Contributing to ns-3

Release ns-3-dev

ns-3 project

Dec 19, 2025

CONTENTS

1 Introduction **3**

2 General **5**
- 2.1 Licensing . . . 5
- 2.2 Copyright . . . 6
- 2.3 Attribution . . . 7
- 2.4 Coding style . . . 8
- 2.5 Creating a patch . . . 8
- 2.6 Maintainers . . . 8

3 Submitting enhancements **11**
- 3.1 GitLab.com trackers . . . 11
- 3.2 Reporting issues . . . 11
- 3.3 Submitting merge requests . . . 12
- 3.4 Feature requests . . . 16

4 Submitting new models **17**
- 4.1 Options for new models . . . 17
- 4.2 Upstreaming new models . . . 18
- 4.3 Code reviews . . . 18
- 4.4 Submission structure . . . 18
- 4.5 Documentation, tests, and examples . . . 20
- 4.6 Module dependencies . . . 23

5 Submitting externally maintained code **25**
- 5.1 Rationale for the app store . . . 25
- 5.2 App types . . . 26
- 5.3 Submitting to the app store . . . 26
- 5.4 Code review for apps . . . 27
- 5.5 Maintaining app store modules . . . 28
- 5.6 Links to related projects . . . 28
- 5.7 Unmaintained, contributed code . . . 28

6 Coding style **29**
- 6.1 Clang-format . . . 29
- 6.2 check-style-clang-format.py . . . 31
- 6.3 Clang-tidy . . . 33
- 6.4 Source code formatting . . . 35
- 6.5 CMake file formatting . . . 58
- 6.6 Python file formatting . . . 59
- 6.7 Markdown Lint . . . 60

7 Best practices **63**
7.1 Development phase 63

This is the *ns-3 Contributing Guide*. Primary documentation for the ns-3 project is organized as follows:

- Several guides that are version controlled for each release (the latest release) and development tree:
  - Tutorial
  - Installation Guide
  - Manual
  - Model Library
  - Contributing Guide *(this document)*
- ns-3 Doxygen: Documentation of the public APIs of the simulator
- ns-3 wiki

This document is written in reStructuredText for Sphinx and is maintained in the `doc/contributing` directory of ns-3's source code. Source file column width is 100 columns.

CHAPTER

ONE

INTRODUCTION

This document is a guide for those who wish to contribute code to *ns-3* or its related projects in some way.

Changes to *ns-3* software are made by or reviewed by *maintainers*. *ns-3* has a small core team of maintainers, and also some specialized maintainers who maintain a portion of the software. The source code branch that is centrally maintained is sometimes called the *mainline*, a term used further herein. End users are encouraged to report issues and bugs, or to propose software or documentation modifications, to be reviewed by and handled by maintainers. End users can also help by reviewing code proposals made by others. Some end users who contribute high quality patches or code reviews over time may ask or be invited to become a maintainer of software within their areas of expertise. Finally, some end users wish to disseminate their *ns-3* work to the community, through the addition of new features or modules to *ns-3*.

A question often asked by newcomers is "How can I contribute to *ns-3*?" or "How do I get started?". This document summarizes the various ways and processes used to contribute to *ns-3*. Contribution by users is essential for a project maintained largely by volunteers. However, one of the most scarce resources on an open source project is the available time of maintainers, so we ask contributors to please become familiar with conventions and processes used by *ns-3* so as to smooth the contribution process.

The very first step is to become familiar with *ns-3* by reading the tutorial, running some example programs, and then reading some of the code and documentation to get a feel for things. From that point, there are a number of options to contribute:

- Contributing a small documentation correction
- Reporting or discussing a bug in the software
- Fixing an already reported bug by submitting a patch
- Reviewing code contributed by others
- Submitting new code for inclusion in the mainline
- Alerting users to code that is maintained or archived elsewhere
- Submitting a module or fork for publishing in the ns-3 app store
- Becoming a maintainer of part of the simulator

ns-3 is mainly a C++ project, and this contribution guide focuses on how to contribute *ns-3* code specifically, but the overall open source project maintains various related codebases written in several other languages, so if you are interested in contributing outside of *ns-3* C++ code, there are several possibilities:

- *ns-3* provides Python bindings to most of its API, and maintains an automated API scanning process that relies on other tools. We can use maintenance help in the C++/Python bindings support.
- Another Python project is the bake build tool, which has a number of open issues.
- See also our Python-based PyViz visualizer; extensions and documentation would be welcome.
- The NetAnim animator is written in Qt and has lacked a maintainer for several years.

- If you are interested in Linux kernel hacking, or use of applications in *ns-3* such as open source routing daemons, we maintain the Direct Code Execution project.
- If you are familiar with Django, we have work to do on our app store infrastructure.
- Our website is written in Jekyll and is in need of more work.

The remainder of this document is organized as follows.

- Chapter 2 covers general issues surrounding code and documentation contributions, including license and copyright;
- Chapter 3 describes approaches for contributing small enhancements to the *ns-3* mainline, such as a documentation or bug fix;
- Chapter 4 outlines the approach for proposing new models for the mainline;
- Chapter 5 describes how to contribute code that will be stored outside of the *ns-3* mainline, with emphasis on the *ns-3 AppStore*; and
- Chapter 6 provides the coding style guidelines that are mandatory for the *ns-3* mainline and strongly suggested for contributed modules.

CHAPTER

TWO

GENERAL

This section pertains to general topics about licensing, coding style, and working with Git features, including patch submission.

2.1 Licensing

All code submitted must conform to the project licensing framework, which is GNU GPLv2 compatibility. All new source files should contain a license statement. In general, we ask that new source files be provided with a GNU GPLv2 license, but the author may select another GNU GPLv2-compatible license if necessary. GNU GPLv3 is not accepted in the *ns-3* mainline. Note that the Free Software Foundation maintains a list of GPLv2-compatible licenses.

If a contribution is based upon or contains copied code that itself uses GNU GPLv2, then the author should in most cases retain the GPLv2 and optionally extend the copyright and/or the author (or 'Modified by') statements.

If a contribution is borrowed from another project under different licensing, the borrowed code must have a compatible license, and the license must be copied over as well as the code. The author may add the GNU GPLv2 if desired, but in such a case, should clarify which aspects of the code (i.e., the modifications) are covered by the GPLv2 and not found in the original. The Software Freedom Law Center has published guidance on handling this case.

Note that it is incumbent upon the submitter to make sure that the licensing attribution is correct and that the code is suitable for *ns-3* inclusion. Do not include code (even snippets) from sources that have incompatible licenses. Even if the code licenses are compatible, do not copy someone else's code without attribution.

2.1.1 Documentation Licensing

Licensing for documentation or for material found on *ns-3* websites is covered by the Creative Commons CC-BY-SA 4.0 license, and documentation submissions should be submitted under that license. Please ensure that any documentation submitted does not violate the terms of another copyright holder, and has correct attribution. In particular, copying of substantial portions of an academic journal paper, or copying or redrawing of figures from such a paper, likely requires explicit permission from the copyright holder.

2.2 Copyright

Copyright is a statement of ownership of code, while licensing describes the terms by which the owners of the code permit the reuse of the code.

The *ns-3* project does not maintain copyright of contributed code. Copyright remains with the author(s) or their employer. Because multiple people or organizations work on the *ns-3* code over time, one can think of the project and even individual files as a "mixed copyright" work.

Copyright can be stated when originating files in *ns-3* or when making "substantial" changes to such files; the definition of substantial is open to interpretation. Copyright need not be claimed explicitly by adding a copyright statement; according to copyright laws in many countries, copyright rights are automatic upon publishing a work. Copyright should not be explicitly listed for *all* changes to a file; for instance, patches to fix small things or make small adjustments or improvements are not considered to be subjected to copyright protection.

Use of copyright statements in open source projects, as a means of author attribution, can be controversial, because having a long list of copyright statements on every file impairs readability. Also, since copyright is automatic, there is little formal legal requirement to add a copyright statement. The *ns-3* project has decided to follow the guidance provided by the Software Freedom Law Center in this regard: https://softwarefreedom.org/resources/2012/ManagingCopyrightInformation.html

2.2.1 Copyright on new files

When originating a new file, the originating author should place his or her copyright statement at the top in the header, preceding the copy of the license. An important exception to this is if the new file is copied from somewhere else and modified to make the new file; please do not delete the previous copyrights from the copyright file! See below for this case.

An example placement of a copyright statement can be found in the file `src/network/model/packet.h`:

```
/*
 * Copyright (c) 2005,2006 INRIA
 *
 * SPDX-License-Identifier: GPL-2.0-only
```

2.2.2 Copyright on existing files

When providing a substantial feature (maintainers and contributors should mutually agree on this point) as a patch to an existing file, the contributor may add a copyright statement that clarifies the new portion of code that is covered by the new copyright. An example is the program `src/lte/model/lte-ue-phy.h`:

```
/*
 * Copyright (c) 2010 TELEMATICS LAB, DEE - Politecnico di Bari
 * Copyright (c) 2018 Fraunhofer ESK : RLF extensions
 *
 * SPDX-License-Identifier: GPL-2.0-only
```

Here, Fraunhofer ESK added extensions to support radio link failure (RLF), and the copyright statement clarifies the extension (separated from the organization by a colon).

2.2.3 Copyright on external code copied from elsewhere

If a contributor borrows code from somewhere else (such as a snippet of code to implement an algorithm, or whole files altogether), it is important to keep the original copyright (and license statement) in the new file. Contributors who fail to do this may be accused of plagiarism.

Some existing examples of code copied from elsewhere are:

- `src/network/utils/error-model.h`
- `src/core/model/valgrind.h`
- `src/core/model/math.h`

If in doubt about how to handle a specific case, ask a maintainer.

2.3 Attribution

Besides copyright, authors also often seek to list attribution, or even credit for funding, in the headers. We request that contributors refrain from aggressively inserting statements of attribution into the code such as:

```
// New Hello Timer implementation contributed by John Doe, 2014
```

especially for small touches to files, because, over time, it clutters the code. Git logs are used to track who contributed what over time.

Likewise, if someone contributes a minor enhancement or a bug fix to an existing file, this is not typically justification to insert an `Authored by` or `Copyright` statement at the top of the file. If everyone who touched a file did this, we would end up with unwieldy lists of authors on many files. In general, we recommend to follow these guidelines:

- if you are authoring a new file or contributing a substantial portion of code (such as 30% or more new or changed statements), you can list yourself as co-author or add a new copyright to the file header
- if you are modifying less than the above, please refrain from adding copyright or author statements as part of your patch
- do not put your name or your organization's name anywhere in the main body of the code, for attribution purposes

An example of a substantial modification that led to extension of the authors section of the header can be found in `src/lte/model/lte-ue-phy.h`:

```
* Author: Giuseppe Piro  <g.piro@poliba.it>
* Author: Marco Miozzo <mmiozzo@cttc.es>
* Modified by:
*          Vignesh Babu <ns3-dev@esk.fraunhofer.de> (RLF extensions)
*/
```

Here, there were two original authors, and then a third added a substantial new feature (RLF extensions).

Please work with maintainers to balance the competing concerns of obtaining proper attribution and avoiding long headers.

2.4 Coding style

We ask that all contributors make their code conform to the coding standard which is outlined in *Coding style*.

The project maintains a Python program called `check-style-clang-format.py` found in the `utils/` directory. This is a wrapper around the `clang-format` utility and can be used to quickly format new source code files proposed for the mainline. The *ns-3* coding style conventions are defined in the corresponding `.clang-format` file.

In addition to formatting source code files with `clang-format`, the Python program also checks trailing whitespace in text files and converts tabs to spaces, in order to comply with the *ns-3* coding style.

2.5 Creating a patch

Patches are preferably submitted as a GitLab.com Merge Request. Short patches can be attached to an issue report or sent to the mailing-lists, but a Merge Request is the best way to submit.

The UNIX diff tool is the most common way of producing a patch: a patch is a text-based representation of the difference between two text files or two directories with text files in them. If you have two files, `original.cc`, and, `modified.cc`, you can generate a patch with the command `diff -u original.cc modified.cc`. If you wish to produce a patch between two directories, use the command `diff -uprN original modified`.

Make sure you specify to the reviewer where the original files came from and make sure that the resulting patch file does not contain unexpected content by performing a final inspection of the patch file yourself.

Patches such as this are sufficient for simple bug fixes or very simple small features.

Git can be used to create a patch of what parts differ from the last committed code; try:

```
$ git diff
```

The output of *git diff* can be redirected into a patch file:

```
$ git diff > proposal.patch
```

Keep in mind that *git diff* could include unrelated changes made locally to files in the repository (a common example is *.vscode/launch.json*). In order to avoid cluttering, please amend your diff file using a normal text editor before submitting it.

Likewise, if you submit a merge request using GitLab, please add only the changes to the relevant files to the branch you're using for the merge request.

2.6 Maintainers

Maintainers are the set of people who make decisions on code and documentation changes. Maintainers are contributors who have demonstrated, over time, knowledge and good judgment as pertains to contributions to *ns-3*, and who have expressed willingness to maintain some code. *ns-3* is like other open source projects in terms of how people gradually become maintainers as their involvement with the project deepens; maintainers are not newcomers to the project.

The list of maintainers for each module is found here: https://www.nsnam.org/developers/maintainers/

Maintainers review code (bug fixes, new models) within scope of their maintenance responsibility. A maintainer of a module should "sign off" (or approve of) changes to an *ns-3* module before it is committed to the main codebase. Note that we typically do not formalize the signing off using Git's sign off feature, but instead, maintainers will indicate their approval of the merge request using GitLab.com.

Note that some modules do not have active maintainers; these types of modules typically receive less maintenance attention than those with active maintainers (and bugs may linger unattended).

The best way to become a maintainer is to start by submitting patches that fix open bugs or otherwise improve some part of the simulator, and to join in on the technical discussions. People who submit quality patches will catch the attention of the maintainers and may be asked to become one at some future date.

People who ask to upstream a new module or model so that it is part of the main *ns-3* distribution will typically be asked to maintain it going forward (or to find new maintainers for it).

CHAPTER

THREE

SUBMITTING ENHANCEMENTS

This chapter covers how to submit fixes and small patches for the existing mainline code and its documentation.

Enhancements (new models) can be proposed for the mainline, and maintainers will make a decision on whether to include it as mainline or recommend that it be supported in the ns-3 App Store. Documentation on hosting code in the ns-3 App Store is provided in the next chapter (*Submitting externally maintained code*). This chapter provides guidance on submitting code for inclusion in the mainline (and much of it applies also as best practice for app store code).

3.1 GitLab.com trackers

ns-3 uses two trackers to keep track of known issues or submitted code. Maintainers prefer to list everything on the tracker so that issues do not slip through the cracks. Users are encouraged to use these to report or comment on issues or merge requests; this requires users to obtain a GitLab.com account.

If a user wants to report an issue with *ns-3*, please first search the issue tracker for something that may be similar, and if nothing is found, please report the new issue.

If a user wants to submit proposed new code for *ns-3*, please submit on the merge request tracker.

More details for each are provided below. Similarly, users who want to report issues on other related repositories under the *nsnam* project (such as the Bake build system) should follow similar steps there.

3.2 Reporting issues

Issues can be reported against the code or the documentation, if you believe that something is incorrect or could be improved. The key to reporting an issue with the code is to try to provide as much information as possible to allow a maintainer to quickly reproduce the problem. After you've determined which module your bug is related to, if it is inside the official distribution (mainline), then create an issue, label it with the name of the module, and provide as much information as possible.

First, perform a cursory search on the open issue list to see if the problem has already been reported. If it has and the issue is still open, add a comment to the existing issue instead of opening a new one.

If you are reporting an issue against an older version of *ns-3*, please scan the most recent Release Notes to see if it has been fixed since that release.

If you then decide to list an issue, include details of your environment:

1. Which version of ns-3 are you using?
2. What's the name and version of the OS you're using?
3. Which modules do you have installed?

4. Have you modified ns-3-dev in any way?

Here are some additional guidelines:

1. Use a clear and descriptive title for the issue to identify the problem.
2. Describe the exact steps which reproduce the problem in as many details as possible. For example, start by explaining how you coded your script; e.g. which functions were called in what order, or else provide an example program. If your program includes modifications to the ns-3 core or a module, please list them (or even better, provide the diffs).
3. Provide specific examples to demonstrate the steps. Include links to files or projects, or copy/pasteable snippets, which you use in those examples. If you're providing snippets in the issue, use Markdown code block formatting.
4. Describe the behavior you observed after following the steps and point out what exactly is the problem with that behavior. Explain which behavior you expected to see instead and why.
5. If you're reporting that ns-3 crashed, include a crash report with a stack trace from the operating system. On macOS, the crash report will be available in Console.app under *"Diagnostic and usage information"* > *"User diagnostic reports"*. Include the crash report in the issue in a code block, or a file attachment.
6. If the problem is related to performance or memory, include a CPU profile capture with your report.

Some issues have suggested resolutions that are trivial and do not require submitting a merge request. For more complicated resolutions, if you have a patch to propose, either attach it to the issue, or submit a merge request (described next).

3.3 Submitting merge requests

To submit code proposed for *ns-3* as one or more commits, use a merge request. The following steps are recommended to smooth the process.

If you are new to public Git repositories, you may want to read this overview of merge requests. If you are familiar with GitHub pull requests, the GitLab.com merge requests are very similar.

In brief, you will want to fork ns-3-dev into your own namespace (i.e., fork the repository into your personal GitLab.com account, via the user interface on GitLab.com), clone your fork of ns-3-dev to your local machine, create a new feature branch that is based on the current tip of ns-3-dev, push that new feature branch up to your fork, and then from the GitLab.com user interface, generate a Merge Request back to the *ns-3* mainline. You will want to monitor and respond to any comments from reviewers, and try to resolve threads.

3.3.1 Remember the documentation

If you add or change API to the simulator, please include Doxygen changes as appropriate. Please scan the module documentation (written in Restructured Text in the *docs* directory) to check if an update is needed to align with the patch proposal.

3.3.2 Commit message format

Commit messages should be written as follows. For examples, please look at the output of *git log* command.

1. The author string should be formatted such as "John Doe <john.doe@example.com>". It is a good idea to run git config on your machine, or hand-edit the *.gitconfig* file in your home directory, to ensure that your name and email are how you want them to be displayed.

2. The first line of the commit message should be limited to 72 columns if possible. This is not a hard requirement but a preference. If you prefer to add more detail, you can add subsequent message lines below the first one, separated by a blank line. Example:

```
commit e6ca9be6fb5a0592a44967f7885545dce3a6da1a
Author: Rediet <getachew.redieteab@orange.com>
Date:   Wed May 19 16:34:01 2021 +0200

  lte: Assign default values

  Fixes crashing optimized/release builds with 'may be used uninitialized' error
```

3. The first line of the commit message should include the relevant module name or names, separated by a colon. Example:

```
commit 15ab50c03132a5f7686045014c6bedf10ac7d421
Author: Stefano Avallone <stavallo@unina.it>
Date:   Wed Jan 27 14:58:54 2021 +0100

  wifi,wave,mesh: Rescan python bindings
```

4. If the commit fixes an issue in the issue tracker, list it in parentheses after the colon (by saying 'fixes #NNN' where NNN is the issue number). This reference alerts future readers to an issue where more may be discussed about the commit. Example:

```
commit 10ef08140ab2a9f2b550f24d1e881e76ea0873ff
Author: Tom Henderson <tomh@tomh.org>
Date:   Fri May 21 11:11:33 2021 -0700

  network: (fixes #404) Use Queue::Dispose() for SimpleNetDevice::DoDispose()
```

5. If the commit is from a merge request, that may also be added in a similar way the same by saying 'merges !NNN'. The exclamation point differentiates merge requests from issues (which use the number sign '#') on GitLab.com. Example:

```
commit d4258b2b32d6254b878eca9200271fa3f4ee7174
Author: Tom Henderson <tomh@tomh.org>
Date:   Sat Mar 27 09:56:55 2021 -0700

  build: (merges !584) Exit configuration if path whitespace detected
```

Here is an example making use of both:

```
commit a97574779b575af70d975f9e2ca899e2405cf497
Author: Federico Guerra <federico@guerra-tlc.com>
Date:   Tue Jan 14 21:14:37 2020 +0100

  uan: (fixes #129, merges !162) EndTx moved to PhyListener
```

6. Use the present tense ("Add feature", not "Added feature") and the imperative mood ("Move cursor to …", not "Moves cursor to…").

3.3.3 Code formatting

ns-3 uses a utility called `clang-format` to check and fix formatting issues with code. Please see the chapter on coding style to learn more about how to use this tool. When submitting code to the project, it is a good idea to check the formatting on your new files and modifications before submission.

3.3.4 Avoid unrelated changes

Do not make changes to unrelated parts of the code (unrelated to your merge request). If in the course of your work on a given topic, you discover improvements to other things (like documentation improvements), please open a separate merge request for separate topics.

3.3.5 Squashing your history

In the course of developing and responding to review comments, you may add more commits, so what started out as a single commit might grow into several. Please consider to squash any such revisions if they do not need to be preserved as separate commits in the mainline Git history.

If you squash commits, you must force-push your branch back to your fork. Do not worry about this; GitLab.com will update the Merge Request automatically. This tutorial may be helpful to learn about Git rebase, force-push, and merge conflicts.

Note that GitLab can squash the commits while merging. However, it is often preferred to keep multiple commit messages, especially when the merge request contains multiple parts or multiple authors.

It is a good practice to NOT squash commits while the merge request is being reviewed and updated (this helps the reviewers), and perform a selective squash before the merge.

3.3.6 Rebasing on ns-3-dev

It is also helpful to maintainers to keep your feature branch up to date so that the commits are appended to the tip of the mainline code. This is not strictly required; maintainers may do such a rebase upon merging your finalized Merge Request. This may help catch possible merge conflicts before the time to merge arrives.

Note that sometimes it is not possible to rebase a merge request through GitLab's web interface. Hence, it is a good practice to keep your merge request in line with the mainline (i.e., rebase it periodically and push the updated branch).

3.3.7 Resolving discussion threads

Any time someone opens a new comment thread on a Merge Request, a counter of 'Unresolved threads' is incremented (near the top of the Merge Request). If you are able to successfully resolve the comment thread (either by changing your code, or convincing the reviewer that no change is needed), then please mark the thread as resolved. Maintainers will look at the count of unresolved threads and make decisions based on this count as to whether the Merge Request is ready. Maintainers prefer that all threads are resolved successfully before moving forward with a merge.

3.3.8 Adding a label

You can use labels to indicate whether the Merge Request is a bug, pertains to a specific module or modules, is documentation related, etc. This is not required; if you do not add a label, a maintainer probably will.

3.3.9 Other metadata

It is not necessary to set other metadata on the Merge Request such as milestone, reviewers, etc.

3.4 Feature requests

Feature requests are tracked as GitLab.com issues. If you want to suggest an enhancement, create an issue and provide the following information:

1. Use a clear and descriptive title for the issue to identify the suggestion.
2. Provide a step-by-step description of the suggested enhancement in as many details as possible.
3. Provide specific examples to demonstrate the steps. Include copy/pasteable snippets which you use in those examples.
4. Describe the current behavior and explain which behavior you expected to see instead and why.
5. Explain why this enhancement would be useful to most ns-3 users.

The *ns-3* project does not have professional developers available to respond to feature requests, so your best bet is to try to implement it yourself and work with maintainers to improve it, but the project does like to hear back from users about what would be a useful improvement, and you may find like-minded collaborators in the community willing to work on it with you.

Use the *enhancement* Label on your feature request.

CHAPTER

FOUR

SUBMITTING NEW MODELS

We actively encourage submission of new features to *ns-3*. Independent submissions are essential for open source projects, and if accepted into the mainline, you will also be credited as an author of future versions of *ns-3*. However, please keep in mind that there is already a large burden on the *ns-3* maintainers to manage the flow of incoming contributions and maintain new and existing code. The goal of this chapter is to outline the options for new models in *ns-3*, and how you can help to minimize the burden on maintainers and thus minimize the average time-to-merge of your code.

4.1 Options for new models

ns-3 is organized into modules, each of which is compiled and linked into a separate library. Users can selectively enable and disable the inclusion of modules (via the *–enable-modules* argument to ns3 configure, or via the selective inclusion of contributed modules).

When proposing new models to *ns-3*, please keep in mind that not all models will be accepted into the mainline. However, we aim to provide at least one or more options for any code contribution.

Because of the long-term maintenance burden, *ns-3* is no longer accepting all new proposals into the mainline. Features that are of general interest are more likely to be approved for the mainline, but features that are more specialized may be recommended for the ns-3 App Store. Some modules that have been in the mainline for a long time, but fall out of use (or lose their maintainers) may also be moved out of the mainline into the App Store in the future.

The options for publishing new models are:

1. Propose a Merge Request for the *ns-3* mainline and follow the guidelines below.
2. Organize your code as a "contributed module" or modules and maintain them in your own public Git repository. A page on the App Store can be made to advertise this to users, and other tooling can be used to ensure that the module stays compatible with the mainline.
3. Organize your code as a "public fork" that evolves separately from the *ns-3* mainline. This option is sometimes chosen for models that require significant intrusive changes to the *ns-3* mainline to support. Some recent examples include the Public Safety models and the millimeter-wave extensions to *ns-3*. This has the benefit of being self-contained, but the drawback of losing compatibility with the mainline over time. A page on the App Store can be made for these forks as well. For maintenance reasons and improved user experience, we prefer to upstream mainline changes so that public forks can be avoided.
4. Archive your code somewhere, or publish in a Git repository, and link to it from the *ns-3* Contributed Code wiki page. This option requires the least amount of work from the contributor, but visibility of the code to new *ns-3* users will likely be reduced. To follow this route, obtain a wiki account from the webmaster, and make edits as appropriate to link your code.

The remainder of the chapter concerns option 1 (upstreaming to ns-3-dev); the other options are described in the next chapter (*Submitting externally maintained code*).

4.2 Upstreaming new models

The term "upstreaming" refers to the process whereby new code is contributed back to an upstream source (the main open source project) whereby that project takes responsibility for further enhancement and maintenance.

Making sure that each code submission fulfills as many items as possible in the following checklist is the best way to ensure quick merging of your code.

In brief, we can summarize the guidelines as follows:

1. Be licensed appropriately (see *General*)
2. Understand how and why *ns-3* conducts code reviews before merging
3. Follow the ns-3 coding style (*Coding style*) and software engineering and consistency feedback that maintainers may provide
4. Write associated documentation, tests, and example programs

If you do not have the time to follow through the process to include your code in the main tree, please see the next chapter (*Submitting externally maintained code*) about contributing ns-3 code that is not maintained in the main tree.

The process can take a long time when submissions are large or when contributors skip some of these steps. Therefore, best results are found when the following guidelines are followed:

- Ask for review of small chunks of code, rather than large patches. Split large submissions into several more manageable pieces.
- Make sure that the code follows the guidelines (coding style, documentation, tests, examples) or you may be asked to fix these things before maintainers look at it again.

4.3 Code reviews

Code submissions to ns-3-dev are expected to go through a review process where one or more maintainers comment on the code. The basic purpose of the code reviews is to ensure the long-term maintenance and overall system integrity of ns-3. Contributors may be asked to revise their code or rewrite portions during this process. New features are also typically only merged during the early stages of a new release cycle, to avoid destabilizing code before release.

ns-3 code reviews are conducted using GitLab.com merge requests, possibly supported by discussion on the ns-developers mailing list for reviews that involve code that cuts across maintainer boundaries or is otherwise controversial. Many examples of ongoing reviews can be browsed on our GitLab.com site.

4.4 Submission structure

For each code submission, include a description of what your code is doing, and why. Ideally, you should be able to provide a summary description in a short paragraph at the top of the Merge Request. If you want to flag to maintainers that your submission is known to be incomplete (e.g., you are looking for early feedback), preface the title of the Merge Request with `Draft:`.

4.4.1 Coherent and multi-part submissions

Large code submissions should be split into smaller submissions where possible. The likelihood of getting maintainers to review your submission is inversely proportional to its size, because large code reviews require a large block of time for a maintainer. For instance, if you are submitting two routing protocol models and each protocol model can stand on its own, submit two Merge Requests rather than a single one containing both models.

Each submission should be a coherent whole: if you need to edit ten files to get a feature to work, then, the submission should contain all the changes for these ten files. Of course, if you can split the feature in sub-features, then, you should do it to decrease the size of the submission as per the previous paragraph.

For example, if you have made changes to optimize one module, and in the course of doing so, you fixed a bug in another module, make sure you separate these two sets of changes in two separate submissions.

When splitting a large submission into separate submissions, (ideally) these submissions should be prefaced (in the Merge Request description) by a detailed explanation of the overall plan such that code reviewers can review each submission separately but within a larger context. This kind of submission typically is split in multiple dependent steps where each step depends on the previous one. If this is the case, make it very clear in your initial explanation. If you can, minimize dependencies between each step such that reviewers can merge each step separately without having to consider the impact of merging one submission on other submissions.

4.4.2 History rewriting

It is good practice to rebase your feature branch onto the tip of the (evolving) ns-3-dev, so that the commits appear on top of a recent commit of ns-3-dev. For instance, you may have started with an initial Merge Request, and it has been some time to gather feedback from maintainers, and now you want to update your Merge Request with that feedback. A rebase onto the current ns-3-dev `master` branch is good practice for any Merge Request update.

In addition, in the course of your development, and in responding to reviewer comments, the git commit log will accumulate lots of small commits that are not helpful to future maintainers: they make it more painful to use the annotate and bisect commands. For this reason, it is good practice to "squash" and "rebase" commits into coherent groups. For example, assume that the commit history, after you have responded to a reviewer's comment, looks like this (*git log* output):

```
commit 4938c01400fb961c37716c3f415f47ba2b04ab5f (HEAD -> master, origin/master, origin/
↪HEAD)
Author: Some Contributor <some.contributor@example.com>
Date:   Mon Jun 15 11:19:49 2020 -0700

  Fix typo in Doxygen based on code review comment

commit 915cf65cb8f464d9398ddd8bea9af167ced64663
Author: Some Contributor <some.contributor@example.com>
Date:   Thu Jun 11 16:44:46 2020 -0400

  Add documentation, tests, and examples for clever protocol

commit a1d51db126fb3d1c7c76427473f02ee792fdfd53
Author: Some Contributor <some.contributor@example.com>
Date:   Thu Jun 11 16:35:48 2020 -0400

  Add new models for clever protocol
```

In the above case, it will be helpful to squash the top commit about the typo into the first commit where the Doxygen was first written, leaving you again with two commits. Git interactive rebase is a good tool for this; ask a maintainer

for help in doing this if you need some help to learn this process.

4.5 Documentation, tests, and examples

Often we have observed that contributors will provide nice new classes implementing a new model, but will fail to include supporting test code, example scripts, and documentation. Therefore, we ask people submitting the code (who are in the best position to do this type of work) to provide documentation, test code, and example scripts.

Note that you may want to go through multiple phases of code reviews, and all of this supporting material may not be needed at the first stage (e.g. when you want some feedback on public API header declarations only, before starting the implementation). However, when it times come to merge your code, you should be prepared to provide these things, as fits your contribution (maintainers will provide some guidance here).

If you add a new features, or make changes to existing features, you need to update existing or write new documentation and example code. Consider the following checklist:

- Doxygen should be added to header files for all public classes and methods, and should be checked for Doxygen errors
- New features should be described in the `RELEASE_NOTES.md`
- Public API changes (if any) must be documented in `CHANGES.md`
- New API or changes to existing API must update the inline Doxygen documentation in header files
- Consider updating or adding a new section to the manual (`doc/manual`) or model library (`doc/models`) as appropriate
- Update the `AUTHORS` file for any new authors

4.5.1 Guidelines to Write Sphinx Documentation

Here are some general guidelines when creating Sphinx documentation:

- When possible, avoid using more than 2 levels of hierarchy (i.e., create only sections and subsections).
- Avoid creating sections or subsections that only contain small paragraphs. If the content is too short, consider combining the content of 2 sections/subsections into one (exceptions include the fixed sections and subsections).
- Include code snippets and/or images (figures or text-based) to better explain your content. This is particularly recommended for the helpers subsection.
- Use references with numbers, rather than using names. Include the reference link when available (use the number for the link).
- Follow the section hierarchy explained in this guide.
- Verify that there is a line between the header and the body of each section or subsection.

For a concrete example, see `src/lr-wpan/doc/lr-wpan.rst`.

Documentation should be created using the following heading hierarchy:

```
Model Name
==========

Section
-------

```

(continues on next page)

(continued from previous page)

```
Subsection
~~~~~~~~~~

SubsubSection (Use only in extreme cases!)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

In general, the number of sections or subsection is free, but there are a few sections and subsections that must be present. The following is the general outline of a module documentation in ns-3:

```
My fantastic model name here
============================

This first part is for describing what the model is trying to accomplish.
General descriptions , design, overview of the project goes in this part without the␣
↪need of any subsections.
An additional visual summary is also recommended.

Scope and Limitations
---------------------

A list of missing capabilities or things not considered in the model goes here.
This section is fixed and must be the first section in the model documentation.

Section A
---------

Free form sections and subsections goes here.

Section B
---------

Free form sections and subsections goes here.

Usage
-----

A brief description of the model usage goes here. This section must be present.

Helpers
~~~~~~~

A description of the helpers used by the model goes here. Snippets of code are
preferable when describing the helpers usage. This subsection must be present.
If the model CANNOT provide helpers write "Not applicable".

Attributes
~~~~~~~~~~

A list of attributes used by the model, each attribute should include a small
description. This subsection must be present.
If the model CANNOT provide attributes write "Not applicable".
```

(continues on next page)

(continued from previous page)

```
Traces
~~~~~~

A list of the source traces used by the model, each trace should include a small
description. This subsection must be present.
If the model CANNOT provide traces write "Not applicable".

Examples and Tests
------------------

Do not forget to add a brief description of each example and test present in the model.
Example:
``my-example.cc : This example demonstrate the use of x in y situation.``
This section must be present.

Validation
----------

Descriptions of how the model was validated must be here. This section must be present.

References
----------

The reference material used in the construction of the model. This section must be␣
↪present.
```

4.5.2 Examples

For many submissions, it will be important to include at least one example that exercises the new code, so the reviewer understands how it is intended to be used. For final submission, please consider to add as many examples as you can that will help new users of your code. The `samples/` and `examples/` directories are places for these files.

4.5.3 Tests

All new models should include automated tests. These should, where appropriate, be unit tests of model correctness, validation tests of stochastic behavior, and overall system tests if the model's interaction with other components must be tested. The test code should try to cover as much model code as possible, and should be checked as much as possible by regression tests.

The ns-3 manual provides documentation and suggestions for how to write tests.

4.6 Module dependencies

Since *ns-3* is a modular system, each module has (typically) dependencies on other modules. These must be kept to the minimum necessary, and circular dependencies must be avoided (they will generate errors). As an example, the `lr-wpan` module depends on `core`, `network`, `mobility`, `spectrum`, and `propagation`. Tests for your new module must not add any other dependency (that was not needed for the model or helper code itself), but examples can use other modules because their module dependencies are handled independently. As an example, the `lr-wpan` examples also use the `stats` module.

This rule might pose limitations to the authoring of tests. In such a case, ask a maintainer for suggestions, or check existing tests to find solutions. Sometimes tests that involve adding additional dependencies are placed into the `src/test` subdirectory.

CHAPTER

FIVE

SUBMITTING EXTERNALLY MAINTAINED CODE

This chapter mainly pertains to code that will be maintained in external repositories (such as a personal or university research group repository, possibly hosted on GitHub or GitLab.com), but for which the contributor wishes to keep consistent and compatible with the *ns-3* mainline.

If the contributor does not want to maintain the external code but wants to make the community aware that it is available with no ongoing support, links to the code can be posted on the *ns-3* wiki contributed code page. A typical example is the graduating student who wishes to make his or her thesis-related code available but who does not plan to update it further. See *Unmaintained, contributed code* below.

However, much of the emphasis of this chapter is on hosting *ns-3* extensions in the *ns-3* App Store.

5.1 Rationale for the app store

Historically, *ns-3* tried to merge all code contributions (that met the contribution guidelines) to the mainline. However, this approach has reached its limits in terms of scalability:

1. As *ns-3* includes more lines of code, it takes longer to compile, while users typically only need a subset of the available *ns-3* model code.
2. Merging to the mainline requires maintainer participation for code reviews (which can be time consuming for maintainers), resulting in some contributions not being merged due to lack of code reviews.

The app store federates the maintenance of *ns-3* modules, and puts more control in the hands of the contributor to maintain their code. The idea is to provide a central place where users can discover *ns-3* extensions and add them to their *ns-3* installation. One of the drawbacks of this approach, however, is the potential for a lot of compatibility headaches, such as a user trying to use two separately developed modules, but the modules are compatible with different *ns-3* mainline versions. Therefore, the app store is designed to assist in testing and clarifying which version of the external module is compatible with which mainline version of *ns-3*.

In brief, what it means for *ns-3* users is the following. There is an empty directory in the *ns-3* source code tree called *contrib*. This directory behaves the same way (with respect to the build system) as the *src* directory, but is the place in which externally developed models can be downloaded and added to *ns-3*.

For contributors, the app store provides a means for module developers to make releases on their own schedule and not be tied to the release schedule of the mainline, nor to be blocked by mainline code reviews or style requirements. It is still possible (and recommended) to obtain code reviews and follow style guidelines for contributed modules, as described below.

5.1.1 Example app

For example, consider a user who wants to use a model of the LoRaWAN sensor networking protocol in *ns-3*. This model is not available in the mainline *ns-3* releases, but the user checks in the App Store. Sensor networking protocols can be found by browsing the category *Sensor Networks* in the left-hand menu, upon which the LoRaWAN module is displayed. This module is maintained by an external research group. Clicking on the app icon leads to a page in which more details about this module are displayed, including *Release History*, *Installation*, and *Maintenance*.

The installation tab suggests two primary ways to add a *lorawan* module to an *ns-3* installation:

1. Download a source archive from the link provided on the app store and unpack it in the *contrib/* directory.
2. Perform a *git clone* operation from within the *ns-3 contrib/* directory.

There is a third way in general, which is to use the Bake build system to add or remove modules. Most modules have a *bakeconf.xml* file associated with them, which can be added to the *contrib/* folder of Bake. This permits bake configuration operations such as:

```
$ ./bake.py configure -e ns-3.34 -e lorawan-0.3.0
$ ./bake.py download
$ ./bake.py build
```

Users will be concerned about which version of the module applies to the version of *ns-3*. The *Release History* tab describes the releases and the minimum (and, optionally, the maximum) *ns-3* version that the release works with.

Finally, information about how the module will continue to be maintained, and how to submit issue reports or ask questions about it, are typically posted in the optional *Maintenance* or *Development* tabs.

5.2 App types

Two types of "apps" are hosted on the app store: 1) contributed modules, and 2) forks of *ns-3*. It benefits users the most when contributed code can be added as modules to the *contrib* directory, because this allows the modules to be used with different versions of the *ns-3* mainline and to combine multiple such modules. However, some authors have chosen to create a full fork of *ns-3*, and to publish full *ns-3* source trees. One reason for such forks is if the fork requires intrusive changes to the *ns-3* mainline code, in such a manner that the patch will likely not be accepted in the mainline. For an example of a full fork in the app store, see the Public Safety Communications app page; this is a full fork because of the intrusive changes to the mainline LTE module necessary to support the public safety extensions.

Sometimes a contributed module does require a very small patch to the mainline despite being largely implemented in a modular way. In this case, the best strategy is to try to create a minimal patch to upstream to the mainline, and work with maintainers to incorporate it. Even if this is not successful or not attempted, one can maintain a small patch file as part of the module, and bake build instructions for the module can be extended to patch the mainline code as the first step in the build process. Please consult the app store maintainers if guidance on this approach is needed.

5.3 Submitting to the app store

If you are interested in adding an app to the app store, contact `webmaster@nsnam.org` or one of the app store maintainers. Before getting started, browse through the existing apps so that you get a feel for what type of information will be required.

The main requirements are:

1. Define a module name that will be the name of the subdirectory in the *contrib* folder. Use *ns-3* naming conventions for directory names; i.e., all lower case, with separate words separated by dashes.

2. Ensure that your module can be successfully cloned into the contrib folder using the proposed module name, that it builds successfully, and that all examples and tests run.
3. Make at least one release on your GitHub or GitLab.com, associated with the latest *ns-3* release if possible. To do this, you will need to decide on a numbering scheme, such as 'v1.0' or '0.1', etc. Consult GitHub or GitLab.com documentation on how to make a (tagged) release.
4. Come up with a small thumbnail icon for your app. A general guideline is a transparent PNG, size 100x100 pixels.
5. Provide names and email addresses of the person(s) who should have edit privileges (accounts) on the app store.

After providing this information, a skeleton page will be set up on the app store and will be initially invisible to users who visit the front page. Contributors can then log in and work with the app store maintainer to finalize the page, and then make it active (visible) to other users when ready.

5.4 Code review for apps

Even though apps are not added to the *ns-3* mainline, it is still possible to request code reviews for them (in order to improve the code). There is a special repository set up for this purpose: ns-3-contrib-reviews. The purpose of this repository is to provide a fork-able repository against which to generate Merge Requests, but no eventual merge ever takes place.

The steps to request a code review are:

1. Fork *ns-3-contrib-reviews* into your own namespace
2. Clone the fork to your local machine
3. Create a new feature branch on what you just checked out for your new code
4. cd to 'contrib' and clone your extension module there
5. remove the .git directory of that module so that git does not treat it as a submodule; e.g., if your module name is *modulename*:

```
$ rm -rf modulename/.git
```

6. Force an add of the module, overcoming the .gitignore file for contrib modules

```
$ git add -f modulename
```

7. Add your files:

```
$ git commit -m"Add modulename for review" -a
```

8. Push the feature branch to your remote repository

```
$ git push -u origin
```

9. Navigate to your github.com page and generate a Merge Request towards the upstream *ns-3-contrib-reviews*
10. Announce on ns-developers mailing list that you would like a review.

5.5 Maintaining app store modules

A recent Google Summer of Code project worked on allowing integration of the app store with continuous integration (CI) testing. This integration is ongoing, but the eventual goal is that regressions that are caused by upstream changes to *ns-3* will be quickly caught, and the app owner will be asked to fix the build of their app, possibly by issuing another module release as well.

5.6 Links to related projects

Some projects choose to maintain their own version of *ns-3*, or maintain models outside of the main tree of the code. In this case, the way to find out about these is to look at the Related Projects page on the *ns-3* wiki.

If you know of externally maintained code that the project does not know about, please email `webmaster@nsnam.org` to request that it be added to the list of external projects.

Going forward, we will tend to prefer to list related projects on the app store so that there is a single place to discover them.

5.7 Unmaintained, contributed code

Anyone who wants to provide access to code that has been developed to extend *ns-3* but that will not be further maintained may list its availability on our website. Furthermore, we can provide, in some circumstances, storage of a compressed archive on a web server if needed. This type of code contribution will not be listed in the app store, although popular extensions might be adopted by a future contributor.

We ask anyone who wishes to do this to provide at least this information on our wiki:

- Authors,
- Name and description of the extension,
- How it is distributed (as a patch or full tarball),
- Location,
- Status (how it is maintained)

Please also make clear in your code the applicable software license. The contribution will be stored on our wiki. If you need web server space for your archive, please contact `webmaster@nsnam.org`.

CHAPTER

SIX

CODING STYLE

When writing code to be contributed to the *ns-3* open source project, we ask that you follow the coding standards, guidelines, and recommendations found below.

6.1 Clang-format

The *ns-3* project uses clang-format to define and enforce the C++ coding style. Clang-format can be easily integrated with modern IDEs or run manually on the command-line.

Besides clang-format, *ns-3* adopts other coding-style guidelines that are not covered by clang-format, which are explained in this document. Read the `check-style-clang-format.py` section below for information on how to use this Python script to check and fix all formatting guidelines followed by *ns-3*.

6.1.1 Clang-format installation

Clang-format can be installed using one of two methods. Please note that you should install one of the supported versions of clang-format, which are listed in the `RELEASE_NOTES.md` file.

The first method is to install clang-format using the package manager available in the Linux distribution (e.g., Ubuntu's `apt`). For example, in Ubuntu 24.04, clang-format 20 can be installed with the following command:

```
sudo apt install clang-format-20
```

If the package manager does not provide one of the clang-format versions supported by *ns-3*, users can install clang-format using Python's pip tool.

The following command will install the latest version of clang-format:

```
pip3 install clang-format
```

To install a specific version of clang-format, use the following command:

```
pip3 install clang-format==<version_number>
```

where <version_number> is something like `20.1.8` (MAJOR.MINOR.PATCH).

Starting with Python 3.11, pip requires users to either create a virtual environment (venv) or add the `--break-system-packages` flag to the installation commands above.

6.1.2 Supported versions of clang-format

Since each new major version of clang-format can add or modify properties, newer versions of clang-format might produce different outputs compared to previous versions.

The list of clang-format versions that are verified to produce consistent output among themselves are listed in the `RELEASE_NOTES.md` document.

6.1.3 Integration with IDEs

Clang-format can be integrated with modern IDEs (e.g., VS Code) that are able to read the `.clang-format` file and automatically format the code on save or on typing.

Please refer to the documentation of your IDE for more information. Some examples of IDE integration are provided in clang-format documentation.

As an example, VS Code's C/C++ extension contains the latest clang-format binary. VS Code can be configured to automatically format code on save, on paste and on type by enabling the following settings:

```
{
  "editor.formatOnSave": true,
  "editor.formatOnPaste": true,
  "editor.formatOnType": true,
}
```

6.1.4 Clang-format usage in the terminal

In addition to IDE support, clang-format can be manually run on the terminal.

To format a file in-place, run the following command:

```
clang-format -i $FILE
```

To check that a file is well formatted, run the following command on the terminal. If the file is not well formatted, clang-format indicates the lines that need to be formatted.

```
clang-format --dry-run $FILE
```

6.1.5 Clang-format Git integration

Clang-format integrates with Git to format Git commits or changes not yet committed, such as pending merge requests on the GitLab repository. The full documentation is available on clang-format Git integration

To fix the formatting of files with Git, run the following commands in the *ns-3* main directory. These commands do not change past commits. Instead, the reformatted files are left in the workspace. These changes should be squashed to the corresponding commits, in order to fix them.

```
# Fix all commits of the current branch, relative to the master branch
git clang-format master

# Fix all staged changes (i.e., changes that have been `git add`ed):
git clang-format
```

(continues on next page)

(continued from previous page)

```
# Fix all changes staged and unstaged:
git clang-format -f

# Fix specific files:
git clang-format path_to_file

# Check what formatting changes are needed (if no files provided, check all staged␣
↪files):
git clang-format --diff
```

Note that this only fixes formatting issues related to clang-format. For other *ns-3* coding style guidelines, read the `check-style-clang-format.py` section below.

In addition to Git patches, clang-format-diff can also be used to reformat existing patches produced with the `diff` tool.

6.1.6 Disable formatting in specific files or lines

To disable formatting in specific lines, surround them with the following C++ comments:

```
// clang-format off
...
// clang-format on
```

To exclude an entire file from being formatted, surround the whole file with the special comments.

6.2 check-style-clang-format.py

To facilitate checking and fixing source code files according to the *ns-3* coding style, *ns-3* maintains the `check-style-clang-format.py` Python script (located in `utils/`). This script is a wrapper to clang-format and provides useful options to check and fix source code files. Additionally, it performs other manual checks and fixes in text files (described below).

We recommend running this script over your newly introduced C++ files prior to submission as a Merge Request.

The script performs multiple style checks. It returns a zero exit code if all files adhere to these rules. If there are files that do not comply with the rules, the process returns a non-zero exit code and lists the respective files. This mode is useful for developers editing their code and for the GitLab CI/CD pipeline to check if the codebase is well formatted.

The script runs the checks explained in the following table. All checks are enabled by default. Users can disable specific checks using the corresponding flags.

Check	Description	Flag to Disable Check
Formatting	Check code formatting using clang-format. Respects clang-format guards.	`--no-formatting`
#include "ns3/" prefixes	Check if local `#include` headers do not use the "ns3/" prefix. Respects clang-format guards.	`--no-include-prefixes`
#include quotes	Check if ns-3 `#include` headers use quotes ("") instead of angle brackets (<>). Respects clang-format guards.	`--no-include-quotes`
Doxygen tags	Check if Doxygen tags use @ rather than \\. Respects clang-format guards.	`--no-doxygen-tags`
SPDX Licenses	Check if source code use SPDX licenses rather than GPL license text. Respects clang-format guards.	`--no-licenses`
Emacs comments	Check if source code does not have emacs file style comments. Respects clang-format guards.	`--no-emacs`
Trailing whitespace	Check if there are no trailing whitespace. Always checked.	`--no-whitespace`
Tabs	Check if there are no tabs. Respects clang-format guards.	`--no-tabs`
File encoding	Check if files have the correct encoding (UTF-8). Always checked.	`--no-encoding`

Additional information about the formatting issues detected by the script can be enabled by adding the `-v, --verbose` flag.

In addition to checking the files, the script can automatically fix detected issues in-place. This mode is enabled by adding the `--fix` flag.

The formatting and tabs checks respect clang-format guards, which mark code blocks that should not be checked. Trailing whitespace is always checked regardless of clang-format guards.

The complete API of the `check-style-clang-format.py` script can be obtained with the following command:

```
./utils/check-style-clang-format.py --help
```

For quick-reference, the most used commands are listed below:

```
# Entire codebase (using paths relative to the ns-3 main directory)
./utils/check-style-clang-format.py --fix .

# Entire codebase (using absolute paths)
/path/to/utils/check-style-clang-format.py --fix /path/to/ns3

# Specific directory or file
/path/to/utils/check-style-clang-format.py --fix absolute_or_relative/path/to/directory_
↪or_file

# Files modified by the current branch, relative to the master branch
git diff --name-only master | xargs ./utils/check-style-clang-format.py --fix
```

6.3 Clang-tidy

The *ns-3* project uses clang-tidy to statically analyze (lint) C++ code and help developers write better code. Clang-tidy can be easily integrated with modern IDEs or run manually on the command-line.

The list of clang-tidy checks currently enabled in the *ns-3* project is saved on the `.clang-tidy` file. The full list of clang-tidy checks and their detailed explanations can be seen in Clang-tidy checks.

6.3.1 Clang-tidy installation

Clang-format can be installed using your OS's package manager. Please note that you should install one of the supported versions of clang-format, which are listed in the following section.

6.3.2 Supported versions of clang-tidy

Since clang-tidy is a linter that analyzes code and outputs errors found during the analysis, developers can use different versions of clang-tidy on the workflow. Newer versions of clang-tidy might produce better results than older versions. Therefore, it is recommended to use the latest version available.

To ensure consistency among developers, *ns-3* defines a minimum version of clang-tidy, whose warnings must not be ignored. Therefore, developers should, at least, scan their code with the minimum version of clang-tidy. However, more recent versions can be used, which will produce better warnings.

The supported versions of clang-tidy are listed in the `RELEASE_NOTES.md` document.

6.3.3 Integration with IDEs

Clang-tidy automatically integrates with modern IDEs (e.g., VS Code) that read the `.clang-tidy` file and automatically checks the code of the currently open file.

Please refer to the documentation of your IDE for more information. Some examples of IDE integration are provided in clang-tidy documentation

6.3.4 Clang-tidy usage

In order to use clang-tidy, *ns-3* must first be configured with the flag `--enable-clang-tidy`. To configure *ns-3* with tests, examples and clang-tidy enabled, run the following command:

```
./ns3 configure --enable-examples --enable-tests --enable-clang-tidy
```

Due to the integration of clang-tidy with CMake, clang-tidy can be run while building *ns-3*. In this way, clang-tidy errors will be shown alongside build errors on the terminal. To build *ns-3* and run clang-tidy, run the the following command:

```
./ns3 build
```

To run clang-tidy without building *ns-3*, use the following commands on the ns-3 main directory:

```
# Analyze (and fix) a single file with clang-tidy
clang-tidy -p cmake-cache/ [--fix] [--format-style=file] [--quiet] $FILE

# Analyze (and fix) multiple files in parallel
```

(continues on next page)

(continued from previous page)

```
run-clang-tidy -p cmake-cache/ [-fix] [-format] [-quiet] $FILE1 $FILE2 ...

# Analyze (and fix) the entire ns-3 codebase in parallel
run-clang-tidy -p cmake-cache/ [-fix] [-format] [-quiet]
```

When running clang-tidy, please note that:

- Clang-tidy only analyzes implementation files (i.e., `*.cc` files). Header files are analyzed when they are included by implementation files, with the `#include "..."` directive.
- Not all clang-tidy checks provide automatic fixes. For those cases, a manual fix must be made by the developer.
- Enabling clang-tidy will add time to the build process (in the order of minutes).

6.3.5 Disable clang-tidy analysis in specific lines

To disable clang-tidy analysis of a particular rule in a specific function, specific clang-tidy comments have to be added to the corresponding function. Please refer to the official clang-tidy documentation for more information.

To disable `modernize-use-override` checking on `func()` only, use one of the following two "special comment" syntaxes:

```
//
// Syntax 1: Comment above the function
//

// NOLINTNEXTLINE(modernize-use-override)
void func();

//
// Syntax 2: Trailing comment
//

void func(); // NOLINT(modernize-use-override)
```

To disable a specific clang-tidy check on a block of code, for instance the `modernize-use-override` check, surround the code block with the following "special comments":

```
// NOLINTBEGIN(modernize-use-override)
void func1();
void func2();
// NOLINTEND(modernize-use-override)
```

To disable all clang-tidy checks on a block of code, surround it with the following "special comments":

```
// NOLINTBEGIN
void func1();
void func2();
// NOLINTEND
```

To exclude an entire file from being checked, surround the whole file with the "special comments".

6.4 Source code formatting

The *ns-3* coding style was changed between the ns-3.36 and ns-3.37 release. Prior to ns-3.37, *ns-3* used a base GNU coding style. Since ns-3.37, *ns-3* changed the base coding style to what is known in the industry as Allman-style braces, with four-space indentation. In clang-format, this is configured by selecting the `Microsoft` base style. The following examples illustrate the style.

6.4.1 Indentation

Indent code with 4 spaces. When breaking statements into multiple lines, indent the following lines with 4 spaces.

```
void
Func()
{
    int x = 1;
}
```

Indent constructor's initialization list with 4 spaces.

```
MyClass::MyClass(int x, int y)
    : m_x(x),
      m_y(y)
{
}
```

Do not use tabs in source code. Always use spaces for indentation and alignment.

6.4.2 Line endings

Files use LF (\n) line endings.

6.4.3 Column limit

Code lines should not extend past 100 characters. This allows reading code in wide-screen monitors without having to scroll horizontally, while also allowing editing two files side-by-side.

6.4.4 Braces style

Braces should be formatted according to the Allman style. Braces are always on a new line and aligned with the start of the corresponding block. The main body is indented with 4 spaces.

Always surround conditional or loop blocks (e.g., `if`, `for`, `while`) with braces, and always add a space before the condition's opening parentheses.

```
void Foo()
{
    if (condition)
    {
        // do stuff here
    }
```

(continues on next page)

(continued from previous page)

```
    else
    {
        // do other stuff here
    }

    for (int i = 0; i < 100; i++)
    {
        // do loop
    }

    while (condition)
    {
        // do while
    }

    do
    {
        // do stuff
    } while (condition);
}
```

6.4.5 Spacing

To increase readability, functions, classes and namespaces are separated by one new line. This spacing is optional when declaring variables or functions. Declare one variable per line. Do not mix multiple statements on the same line.

Do not add a space between the function name and the opening parentheses. This rule applies to both function (and method) declarations and invocations.

```
void Func(const T&);  // OK
void Func (const T&); // Not OK
```

6.4.6 Trailing whitespace

Source code and text files must not have trailing whitespace.

6.4.7 Code alignment

To improve code readability, trailing comments should be aligned.

```
int varOne;    // Variable one
double varTwo; // Variable two
```

The trailing \ character of macros should be aligned to the far right (equal to the column limit). This increases the readability of the macro's body, without forcing unnecessary whitespace diffs on surrounding lines when only one line is changed.

```
#define MY_MACRO(msg)                       \
    do                                      \
```

(continues on next page)

(continued from previous page)

```
    {                                           \
        std::cout << msg << std::endl;          \
    } while (false);
```

6.4.8 Class members

Definition blocks within a class should be organized in descending order of public exposure, that is: `static` > `public` > `protected` > `private`. Separate each block with a new line.

```
class MyClass
{
public:
    static int m_counter = 0;

    MyClass(int x, int y);

private:
    int x;
    int y;
};
```

6.4.9 Function arguments bin packing

Function arguments should be declared in the same line as the function declaration. If the arguments list does not fit the maximum column width, declare each one on a separate line and align them vertically.

```
void ShortFunction(int x, int y);

void VeryLongFunctionWithLongArgumentList(int x,
                                          int y,
                                          int z);
```

The constructor initializers are always declared one per line, with a trailing comma:

```
void
MyClass::MyClass(int x, int y)
    : m_x(x),
      m_y(y)
{
}
```

6.4.10 Function return types

In function declarations, return types are declared on the same line. In function implementations, return types are declared on a separate line.

```
// Function declaration
void Func(int x, int y);

// Function implementation
void
Func(int x, int y)
{
    // (...)
}
```

6.4.11 Templates

Template definitions are always declared in a separate line from the main function declaration:

```
template <class T>
void Func(T t);
```

6.4.12 Naming

Name encoding

Function, method, and type names should follow the CamelCase convention: words are joined without spaces and are capitalized. For example, "my computer" is transformed into `MyComputer`. Do not use all capital letters such as `MAC` or `PHY`, but choose instead `Mac` or `Phy`. Do not use all capital letters, even for acronyms such as `EDCA`; use `Edca` instead. This applies also to two-letter acronyms, such as `IP` (which becomes `Ip`). The goal of the CamelCase convention is to ensure that the words which make up a name can be separated by the eye: the initial Caps fills that role. Use PascalCasing (CamelCase with first letter capitalized) for function, property, event, and class names.

Variable names should follow a slight variation on the base CamelCase convention: camelBack. For example, the variable `user name` would be named `userName`. This variation on the basic naming pattern is used to allow a reader to distinguish a variable name from its type. For example, `UserName userName` would be used to declare a variable named `userName` of type `UserName`.

Global variables should be prefixed with a `g_` and member variables (including static member variables) should be prefixed with a `m_`. The goal of that prefix is to give a reader a sense of where a variable of a given name is declared to allow the reader to locate the variable declaration and infer the variable type from that declaration. Defined types will start with an upper case letter, consist of upper and lower case letters, and may optionally end with a `_t`. Defined constants (such as static const class members, or enum constants) will be all uppercase letters or numeric digits, with an underscore character separating words. Otherwise, the underscore character should not be used in a variable name. For example, you could declare in your class header `my-class.h`:

```
typedef int NewTypeOfInt_t;
constexpr uint8_t PORT_NUMBER = 17;

class MyClass
{
    void MyMethod(int aVar);
```

(continues on next page)

(continued from previous page)

```
    int m_aVar;
    static int m_anotherVar;
};
```

and implement in your class file `my-class.cc`:

```
int MyClass::m_anotherVar = 10;
static int g_aStaticVar = 100;
int g_aGlobalVar = 1000;

void
MyClass::MyMethod(int aVar)
{
    m_aVar = aVar;
}
```

As an exception to the above, the members of structures do not need to be prefixed with an `m_`.

Finally, do not use Hungarian notation, and do not prefix enums, classes, or delegates with any letter.

Choosing names

Variable, function, method, and type names should be based on the English language, American spelling. Furthermore, always try to choose descriptive names for them. Types are often english names such as: Packet, Buffer, Mac, or Phy. Functions and methods are often named based on verbs and adjectives: `GetX`, `DoDispose`, `ClearArray`, etc.

A long descriptive name which requires a lot of typing is always better than a short name which is hard to decipher. Do not use abbreviations in names unless the abbreviation is really unambiguous and obvious to everyone (e.g., use `size` over `sz`). Do not use short inappropriate names such as foo, bar, or baz. The name of an item should always match its purpose. As such, names such as `tmp` to identify a temporary variable, or such as `i` to identify a loop index are OK.

If you use predicates (that is, functions, variables or methods which return a single boolean value), prefix the name with "is" or "has".

6.4.13 File layout and code organization

A class named `MyClass` should be declared in a header named `my-class.h` and implemented in a source file named `my-class.cc`. The goal of this naming pattern is to allow a reader to quickly navigate through the *ns-3* codebase to locate the source file relevant to a specific type.

Each `my-class.h` header should start with the following comment to ensure that your code is licensed under the GPL, that the copyright holders are properly identified (typically, you or your employer), and that the actual author of the code is identified. The latter is purely informational and we use it to try to track the most appropriate person to review a patch or fix a bug. Please do not add the "All Rights Reserved" phrase after the copyright statement.

```
/*
 * Copyright (c) YEAR COPYRIGHTHOLDER
 *
 * SPDX-License-Identifier: GPL-2.0-only
 *
 * Author: MyName <myemail@example.com>
 */
```

Below these C-style comments, always include the following which defines a set of header guards (`MY_CLASS_H`) used to avoid multiple header includes, which ensures that your code is included in the *ns-3* namespace and which provides a set of Doxygen comments for the public part of your class API. Detailed information on the set of tags available for doxygen documentation is described in the Doxygen website.

```
#ifndef MY_CLASS_H
#define MY_CLASS_H

namespace ns3
{

/**
 * @brief short one-line description of the purpose of your class
 *
 * A longer description of the purpose of your class after a blank
 * empty line.
 */
class MyClass
{
public:
    MyClass();

    /**
     * A detailed description of the purpose of the method.
     *
     * @param firstParam a short description of the purpose of this parameter
     * @return a short description of what is returned from this function.
     */
    int DoSomething(int firstParam);

private:
    /**
     * Private method doxygen is also recommended
     */
    void MyPrivateMethod();

    int m_myPrivateMemberVariable; ///< Brief description of member variable
};

} // namespace ns3

#endif // MY_CLASS_H
```

The `my-class.cc` file is structured similarly:

```
/*
 * Copyright (c) YEAR COPYRIGHTHOLDER
 *
 * SPDX-License-Identifier: GPL-2.0-only
 *
 * Author: MyName <myemail@foo.com>
 */

#include "my-class.h"
```

(continues on next page)

(continued from previous page)

```
namespace ns3
{

MyClass::MyClass()
{
}

...

} // namespace ns3
```

6.4.14 Header file includes

Included header files should be organized by source location. The sorting order is as follows:

```
// Header class (applicable for *.cc files)
#include "my-class.h"

// Includes from the same module
#include "header-from-same-module.h"

// Includes from other modules
#include "ns3/header-from-different-module.h"

// External headers (e.g., STL libraries)
#include <iostream>
```

Groups should be separated by a new line. Within each group, headers should be sorted alphabetically.

For standard headers, use the C++ style of inclusion:

```
#include <cstring>  // OK
#include <string.h> // Avoid
```

- inside .h files, always use

  ```
  #include "ns3/header.h"
  ```

- inside .cc files, use

  ```
  #include "header.h"
  ```

  if file is in same directory, otherwise use

  ```
  #include "ns3/header.h"
  ```

6.4.15 Variables and constants

Each variable declaration is on a separate line. Variables should be declared at the point in the code where they are needed, and should be assigned an initial value at the time of declaration.

```
// Do not declare multiple variables per line
int x, y;

// Declare one variable per line and assign an initial value
int x = 0;
int y = 0;
```

Named constants defined in classes should be declared as `static constexpr` instead of macros, const, or enums. Use of `static constexpr` allows a single instance to be evaluated at compile-time. Declaring the constant in the class enables it to share the scope of the class.

If the constant is only used in one file, consider declaring the constant in the implementation file (`*.cc`).

```
// Avoid declaring constants as enum
class LteRlcAmHeader : public Header
{
    enum ControlPduType_t
    {
        STATUS_PDU = 000,
    };
};

// Prefer to declare them as static constexpr (in class)
class LteRlcAmHeader : public Header
{
    static constexpr uint8_t STATUS_PDU{0};
};

// Or as constexpr (in implementation files)
constexpr uint8_t STATUS_PDU{0};
```

When declaring variables that are easily deducible from context, prefer to declare them with `auto` instead of repeating the type name. Not only does this improve code readability, by making lines shorter, but it also facilitates future code refactoring.

```
// Avoid repeating the type name when declaring iterators or pointers, and casting␣
↪variables
std::map<uint32_t, std::string>::const_iterator it = myMap.find(key);
int* ptr = new int[10];
uint8_t m = static_cast<uint8_t>(97 + (i % 26));

// Prefer to declare them with auto
auto it = myMap.find(key);
auto* ptr = new int[10];
auto m = static_cast<uint8_t>(97 + (i % 26));
```

6.4.16 Initialization

When declaring variables, prefer to use direct-initialization, to avoid repeating the type name.

```
// Avoid splitting the declaration and initialization of variables
Ipv4Address ipv4Address = Ipv4Address("192.168.0.1")

// Prefer to use direct-initialization
Ipv4Address ipv4Address("192.168.0.1")
```

Variables with no default constructor or of primitive types should be initialized when declared.

Variables with default constructors do not need to be explicitly initialized, since the compiler already does that. An example of this is the `ns3::Time` class, which will initialize to zero.

Member variables of structs and classes should be initialized unless the member has a default constructor that guarantees initialization. Preferably, variables should be initialized together with the declaration (in the header file). Alternatively, they can be initialized in the default constructor (in the implementation file), and you may see instances of this in the codebase, but direct initialization upon declaration is preferred going forward.

If all member variables of a class / struct are directly initialized (see above), they do not require explicit default initialization. But if not all variables are initialized, those non-initialized variables will contain garbage. Therefore, initializing the class object with `{}` allows all member variables to always be initialized – either with the provided default initialization or with the primitive type's default value (typically 0).

C++ supports two syntax choices for direct initialization, either `()` or `{}`. There are various tradeoffs in the choices for more complicated types (consult the C++ literature on brace vs. parentheses initialization), but for the fundamental types like `double`, either is acceptable (please use consistently within files).

Regarding `ns3::Time`, do not initialize to non-zero integer values as follows, assuming that it will be converted to nanoseconds:

```
Time t{1000000};  // This is disallowed
```

The value will be interpreted according to the current resolution, which is ambiguous. A user's program may have already changed the resolution from the default of nanoseconds to something else by the time of this initialization, and it will be instead interpreted according to 10^6 * the new resolution unit.

Time initialization to raw floating-point values is additionally fraught, because of rounding. Doing so with small values has led to bugs in practice such as timer timeout values of zero time.

When declaring or manipulating `Time` objects with known values, prefer to use integer-based representations and arguments over floating-point fractions, where possible, because integer-based is faster. This means preferring the use of `NanoSeconds`, `MicroSeconds`, and `MilliSeconds` over `Seconds`. For example, to represent a tenth of a second, prefer `MilliSeconds(100)` to `Seconds(0.1)`.

To summarize Time declaration and initialization, consider the following examples and comments:

```
Time t;  // OK, will be value-initialized to integer zero
Time t{MilliSeconds(100)};  // OK, fastest, no floating point involved
Time t{"100ms"}; // OK, will perform a string conversion; integer would be faster
Time t{Seconds(0.1)};  // OK, will invoke Seconds(double); integer would be faster
Time t{100000000}; // NOT OK, is interpreted differently when ``Time::SetResolution()``␣
↪called
Time t{0.1}; // NOT OK, will round to zero; see above and also merge request !2007
```

6.4.17 Comments

The project uses Doxygen to document the interfaces, and uses comments for improving the clarity of the code internally. All classes, methods, and members should have Doxygen comments. Doxygen comments should use the C-style comment (also known as Javadoc) style. For comments that are intended to not be exposed publicly in the Doxygen output, use the `@internal` and `@endinternal` tags. Please use the `@see` tag for cross-referencing. All parameters and return values should be documented. The *ns-3* codebase prefers the `@` character for tag identification. This character is recognized by clang-format as the start of Doxygen tags, which enables it to keep tags properly formatted; therefore please don't use \ as the delimiter.

```
/**
 * MyClass description.
 */
class MyClass
{
    /**
     * Constructor.
     *
     * @param n Number of elements.
     */
    MyClass(int n);
};
```

All the functions and variables must be documented, with the exception of member functions inherited from parent classes (the documentation is copied automatically from the parent class), and default constructor/destructor.

It is strongly suggested to use grouping to bind together logically related classes (e.g., all the classes in a module). E.g.;

```
/**
 * @defgroup mynewmodule This is a new module
 */

/**
 * @ingroup mynewmodule
 *
 * MyClassOne description.
 */
class MyClassOne
{
};

/**
 * @ingroup mynewmodule
 *
 * MyClassTwo description.
 */
class MyClassTwo
{
};
```

In the tests for the module, it is suggested to add an ancillary group:

```
/**
 * @defgroup mynewmodule-test Tests for new module
```

(continues on next page)

(continued from previous page)

```
 * @ingroup mynewmodule
 * @ingroup tests
 */

/**
 * @ingroup mynewmodule-tests
 * @brief MyNewModule Test
 */
class MyNewModuleTest : public TestCase
{
};

/**
 * @ingroup mynewmodule-tests
 * @brief MyNewModule TestSuite
 */
class MyNewModuleTestSuite : public TestSuite
{
  public:
    MyNewModuleTestSuite();
};

/**
 * @ingroup mynewmodule-tests
 * Static variable for test initialization
 */
static MyNewModuleTestSuite g_myNewModuleTestSuite;
```

As for comments within the code, comments should be used to describe intention or algorithmic overview where is it not immediately obvious from reading the code alone. There are no minimum comment requirements and small routines probably need no commenting at all, but it is hoped that many larger routines will have commenting to aid future maintainers. Please write complete English sentences and capitalize the first word unless a lower-case identifier begins the sentence. Two spaces after each sentence helps to make emacs sentence commands work. Sometimes `NS_LOG_DEBUG` statements can be also used in place of comments.

Short one-line comments and long comments can use the C++ comment style; that is, `//`, but longer comments may use C-style comments. Use one space after `//` or `/*`.

```
/*
 * A longer comment,
 * with multiple lines.
 */
```

Variable declaration should have a short, one or two line comment describing the purpose of the variable, unless it is a local variable whose use is obvious from the context. The short comment should be on the same line as the variable declaration, unless it is too long, in which case it should be on the preceding lines.

```
int nNodes = 3; // Number of nodes

/// Node container with the Wi-Fi stations
NodeContainer wifiStations(3);
```

Comments in closing braces are generally discouraged, to allow for consistent style formatting across recent versions of clang-format (see MRs !1899 and !2070). This rule may be overridden in cases where the comment improves the

code's readability. For example, in class declarations in files with multiple classes, classes within parent classes, and inline class functions. To ensure consistent style formatting, prefer placing the comment marking the end of the class in a new line before the brace.

An exception to this rule are the comments in the closing brace of a namespace, which identifies the corresponding namespace.

The following examples illustrate the above guidelines.

```
// File with only one class

namespace ns3
{

int
MyClass::Func(int x)
{
    while (...)
    {
        if (...)
        {
        } // end if // Do not add this comment
    } // end while  // Do not add this comment
} // end Func       // Do not add this comment

} // namespace ns3  // Keep this comment
```

```
// Example of file with multiple classes, and classes within classes

class MyClass
{
    class InlineClass
    {
        ...
        int var; //!< Some variable

        // end of class InlineClass  // This comment is allowed
    };

    ...

    // end of class MyClass  // This comment is allowed
};
```

6.4.18 Casts

Where casts are necessary, use the Google C++ guidance: "Use C++-style casts like `static_cast<float>(double_value)`, or brace initialization for conversion of arithmetic types like `int64 y = int64{1} << 42`." Do not use C-style casts, since they can be unsafe.

Try to avoid (and remove current instances of) casting of `uint8_t` type to larger integers in our logging output by overriding these types within the logging system itself. Also, the unary + operator can be used to print the numeric value of any variable, such as:

```
uint8_t flags = 5;
std::cout << "Flags numeric value: " << +flags << std::endl;
```

Avoid unnecessary casts if minor changes to variable declarations can solve the issue. In the following example, `x` can be declared as `float` instead of `int` to avoid the cast, or write numbers in decimal format:

```
// Do not declare x as int, to avoid casting it to float
int x = 3;
float y = 1 / static_cast<float>(x);

// Prefer to declare x as float
float x = 3.0;
float y = 1 / x;

// Or use 1.0 instead of just 1
int x = 3;
float y = 1.0 / x;
```

6.4.19 Namespaces

Code should always be included in a given namespace, namely `ns3`. In order to avoid exposing internal symbols, consider placing the code in an anonymous namespace, which can only be accessed by functions in the same file.

Code within namespaces should not be indented. To more easily identify the end of a namespace, add a trailing comment to its closing brace.

```
namespace ns3
{

// (...)

} // namespace ns3
```

Namespace names should follow the snake_case convention.

6.4.20 Unused variables

Compilers will typically issue warnings on unused entities (e.g., variables, function parameters). Use the `[[maybe_unused]]` attribute to suppress such warnings when the entity may be unused depending on how the code is compiled (e.g., if the entity is only used in a logging statement or an assert statement).

The general guidelines are as follows:

- If a function's or a method's parameter is definitely unused, prefer to leave it unnamed. In the following example, the second parameter is unnamed.

```
void
UanMacAloha::RxPacketGood(Ptr<Packet> pkt, double, UanTxMode txMode)
{
    UanHeaderCommon header;
    pkt->RemoveHeader(header);
    ...
}
```

  In this case, the parameter is also not referenced by Doxygen; e.g.,:

```
/**
 * Receive packet from lower layer (passed to PHY as callback).
 *
 * @param pkt Packet being received.
 * @param txMode Mode of received packet.
 */
void RxPacketGood(Ptr<Packet> pkt, double, UanTxMode txMode);
```

  The omission is preferred to commenting out unused parameters, such as:

```
void
UanMacAloha::RxPacketGood(Ptr<Packet> pkt, double /*sinr*/, UanTxMode txMode)
{
    UanHeaderCommon header;
    pkt->RemoveHeader(header);
    ...
}
```

- If a function's parameter is only used in certain cases (e.g., logging), or it is part of the function's Doxygen, mark it as `[[maybe_unused]]`.

```
void
TcpSocketBase::CompleteFork(Ptr<Packet> p [[maybe_unused]],
                            const TcpHeader& h,
                            const Address& fromAddress,
                            const Address& toAddress)
{
    NS_LOG_FUNCTION(this << p << h << fromAddress << toAddress);

    // Remaining code that definitely uses 'h', 'fromAddress' and 'toAddress'
    ...
}
```

- If a local variable saves the result of a function that must always run, but whose value may not be used, declare it `[[maybe_unused]]`.

```
void
MyFunction()
{
    int result [[maybe_unused]] = MandatoryFunction();
    NS_LOG_DEBUG("result = " << result);
}
```

- If a local variable saves the result of a function that is only run in certain cases, prefer to not declare the variable and use the function's return value directly where needed. This avoids unnecessarily calling the function if its result is not used.

```
void
MyFunction()
{
    // Prefer to call GetDebugInfo() directly on the log statement
    NS_LOG_DEBUG("Debug information: " << GetDebugInfo());

    // Avoid declaring a local variable with the result of GetDebugInfo()
    int debugInfo [[maybe_unused]] = GetDebugInfo();
    NS_LOG_DEBUG("Debug information: " << debugInfo);
}
```

If the calculation of the maybe unused variable is complex, consider wrapping the calculation of its value in a conditional block that is only run if the variable is used.

```
if (g_log.IsEnabled(ns3::LOG_DEBUG))
{
    auto debugInfo = GetDebugInfo();
    auto value = DoComplexCalculation(debugInfo);

    NS_LOG_DEBUG("The value is " << value);
}
```

6.4.21 Unnecessary else after return

In order to increase readability and avoid deep code nests, consider not adding an `else` block if the `if` block breaks the control flow (i.e., when using `return`, `break`, `continue`, etc.).

For instance, the following code:

```
if (n < 0)
{
    return false;
}
else
{
    n += 3;
    return n;
}
```

can be rewritten as:

```
if (n < 0)
{
    return false;
}

n += 3;
return n;
```

6.4.22 Boolean Simplifications

In order to increase readability and performance, avoid unnecessarily complex boolean expressions in if statements and variable declarations.

For instance, the following code:

```
bool IsPositive(int n)
{
    if (n > 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void ProcessNumber(int n)
{
    if (IsPositive(n) == true)
    {
        ...
    }
}
```

can be rewritten as:

```
bool IsPositive(int n)
{
    return n > 0;
}

void ProcessNumber(int n)
{
    if (IsPositive(n))
    {
        ...
    }
}
```

6.4.23 Smart pointer boolean comparisons

As explained in this issue, the *ns-3* smart pointer class `Ptr` should be used in boolean comparisons as follows:

```
for Ptr<> p, do not use:            use instead:
========================            ================================
if (p != nullptr) {...}             if (p)      {...}
if (p != NULL)    {...}
if (p != 0)       {...}             if (p)      {...}

if (p == nullptr) {...}             if (!p)     {...}
if (p == NULL)    {...}
if (p == 0)       {...}

NS_ASSERT...(p != nullptr, ...)     NS_ASSERT...(p, ...)
NS_ABORT... (p != nullptr, ...)     NS_ABORT... (p, ...)

NS_ASSERT...(p == nullptr, ...)     NS_ASSERT...(!p, ...)
NS_ABORT... (p == nullptr, ...)     NS_ABORT... (!p, ...)

NS_TEST...  (p, nullptr, ...)       NS_TEST...  (p, nullptr, ...)
```

6.4.24 Code performance tips

While developing code, consider the following tips to improve the code's performance. Some tips are general recommendations, but are not strictly enforced. Other tips are enforced by clang-tidy. Please refer to the clang-tidy section below for more details.

- Prefer to use `.emplace_back()` over `.push_back()` to optimize performance.
- When initializing STL containers (e.g., `std::vector`) with known size, reserve memory to store all items, before pushing them in a loop.

```
constexpr int N_ITEMS = 5;

std::vector<int> myVector;
myVector.reserve(N_ITEMS); // Reserve memory to store all items

for (int i = 0; i < N_ITEMS; i++)
{
    myVector.emplace_back(i);
}
```

- Prefer to initialize STL containers (e.g., `std::vector`, `std::map`, etc.) directly through the constructor or with a braced-init-list, instead of pushing elements one-by-one.

```
// Prefer to initialize containers directly
std::vector<int> myVector1{1, 2, 3};
std::vector<int> myVector2(myVector1.begin(), myVector1.end());
std::vector<bool> myVector3(myVector2.size(), true);

// Avoid pushing elements one-by-one
std::vector<int> myVector1;
```

(continues on next page)

(continued from previous page)

```
myVector1.reserve(3);
myVector1.emplace_back(1);
myVector1.emplace_back(2);
myVector1.emplace_back(3);

std::vector<int> myVector2;
myVector2.reserve(myVector1.size());
for (const auto& v : myVector1)
{
    myVector2.emplace_back(v);
}

std::vector<bool> myVector3;
myVector3.reserve(myVector1.size());
for (std::size_t i = 0; i < myVector1.size(); i++)
{
    myVector3.emplace_back(true);
}
```

- When looping through containers, prefer to use const-ref syntax over copying elements.

```
std::vector<int> myVector{1, 2, 3};

for (const auto& v : myVector) { ... }  // OK
for (auto v : myVector) { ... }         // Avoid
```

- Prefer to use the `empty()` function of STL containers (e.g., `std::vector`), instead of the condition `size() > 0`, to avoid unnecessarily calculating the size of the container.
- Avoid unnecessary calls to the functions `.c_str()` and `.data()` of `std::string`.
- Avoid unnecessarily dereferencing std smart pointers (`std::shared_ptr`, `std::unique_ptr`) with calls to their member function `.get()`. Prefer to use the std smart pointer directly where needed.

```
auto ptr = std::make_shared<Node>();

// OK
if (ptr) { ... }

// Avoid
if (ptr.get()) { ... }
```

- Consider caching frequently-used results (especially expensive calculations, such as mathematical functions) in a temporary variable, instead of calculating them in every loop.

```
// Prefer to cache intermediate results
const double sinTheta = std::sin(theta);
const double cosTheta = std::cos(theta);

for (uint8_t i = 0; i < NUM_VALUES; i++)
{
    double power = std::pow(2, i);

    array1[i] = (power * sinTheta) + cosTheta;
```

(continues on next page)

(continued from previous page)

```
    array2[i] = (power * cosTheta) + sinTheta;
}

// Avoid repeating calculations
for (uint8_t i = 0; i < NUM_VALUES; i++)
{
    array1[i] = (std::pow(2, i) * std::sin(theta)) + std::cos(theta);
    array2[i] = (std::pow(2, i) * std::cos(theta)) + std::sin(theta);
}
```

- Do not include inline implementations in header files; put all implementation in a `.cc` file (unless implementation in the header file brings demonstrable and significant performance improvement).
- Avoid declaring trivial destructors, to optimize performance.
- When declaring default destructors with `~Class() = default;`, be aware that classes derived from `SimpleRefCount<T>` must have this declaration on the source file (`.cc`). The header file (`.h`) should contain the plain destructor declaration `~Class();`. This is due to PIMPL's opaque pointer, as explained in Andrea Fertig's blog post When an empty destructor is required. See class WifiPpdu's destructor for an example.

6.4.25 C++ standard

As of ns-3.36, *ns-3* permits the use of C++-17 (or earlier) features in the implementation files.

If a developer would like to propose to raise this bar to include more features than this, please email the developers list. We will move this language support forward as our minimally supported compiler moves forward.

6.4.26 Guidelines for using maps

Maps (associative containers) are used heavily in ns-3 models to store key/value pairs. The C++ standard, over time, has added various methods to insert elements to maps, and the ns-3 codebase has made use of most or all of these constructs. For the sake of uniformity and readability, the following guidelines are recommended for any new code.

Prefer the use of `std::map` to `std::unordered_map` unless there is a measurable performance advantage. Use `std::unordered_map` only for use cases in which the map does not need to be iterated or the iteration order does not affect the results of the operation (because different implementations of the hash function may lead to different iteration orders on different systems).

Keep in mind that C++ now allows several methods to insert values into maps, and the behavior can be different when a value already exists for a key. If the intended behavior is that the insertion should not overwrite an existing value for the key, `try_emplace()` can be a good choice. If the intention is to allow the overwriting of a key/value pair, `insert_or_assign()` can be a good choice. Both of the above methods provide return values that can be checked– in the case of `try_emplace()`, whether the insertion succeeded or did not occur, and in the case of `insert_or_assign()`, whether an insertion or assignment occurred.

6.4.27 Miscellaneous items

- `NS_LOG_COMPONENT_DEFINE("log-component-name");` statements should be placed within `namespace ns3` (for module code) and after the `using namespace ns3;`. In examples, `NS_OBJECT_ENSURE_REGISTERED()` should also be placed within `namespace ns3`.
- Pointers and references are left-aligned:

```
int x = 1;
int* ptr = &x;
int& ref = x;
```

- Use a trailing comma in braced-init-lists, so that each item is positioned in a new line.

```
const std::vector<std::string> myVector{
  "string-1",
  "string-2",
  "string-3",
};

const std::map<int, std::string> myMap{
  {1, "string-1"},
  {2, "string-2"},
  {3, "string-3"},
};
```

- Const reference syntax:

```
void MySub(const T& t);  // OK
void MySub(T const& t);  // Not OK
```

- Do not use `NULL`, `nil` or `0` constants; use `nullptr` (improves portability)
- Consider whether you want the default constructor, copy constructor, or assignment operator in your class. If not, explicitly mark them as deleted and make the declaration public:

```
class MyClass
{
  public:
    // Allowed constructors
    MyClass(int i);

    // Deleted constructors.
    // Explain why they are not supported.
    MyClass() = delete;
    MyClass(const MyClass&) = delete;
    MyClass& operator=(const MyClass&) = delete;
};
```

- Avoid returning a reference to an internal or local member of an object:

```
MyType& foo();        // Avoid. Prefer to return a pointer or an object.
const MyType& foo();  // Same as above.
```

  This guidance does not apply to the use of references to implement operators.

- Expose class members through access functions, rather than direct access to a public object. The access functions are typically named `Get` and `Set` followed by the member's name. For example, a member `m_delayTime` might have accessor functions `GetDelayTime()` and `SetDelayTime()`.
- Do not bring the C++ standard library namespace into *ns-3* source files by using the `using namespace std;` directive.
- Do not use the C++ `goto` statement.
- Do not add the `enum` or `struct` specifiers when declaring the variable's type.
- Do not unnecessarily add `typedef` to `struct` or `enum`.

```
// Avoid
typedef struct
{
    ...
} MyStruct;

// Prefer
struct MyStruct
{
    ...
};
```

- When checking whether a Time value is zero, use `Time::IsZero()` rather than comparing it to a zero-valued time object with `operator==`, to avoid construction of a temporary. Similar guidance applies to the related functions `Time::IsPositive()`, `Time::IsNegative()`, `Time::IsStrictlyPositive`, and `Time::IsStrictlyNegative()`.

```
Time t = ...;
// prefer the below:
if (t.IsStrictlyPositive())
{...}
// to this alternative:
if (t > Seconds(0))
{...}
```

- Use `std::array` instead of C-style arrays, e.g.:

```
// Avoid
uint8_t myArray[16];

// Prefer
std::array<uint8_t, 16> myArray;
```

  This enables many useful operations without writing extra code, such as copy constructors, default comparison operators, etc.

- In C++ 20, a new comparison operator `<=>` ... was added, and a new pattern started to become a C++ best practice: to avoid defining operators separately but to make use of a member declaration such as:

```
#include <compare>
struct IntWrapper
{
  int value{0};
```

(continues on next page)

(continued from previous page)

```
  constexpr IntWrapper(int value): value{value} { }
  auto operator<=>(const IntWrapper&) const = default;
  bool operator==(const IntWrapper&) const = default; // Unnecessary, derived from
↪<=>
  bool operator!=(const IntWrapper&) const = default; // Unnecessary, derived from␣
↪==
};
```

For new ns-3 code, we recommend using this operator where possible. There are cases in which the default does not apply, and you can read about some of them here and here. You may notice in the ns-3 codebase that most code defines operators separately, because the code predates C++20. Some of this code may be changed to use <=> over time.

Two examples are in `ns3::LollipopCounter`, and `ns3::SequenceNumber`.

6.4.28 Clang-tidy rules

Please refer to the `.clang-tidy` file in the *ns-3* main directory for the full list of rules that should be observed while developing code.

Some rules are explained in the corresponding sections above. The remaining rules are explained here.

- Explicitly mark inherited functions with the `override` specifier.
- When creating STL smart pointers, prefer to use `std::make_shared` or `std::make_unique`, instead of creating the smart pointer with `new`.

```
auto node = std::make_shared<Node>();           // OK
auto node = std::shared_ptr<Node>(new Node());  // Avoid
```

- When looping through containers, prefer to use range-based for loops rather than index-based loops.

```
std::vector<int> myVector{1, 2, 3};

for (const auto& v : myVector) { ... }             // Prefer
for (int i = 0; i < myVector.size(); i++) { ... }  // Avoid
```

- Avoid accessing class static functions and members through objects. Instead, prefer to access them through the class.

```
// OK
MyClass::StaticFunction();

// Avoid
MyClass myClass;
MyClass.StaticFunction();
```

- Prefer using type traits in short form `traits_t<...>` and `traits_v<...>`, instead of the long form `traits<...>::type` and `traits<...>::value`, respectively.

```
// Prefer using the shorter version of type traits
std::is_same_v<int, float>
std::is_integral_v<T>
std::enable_if_t<std::is_integral_v<T>, Time>
```

(continues on next page)

(continued from previous page)

```
// Avoid the longer form of type traits
std::is_same<int, float>::value
std::is_integral<T>::value
std::enable_if<std::is_integral<T>::value, Time>::type
```

- Avoid using integer values (`1` or `0`) to represent boolean variables (`true` or `false`), to improve code readability and avoid implicit conversions.
- Prefer to use `static_assert()` over `NS_ASSERT()` when conditions can be evaluated at compile-time.
- Prefer using transparent functors to non-transparent ones, to avoid repeating the type name. This improves readability and avoids errors when refactoring code.

```
// Prefer using transparent functors
std::map<MyClass, int, std::less<>> myMap;

// Avoid repeating the type name "MyClass" in std::less<>
std::map<MyClass, int, std::less<MyClass>> myMap;
```

- In conditional control blocks (i.e., if-else and switch-case), avoid declaring multiple branch conditions with the same content to avoid duplicating code.

  In if-else blocks, prefer grouping the identical bodies in a single if condition with a disjunction of the multiple conditions.

```
if (condition1)
{
    Foo();
}
else if (condition2)
{
    // Same body as condition 1
    Foo();
}
else
{
    Bar();
}

// Prefer grouping the two conditions
if (condition1 || condition2)
{
    Foo();
}
else
{
    Bar();
}
```

  In switch-case blocks, prefer grouping identical `case` labels by removing the duplicate bodies of the former `case` labels.

```
switch (condition)
{
case 1:
    Foo();
    break;
case 2: // case 2 has the same body as case 1
    Foo();
    break;
case 3:
    Bar();
    break;
}

switch (condition)
{
// Group identical cases by removing the content of case 1 and letting it␣
↪fallthrough to case 2
case 1:
case 2:
    Foo();
    break;
case 3:
    Bar();
    break;
}
```

6.5 CMake file formatting

The `CMakeLists.txt` and other `*.cmake` files follow the formatting rules defined in `build-support/cmake-format.yaml` and `build-support/cmake-format-modules.yaml`.

The first set of rules applies to CMake files in all directories that are not modules, while the second one applies to files within modules.

Those rules are enforced via the cmake-format tool, that can be installed via Pip.

```
pip install cmake-format pyyaml
```

After installing cmake-format, it can be called to fix the formatting of a CMake file with the following command:

```
cmake-format -c ./build-support/cmake-format.yaml CMakeLists.txt
```

To check the formatting, add the *–check* option to the command, before specifying the list of CMake files.

Instead of calling this command for every single CMake file, it is recommended to use the `ns3` script to run the custom targets that do that automatically.

```
# Check CMake formatting
./ns3 build cmake-format-check

# Check and fix CMake formatting
./ns3 build cmake-format
```

Custom functions and macros need to be explicitly configured in the `cmake-format.yaml` files, otherwise their formatting will be broken.

6.6 Python file formatting

Python format style and rule enforcement is based on the default settings for the Black formatter tool and Isort import sorter tool. Black default format is detailed in Black current style.

The custom settings for both tools are set in the `pyproject.toml` file.

These tools that can be installed via Pip, using the following command:

```
pip install black isort
```

To check the formatting, add the *–check* option to the command:

```
black --check .
isort --check .
```

To check and fix the formatting, run the commands as follows:

```
black .
isort .
```

For VS Code users, MS Black formatter and MS Isort extensions, which repackage Black and Isort for VS Code, can be installed to apply fixes regularly. To configure VS Code to automatically format code when saving, editing or pasting code, add the following configuration to `.vscode/settings.json`:

```
{
  "editor.formatOnPaste": true,
  "editor.formatOnSave": true,
  "editor.formatOnType": true,
  "[python]": {
    "editor.defaultFormatter": "ms-python.black-formatter",
    "editor.codeActionsOnSave": {
        "source.organizeImports": "explicit",
    },
  },
  "black-formatter.args": [
    "--config",
    "pyproject.toml",
  ],
  "isort.check": true,
  "isort.args": [
    "--sp",
    "pyproject.toml",
  ],
}
```

6.7 Markdown Lint

ns-3 uses MarkdownLint as a linter of Markdown files. This linter checks if Markdown files follow a set of defined rules, in order to encourage standardization and consistency of Markdown files across parsers. It also ensures that Markdown files are correctly interpreted and rendered.

MarkdownLint detects linting issues and can fix most of them automatically. Some issues may need to be manually fixed.

6.7.1 MarkdownLint configuration

MarkdownLint's settings are saved in the file `.markdownlint.yml`. This schema of this file is defined in MarkdownLint Configuration File, which explains how to customize the tool to enable / disable rules or customize its parameters.

The list of Markdown rules supported by MarkdownLint is available in MarkdownLint Rules.

6.7.2 Install and Run MarkdownLint

MarkdownLint is written in NodeJS. To run MarkdownLint, either use the official MarkdownLint Docker image, install it natively in macOS via Homebrew, or install MarkdownLint with NodeJS / npm.

Run MarkdownLint with Docker image

MarkdownLint has an official Docker image in MarkdownLint Docker with the tool and all dependencies installed.

To run MarkdownLint in a Docker container, use the following command:

```
# Check all Markdown files in the current directory and subdirectories
docker run --rm -v $PWD:/workdir ghcr.io/igorshubovych/markdownlint-cli:latest . [--fix]

# Check specific Markdown file
docker run --rm -v $PWD:/workdir ghcr.io/igorshubovych/markdownlint-cli:latest PATH_TO_
↪FILE [--fix]
```

If the `fix` flag is used, the tool tries to automatically fix the detected issues. Otherwise, it only reports the issues found.

Install and Run MarkdownLint natively

To install MarkdownLint natively, either on macOS via Homebrew or using NodeJS / npm, follow the instructions available in MarkdownLint Installation.

To run MarkdownLint, use the following command:

```
# Check all Markdown files in the current directory and subdirectories
markdownlint-cli . [--fix]

# Check specific Markdown file
markdownlint-cli PATH_TO_FILE [--fix]
```

6.7.3 VS Code Extension

For VS Code users, the MarkdownLint VS Code Extension extension is available in the marketplace. This extension uses the same engine and respects the configuration file.

The MarkdownLint extension automatically analyzes files open in the editor and provides inline hints when issues are detected. It can automatically fix most issues related with formatting. As explained in the "Integration with IDEs" section, VS Code can be configured to automatically format code when saving, editing or pasting code.

CHAPTER

SEVEN

BEST PRACTICES

When writing code to be contributed to the *ns-3* open source project, we recommend following the best practices listed below. These practices apply to different phases of software development and testing, and are organized according to the typical application lifecycle.

7.1 Development phase

7.1.1 Refactor-or-die: Preliminary refactoring before introducing functional changes

As presented in Mikhail Matrosov's "Refactor or die" talk in CppCon 2017 [*1*], mixing refactoring and functional changes is a bad practice.

It is considered a bad practice because refactoring may involve a lot of different files, classes, API changes to accommodate new parameters, for example. If a functional change is mixed in between, there is a higher probability of introducing a bug. This results in more code to inspect and debug to isolate and fix the bug. Which translates to unproductive work to isolate and fix the bug, which results in extra hours, working on weekends, etc, etc.

The figure below depicts the scope differences in refactoring and functional changes.

This is why refactoring should be done as a preliminary step, before functional changes. This will drastically reduce the number of lines of code which may have introduced a potential bug, saving development/maintenance time in the long run.

The figure below depicts the effects of mixed refactoring AND functional changes (left orange rectangle), versus the preliminary refactoring (right green rectangle) PLUS functional changes (right orange rectangle).

Compile-or-die: Individual commit compilation and testing

Since preliminary refactoring is not always trivial, due to the potentially unknown requirements, it is very unlikely to completely avoid mixing refactoring and functional changes. However, these changes can and should be separated before merging them upstream.

This can be achieved through commit history rewriting, which can also introduce new bugs, if commits are not properly rewritten.

A suggested practice is to at least compile and run tests for every single commit in the branch before merging it. This can be accomplished by checking out each commit, hard-resetting to that commit, reconfiguring the project and running test.py.

This can be done automatically using *./ns3 –compile-or-die base_commit head_commit*.

Note: ALWAYS back up your ns-3 directory and sync your local branches with the remote server.

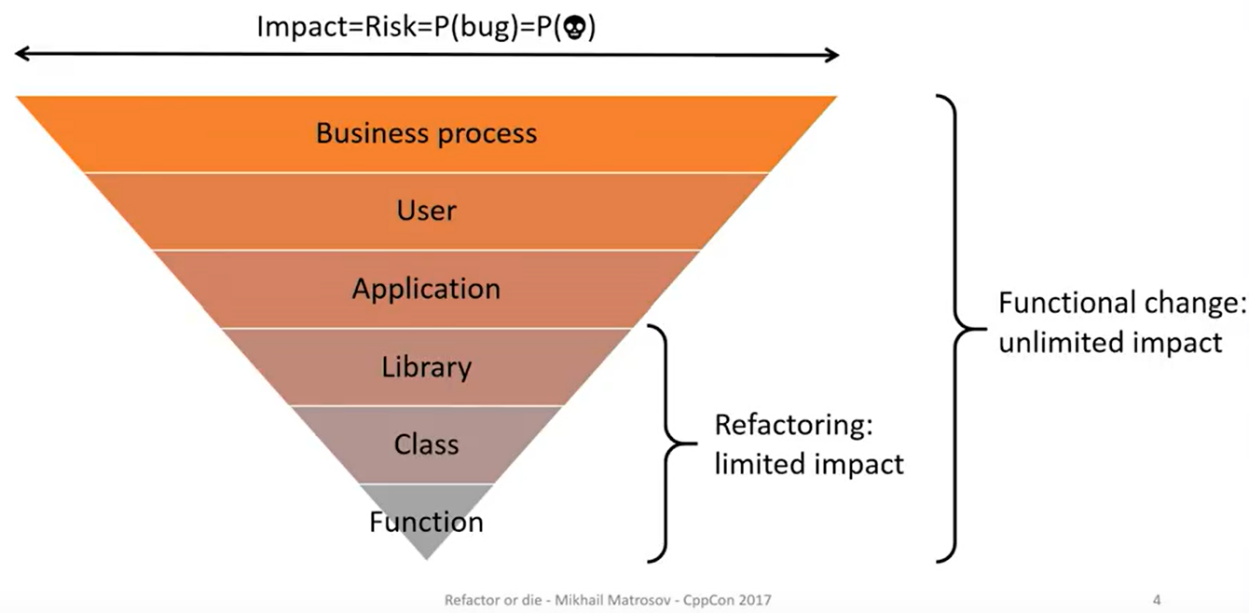

Fig. 1: Impact of different scope changes (inverted triangle), and scopes affected by refactoring and functional changes. [1]

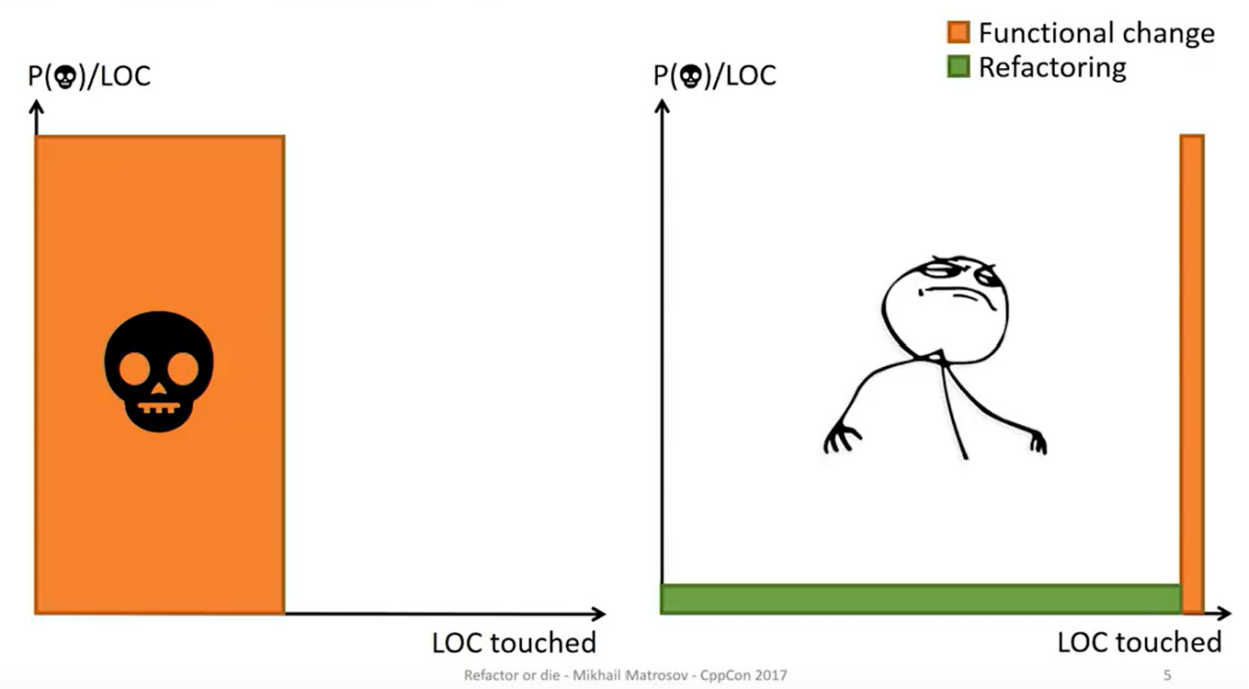

Fig. 2: Assume the existence of a bug kills you. If you mix functional and refactoring changes, the number of lines potentially containing a bug increases. As such, the probability of dying is unnecessarily augmented. If you do not mix these changes, the probability of dying is significantly reduced. [1]

```
~ns-3-dev/$ ./ns3 --compile-or-die 757a2bfc2abb5f3584593609434c40e5ac678e8e␣
↪5a30398332d70646279285a2ef8997cea0ed9e43
Compile-or-die with commits: ['757a2bfc2abb5f3584593609434c40e5ac678e8e',
↪'5a30398332d70646279285a2ef8997cea0ed9e43']
        Testing commit 757a2bfc2abb5f3584593609434c40e5ac678e8e
        Testing commit 5a30398332d70646279285a2ef8997cea0ed9e43
```

In case there are uncommitted changes, the script won't continue to prevent potential data loss. In case there is no associated branch with the current git head, a backup branch will be created to prevent data loss. Then it will create a new temporary test branch, that will be used to checkout and test each commit between base and head commits.

If there is no error message, all commits succeed. Which only means there is no issue preventing compilation or causing one of the existing tests to fail. That is, we have anecdotal evidence that refactoring was done correctly.

Two branches are created automatically: the current HEAD is tagged as `compileOrDieBackup` if not currently tagged, and the commit being currently tested is tagged as `compileOrDieTest`. These won't be cleaned automatically to prevent potential data loss. So users should verify and delete tags manually.

[1] Mikhail Matrosov. Refactor or die. CppCon 2017. Available in YouTube.