Development Best Practices:

An overview of working in the software world

Anil Belur
abelur@linux.com
@askb23

Agenda

THE LINUX FOUNDATION

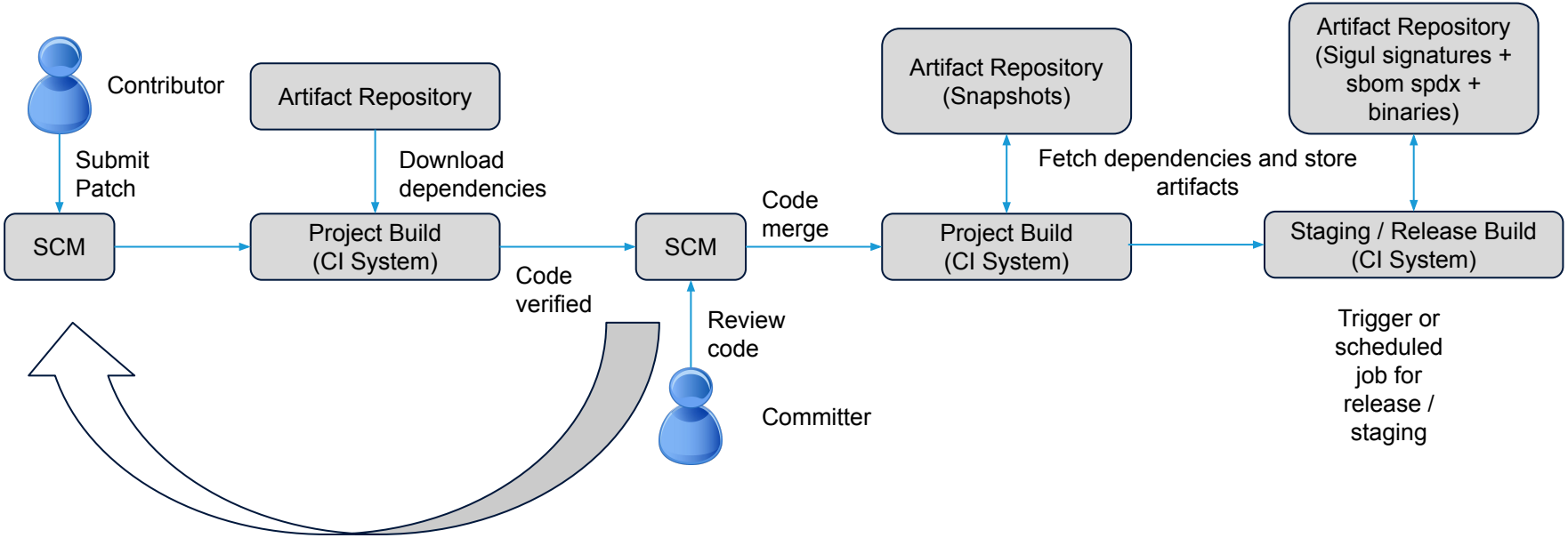
- Introduction
- The Linux Foundation (LF)
- What is Release Engineering?
- Typical Development Workflow
- Source Code Management (SCM) practices
- Git Workflows
- Performing relevant code review
- Validation testing
- Continuous Integration / Continuous Delivery (CI/CD)
- Resources

The Linux Foundation

THE LINUX FOUNDATION

“The Linux Foundation is creating the greatest shared technology investment in history by enabling open source collaboration across companies, developers and users”

Typical Code Development Workflow



What is Release Engineering?

THE LINUX FOUNDATION

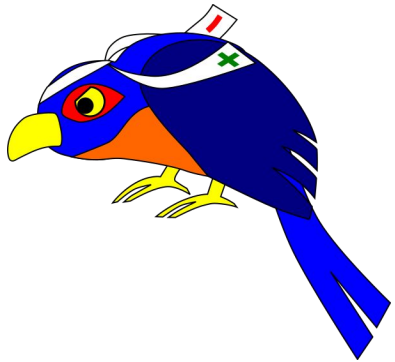
Release Engineering is the difference between manufacturing software in small teams or startups and manufacturing software in an industrial way that is repeatable, gives predictable results, and scales well. These industrial style practices not only contribute to the growth of a company but also are key factors in enabling growth.

-- Boris Debic

Source Code Management (SCM)

How, why, what, and when to commit

Source Code Management (SCM)



SCM: What is a commit?

THE LINUX FOUNDATION

- An object in an SCM that contains a message about a change and *usually* the change itself
- An empty commit is when it is just a message with no direct data change (repository tags and merge commits are forms of empty commits)
- Affect a branch of code. The default branch in git was historically named master but has in general shifted to be named main

SCM: What is a commit? (cont)

- Code changes are stored as differences (diffs) between changes thereby keeping the size of stored repositories low
- Binary objects may be stored in changes. This practices is strongly discouraged in general as doing so causes repository bloat as the entire object must be saved with each change.

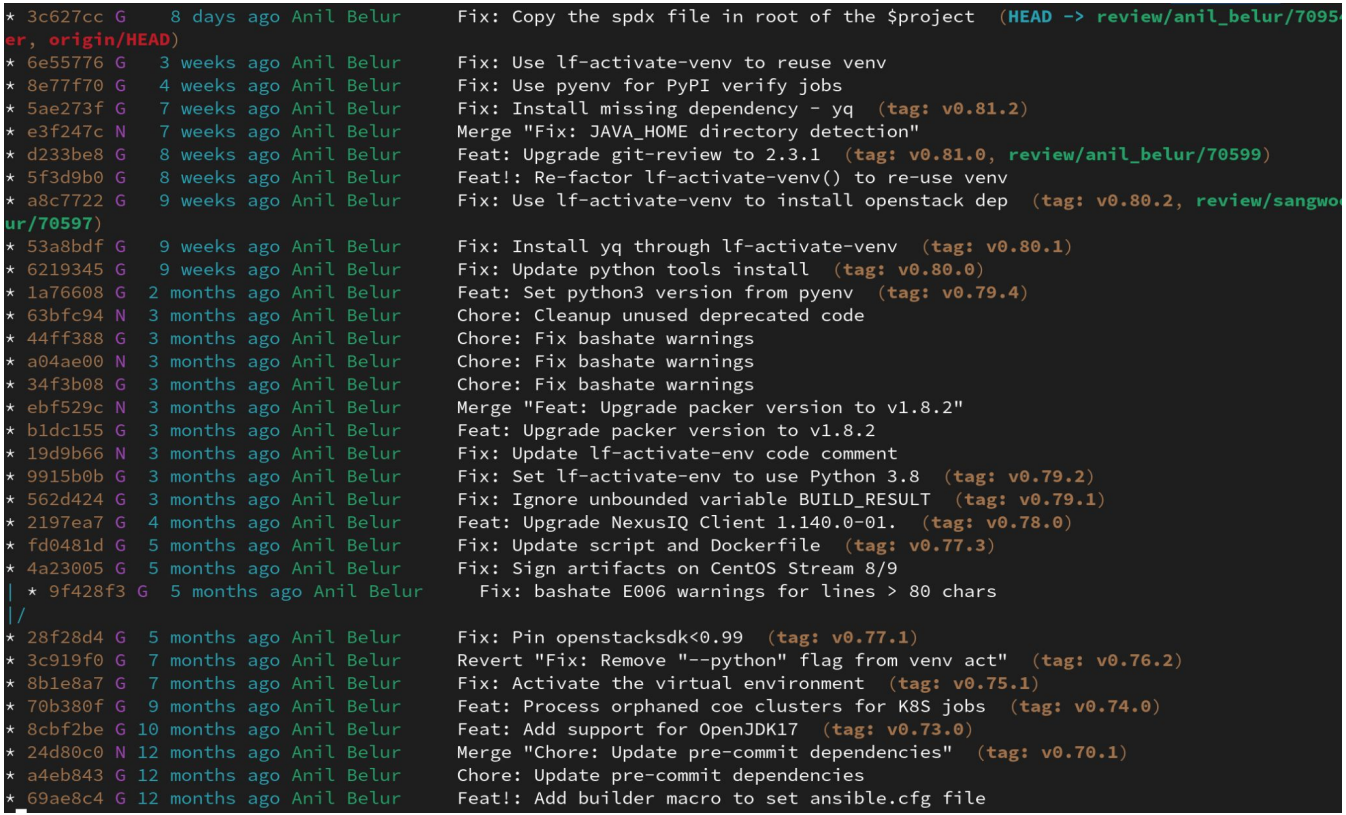
SCM: Telling your commitment story. A tale in two images

THE LINUX FOUNDATION

Not very useful message summaries

https://xkcd.com/1296/

SCM: Telling my commitment story (pt2)



SCM: Semantic (Conventional) commits

Easy set of rules for creating clean commit history and allows easier to write automated tools.

Structure of a commit message:

```
<type>[optional scope]: <description>

[optional body]
[optional footer(s)]
```

Structural elements:

1. fix: *type* fix patches a bug (correlates with **PATCH** in Semantic Versioning).
2. feat: *type* feat introduces a new feature (correlates with **MINOR** in Semantic Versioning).
3. BREAKING CHANGE: in the footer, or appends a "**!**" after the type/scope (correlates with **MAJOR** in Semantic Versioning). A BREAKING CHANGE can be part of commits of any *type*.
4. Other *types* build:, chore:, ci:, docs:, style:, refactor:, perf:, test:, and others.
5. *footers* other than BREAKING CHANGE: <description> may be provided and follow a convention similar to **git trailer format**.

Source: https://www.conventionalcommits.org/en/v1.0.0/

SCM: The seven rules of commit messages

THE LINUX FOUNDATION

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject
6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. *how*

Source: https://chris.beams.io/posts/git-commit/

SCM: Your commit story

- The body of a commit should give story that tells “why” the change is needed, not how it is being done
- Do not repeat information in subject lines in the body / footer as meta-data
- A commit with just a subject line may be valid if it conveys everything that is needed
  - Ex: “Chore: Update common-packer to the latest 0.11.1”

SCM: What to commit

THE LINUX FOUNDATION

- Commit in small atomic pieces
  - Commits should should be focused on one category of change at a time (Fix, Feat(ure), Chore, Revert, etc)
  - The commit should be the smallest amount of change needed to be properly testable
  - The change should be something that is easily reviewable

SCM: When to commit

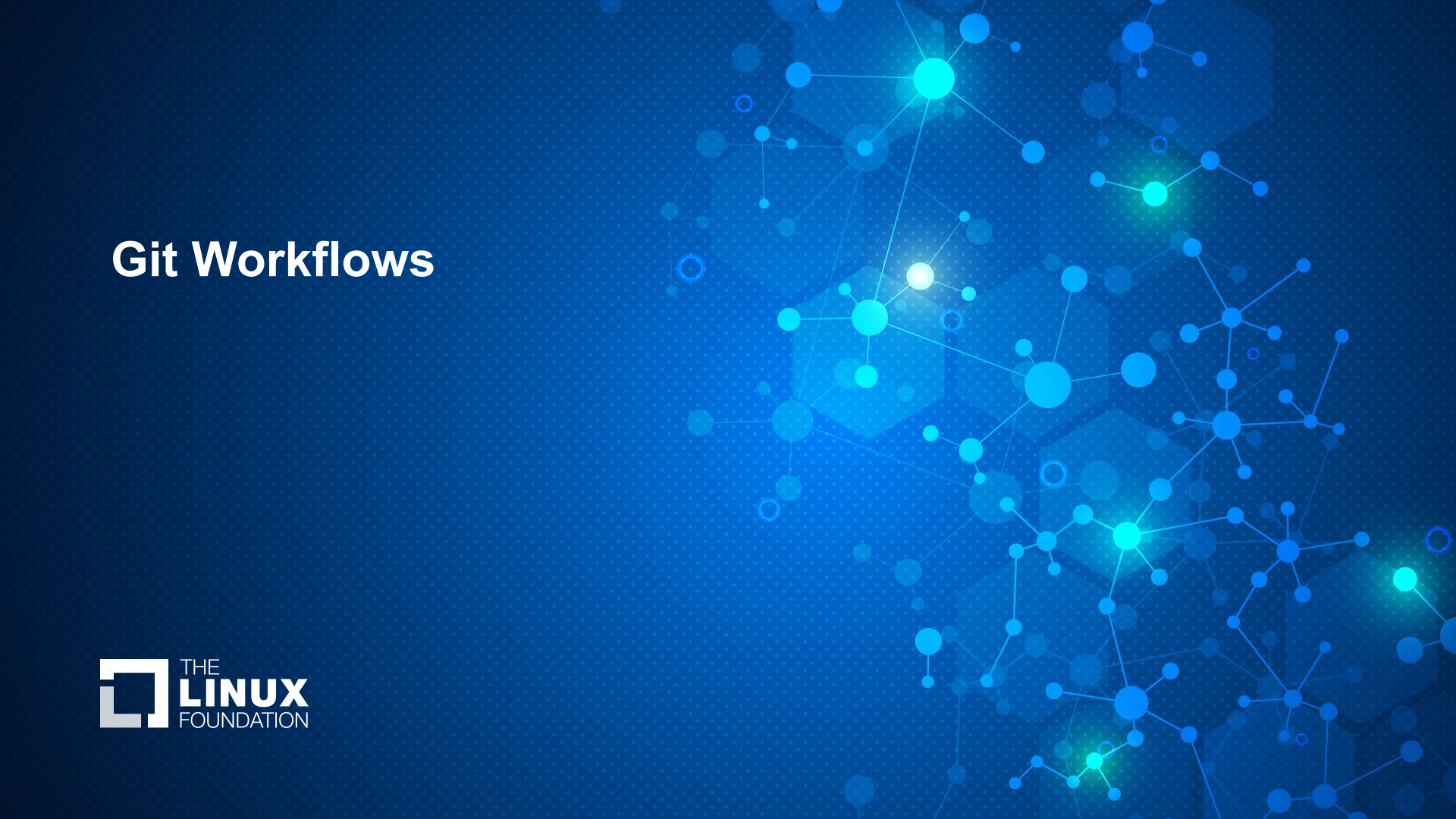
- Commit regularly
  - Backup of your work EOD
  - Incremental and shows progress
  - Mark it as Work In Progress (WIP).
  - Regular updates give an opportunity to collaborate with your team and seek early feedback
  - Multiple changes can be “squashed” down to fewer changes when ready for final merge

SCM: Commit meta-data

THE LINUX FOUNDATION

- Use commit footers for tracking needed information (aka: don't stick issue tracking in the subject line!)
- Common footers include (but are not limited to)
  - Signed-off-by: aka Developer Certificate of Origin (DCO): name and email
  - Issue-ID: Links to issue, bug tracking, PR number
- Other meta-data used by some of SCM's
  - Change-ID: Specific to Gerrit
  - Co-authored-by: multiple committers collaborate

Git Workflows



Git Workflows: GitHub

THE LINUX FOUNDATION

1. Ensure you have the latest changes (origin or upstream)
2. Make a local branch
3. Make changes locally
4. Commit and write a good message (see previous discussion)
5. Repeat until your change ready: Goto 3.
6. Push your local branch up for review
7. Open a Pull Request (PR) against the origin or upstream

Git Workflows: GitHub as an example

THE LINUX FOUNDATION

1. git checkout main
2. git pull
3. git checkout -b new_feature
4. # make your changes
5. git add <the_files_you_modified>
6. git commit -s
    - # -s adds that DCO line for you
    - # write a good message
7. # repeat steps 4 - 6 as needed
8. git push origin new_feature
9. # open your PR
10. # After the PR has finally been merged, delete your local (and remote) versions of the new_feature branch

Git Workflows: Gerrit as an example

THE LINUX FOUNDATION

1. git checkout main
2. git pull
3. git checkout -b new_feature
4. # make your changes
5. git add <the_files_you_modified>
6. git commit -s
    - # -s adds that DCO line for you
    - # write a good message
7. # repeat steps 4 - 6 as needed
8. git review

Source: https://docs.releng.linuxfoundation.org/en/latest/git.html#workflow-sample-1

Code Reviews

Helping Others and Yourself Improve Code

Code Review: Before you request

- Do a self-review
- Local tests pass aren't enough. Add new tests?
- Commit message is relevant and links to issue tracking
- Re-working code - address all the concerns raised previously
- Re-base on the latest HEAD
- Relevant documentation and unit tests
- Language’s standard linting (style, unreachable code …)
- Run tox and pre-commit locally

Code Review: Asking for Review (raising a PR)

- Push code from your local to remote repository on a branch (other than main)
- On GitHub UI open a PR (when pushing from the command line you will receive a URL in the response on how to open a PR as well)
- Trigger's CI automation on the receiving repository passed (GitHub Actions or other CI platforms)
- The maintainers of a project will be notified of the new change in the queue.

Code Review: Evaluating someone's code

THE LINUX FOUNDATION

- Wait for the change to pass CI verify/tests/gating jobs
- Read the PR cover, commit messages and understand what the contributor is trying to resolve
- Evaluate the code as though its your own changes
  - The best time to clean new code is when it's introduced!
- Evaluate if the problem resolution is satisfactory
- Evaluate if change abides the project's coding standards

Code Review: Evaluating someone's code (cont)

THE LINUX FOUNDATION

- Evaluate if code is secure
- Leave actionable recommendations as review comments
- Ensure you are able to justify the recommendations
- Don't be afraid to ask for changes. If the contributor disagrees, debate but do so in an agreeable manner
- Work through iterations
- Engage the contributors

Code Review: Evaluating someone’s code (cont pt2)

- Scope creep - don't ask for changes outside the scope.
- Be patient, Be polite, remember there is another person on the other side of the review

Validation Testing

Local Testing

Validation Testing

THE LINUX FOUNDATION

- Every language that is in general use has one (or more) testing systems that have been developed for it. Find out what it is for your language and learn it
- Testing of code should happen locally when possible before being put up for review
- The best method of developing your tests is to write them before you write your code. This is referred to as Test Driven Development (TDD). The development loop to follow is:
  - Create tests
  - Validate that the tests ***FAIL***
  - Write the code that is needed (the smallest amount possible)
  - Validate that the tests ***PASS***
  - Refactor the code as needed to make it acceptable
  - Propose your code, tests, and relevant documentation for review

Validation Testing (cont.)

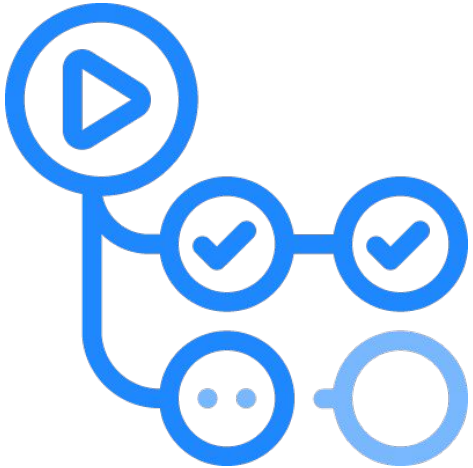
- Test coverage should be comprehensive
  - positive and negative outcomes
  - edge and corner cases
- Well written tests will allow you to later refactor your code and verify that you are not breaking any functionality

Continuous Integration and Deliver (CI/CD)

Automation to Help You Succeed

CI/CD: Robots Verifying and Deploying Code

THE LINUX FOUNDATION



CI/CD

CI/CD (cont.)

- Use pre-commit to catch issues before they go up for review
- Primary purpose of a CI/CD system is to validate code and secondary purpose is to deploy the code
- Use external validation tools in an automated manner
- Many systems are used for deploying the built code to production environments automatically
- The point of CI is to use it conjunction with the test that are developed in the code and to allow many people to collaborate on a code base more effectively
- If you have a CI system in use that validates code, allow it to operate and give feedback. If the project has well designed tests you may even consider removing the ability of maintainers from merging code without a vote by the CI system

Resources

Resources

- LF Release Engineering Best Practices: https://docs.releng.linuxfoundation.org/en/latest/best-practices.html
- Semantic (Conventional) Commit Messages: https://www.conventionalcommits.org
- Developer Certificate of Origin (DCO): https://developercertificate.org
- Pre-commit: https://pre-commit.com/
- Linux Foundation Mentorship Program: https://linuxfoundation.org/diversity-inclusivity/
- Practice TDD at https://cyber-dojo.org/

Development Best Practices:

An overview of working in the software world

Anil Belur
abelur@linux.com
@askb23

Andrew Grimberg
agrimberg@linuxfoundation.org
@tykeal