization. Both VMWare Workstation Player and Oracle VirtualBox allow you to create virtual machines, in which one operating system can run as a program under another operating system. So, for example, an Ubuntu virtual machine can run within Windows.

Docker provides a lighter-weight alternative to virtual machines. Rather than bundling the entire operating system, a Docker container bundles Linux-compliant software along with the environment it needs to build and run, including files, libraries and system tools. These containers can be deployed and run within a Linux operating system or, for Windows, within a VirtualBox Linux virtual machine.

How will your software and documentation adhere to disability accessibility guidelines?

*Recommandations*:

**Questions to consider:**

- Will the same information be presented in multiple ways?
- Will keyboard-only navigation through GUIs and web interfaces be supported?
- Will users be able to customise the appearance of GUIs or web interfaces?
- Will text and images be re-sizable without loss of clarity?
- Will documentation be readable in black-and-white without loss of clarity?

**Guidance:**

Accessibility refers to the ease with which users, regardless of disabilities or impairments, can use your software and documentation. For example, the continuous integration service Travis CI renders test successes with a green tick and failures with a red cross, which accommodates colour blind users. Using HTML “ALT” tags when developing web portals so that non-text content, such as images, have a textual alternative, allows these to be detected, and spoken by a screen reader.

There are a number of guides and checklists for designing accessible software and documentation. See, for example:

- MSDN’s guide on “Designing Accessible Applications”.
- IBM’s “Software accessibility checklist”.
- IBM’s “Web accessibility checklist”.
- Jakob Nielsen’s “Accessible Design for Users With Disabilities”.
- IBM’s “Documentation accessibility checklist”.
- WebAIM () resources for web accessibility.

## Managing your dependencies

What third-party software, models, tools, libraries and services will you use?

*Recommandations*:

**Questions to consider:**

- What are they needed for?
- Are they open source or proprietary?
- Are they free or do you have to pay a fee for use? If you have to pay, can you afford the payments for the lifetime of your project? Will your users need to pay too? Will this be acceptable to them?
- Do their terms and conditions of use or licences put any obligations or constraints on you, your software or your users?
- Can you redistribute them or do your users have to download them?
- Do they look like they will be around, and supported, for as long as you need them?
- What would be the impact if any of them were to disappear or cease to be supported? Are there any alternatives that are also suitable?
- How will you design your code to minimise the coupling to these dependencies as far as possible?

**Guidance:**

Using open source software can give you confidence that if the original authors disappear, or their project ends, or ceases to provide support, you at least have the means to be able to access, fix, improve or extend your software yourself. It also gives you the potential to make such changes when you need them, rather than having to wait for someone to do it on your behalf. See the Software Sustainability Institute’s “Choosing the right open source software for your project”. Much of its advice applies not just to choosing open source software, but any third-party software.

A common obligation when using third-party software is that you acknowledge its use, and, if including it in your software, that you document its inclusion and provide its licence with your software. “License compatibility” is “an issue that arises when licenses applied to copyrighted works, particularly licenses of software packages, can contain contradictory requirements, rendering it impossible to combine source code or content from such works in order to create new ones.” See David Wheeler’s “Free-Libre / Open Source Software (FLOSS) License” slide which summarises how popular open source licences can be combined, and GNU’s “Various Licenses and Comments about Them”.

You may plan for your software to invoke online services for specific computation or data manipulation functions. As users and developers of your software need to know about, and have access to, these services, the use of your software becomes dependent upon the availability of these services. You may want to consider whether you could embed the service’s functionality within your software, if the code implementing the service is open source, or implement the functionality yourself.

See the Software Sustainability Institute’s “Defending your code against dependency problems”.

What third-party data sets and online databases will you use?

*Recommandations*:

**Questions to consider:**

- What are they needed for?
- Are they open or proprietary?

- Are they free or do you have to pay a fee for use? If you have to pay, can you afford the payments for the lifetime of your project? Will your users need to pay too? Will this be acceptable to them?
- Do their terms and conditions of use or licences put any obligations or constraints on you, your software or your users?
- Can you redistribute them or do your users have to access or download them?
- Do they look like they will be around, and supported, for as long as you need them?
- What would be the impact if any of them were to disappear or cease to be supported? Are there any alternatives that are also suitable?
- How will you design your code to minimise the coupling to these dependencies as far as possible?
- Do you have a data management plan?

**Guidance:**

A common obligation when using third-party data is that you acknowledge its use or provide a citation.

See the Software Sustainability Institute's "Defending your code against dependency problems".

A Data Management Plan can help you plan for the effective management of data you will use, to enable you to get the most out of your research. See the Digital Curation Centre's "How to Develop a Data Management and Sharing Plan".

What communications protocols and data formats will you use?

*Recommandations*:

**Questions to consider:**

- Which communications protocols and data formats are commonly used within your domain?
- What are they needed for?
- Are they open or proprietary?
- Are they free or do you have to pay a fee for use? If you have to pay, can you afford the payments for the lifetime of your project? Will your users need to pay too? Will this be acceptable to them?
- Do their terms and conditions of use or licences put any obligations or constraints on you, your software or your users?
- Are they mature, ratified standards?
- Are there any alternatives that are also suitable?
- How will you design your code to minimise the coupling to these dependencies as far as possible?

**Guidance:**

Try to use any communications protocols and data formats that are commonly used within your domain. This can help to ensure interoperability with other software within your domain.

Open communications protocols are those whose specifications have been published and can be used and implemented by anyone. These include both generic protocols (e.g. application layer protocols such as REST, SOAP, HTTP, XMPP, SMTP and SSH, or lower transport level protocols such as TCP, or UDP). Similarly, open data formats are those whose specifications have been published and can be used and implemented by anyone. These include both generic data formats (e.g. GIF, SVG for images, HTML and XML for documents, tar and zip for archives, CSV, JSON, or NetCDF for data) and domain specific ones. See, for example, Wikipedia's list of "open formats".

Software that can communicate using an open communications protocol can be used with any software that uses this protocol. Likewise, if data can be imported and exported in an open format, then it can be used with any software that uses this format. Software that supports open communications protocols and data formats does not lock users into that software, because users can use, or develop, alternative software, if necessary. This can make it easier for users of other software may switch to your software, if your software is more innovative, efficient, robust, scalable or functional than that of your competitors'.

Try and adopt open protocols, interfaces and data formats that are mature, ratified standards, if possible. Standards can go through many iterations, because they evolve as ideas are proposed and debated and the scope, remit and intent of the standards are agreed. If a standard changes, then any software that uses the standard needs to be changed to keep up to date. Most of the big changes occur early in the lifetime of a standard. Mature and ratified standards are less likely to change significantly or frequently, which reduces the risk of you having to modify your software in response.

How will you document your dependencies?

*Recommandations*:

**Questions to consider:**

- What dependencies do users need?
- What dependencies do developers need?
- What information will you record about your dependencies?
- How and where will you record this information?

**Guidance:**

Documenting your dependencies, helps you, your users and your developers to understand all the components that are needed to build, develop, test and run your software.

Dependencies should be explicitly documented and not implicitly buried within your source code. It is waste of a user's time to build, deploy and run your software only to have it fail an hour later due to a missing dependency. You should also document your implicit dependencies on operating systems, languages, and browsers. Even a 10 line script has a

dependency on the language used to implement it and the operating systems it runs under.

Users need to know about the dependencies needed to run and, if applicable, build your software. Developers need to know about the additional dependencies needed to develop and test your software. Dependency information can include: name; purpose; whether it is mandatory or optional; origin, for example, web site, personal communication; copyright and licence; whether it is free or needs payment of a fee; version information, for example version number, repository commit identifier, date received, or digital object identifier (DOI); location of any local copies, for example in your source code repository with a summary of any local changes made (if applicable).

Understanding the licenses of third-party dependencies is important for your users and developers as these will impose constraints and obligations on how you can use, modify or redistribute these dependencies, which may affect how your software can be used, modified or distributed.

Version information is important. Different versions of software, models, tools, libraries, data formats, protocols, interfaces, services, databases, operating systems, languages and browsers, can differ in terms of syntax, behaviour or content. Software written to use one version might not be compatible with earlier or later versions. For example, Python code with the statement “print ‘hello’” will not run under Python 3, but code with the statement “print(‘hello’)” will run under both Python 2 and 3.

Dependency information can form part of your user and developer documentation, distributed with your software itself or published on a resource such as a web site or wiki.

See the Software Sustainability Institute’s “How to cite and describe software” and “Oh research software, how shalt I cite thee?”.

How will you track changes to your dependencies?

*Recommandations*:

**Questions to consider:**

- How will you track changes to your dependencies?
- Will you check for updates to your dependencies on a regular basis?
- Will you update dependencies in response to requests from your users?

**Guidance:**

Dependencies can evolve over time, with new features being added, and old features being deprecated. You may want your software to evolve to support newer versions so you can exploit new features benefit from bug fixes or optimisations, or continue to support the latest data formats, interfaces or protocols. Your users may also request that you update your software in response to updates to dependencies.

You should also keep an eye on any changes licensing conditions for subsequent versions of your dependencies. Changes in licencing terms and conditions could have consequences for you, your users and developers, which may affect how your software can be used, modified or distributed.

Will you use dependency management tools?

*Recommandations*:

**Guidance:**

There are tools available to help automate management of software dependencies, including:

- Ivy and Maven for Java.
- Python pip and setuptools.
- PHP Composer.
- Ruby gems.
- R PackRat.
- CPAN-related tools for the Perl CPAN archive.

These tools provide a means of documenting information about dependencies, including name, version and origin. They also pull in dependencies automatically, either from the web, or from a local directory containing the dependencies, saving users and developers from having to do this themselves.

## Managing your software development

What effort will be available to develop your software?

*Recommandations*:

**Questions to consider:**

- What funded effort will you have?
- What unfunded, or additional, effort do you have available?
- Will you accept contributions from your users?
- Will you encourage your users to contribute to your software?

**Guidance:**

You might have access to unfunded effort that can help with the myriad tasks around developing your software. You may have PhD students who can spend some of their time working on your software. Masters and degree students might

provide another source of effort, either for free, as part of a degree project or a summer job.

If you release your software, or make it available for use, then you may get contributions from your target users, including: bug fixes, new features, enhancements, corrections to documentation, case studies, tutorials or walkthroughs. If managed correctly, contributions can provide you with free effort for your project, and can provide fixes and features that you do not have time or effort to implement yourself.

See the Software Sustainability Institute's "Recruiting student developers".

How will software development roles be assigned?

*Recommandations*:

**Questions to consider:**

- Who will assign roles to those working on your software?

**Guidance:**

Writing research software, especially that which is to be used by others, is more than just writing code. It can include: writing documentation; reviewing code and documentation; preparing releases, responding to bug reports, feature requests and other questions from users; preparing presentations and demonstrations; writing papers; writing reports for stakeholders; preparing papers; setting up, and maintaining, project infrastructure; porting to cloud or super-computing resources etc.

If it is just you developing your software, then you will fulfil all these roles. If you are part of a team, then you need to decide who does what.

How you will track who is doing what and when it needs to be done by?

*Recommandations*:

**Questions to consider:**

- What information will you need for monitoring progress?
- What information will you need to report to other stakeholders, for example, funders?
- How will this information be recorded?
- Who will keep this information up-to-date?
- Who will have access to this information?

**Guidance:**

Ticketing systems such as JIRA, Bugzilla and Trac and those provided by popular third-party services for source code repositories such as GitHub, GitLab, BitBucket, LaunchPad and Assembla, as well as project management tools like Trello can be used to help assign tasks and monitor progress.

What software development model will you use?

*Recommandations*:

**Questions to consider:**

- Are your requirements known, or will they be known, before development begins?
- Will your requirements only become known once your intended users have a working version they can use?
- Do you need a proof-of-concept of your software completed as soon as possible?
- Do you have scope to develop your software in phases, rather than implementing all its required functionality in one go?

**Guidance:**

Software development involves a number of activities including identifying the requirements that the software must satisfy, designing the software, implementing the software, testing the software, deploying or releasing the software, and maintaining the software. When and how these are done is the remit of software development models, also called lifecycles or processes. Many software development models are available, varying in their formality and flexibility. Under the traditional waterfall process, software development proceeds sequentially through the activities above. Iterative development can be viewed as a sequence of waterfalls, allowing for feedback on earlier versions to contribute to the design of later versions. Agile methods promote also iterative development so that working software is released early and then is evolved and extended. These are complemented with continuous user engagement, where requirements are ever evolving, and allow for rapid and flexible changes in goal in response to these.

See Wikipedia's "Software development processes" and Robert Half Technology's "6 Basic SDLC Methodologies: The Pros and Cons", which includes waterfall, iterative and agile models.

How you will manage releases of your software or updates to your services?

*Recommandations*:

**Questions to consider:**

- Who decides what features, enhancements and bug fixes a release will contain?
- Who decides when the software is ready for release, and schedules the release?
- Who prepares, checks, and publishes a release?

- How is the history of releases recorded?

**Guidance:**

If you do not plan to release your software, or people will access your software as a service or via a web portal, then this question relates to deciding when to update the software that underpins your deployed services.

A release history – including release dates, version numbers and/or commit identifiers, key features and changes of each release – allows you, and your users, to see how your software has evolved. It also demonstrates to users how active you are in developing and maintaining your software. Software that is seen to be regularly fixed, updated and extended will be more appealing than software that seems to have stagnated.

How will you ensure that information is not lost when a developer leaves?

*Recommandations*:

**Questions to consider:**

- Will you ensure that information from developers is captured and documented?
- How will you ensure knowledge is captured and exchanged so that knowledge is not lost when people leave?

**Guidance:**

See the Software Sustainability Institute's "Top tips for software development hand overs".

How often will you review and revise your Software Management Plan?

*Recommandations*:

Your Software Management Plan is not static but should be reviewed and adjusted as a project progresses. Questions that could not be answered at the outset of a project might now have answers. Some processes may not have worked well and need to be replaced. New options for hosting or dependencies might become available.

Updating the Software Management Plan allows you to reflect on how your project is progressing and to take corrective action.

How does your Software Management Plan relate to any Data Management Plan?

*Recommandations*:

**Questions to consider:**

- Will your software rely on data produced by others?
- Will your software produce data that needs to be published, shared or preserved?

**Guidance:**

A Data Management Plan helps you plan for the effective creation, management and sharing of your data, to enable you to get the most out of your research. See the Digital Curation Centre's "How to Develop a Data Management and Sharing Plan".

## Engaging with your users

How will you promote what your software does and who has used it?

*Recommandations*:

**Questions to consider:**

- Will you blog or tweet regularly?
- Will you publish case studies to show how yourself and others have used your software?
- Will you publish a list of your publications, and those of others who have used your software?
- Will you submit papers or posters to conferences or workshops?
- Will you run demonstration sessions or tutorials co-located with conferences or workshops?
- Will you run your own workshops based around your software?
- Will you publish statistics on the number of users you have?

**Guidance:**

Blogs, Twitter, Facebook and RSS feeds are all effective, low cost, ways of promoting your software. They also provide a way for you to engage with your users, for example, by encouraging them to follow you, or inviting your users to write guest blog posts.

Case studies can provide intended users with examples of how your software can be used in practice, and the contribution it has made to your, and others', research.

Listing publications provides an academic perspective on the value of your software. It can also help your users, and other stakeholders, such as current or potential funders, to understand, in detail, how your software contributes to research, and what scientific problems it has helped to solve. These can also help to show where your software sits in relation to other software that fulfils a similar need, and what makes yours different, and better.

Promoting the existence of a user community can help persuade others to adopt your software also, giving them confidence that your software can benefit them, and that, if they run into problems, there is a community who can help, and who they can share experiences with.

See the Software Sustainability Institute's "Five top tips for promoting your software" and "Top tips for expanding your user community".

How you will support your users when they ask for help?

*Recommandations*:

**Questions to consider:**

- How will your users ask for help?
- How will you manage their requests for help?
- How much effort will be available to support your users?
- What level of support will you offer?
- Where will you publish information about the nature and level of support available?
- Will users be able to see what other users have asked and the associated answers?

**Guidance:**

If you release your software, or make it available for use, then you will get questions about how to use it. An ignored request for help can lead to a disgruntled user who may bad-mouth your software or you. Responding to requests for help does not imply that you have to spend time in fixing bugs or implementing features when they ask, it merely acknowledges that you've received their request. No one has a right to expect support for freely provided software.

There are many ways in which you can have your users request help and support, including: an e-mail to you, via telephone, to an e-mail list or forum, or creating an issue in a ticketing system.

A ticketing system records who asked what, and when, and allows you, and them, to record additional information about a query, to assign someone to handle a specific query, and to prioritise queries so that you can work on the most important first. Examples of ticketing systems include JIRA, Bugzilla and Trac. Many source code repository hosts, including GitHub, BitBucket, LaunchPad and SourceForge also provide ticketing systems.

How much effort you have available to support users will be up to you. You can even choose to provide no support. It is always good to make what, if any, support you will provide clear on your website or in your documentation, e.g. "we will reply to all e-mails within a week and will let you know when, or if, we can address your issue". A user will always want their problem to be solved as quickly as possible, and may become disgruntled, and might even stop being a user, if this is not the case. If you are clear and honest about the level of support you can provide, then they have fewer grounds for complaint. At the very least, the information about the nature and level of support available should be provided in your software's documentation. It should also be clearly visible from any place where users access your software.

There are many ways to publish information about the help and support requested by other users, and how these were resolved, including: e-mail archives, lists of frequently asked questions, or a publicly visible ticketing system. Encouraging users to search these resources before getting in touch can help users to help themselves, and reduces the overhead you need to spend on support.

See the Software Sustainability Institute's "Supporting open source software" and "Top tips for managing support requests". Many of the points apply not just to supporting open source software, but any software.

How will your users be able to contribute to your software?

*Recommandations*:

**Questions to consider:**

- Will you accept contributions from your users?
- Will you encourage your users to contribute to your software?
- Will contributors retain copyright over their contributions?
- Will you define a contributions policy?

**Guidance:**

If you release your software, or make it available for use, then you may get contributions from your users, including: bug fixes, new features, enhancements, corrections to documentation, case studies, tutorials or walkthroughs. If managed correctly, contributions can provide you with free effort for your project, and can provide fixes and features that you do not have time or effort to implement yourself. It can also help evolve a community of users and developers around your software.

Asking contributors to sign over their copyright and intellectual property can deter users from contributing. It, in effect, asks them to give away ownership of something that may be novel and which may represent a key aspect of their research. Allowing contributors to keep their own copyright and intellectual property removes this barrier, thereby making contribution a more attractive option. It also helps to promote a community round your software, as everyone is encouraged to share their outputs, without loss of credit or ownership.

A contributions policy provides information to your users on: what they can contribute; how they can contribute it, for example, via e-mail, patch files, or GitHub pull requests; any requirements they must satisfy, for example, compliance to coding or style conventions, passing required tests, making their code available under a certain licence, signing over their copyright; and, what happens to their contributions once they have submitted it, for example how it is reviewed and then added into your code, documentation or web site.

Users might not contribute if they do not know that they can contribute, so your contributions policy should be clearly visible from any place where users access your software.

For information on how to manage contributions, see OSS Watch's

"Contributor Licence Agreements".

## Intellectual property, copyright and licencing

Who will own the copyright of your software?

*Recommandations*:

**Questions to consider:**

- Does your funder, project or employer have a copyright statement you are required to or are recommended to use?
- Does your contract of employment say anything about copyright?
- Who will own the copyright of any contributions to your software that come from outside your institution or project?
- Has the copyright been agreed with your stakeholders? If not then why not and how do you plan to resolve any disagreements?

**Guidance:**

Copyright holders must be legal entities such as people, companies, or institutions, or groups of these.

If you are an employee of an institution or company, check your contract and with your employer, since they may own the copyright on any work you do as an employee.

If you work for a university, you may have a research exploitation office that can advise you on who owns the copyright, and how to resolve any disagreements within your stakeholders. Your department may also be able to advise you.

Users may contribute enhancements, extensions and bug fixes to your software. These contributors can either retain copyright of their contributions or transfer their copyright to you, via a formal copyright transfer agreement. The latter gives you ownership of their code, but can deter users from making contributions to your code or documentation that you might find useful. See OSS Watch's "Contributor Licence Agreements". Much of its advice applies not just to open source software, but any software.

For an introduction to intellectual property and the intellectual property rights of copyright, patents and trade-marks, see JISC's "Intellectual property law". See also JISC's "Copyright law.

What licence will you choose?

*Recommandations*:

**Questions to consider:**

- Will you be releasing your software?
- Does your funder, project or employer have a licence you are required to or are recommended to use?
- Will you choose a proprietary licence, an open source licence, or will you release your software into the public domain?
- If choosing an open source licence, will you choose an OSI-approved open source licence?
- Is your chosen licence valid under your national laws?
- Is your chosen licence acceptable to your stakeholders? If not then why not and how do you plan to resolve any disagreements?

**Guidance:**

If you do not plan to release your software, or people will access your software as a service or via a web portal, then this question may not be applicable.

For each licence, specify its name, a link to its licence text (if this is online), and a brief explanation as to why it has been chosen. If there is no online link, summarise the key terms and conditions.

If you are a copyright owner, you can licence your software in as many ways as you want.

If you plan to modify (and especially distribute) third-party software then your choice of licence may be affected by the licences of that software. For example, if you modify code released under the GNU Public Licence then you are required to release the source code of your modifications along with any binaries.

Making your source code available can give your users confidence that if you disappear, or your project ends, they at least have the means to be able to access, fix, improve or extend your software themselves, or employ others to do so. It also gives them the potential to make such changes when they need them, rather than having to wait for you to do it on their behalf. See OSS Watch's "Benefits of Open Source" and the Software Sustainability Institute's "Choosing an open source licence".

You might want to consider dual licencing, a model where software can be released for free under an open source licence such as the GNU Public Licence (so users who make modifications have to release the source code of their modifications along with any binaries) but also released under a proprietary, and paid-for, licence to those users who do not want to release the source code of their modifications. A variant is the so-called "Freemium" model, where you release a version of your software that has a baseline set of functionality your users will find useful under an open source licence, and to make available more powerful, efficient or novel components as plug-ins, licenced under a proprietary licence. See OSS Watch's "Dual licensing".

The Open Source Initiative (OSI) has produced an "Open Source Definition". This promotes a shared understanding of the term 'open source'. Some open source project hosting services will only host code licenced under an OSI-approved licence. See the OSI's list of "OSI-approved licences".

Certain licences may not be valid within certain countries. For example, it is not clear whether theMIT Public Licence is valid under UK law as under UK law you cannot reject liability for personal injury or death. In this case, choosing an

alternative free, open source, OSI-approved licence, e.g. Apache 2, would provide similar licence conditions to the MIT Public License, but provides a limitation of liability that does not exclude applicable laws. Again, your research exploitation office or department may be able to advise you on whether a licence is valid.

The Lindat License Selector is a tool which may help you select licences for both your software and, if applicable, data.

You might choose to waive all copyright and release your software into the public domain, so anyone can do anything with it. See the Open Source Initiative's "What about software in the "public domain"? Is that Open Source?".

If one or more partners are unhappy with the licence then they may be reluctant to use or contribute code, which could jeopardise the success of your project. Another choice of licence may be more acceptable to you and your stakeholders.

If you work for a university, you may have a research exploitation office that can advise you on which licence to choose, and how to resolve any disagreements within your stakeholders. Your department may also be able to advise you. Your funder may have a recommended licence, or type of licence, you are required to use.

Where will you publish your copyright and licence?

*Recommandations*:

A statement of copyright for your software and documentation makes it clear to users who created, and owns the rights to, the software and documentation. Users also need to know the licensing conditions of your software, and also of any third-party software bundled with it, since these may impose constraints and obligations on how, or whether, they can use, modify or redistribute your software.

You can publish your copyright statement and license in myriad places, including: on a web site, on a download page, within an installer screen, within your user documentation, within a README file in a source or binary archive or source code repository, or as a Help => About menu option in a graphical user interface.

Including a copyright and licence statement in each of your source code files, as a comment, means that the copyright and licence statement stays with the source code, even if individual files, or sets of files, are copied and used in other software. As full licences can be very verbose, you may just want to have a short comment that states the name of the licence, its key points and a link to the full licence text online. Many common licences provide examples of such comments for use. See, for example "How to Apply These Terms to Your New Programs" for the GNU General Public License version 3 or "Appendix: How to Apply the Apache License to Your Work" for the Apache License Version 2.0.

## Preserving your software

Do you have a preservation plan?

*Recommandations*:

**Questions to consider:**

- Is your software covered by a preservation policy or strategy from your funders or stakeholders?
- Is there a clear purpose in preserving your software?
- Is there a clear time period for preservation?
- Do the predicted benefits exceed the predicted costs?
- Is there motivation for preserving your software?
- Is the necessary capability available?
- Is the necessary capacity available?

**Guidance:**

See Neil Chue Hong, Steve Crouch, Simon Hettrick, Tim Parkinson, Matt Shreeve, "Software Preservation Benefits Framework", The Software Sustainability Institute and Curtis+Cartwright Consulting Ltd, 7 December 2010.

Where can you deposit your software for long-term preservation?

*Recommandations*:

**Questions to consider:**

- Will you preserve your software yourself?
- Is there a departmental or institutional service you can use?
- Does your funder recommend, or mandate, a service that you can use?
- Can you continue to use the host of your source code repository?
- Will everyone who needs to access the service be able to access it?
- Is the host's quality of service for the service acceptable for your needs?
- Is the service free or do you have to pay a fee to the host? If you have to pay a fee, can you afford the payments for as long as you need?
- Does the service, and host, look like they will be around for as long as you need them? How much advance warning will you be given if the service is discontinued?
- Are there any alternatives that are also suitable, should you need to migrate in future?

**Guidance:**

You may need to change your host in future if their policies or practices change or they cease to exist. For example, in September 2010 the UK e-Science repository service, NeSCForge was terminated after many years, and in 2015 there was controversy over SourceForge's handling of projects they had perceived to be abandoned.

See the Software Sustainability Institute's "Choosing a repository for your software project".

Do you plan to evolve your project into an open source project?

*Recommandations*:

**Questions to consider:**

- Do you have a governance model?
- Do you have the infrastructure to run an open source project?

**Guidance:**

A governance model sets out how an open source project is run. It describes the roles within the project and its community and the responsibilities associated with each role; how the project supports its community; what contributions can be made to the project, how they are made, any conditions the contributions must conform to, who retains copyright of the contributions and the process followed by the project in accepting the contribution; and, the decision-making process in within the project. Though they are designed for open source projects, many of their concerns are relevant to any software project.

For an open source project, you will need similar infrastructure to that which you have used during your project to date. OSS Watch recommend, at a minimum, a web site, a developer mailing list or forum, version control and an issue tracker. Popular third-party hosts for open source projects include GitHub, GitLab, BitBucket, LaunchPad and Assembla. These hosts can differ in their pricing and levels of service. Some charge for private repositories. Some offer discounts or free hosting for academics.

See OSS Watch's "Governance models" and "Essential Tools For Running A Community-Led Project".