The Crunchy Container Suite is an open source project primarily authored and

maintained by the team at Crunchy Data. All contributions are welcome: the

Crunchy Container Suite uses the Apache 2.0 license and does not require any

contributor agreement to submit patches.

Our contributors try to follow good software development practices to help

ensure that the code that we ship to our users is stable. If you wish to

contribute to the Crunchy Container Suite, be it code or documentation, please

follow the guidelines below.

Thanks! We look forward to your contribution.

# General Contributing Guidelines

All ongoing development for an upcoming release gets committed to the

\*\*`master`\*\* branch. The `master` branch technically serves as the "development"

branch as well, but all code that is committed to the `master` branch should be

considered \_stable\_, even if it is part of an ongoing release cycle.

All fixes for a supported release should be committed to the supported release

branch. For example, the 4.3 release is maintained on the `REL\_4\_3` branch.

Please see the section on \_Supported Releases\_ for more information.

Ensure any changes are clear and well-documented. When we say "well-documented":

- If the changes include code, ensure all additional code has corresponding

documentation in and around it. This includes documenting the definition of

functions, statements in code, sections.

- The most helpful code comments explain why, establish context, or efficiently

summarize how. Avoid simply repeating details from declarations,. When in doubt,

favor overexplaining to underexplaining.

- Code comments should be consistent with their language conventions. For

example, please use GoDoc conventions for Go source code.

- Any new features must have corresponding user documentation. Any removed

features must have their user documentation removed from the documents.

- Do not submit commented-out code. If the code does not need to be used

anymore, please remove it.

- While `TODO` comments are frowned upon, every now and then it is ok to put a

`TODO` to note that a particular section of code needs to be worked on in the

future. However, it is also known that "TODOs" often do not get worked on, and

as such, it is more likely you will be asked to complete the TODO at the time

you submit it.

- Write clear, descriptive commit messages. A guide for this is featured later

on in the documentation.

Please provide unit tests with your code if possible. If you are unable to

provide a unit test, please provide an explanation as to why in your pull

request, including a description of the steps used to manually verify the

changes.

Ensure your commits are atomic. Each commit tells a story about what changes

are being made. This makes it easier to identify when a bug is introduced into

the codebase, and as such makes it easier to fix.

All commits must either be rebased in atomic order or squashed (if the squashed

commit is considered atomic). Merge commits are not accepted. All conflicts must

be resolved prior to pushing changes.

\*\*All pull requests should be made from the `master` branch\*\* unless it is a fix

for a specific supported release.

Once a major or minor release is made, no new features are added into the

release branch, only bug fixes. Any new features are added to the `master`

branch until the time that said new features are released.

# Commit Messages

Commit messages should be as descriptive and should follow the general format:

```

A one-sentence summary of what the commit is.

Further details of the commit messages go in here. Try to be as descriptive of

possible as to what the changes are. Good things to include:

- What the changes is.

- Why the change was made.

- What to expect now that the change is in place.

- Any advice that can be helpful if someone needs to review this commit and

understand.

```

If you wish to tag a Github issue or another project management tracker, please

do so at the bottom of the commit message, and make it clearly labeled like so:

```

Issue: #123

Issue: [ch1234]

```

# Submitting Pull Requests

All work should be made in your own repository fork. When you believe your work

is ready to be committed, please follow the guidance below for creating a pull

request.

## Upcoming Releases / Features

Ongoing work for new features should occur in branches off of the `master`

branch. It is suggested, but not required, that the branch name should reflect

that this is for an upcoming release, i.e. `upstream/branch-name` where the

`branch-name` is something descriptive for what you're working on.

## Supported Releases / Fixes

While not required, it is recommended to make your branch name along the lines

of: `REL\_X\_Y/branch-name` where the `branch-name` is something descriptive

for what you're working on.

# Releases & Versioning

Overall, release tags attempt to follow the

[semantic versioning](https://semver.org) scheme. The Crunchy Container Suite is

closely related to the

[PostgreSQL Operator](https://github.com/CrunchyData/postgres-operator) and in

some cases may not fully follow semantic versioning if a change in the

PostgreSQL Operator project necessitates a major or minor version change.

"Supported releases" (described in the next section) occur on "minor" release

branches (e.g. the `x.y` portion of the `x.y.z`).

One or more "patch" releases can occur after a minor release. A patch release is

used to fix bugs and other issues that may be found after a supported release.

Fixes found on the `master` branch can be backported to a support release

branch. Any fixes for a supported release must have a pull request off of the

supported release branch, which is detailed below.

## Supported Releases

When a "minor" release is made, the release is stamped using the `vx.y.0` format

as denoted above, and a branch is created with the name `REL\_X\_Y`. Once a

minor release occurs, no new features are added to the `REL\_X\_Y` branch.

However, bug fixes can (and if found, should) be added to this branch.

To contribute a bug fix to a supported release, please make a pull request off

of the supported release branch. For instance, if you find a bug in the 4.3

release, then you would make a pull request off of the `REL\_4\_3` branch.

## Unsupported Releases

When a release is no longer supported, the branch will be renamed following the

pattern `REL\_X\_Y\_FINAL` with the key suffix being \_FINAL\_. For example,

`REL\_3\_2\_FINAL` indicates that the 3.2 release is no longer supported.

Nothing should ever be pushed to a `REL\_X\_Y\_FINAL` branch once `FINAL` is on

the branch name.

## Alpha, Beta, Release Candidate Releases

At any point in the release cycle for a new release, there could exist one or

more alpha, beta, or release candidate (RC) release. Alpha, beta, and release

candidates \*\*should not be used in production environments\*\*.

Alpha is the early stage of a release cycle and is typically made to test the

mechanics of an upcoming release. These should be considered relatively

unstable. The format for an alpha release tag is `v4.3.0-alpha.1`, which in this

case indicates it is the first alpha release for 4.2.

Beta occurs during the later stage of a release cycle. At this point, the

release should be considered feature complete and the beta is used to

distribute, test, and collect feedback on the upcoming release. The betas should

be considered unstable, but as mentioned feature complete. The format for an

beta release tag is `v4.3.0-beta.1`, which in this case indicates it is the

first beta release for 4.2.

Release candidates (RCs) occur just before a release. A release candidate should

be considered stable, and is typically used for a final round of bug checking

and testing. Multiple release candidates can occur in the event of serious bugs.

The format for a release candidate tag is `v4.3.0-rc.1`, which in this

case indicates it is the first release candidate for 4.2.

\*\*After a major or minor release, no alpha, beta, or release candidate releases

are supported\*\*. In fact, any newer release of an alpha, beta, or RC immediately

deprecates any older alpha, beta or RC. (Naturally, a beta deprecates an alpha,

and a RC deprecates a beta).

If you are testing on an older alpha, beta or RC, bug reports will not be

accepted. Please ensure you are testing on the latest version.

# Testing

We greatly appreciate any and all testing for the project. When testing, please

be sure you do the following:

- If testing against a release, ensure your tests are performed against the

latest minor version (the last number in the release denotes the minor version,

e.g. the "3" in the 4.3.3)

- If testing against a pre-release (alpha, beta, RC), ensure your tests are

performed against latest version

- If testing against a development (`master`) or release (`REL\_X\_Y`) branch,

ensure your tests are performed against the latest commit

Please do not test against unsupported versions (e.g. any release that is marked

final).

There are several ways to help with the testing effort:

- Manual testing: testing particular features with a series of manual commands

or custom scripts

- Writing unit tests: testing specific sections of the code

- Writing integration tests: automatically testing scenarios that require a

defined series of steps, such as end-to-end tests

- Environmental & workload testing: testing the code against specific workloads,

deployment platforms, deployment models, etc.