How to contribute

!!! note

The NOMAD source code is maintained in two synchronized projects on GitHub (https://github.com/nomad-coe/nomad){:target="_blank"} and a GitLab run by MPCDF (https://gitlab.mpcdf.mpg.de/nomad-lab/nomad-FAIR){:target="_blank"}.

Everyone can contribute on GitHub. The GitLab instance requires an account for active contribution.

This not an ideal situation: there are historic reasons and there is a lot of buy-in into the GitLab CI/CD system. This guide addresses contributions to both projects.

Issues

Issue trackers

Everyone can open a new issue (https://github.com/nomad-coe/nomad/issues/new){:target="_blank"} in our main GitHub project (https://github.com/nomad-coe/nomad){:target="_blank"}.

Use issues to ask questions, report bugs, or suggest features. If in doubt, use the main project to engage with us. If you address a specific plugin (e.g. parser), you can also post into the respective projects. See also the list of parsers (../../reference/parsers.md) and the list of built-in plugins (../../reference/plugins.md).

If you are a member of FAIRmat, the NOMAD CoE, or are a close collaborator, you probably have an MPCDF GitLab account (or should ask us for one). Please use the issue tracker on our main GitLab project (https://gitlab.mpcdf.mpg.de/nomad-lab/nomad-FAIR){:target="_blank"}. This is where most of the implementation work is planned and executed.

Issue content

A few tips that will help us to solve your issues quicker:

- Focus on the issue. You don't need to greet us or say thanks and goodbye. Let's keep it
  technical.
- Use descriptive short issue titles. Use specific words over general words.
- Describe the observed problem and not the assumed causes. Clearly separate speculation
  from the problem description.
- **Bugs**: Think how we could reproduce the problem:
    - What NOMAD URL are you using (UI), which package version (Python)?
    - Is there an upload or entry id that we can look at?
    - Example files or code snippets?
    - Don't screenshot code, copy and paste instead. Use code blocks (https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/creating-and-highlighting-code-blocks#syntax-highlighting){:target="_blank"}.
- **Features**: Augment your feature descriptions with a use case that helps us understand
  the feature and its scope.

Issues labels (GitLab)

On the main GitLab project, there are three main categories for labels. Ideally, each issue gets one of each category:

- *State* label (grey): These are used to manage when and how the issue is addressed. This
should be given by the NOMAD team member who is currently responsible to moderate

the

development. If it's a simple fix and you want to do it yourself, assign yourself and use "bugfixes".

- *Component* label (purple): These denote the part of the NOMAD software that is most

likely effected. Multiple purple labels are possible.

- *Kind* label (red): Whether this is a bug, feature, refactoring, or documentation issue.

Unlabeled issues will get labeled by the NOMAD team as soon as possible. You can provide

labels yourself.

Documentation

The documentation is part of NOMAD's main source code project. You can

raise issues (#issues) about the documentation as usual. To make changes, create a merge (#merge-requests-mr-gitlab) or pull (#pull-requests-pr-github) request.

See also the documentation part (./code.md#documentation) in our code navigation guide.

Plugins

Also read the guide on how to develop, publish, and distribute plugins (../plugins/plugins.md).

Built-in plugins (and submodules)

Most plugins that are maintained by the NOMAD team are built-in plugins (e.g. all the parsers). These plugins are also available on the public NOMAD service.

These plugins are tied to the main project's source code via submodules. They are included

in the build and therefore automatically distributed as part of the NOMAD docker images

and Python package.

To contribute to these plugins, use the respective GitHub projects. See also the list of parsers (../../reference/parsers.md) and the list of built-in plugins (../../reference/plugins.md). The same rules apply there. A merge request to the main project will also be required to update the submodule.

All these submodules are placed in the `dependencies` directory. After merging or checking out, you have to make sure that the modules are updated to not accidentally commit old submodule commits again. Usually you do the following to check if you really have a clean working directory:

```shell
git checkout something-with-changes
git submodule update --init --recursive
git status
```

3rd-party plugins

Please open an issue, if you want to announce a plugin or contribute a plugin as a potential built-in plugin (i.e. as part of the public NOMAD service).

Branches and Tags

On the main GitLab project (https://gitlab.mpcdf.mpg.de/nomad-lab/nomad-FAIR){:target="_blank"} we use *protected* and *feature* branches. You must not commit to protected branches directly (even if you have the rights).

- `develop`: a *protected* branch and the *default* branch. It contains the latest,

potentially unreleased, features and fixes. This is the main working branch.
- `master`: a *protected* branch. Represents the latest stable release (usually what the current official NOMAD runs on).
- *feature branches*: this is where you work. Typically they are automatically (use the "Create merge request" button) named after issues: `-`.
- `vX.X.X` or `vX.X.XrcX`: *tags* for (pre-)releases.

The `develop` branch and release tags are automatically synchronized to the GitHub project (https://github.com/nomad-coe/nomad){:target="_blank"}. Otherwise, this project is mostly the target for pull requests (#pull-requests-pr-github) and does not contain other relevant branches.

Merge requests (MR, GitLab)

Create the MR

Ideally, have an issue first and create the merge request (and branch) in the GitLab UI. There is a "Create merge request" button on each issue. When done manually, branches should be based on the `develop` branch and merge request should target `develop` as well.

Commit

Make sure you follow our code guidelines (../../reference/code_guidelines.md) and set up your IDE (./setup.md#set-up-your-ide) to enforce style checks, linting, and static analysis. You can also run tests locally (./setup.md#running-tests). Try to keep a clean commit history and follow our Git tips (#tips-for-a-clean-git-history).

Usually only one person is working on a feature branch. Rebasing, amendments, and force

pushes are allowed.

Changelog

We have an automatically generated changelog in the repository file `CHANGELOG.md`.

This changelog is produced from commit messages and to maintain this file, you need to write commit messages accordingly.

To trigger a changelog entry, your commit needs to end with a so-called *Git trailer* named `Changelog`. A typical commit message for a changelog entry should look like this:

```text
A brief one-line title of the change.

A longer *Markdown*-formatted description of the change. Keep in mind that GitLab will automatically link the changelog entry with this commit and a respective merge requests.

You do not need to manually link to any GitLab resources.

This could span multiple paragraphs. However, keep it short. Documentation should go into
the actual documentation, but you should mention breaks in backward compatibility, deprecation of features, etc.

Changelog: Fixed
```

The trailer value (`Fixed` in the example) has to be one of the following values:

- `Fixed` for bugfixes.
- `Added` for new features.
- `Changed` for general improvements, e.g. updated documentation, refactoring, improving performance, etc.

These categories are consistent with keepachangelog.com (https://keepachangelog.com/){:target="_blank"}.

For more information about the changelog generation read the GitLab documentation (https://docs.gitlab.com/ee/api/repositories.html#add-changelog-data-to-a-changelog-file){:target="_blank"}.

CI/CD pipeline and review

If you push to your merge requests, GitLab will run an extensive CI/CD pipeline.

You need to pay attention to failures and resolve them before review.

To review GUI changes, you can deploy your branch to the dev cluster via CI/CD actions.

Find someone on the NOMAD developer team to review your merge request and request a review

through GitLab. The review should be performed shortly and should not stall the merge request longer than two full work days.

The reviewer needs to be able to learn about the merge request and what it tries to achieve. This information can come from:

- the linked issue
- the merge request description (may contain your commit message) and your comments
- threads that were opened by the author to attach comments to specific places in the code

The reviewer will open *threads* that need to be solved by the merge request author. If all threads are resolved, you can request another review.

For complex merge requests, you can also comment your code and create *threads* for the reviewer to resolve. This will allow you to explain your changes.

Merge

The branch should be recently rebased with `develop` to allow a smooth merge:

```shell
git fetch
git rebase origin/develop
git push origin  -f
```

The merge is usually performed by the merge request author after a positive review.

If you could not keep a clean Git history with your commits, squash the merge request.

Make sure to edit the commit message when squashing to have a changelog (#changelog)

entry.

Make sure to delete the branch on merge. The respective issue usually closes itself, due to the references put in by GitLab.

Pull requests (PR, GitHub)

You can fork the main NOMAD project (https://github.com/nomad-coe/nomad){:target="_blank"} and create pull

requests following the usual GitHub flow. Make sure to target the `develop` branch. A team

member will pick up your pull request and automatically copy it to GitLab to run the pipeline and potentially perform the merge. This process is made transparent in the pull request discussion. Your commits and your authorship is maintained in this process.

Similar rules apply to all the parser and plugin projects. Here, pipelines are run

directly on GitHub. A team member will review and potentially merge your pull requests.

Tips for a clean Git history

It is often necessary to consider code history to reason about potential problems in our code. This can only be done if we keep a "clean" history.

- Use descriptive commit messages. Use simple verbs (*added*, *removed*, *refactored*,
  etc.) name features and changed components.
  Include issue numbers (https://docs.gitlab.com/ee/user/project/issues/crosslinking_issues.html){:target="_blank"}
  to create links in GitLab.
- Learn how to *amend* to avoid lists of small related commits.
- Learn how to *rebase*. Only merging feature branches should create merge commits.
- Squash commits when merging.
- Some videos on more advanced Git usage:
  - Tools & Concepts for Matering Version Control with Git (https://youtu.be/Uszj_k0DGsg){:target="_blank"}
  - Interactive Rebase, Cherry-Picking, Reflog, Submodules, and more (https://youtu.be/qsTthZi23VE){:target="_blank"}

Amend

While working on a feature, there are certain practices that will help us to create
a clean history with coherent commits, where each commit stands on its own.

```shell
git commit --amend
```

If you committed something to your own feature branch and then realize by CI that you have
some tiny error in it you need to fix, try to amend this fix to the last commit.

This will avoid unnecessary tiny commits and foster more coherent single commits. With
`--amend` you are adding changes to the last commit, i.e. editing the last commit. When
you push, you need to force it, e.g. `git push origin feature-branch --force-with-lease`.
So be careful, and only use this on your own branches.

Rebase

```shell
git rebase
```

Let's assume you work on a bigger feature that takes more time. You might want to merge
the version branch into your feature branch from time to time to get the recent changes.
In these cases, use `rebase` and not `merge`. Rebasing puts your branch commits in front
of the merged commits instead of creating a new commit with two ancestors. It moves the
point where you initially branched away from the version branch to the current position in
the version branch. This will avoid merges, merge commits, and generally leaves us with a
more consistent history. You can also rebase before creating a merge request, which
allows no-op merges. Ideally, the only real merges that we ever have are between
version branches.

Squash

```shell
git merge --squash
```

When you need multiple branches to implement a feature and merge between them, try to
use `--squash`. Squashing puts all commits of the merged branch into a single commit. It allows you to have many commits and then squash them into one. This is useful if these commits were made just to synchronize between workstations, due to unexpected errors in CI/CD, because you needed a save point, etc. Again the goal is to have coherent commits, where each commit makes sense on its own.
Squashing can also be applied on a selection of commits during an interactive rebase (https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History#_squashing){:target="_blank"}.