Contributing to Neovim

Getting started

If you want to help but don't know where to start, here are some low-risk/isolated tasks:

- Try a complexity:low issue.
- Fix bugs found by Clang, PVS or Coverity.
- Improve documentation
- Merge a Vim patch (Familiarity with Vim is *strongly* recommended)

Reporting problems

- Check the FAQ.
- Search existing issues (including closed!)
- Update Neovim to the latest version to see if your problem persists.
- Try to reproduce with `nvim --clean` ("factory defaults").
- Bisect your config: disable plugins incrementally, to narrow down the cause of the issue.
- Bisect Neovim's source code to find the cause of a regression, if you can. This is *extremely* helpful.
- When reporting a crash, include a stacktrace.
- Use ASAN/UBSAN to get detailed errors for segfaults and undefined behavior.
- Check the logs. `:edit $NVIM_LOG_FILE`
- Include `cmake --system-information` for build-related issues.

Developer guidelines

- Read `:help dev` if you are working on Nvim core.
- Read `:help dev-ui` if you are developing a UI.
- Read `:help dev-api-client` if you are developing an API client.
- Install `ninja` for faster builds of Nvim.

  ```
  sudo apt-get install ninja-build
  make distclean
  make  # Nvim build system uses ninja automatically, if available.
  ```

Pull requests (PRs)

- To avoid duplicate work, create a draft pull request.
- Your PR must include test coverage.
- Avoid cosmetic changes to unrelated files in the same commit.
- Use a feature branch instead of the master branch.
- Use a **rebase workflow** for small PRs.
  - After addressing review comments, it's fine to rebase and force-push.
- Use a **merge workflow** for big, high-risk PRs.
  - Merge `master` into your PR when there are conflicts or when master introduces breaking changes.
  - Use the `ri` git alias:

    ```
    [alias]
    ri = "!sh -c 't=\"${1:-master}\"; s=\"${2:-HEAD}\"; mb=\"$(git merge-
    ```

```
base \"$t\" \"$s\")\"; if test \"x$mb\" = x ; then o=\"$t\"; else
lm=\"$(git log -n1 --merges \"$t..$s\" --pretty=%H)\"; if test \"x$lm\"
= x ; then o=\"$mb\"; else o=\"$lm\"; fi; fi; test $# -gt 0 && shift;
test $# -gt 0 && shift; git rebase --interactive \"$o\" \"$@\"'"
```

  This avoids unnecessary rebases yet still allows you to combine related commits, separate monolithic commits, etc.
  - Do not edit commits that come before the merge commit.
- During a squash/fixup, use `exec make -C build unittest` between each pick/edit/reword.

Stages: Draft and Ready for review

Pull requests have two stages: Draft and Ready for review.

1. Create a Draft PR while you are *not* requesting feedback as you are still working on the PR.
   - You can skip this if your PR is ready for review.
2. Change your PR to ready when the PR is ready for review.
   - You can convert back to Draft at any time.

Do **not** add labels like `[RFC]` or `[WIP]` in the title to indicate the state of your PR: this just adds noise. Non-Draft PRs are assumed to be open for comments; if you want feedback from specific people, `@`-mention them in a comment.

Commit messages

Follow the conventional commits guidelines to *make reviews easier* and to make the VCS/git logs more valuable. The general structure of a commit message is:

```
<type>([optional scope]): <description>

[optional body]

[optional footer(s)]
```

- Prefix the commit subject with one of these *types*:
  - `build`, `ci`, `docs`, `feat`, `fix`, `perf`, `refactor`, `revert`, `test`, `vim-patch`, `chore`
  - You can **ignore this for "fixup" commits** or any commits you expect to be squashed.
- Append optional scope to *type* such as `(lsp)`, `(treesitter)`, `(float)`, ...
- *Description* shouldn't start with a capital letter or end in a period.
- Use the *imperative voice*: "Fix bug" rather than "Fixed bug" or "Fixes bug."
- Try to keep the first line under 72 characters.
- A blank line must follow the subject.
- Breaking API changes must be indicated by
  1. "!" after the type/scope, and
  2. a "BREAKING CHANGE" footer describing the change. Example:

  ```
  refactor(provider)!: drop support for Python 2

  BREAKING CHANGE: refactor to use Python 3 features since Python 2 is no
  longer supported.
  ```

Automated builds (CI)

Each pull request must pass the automated builds on sourcehut and GitHub Actions.

- CI builds are compiled with `-Werror`, so compiler warnings will fail the build.
- If any tests fail, the build will fail. See test/README.md#running-tests to run tests locally. Passing locally doesn't guarantee passing the CI build, because of the different compilers and platforms tested against.
- CI runs ASan and other analyzers.
  - To run valgrind locally: `VALGRIND=1 make test`
  - To run Clang ASan/UBSan locally: `CC=clang make CMAKE_FLAGS="-DCLANG_ASAN_UBSAN=ON"`
- The lint build checks modified lines *and their immediate neighbors*, to encourage incrementally updating the legacy style to meet our style. (See #3174 for background.)
- CI for freebsd and openbsd runs on sourcehut.
  - To get a backtrace on freebsd (after connecting via ssh):

    ```
    sudo pkg install tmux  # If you want tmux.
    lldb build/bin/nvim -c nvim.core

    # To get a full backtrace:
    #   1. Rebuild with debug info.
    rm -rf nvim.core build
    gmake CMAKE_BUILD_TYPE=RelWithDebInfo CMAKE_EXTRA_FLAGS="-DCI_BUILD=ON
    -DMIN_LOG_LEVEL=3" nvim
    #   2. Run the failing test to generate a new core file.
    TEST_FILE=test/functional/foo.lua gmake functionaltest
    lldb build/bin/nvim -c nvim.core
    ```

Clang scan-build

View the Clang report to see potential bugs found by the Clang scan-build analyzer.

- Search the Neovim commit history to find examples:

  ```
  git log --oneline --no-merges --grep clang
  ```

- To verify a fix locally, run `scan-build` like this:

  ```
  rm -rf build/
  scan-build --use-analyzer=/usr/bin/clang make
  ```

PVS-Studio

View the PVS report to see potential bugs found by PVS Studio.

- Use this format for commit messages (where `{id}` is the PVS warning-id)):

  ```
  fix(PVS/V{id}): {description}
  ```

- Search the Neovim commit history to find examples:

```
git log --oneline --no-merges --grep PVS
```

- Try `./scripts/pvscheck.sh` to run PVS locally.

Coverity

Coverity runs against the master build. To view the defects, just request access; you will be approved.

- Use this format for commit messages (where `{id}` is the CID (Coverity ID); (example)):

  ```
  fix(coverity/{id}): {description}
  ```

- Search the Neovim commit history to find examples:

  ```
  git log --oneline --no-merges --grep coverity
  ```

Clang sanitizers (ASAN and UBSAN)

ASAN/UBSAN can be used to detect memory errors and other common forms of undefined behavior at runtime in debug builds.

- To build Neovim with sanitizers enabled, use

  ```
  rm -rf build && CMAKE_EXTRA_FLAGS="-DCMAKE_C_COMPILER=clang -
  DCLANG_ASAN_UBSAN=1" make
  ```

- When running Neovim, use

  ```
  UBSAN_OPTIONS=print_stacktrace=1 ASAN_OPTIONS=log_path=/tmp/nvim_asan nvim
  args...
  ```

- If Neovim exits unexpectedly, check `/tmp/nvim_asan.{PID}` (or your preferred `log_path`) for log files with error messages.

Coding

Lint

You can run the linter locally by:

```
make lint
```

The lint step downloads the master error list and excludes them, so only lint errors related to the local changes are reported.

You can lint a single file (but this will *not* exclude legacy errors):

```
./src/clint.py src/nvim/ops.c
```

Style

- Style rules are (mostly) defined by `src/uncrustify.cfg` which tries to match the style-guide. To use the Nvim `gq` command with `uncrustify`:

```
if !empty(findfile('src/uncrustify.cfg', ';'))
  setlocal formatprg=uncrustify\ -q\ -l\ C\ -c\ src/uncrustify.cfg\ --no-backup
endif
```

The required version of `uncrustify` is specified in `uncrustify.cfg`.

- There is also `.clang-format` which has drifted from the style-guide, but is available for reference. To use the Nvim `gq` command with `clang-format`:

```
if !empty(findfile('.clang-format', ';'))
  setlocal formatprg=clang-format\ -style=file
endif
```

Navigate

- Set `blame.ignoreRevsFile` to ignore noise commits in git blame:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

- Use **universal-ctags**. ("Exuberant ctags", the typical `ctags` binary provided by your distro, is unmaintained and won't recognize many function signatures in Neovim source.)
- Explore the source code on the web.
- If using lua-language-server, symlink `contrib/luarc.json` into the project root:

```
$ ln -s contrib/luarc.json .luarc.json
```

Reviewing

To help review pull requests, start with this checklist.

Reviewing can be done on GitHub, but you may find it easier to do locally. Using GitHub CLI, you can create a new branch with the contents of a pull request, e.g. #1820:

```
gh pr checkout https://github.com/neovim/neovim/pull/1820
```

Use `git log -p master..FETCH_HEAD` to list all commits in the feature branch which aren't in the `master` branch; `-p` shows each commit's diff. To show the whole surrounding function of a change as context, use the `-W` argument as well.