Recommendations: ePIC Source Code Management

This set of recommendations follows RFC 2119 definitions for must, should, and may.

In particular, the word “should”, or the adjective "recommended", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course. The word “may”, or the adjective "optional", mean that an item is truly optional.

Scope of Recommendations

- Repositories under the github.com/eic organization should follow these recommendations.

Metadata, Community Documents

- Repositories should choose an open source license.
    - Rationale: The copyrights of all source code are held by the author(s). Without an explicit license, third parties have no implicit rights to copy or distribute the source code. In a collaborative environment, the absence of a license either hampers collaboration or encourages routine ignoring of intellectual property. Similar to academic citation practices that require attribution in recognition of previous work, and in line with collaboration practices that encourage contributions to the common good, open source licenses are considered the appropriate way to license source code in the ePIC collaboration.
- Preference should be given to open source licenses of core dependencies, predecessor projects, or examples upon which the source code is based.
    - Rationale: Much of the ePIC source code is not completely independent but builds on other original projects through examples or partial code reuse. Adherence to the license conditions of the original projects is often facilitated by using the same license as the original project.
- Repositories should not be licensed based on the preferred license of employers or of individual contributors.
    - Rationale: Collaboration source code is the intellectual property of the entire collaboration, not of individuals or their employers. Preferring the license by one employer introduces the appearance of a preference for some contributors over others. We are not aware of any collaboration members who are not able to contribute to collaboration software projects because of their open source licenses.
- In case there is no clear preference, we recommend using the LGPL.

Accessibility and Permissions

- Repositories should be public.
    - Rationale: eEIC is a publicly-funded international collaborative project. Restrictions on access go against the spirit of collaboration.
- Repositories may list documentation to justify deviations from these recommendations.
- Private repositories should be transferred to the github.com/eic organization as soon as either the number of ePIC developers and users is larger than five.
    - Rationale: Over the course of source code development, there is often a transition from a phase where it makes sense to have a private repository to when it makes sense to host repositories centrally. This introduces a recommended point for this transition.
- Repositories should use write permissions based on groups defined through the github.com/eic organization. Repositories should not add contributors as external collaborators.
    - Rationale: Consistent access management of a large number of users is facilitated by group-based permissions schemes. This leads to transparency to the users.

Coding Style and Formatting

- Repositories should define a coding style.
    - Rationale: The ePIC collaboration does not impose an overall style. Nevertheless, each source code repository will benefit from a uniform style: readability through consistency is more important than the details of a source code style guide.
- Repositories may enforce a coding style through pre-commit, continuous integration, or other automatic mechanisms.
    - Rationale: Automatic processes are preferred over requiring people to spend time worrying about coding style.

Branch Protection, Code Reviews, and Code Check

- Repositories should protect default branches from direct commits.
    - Rationale: Default branches in repositories are often assumed to be stable. This requires that some minimal checks are implemented to ensure this stability is maintained.
- Repositories should use pull requests before merging into default branches.
    - Rationale: Direct commits or merges into default branches move the discussion and decision making into private areas, away from collaborative interfaces.
- Repositories should use code reviews before merging into default branches.
    - Rationale: Everyone sometimes makes mistakes that could be caught during a code review phase.
- Code review should be shared between multiple core developers.

    - Rationale: If code review is always performed by one person, there is a tendency to start to think of the repository as ‘owned’ by that person. We want to avoid using code review as a gatekeeping mechanism. A second round of code review is often a good chance to bring in a new person.
- Code review should be shared between core developers and other contributors.
    - Rationale: To ensure sustainability and workforce development, core developers are encouraged to assign code review to others outside of the core developer team. This builds a culture of shared ownership and allows other contributors to develop the leadership skills they may use as they grow into core developers.
- Repositories should use either pull requests from forks only (preferred) or pull requests from branches in the main repository only. Repositories should not use a mixture of both methods.
    - Rationale: Repositories can have good reasons to prefer branches over forks (in particular as it concerns repository secret sharing). When functionality requires branches over forks, all pull requests should come from branches. When full functionality is available via forks, then that leads to overall lower use of privilege.
- Repositories should remove merged branches.
    - Rationale: Maintaining repositories with a limited number of branches makes it clearer which branches are inactive (rather than merged), and avoids contributors continuing to use the same branch for development.
- Continuous integration may consist of two types of tests: functionality tests ensure compliance with the technical interfaces (compilation tests, smoke tests, unit tests) and validation tests assess changes in performance where no clear correctness is implied (standardized plots, benchmark suites).
- Repositories should use continuous integration for functionality tests.
    - Rationale: Functionality tests allow to preserve and share knowledge of how software needs to be checked. Good functionality tests become a tool that reduces friction and allows for software development to advance confidently. Continuous Integration reduces the impact of the human factor.
- Repositories may use continuous integration for validation tests.
    - Rationale: Validation tests allow for an assessment of the quality of the software product. Good validation tests compare the software to known quantities (MC truth or artificial input), experimental input (test beam measurements), or other widely accepted software.
- Merging of pull requests should be done using squash merges.
    - Rationale: During development in branches, the code may be in states where it does not pass continuous integration tests. With regular merges, these individual commits end up in the default branch history. Identifying when different behavior was introduced through bisecting is greatly hampered by failing commits. With squash merges the history of the default branch should be in a working state for all commits.

Stability and Reproducibility

- Mission-critical repositories are those repositories that are used in a production campaign.
- Mission-critical repositories should use a versioning scheme that facilitates a summary understanding of when new features are introduced versus when bug fixes are applied.
    - Rationale: Mission-critical software often requires extensive more validation when new features are introduced than if bug fixes are applied. Many software version schemes (such as semantic versioning) allow for a patch number. When only the patch number is increased, no new features are expected to have been introduced.