DevOps
SC-Camp 2021
Pedro Velho
PhD - UGA 2004-2010

...  
Engineer - Ryax - since 2019
Introduction to DevOps
Culture

Cross-functional Team
Developer, Operators, Security, and QA in the same team

End-to-end responsibility
"You build it, you run it"

Transparency
Share Knowledge, Collaborate, Data driven decisions

Customer centric
Understand business needs, Focus on features
Organization Transformation

**Functional team:**
- Focus on technologies
- Long V cycle of release
- Lots of communication friction
- No understanding of user needs
- No knowledge sharing or ownership

  - Developer Team
  - Security/QA Team
  - Operation Team

**Product Team:**
- Focus on product
- Fast Agile cycle of release
- More efficient communication
- Real understanding of user needs
- Share knowledge and ownership

  - Product A Team
  - Product B Team
Principles And Good Practices

Automate
Automate every steps with CI/CD

Shift left
Find bug and security issues early with automated tools

Release often, release early
Quick feature release and bug fixes

Agile Methodologies
Have rituals to synchronize a collaborate efficiently

Infrastructure as Code
Build reproducible and immutable containers and infrastructure

Fast Feedback
Observability: logs, metrics, and traces
DevOps Best Practices

Agile methodologies
1. Define your rituals (Daily, review, postmortem, retrospective, ...)
2. Randomly choose a Scrum Master
3. Design feature with the whole team, PO brings only business needs
4. Iterate and improve
   Shift left
5. Create unit tests and integration
6. Select code quality tools
7. Put tests and quality checks in the CI pipeline
8. Only integrate code if all the quality is good
   Automate:
9. Track all manual/implicit processes
10. Choose a documentation tool
11. Document them
12. Automate them if possible

Infrastructure as Code
1. Choose appropriate tool
2. Code your infrastructure
3. Deploy it within the CI
4. Forbid any manual changes
   Fast Feedback
5. Get metrics, logs and traces from your apps
6. Setup alerting on user focus signals
7. Setup dashboards to observe and explore
   Release often, early
8. Implement other practices
9. Automate deployment with CD
10. Monitor and rollback if necessary
11. Track the time-to-deliver and try to improve it!
DevOps Life Cycle Tools

Plan
• Use knowledge sharing tools
  Example: wiki, issue tracker, whiteboard and post-it

Code
• Use an IDE with integrated quality check tools
  Example: VSCode, PyCharm
• Enforce code integration process with code review
  Example: Gitlab, Github

Build
• Enforce reproducible build
  Example: Poetry (Python), Yarn (NodeJs)
• Package in a container
  Example: Docker, Kaniko, Nix, Buildah

Test
• Create a pyramid of test with coverage
  Example: Pytest (Python)

Release
• Push build artifact to a registry with a unique version tag
  Example: Skopeo, DockerHub, Harbor

Deploy
• Deploy your container(s) on a cluster
  Example: Kubernetes, Nomad

Operate
• Manage infrastructure updates
  Example: Helm, Kubeadm

Monitor
• Monitor important user-facing signals (latency, traffic, errors, saturation)
  Example: Prometheus, Grafana
• Setup an alerting system
  Example: Prometheus, Slack
DevOps: Culture, Practices, Tooling

Culture

Principles And Good Practices

Dev

Ops

- code
- plan
- release
- deploy
- monitor
- operate
- test
- plan
- line
Versioning

- Introduction to versioning with git
- commit
- pull/push
- merge
- why not use rebase?
Versioning

• Code Versioning Systems propose:
  Keep track of changes, who did what and when?

• Examples:
  Fast rewind to find code before a change/feature
  Fast forward to a new experimental feature
  Find first change that introduced a bug
  Who introduced the bug and when? Blame!

Jhon Romeros talk: *Doom's a year of madness*: “How did you handle version control back then? There was none,... We just watched not to touch other presons' files”, see full talk here [https://www.youtube.com/watch?v=eBU34NZhW7I](https://www.youtube.com/watch?v=eBU34NZhW7I)
Versioning

• Git, a bit of history

  Other CVS exist: cvs, mercury, svn
  Not adapt well to the linux kernel development culture
    Mainly remote devs and volunteers geographically spread
  Git was one of the first to introduce CVS in a distributed manner
    Heavily influenced by the non-free solution bitkeeper
  Git book is free to read online https://git-scm.com/book/en/v2
Versioning

- Introduction to git
  Every code change is explicit commit by the dev
  The commits make a history tree (formaly a graph)
  Every node on the tree is a commit (code change)
  Parallel changes are allowed (branch)
  Parallel changes can be merged
Versioning

- Bump 0.0.25
- Remove setuptools from ci, it is already there.

- Bump 0.0.24
- Merge branch 'add_project' into 'master'
- add project to CLI
- Fix yaml
- Fix release shell
- Add a release step in the CI to create tarball
Versioning

- Bump 0.0.25
- Remove setuptools from ci, it is already there.
- Bump 0.0.24
- Merge branch 'add_project' into 'master'
- Add project to CLI
- Fix yaml
- Fix release shell
- Add a release step in the CI to create tarball
Versioning

0.0.25
- Bump 0.0.25
- Remove setuptools from ci, it is already there.

0.0.24
- Bump 0.0.24
- Merge branch 'add_project' into 'master'
- Add project to CLI
- Fix yaml
- Fix release shell
- Add a release step in the CI to create tarball
Versioning

Commit

Branch

Merge

- Bump 0.0.25
- Remove setuptools from ci, it is already there.
- Bump 0.0.24
- Merge branch 'add_project' into 'master'
- Add project to CLI
- Fix yaml
- Fix release shell
- Add a release step in the CI to create tarball
Versioning

- The remote history (commit tree) is shared amongst devs
- Every dev copy the full history to start working
- Devs then make changes on the local history
  - Branch
  - Commit
  - Merge
- Once changes are ready devs can **push** changes to the remote
- Devs can **pull** remote changes as well
- Beware of conflicts
Versioning

- Working example 1 dev (Bob)
  - Clone from initial commit
    - `git clone`
  - Add a README.md file
    - `git add README.md`
    - `git commit -m "Add README.md FILE"`
  - Update README.md file
    - `git add README.md`
    - `git commit -m "Update README.md file"`
Versioning

- Working example 1 dev (Bob)
  - Clone from initial commit
    
    
  - git clone
  - Add a README.md file
    
    
  - git add README.md
  - git commit -m "Add README.md FILE"
  - Update README.md file
    
    
  - git add README.md
  - git commit -m "Update README.md file"
Versioning

- Working example 1 dev (Bob)
  
  Clone from initial commit

  ```
  git clone
  ```

  Add a README.md file

  ```
  git add README.md
  ```

  Update README.md file

  ```
  git add README.md
  ```

  ```
  git commit -m "Add README.md FILE"
  ```

  Update README.md file

  ```
  git add README.md
  ```

  ```
  git commit -m "Update README.md file"
  ```
Versioning

- Working example 1 dev
  Remote history only add B1 and B2 after bob calls
  `git push`

```
Remote history

Bob's history

R1 Initial commit

B1 Add README.md file

B2 Update README.md file
```
Versioning

- Working example 1 dev
  Remote history only add B1 and B2 after bob calls git push
Versioning

- Working example 2 devs (ana & bob)
  Ana start fresh clone from R1
  Bob start fresh clone from R1
  Bob finishes and push his work to remote upstream
  Ana finishes and push her work to the upstream
  What happens?
Versioning

- Working example 2 devs (ana & bob)

**Remote history**
- B3: Update README.md file
- B2: Add README.md file
- R1: Initial commit

**Bob's history**
- B3: Update README.md file
- B2: Add README.md file
- R1: Initial commit

**Ana's history**
- A1: Add license.txt file
- R1: Initial commit
Versioning

• Working example 2 devs (ana & bob)

Remote history

B3
Update README.md file

B2
Add README.md file

R1
Initial commit

Bob's history

B3
Update README.md file

B2
Add README.md file

R1
Initial commit

Ana's history

A1
Add license.txt file

R1
Initial commit
Versioning

- Working example 2 devs (ana & bob)

Remote history

- B3: Update README.md file
- B2: Add README.md file
- R1: Initial commit

Bob's history

- B3: Update README.md file
- B2: Add README.md file
- R1: Initial commit

Ana's history

- A1: Add license.txt file

Bob clones project

Ana clones project

git add
git commit
Versioning

- Working example 2 devs (ana & bob)

Remote history
- B3: Update README.md file
- B2: Add README.md file
- R1: Initial commit

Bob's history
- B3: Update README.md file
- B2: Add README.md file
- R1: Initial commit
- Bob clones project

Ana's history
- A1: Add license.txt file
- R1: Initial commit
- Ana clones project
Versioning

- Working example 2 devs (ana & bob)

Remote history

Bob's history

Ana's history

Bob clones project

Initial commit

Add README.md file

git add

git commit

Update README.md file

Bob pushes his changes first

Ana clones project

Initial commit

Add license.txt file

Update README.md file

Bob pushes his changes first

Add README.md file

Ana pushes her changes first
Versioning

• Working example 2 devs (ana & bob)

Remote history

Bob's history

Ana's history

Bob pushes his changes first

Bob clones project

Ana clones project

Push is rejected! Ana need to sync with remote history first

Initial commit

Add README.md file

git add

git commit

Update README.md file

Add license.txt file

Update README.md file

Add README.md file

Add README.md file

B3

B2

B3

B2

A1

R1

R1

R1
Versioning

- Working example 2 devs (ana & bob)

Ana execute command to pull remote changes

`git pull`

When pulling the differer

`git push`

Ana's commit goes after
Versioning

- Conflicts can happen and will happen
- Always branch before start working
  ```
  git checkout -b "my-branch-name"
  ```
- 2 devs work on the same file and same branch
  Last dev to push, is the one that must solve the conflict
- Avoid commands that rewrite the history
  Never use -> git rebase
- Avoid commiting on the master branch
  Create a new branch for each dev
Versioning

- Gitlab merge-request
- Every change on the main branch affect all users and the new release
- In gitlab a merge-request allows to keep track of changes on the main branch (simplification)
- Advantages on the CI process:
  - Keep a clear track of what changes are upcoming
  - Show the code changes for a specific request
  - Allows a review process to take place
    - Ideally not the same dev will look at the code and validate
    - Ideally not the same dev will merge the changes
Versioning

- Gitlab merge-request
Versioning

- Gitlab merge-request
Versioning

- Gitlab merge-request

Choose a source branch

Choose a target branch

Compare branches before creating the MR
Versioning

- Gitlab merge-request
Versioning

- Gitlab merge-request
Versioning

- Gitlab merge-request

Choose a source branch

Choose a target branch
Versioning

- Gitlab merge-request

Choose a source branch

Choose a target branch

Compare branches before creating the MR
Versioning

- Gitlab merge-request
  - Set a title and description
- Assignees
  - Person that responds to this MR
- Reviewers
  - Person to approve this MR
Versioning

- Gitlab merge-request

Scrolling down we can see

Changes

Commits
Versioning

- Settings -> General
- Merge request approvals
  # reviewers that must approve
Versioning

• How this scenario would work with gitlab review?

Remote history  Bob's history  Ana's history

Bob pushes his changes first

Bob clones project

Update README.md file

Add README.md file

Initial commit

B3  B3  A1

git add git commit

git add git commit

R1  R1  R1

Ana clones project

Add license.txt file

Initial commit

Initial commit
Versioning

- How this scenario would work with gitlab review?

Remote history

<table>
<thead>
<tr>
<th>Bob's history</th>
<th>Ana's history</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3 Update README.md file</td>
<td>A1 Add license.txt file</td>
</tr>
<tr>
<td>B2 Add README.md file</td>
<td></td>
</tr>
<tr>
<td>R1 Initial commit</td>
<td>R1 Initial commit</td>
</tr>
</tbody>
</table>

Bob can create a MR and Ana reviews and approves it on main branch.
Versioning

• Work on a multi-tenant gitlab project
  • Precommit - add basic coding checks style
  • merge request
  • review process
  • approval system