Git
&
Gitlab
For Engineers

Tejas Roysam

JSC/EV3

Agenda

Part 2: Gitlab

- Goals
- Overview
- Workflow
- Gitlab CI/CD

Goals

- To promote awareness of centralized tools for project coordination and development.
- To demonstrate the value of Gitlab as one such tool.
- To step through the core (free/open source) Gitlab features.
- To understand Continuous Integration and Deployment.

What does Gitlab add?

- Git is the focal point of Gitlab
- Gitlab brings centralization
  - Single interface
  - Single data hub
  - Single security gate
  - Single collaborative focus
- Bridge between project development and project operations
- Gitlab is a project life-cycle tool
  - Ex: NASA Spaceflight Program and Project Lifecycles are defined by NPR 7120.5

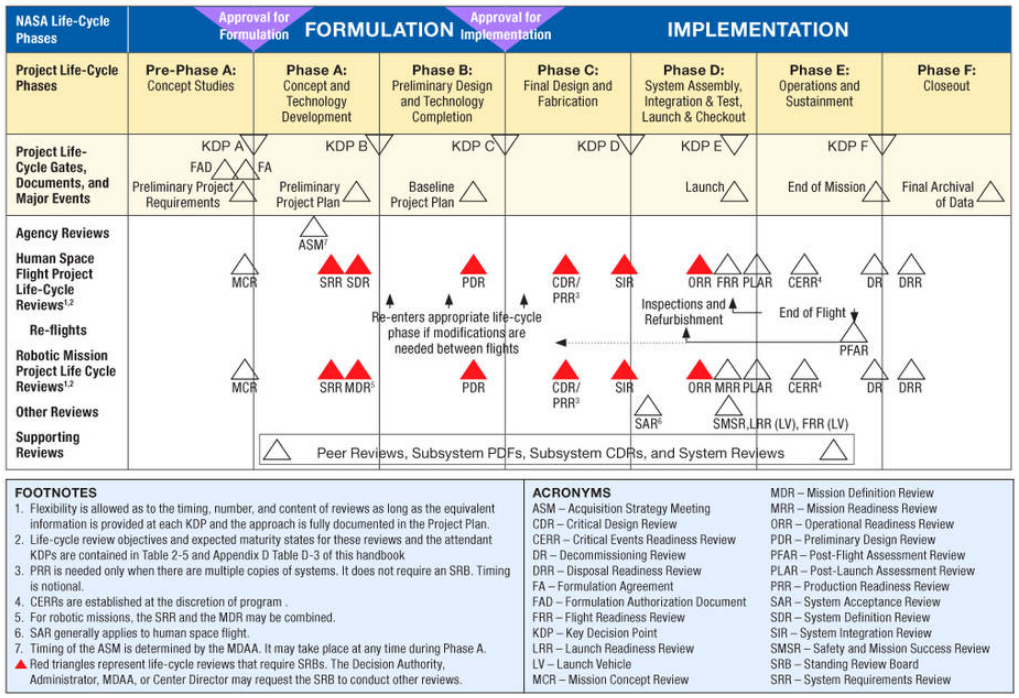


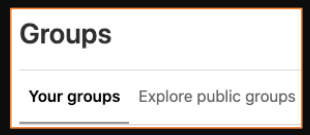
FIGURE 3.0-1 NASA Space Flight Project Life Cycle from NPR 7120.5E

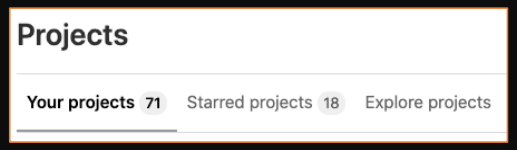
Why bother?

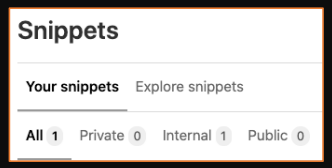
- **Engineers work together**: One-stop shop for collaboration
  - Intra-team
  - Inter-team
  - Access control & contribution control
  - Reused work > repeated work
- **Project unity**
  - Priorities, timetables, understandings between teams should align
  - Project managers at any level (component, subsystem, system) have clear and granular visibility of progress, action items, upcoming issues and deadlines, etc.
- **Product management**: Code is not the only thing that changes frequently
  - Tracking drawings, documents, schematics, lessons learned, etc.
- **Peer review**
- **Test automation**
  - Regression testing, HSI testing
- **Documentation automation**

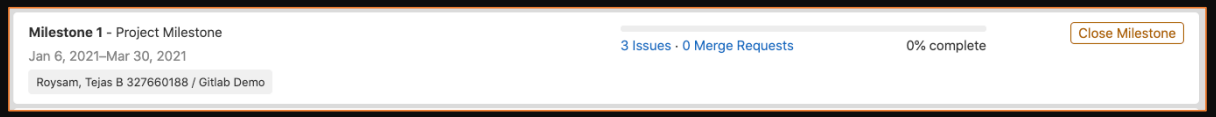
Gitlab Fundamentals Global View

- Projects
  - Find projects to contribute to
  - Create new projects
  - Explore existing projects for solutions or products
- Groups
  - Collections of related projects
- Snippets
  - Share code, info, tips, etc. with individuals, groups, or everyone
- Milestones
  - Group issues/tasks into project milestones



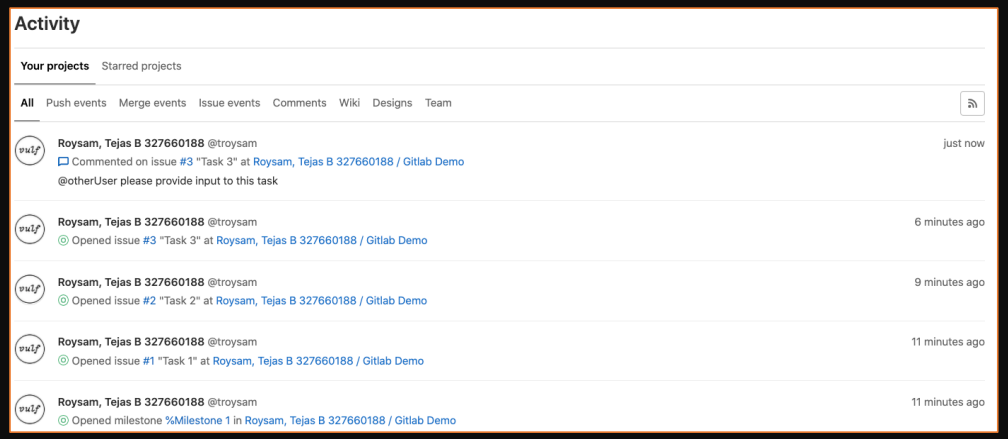


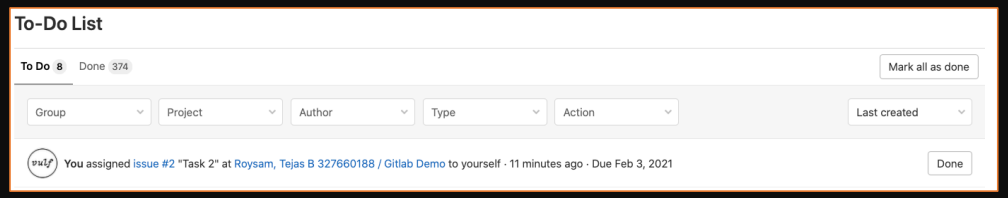


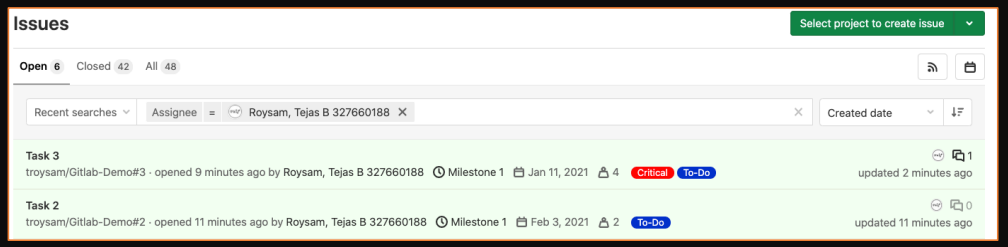


Gitlab Fundamentals Global View

- Activity Feed
  - Monitor goings-on
- To-Do lists
  - Track assigned tasks
  - Upcoming peer reviews
  - Mentions from other users
- Issue tracking
  - View all issues assigned to you by yourself or others, across all your projects

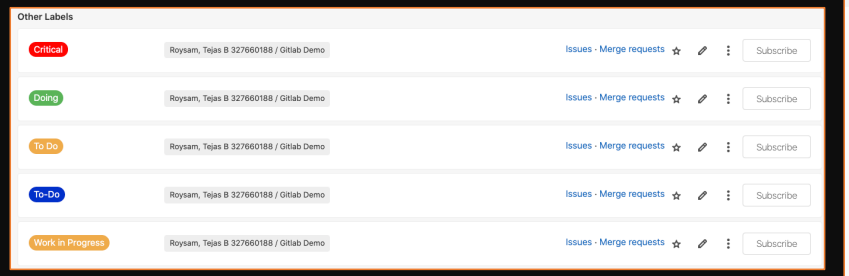


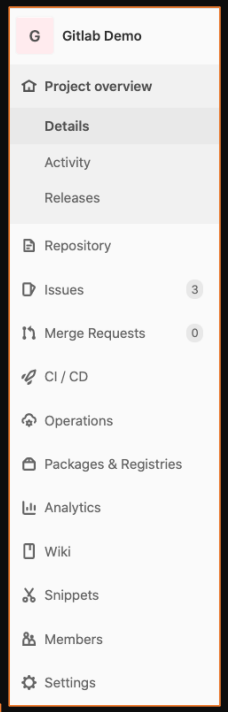


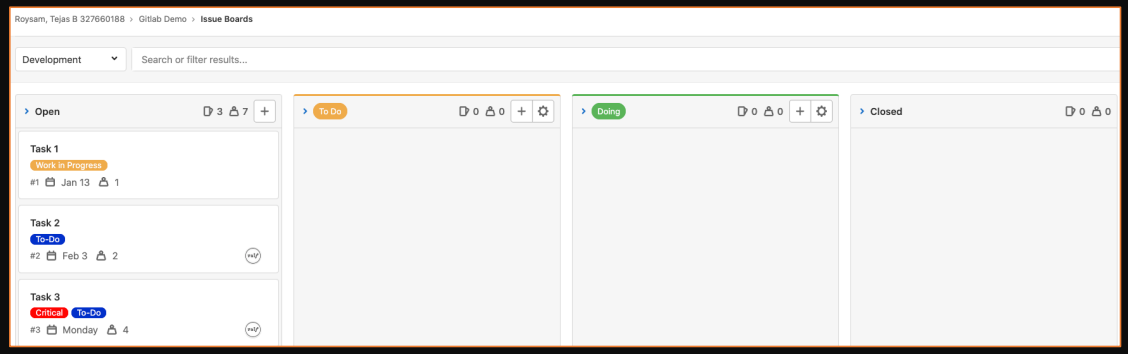


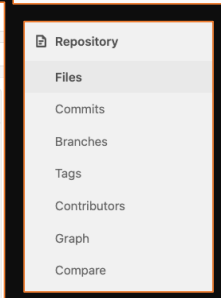
Gitlab Fundamentals
Project View

- Repository – all things Git
  - Browse all tracked content at any point in history
  - See details of contributions/authorship, releases, etc.
- Issues
  - Customizable Kanban Board view
  - Service desk: allow customers/outside parties to submit issues
  - Labels: organization and delegation
  - Due dates
  - Weights
  - Visibility/Confidentiality
  - Time tracking









Gitlab Fundamentals
Project View

Merge requests

Peer review of all contributions to the protected/revision-controlled body of work

Continuous Integration/Continuous Deployment

Automation of testing, generation tasks, asset deployment, and anything else automatable

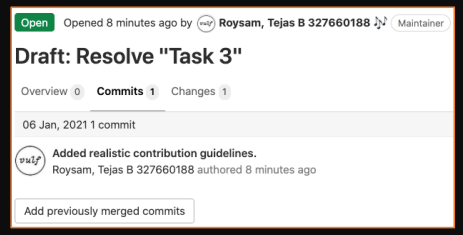
Big topic – deep dive ahead

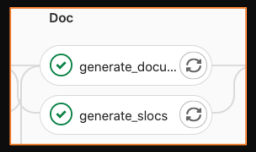
Operations

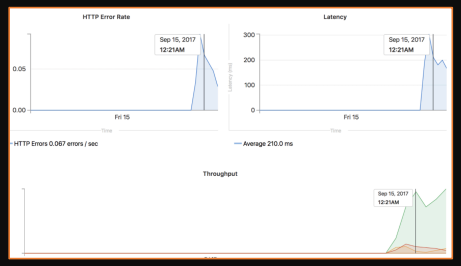
Tools for software continuously deployed to environments

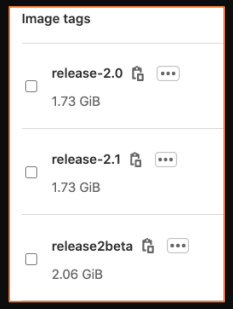
Package/Container Registry

Interface for publishing and sharing packages and virtual images



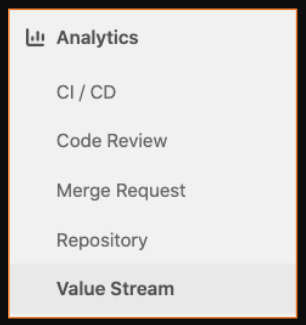


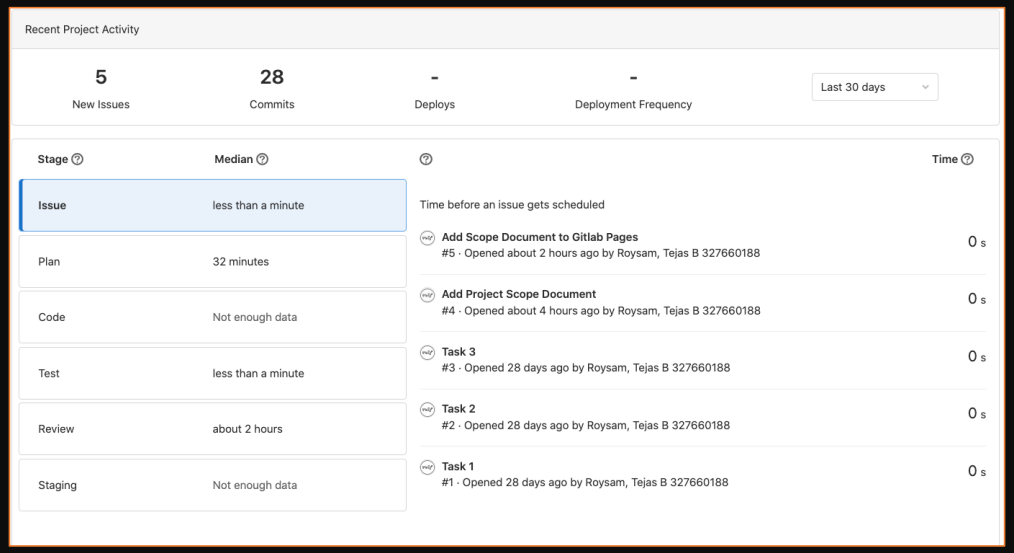




Gitlab Fundamentals
Project View

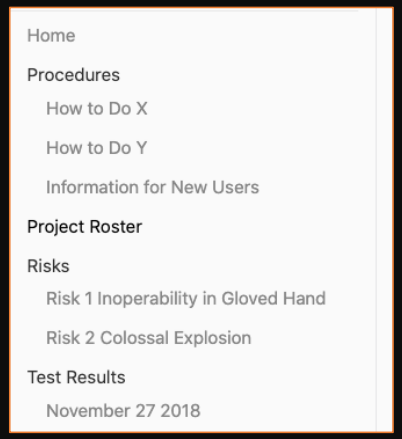
- Analytics
  - Viewport for project management:
    - Pass/fail rates of automated testing
    - Review quality + time spent peer reviewing + pace of work
    - Test coverage, contribution frequency of individuals
  - Provides value stream at-a-glance report
  - Broken down by workflow stage

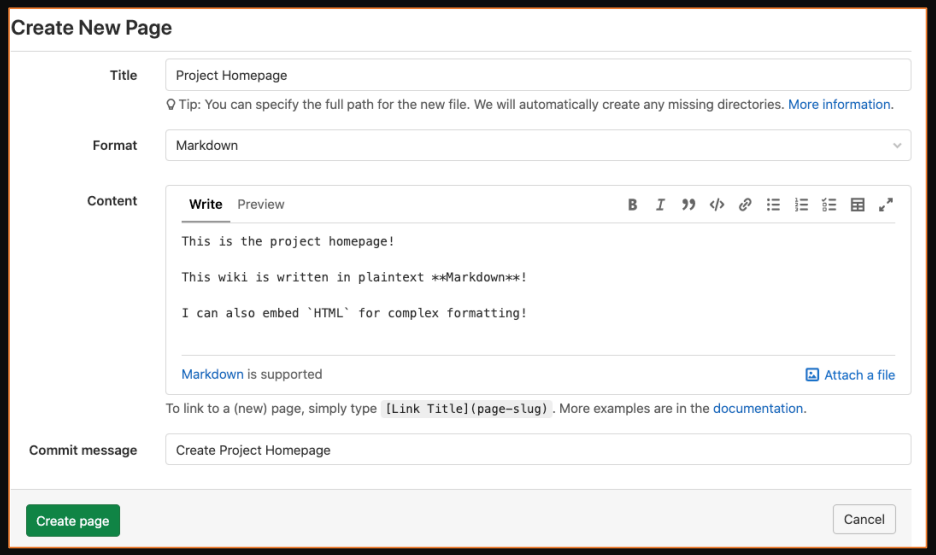




Gitlab Fundamentals
Project View

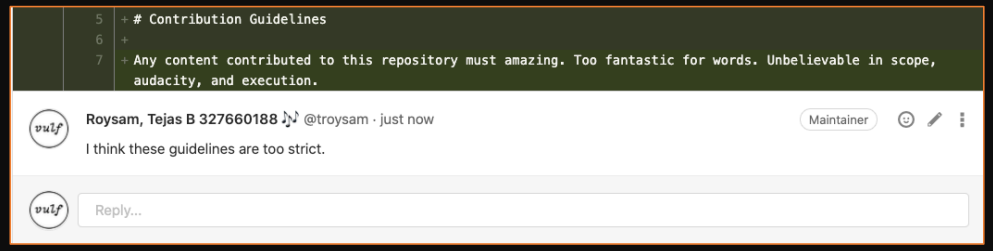
- Wikis
  - Structured like a repository
    - Maintains contribution history
    - Allows files/images to be uploaded
    - Can be hierarchically structured and/or notionally grouped
    - References to other articles, issues, merge requests, etc. (Similar to Wikia)
  - Uses Markdown for simple formatting, supports HTML for complex formatting

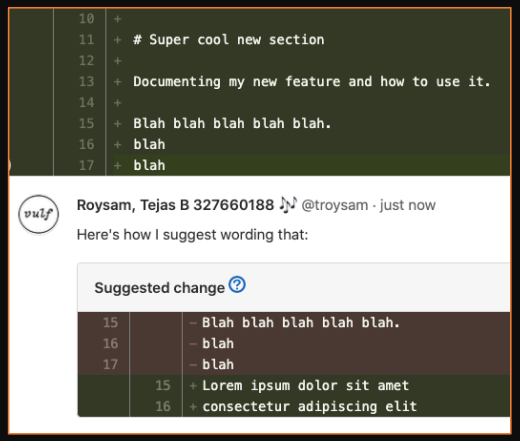




Gitlab Fundamentals Collaboration

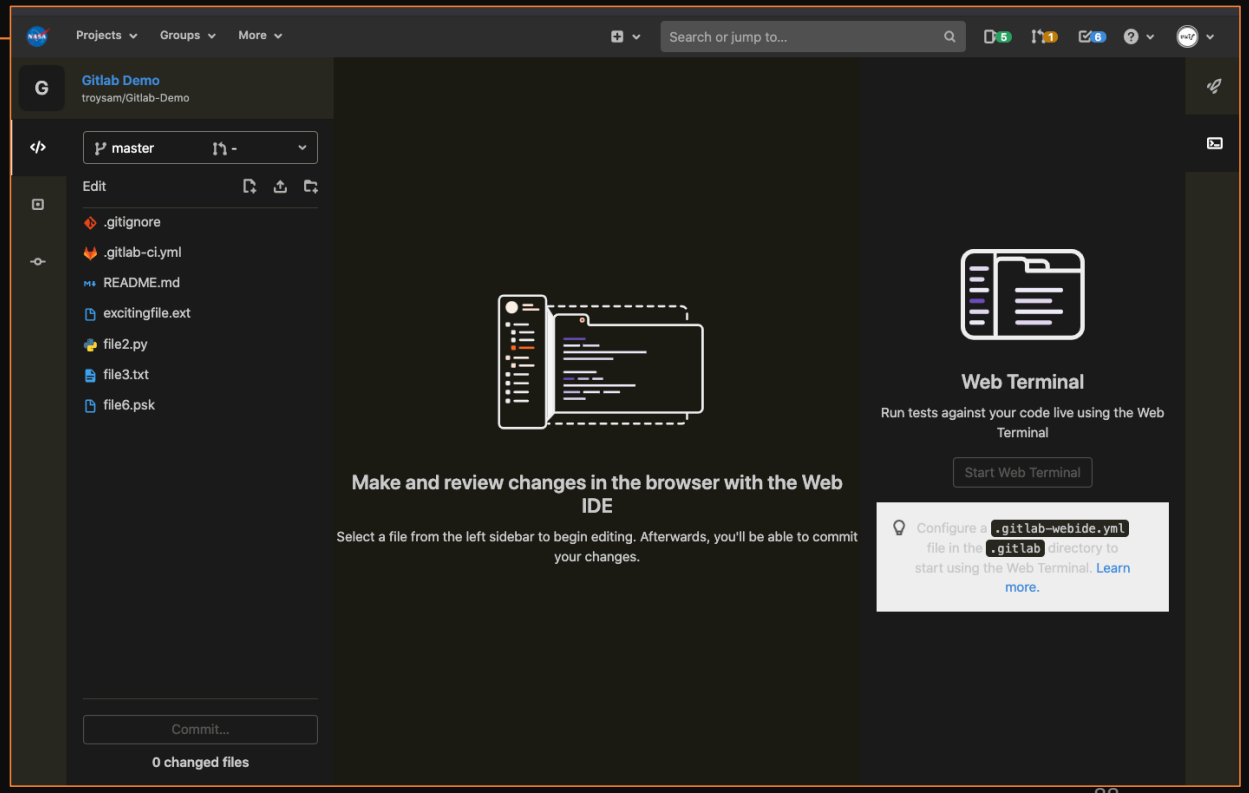
- Issues, commits, merge requests
  - Opportunity for comment or discussion everywhere
  - Offering suggestions/corrections is less “scary” (more accessible/more impersonal)
- Required reviewers
- Permission levels
- Service desk
- Static webpages and wikis
- Global search - underrated





Gitlab Fundamentals
Web IDE

- You can use Gitlab without any need for Git, and without any need for working locally
- Gitlab's web IDE lets you add, remove, upload, and modify files directly on your Gitlab instance
- Web IDE is also an interface for Git. You can:
  - Commit & Push
  - Change branches
  - Review diffs
  - Run pipelines
  - Use a "web terminal" to interact with your repository (sandbox)



Gitlab Fundamentals Integrations

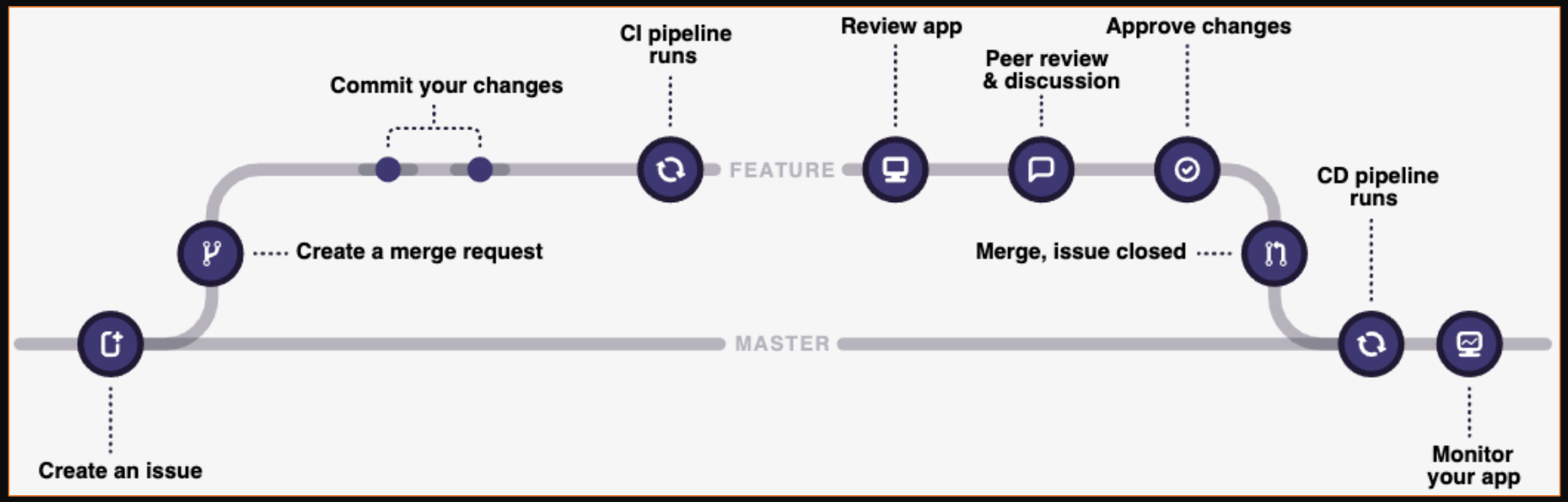
- Gitlab can integrate with other project management/configuration management tools such as:
  - Docker
  - Kubernetes
  - Bitbucket, LDAP, SAML authentication (e.g. NASA Launchpad)
  - Jenkins CI/Bamboo CI/Drone CI (alternatives to Gitlab's own Continuous Integration)
  - Github
  - Slack/Hangouts/Discord/Teams/Webex/Campfire/Email
  - Atlassian (Confluence/Jira/Crucible)
  - Trello/Redmine/Bugzilla
  - UML clients – generate diagrams

Gitlab Fundamentals
Expanded Capabilities

- We are focusing only on self-managed instances (rather than SaaS), since that suits NASA's security needs.
- We are also limiting ourselves to discussing the basic capabilities offered with the free, open source edition of Gitlab.
- There are paid editions which offer additional features like:
  - Requirements management
    - CI-automated requirements verification
    - Import/Export requirements
  - Dependency, Security, and Vulnerability management
  - Additional metrics and analytics
  - Etc.
- Full comparison: https://about.gitlab.com/pricing/self-managed/feature-comparison/

Gitlab Flow

- Unify the core features of Gitlab into a single workflow:



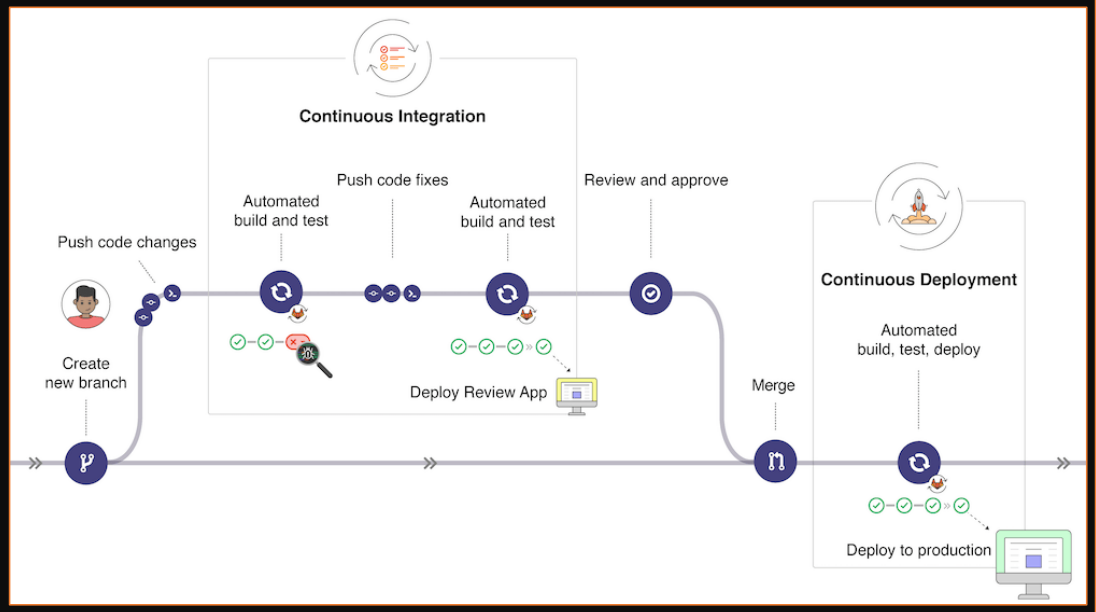
Gitlab Flow

Driving concepts of workflow:

- Changes to the repository -> tied to issue describing goal
- Incremental team review at merge requests -> call out specific people or groups for review
- Feature branches deleted after incorporation -> active feature branches = work in progress
- Commit and push frequently -> leverage Gitlab CI/CD, increase clarity of changes

Gitlab CI/CD Overview

- CI = Continuous Integration
  - Every incremental change triggers a full suite of independent builds, tests, and validations.
  - Run on every push, branch creation, merge event, etc. (also configurable).
  - Early error detection, limited integration issues
- CD = Continuous Delivery and Deployment
  - Automate the release process of products of the repository.
  - Deploy to servers, webpages, production environments, etc. (Customers)
  - Lower-risk releases & faster feedback



A simple example of Gitlab CI/CD

Image: Gitlab.com

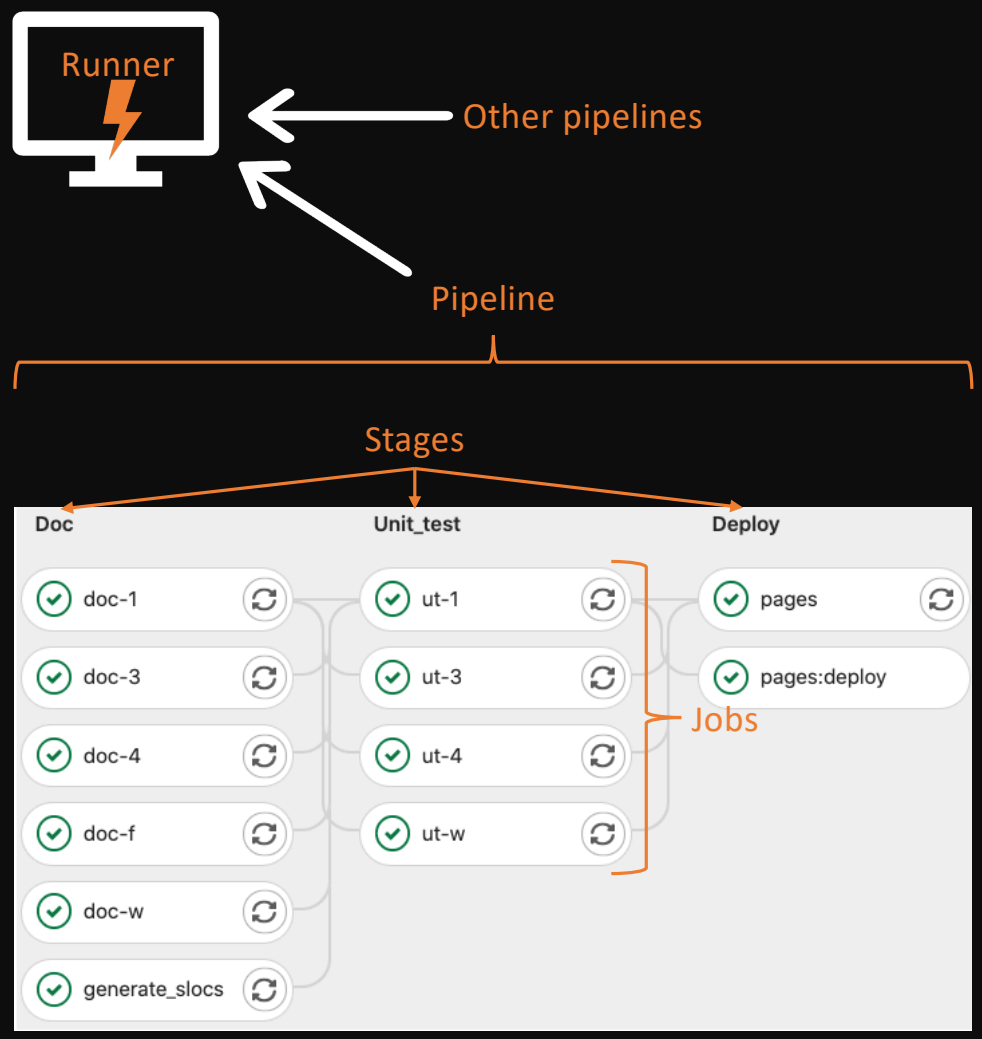
Gitlab CI/CD Components

Job: A job is a series of commands to be executed, as defined by a user.

Stage: A stage is a collection of jobs that are run in parallel (i.e. they have no dependencies on each other).

Pipeline: A pipeline is the collection of all of the jobs defined in a .gitlab-ci.yml file, organized chronologically by stage.

Runner: A runner is a computer that has been configured to accept jobs from one or more GitLab projects.



Gitlab CI/CD Containers

- Containerization
  - Create ”fresh copies” of managed virtual environments
  - ”Use and throw away”
  - Containers are created from a master “recipe” (image) which is pre-built.
- Containers allow tests to be
  - Isolated
  - Repeatable
  - Multiplatform
  - Fast
- For CI/CD we use containers:
  - As a basis for test jobs
  - As a sync point for teammates
  - To host (and gate access to) custom environments
    - Ex: Images with NASA/Federal CA certificates

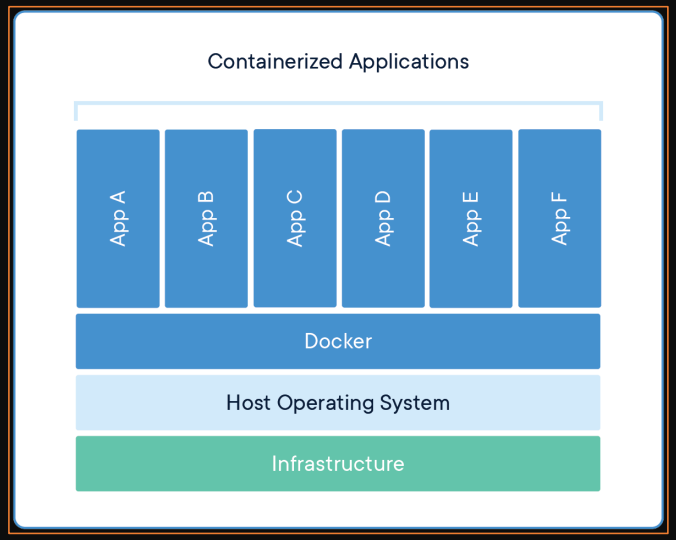
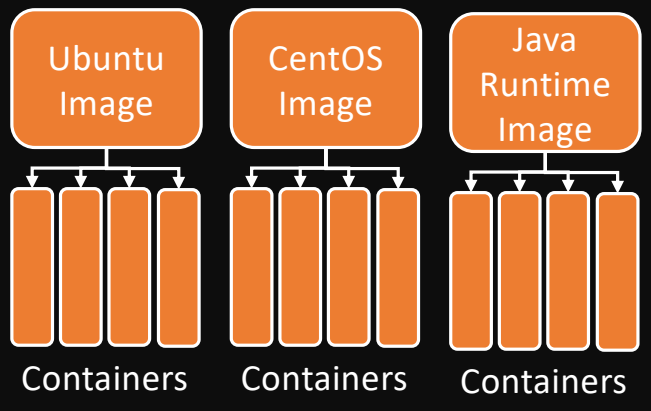


Image: Docker.com



Gitlab CI/CD
The .gitlab-ci.yml

- Aside from your Git repository, all you need to start using Continuous Integration is a gitlab-ci.yml.
- YAML (human-readable configuration file)
- Defines the entire structure and sequence of the pipeline, including:
  - Sources
  - Dependencies
  - Parallel and sequential tasks
- “Instruction set” for a Gitlab Runner to actually perform the jobs

```yaml
image: js-er-code.jsc.nasa.gov:5005/xemu/plss_cws_image_centos7_base:latest
stages:
  - generate-objects
  - static-analysis
  - document
  - build-code
  - test-code
  - deploy-webpage

before_script:
    echo "This before_script is run before every job (unless the job overwrites it)."

run_cpp_check:
    stage: static-analysis
    allow_failure: true
    variables:
        OUTPUT_FILE: "static-analysis-results.txt"
    script:
        - cppcheck --enable=all --force --error-exitcode=1 -regex '.*/.*\.\(c\|cpp\|h\)$'`. 2>> $OUTPUT_FILE; echo "\n" >> $OUTPUT_FILE
    after_script:
        # display results
        - cat $OUTPUT_FILE
    artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - $OUTPUT_FILE
    when: on_failure
    expire_in: 1 day
  tags:
    - docker
```

Gitlab CI/CD
.gitlab-ci.yml - General

- First segment (before first job) are global definitions
  - Configurations that apply to all jobs by default (but can be overridden)
  - Definition of stages
- There are LOTS of “attributes” for making your overall pipeline and individual jobs behave exactly how you desire them to.
  - We will outline the core attributes.
- Ex: Here we specify a global Docker `image` and a global `before_script` (a section of the script that is run at the start of every job). We also list the `stages` that comprise the pipeline.

```
image: my-gitlab-instance.jsc.nasa.gov:5005/my-group/my-docker-image:latest
stages:
  - generate-objects
  - static-analysis
  - document
  - build-code
  - test-code
  - deploy-webpage

before_script:
    - echo "This before_script is run before every job (unless the job overwrites it)."
```

Gitlab CI/CD
.gitlab-ci.yml - Jobs

Jobs are defined with unique names, ideally descriptive, and can have many attributes.

```yaml
run_cpp_check:
  stage: static-analysis
  allow_failure: true
  variables:
      OUTPUT_FILE: "static-analysis-results.txt"
  script:
      - cppcheck --enable=all --force --error-exitcode=1 -regex '.*/.*\.\(c\|cpp\|h\)$'`. 2>> $OUTPUT_FILE; echo "\n" >> $OUTPUT_FILE
  after_script:
      # display results
      - cat $OUTPUT_FILE
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - $OUTPUT_FILE
    when: on_failure
    expire_in: 1 day
  tags:
    - my-project-runner
```

`stage` - define the pipeline stage in which the job is executed
`allow_failure` - permit the pipeline to pass even if it encounters a failure condition
`before_script` - a set of commands that are executed before the job script
`script` - the shell script that is executed by the Gitlab Runner
`after_script` - a set of commands that are executed after the job script
`tags` - criterion determining which runners should execute this job

Gitlab CI/CD
.gitlab-ci.yml - Artifacts

Artifacts are files and directories from a job that are saved for use by future jobs, and/or for human inspection.

```yaml
run_cpp_check:
  stage: static-analysis
  allow_failure: true
  variables:
      OUTPUT_FILE: "static-analysis-results.txt"
  script:
      - cppcheck --enable=all --force --error-exitcode=1 -regex '.*/.*\.\(c\|cpp\|h\)$'`. 2>> $OUTPUT_FILE; echo "\n" >> $OUTPUT_FILE
  after_script:
      # display results
      - cat $OUTPUT_FILE
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - $OUTPUT_FILE
    when: on_failure
    expire_in: 1 day
  tags:
    - my-project-runner
```

`artifacts` - can either provide a list of files/paths, or a fully descriptive section with attributes like:
`artifacts:name` - identifier for the artifact archive for this job
`artifacts:paths` - list of files/paths to artifact
`artifacts:when` - when to store artifacts, could be `on_success`, `on_failure`, or `always`
`artifacts:expire_in` - tell Gitlab how long to preserve the artifacts for human inspection
`artifacts:reports` - collect and generate reports as applicable (metrics, performance, licensing)

Gitlab CI/CD
.gitlab-ci.yml - Variables

- Gitlab's CI process provides a predefined set of variables that can help organize + augment your CI
- Ex:
  - `CI_COMMIT_SHA`
  - `CI_COMMIT_BRANCH`
  - `CI_JOB_STATUS`
  - `CI_JOB_ID`
  - …
- You can also specify `variables` on a job level and globally.

```
run_cpp_check:
  stage: static-analysis
  allow_failure: true
  variables:
      OUTPUT_FILE: "static-analysis-results.txt"
  script:
      - cppcheck --enable=all --force --error-exitcode=1 -regex '.*/.*\.\(c\|cpp\|h\)$'`. 2>> $OUTPUT_FILE; echo "\n" >> $OUTPUT_FILE
  after_script:
      # display results
      - cat $OUTPUT_FILE
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - $OUTPUT_FILE
    when: on_failure
    expire_in: 1 day
  tags:
    - my-project-runner

generate-doxygen:
  image: my-gitlab-instance.jsc.nasa.gov:5005/my-group/my-alternative-docker-image:latest
  stage: document
  script:
    - ./generate-doxygen.sh
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - doc/release/
    expire_in: 1 day
  tags:
    - docker
```

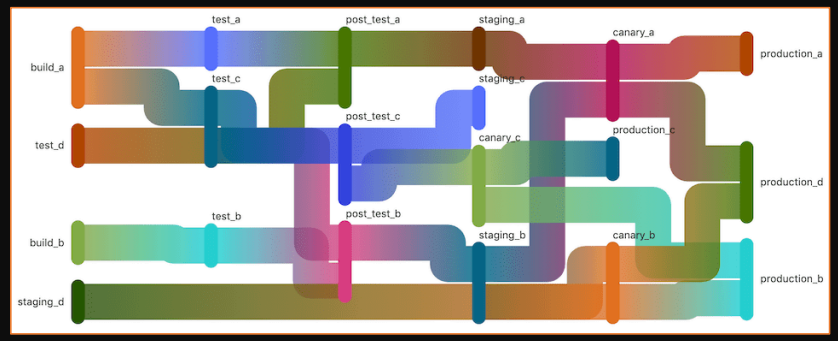
https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

Gitlab CI/CD Ordering

- Jobs can execute concurrently, as soon as a runner is available.
- Sequential jobs are a result of dependency chains
- Jobs define what prior stage jobs they need artifacts from using the `dependencies` attribute
- Alternatively, a list of jobs can be provided using the `needs` attribute. This ignores stage order and runs jobs in whatever order `needs` dictates. Gitlab will generate a directed acyclic graph (DAG) to visualize this.

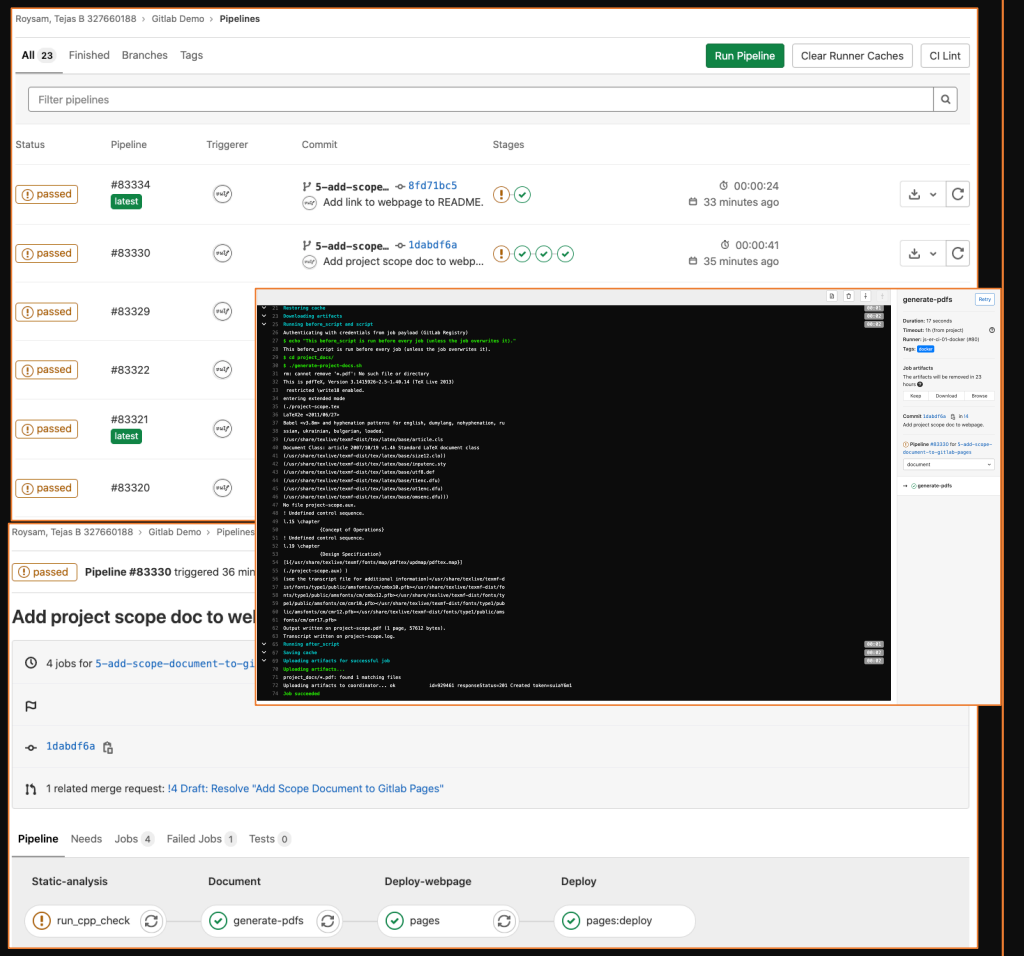
```yaml
generate-doxygen:
  image: my-gitlab-instance.jsc.nasa.gov:5005/my-group/my-alternative-docker-image:latest
  stage: document
  script:
    - ./generate-doxygen.sh
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - doc/release/
    expire_in: 1 day
  tags:
    - docker

 generate-pdfs:
  stage: document
  script:
    - ./generate-pdfs.sh
  dependencies:
    - generate-doxygen
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - doc/release/*.pdf
    expire_in: 1 day
  tags:
    - docker
```



Gitlab CI/CD Pipeline Results

- Gitlab CI maintains a history of pipeline results
- Each pipeline records the terminal output, artifacting, and pass/fail status for each job
- Each pipeline also contains information about the Git revision, related open events, and a connected graph of jobs run and their results.
- The overall pipeline pass/fail will reflect whether any jobs failed or were allowed to fail
- Pipelines can also be manually run and/or scheduled at specific times
- Pipelines can be multi-project
  - ”Component pipelines”
  - Auto-generated graphs of cross-project pipeline flow



Gitlab CI/CD
Gitlab Pages

- Gitlab Pages allows you to create and publish static webpages directly from a repository.
  - Static = HTML, CSS, Javascript (No server-side processing)
- Gitlab Pages are built and published as part of the CI/CD process
- Specifying a job named `pages` will indicate to Gitlab that you are deploying a Pages static webpage.
  - Place all static webpage content into a public/ folder, with at minimum a homepage (index.html)
- In the job, we can also specify rules so that the Pages webpage is only updated at useful times.

```yaml
pages:
  stage: deploy-webpage
  script:
    #Move into the Gitlab-created public folder
    - mv html/ public/
  needs:
    - generate-doxygen
    - generate-pdfs
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'
    - if: '$CI_COMMIT_BRANCH == "development"'
    - if: '$DEPLOY'
```

Live Demo

References

- Gitlab documentation is excellent and intuitive:
  - User documentation: https://docs.gitlab.com/ee/user/index.html
  - CI/CD documentation: https://docs.gitlab.com/ee/ci/README.html
- An explanation of Gitlab flow: https://docs.gitlab.com/ee/topics/gitlab_flow.html

- Example public projects on Gitlab SaaS: https://gitlab.com/explore/projects/trending