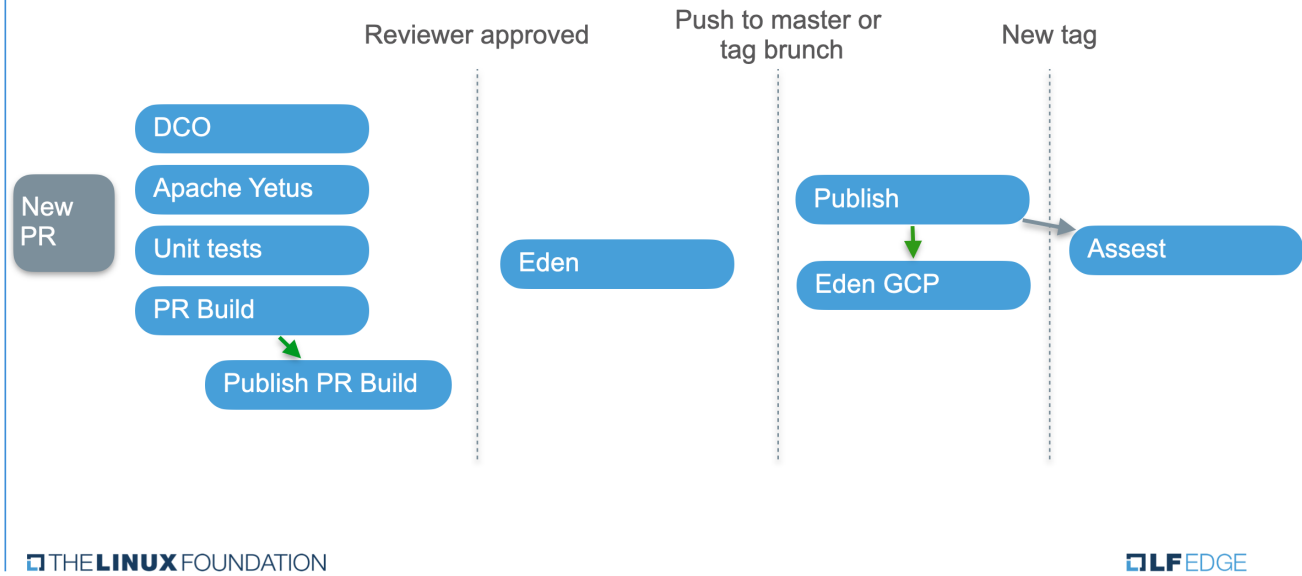


CI/CD infrastructure

Github Actions/ EVE-OS



EVE development happens on GitHub and we use GitHub's Actions as the core of our CI/CD infrastructure. Within GitHub actions, we try to do as much as possible with the stock Github ubuntu image (which as of December 2020 is `ubuntu-20.04`) VM that gets instantiated from scratch every time jobs need to run as part of a given action. However, since we also have to produce ARM builds, we have a hosted runner tagged `ARM64` that we use for a subset of the actions. EVE uses a standard OCI Registry (currently on [Docker HUB](#)) to store all of its CI/CD build artifacts.

Definitions of EVE CI/CD actions are available under [.github/workflows](#) folder and they currently include:

- **Publish** publishes various EVE build artifacts into an OCI Registry
- **Pulls** manages to pull requests
- **Apache Yetus** runs static, lint-like **Apache Yetus** checks on EVE
- **Eden** runs integration tests from EVE's sister project **Eden**

Since EVE community generally frowns on branch-heavy development most of the descriptions below will only apply to the master branch (unless stated otherwise).

Publish

This is the only action that uses both x86 and ARM64 runners. It gets triggered on either updates to the `master` branch or creation of release tags. Publish produces a set of tagged artifacts and pushes them into EVE's OCI Registry repositories. Logic responsible for figuring out correct tags for these artifacts resides in EVE's top-level Makefile and can be summarized as:

- If Publish gets triggered on a release tag (release tags are the ones with `x.y.z` names) that compares as greater than all the other release tags currently present in a repo that release tag is considered to be the latest. All artifacts originating from the Publish triggered on the latest tag get matching `x.y.z` OCI registry tags AND get an additional symbolic tag `latest`. Since the artifacts are typically retrieved with a `docker pull lfedge/eve-XXX` this policy corresponds well with the expectation that such a pull is expected to retrieve `lfedge/eve-XXX:latest` even though it is a mere alias for `lfedge/eve-XXX:x.y.z`. Another crucial distinction of a Publish action triggered on the latest tag is that it gets the benefit of a full (from scratch) build as opposed to using pre-cached artifacts. This is done to ensure that EVE builds can be rebuilt in the future in a self-contained manner but it does, of course, increase the build time dramatically.
- If Publish gets triggered on a release tag that is NOT considered to be the latest (for example when building `.z` patch releases for older `x.y.(z-1)` releases) the OCI registry artifacts only get a matching `x.y.z` tag and no additional aliases are generated. The build is also typically much faster since it uses pre-cached artifacts in OCI registry if the corresponding source package in the EVE tree didn't change. Note, however, that building a patch release for the previous latest tag becomes a new latest tag and thus subject to the latest tag treatment (e.g. `x.y.z` being latest means `x.y.(z+1)` will now compare as greater than any other tag since it is greater than `x.y.z` itself).
- If Publish gets triggered on a push to the `master` branch (e.g. after PR merge) all EVE packages in an OCI registry get two tags assigned:
 1. symbolic tag `snapshot` (to facilitate `docker pull lfedge/eve-pillar:snapshot` type of usage)
 2. source-derived unique tag Source-derived unique tag is expected to be a SHA-like ID that changes only when any part of the source tree corresponding to the EVE packages changes and thus allows for lazy (CAS-like) caching of build updates. For all packages with actual

source code this is linuxkit generated git tree sha (e.g. lfedge/eve-pillar:b8c0b1813a7819ee9eefdc77f1883da5613e7017).

For rootfs and its package this is a synthetic version ID generated by the makefile of the form 0.0.0-master-<git sha>.

- If Publish gets triggered on any other branch (although this is rare and is not supported by default in our Publish action) the Makefile logic will be identical to what's happening with a master branch with the only two exceptions being around the name of the symbolic tag. Instead of a snapshot it will be either:
 1. x.y.z-<# commits on top>-<git sha> (e.g. 5.15.1-6-g6658d671f) for branches that can be traced back to a git tag
 2. branch name (e.g. foo) for branches that do NOT contain a single git tag in their history

Pulls

Pulls is, effectively, a much more constrained version of the Publish action. The reason it needs to be constrained is that it triggers on any pull request submitted to the master branch and that may include anonymous pull requests containing malicious code in them. Builds triggered by this action do NOT deposit artifacts into the regular OCI registry. The registry that is being used is a quarantined one over at evebuild/danger and the only artifact that is ever being deposited there is a rootfs container tagged as pr-ID (e.g. docker pull evebuild/danger:pr-12345 will fetch a rootfs container for Pull Request 12345).

For the same reasons, Pulls action only runs on x86 runner since even GitHub itself strongly discourages running anything that gets triggered by pull requests on hosted runners (which is what our ARM64 runner is). This, in turn, may result in pull requests that end up breaking our ARM builds but that seems to be rare enough of an event for now.

Apache Yetus and Eden

These two actions run on x86 runner only. Yetus doesn't need to have an ARM64 run and Eden would've benefited from one, but the underlying infrastructure required for that is still too heavy to be hooked up to the GitHub actions (and it can't even be a hosted runner because of the security concerns).

Both of these actions get triggered on PRs submitted to the master (and no other!) branch and in addition Eden also gets triggered on updates to the master branch itself. We do NOT run either of these on tags (releases).

In order to avoid lengthy rebuilds of EVE's rootfs, Eden action checks for evebuild/danger:pr-12345 artifacts and if one is available it uses it. If not it rebuilds it from scratch and then uses the local build.

It may be marginally useful to run Yetus on updates to the master itself, but Yetus in its full glory will generate too much feedback and the incremental updates are already covered by Yetus triggering on pull request submissions.

Maintaining Dockerfile Versions

The Dockerfiles get out of date because they use base OS images that are periodically updated. It is important to stay up-to-date in general for bug fixes and security updates, but sometimes the Dockerfiles must be updated immediately when the OS image removes a version of a package.

Dockerfile Version Updates with Dependabot

GitHub provides a feature called **Dependabot** that fixes several kinds of out of date versions including the OS images in Dockerfiles. The limitation is that it only updates the base OS image, not the package versions in the apk add command, so it can only be used in EVE packages that do not hardwire package versions.

Dependabot should not be enabled in the reference lf-edge tree. Instead use it in user forks and let Dependabot create its PRs there. Then the user can cherry-pick the version changes, test, tweak, squash, and submit their own PR to lf-edge.

To use Dependabot in a user fork:

- enable Dependabot options in: Settings -> Security & analysis
- open: Insights -> Dependency graph -> Dependabot
- re-run by: modify this file, push to master
- review generated PRs
- cherry-pick from dependabot/docker/* branches

ver-update to List Package Versions

Dependabot will not help in EVE packages that use hardwired apk package versions. Updating them is a manual process. The hard part is finding the out-of-date versions and knowing what to replace them with. That part can be done automatically with the scripts in build-tools/ver-update. The main steps are:

```
cd build-tools/ver-update
make clean
make
grep pkg/my-pkg out-alp-ver-diff
```

It can generate the versions to change using different base OS versions. That version is specified by ALP_BASE_OS in Makefile.

Limitation

Job execution time - Each job in a workflow can run for up to 6 hours of execution time. If a job reaches this limit, the job is terminated and fails to complete.

Workflow run time - Each workflow run is limited to 72 hours. If a workflow run reaches this limit, the workflow run is cancelled.

API requests - You can execute up to 1000 API requests in an hour across all actions within a repository. If exceeded, additional API calls will fail, which might cause jobs to fail.

Concurrent jobs - The number of concurrent jobs you can run in your account depends on your GitHub plan, as indicated in the following table. If exceeded, any additional jobs are queued.

Architectures

The following processor architectures are supported for the self-hosted runner application.

x64 - Linux, macOS, Windows.

ARM64 - Linux only.

ARM32 - Linux only.

Release cycle - 2 weeks

Semantic versioning uses a structure like **<MAJOR>.<MINOR>.<PATCH>**

For a new release (i.e., a published version), we increment...

MAJOR when making incompatible API changes,

MINOR when adding backwards-compatible functionality,

PATCH when making backwards-compatible bug fixes.

FAQ

1) If we create a new workflow on master (say, new test), should we add this workflow onto our releases branch or GH do it automatically. Seems, it may also depend on the triggers of such workflow (has it branch-based filter or not).

The order of operations in a workflow run triggered by push or pull request is described in the reference documentation:

The `.github/workflows` directory in your repository is searched for workflow files at the associated commit SHA or Git ref. The workflow files must be present in that commit SHA or Git ref to be considered.

For example, if the event occurred on a particular repository branch, then the workflow files must be present in the repository on that branch.

2) If we change our workflow on master (change steps inside workflow), will it be updated on another branch we have (say, we add a step to this one). Seems, it may also depend on the triggers of such workflow (has it branch-based filter or not).

We must push changes to the brunch as well.

workflow:

2.1. Create an empty YAML file in the `.github/workflows` folder or change current

2.2. Create a PR to move that file to your branch

2.3 In your branch, you can now do the necessary edits to get your GH Action up & running. NOTE: next to updating your YAML, you also need to make a change that actually triggers the workflow