4D-STAR Development Guidelines

Adapted from Brock University, LAPELS, and NASA Development Standards

Version 0.0.1 (pre-approval)

4D-STAR Development Team

September 6, 2024

Contents

1 Introduction

As a multi-node and multi-developer project 4D-STAR requires a set of consistent and robust contribution guidelines. As of the August 2024 “coding spring” held at the Dartmouth College node a subset of the software development team along with three of the four PIs of the project have agreed to propose a development and contribution guide. This guide will lay out topics such as code style for various languages, issue tracking, pull request review, and git commit standards.

Any comments and critiques which the 4D-STAR team has relating to this file are welcome and can be incorporated before the adoption of these coding standards. Finally, this is and should be a living document. As we learn from the process of development we may decide to change or adopt new standard. This document will aim to lay out a process for suggesting changes to coding standards.

2 Standards Summary

This section provides a brief overview of the key standards highlighted in this guide. For more comprehensive details and examples, please refer to the relevant sections.

2.1 Project Tracking

- Centralized task management using the GitHub Projects board for 4D-ESTER Development.
- Add new tasks manually or link existing GitHub issues to the project board.
- Use GitHub Issues for managing tasks, tracking bugs, and communicating development progress.
- Create issues using the provided template and follow guidelines for managing and commenting on them.

2.2 Git Considerations

- Adhere to the branching strategy, including the use of main, feature, bugfix, hotfix, and release branches.
- Use Commitizen to create standardized and descriptive commit messages.

- Use pre-commit hooks or custom Git hooks to enforce linting and formatting before committing code.
- Maintain a .gitignore file to exclude unnecessary files from the repository.
- Use GitHub Actions to automate workflows, including CI/CD, linting, and deployment.

2.3 Code Review

- All code changes must be reviewed by at least one other developer before being merged into the main branch.
- Reviewers should use the provided checklist to evaluate code quality, adherence to standards, test coverage, efficiency, security, and commit history.
- Provide clear and constructive feedback in the pull request comments.
- Code should only be merged after approval from at least one reviewer.

2.4 Style Standards

- Follow the language-specific style guides for C, C++, Fortran, and Python, as outlined in this guide.
- Use consistent indentation, naming conventions, and comment styles.
- Document your code thoroughly using comments and docstrings.

Please consult the respective sections in this guide for further information and specific examples on each of these standards.

3 Project Tracking

In order to limit duplicated work we will make consisent and heavy use of github projects. Specifically, the project entitled `4D-ESTER Development` (https://github.com/orgs/4D-STAR/projects/1/) (hereafter "the project") will be the **centralized location for all outstanding tasks related to the development of the 4D version of ESTER**. The benefit of this is that development teams across nodes will be able to maintain a more consistent sense of the progress which other nodes are making. Further, by tracking issues in a centralized location priorities can be more easily compared across tasks.

3.1 Using the Project

The GitHub project board for 4D-ESTER Development is structured to allow for effective task management and collaboration across any node participating in software developmen. The project board currently offers two views:

- **Tasks (Kanban Board):** This view provides a column-based layout where tasks are organized by their status, ranging from "To Do" to "In Progress" and "Done." Each task is represented by a card, which can include details such as the assigned developer, due dates, and a brief description. Developers are expected to:
  - Regularly update the status of their tasks by moving the cards across columns as work progresses.
  - Provide clear and concise descriptions for each task, including relevant links or references to the codebase.
  - Assign themselves to tasks they are working on and unassign themselves if they are no longer working on a task.
  - Comment on the task cards to provide updates, request feedback, or report major issues which could block further progress on a task.
- **Timeline:** The timeline view presents tasks in chronological order, showing their start and end dates. The goal of this view is to understand the overall schedule of the project and to help ensure that deadlines are met. Developers should:
  - Ensure that all tasks have realistic start and end dates, which should be updated if project timelines shift.
  - Use this view to identify dependencies between tasks and coordinate with others to manage these.
  - Review the timeline regularly to make sure that the rate of progress is acceptable.

  This centralized approach to project management will help minimize duplication of effort across nodes as well as provide a authoritative source to cite when reporting on development progress.

3.1.1 Adding a Task

To ensure that all development activities are properly tracked, it is important to add tasks to the GitHub Projects board whenever new work is identified. Tasks can be added manually or linked directly to existing GitHub issues. Here’s how to do it:

1. **Manual Task Creation:**
   - Navigate to the GitHub Projects board and select the "Tasks" view.
   - Click on the “+ Add a card” button located at the bottom of any column (e.g., "To Do").
   - A new card will appear. Enter a concise title for the task that clearly describes the work to be done.
   - Optionally, click on the card to expand it and add more details, such as a description, due date, and assignee.
   - Once the task is created, move the card to the appropriate column that reflects its current status (e.g., "To Do").
2. **Linking a Task to a GitHub Issue:**
   - If the task corresponds to an existing GitHub issue, you can link it directly to the project board for better tracking and integration.
   - Start by opening the GitHub issue that you want to link.
   - On the right-hand side of the issue page, locate the "Projects" section.
   - Click the “+” button next to "Projects" and select the "4D-ESTER Development" project. This will automatically add the issue as a card on the project board.
   - The issue card will now appear on the board, and any updates made in the issue (e.g., comments, status changes) will be reflected on the card.
   - You can move the card across the columns on the project board just like a manually created task.

Linking tasks to GitHub issues provides additional benefits, such as easier tracking of discussions, automated status updates based on issue closures, and better integration with the overall development workflow. **Where possible tasks should be linked to GitHub issues.**

3.2 GitHub Issues

GitHub Issues are a powerful tool for managing tasks, tracking bugs, and coordinating development efforts within the 4D-ESTER project. They can serve as a primary means of communication between developers by helping to document work that needs to be done, reporting and resolving bugs, and proposing enhancements.Issues also provide a historical record of development decisions and discussions, making it easier to track progress, revisit previous work, and report to funding agencies progress which has been made.

3.2.1 Creating an Issue

Issues on GitHub are linked to a repository. In order to create an issue navigate the repository then in the top menu bar select “Issues” and then select “New Issue”. This will open the issue creation form. Note that issues are a GitHub feature, **not** a git feature. This means that generally you will not be able to track issues using the git command line tools. GitHub does provide a command line interface which allows for issue tracking from the command line.

- **Template Usage:** All issues should be created using the standardized issue template provided in the repository. This template includes sections for a clear description, steps to reproduce (if applicable), expected behavior, actual behavior, and any relevant screenshots or logs.
- **Title and Description:** Write a concise and descriptive title for the issue. In the description, provide as much detail as necessary to understand the issue or task, including relevant context, affected code, and references to related issues or pull requests.
- **Labels and Assignees:** Apply appropriate labels (e.g., bug, enhancement, documentation) to categorize the issue. If possible, assign the issue to a developer or a team (such as the Dartmouth-Node team) to ensure it is addressed promptly.
- **Milestones and Projects:** Whenever applicable, associate the issue with a milestone or project to link it to broader development goals. This helps in tracking the progress of larger features or releases.

3.2.2 Managing Issues

- **Prioritization:** Review and update the priority of issues regularly to ensure that critical tasks are addressed first. Use the priority labels (low, medium, high) to indicate the urgency of the issue.
- **Communication:** Use the comments section within the issue to provide updates,

ask questions, or discuss potential solutions. This ensures that all relevant information is centralized and accessible.

- **Closing Issues:** An issue should only be closed once it has been fully resolved. If the issue is linked to a pull request, it will automatically close when the pull request is merged. Ensure that the resolution is clearly documented in the issue comments before closing.

3.2.3 Examples of Issues

Below are a few examples of what various kinds of issues may look like. Keep in mind that when issues are submitted they should **always** be based on an existing issue template. If no issue template exists which captures the spirit of the issue being submitted then that should likely not be an issue (perhaps it is either too general or too specific?). If you believe it should be and no issue template exists then create an issue template following the format laid out in the 4D-Star/development-assets repository.

Bug Report Issue

Title: Application crashes when opening large opacity tables
Description: ESTER crashes when attempting to open opacity tables larger than 1GB. This issue was observed on both Linux and macOS environments, and it appears to be related to memory handling during the file loading process.
Steps to Reproduce:

1. Attempt to evolve a 1d model using an opacity table larger than 1GB (attached/linked).
2. Observe ESTER crashing without an error message.

Expected Behavior: ESTER should handle large opacity tables gracefully, either by loading the file without crashing or by displaying an appropriate error message if the file size exceeds the ESTERS capabilities or some user configured max size.
Actual Behavior: ESTER crashes immediately upon attempting to open a large opacity table.
Screenshots/Logs: *(Attach any relevant screenshots, logs, or datafiles here.)*
Environment:

- OS: macOS 13.3, Ubuntu 22.04
- Version: 4D-ESTER v0.0.1a
- SDK Version: 4D-ESTER SDK v0.0.3a
- Memory: 16GB RAM

Labels: *bug*, *high priority*
Assignee(s):

- John Doe

Feature Request Issue

Title: Add support for saving ESTER models as csv
Description: csv files are generally easier for community members to quickly use. While the h5 file format maintains a high degree of flexibility it would be useful to also be able to export evolutionary tracks as csv files to allow for easier community engagement.
Use Case:

- non computer focused astronomers use of ESTER
- class / student use of ESTER

Potential Risks: CSV files will, for the same amount of underlying data, tend to be larger as all information has to be encoded via text. Further, many of the memory streaming benefits and hierarchical benefits of h5 are lost. This is acceptable as I do not propose to remove h5 output, simply to supplement it for community members who are not comfortable interfacing with h5 files.
Labels: *enhancment*, *low priority*
Assignee(s):

- Jane Doe

Pull Request

Title: Refactor memory management in the opacity table loading function
Description: This pull request refactors the memory management in the opacity table loading function to address issues with handling large opacity tables. The changes improve stability and prevent crashes when loading tables over 1GB in size.
Linked Issues:

- Fixes #23: Application crashes when opening large opacity tables

Changes Made:

- Introduced a buffer system to handle large files in smaller chunks. Updated error handling to provide feedback if a file is too large to load.
- Added unit tests to verify the changes.

Testing:

- Successfully tested loading opacity tables of various sizes, including those larger than 1GB.
- Passed all existing unit tests.
- Ran performance benchmarks to ensure no significant impact on opacity loading.

Labels: *refactor*, *high priority*
Reviewer: Open for review by any team member

3.2.4 Commenting On Issues

Commenting on GitHub Issues is essential to issues being a useful tool. Comments provide a way to discuss the issue, clarify details, and track progress. Some guidelines to ensure comments are clear, constructive, and contribute positively to the project follow:

1. Why You Should Comment
   - **Clarify Details**: If you have questions about the issue or need further clarification, leaving a comment is the best way to get the information you need.
   - **Provide Updates**: Use comments to update the development-team on your progress, any challenges you encounter, or if you've resolved the issue. Comments on issues are easier to trace and conglomerate than slack direct-messages.
   - **Suggest Solutions**: If you have a potential solution or workaround, share it in the comments. This can both help the issue be resolved and also provide a ledger of solutions that have been tried in the past.

- **Request Feedback**: If you need feedback on an approach or decision, a comment can initiate that discussion.
- **Acknowledge Contributions**: Use comments to acknowledge other team members' suggestions or contributions.

2. How to Reference Other Issues
    - **Linking to Issues**: When referencing another issue, always use the # followed by the issue number (e.g., #123). This automatically creates a hyperlink to the referenced issue, making it easy for others to navigate between related discussions.
    - **Describing Relationships**: If the issue you're referencing is closely related or dependent on the current issue, describe the relationship in your comment. For example, "This issue is related to #456, where we discussed a similar problem with the spectral methods."
    - **Cross-Referencing**: If your comment is relevant to multiple issues, cross-reference them by listing all applicable issue numbers (e.g., "This solution also addresses concerns raised in #789 and #1011").
3. How Comments Should Be Written
    - **Be Clear and Concise**: Keep your comments focused and to the point. Use bullet points or numbered lists if you're covering multiple points.
    - **Use Proper Grammar and Spelling**: Write comments with the same care you would give to any other part of your work.
    - **Stay Professional and Respectful**: Always maintain a professional tone, and be respectful of others' ideas and contributions. Constructive feedback is encouraged, but it should be given in a way that is supportive and helpful.
    - **Format Code and Links**: If you need to include code snippets in your comments, use backticks for inline code or triple backticks followed by the name of the language for blocks of code. For example ``` ``This is inline code`` ```.
    - **Summarize Long Discussions**: If a comment thread becomes long or complex, periodically summarize key points or decisions to keep the conversation on track and make it easier for others to catch up.
4. Best Practices

- **Comment Frequently**: Regular comments help keep the development-team informed and engaged. Even if it's a simple update, it's worth sharing.
- **Use Mentions Sparingly**: Use @username mentions to notify specific team members when their input is needed, but avoid overusing mentions as it can lead to notification fatigue.
- **Keep the Conversation on Topic**: Ensure that your comments are relevant to the issue at hand. If a discussion veers off-topic, consider opening a new issue or starting a conversation in another appropriate channel.
- **Resolve Conflicts**: If there's a disagreement in the comments, aim to resolve it constructively. Acknowledge different viewpoints and work towards a consensus or a compromise.
- **Document Decisions**: If a decision is made as a result of a comment thread, summarize it clearly in the comments and consider updating the issue description or linked documentation to reflect that decision.

4 Other Git Considerations

In addition to making use of GitHub Projects and Issues, as a development team we must follow consistent and standardized rules for using Git at large. This section lays out the guidelines for branches, commits, .gitignore files, and GitHub Actions.

4.1 Branches

To maintain a clean and organized codebase, we follow a structured branching strategy:

- **Main Branch (main)**: This branch should always contain stable, production-ready code. Direct commits to the main branch are prohibited; all changes must be introduced through pull requests.
- **Feature Branches (feature/)**: Use feature branches for developing new features or significant changes. Name the branch descriptively, prefixed with feature/ (e.g., feature/add-csv-export).
- **Bugfix Branches (bugfix/)**: For fixing bug, use branches prefixed with bugfix/. Ensure the branch name clearly indicates the nature of the bug (e.g., bugfix/fix-crash-large-files).

- **Hotfix Branches (hotfix/)**: For urgent fixes to production code, use hotfix/ branches (e.g., hotfix/security-patch).
- **Release Branches (release/)**: When preparing for a new release, create a release/ branch to finalize features, bug fixes, and documentation for that release (e.g., release/v1.0.0).
- **Naming Conventions**: Always use lowercase with hyphens (-) to separate words. Avoid using spaces or underscores.

4.1.1 Branching Workflow:

1. Create a Branch: Before starting new work, create a branch from main.

```
$ git checkout -b bugfix/opal-null-pointer
$ git add src/opal/load
$ cz c
$ git push -u origin bugfix/opal-null-pointer
```

N.B: You may notice in the above example use of cz c instead of git commit -m ... See Section **??** for more details on what this is and why we use commitizen (which brings the cz command) instead of raw git commits.

2. Work on the Branch: Commit your changes to the branch as you develop. Push the Branch: Once your work is ready, push the branch to the remote repository.

```
$ git push origin feature/your-feature-name
```

3. Submit a Pull Request: After pushing, submit a pull request to merge your branch into main.

4.1.2 Checking Out Branches

Checking out a remote branch will be a task that is somewhat common. Below is a breif example of how one might do this if there is a branch on the 4D-ester repository called feature/add-mesa-rates

```
git fetch origin
git checkout -b feature/add-mesa-rates origin/feature/add-mesa-rates
```

4.2 Commits

Commits should be atomic and descriptive, with each commit representing a single logical change. This means that when commiting a change add only the files related to that commit message. If you have changed multiple, parts of the code base since your last commit this should result in multiple commits instead of one large commit. It is important that we avoid commit messages of the form "Bug Fix" or "Made Changes". We use Commitizen to enforce a standardized format for commit messages, which enhances readability and consistency.

4.2.1 Commit Process

1. **Install Commitizen**: Ensure Commitizen is installed in your project. If it is not already installed it can be installed using the node package manager.

   ```
   $ npm install -g commitizen
   ```

2. **Add and Commit Changes**: Instead of using git commit, use cz to commit your changes.

   ```
   $ git add <path/to/file/a> <path/to/file/b> ...
   $ cz c
   ```

3. **Commitizen Prompts**: Commitizen will prompt you to provide details for the commit message, such as type, scope, and a brief description. Below is a quick example of what walking through these prompts might look like

```
? Select the type of change you are committing
   fix: A bug fix. Correlates with PATCH in SemVer
 » feat: A new feature. Correlates with MINOR in SemVer
   docs: Documentation only changes
   style: Changes that don't affect the meaning of the code
   refactor: Neither fixing a bug nor adding a feature
   perf: A code change that improves performance
   test: Adding missing or correcting existing tests
   build: Build system or external dependencies changes
   ci: Changes to CI configuration files and scripts
```

```
? Select the type of change you are committing feat: A new
feature. Correlates with MINOR in SemVer
? What is the scope of this change? (class or file name):
(press [enter] to skip)
 neu/calc.c
```

```
? Select the type of change you are committing feat: A new
feature. Correlates with MINOR in SemVer
? What is the scope of this change? (class or file name):
(press [enter] to skip)
 neu/calc.c
? Write a short and imperative summary of the code changes:
(lower case and no period)
 fixed log / ln error
```

```
? Select the type of change you are committing feat: A new
feature. Correlates with MINOR in SemVer
? What is the scope of this change? (class or file name):
(press [enter] to skip)
 neu/calc.c
? Write a short and imperative summary of the code changes:
(lower case and no period)
 fixed log / ln error
? Provide additional contextual information about the code
changes: (press [enter] to skip)
 In neu/calc.c in the calc_rates function the log base 10
of the density was being used. However, the tabulated data
is a function of natural log of density. This was causing
the interpolation routine to go outside of the bounds of
the table.
```

```
? Select the type of change you are committing feat: A new
feature. Correlates with MINOR in SemVer
? What is the scope of this change? (class or file name):
(press [enter] to skip)
 neu/calc.c
? Write a short and imperative summary of the code changes:
(lower case and no period)
 fixed log / ln error
? Provide additional contextual information about the code
changes: (press [enter] to skip)
 In neu/calc.c in the calc_rates function the log base 10
of the density was being used. However, the tabulated data
is a function of natural log of density. This was causing
the interpolation routine to go outside of the bounds of
the table.
? Is this a BREAKING CHANGE? Correlates with MAJOR in SemVer
No
```

```
? Select the type of change you are committing feat: A new
feature. Correlates with MINOR in SemVer
? What is the scope of this change? (class or file name):
(press [enter] to skip)
 neu/calc.c
? Write a short and imperative summary of the code changes:
(lower case and no period)
 fixed log / ln error
? Provide additional contextual information about the code
changes: (press [enter] to skip)
 In neu/calc.c in the calc_rates function the log base 10
of the density was being used. However, the tabulated data
is a function of natural log of density. This was causing
the interpolation routine to go outside of the bounds of
the table.
?  Is this a BREAKING CHANGE? Correlates with MAJOR in
SemVer No
fix(neu/calc.c): fixed log / ln error

In neu/calc.c in the calc_rates function the log base 10
of the density was being used. However, the tabulated data
is a function of nautral log of density. This was causing
the interpolation routine to go outside of the bounds of
the table.


[dev 1714e72] fix(neu/calc.c): fixed log / ln error
1 file changed, 1 insertion(+), 2 deletions(-)

Commit successful!
```

4. **Pre-commit Hooks**: Linting and Formatting: To maintain code quality, set up Git hooks to run linters automatically before each commit. This can be done using pre-commit

   - First, if not already installed, install pre-commit. This can be done with the python package manager

```
$ pip install pre-commit
```

- Now we need to tell pre-commit what to to (hooks) before each commit. This is done by adding a .pre-commit-hooks.yaml file to the root of your git repo. in developer-assets there are premade .pre-commit-hooks.yaml files for various languages (C, C++, Fortran, and Python). There is also a unified .pre-commit-hooks.yaml file. In general all you should have to do is copy the unified .pre-commit-hooks.yaml file to your repository. The structure of these files looks like

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.3.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml

  - repo: https://github.com/pre-commit/mirrors-clang-format
    rev: v13.0.0
    hooks:
      - id: clang-format
        args: [--style=file]

  - repo: https://github.com/pycqa/flake8
    rev: v4.0.1
    hooks:
      - id: flake8
        additional_dependencies: [flake8]

  - repo: local
    hooks:
      - id: run-make
        name: Run Makefile
        entry: make
        language: system
        files: \.(c|cpp|h|f|f90)$
        stages: [commit]
```

5. This configuration does the following:
    - **Trailing Whitespace, End-of-File Fixer, Check YAML**: Basic hooks to clean up common formatting issues.
    - **Clang-Format**: Formats C/C++ code according to the style specified in a .clang-format file.
    - **Flake8**: Lints Python code.
    - Run Make: Runs the Makefile to ensure the build is successful before allowing a commit.
6. **Install the Git Hooks**: Run the following command to install the hooks defined in your .pre-commit-config.yaml file

```
$ pre-commit install
```

7. **Running the Hooks**: The hooks will now automatically run every time you attempt to commit code. If any issues are found, the commit will be blocked until they are resolved.
8. **Manually Running Hooks**: You can also run the hooks manually on all files:

```
$ pre-commit run --all-files
```

4.2.2 Custom Git Hooks

Alternatively, you can write your own custom Git hooks directly in the .git/hooks/ directory:

- **Create a Hook Script**: Create a script in the .git/hooks/ directory, such as pre-commit, and make it executable:

Custom git hook

```
#!/bin/sh
make
clang-format -i *.c *.cpp *.h *.f *.f90
flake8 .
```

- **Make the Script Executable**:

  ```
  $ chmod +x .git/hooks/pre-commit
  ```

- **Automate Linting and Testing**: The script can include commands to run linters (like clang-format for C/C++ or flake8 for Python) and tests. If any of these steps fail, the commit will be blocked.

9. **Running Tests**: Ensure that all tests pass before committing.

4.3 .gitignore

The .gitignore file is essential for keeping unnecessary files out of the repository. It specifies which files and directories Git should ignore.

4.3.1 Best Practices for .gitignore

- **Ignore System Files**: Files generated by your operating system, such as .DS_Store (macOS) or Thumbs.db (Windows).
- **Ignore Build Artifacts**: Compiled files, object files, and other build artifacts should be excluded (e.g., *.o, *.exe, build/).
- **Ignore Dependencies**: If your project includes external dependencies, such as Python's __pycache__/, Node.js's node_modules/, or compiled binaries, ensure these are ignored.
- **Ignore Environment Configs**: Local environment configuration files like .env or secrets.json should be excluded from version control. Sample .gitignore for a C/C++/Python/Fortran Project:

Example .gitignore File

```
# Operating system files
.DS_Store
Thumbs.db

# Python
__pycache__/
*.pyc
.venv/
.env

# C/C++
*.o
*.exe
build/
cmake-build-debug/

# Fortran
*.mod
*.out
*.lst

# Logs c.log

# Temporary files
*.tmp
*.swp
```

4.4 Actions

GitHub Actions provide automated workflows that can be triggered by events in your repository, such as pushing code, opening pull requests, or scheduling regular tasks. These workflows ensure that testing standards laid out in this document are automatically applied to all development work.

4.4.1 Setting Up GitHub Actions

- **Creating a Workflow File**: Workflow files are stored in the .github/workflows/ directory of your repository. Each file defines a set of jobs and steps to be executed when triggered. Example Workflow for CI/CD:

Example Workflow for CI/CD

```yaml
name: CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.x'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run linting
        run: |
          flake8 .

      - name: Run tests
        run: |
          pytest
```

- **Linting and Formatting**: Use GitHub Actions to automatically lint and format your code on each push or pull request.
- **Continuous Integration (CI)**: Set up workflows that automatically run tests when-

ever code is pushed or a pull request is created. This ensures that the codebase remains stable and functional.

- **Deployment**: Automate deployment to production or staging environments using GitHub Actions. This can include building Docker images, uploading assets to a cloud service, or triggering a deployment script.

4.4.2 Best Practices for Actions

- **Modularize Workflows**: Break down complex workflows into smaller, reusable components. Use reusable workflows or composite actions to manage common tasks.
- **Secure Secrets**: Store sensitive information such as API keys or passwords in GitHub Secrets, and reference them securely in your workflows.
- **Monitor Workflow Runs**: Regularly monitor the outcomes of your workflows. Set up notifications for failed runs to ensure prompt action.

5 Code Review

Code review is a critical component of the development workflow, ensuring that code quality remains high, best practices are followed, and knowledge is shared across the team. This section outlines the standards and procedures for conducting code reviews within the 4D-ESTER project. Generally what this means is that **no development should take place on any main branches; rather, development should take place in development branches**.

5.1 Purpose of Code Review

- **Ensure Code Quality**: Code reviews help catch bugs, performance issues, and other potential problems before code is merged into the main branch.
- **Enforce Coding Standards**: Reviews ensure that all code adheres to the coding standards laid out in this document, maintaining consistency across the codebase.
- **Knowledge Sharing**: Code reviews are an opportunity for those at different nodes to learn from each other, share best practices, and improve their own coding skills.

5.2 Standards for Code Review

- **Adherence to Coding Guidelines**: Ensure that the code follows the coding standards outlined in this document, including naming conventions, code structure, and documentation.
- **Test Coverage**: Verify that the code is adequately tested. All new features or bug fixes must be accompanied by appropriate unit tests, and all tests should pass on the Continuous Integration (CI) system.
- **Code Efficiency**: Review the code for efficiency in terms of performance and resource usage. Ensure that the code does not introduce unnecessary complexity or slowdowns.
- **Readability and Maintainability**: The code should be easy to understand and maintain. Check for clear naming, proper use of comments, and adherence to the DRY (Don't Repeat Yourself) principle.
- **Error Traceability** Ensure that the newly introduced code handles errors gracefully and in a manner which is interoperable with the overall error handling strategy of 4D-ESTER.
- **Compliance with Commit Standards**: Ensure that commit messages follow the standard format enforced by Commitizen, and that the commits are logical and well-organized.

5.3 How Code Review Will Work

- **Assigning a Reviewer**: When a developer submits a pull request (PR), any other team member can pick it up for review. The reviewer should not be the same person who wrote the code. The developer may use slack to notify the development-team that a pull request has been submitted.
- **Review Checklist**: Reviewers should use the following checklist to guide their review:
  - Is the code functional and free of obvious bugs?
  - Does the code adhere to 4D-STAR's coding standards?
  - Are all new features or fixes adequately tested?
  - Is the code efficient and maintainable?

    - Are security best practices followed?
    - Is the commit history clean and well-organized?

  There is a template in developer-assets which should be used by code reviewers as a base and includes a checklist of items which a pull request must pass in order to be merged.

- **Providing Feedback**: Reviewers should provide clear, constructive feedback in the PR comments. Highlight both positive aspects of the code and areas that need improvement. If significant changes are required, the reviewer should clearly explain why and suggest alternatives.
- **Approval and Merging**: Once the reviewer is satisfied that the code meets all standards, they can approve the PR. The code should only be merged into the main branch after it has been approved by at least one reviewer. If there are multiple significant changes requested, a re-review should be conducted after revisions are made.
- **Rejecting a PR**: If the code has significant issues that cannot be resolved within the scope of the current PR, the reviewer may reject the PR with a detailed explanation. The developer can then revise the code and submit a new PR.
- **Reviewing in a Timely Manner**: Reviews should be conducted promptly to avoid bottlenecks in the development process. Reviewers should aim to provide initial feedback within one week of the PR being submitted.

5.4 Best Practices for Code Review

- **Be Respectful and Constructive**: Approach the review process as a collaborative effort. Always be respectful in your feedback.
- **Focus on the Code, Not the Developer**: Reviews should be about the code, not the person who wrote it. Keep your comments objective and focused on the specific issues or improvements.
- **Keep the Scope in Mind**: Review the code within the context of the task it was intended to solve. Avoid getting sidetracked by unrelated issues.
- **Use the Pull Request Template**: Ensure that the PR template (if available) is fully filled out. This includes providing a clear description, linking to relevant issues, and detailing the changes made.

- **Continuous Improvement**: Use the review process to identify patterns or recurring issues in the codebase. If the same issues keep coming up, consider updating the this document to address these issues.

5.5 Post-Review Process

- **Merging the Code**: Once the code has been reviewed and approved, it can be merged into the main branch. Ensure that all automated tests pass on the CI system before merging.
- **Documentation**: If the code introduces new features or changes existing functionality, ensure that relevant documentation is updated accordingly.
- **Retrospective**: Periodically review the code review process itself to identify areas for improvement.

6 Style Standards

Consistent code style is essential for maintaining a readable and maintainable codebase, especially in a multi-developer, multi-language project like 4D-ESTER. This section outlines the style standards for each of the languages used in our project: C, C++, Fortran, and Python. Adherence to these standards will ensure that our code remains consistent, understandable, and easy to review.

6.1 General Style Standards

- Comment code in a way which describes why something is being done not what is being done
- Include docstrings on all functions, classes, subroutines, modules, structs, etc...
- docstrings should **all** be doxygen compliant.
- Use double precision floating point numbers (double in c, real*8 in fortran, and the default float type in python)
- Document code as you write it
- Do not use tabs

- Use reasonable and descriptive variable names. You should be able to tell what a variable is by reading its name.
- As much as possible try to limit all lines to a maximum of 80 characters.
- **Version Control**: Commit often, with clear, descriptive messages. Follow the commit message conventions outlined in the "Commits" section. Use branches to develop features or fix bugs, and always submit changes through pull requests.
- **Documentation**: Document your code comprehensively using comments and docstrings. All docstrings should be copatible with doxygen so that we can maintain a unified API reference. Maintain an up-to-date README.md and other relevant documentation files in the project root.
- **Security**: While security may not be at the forefront of our minds as astronomer's we should be aware that there has been conversation about web interfaces to ESTER and the final programs produced by 4D-STAR. We do not need to overly focus on code security; however, we do need to be aware, at least at a low level, of security concerns such as validating input files, and proper memory management.
- **Profiling**: Profile code for performance and memory usage regularly or as part of CI/CD so that we have a trace of where improvements may be made. Contribute profiles to the profiling database when that is operational.

6.2 C Style Standards

- **Indentation**: Use 4 spaces per indentation level. Do not use tabs.
- **Braces**: Use the K&R style for braces (brace styles):

Braces Example

```c
if (condition) {
    // Code
}
else {
    // Code
}
```

- **Naming Conventions**:
  - *Variables*: Use snake_case for variable names (e.g., total_count).

  - *Functions*: Use snake_case for function names (e.g., calculate_sum).
  - *Constants*: Use UPPER_CASE with underscores for constants (e.g., MAX_BUFFER_SIZE).
  - *Global Variables*: Prefix with g_ to indicate they are global (e.g., g_error_code).

- **Comments**:

  Use /* ... */ for block comments and // for single-line comments. Place comments above the code they refer to, and use them to explain why something is being done, not just what is being done.

  Single Line Comments

  ```c
  // Initialize the counter
  int counter = 0;
  ```

- **Docstrings**: Doxygen-style comments are used to generate documentation from annotated source code. These comments provide descriptions for functions, classes, parameters, return values, and other relevant information. Place the Doxygen comments directly above the code they refer to. Docstrings should be given in the header file and not in the implimentation file.

  Docstrings

  ```c
  /**
   * @brief Adds two integers.
   *
   * This function takes two integers and returns their sum.
   *
   * @param a The first integer.
   * @param b The second integer.
   * @return The sum of a and b.
   */
  int add(int a, int b) {
      return a + b;
  }
  ```

- **Functions**: Functions should have a single responsibility and be kept short. Function prototypes should be declared in header files (.h). Always check return values, especially for functions that might fail (e.g., malloc).

- **Header Files**: Use include guards to prevent multiple inclusions. Guards should use the format FILENAME_H (all caps)

Include Guards

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Declarations

#endif // MY_HEADER_H
```

- **Pointer Usage**: Place the asterisk next to the variable name (not the type deceleration) in pointer declarations

Pointer Decleration

```
int *ptr;
```

- **File Organization**: Each source file should have an associated header file. Implementation goes in .c files, and declarations go in .h files. Implementation and declaration files should have the same file name aside from the extension.

6.3 C++ Style Standards

Many Of the C++ style standards will be the same as the C style standards.

- **Indentation**: Use 4 spaces per indentation level. Avoid using tabs.
- **Braces**: Use the K&R style for braces, similar to C:
- **Naming Conventions**:
  - *Variables*: Use camelCase for variable names (e.g., totalCount).
  - *Functions*: Use camelCase for function names (e.g., calculateSum).
  - *Classes*: Use PascalCase for class names (e.g., DataProcessor).
  - *Constants*: Use UPPER_CASE with underscores for constants (e.g., MAX_BUFFER_SIZE).
  - *Member Variables*: Prefix with m_ (e.g., m_bufferSize).
- **Comments**: Use // for single-line comments and /* ... */ for multi-line comments. Document all public classes and methods with Doxygen-compatible comments. See C style reference for Doxygen-compatible format.

- **Classes**:
  - *Encapsulation*: Keep data members private, provide access through getter and setter methods.
  - *Constructors and Destructors*: Always define a default constructor and destructor. Use member initialization lists where possible.
  - *Inheritance*: Prefer composition over inheritance. Use virtual only when necessary, and mark overridden methods with override.
- **Functions**: Functions should be small and perform a single task. Prefer using references over pointers when passing objects to functions. Mark functions as const if they do not modify the object's state.
- **Header Files**: Use include guards or #pragma once to prevent multiple inclusions. Place class declarations in header files (.h) and implementations in source files (.cpp).
- **Templates**: Template definitions should be placed in header files. Use templates judiciously to avoid code bloat.
- **STL and Modern C++**: Prefer the use of STL containers (std::vector, std::map) over raw arrays and custom data structures. Use modern C++ features like auto, range-based for loops, and smart pointers (std::unique_ptr, std::shared_ptr) to manage resources.

6.4 Fortran Style Standards

- **Indentation**: Use 2 spaces per indentation level. Avoid using tabs.
- **Naming Conventions**:
  - *Variables*: Use camelCase for variable names (e.g., totalCount).
  - *Subroutines and Functions*: Use camelCase for subroutine and function names (e.g., calculateSum).
  - *Modules*: Use PascalCase for module names (e.g., DataProcessing).
  - *Constants*: Use UPPER_CASE for constants (e.g., MAX_BUFFER_SIZE).
- **Comments**: Use ! for comments. Place comments on a separate line above the code they describe.

Fortran Single Line Comment

```fortran
! Initialize the counter
integer :: counter = 0
```

- **Docstrings** Fortran supports Doxygen-compatible comments to document modules, subroutines, and functions. These comments help clarify the purpose and usage of the code, making it easier for other developers to understand and use.

Fortran Docstring

```fortran
!> @brief Computes the factorial of a number.
!!
!! Calculates the factorial of a given integer n.
!!
!! @param n The integer whose factorial is to be computed.
!! @return The factorial of n.
integer function factorial(n)
    integer, intent(in) :: n
    integer :: i

    factorial = 1
    do i = 2, n
        factorial = factorial * i
    end do
end function factorial
```

- **Modules**: Use modules to encapsulate related procedures and data. This promotes reusability and cleaner code organization. Always declare the intent (in, out, inout) for subroutine arguments.
- **Subroutines and Functions**:
  - Subroutines should perform a single, well-defined task.
  - Functions should be side-effect-free and used to return values.
  - Use implicit none in all subroutines, functions, and modules to prevent the use of undeclared variables.
- **Precision**:

Fortran Types

```fortran
integer, parameter :: dp = kind(1.0d0)
real(dp) :: myVariable
```

- **Error Handling**: Implement error handling in subroutines and functions, returning appropriate status codes and error messages.

6.5 Python Style Standards

In general we will follow the PEP 8 style guide

- **Indentation**: Use 4 spaces per indentation level. Do not use tabs.
- **Naming Conventions**:
  - *Variables and Functions*: Use snake_case for variables and function names (e.g., calculate_sum).
  - *Classes*: Use PascalCase for class names (e.g., DataProcessor).
  - *Constants*: Use UPPER_CASE with underscores for constants (e.g., MAX_BUFFER_SIZE).
- **Comments and Docstrings**:
  - Use # for single-line comments. Place comments on a new line above the code they describe.
  - Use triple double quotes (""" ... """) for docstrings in modules, classes, and functions. The first line of a docstring should be a short description, followed by a more detailed explanation if necessary.
  - We will use doxygen for python documentation despite the fact that Sphinx is generally more popular in python development environments. Given that we are already using Doxygen for C, C++, and Fortran documentation it makes sense to stick with it for python so that we have a consistent documentation system. Because of this, python functions, modules, and classes should use Doxygen style docstrings.

Example Docstring

```
def read_ester_model_as_pandas(path : str): -> "pd.DataFrame"
"""
@brief read an ester model as a dataframe.

Given some path to a ester h5 model read it and return it as
    a pandas dataframe

@param path (str). Path to h5 file
@return Dataframe representation of the ester model
@raises FileNotFoundError if the path does not exist
@raises EsterBadModelFileError If the h5 file is poorly
        formatted

@example
>>> df = read_ester_model_as_pandas("M5.m5")
"""
```

- **Imports**:
  - Group imports into three sections: standard library imports, third-party imports, and local imports. Separate each group with a blank line.
  - Use absolute imports instead of relative imports.
  - Import only what you need (e.g., from module import ClassName).
- **Functions and Methods**:
  - Keep functions small and focused on a single task.
  - Use type hints for function arguments and return types (e.g., def add(a: int, b: int) -> int:).
  - Make **heavy** use of the typing module to hint at more complex return types such as Tuples or optional returns.
  - Use default arguments instead of overloading functions.
- **Classes**:
  - Use class methods (@classmethod) and static methods (@staticmethod) where appropriate.

  - Use properties (@property) instead of getter and setter methods to provide a more Pythonic interface.

- **Error Handling**:
  - Use exceptions for error handling. Catch specific exceptions rather than using a generic except clause.
  - Use built in exceptions as much as possible.
  - Raise custom exceptions only where more program specific information is actually useful.
  - Prefer explicit error catching to implicit error catching (i.e. as much as possible use conditional statements to check conditions and if they are not met raise error as opposed to try...except blocks)

- **List Comprehensions and Generators**: Prefer list comprehensions and generator expressions for creating new lists or iterating over sequences.

List Comprehension

```
squares = [x**2 for x in range(10)]
```

- **Testing**: Write unit tests for all functions and methods. Use pytest for running tests. Ensure that your tests cover a wide range of input scenarios, including edge cases.

- **Virtual Environments**: Always use a virtual environment to manage dependencies for your project. This ensures that your project remains isolated and dependencies do not conflict with other projects.

```
$ python -m venv venv
$ source venv/bin/activate
```

- **Logging**: Use the logging module for logging information, rather than printing to stdout. This allows for better control over log output and is more suitable for production environments.

Logging Module

```python
import logging
from ester4d import get_logger

logger = get_logger()

logger.info("This is an info message")
logger.error("This is an error message")
logger.evolve("This is a custom level added for evolution info")
```

- **Code Organization**:
  - Organize your code into modules and packages. Each module should have a clear purpose, and related modules should be grouped into packages.
  - Place unit tests in a tests/ directory at the root of the project, with a clear structure that mirrors the main codebase.

7 Testing Procedures and Standards

Effective testing is essential for ensuring the reliability, performance, and maintainability of the software developed within the 4D-ESTER project. This section outlines the proper testing procedures, standard tools to use for each language in the project, and additional testing considerations that should be integrated from the start.

7.1 General Testing Principles

Testing should be a fundamental part of the development process, not an afterthought. The following principles apply to all languages used in the project:

- **Test Early and Often:** Begin writing tests as soon as development starts. Continuously integrate and run tests to catch issues early in the development cycle.
- **Automate Testing:** Use automated testing tools and frameworks to run tests frequently and consistently, ideally with every commit or pull request.
- **Test Coverage:** Aim for high test coverage, particularly for critical code paths. Ensure that both normal operation and edge cases are tested.

- **Test Types:** Include a variety of test types, such as unit tests, integration tests, and system tests, to comprehensively assess the software.
- **Continuous Integration (CI):** Integrate testing into the CI pipeline. Ensure that tests are run automatically for every commit, with results reported promptly to developers.

7.2 C Testing Procedures and Tools

For C development, testing should be structured and automated using the following tools and techniques:

- **Unit Testing:**
  - Use `CMocka` or `Unity` for unit testing. These lightweight frameworks are designed for C and provide easy-to-use APIs for creating and running unit tests.
  - Structure tests to cover all functions, particularly those with complex logic. Ensure that edge cases and error conditions are tested.
- **Integration Testing:**
  - Use `CTest`, the testing component of CMake, for integration testing. It allows you to organize and run a suite of tests and integrates well with other CMake-based projects.
  - Integration tests should focus on the interaction between different modules or components, ensuring that they work together as expected.
- **Static Analysis:**
  - Employ `Cppcheck` and `Clang Static Analyzer` to perform static code analysis, identifying potential bugs, memory leaks, and undefined behaviors.
  - Incorporate static analysis into the CI pipeline to ensure that all commits are checked for common issues.
- **Code Coverage:**
  - Use `gcov` in combination with `lcov` to measure test coverage. Aim for as close to 100% coverage as feasible, focusing on critical code paths.
  - Integrate code coverage reports with the CI pipeline, providing visibility into areas of the codebase that may need more testing.

7.3 C++ Testing Procedures and Tools

C++ shares many testing practices with C but offers additional tools and frameworks:

- **Unit Testing:**
  - Use `Google Test (gtest)` or `Catch2` for unit testing. Both frameworks are widely used in the C++ community and offer extensive features for writing and organizing tests.
  - Write unit tests to cover all classes and functions, ensuring that both expected and unexpected inputs are tested.
- **Integration Testing:**
  - Use `CTest` in conjunction with CMake for running integration tests. This allows for seamless integration with other parts of the build system.
  - Focus on testing the interactions between different components, especially those involving complex inheritance or polymorphism.
- **Static Analysis:**
  - Utilize `Cppcheck`, `Clang Static Analyzer`, and `SonarQube` for static code analysis. These tools help detect common issues, potential bugs, and performance bottlenecks.
  - Regularly run static analysis as part of the CI pipeline to catch issues early in the development process.
- **Code Coverage:**
  - Use `gcov`, `lcov`, or `Codecov` for measuring code coverage. Ensure that tests cover all major code paths, especially in classes with complex logic.
  - Include coverage reports in the CI pipeline to monitor coverage trends and identify areas that need more testing.
- **Memory and Performance Profiling:**
  - Use tools like `Valgrind` for memory profiling to detect memory leaks and errors. For performance profiling, consider using `gprof` or `perf`.
  - Integrate memory and performance profiling into the testing process, especially for performance-critical sections of code.

7.4 Fortran Testing Procedures and Tools

For Fortran, testing can be more challenging due to the language's older ecosystem, but there are effective tools available:

- **Unit Testing:**
  - Use `FortranTest` or `fUnit` for unit testing. These frameworks provide a structured way to write and run tests for Fortran code.
  - Focus on testing mathematical functions, array manipulations, and module interactions to ensure correctness and performance.
- **Integration Testing:**
  - Integration tests should verify that different Fortran modules work together as expected. Given the scientific nature of most Fortran applications, focus on ensuring that numerical results are accurate and consistent.
  - Use custom scripts or tools like `CTest` (with CMake integration) to manage and run integration tests.
- **Static Analysis:**
  - Use `Forcheck` or `Flint` for static analysis. These tools help identify potential issues in Fortran code, such as uninitialized variables and array bounds errors.
  - Static analysis should be part of the regular testing routine, especially for legacy code or when making significant changes.
- **Code Coverage:**
  - Use tools like `GCOVR` to measure code coverage in Fortran programs. Strive to cover all critical code paths, particularly in modules that perform complex computations.
  - Integrate code coverage tools with CI pipelines to ensure that coverage levels are maintained as the project evolves.
- **Numerical Accuracy Testing:**
  - Fortran is often used in numerical simulations and scientific computing, making numerical accuracy critical. Implement tests that compare computed results against known benchmarks or analytical solutions.

- Consider using relative and absolute tolerance thresholds in tests to account for floating-point arithmetic precision.

7.5 Python Testing Procedures and Tools

Python is known for its strong testing culture and has a variety of tools available:

- **Unit Testing:**
  - Use `unittest` (built-in), `pytest`, or `nose2` for unit testing. `pytest` is particularly popular due to its simplicity and powerful features.
  - Write unit tests for all functions and methods, ensuring that edge cases and error conditions are thoroughly tested.
- **Integration Testing:**
  - Integration tests should be written to verify that different modules and components work together as expected. Use `pytest` for its easy integration with fixtures and setup/teardown mechanisms.
  - Focus on testing the interactions between components, especially when dealing with external dependencies like databases or APIs.
- **Static Analysis:**
  - Use `flake8`, `pylint`, or `mypy` for static code analysis. These tools help enforce coding standards, identify potential bugs, and ensure type correctness.
  - Integrate static analysis tools into the CI pipeline to ensure code quality is maintained across all commits.
- **Code Coverage:**
  - Use `coverage.py` to measure test coverage. Aim for comprehensive coverage, particularly in modules with complex logic or critical functionality.
  - Include coverage reports in the CI pipeline and set coverage thresholds to ensure that new code does not decrease overall coverage.
- **Test Automation:**
  - Use `tox` to automate testing across multiple Python versions and environments. `tox` ensures that the code runs consistently across different setups.

  - Integrate `tox` with CI systems to automate testing, linting, and coverage reporting in one streamlined process.

7.6 Other Testing Considerations

Beyond the tools and procedures specific to each language, there are additional considerations that should be integrated into the testing strategy from the start:

- **Continuous Integration (CI):**
  - Integrate all testing tools and procedures with GitHub Actions. The CI system should run all tests automatically on each commit or pull request, providing immediate feedback to developers. Should we do this on every branch or just main?
  - Ensure that the CI pipeline includes steps for static analysis, unit tests, integration tests, code coverage, and any other relevant checks.
- **Documentation of Tests:**
  - Document the testing procedures for each module or component of 4D-ESTER. Include information on how to run tests, the expected outcomes, and how to interpret test results.
  - Ensure that test cases themselves are well-documented, including the purpose of each test, the input conditions, and the expected output.
- **Mocking and Stubbing:**
  - Use mocking and stubbing techniques to isolate the code under test. This is particularly important in integration tests, where you may need to simulate the behavior of computationally expensive portions of the code base.
  - In Python, use libraries like `unittest.mock` or `pytest-mock`. In C/C++, use `Google Mock`.
- **Performance and Load Testing:**
  - Regularly run performance tests to ensure that the application meets its performance requirements, especially after significant changes.
  - Build a profiling database which includes detailed information on the CPU and memory use of each commit hash.