Release Engineering

Readiness Assessment Framework

Contents

Introduction

A release engineering is a new and rapidly growing specialist subject of software engineering where the standards for application development are derived and will be seriously followed in order to get the reliable product when release happens. A release engineer has a great knowledge about multiple disciplines like source code management, configuration management, development process and customer support. The release engineering is a flexible process which moulded bases on the requirements of the product.

Part-I Development Guidelines

This is the first part of the release engineering process definition which enlists all the guidelines that must be adhered to, by the developers.

Clause 1(a) - Branching Model

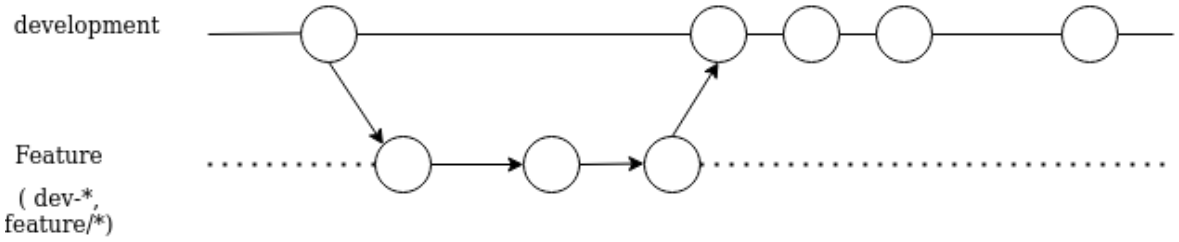
Git branches are reasonable and affordable to create and maintain in the long run. This clause states that a proper gitflow branching model must be followed during development. Respective guidelines for every branch is listed below:

Reference to the working of gitflow branching model.

1. **Master branch**

   Source code master branch represents the production ready state. Master contains most stable code and will be deployed to production. The master branch officially reserves the release history of any software application's source code. All the commits on the master branch must be tagged with a version number. Developers should not commit anything directly into the master branch. Without the approval of the QA team the code shouldn't be merged into the master branch.

2. **Develop branch**



Instead of a single master branch, our workflow will use two branches to record the history of the project. The develop branch serves as a branch that integrates various feature branches. Both developers and maintainers have the privilege to merge feature branches to develop branches. While merging branches into develop the developers should provide a little description about the features they worked on that branch.

3. **Feature branch:**

Feature branch is a child of develop branch i.e. instead of branching off of master, feature branches use develop as their parent branch. Using feature branches, the developers can work on multiple features at the same time. Once new features are developed and tested, the feature branch will merge into the develop branch.

   a. The code for each new feature for your application should reside in its own particular feature branch, which can be pushed to the central source code repository for backup and collaboration.

   b. All the feature branches should exist only on developers repos, not in origin.

   c. The feature branches never interact with or get merged into the master branch directly. When a feature is complete, it's respective branch always gets merged back into the develop branch.

4. **UAT branch:**

UAT is the pre-production branch also known as staging area, once code is developed and tested in the develop branch it gets merged into the uat branch.

a. The UAT branch should be locked so that developers couldn't commit directly to this branch. Only the maintainer should have permission to accept the merge/pull request in the UAT branch.
b. The UAT branch should be used for creating a release branch once all testing is done. Bug fixes in UAT shouldn't be directly merged into the uat branch, all the bug fixes branch should be merged into develop branch first.

4. **Release branch:**
   Release branch will be derived from the UAT branch. The creation of this branch is done to start the next release cycle, and it indicates that no new features can be added after this—only documentation, bug fixes, and other tasks related to the release should be done in this branch. Source code tagging and release versioning are also done here. Once the code is ready to be shipped, the release branch is merged into the master branch. It is tagged with a version number too. Additionally, it should also be merged back into the develop branch as well, which may have progressed since the release was initiated.

6. **Hotfix/Bugfix branch:**
   Hotfix branches also known as maintenance branches are used for quickly patching the production releases. The bugfix branch is derived from the master branch when there are some bugs in the production environment. As soon as the bug is fixed, the hotfix branch gets merged back into both the master and the develop branches along with the ongoing release branch, and then the master branch is tagged with an updated version number too.

Clause 1(b) - Branch Naming Conventions

This clause states that proper naming conventions must be followed for each branch mentioned below.

Branch names should only contain lowercase letters, hyphens, slash and numbers, regexp: /[a-z0-9\-]+/.

1. **Feature branch:**
   You can use slash or hyphen as a separator while naming a feature branch but make

sure the feature branch naming should be homogeneous, don't mix multiple feature branch naming conventions, before starting working on a project decide one naming convention and stick to that.

   a. feature/*
   b. feature/gitlab-integration
   c. feature/ticket_id
   d. feature-*
   e. feature-slack-integration

2. **Release branch:**

   a. release-*

   b. release-1.0.0

3. **Hotfix branch:**

   a. hotfix/*

   b. hotfix/broken-link

Clause 1(c) - Merge Process

Developer should follow proper Merge Process/Request and Release Branching Process

Before performing a merge there are a couple of preparation steps to take to ensure that the merge goes smoothly. Use a short description before merging the code to any branch.

I. Confirm the destination branch

II. Fetch latest remote commits

III. Apply Merge

Merge flow:

feature/branch_name → develop → UAT→ Release→ Master

Release branching refers to the concept of making a release reside within a particular branch. A branch gets created when the team starts working on the new release (e.g., "Release 2.1"), and all work done until the next release is stored in this branch.

Clause 1(d) - Git Commit Guidelines

This clause states the guidelines for Git Commit.

When you're working in a group project or contributing in an open source project the commit message is the best way to communicate the context about a change to other developers working on that project, and indeed, to your future self.

But most of the time developers don't focus on commit messages, they use weird commit messages that don't help anybody and also these messages can't convey what changes are done and why. To make commit messages significant a developer must follow the following guidelines.

Commit message should be short and descriptive, try not to exceed 50 chars (max 72 chars)

1. Commit message should be imperative
   - "Add Dockerfile"
   - "Added Dockerfile"
2. Commit messages should describe why the change is made and how it addresses the issue.
3. Capitalize your subject in the commit message.
   - "Fix github integration error"
   - "fix github integration error"
4. If commit refers to an issue/bug make sure to add issue/bug number in the commit message.
5. Use characters to make the commit message short.
   - "Add hindi & english translation"
   - "Add hindi and english translation"
6. Subject line should not end with a period/Full stop.
   - "Add new Dockerfile"
   - "add new Dockerfile."

Part-II CI/CD Guidelines

This is the second part of the release engineering process definition which enlists all the guidelines that must be adhered to, by the release engineers.

Reference to the required process flow diagram.

Clause 2(a) - CI/CD Stages

Every branch mentioned in the Part-I above should have a CI/CD pipeline that must strictly consist of the below-mentioned stages.

01. Feature/HotFix Branch
    a. Unit Testing
    b. Integration Testing
    c. Build
    d. Push
    e. Deploy code (if required)
02. Develop Branch
    a. Code Quality Check
    b. Unit Testing
    c. Integration Testing
    d. Functional Testing
    e. Build
    f. Push
    g. Deploy to Dev
03. UAT Branch
    a. SAST
    b. DAST

c. Browser Performance Testing

d. Load Testing

e. Fuzz Testing

f. Build

g. Push

h. Deploy to UAT

04. Release Branch: This branch should not contain any pipeline. This branch only contains code ready for the release. Only the documentation and final touches and minor bug fixes to the code are to be focused on.

05. Master Branch

a. Build

b. Push

c. Deploy to Prod

Definition of each stage:

1. Code Quality Check:
   This will help in analyzing the source code quality and ensuring that the project code stays simple, readable and easier to contribute to. Source code will be automatically analyzed to surface issues to check if the quality of the code is improving or getting worse with the latest commit.

2. Unit Testing:
   Unit testing is a stage where discrete functions are tested at the source code level. It is a type of testing where we test the individual units or components of a software application. Unit testing fulfills the purpose of validating the performance of each unit of the software code.

3. Integration Testing:
   The stage of integration testing is included to perform a level of software testing where

individual units/components are integrated and tested as a whole.It fulfills the purpose of exposing faults in the interaction between the integrated units.

4. Functional Testing:

   Functional testing is a quality assurance process and a type of black-box software testing that validates the software system against the functional requirements/specifications. The purpose of functional tests is to test each function of the software application, by providing appropriate input, verifying the output against the functional requirements, and internal program structure is rarely considered.

5. SAST:

   SAST stands for Static Application Security Testing. Its aim is to scan the application source code and binaries to perform static code analysis. This helps to spot potential vulnerabilities before the application deployment. Every merge request triggers the scanning, collects the results, and presents the list of vulnerabilities in a single report.

6. DAST:

   Dynamic Application Security Testing analyzes the running web application for known runtime vulnerabilities. It introduces enhanced security since it is a black box testing where communication with the web application is done through the front-end in order to identify potential security vulnerabilities in the web application and architectural weaknesses.

7. Browser Performance Testing:

   This is a web performance testing where we ensure the performance of the software in-browser using automated browser performance testing. This would measure the performance of a web page using an open source tool and create a report which should be uploaded as an artifact. This feature is introduced with the intention to provide the overall performance score of each web page.

8. Load Testing:

   This is a typically important part of the CI process where developers are able to make performance decisions earlier in the development process by measuring how software responds under load as and when changes are introduced. This can be categorised as performance testing.

9. Fuzz Testing:

   Fuzz testing, or fuzzing, measures the application's response and stability by providing unexpected inputs like some malformed or random data. This helps in monitoring the unexpected behavior or application crashes. Fuzz testing is important because it finds issues that traditional testing methods typically do not. It lets you discover software defects that should be addressed on priority as these defects may lead to highly exploitable vulnerabilities.

10. Build:

    A runnable instance of our product is built and shipped to the end users by combining the application's source code and its dependencies. Regardless of the language, cloud-native software is typically deployed with Docker, in which case this stage of the CI/CD pipeline packages the entire code along with its dependencies into images and builds the Docker containers.

11. Push:

    This is a stage where the built docker images are pushed to our secure and private harbor registry with relevant tags so that they can later be retrieved for deployment as per the requirement.

12. Deploy:

    This is the final stage of any pipeline where the application is deployed to an environment relevant to the branch i.e. dev, uat or prod.

Clause 2(b) - Practises to Design CI/CD Stage

With reference to the clause 2(a) mentioned above, it must be ensured that:

I. Build and push stages should be inculcated in the pipeline of every branch namely feature/hotfix, develop, uat and master except release.

II. Deployment stage in the pipeline of the feature/hotfix branch shall be inculcated only if required for testing purposes. Further, code present in the develop branch must be deployed to a dedicated dev environment. Similarly, code present in the uat branch must be deployed to the uat environment for manual testing by the QA team and final

deployment to the production environment for end users must be done within the pipeline of the master branch.

III. Release branches must not have any pipeline as these branches should only be dedicated to documentation and final touches to the release.

IV. As far as testing is concerned, following guideline should be followed:

   A. Pipeline in the Feature/Hotfix branch must have unit and integration testing only.

   B. Pipeline in the Develop branch must have unit, integration, and functional testing only.

   C. Pipeline in the UAT branch must be dedicated to all types of security and performance testings namely, SAST, DAST, Browser Performance Testing, Load Testing and Fuzz Testing.

   D. Pipeline in the master branch should not have any stage of testing.

Clause 2(c) - Defining Environments

This clause states that there should be isolated deployment environments for DEV, UAT and PROD.

I. Isolation of each environment must be strictly considered to avoid any point of conflict. Ideally, the safest and most secure way for isolation of different environments is to use a separate cluster for each environment. This option minimizes the risk of putting the production environment in danger due to potential human mistakes and machine failures. However, this increases the maintenance and administration overhead, reduced utilization and also the cost of more infrastructure management.

II. Alternatively, a single cluster with different namespaces can be used to isolate different environments. K8s community has already evolved and matured a lot that enables us to address the security issues through proper usage of network policies, node selectors, seccomp, access control, etc.

III. Still, to have a strong separation between production and non-production environment, we can have one cluster for DEV + UAT environments and one cluster completely dedicated for the final production environment.

As a result, this choice completely depends upon the need of the hour. The best option relevant to the use-case shall be implemented, considering all the probable pros and cons of each.

Clause 2(d): This clause states that proper naming conventions should be followed for accessing the applications deployed in different environments.

Suppose an application with the hostname ‘example.com’ is getting developed. So, it must be ensured that:

I. The hostname of the application deployed in DEV environment should be dev.example.com

II. The hostname of the application deployed in UAT environment should be uat.example.com

III. The hostname of the application deployed finally in the PROD environment should be example.com

Clause 2(e): This clause states that proper artifactory management must be given prime importance

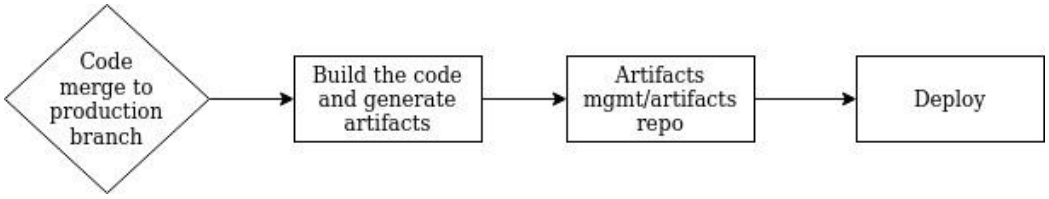
An artifact is referred to as the by-product of software development. It is considered as a deployable component of an application; for eg: docker images, jar files, wheel files, etc. Every release in the DevOps CI/CD processes is a collection of artifacts that must be preserved and managed properly.

I. Source code, meeting notes, workflow diagrams, data models, risk assessments, use cases, prototypes, and the compiled application can all be considered as artifacts. There must be a list drawn up during the planning stage that covers all of the required artifacts that must be produced.

II. Stable tagging of artifacts like docker images must be ensured for proper segregation and easy identification.

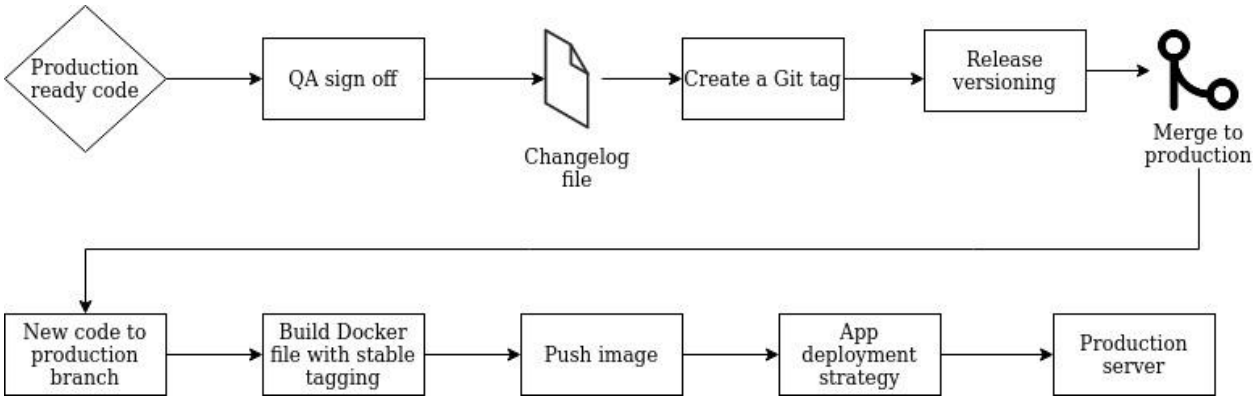
The tag might include the project name, repository name, branch name, pipeline ID, etc.
For eg: lifedata/stt-server:develop123456

III. An Artifactory repository must be used for storing and managing artifacts. It is an application designed to store, version, and deploy artifacts for builds. It is both a source for artifacts needed for a build, and a target to deploy artifacts generated in the build process.



Reference to the release plan including artifactory management.

Part-III Release Your Product to End Users



Clause 3(a) - Product Shipping to end Users

This clause states the process of product shipping to end users.

Once your code is ready to be shipped to the live environment the product team must follow the following clauses for smooth release of the product to the end users.

1. The developer should create Changelog files for each and every microservices of the product. The developers must stick to the given link for the Changelog file structure.
   Changelog file format: https://keepachangelog.com/en/1.0.0/

2. The production ready code should also contain README files.
3. The QA team should send the QA signoff to the developers and release engineers, without the QA sign off the code should never be merged into the production.
4. The product team should be ready with all the user documents and the reference documents.
5. The sales and marketing team should have a basic idea of the product so that they can talk about its features with the customers.
6. The support team should be properly trained to counter any user issues in the product.
7. The release engineers should create a checklist and cover all the things that are essential for the release.
8. Once everything in the checklist is checked off as completed, the developer must create a release branch for the release versioning and the tagging.
9. The developer should follow semantic versioning while creating a tag or a release. Semantic Versioning: https://semver.org
10. Once the release versioning is done the branch should be merged into the production branch by the developers.

Clause 3(b) - Deployment Strategies

This clause states the importance of appropriate application deployment strategy.

1. Recreate: Version A is terminated then version B is rolled out.
2. Ramped (also known as rolling-update or incremental): Version B is slowly rolled out, replacing version A.
3. Blue/Green: Version B is released alongside version A, then the traffic is switched to version B.
4. Canary: Version B is released to a subset of users, then proceeds to a full rollout.

5. A/B testing: Version B is released to a subset of users under specific conditions.
6. Shadow: Version B receives real-world traffic alongside version A and doesn't impact the response.

Clause 3(c) - Choosing the right deployment strategies

Recreate

This type of deployment strategy states that the old pods will be killed all at once and get replaced with the new ones. This deployment approach involves downtime that occurs, while the old versions are getting brought down and the new versions are starting up.

Situations/Considerations:

- Your application does not support multiple versions and can withstand a short amount of downtime.
- You have a ReadWriteOnce volume mounted to your Pod and you cannot share it with a Replica.
- You want to stop processing old data and need to run some prerequisites before you start up your new application.

Rolling Update

It's the default strategy in Kubernetes, in which new pods are slowly rolled out by replacing the old pods without causing any down time. A rolling update doesn't scale down the old pods unless it validates via the readiness probe and confirms that the new pods have become ready. If there is any problem, you can abort the rolling update or deployment without bringing the whole cluster down.

The rolling update configuration consists of:

- maxSurge: Maximum numbers of pods that can be created over a desired no of pods.
- maxUnavailable: no of pods that can be unavailable during the update process

Benefits/Considerations:

- Releasing the application/service to development/staging environments.
- Downtime in an application that processes a huge number of transactions per minute can cause lots of problems. In the Rolling update, the application just continues to operate with Zero Downtime.
- Feature of “track/record” deployments, useful in rollback.

Blue-Green

In this strategy the old version and the new version of the application are deployed at the same time. The users will be able to access only the old version and the new version is firstly available for the QA team for testing, once they’re done testing the traffic of the users is shifted to the new version.
It helps in testing a production-quality environment before it is made public. In contrast to the rolling and canary deployment approaches, it also enables them to switch all users over to a new release at once.

Benefits/Considerations:

- Quick rollbacks, and easy disaster recovery.
- Blue/green deployment is zero-downtime, so the team can make the switch and let the load balancing system automatically shift all users to the green version. The old, blue version of the application is ready and waiting in case something goes wrong and requires it to be rolled back.
- **Overhead:** running two identical environments is expensive.

Canary

In the Canary deployment strategy we shift a controlled percentage of user traffic to the new version of the application, we use this strategy mostly when we’re not confident about the stability of the new version.

A key advantage of this app deployment strategy over blue/green deployment strategy is the early access to bug identification and feedback. It finds weaknesses and improves the update before the IT team rolls it out to all users.

Benefits/Considerations:

- Great to try out new features and see how the application and system behaves when the traffic is routed to a new version.
- Canary deployments will only be useful to you if you can track their impact on your system.

A/B testing

A/B testing deployment strategy is all about routing a subset of users to a new functionality under specific conditions. It is usually a technique for making business decisions based on statistics, rather than a deployment strategy.

Benefits/Considerations:

- Gives full control over the traffic distribution and you can track the results in terms of customer behavior and revenue.
- Several versions run in parallel, It's easy to revert back to the older version in case the new version does not provide significant benefits.
- **Overhead:** Expensive Setup

Shadow

A shadow deployment consists of releasing version B alongside version A, fork version A's incoming requests and send them to version B as well without impacting production traffic. This is particularly useful to test production load on a new feature. A rollout of the application is triggered when stability and performance meet the requirements.

Benefits/Considerations:

- Frees you from setting up a dedicated load test environment, the load tests are based on actual traffic in your existing environment.

- Performance testing of the application can be done with production traffic for more accuracy.
- **Overhead:** Expensive and complexity of setup(duplicate transactions or requests).

Clause 3(d) - Rollback Process

Rollback: it is an operation which returns the object to some previous state

We do the Rollback to an earlier Deployment revision if the current state of the Deployment is not stable or not giving expected results due to the application code or the configuration.

There are scenarios where even the perfect deployment gets your application halted when deployment is integrated with a newer version, so at that time we have to get back to the version of the application which was running fine with the help of revisions available for the deployments.
Steps to follow while performing a rollback:

1. Firstly find the cause of the error because of which application is getting into problems. Can be reviewed through the logs and state of the deployment(its pods). Some error references: CrashLoopBack, ImagePullBackOff, etc.

2. If the problem cannot be resolved and there is a need to rollback, we follow the rollback process.
    a. Check the rollout history for the revisions available, if **--record** flag was used while creating the deployment, you can check the change-cause as well.

    ```
    kubectl rollout history deployment.v1.apps/<deployment-name>
    ```

    b. Pick the revision which you consider stable and working as expected, make its(revision) entry in the **--to-revision** flag

    ```
    kubectl rollout undo deployment.v1.apps/<deployment-name>
    --to-revision=<number-of-the-revision>
    ```

    c. You can describe the rolled back deployment and verify its configuration.

```
kubectl describe deployment nginx-deployment
```

Note: By default Kubernetes stores the last 10 ReplicaSets. But you can change the number of ReplicaSets to be retained by changing the spec.revisionHistoryLimit key in your Deployment file.

Conclusion

A release engineering is a fast growing profile of a software engineering and is a recurring process from development to release of reliable product. An organization must establish the release engineering process in order to deliver their product rapidly with high quality.