We welcome contributions from the community. Please read the following guidelines carefully to

maximize the chances of your PR being merged.

# Communication

\* Before starting work on a major feature, please reach out to us via GitHub, Slack,

email, etc. We will make sure no one else is already working on it and ask you to open a

GitHub issue.

\* A "major feature" is defined as any change that is > 100 LOC altered (not including tests), or

changes any user-facing behavior. We will use the GitHub issue to discuss the feature and come to

agreement. This is to prevent your time being wasted, as well as ours. The GitHub review process

for major features is also important so that [organizations with commit access](OWNERS.md) can

come to agreement on design. If it is appropriate to write a design document, the document must

be hosted either in the GitHub tracking issue, or linked to from the issue and hosted in a

world-readable location.

\* Specifically, if the goal is to add a new [extension](REPO\_LAYOUT.md#sourceextensions-layout),

please read the [extension policy](GOVERNANCE.md#extension-addition-policy).

\* Small patches and bug fixes don't need prior communication.

# Coding style

\* See [STYLE.md](STYLE.md)

# Inclusive language policy

The Envoy community has an explicit goal to be inclusive to all. As such, all PRs must adhere to the

following guidelines for all code, APIs, and documentation:

\* The following words and phrases are not allowed:

\* \*Whitelist\*: use allowlist instead.

\* \*Blacklist\*: use denylist or blocklist instead.

\* \*Master\*: use primary instead.

\* \*Slave\*: use secondary or replica instead.

\* Documentation should be written in an inclusive style. The [Google developer

documentation](https://developers.google.com/style/inclusive-documentation) contains an excellent

reference on this topic.

\* The above policy is not considered definitive and may be amended in the future as industry best

practices evolve. Additional comments on this topic may be provided by maintainers during code

review.

# Breaking change policy

Both API and implementation stability are important to Envoy. Since the API is consumed by clients

beyond Envoy, it has a distinct set of [versioning guidelines](api/API\_VERSIONING.md). Below, we

articulate the Envoy implementation stability rules, which operate within the context of the API

versioning guidelines:

\* Features may be marked as deprecated in a given versioned API at any point in time, but this may

only be done when a replacement implementation and configuration path is available in Envoy on

master. Deprecators must implement a conversion from the deprecated configuration to the latest

`vNalpha` (with the deprecated field) that Envoy uses internally. A field may be deprecated if

this tool would be able to perform the conversion. For example, removing a field to describe

HTTP/2 window settings is valid if a more comprehensive HTTP/2 protocol options field is being

introduced to replace it. The PR author deprecating the old configuration is responsible for

updating all tests and canonical configuration, or guarding them with the

`DEPRECATED\_FEATURE\_TEST()` macro. This will be validated by the `bazel.compile\_time\_options`

target, which will hard-fail when deprecated configuration is used. The majority of tests and

configuration for a feature should be expressed in terms of the latest Envoy internal

configuration (i.e. `vNalpha`), only a minimal number of tests necessary to validate configuration

translation should be guarded via the `DEPRECATED\_FEATURE\_TEST()` macro.

\* We will delete deprecated configuration across major API versions. E.g. a field marked deprecated

in v2 will be removed in v3.

\* Unless the community and Envoy maintainer team agrees on an exception, during the

first release cycle after a feature has been deprecated, use of that feature

will cause a logged warning, and incrementing the

[runtime](https://www.envoyproxy.io/docs/envoy/latest/configuration/operations/runtime#statistics)

`runtime.deprecated\_feature\_use` stat.

During the second release cycle, use of the deprecated configuration will

cause a configuration load failure, unless the feature in question is

explicitly overridden in

[runtime](https://www.envoyproxy.io/docs/envoy/latest/configuration/operations/runtime#using-runtime-overrides-for-deprecated-features)

config ([example](configs/using\_deprecated\_config.v2.yaml)). Finally, following the deprecation

of the API major version where the field was first

marked deprecated, the entire implementation code will be removed from the Envoy implementation.

\* This policy means that organizations deploying master should have some time to get ready for

breaking changes at the next major API version. This is typically a window of at least 12 months

or until the organization moves to the next major API version.

\* The breaking change policy also applies to source level extensions (e.g., filters). Code that

conforms to the public interface documentation should continue to compile and work within the

deprecation window. Within this window, a warning of deprecation should be carefully logged (some

features might need rate limiting for logging this). We make no guarantees about code or deployments

that rely on undocumented behavior.

\* All deprecations/breaking changes will be clearly listed in the [version history](docs/root/version\_history/).

\* High risk deprecations/breaking changes may be announced to the

[envoy-announce](https://groups.google.com/forum/#!forum/envoy-announce) email list but by default

it is expected the multi-phase warn-by-default/fail-by-default is sufficient to warn users to move

away from deprecated features.

# Submitting a PR

\* Fork the repo.

\* In your local repo, install the git hooks that implement various important pre-commit and

pre-push checks:

```

./support/bootstrap

```

Please see [support/README.md](support/README.md) for more information on these hooks.

\* Create your PR.

\* Tests will automatically run for you.

\* We will \*\*not\*\* merge any PR that is not passing tests.

\* PRs are expected to have 100% test coverage for added code. This can be verified with a coverage

build. If your PR cannot have 100% coverage for some reason please clearly explain why when you

open it.

\* Any PR that changes user-facing behavior \*\*must\*\* have associated documentation in [docs](docs) as

well as [release notes](docs/root/version\_history/current.rst). API changes should be documented

inline with protos as per the [API contribution guidelines](api/CONTRIBUTING.md). If a change applies

to multiple sections of the release notes, it should be noted in the first (most important) section

that applies. For instance, a bug fix that introduces incompatible behavior should be noted in

`Incompatible Behavior Changes` but not in `Bug Fixes`.

\* All code comments and documentation are expected to have proper English grammar and punctuation.

If you are not a fluent English speaker (or a bad writer ;-)) please let us know and we will try

to find some help but there are no guarantees.

\* Your PR title should be descriptive, and generally start with a subsystem name followed by a

colon. Examples:

\* "docs: fix grammar error"

\* "http conn man: add new feature"

\* Your PR commit message will be used as the commit message when your PR is merged. You should

update this field if your PR diverges during review.

\* Your PR description should have details on what the PR does. If it fixes an existing issue it

should end with "Fixes #XXX".

\* If your PR is co-authored or based on an earlier PR from another contributor,

please attribute them with `Co-authored-by: name <name@example.com>`. See

GitHub's [multiple author

guidance](https://help.github.com/en/github/committing-changes-to-your-project/creating-a-commit-with-multiple-authors)

for further details.

\* When all of the tests are passing and all other conditions described herein are satisfied, a

maintainer will be assigned to review and merge the PR.

\* Once you submit a PR, \*please do not rebase it\*. It's much easier to review if subsequent commits

are new commits and/or merges. We squash rebase the final merged commit so the number of commits

you have in the PR don't matter.

\* We expect that once a PR is opened, it will be actively worked on until it is merged or closed.

We reserve the right to close PRs that are not making progress. This is generally defined as no

changes for 7 days. Obviously PRs that are closed due to lack of activity can be reopened later.

Closing stale PRs helps us to keep on top of all of the work currently in flight.

\* If a commit deprecates a feature, the commit message must mention what has been deprecated.

Additionally, the [version history](docs/root/version\_history/current.rst) must be updated with

relevant RST links for fields and messages as part of the commit.

\* Please consider joining the [envoy-dev](https://groups.google.com/forum/#!forum/envoy-dev)

mailing list.

\* If your PR involves any changes to

[envoy-filter-example](https://github.com/envoyproxy/envoy-filter-example) (for example making a new

branch so that CI can pass) it is your responsibility to follow through with merging those

changes back to master once the CI dance is done.

\* If your PR is a high risk change, the reviewer may ask that you runtime guard

it. See the section on runtime guarding below.

# Runtime guarding

Some changes in Envoy are deemed worthy of runtime guarding. Instead of just replacing

old code with new code, both code paths are supported for between one Envoy release (if it is

guarded due to performance concerns) and a full deprecation cycle (if it is a high risk behavioral

change). Generally as a community we try to guard both high risk changes (major

refactors such as replacing Envoy's buffer implementation) and most user-visible

non-config-guarded changes to protocol processing (for example additions or changes to HTTP headers or

how HTTP is serialized out) for non-alpha features. Feel free to tag @envoyproxy/maintainers

if you aren't sure if a given change merits runtime guarding.

The canonical way to runtime guard a feature is

```

if (Runtime::runtimeFeatureEnabled("envoy.reloadable\_features.my\_feature\_name")) {

[new code path]

} else {

[old\_code\_path]

}

```

Runtime guarded features named with the "envoy.reloadable\_features." prefix must be safe to flip

true or false on running Envoy instances. In some situations it may make more sense to

latch the value in a member variable on class creation, for example:

```

bool use\_new\_code\_path\_ =

Runtime::runtimeFeatureEnabled("envoy.reloadable\_features.my\_feature\_name")

```

This should only be done if the lifetime of the object in question is relatively short compared to

the lifetime of most Envoy instances, i.e. latching state on creation of the

Http::ConnectionManagerImpl or all Network::ConnectionImpl classes, to ensure that the new behavior

will be exercised as the runtime value is flipped, and that the old behavior will trail off over

time.

Runtime guarded features may either set true (running the new code by default) in the initial PR,

after a testing interval, or during the next release cycle, at the PR author's and reviewing

maintainer's discretion. Generally all runtime guarded features will be set true when a

release is cut. Old code paths for refactors can be cleaned up after a release and there has been

some production run time. Old code for behavioral changes will be deprecated after six months.

Runtime features are set true by default by inclusion in

[source/common/runtime/runtime\_features.cc](https://github.com/envoyproxy/envoy/blob/master/source/common/runtime/runtime\_features.cc)

There are four suggested options for testing new runtime features:

1. Create a per-test Runtime::LoaderSingleton as done in [DeprecatedFieldsTest.IndividualFieldDisallowedWithRuntimeOverride](https://github.com/envoyproxy/envoy/blob/master/test/common/protobuf/utility\_test.cc)

2. Create a [parameterized test](https://github.com/google/googletest/blob/master/googletest/docs/advanced.md#how-to-write-value-parameterized-tests)

where the set up of the test sets the new runtime value explicitly to

GetParam() as outlined in (1).

3. Set up integration tests with custom runtime defaults as documented in the

[integration test README](https://github.com/envoyproxy/envoy/blob/master/test/integration/README.md)

4. Run a given unit test with the new runtime value explicitly set true or false as done

for [runtime\_flag\_override\_test](https://github.com/envoyproxy/envoy/blob/master/test/common/runtime/BUILD)

Runtime code is held to the same standard as regular Envoy code, so both the old

path and the new should have 100% coverage both with the feature defaulting true

and false.

# PR review policy for maintainers

\* Typically we try to turn around reviews within one business day.

\* See [OWNERS.md](OWNERS.md) for the current list of maintainers.

\* It is generally expected that a senior maintainer should review every PR.

\* It is also generally expected that a "domain expert" for the code the PR touches should review the

PR. This person does not necessarily need to have commit access.

\* The previous two points generally mean that every PR should have two approvals. (Exceptions can

be made by the senior maintainers).

\* The above rules may be waived for PRs which only update docs or comments, or trivial changes to

tests and tools (where trivial is decided by the maintainer in question).

\* In general, we should also attempt to make sure that at least one of the approvals is \*from an

organization different from the PR author.\* E.g., if Lyft authors a PR, at least one approver

should be from an organization other than Lyft. This helps us make sure that we aren't putting

organization specific shortcuts into the code.

\* If there is a question on who should review a PR please discuss in Slack.

\* Anyone is welcome to review any PR that they want, whether they are a maintainer or not.

\* Please make sure that the PR title, commit message, and description are updated if the PR changes

significantly during review.

\* Please \*\*clean up the title and body\*\* before merging. By default, GitHub fills the squash merge

title with the original title, and the commit body with every individual commit from the PR.

The maintainer doing the merge should make sure the title follows the guidelines above and should

overwrite the body with the original commit message from the PR (cleaning it up if necessary)

while preserving the PR author's final DCO sign-off.

\* If a PR includes a deprecation/breaking change, notification should be sent to the

[envoy-announce](https://groups.google.com/forum/#!forum/envoy-announce) email list.

# Adding new extensions

For developers adding a new extension, one can take an existing extension as the starting point.

Extension configuration should be located in a directory structure like

`api/envoy/extensions/area/plugin/`, for example `api/envoy/extensions/access\_loggers/file/`

The code for the extension should be located under the equivalent

`source/extensions/area/plugin`, and include an \*envoy\_cc\_extension\* with the

configuration and tagged with the appropriate security posture, and an

\*envoy\_cc\_library\* with the code. More details on how to add a new extension

API can be found [here](api/STYLE.md#adding-an-extension-configuration-to-the-api):

Other changes will likely include

\* Editing [source/extensions/extensions\_build\_config.bzl](source/extensions/extensions\_build\_config.bzl) to include the new extensions

\* Editing [docs/root/api-v3/config/config.rst](docs/root/api-v3/config/config.rst) to add area/area

\* Adding `docs/root/api-v3/config/area/area.rst` to add a table of contents for the API docs

\* Adding `source/extensions/area/well\_known\_names.h` for registered plugins

# DCO: Sign your work

Envoy ships commit hooks that allow you to auto-generate the DCO signoff line if

it doesn't exist when you run `git commit`. Simply navigate to the Envoy project

root and run:

```bash

./support/bootstrap

```

From here, simply commit as normal, and you will see the signoff at the bottom

of each commit.

The sign-off is a simple line at the end of the explanation for the

patch, which certifies that you wrote it or otherwise have the right to

pass it on as an open-source patch. The rules are pretty simple: if you

can certify the below (from

[developercertificate.org](https://developercertificate.org/)):

```

Developer Certificate of Origin

Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.

660 York Street, Suite 102,

San Francisco, CA 94110 USA

Everyone is permitted to copy and distribute verbatim copies of this

license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I

have the right to submit it under the open source license

indicated in the file; or

(b) The contribution is based upon previous work that, to the best

of my knowledge, is covered under an appropriate open source

license and I have the right under that license to submit that

work with modifications, whether created in whole or in part

by me, under the same open source license (unless I am

permitted to submit under a different license), as indicated

in the file; or

(c) The contribution was provided directly to me by some other

person who certified (a), (b) or (c) and I have not modified

it.

(d) I understand and agree that this project and the contribution

are public and that a record of the contribution (including all

personal information I submit with it, including my sign-off) is

maintained indefinitely and may be redistributed consistent with

this project or the open source license(s) involved.

```

then you just add a line to every git commit message:

Signed-off-by: Joe Smith <joe@gmail.com>

using your real name (sorry, no pseudonyms or anonymous contributions.)

You can add the sign off when creating the git commit via `git commit -s`.

If you want this to be automatic you can set up some aliases:

```bash

git config --add alias.amend "commit -s --amend"

git config --add alias.c "commit -s"

```

## Fixing DCO

If your PR fails the DCO check, it's necessary to fix the entire commit history in the PR. Best

practice is to [squash](https://gitready.com/advanced/2009/02/10/squashing-commits-with-rebase.html)

the commit history to a single commit, append the DCO sign-off as described above, and [force

push](https://git-scm.com/docs/git-push#git-push---force). For example, if you have 2 commits in

your history:

```bash

git rebase -i HEAD^^

(interactive squash + DCO append)

git push origin -f

```

Note, that in general rewriting history in this way is a hindrance to the review process and this

should only be done to correct a DCO mistake.

## Triggering CI re-run without making changes

To rerun failed tasks in CI, add a comment with the the line

```

/retest

```

in it. This should rebuild only the failed tasks.

Sometimes tasks will be stuck in CI and won't be marked as failed, which means

the above command won't work. Should this happen, pushing an empty commit should

re-run all the CI tasks. Consider adding an alias into your `.gitconfig` file:

```

[alias]

kick-ci = !"git commit -s --allow-empty -m 'Kick CI' && git push"

```

Once you add this alias you can issue the command `git kick-ci` and the PR

will be sent back for a retest.