# Business-level Requirements for TF Release Process

## The entirety of the commit review test integration build release process

### Introduction

This document provides a clearing house to provide high level guidance to the TF TSC for delivering more detailed requirements for the entire release process, from code commit, to review, through testing, and final build and release.

### Questions to Answer / Outcomes to Accomplish (please add)

- What is the release cadence?
  - JG I think anything faster than 6 months is not realistic, maybe we should start with 9 months.
  - DAH How will we communicate features / APIs that get deprecated?
  - WS - If releases are on a longer cycle than 6 months, in my experience, it is hard to keep the community engaged. I think initially it could be longer as things like CI and such are setup, but I feel it should not be longer than 6 months in general.
- Are there multiple types of release? e.g. experimental, daily, stable, production?
  - JG Let's start with having CI in place and just one release (doesn't need to be called anything). Small steps to start with, we can go faster if we feel comfortable later.
  - WS - I agree that there should only be one type of release for now. CI is the most important piece here and will be the enabling factor for being able to manage more than one type of release.
- Will we have LTS releases or is that a problem for vendors making distributions?
  - JG No need for LTS releases to start with IMHO
  - WS - Once the release details and cadence have been ironed out and are working well, then an LTS release would be nice, but not to start with.
- Is there variance in the release cadence for the different types of release?
- How are these releases tested differently? For example:
  - Experimentals get basic sanity tests that are hours in length
  - Weekly stable releases get multi-day sanity/burn in tests
  - Production releases get month long scalability, performance tests, and synthetic transaction testing with VNFs or other workloads
  - Edward Ting For community releases, maybe it will be ok to have 1 ~ 2 weeks of soak test instead of month long.
    - WS - I can not stress enough how important CI is. I have served as a release manager for a large open source project and I can confirm that releases live and die by the quality of the CI. I would recommend that a general set of tests be run against every pull request, a set of tests that takes less than 2-4 hours. Then if the community maintains a much broader set of tests which can take >12 hours to run, that only ever gets run against master and is on a schedule, like running once a week. The code should be constantly tested, not just around release period. If a developer introduces an issue, that has to be flagged as soon as possible because it is likely that developer will be forced to move onto different tasks soon after submitting their work, so if you want an engaged community, the feedback loop for developers has to be relatively quick.
- How do we determine what goes in a release?
  - JG From the blueprints
  - Edward Ting we need to able to prioritize the blueprints going to the releases within the 1st month of the release cycle.
  - Edward Ting suggest to have close the gate if a blueprint is not done or stable by a certain time before the release date, says 1 month.
  - WS - I agree that the blueprints are a good source for setting an expectation around a release roadmap. I also agree that there should be a freeze period, like 1 month prior to a release, where no new features can be introduced, only bug fixes and stabilizing changes.
- Are releases tagged? Are features tagged? Are tagged features included in a release?
  - JG Absolutely releases should be tagged. Big features can have their branches and then those branches get merged back.
  - WS - I agree. Releases should be tagged for sure. I wouldn't tag features, but I do think features should be developed in their own branches. I am not sure if the feature branches are expected to be managed on the developer forks only or if they are expected to be present in the upstream repo as well. If feature branches are managed in the upstream repo, they need to be diligently pruned when features are merged. If they are not automatically removed, the upstream branches can get chaotic, especially if features are not merged into master in a timely fashion.
- Or do we operate in branches?
  - WS - Development of new features should be done on their own branches IMHO. This is the standard development workflow which most developers are comfortable with, so I wouldn't recommend we try to change that.
- What infrastructure is required to support the release plan?
  - Where is it hosted?
  - Who is contributing?
  - JG These are important questions that need to be answered. We need CI/CD in place. Without it it is nearly impossible to have stable releases. We need to also have performance testing as part of CI. We do need a contributors list too. Both developers and devops that run the whole CI engine.
  - WS - I agree with JG. I would argue that CI/CD is absolutely critical for a stable open source project. Additionally, since TF can be deployed in different configurations (OpenStack or Kubernetes for example), it is very important that there is CI/test coverage for each of the different deployment patterns which are expected to be stable. I agree that in the context of TF, performance testing is also important and can not be overlooked.
- What are the quality criteria for release?
  - Edward Ting I have proposed to start with 3 use cases, CentOS, Kubernetes, and Ubuntu, using ansible-deployer, setting up 1 control node (K8S master) and 2 compute nodes (K8S slaves). Plus one DPDK performance. (TF OpenLab already has such setup.) These are the absolute minimum baseline. Community members can contribute different use cases.
  - Edward Ting At the 2nd phase, we should add HA scenarios repeating the non-HA 3 use cases above. Plus we should increase # of compute nodes to something meaningful, says, 10, 20, 30,...
- What are the security criteria for release?
  - Edward Ting I have authored security blueprint. Suggest to review that blueprint as a reference.
- What are the documentation criteria for release?

    - Edward Ting Release documentation should be automatically generated. Features are the blueprints and fixed issues can be retrieved from gerrit, same as the known issues. There is no point to duplicate the efforts. The author of the blueprint should also provide the engineering documentation.
    - WS - I agree that the detailed release notes should be auto-generated based on what code has been merged into the repo since the last release. That said, I also think it is important to provide a marketing focused summary of the release which just touches the tips of the waves and can be distributed more generally. This would likely need to be manually curated, but would leverage and reference the details in the auto-generated release notes.
- What assets or artifacts need to be built, tested and released together to create a usable deployment?
    - WS - Currently there is a major challenge in the usability of TF because there is no tie between a specific build or release and the appropriate kubernetes manifests which enable those artifacts to be deployed. The builds are being published here and here, but there is no link between the tagged releases in those repos and an appropriate manifest for those artifacts. Currently, the manifests are managed here, but unfortunately, they are only specific to a point in time. The manifests evolve over time, but that evolution is not done in lock step with the images being built. So if I want to deploy build `master-588` (for example), there is no way to know what the appropriate manifests are to be able to successfully deploy that build of the code. Not only that, the manifests are too complicated for an end user. If I take this manifest, for example, it takes 30 different unique variables, *all of which are not documented anywhere*. I think others recognize that it is not realistic to expect an end user to know how to populate these variables, so this script was created to resolve the variables, which in turn calls this script which, through 'magic', tries to discover the values of the variables based on the local system. While this may work in some cases (definitely not all), it does not make the resulting manifests transferable to other setups. The documented `one line install` process tries to solve for this, but unfortunately, the manifests which are referenced in that documentation are not maintained in the current build/release cycle.
      **In summary**, there is currently NO WAY to consistently produce deployments targeting different builds/releases because the manifests are not tracked in lock step with the builds. Because of this, it is impossible to ensure the end user experience is consistent and reproducible, which I feel is a fundamental flaw that must be addressed if we expect TF to be able to be adopted by an end user community.
      **My proposal**, is that every build of the image artifacts ALSO builds the appropriate manifests for that build and publishes them to a publicly curlable location (github or an object store) based on the build or release name (`master-588` for example). In addition, I think there should be automatic variable substitution for all of the variables that make sense to be replaced. For example, since the manifest will be created for a specific build, which is being pushed to an upstream image repo (docker hub), variables such as `{{ CONTRAIL_REGISTRY }}` and `{{ CONTRAIL_CONTAINER_TAG }}` should be automatically filled. I am sure there are other variables which can be auto populated, but this illustrates the idea.
      Additionally, I would argue that the CI should be testing the combination of the built images with the generated manifests so if there is any regression in their functionality together, it is caught by CI (instead of our users). This will give us a solid foundation on which we can build forward maintainable documentation which will ensure the path we set the end user on through the documentation has been validated.

## Requirements

1. Releases **MUST** have a web/wiki page that explains exactly what is in the release
    a. The page **SHOULD** be automatically generated if at all possible
2. Releases **MUST** be in the form of docker images stored in DockerHub
    a. Releases **MUST** be architected as microservices and be Kubernetes compatible
3. Releases **MUST** have a regular cadence
    a. Ideally releases **WOULD** happen every month
    b. The minimum cadence velocity **MUST** be 3 months, not 6 months
4. Releases **MUST** have some form of labels, tags, or branching scheme that allows for mapping included features and bug fixes into a CHANGELOG and RELEASE NOTES
    a. See #1 above
    b. The scheme **MUST** be documented here in the Wiki
5. There **MUST** be a variety of release types for various purposes to be determined; a starting point might be:
    a. Experimental releases for nightly or high interval testing looking for build breakages
        i. Lightly tested builds
    b. Stable regular builds for doing downstream integration testing into commercial distributions
        i. Standard testing
    c. Production builds that can be used for official releases
        i. Extensive scalability and performance testing (as possible)
6. **SHOULD** develop a scheme for LTS builds based on production builds (above)
    a. QUESTION: what does an LTS build mean for an open source project that has no official support?