# Contributing guidelines

If you have improvements to the oneDNN code, please send us your pull

requests! For getting started, see GitHub

[howto](https://help.github.com/en/articles/about-pull-requests).

The current guidelines are work in progress.

## Pull request checklist

Before sending your pull requests, please make sure that you followed this

list.

\* If you are contributing a new compute primitive, check the

[library functionality guidelines](CONTRIBUTING.md#library\_functionality\_guidelines).

It is strongly advised to first open an

[RFC pull request](CONTRIBUTING.md#RFC\_pull\_requests) with a

detailed explanation of expected use cases and performance benefits.

\* Ensure that the changes are consistent with the

[code contribution guidelines](CONTRIBUTING.md#code\_contribution\_guidelines).

\* Check that the changes are consistent with the

[coding style](CONTRIBUTING.md#coding\_style).

\* Check that [unit tests](CONTRIBUTING.md#unit\_tests) pass.

## Library functionality guidelines

oneDNN focuses on functionality that satisfies all of the following

criteria:

1. \*Performance\*: the functionality has material impact on a workload level.

In other words, this means that for a new primitive it should be

demonstrated that it brings visible performance improvement to some

workload.

2. \*Generality\*: the functionality is useful in a wide range of deep learning

applications. This implies that when introducing a new primitive, its API

needs to be general enough to be integrated into multiple deep learning

frameworks that have similar functionality.

3. \*Complexity\*: it is not trivial to implement the functionality directly in

a deep learning application.

### RFC pull requests

Significant library changes (new primitives, library architecture changes,

API modifications, etc) require approval from oneDNN maintainers before

opening a Pull Request with such implementation. For that we use the Request

For Comments (RFC) process, which consists of opening, discussing, and

accepting (promoting) RFC pull requests.

More information about the process can be found in the dedicated

[`rfcs`](https://github.com/oneapi-src/oneDNN/tree/rfcs) branch.

## Code contribution guidelines

The code must be:

\* \*Tested\*: oneDNN uses gtests for lightweight functional testing and

benchdnn for functionality that requires both performance and functional

testing.

\* \*Documented\*: oneDNN uses Doxygen for inline comments in public header

files that is used to build reference manual and markdown (also processed by

Doxygen) for user guide.

\* \*Portable\*: oneDNN supports different operating systems, CPU and GPU

architectures, compilers, and run-times. The new code should be compliant

with the [System Requirements](README.md#system-requirements).

## Coding style

The general principle is to follow the style of existing / surrounding code.

Particularly:

\* Use 4-space indentation.

\* Limit line length to 80 columns.

\* Do put spaces after `if`, `for`, `switch`; otherwise, do not put spaces

around braces, parenthesis, square or angle brackets.

\* Do put spaces around binary arithmetic operators.

\* Avoid trailing and double spaces (unless used for indentation).

\* Do not indent namespaces, `private:`, `public:`, `protected:` and case

labels.

\* Keep opening brace on the same line as the statement or function.

If in doubt, use the `clang-format`:

```sh

clang-format -style=file -i foo.cpp

```

This will format code using the `\_clang\_format` file found in the oneDNN

top level directory.

Coding style is secondary to the general code design.

## Unit tests

oneDNN uses gtests for lightweight functional testing and benchdnn for

performance and functional testing.

Be sure to extend the existing tests when fixing an issue.

Developing new benchdnn tests can be hard, so it is a good idea to start with

gtests first.