# Version control with Git, Github and Gitlab

Team project workshop

Michal Sojka
ČVUT, CIIRC

8. 3. 2022

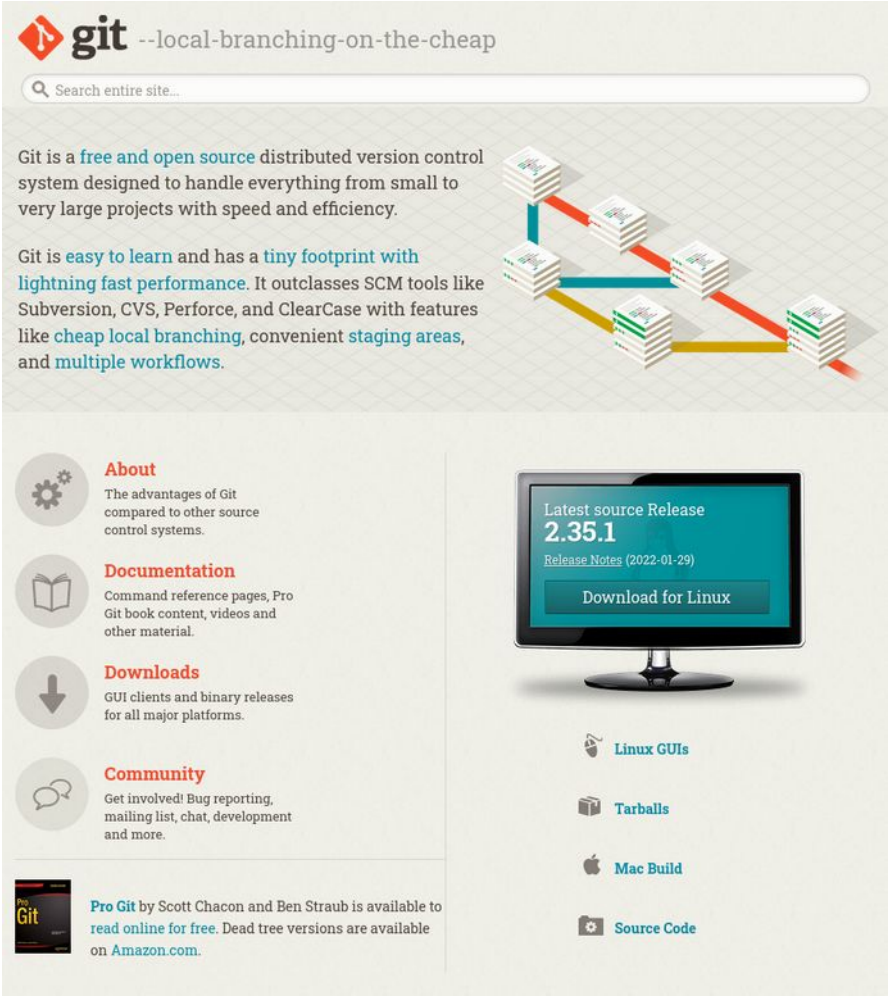
# Contents



Presentation available from:
https://github.com/pvty/presentation/

# Git – https://git-scm.com/

- Version control system
- Started by Linux Torvalds for version control of the Linux kernel
- Distributed – no need for a central repository (peer-to-peer)
- Fast, efficient
- Command line tool + many GUIs
  - git <subcommand> ...



# Git use cases

- Source code version management
- Documentation (e.g. Markdown)
- Wiki
- Website
- Configuration management
- File comparison (advanced diff)
  - observing changes in program output
  - ...
- Storage of virtual machine images...
  - extensions like *git lfs* and *git annex*

# Basics commands (local repo)

- Basic commands
  - git init – create an empty repository
  - git add – add files to index (staging area)
    - git add -p
  - git commit
  - git log
  - git diff
    - git diff –color-words
  - git help, git <command> --help
  - git stash
- “Built-in” GUI
  - git gui – for creating commits
  - gitk – history viewer
- Branches, tags
  - Branch = “pointer” to a commit
    - typically the last commit in development history
    - creating a commit updates the pointer
  - Tag = “fixed pointer” to a commit
    - versions, releases
  - Merging, rebasing – see later
  - git branch
  - git checkout/switch
- HEAD – points to the currently active commit
  - usually same as current branch
  - can be detached from a branch (risky)

# Working with remote repositories

- git clone – create a local copy of a remote repo
- git fetch – download updates from remote repo, but don't modify any local files, commits etc.
- git push – push local changes to the remote repo
  - diverging branches
  - force pushing (--force, --force-with-lease)
- git pull – pull remote changes to the local repo
  - fetch + merge
- git remote – working with multiple remote repos
  - upstream (typically read-only)
  - personal fork (read-write)
  - local CI server
  - multiple personal computers
- Remote branches
  - Each remote has its own branch namespace
  - Easy to compare state of repositories
  - local branch can be set up to track a remote branch
    - git push/pull works automatically
    - visible in `git branch -vv`

# Investigating git history

- git log ≅ gitk – show history, accepts similar arguments
- Specifying revisions and ranges:
  - <commit> = hash, branch, tag, ...
  - git rev-parse –help
- Revisions:
  - gitk = gitk HEAD
  - gitk HEAD^, gitk HEAD^^ – parent commits
  - gitk mybranch~5 – five commits back from mybranch
  - gitk main mybranch – show two branches
- Ranges
  - gitk HEAD~5..HEAD = HEAD~5.. - five last commits
  - gitk main..myfeature = gitk myfeature ^main
    Commits in myfeature but not in main.
  - Symetrical difference:
    gitk main...myfeature
- Filter by file, directory
  - gitk ... [--] subdirectory
  - Follow file renames:
    gitk --follow dir/file

# Git internals

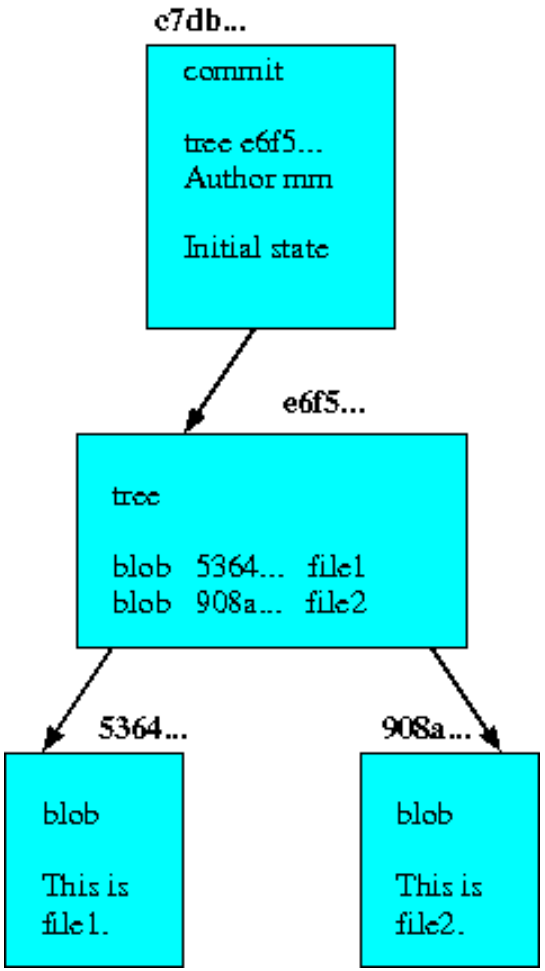
- Very efficient storage of history:
  - Unpacked Linux 5.16 sources:
    - du -sh --exclude=.git: **1.2 GiB**
  - 17 years of Linux development history = 1.06 million commits = 170 commits every day (in average)
    - du -sh .git: **2.6 GiB**

# Git internals – components

- **Object Database**
  - collects objects of four types:
    - blob,
    - tree,
    - commit,
    - tag
  - objects are addressed by SHA1 hash of their content
  - individual objects can shown by `git cat-file -p <hash-or-similar>`
- **Index (staging area)**
  - current tree **cache**
  - stores the next revision to be committed

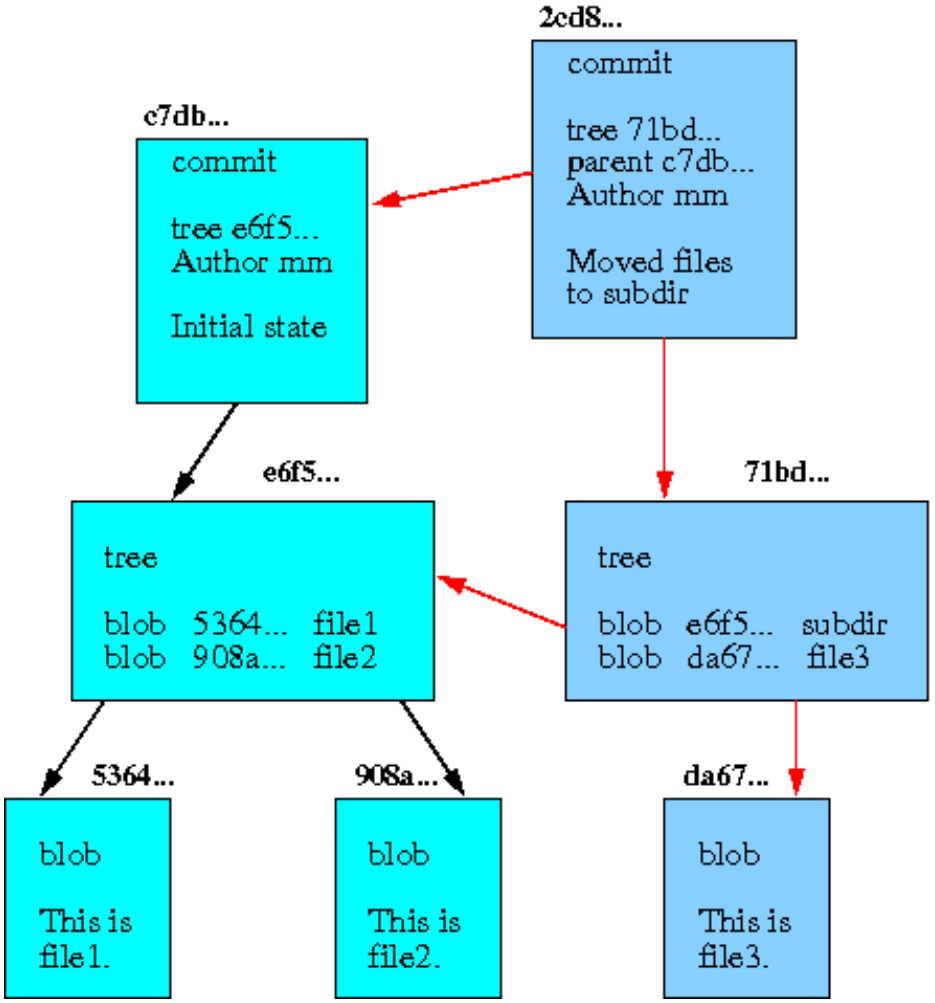
# Example: Object Database I.

- 1. Start with a new repository
- 2. Create file1 with the content: “This is file1.”
- 3. Create file2 with the content: “This is file2.”
- 4. Update the index
- 5. Make an initial commit



# Example: Object Database II.

- 1. Move file1 and file2 into subdirectory
- 2. At top level, create file3 with the content: “This is file3.”
- 3. Update the index
- 4. Make a commit



# Object Database I.

- **blob object**
  - represents **contents** = one version of a file
  - if two files in a directory tree (or in multiple different versions of the repository) have the same contents, they will share the same blob object
- **tree object**
  - represents one directory
  - contains sorted list of text lines with the following information: *mode,object type, SHA1, path name*
  - information about blobs and tree objects lying in the directory
  - several tree objects forms hierarchical directory **structure**

# Object Database II.

- **commit object**
  - contains by the reference to related tree object, the parent commits, commentary
  - sequence of commit objects provides the history
  - commit objects tie the directory structures together into a  acyclic graph (DAG)
- **tag object**
  - assigns symbolic name to particular object reference e.g. commit object associated with a named release
  - contains SHA1, object type, symbolic name of referenced object and optionally a signature

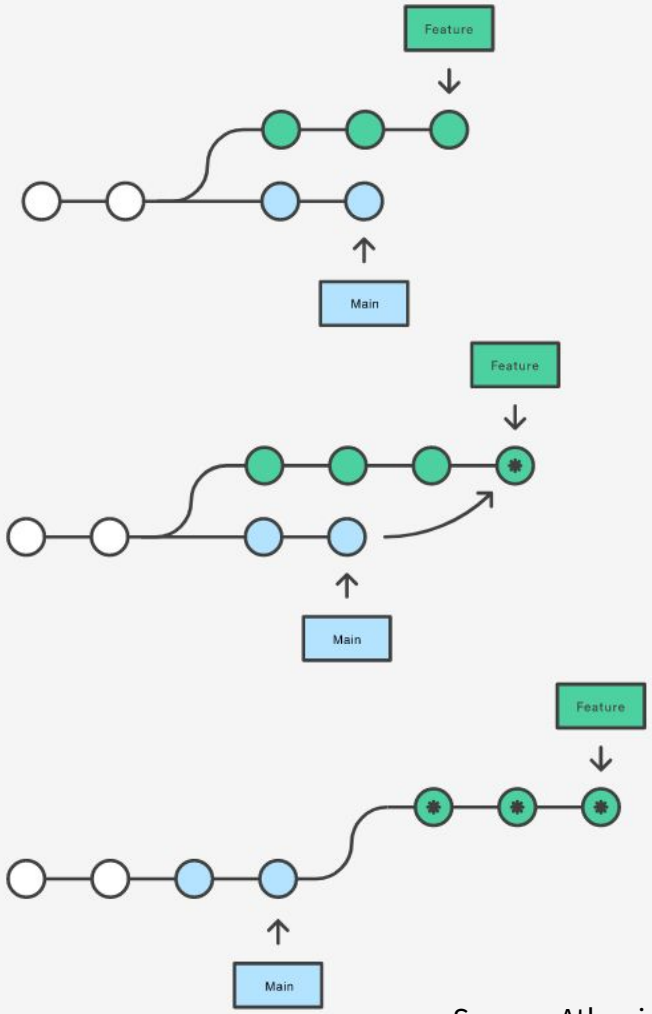
Objects are not compressed individually, but together – multiple (similar) versions of the same blob are compressed together ⇒ efficient.

# Index

- simple binary file, which contains an efficient representation of a virtual directory content
- it is implemented by a simple array that associates a set of names, dates, permissions and content (blob) objects together
- serves as staging area to prepare commits
- helps with merge conflict resolution
- improves performance (speed of operations)

# Merging, rebasing, conflicts

- git checkout feature
- git merge main
  - Creates a commit with two parents and merged content
- git rebase main
  - Moves all commits from feature on top of main
  - Don’t use on public branches!!!
- git rebase -i
  - interactive rebase = editing history (commits, commit messages, ...)




Source: Atlassian

# Handling of merge/rebase conflicts

- Why conflicts emerge? Two people edit the same piece of code.
- Must be resolved manually
- Tools can help (e.g. gitk –merge, git mergetool)
- `git merge branch`
  `Auto-merging src/boardproxy.cpp`
  `CONFLICT (content): Merge conflict in src/boardproxy.cpp`
  `Recorded preimage for 'src/boardproxy.cpp'`
  `Automatic merge failed; fix conflicts and then` **`commit`** `the result.`
- 

```
static struct argp argp = {
    options, parse_opt, "[SOCK_DIR]",

<<<<<<< HEAD
    "Great tool to access a pool of remote boards (or other resources) with "
||||||| c05dace
    "Access a pool of remote boards (or other resources) with "
=======
    "Poor tool to access a pool of remote boards (or other resources) with "
>>>>>>> conflict
    "connection proxying via SSH port forwarding."
};

int main(int argc, char *argv[])
{
    argp_parse(&argp, argc, argv, 0, 0, NULL);
```

- After merge conflict:
  - git merge --abort
  - Fix conflict and then:
    git merge - -continue
- After rebase conflict:
  - git rebase --abort
  - git rebase --skip
  - Fix conflict and then:
    git rebase --continue

# Git forges

- Github.com
  - proprietary (Microsoft)
  - most popular
  - single point of failure for open source movement
  - Used mostly in the following slides

- Gitlab
  - open source
  - gitlab.com, gitlab.fel.cvut.cz
  - feature-wise comparable to github

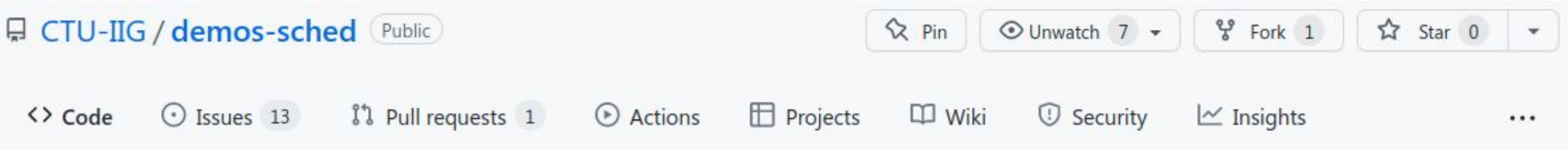
- Gitea
  - open source (https://gitea.io/)
  - very good for self-hosting (easy to install)
  - codeberg.org

- sourcehut
  - open source
  - hacker friendly
  - http://sr.ht

# Typical forge features

- Organizations/Teams/Repos
- Issues/Tickets
- Pull requests (Merge requests)
- Continuous integration (CI/CD)
- Integration with 3rd party tools (webhooks, ...)
- Websites (pages), Wiki
- Release management
- Project management
- Development statistics





# Repositories

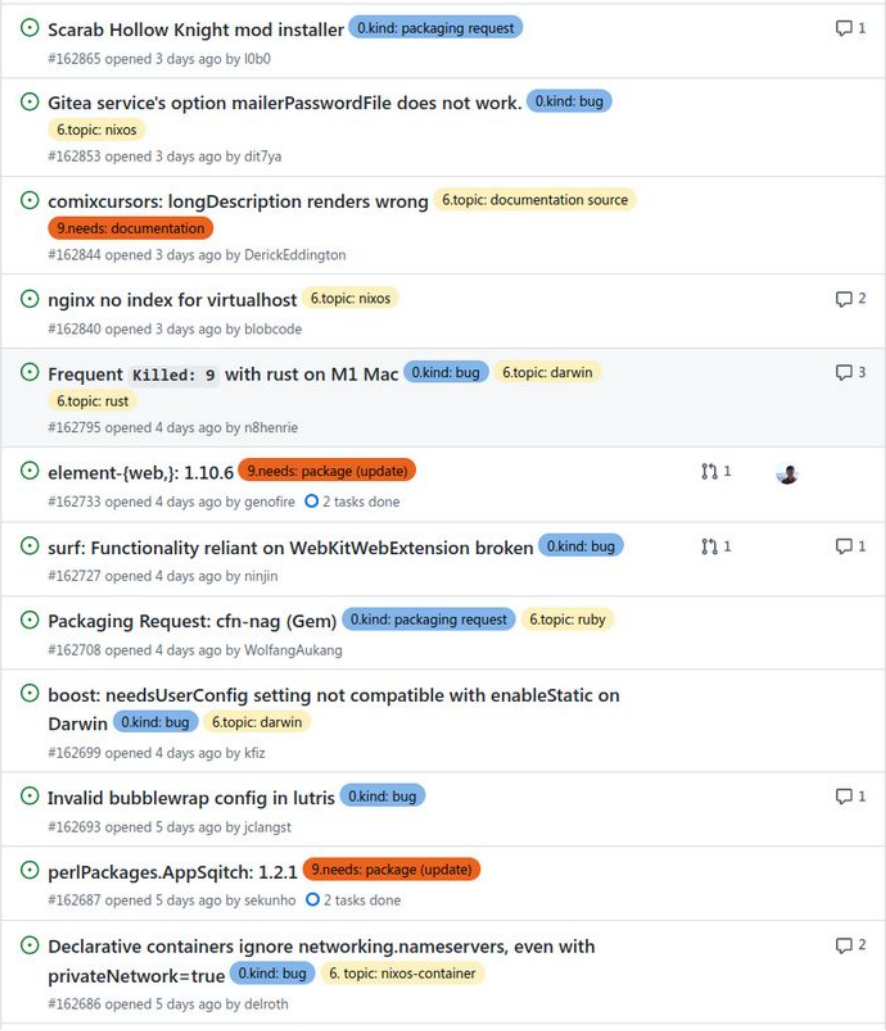
- Creating a new repository (few clicks)
  - Specify the license for public repos (otherwise, nobody can use your project legally)
- Forking existing repositories
  - Mostly for proposing changes to the original (upstream) repo

**Types of repositories on Github**

- Personal repos: username/reponame
  - simple permission management (write or nothing)
- Organizations: orgname/reponame
  - Team member management
  - Role management
    (admin, maintain, write, triage, read)

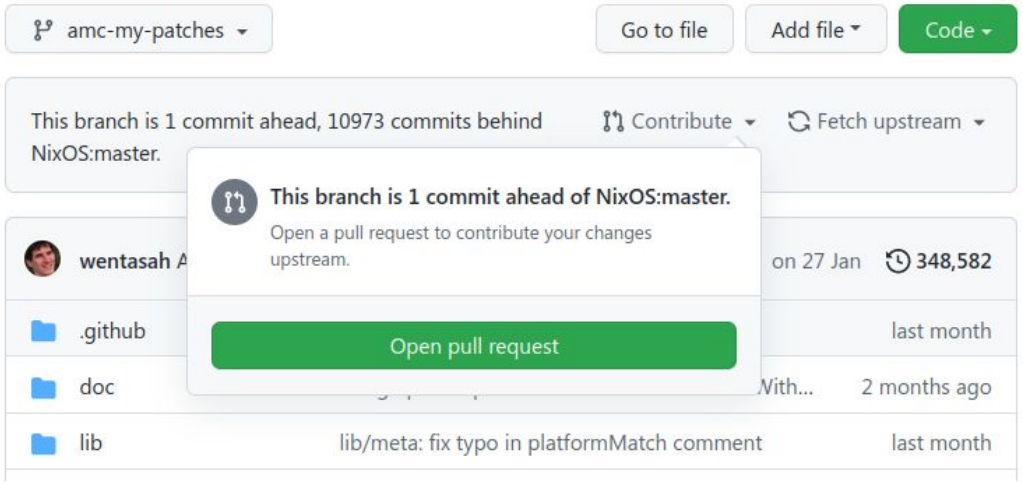
# Issues

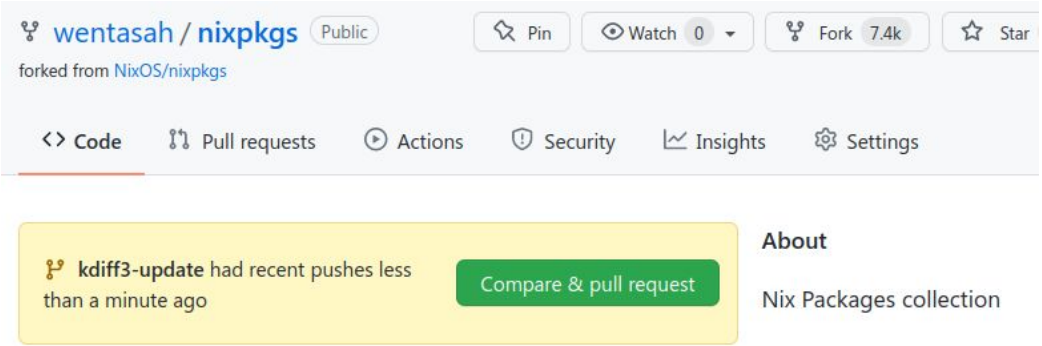
- Communication between users
- Templates (documentation)
  .github/ISSUE_TEMPLATES/...
- Labels (bug, question, ...)
- References (documentation)
  - to other issues: #NN
  - to issues in other projects: owner/repo#NN
- Automatic closing of issues with keywords in commits/PRs (documentation):
  *Closes #13, Fixes #34, ...*



# Pull requests (PR, Github), Merge requests (MR, Gitlab)

- For proposing changes and discussing about them
- Opened via “contribute” link or offered after push:
- source branch, destination branch
  - same repo
  - from fork to upstream repo
- Protected branches
  - restriction of direct pushes to the branch
  - configurable rules:
    - Require PR before merging
    - Required number of approvals
    - CI status

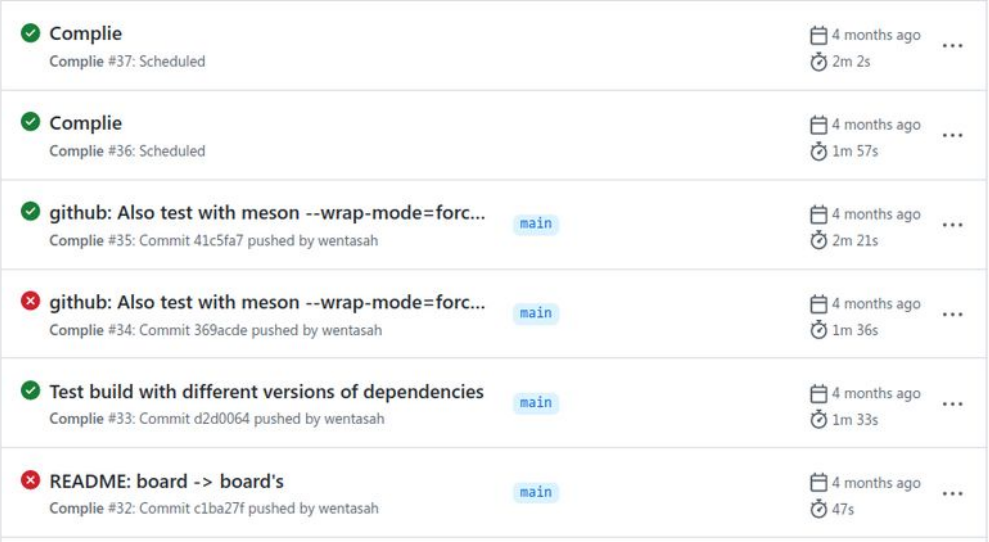


# Continuous integration/deployment

- Run test suite, checks, ... on every push, PR, time, ...
- Github Actions (quickstart docs)
  - Configured with a YAML file (doc) in .github/workflows directory in the repo.
    - When to run, what to run
    - Test matrix (test your code with multiple versions of a dependency)
  - Github Actions run in the cloud (2 CPUs, 7 GB RAM) or on self-hosted runners (doc)
  - You can execute plain (Linux) commands (*run* keyword) or whole actions (*uses* keyword)
- Secret management
- problem-matchers

- Example with a simple (1D) matrix and 3rd-party actions:
  https://github.com/wentasah/meson-mode/blob/master/.github/workflows/run-tests.yml
- Example with combination of 3rd-party actions and "run" commands:
  https://github.com/wentasah/boardproxy/blob/main/.github/workflows/test.yml



# Integration of 3rd-party services

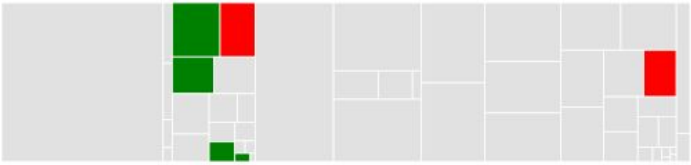
- Example: https://codecov.io
- Generate test coverage report as a part of your CI
- Upload the report to codecov.io:
- Example workflow

```yaml
steps:
  - ...
  - name: Generate coverage report
    run: ninja -C build coverage-xml
  - name: Upload coverage to Codecov
    uses: codecov/codecov-action@v2
    with:
      directory: ./build/meson-logs
      fail_ci_if_error: true
      verbose: true
```

**codecov-commenter** commented on 14 Dec 2021

## Codecov Report

Merging #74 ( `073f056` ) into master ( `51086b7` ) will decrease coverage by `1.60%`.
The diff coverage is `2.47%`.



```
@@            Coverage Diff            @@
##           master      #74      +/-   ##
==========================================
- Coverage   36.39%   34.79%   -1.61%
==========================================
  Files          46       50       +4
  Lines        2459     2578     +119
  Branches     1117     1172      +55
==========================================
+ Hits          895      897       +2
- Misses       1094     1209     +115
- Partials      470      472       +2
```

| Impacted Files | Coverage Δ | |
|---|---|---|
| src/power_policy/_power_policy.cpp | 22.22% <0.00> (-1.59%) | |

# Gitlab webhook examples

- Run a script on every push:
  https://gitlab.fel.cvut.cz/sojkam1/brute-ae-sync/-/blob/master/gitlab-webhook-receiver.py
- Manage issues for osy.pages.fel.cvut.cz (uses Gitlab HTTP API):

```python
#!/usr/bin/env python3
import gitlab

cviceni = { ... }
teachers = { .... }

def application(env, start_response):

    def handle_event(env, event):
        if \
           not "HTTP_X_GITLAB_TOKEN" in env \
           or env['HTTP_X_GITLAB_TOKEN'] != \
              HTTP_X_GITLAB_TOKEN:
            return False

        if event['object_kind'] == 'issue' and \
           event['object_attributes']['action'] == 'open':
            return handle_open_issue(env, event)

        return True
```

```python
def handle_open_issue(env, event):
    gl = gitlab.Gitlab('https://gitlab.fel.cvut.cz', env['GITLAB_API_TOK
    p = gl.projects.get(event['project']['id'])
    student = event['user']['username']
    teacher = teachers[cviceni[student]]
    users = p.users.list(search=teacher)
    for u in users:     # Find exact username
        if u.username == teacher:
            user = u
    issue = p.issues.get(event['object_attributes']['iid'])
    if re.search('přednáška.*slide', event['object_attributes']['title']
                 re.IGNORECASE) is not None:
        # Přednáška
        issue.labels.append('Přednášky')
    else:
        # Cvičení - přiřaď cvičícího
        if not event['object_attributes']['assignee_ids']:
            issue.assignee_ids = [user.id]
    issue.save()

    if event['object_attributes']['confidential']:
        issue.notes.create({'body': textwrap.dedent("""

        Je opravdu nutné utajovat toto issue? Sdělujete nám
        zde svá hesla, čísla platebních karet apod.? Pokud ne,
        změňte prosím toto issue na veřejné, aby odpovědi
        učitelů byly dostupné i ostatním.

        """)})
    return True
```

# Github/Gitlab pages

- Maintain a website in a git repo
  - Documentation to a project
  - Plain web sites, ...
- Content in Markdown (or similar) formats
- Every push triggeres CI/CD and updates the web site
- Github – Jekyll static site generator
- Other SSG: Hugo
- Gitlab example: http://osy.pages.fel.cvut.cz/
  - CI config with Hugo