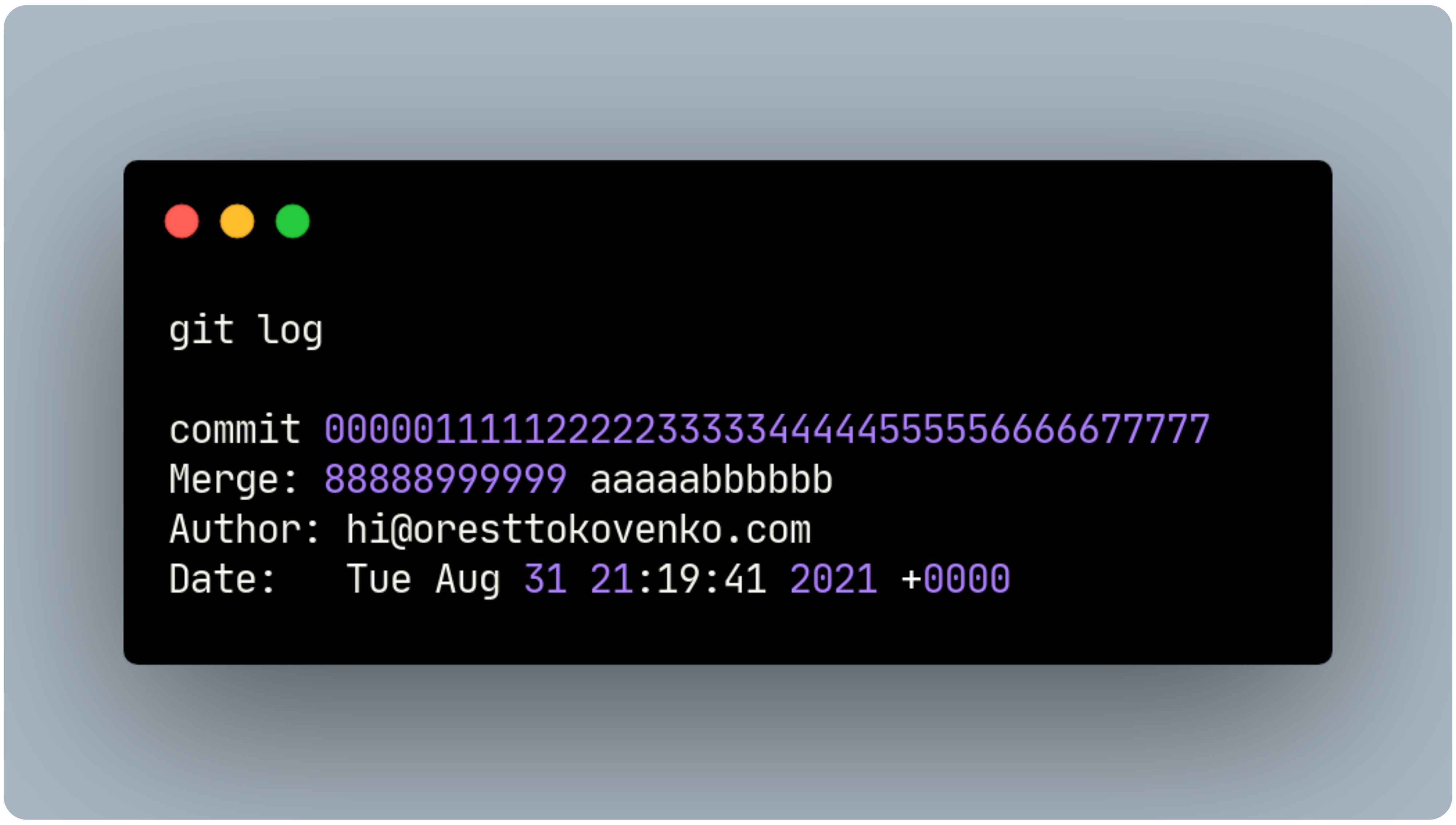
Git Gitlab Study Guide

Study Guide for GitLab Certified Associate Certification

Orest Tokovenko – 30 December 2021



```
git log

commit 0000011111222223333344444555556666677777
Merge: 88888999999 aaaaabbbbbb
Author: hi@oresttokovenko.com
Date:   Tue Aug 31 21:19:41 2021 +0000
```

This study guide covers the GitLab Certified Associate Certification. It is a technical certification offered by GitLab Professional Services to help validate individuals' ability to apply GitLab and Git in Version Control. To earn this certification, candidates must first pass a 1-hour written assessment, followed by a 2-hour hands-on lab assessment graded by GitLab Professional Services engineers.

There are two parts, a multiple-choice selection on a set of 14 questions, with 80% accuracy required to move on to the next stage that can be retaken indefinitely, and a hands-on set of 14 tasks which must be completed on a project you create.

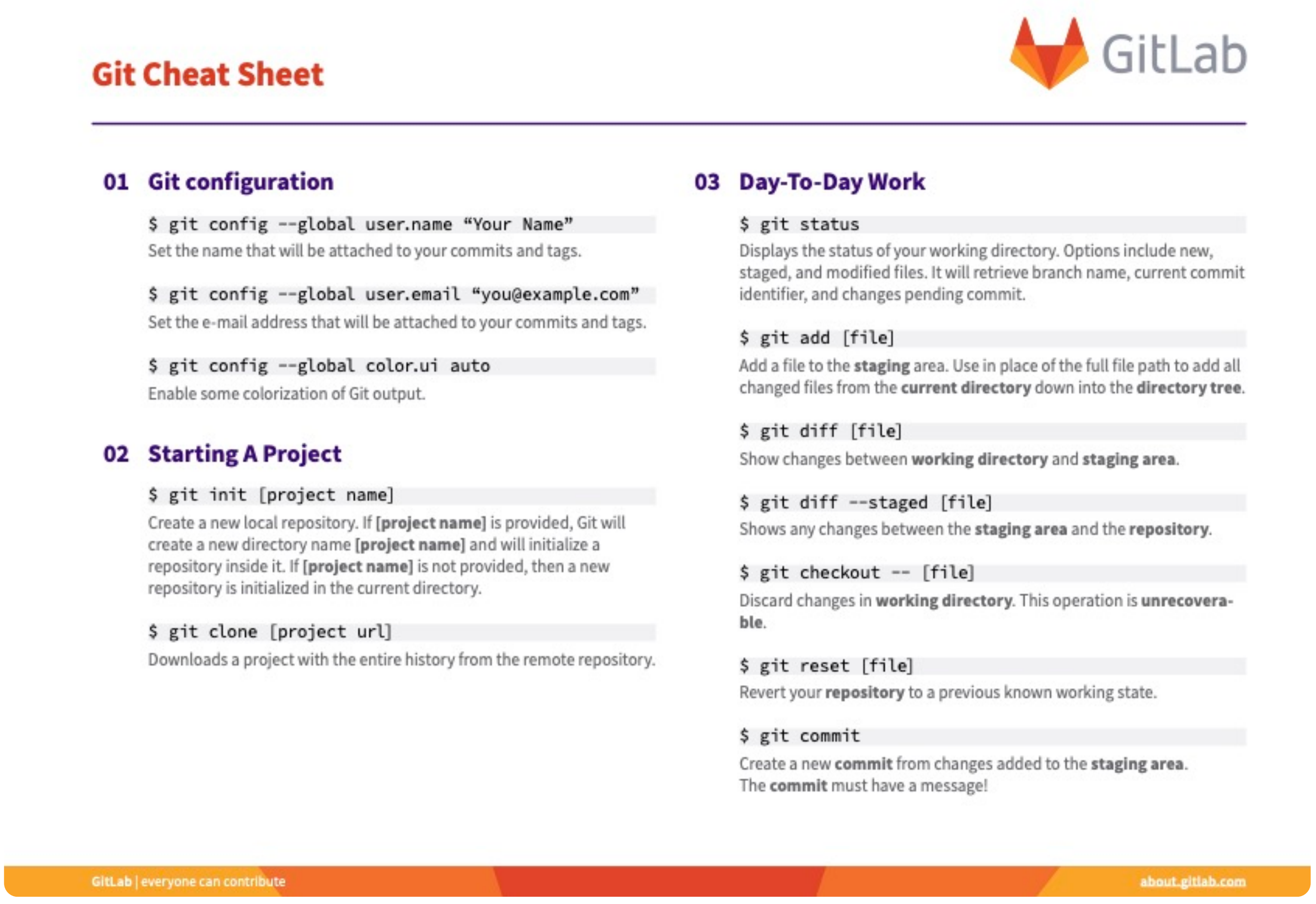
The exam is generally things about GitLab which you've covered in the course, but included two questions about using Git that were not covered in any of the modules. For this reason, I'd suggest when you get to those questions, open a git environment, and try each of the commands offered given the specific scenario.

Study Guide

Git Basics

- **Git Term Definitions**
  - **Branch**
    - A branch is an independent line of development
  - **Tag**
    - Mark a specific point in time on a branch.
  - **Checkout**
    - Get a specific branch to start making your changes.
  - **Commit**
    - Add changes you've made to the repository.
  - **Push**
    - Send changes to a remote directory.
  - **Workspace**
    - Directory where you store the repository on your computer.
  - **Untracked Files**
    - New files that Git has not been told to track previously
  - **Working Area**
    - Files that have been modified but not committed
  - **Staging Area**
    - Modified/Added files that are marked to go into the next commit
  - **Local Repo**
    - Local copy of the upstream repo
  - **Remote/Upstream Repo**
    - Hosted repository on a shared server (GitLab)
- **Centralized vs. Distributed**
  - **It is safer for data loss**:
    - Each developer has a clone from the repo
    - Even a CI server has a trustable source of backup
  - **It is faster**:

        - Most of the operations doesn't require network
        - It's less bureaucratic- you can create branches, commits, and merges without accessing the remote repository
    - **More collaborative**:
        - You can have several remote repositories
        - Allows you collaborate with different developers without changing the official repository (using forks for example)
- **GitLab Cheat Sheets**



Git Cheat Sheet

GitLab

01 Git configuration

```
$ git config --global user.name "Your Name"
```
Set the name that will be attached to your commits and tags.

```
$ git config --global user.email "you@example.com"
```
Set the e-mail address that will be attached to your commits and tags.

```
$ git config --global color.ui auto
```
Enable some colorization of Git output.

02 Starting A Project

```
$ git init [project name]
```
Create a new local repository. If **[project name]** is provided, Git will create a new directory name **[project name]** and will initialize a repository inside it. If **[project name]** is not provided, then a new repository is initialized in the current directory.

```
$ git clone [project url]
```
Downloads a project with the entire history from the remote repository.

03 Day-To-Day Work

```
$ git status
```
Displays the status of your working directory. Options include new, staged, and modified files. It will retrieve branch name, current commit identifier, and changes pending commit.

```
$ git add [file]
```
Add a file to the **staging** area. Use in place of the full file path to add all changed files from the **current directory** down into the **directory tree**.

```
$ git diff [file]
```
Show changes between **working directory** and **staging area**.

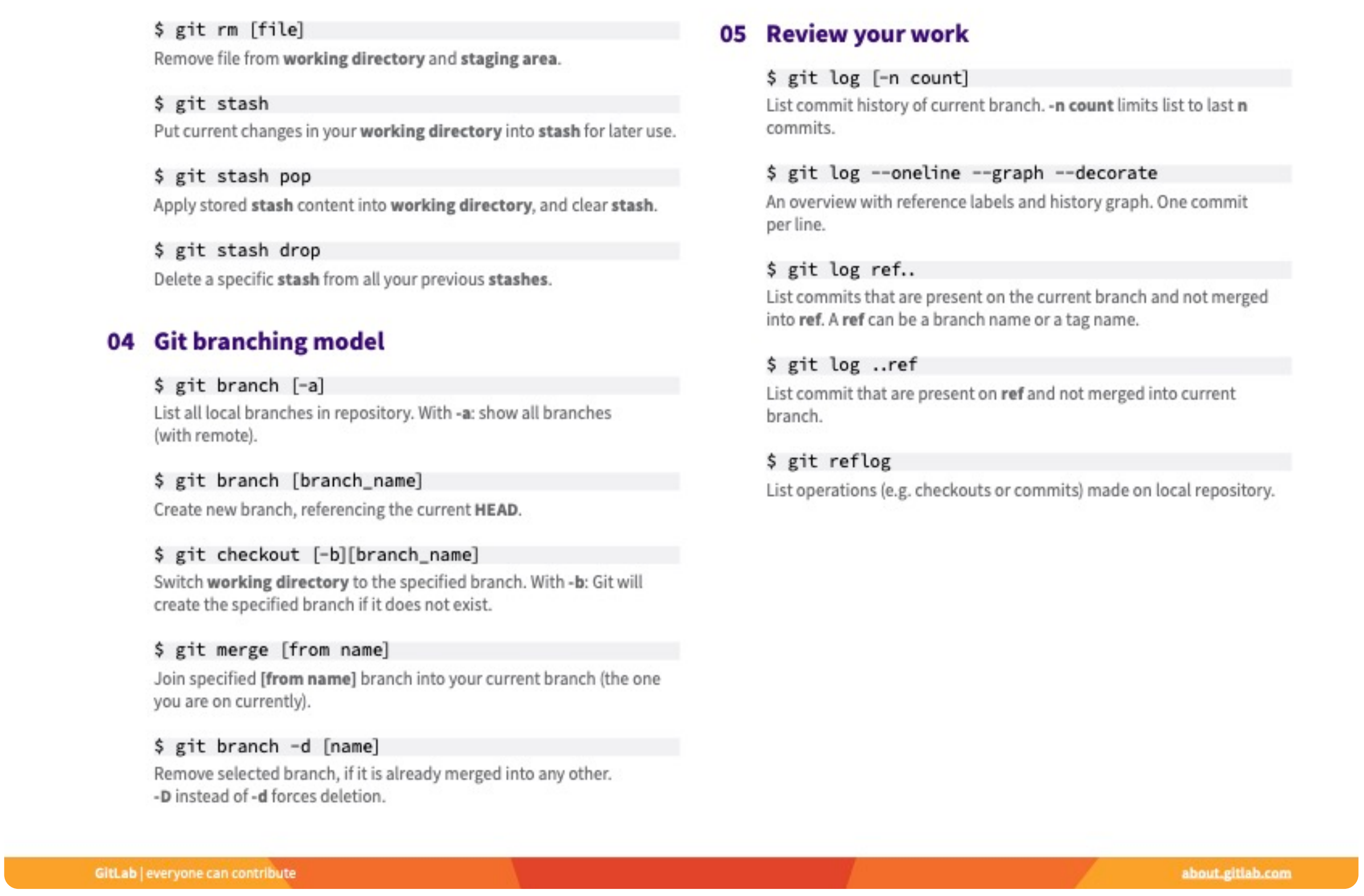
```
$ git diff --staged [file]
```
Shows any changes between the **staging area** and the **repository**.

```
$ git checkout -- [file]
```
Discard changes in **working directory**. This operation is **unrecovera-ble**.

```
$ git reset [file]
```
Revert your **repository** to a previous known working state.

```
$ git commit
```
Create a new **commit** from changes added to the **staging area**. The **commit** must have a message!

GitLab | everyone can contribute about.gitlab.com



```
$ git rm [file]
```
Remove file from **working directory** and **staging area**.

```
$ git stash
```
Put current changes in your **working directory** into **stash** for later use.

```
$ git stash pop
```
Apply stored **stash** content into **working directory**, and clear **stash**.

```
$ git stash drop
```
Delete a specific **stash** from all your previous **stashes**.

04 Git branching model

```
$ git branch [-a]
```
List all local branches in repository. With **-a**: show all branches (with remote).

```
$ git branch [branch_name]
```
Create new branch, referencing the current **HEAD**.

```
$ git checkout [-b][branch_name]
```
Switch **working directory** to the specified branch. With **-b**: Git will create the specified branch if it does not exist.

```
$ git merge [from name]
```
Join specified **[from name]** branch into your current branch (the one you are on currently).

```
$ git branch -d [name]
```
Remove selected branch, if it is already merged into any other. **-D** instead of **-d** forces deletion.

05 Review your work

```
$ git log [-n count]
```
List commit history of current branch. **-n count** limits list to last **n** commits.

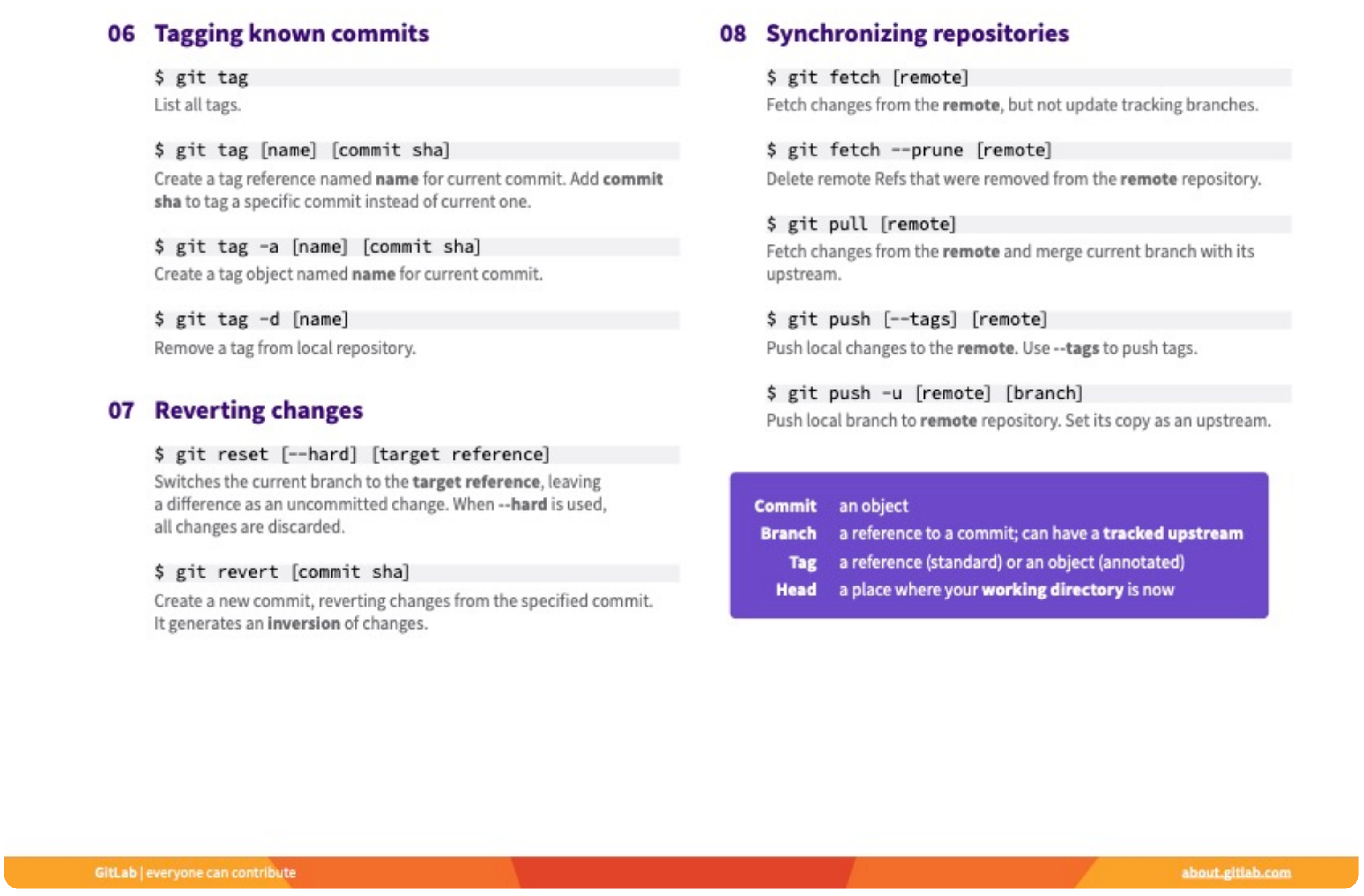
```
$ git log --oneline --graph --decorate
```
An overview with reference labels and history graph. One commit per line.

```
$ git log ref..
```
List commits that are present on the current branch and not merged into **ref**. A **ref** can be a branch name or a tag name.

```
$ git log ..ref
```
List commit that are present on **ref** and not merged into current branch.

```
$ git reflog
```
List operations (e.g. checkouts or commits) made on local repository.

GitLab | everyone can contribute about.gitlab.com



06 Tagging known commits

```
$ git tag
```

List all tags.

```
$ git tag [name] [commit sha]
```

Create a tag reference named **name** for current commit. Add **commit sha** to tag a specific commit instead of current one.

```
$ git tag -a [name] [commit sha]
```

Create a tag object named **name** for current commit.

```
$ git tag -d [name]
```

Remove a tag from local repository.

07 Reverting changes

```
$ git reset [--hard] [target reference]
```

Switches the current branch to the **target reference**, leaving a difference as an uncommitted change. When **--hard** is used, all changes are discarded.

```
$ git revert [commit sha]
```

Create a new commit, reverting changes from the specified commit. It generates an **inversion** of changes.

08 Synchronizing repositories

```
$ git fetch [remote]
```

Fetch changes from the **remote**, but not update tracking branches.

```
$ git fetch --prune [remote]
```

Delete remote Refs that were removed from the **remote** repository.

```
$ git pull [remote]
```

Fetch changes from the **remote** and merge current branch with its upstream.

```
$ git push [--tags] [remote]
```

Push local changes to the **remote**. Use **--tags** to push tags.

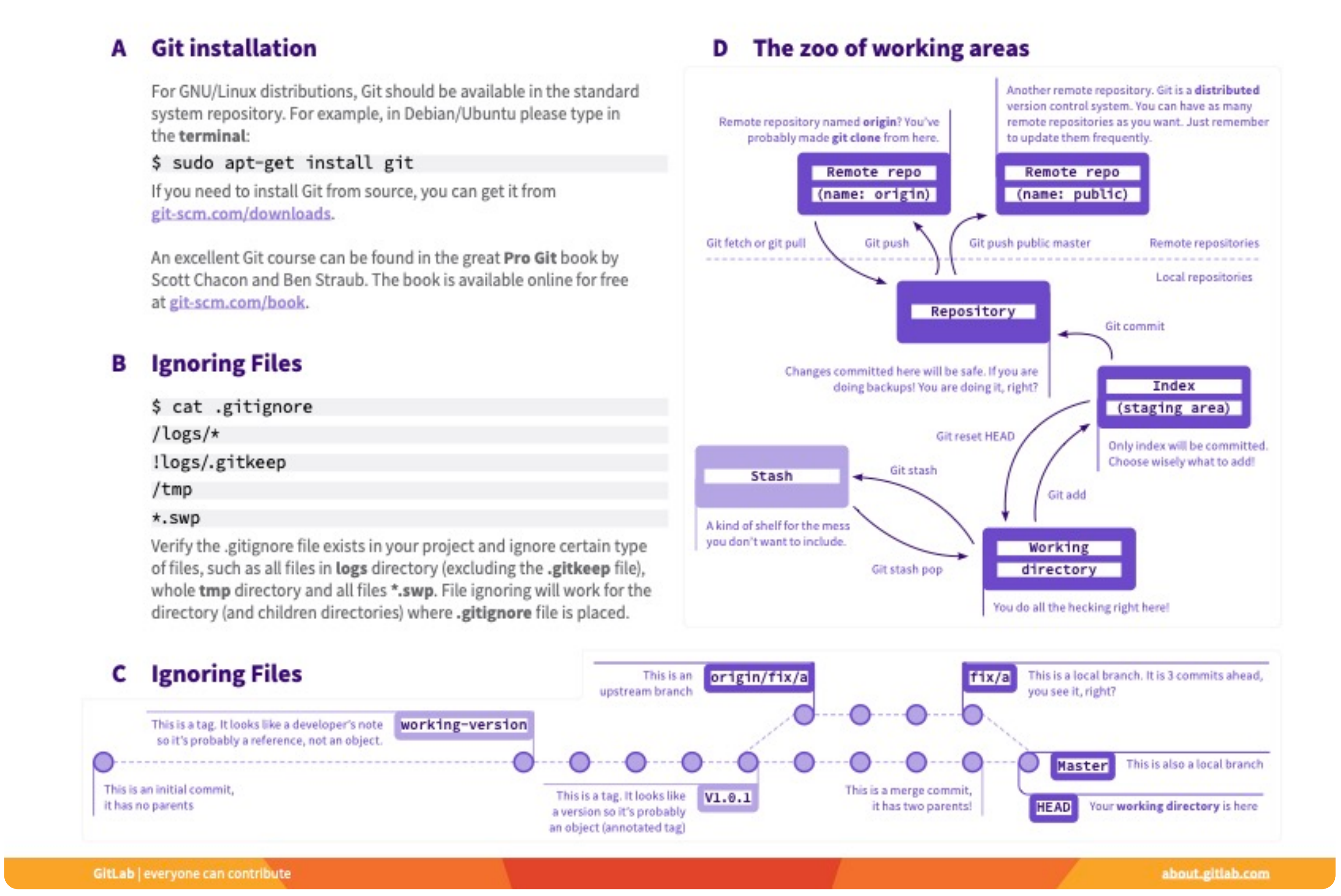
```
$ git push -u [remote] [branch]
```

Push local branch to **remote** repository. Set its copy as an upstream.

Commit	an object
Branch	a reference to a commit; can have a **tracked upstream**
Tag	a reference (standard) or an object (annotated)
Head	a place where your **working directory** is now

GitLab | everyone can contribute

about.gitlab.com



A Git installation

For GNU/Linux distributions, Git should be available in the standard system repository. For example, in Debian/Ubuntu please type in the **terminal**:

```
$ sudo apt-get install git
```

If you need to install Git from source, you can get it from git-scm.com/downloads.

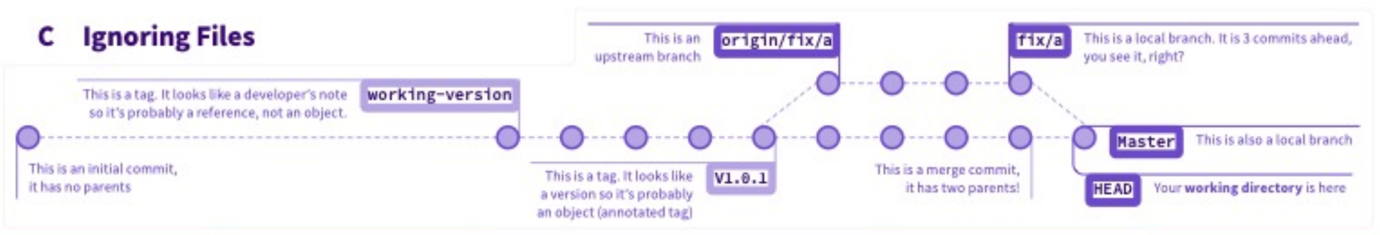
An excellent Git course can be found in the great **Pro Git** book by Scott Chacon and Ben Straub. The book is available online for free at git-scm.com/book.

B Ignoring Files

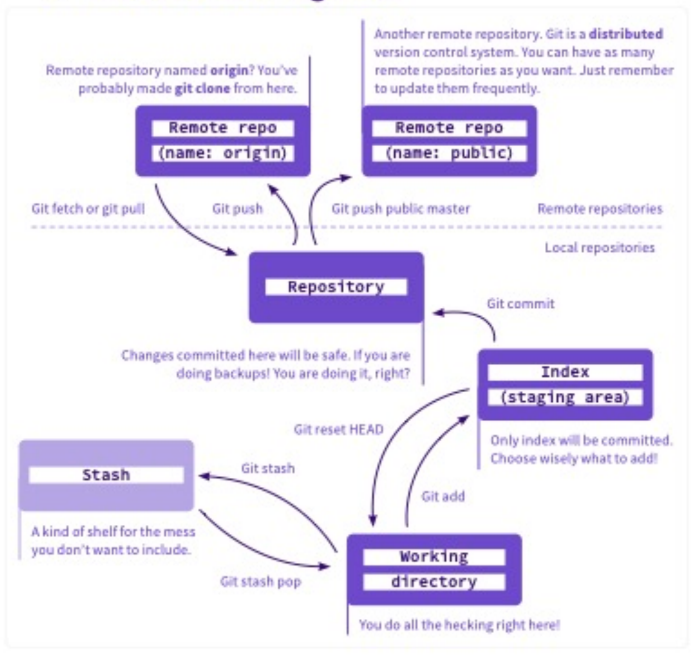
```
$ cat .gitignore
/logs/*
!logs/.gitkeep
/tmp
*.swp
```

Verify the .gitignore file exists in your project and ignore certain type of files, such as all files in **logs** directory (excluding the **.gitkeep** file), whole **tmp** directory and all files ***.swp**. File ignoring will work for the directory (and children directories) where **.gitignore** file is placed.

C Ignoring Files


This is an upstream branch
origin/fix/a
fix/a
This is a local branch. It is 3 commits ahead, you see it, right?
This is a tag. It looks like a developer's note so it's probably a reference, not an object.
working-version
Master
This is also a local branch
This is an initial commit, it has no parents
This is a tag. It looks like a version so it's probably an object (annotated tag)
V1.0.1
This is a merge commit, it has two parents!
HEAD
Your working directory is here

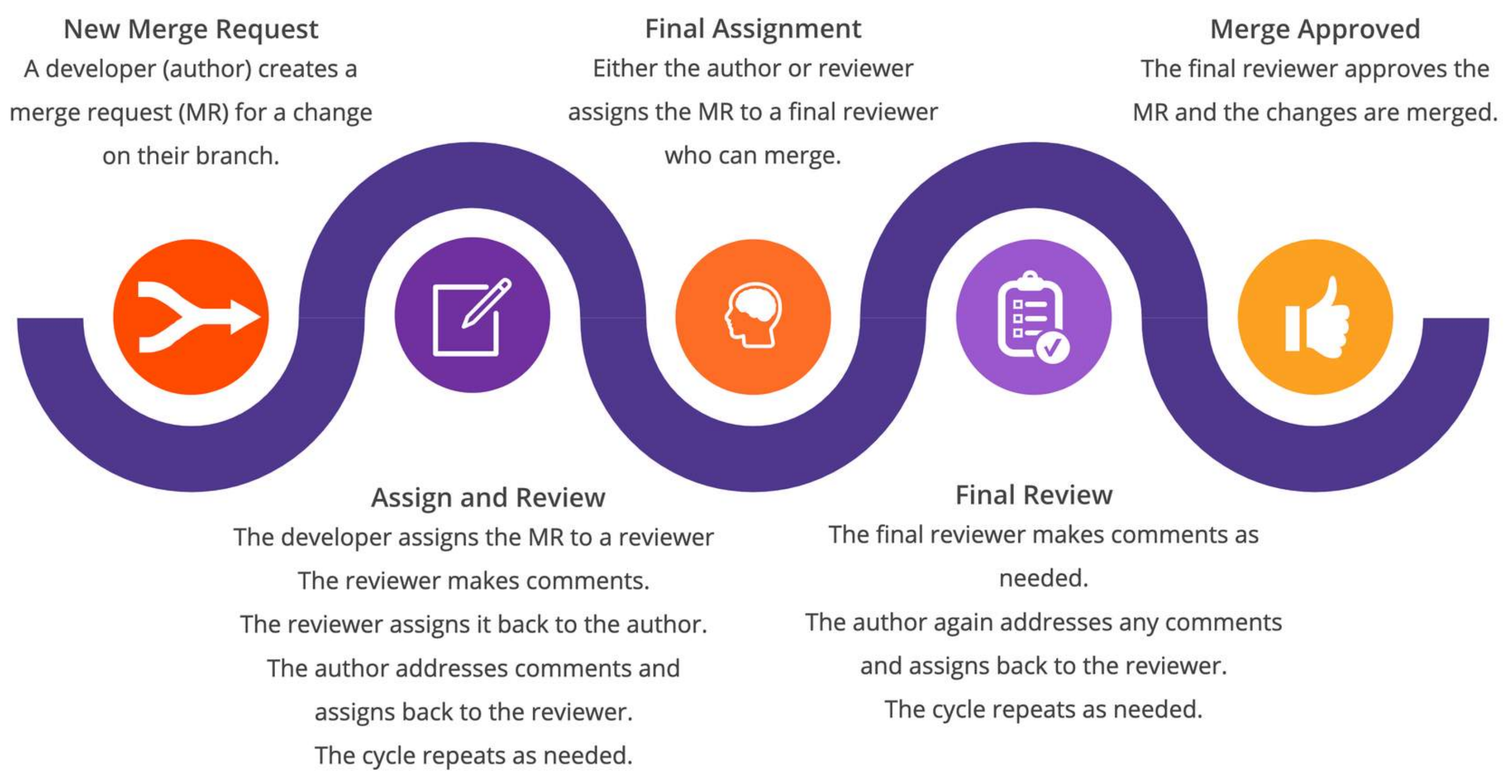
D The zoo of working areas


Remote repository named origin? You've probably made git clone from here.
Another remote repository. Git is a distributed version control system. You can have as many remote repositories as you want. Just remember to update them frequently.
Remote repo (name: origin)
Remote repo (name: public)
Git fetch or git pull
Git push
Git push public master
Remote repositories
Local repositories
Repository
Git commit
Changes committed here will be safe. If you are doing backups! You are doing it, right?
Index (staging area)
Only index will be committed. Choose wisely what to add!
Git reset HEAD
Stash
Git stash
Git add
A kind of shelf for the mess you don't want to include.
Git stash pop
Working directory
You do all the hecking right here!

GitLab | everyone can contribute

about.gitlab.com

- **Typical Workflow**


New Merge Request
A developer (author) creates a merge request (MR) for a change on their branch.
Final Assignment
Either the author or reviewer assigns the MR to a final reviewer who can merge.
Merge Approved
The final reviewer approves the MR and the changes are merged.
Assign and Review
The developer assigns the MR to a reviewer
The reviewer makes comments.
The reviewer assigns it back to the author.
The author addresses comments and assigns back to the reviewer.
The cycle repeats as needed.
Final Review
The final reviewer makes comments as needed.
The author again addresses any comments and assigns back to the reviewer.
The cycle repeats as needed.

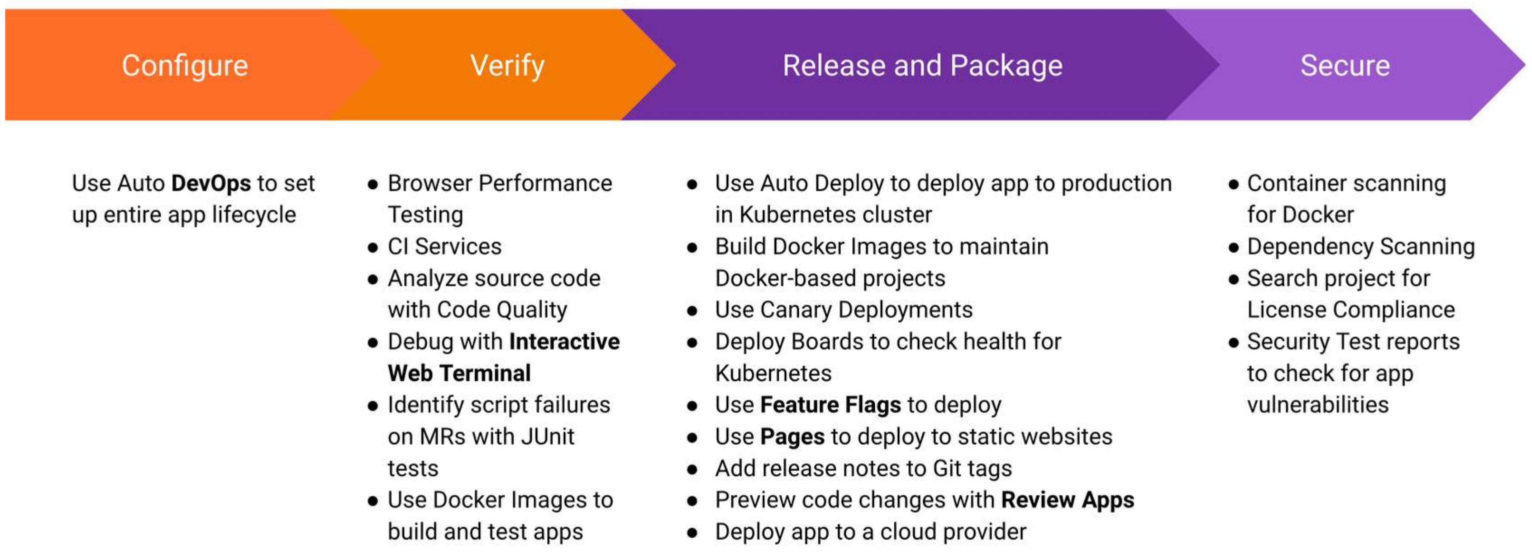
- **Code Review Workflow- GitLab Tools to Use**
  - **New Merge Request**
    - Edit files inline on branch
    - Commit Changes
      - Squash locally
      - Or one commit per file in GUI
    - Merge Request
    - Resolve merge conflicts
  - **Assign & Review**
    - Assignments
    - Changes tab to view differences
    - Edit Inline
    - Make comments inline
    - Discussion tab to view comments
  - **Final Assignment**
    - Assignments feature in GitLab
  - **Final Review**
    - Assignments
    - Changes tab to view differences
    - Edit Inline
    - Make comments inline
    - Discussion tab to view comments
  - **Merged Approved**
    - Approval feature in GitLab.
  - **Additional Tools for Code Review**
    - Wiki
    - Web IDE

        - Snippets
- **Additional Tools Used**
    - **Snippets**
        - With GitLab Snippets you can store and share bits of code and text with other users.
    - **Wiki**
        - A separate system for documentation called Wiki, is built right into each GitLab project. It is enabled by default on all new projects and you can find it under Wiki in your project.
        - Wikis are very convenient if you don’t want to keep your documentation in your repository, but you do want to keep it in the same project where your code resides.
        - You can create Wiki pages in the web interface or locally using Git since every Wiki is a separate Git repository.
    - **Web IDE**
        - The Web IDE editor makes it faster and easier to contribute changes to your projects by providing an advanced editor with commit staging.
- **Defining CI/CD**
    - Continuous Integration is the practice of merging all the code that is being produced by developers. The merging usually takes place several times a day in a shared repository. From within the repository, or production environment, building and automated testing are carried out that ensure no integration issues and the early identification of any problems.
    - Continuous Delivery adds that the software can be released to production at any time, often by automatically pushing changes to a staging system.
    - Continuous Deployment goes further and pushes changes to production automatically.
    - Together, CI and CD act to accelerate how quickly your team can deliver results for your customers and stakeholders. CI helps you catch and reduce bugs early in the development cycle, and CD moves verified code to your applications faster.
- **Benefits of GitLab CI/CD**
    - **Error Detection**
        - Detects errors as quickly as possible: fix problems while they are still fresh in developers mind.
    - **Increased Efficiency**
        - Reduces integration problems: smaller problems are easier to digest and fix immediately. Bugs won’t shut your whole system down.
    - **No Snowball Effect**
        - Avoid compounding problems: allows teams to develop faster, with more confidence and collaboration.
    - **Release Stages**
        - Ensures every change is releasable: test everything, including deployment,

before calling it done with less risk on each release.

- **Valuable Delivery**
  - Delivers value more frequently: reliable deployments mean more releases
- **Better Feedback Processes**
  - Tight customer feedback loops: fast and frequent customer feedback on changes allow for continuous improvement for your product.

- **CI/CD Features in GitLab- by Lifecycle Stage**



Configure

Use Auto **DevOps** to set up entire app lifecycle

Verify

- Browser Performance Testing
- CI Services
- Analyze source code with Code Quality
- Debug with **Interactive Web Terminal**
- Identify script failures on MRs with JUnit tests
- Use Docker Images to build and test apps

Release and Package

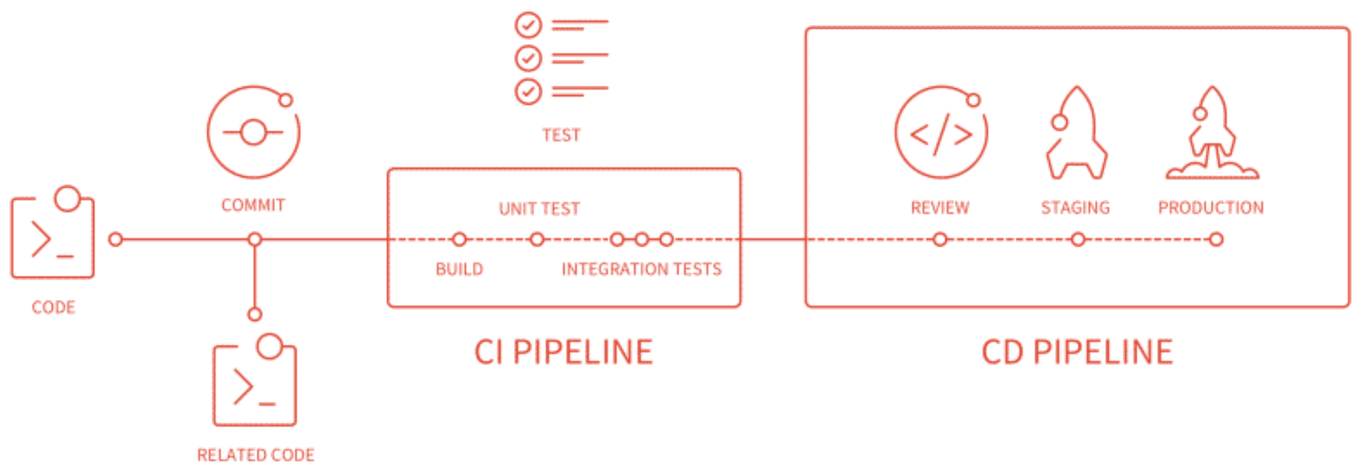
- Use Auto Deploy to deploy app to production in Kubernetes cluster
- Build Docker Images to maintain Docker-based projects
- Use Canary Deployments
- Deploy Boards to check health for Kubernetes
- Use **Feature Flags** to deploy
- Use **Pages** to deploy to static websites
- Add release notes to Git tags
- Preview code changes with **Review Apps**
- Deploy app to a cloud provider

Secure

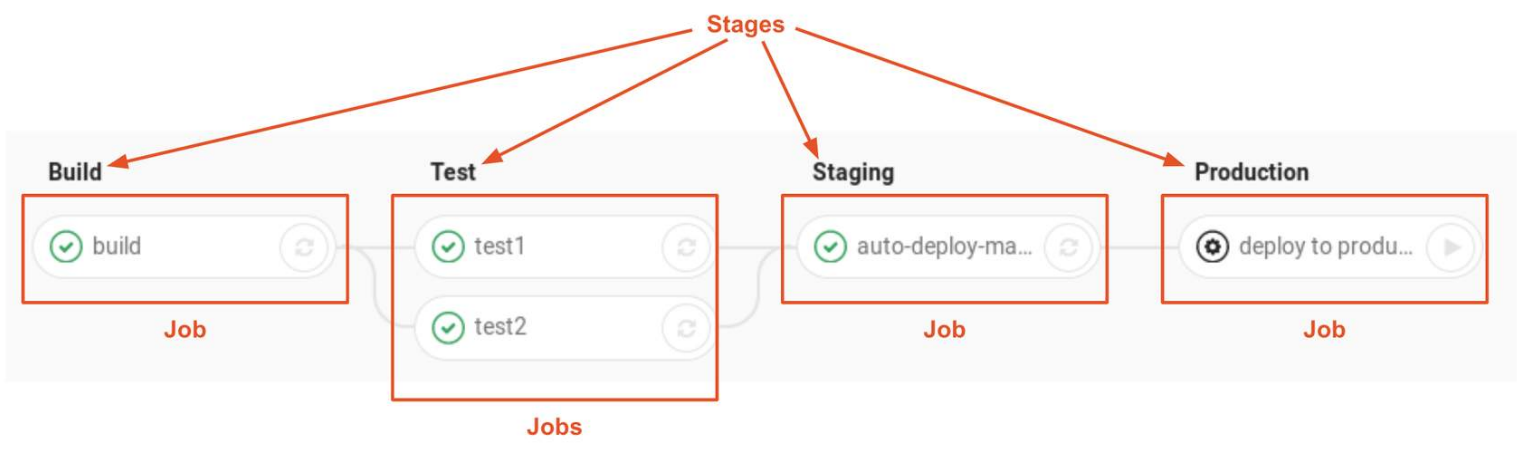
- Container scanning for Docker
- Dependency Scanning
- Search project for License Compliance
- Security Test reports to check for app vulnerabilities

- To use GitLab CI/CD you or your GitLab administrator must first define a pipeline within a YAML file called .gitlab-ci.yml and then install and configure a Gitlab Runner.
  - The YAML file is the pipeline definition file. It specified the stages, jobs, and actions that you want to perform. Think of the YAML file as the brains, and the runner as the body.
  - A GitLab Runner, a file written in Go, will run the jobs specified in the YAML file using an API to communicate with GitLab. Your GitLab administrator can configure shared runners to run on multiple projects, and you can set up your own by project.

- **How GitLab CI/CD Works**



- **Anatomy of a CI/CD Pipeline**



- **Anatomy of a CI/CD Pipeline**
  - **GitLab's Package Stage**
    - GitLab enables teams to package their applications and dependencies, manage containers, and build artifacts with ease. The private, secure container registry and artifact repositories are built-in and preconfigured out-of-the box to work seamlessly with GitLab source code management and CI/CD pipelines. Ensure DevOps acceleration with automated software pipelines that flow freely without interruption.
  - **Package Registry**
    - Every team needs a place to store their packages and dependencies. GitLab aims to provide a comprehensive solution, integrated into our single application, that supports package management for all commonly used languages and binary formats.
  - **Container Registry**
    - A container registry* is a secure and private registry for Docker images built-in to GitLab. Creating, pushing, and retrieving images works out of the box with GitLab CI/CD.
  - **Helm Chart Registry**
    - Kubernetes cluster integrations can take advantage of Helm charts to standardize their distribution and install processes. Supporting a built-in helm chart registry allows for better, self-managed container orchestration.
  - **Dependency Proxy**
    - The GitLab dependency proxy* can serve as an intermediary between your local developers and automation and the world of packages that need to be fetched from remote repositories. By adding a security and validation layer to a caching proxy, you can ensure reliability, accuracy, and audit-ability for the packages you depend on.
  - **Jupyter Notebooks**
    - Jupyter Notebooks are a common type of code used for data-science use cases. With GitLab you can store and version control those notebooks in the same way you store packages and application code.
  - **Git LFS**
    - Git LFS (Large File Storage) is a Git extension, which reduces the impact of large files in your repository by downloading the relevant versions of them lazily. Specifically, large files are downloaded during the checkout process rather than during cloning or fetching.
- **GitLab's Release Stage**

- **Pages**
  - GitLab Pages is a feature that allows you to publish static websites directly from a repository in GitLab. You can use it either for personal or business websites, such as portfolios, documentation, manifestos, and business presentations. You can also attribute any license to your content.
- **Review Apps**
  - Automatic Live Preview
    - Code, commit, and preview your branch in a live environment. Review Apps automatically spin up dynamic environments for your merge requests.
  - One-click to Collaborate
    - Designers and product managers won't need to check out your branch and run it in a staging environment. Simply send the team a link and let them click around.
  - Fully-Integrated
    - With GitLab's code review, built-in CI/CD, and Review Apps, you can speed up your development process with one tool for coding, testing, and previewing your changes.
  - Deployment Flexibility
    - Deploy to Kubernetes, Heroku, FTP, and more. You can deploy anywhere that you can script with .gitlab-ci.yml and you have full control to deploy as many different kinds of review apps as your team needs.
- **Incremental Rollout**
  - When you have a new version of your app to deploy in production, you may want to use an incremental rollout to replace just a few pods with the latest code. This will allow you to first check how the app is behaving, and later manually increasing the rollout up to 100%.
- **Feature Flags**
  - Feature flags enable teams to achieve CD by letting them deploy dark features to production as smaller batches for controlled testing, separating feature delivery from customer launch, and removing risk from delivery.
- **Release Orchestration**
  - Management and orchestration of releases-as-code built on intelligent notifications, scheduling of delivery and shared resources, blackout periods, relationships, parallelization, and sequencing, as well as support for integrating manual processes and interventions.
- **Release Evidence**
  - Release Evidence includes features such as deploy-time security controls to ensure only trusted container images are deployed on Kubernetes Engine, and more broadly includes all the assurances and evidence collection that are necessary for you to trust the changes you're delivering.
- **Secrets Management**
  - Vault is a secrets management application offered by HashiCorp. It allows you to store and manage sensitive information such as secret environment variables, encryption keys, and authentication tokens. Vault offers Identity-based Access, which means Vault users can authenticate through several of their preferred cloud providers.

- **GitLab Security Scanning**
  - Integrating a security scanner into GitLab consists of providing end users with a CI job definition they can add to their CI configuration files to scan their GitLab projects. This CI job should then output its results in a GitLab-specified format. These results are then automatically presented in various places in GitLab, such as the Pipeline view, Merge Request widget, and Security Dashboard.
  - Static Application Security Testing (SAST)
    - If you’re using GitLab CI/CD, you can use Static Application Security Testing (SAST) to check your source code for known vulnerabilities. If the pipeline is associated with a merge request, the SAST analysis is compared with the results of the target branch’s analysis (if available). The results of that comparison are shown in the merge request. If the pipeline is running from the default branch, the results of the SAST analysis are available in the security dashboards.
    - The results are sorted by the priority of the vulnerability:
      - Critical
      - High
      - Medium
      - Low
      - Info
      - Unknown
    - Use Cases:
      - Your code has a potentially dangerous attribute in a class, or unsafe code that can lead to unintended code execution.
      - Your application is vulnerable to cross-site scripting (XSS) attacks that can be leveraged to unauthorized access to session data.
    - SAST features covered under GitLab Free
      - Configure SAST Scanners
      - Customize SAST Settings
      - View JSON Report
    - SAST supports the following analyzers
      - bandit (Bandit)
      - brakeman (Brakeman)
      - eslint (ESLint (JavaScript and React))
      - flawfinder (Flawfinder)
      - gosec (Gosec)
      - kubesec (Kubesec)
      - mobsf (MobSF (beta))
      - nodejs-scan (NodeJsScan)
      - phpcs-security-audit (PHP CS security-audit)
      - pmd-apex (PMD (Apex only))

- security-code-scan (Security Code Scan (.NET))
- semgrep (Semgrep)
- sobelow (Sobelow (Elixir Phoenix))
- spotbugs (SpotBugs with the Find Sec Bugs plugin (Ant, Gradle and wrapper, Grails, Maven and wrapper, SBT))

Once you pass, you'll earn the beautiful badge below!

GitLab Certified Associate Badge

You may also like

See All Git

10 Jan 2022

Study Guide for AWS Data Analytics Specialty Certification

This study guide covers AWS Certification for Data Analytics Specialty. This ...

27 Dec 2021

Study Guide for DAG Authoring for Apache Airflow Certification

This study guide covers the Astronomer Certification DAG Authoring for Apache...

22 Dec 2021

Study Guide for Apache Airflow Fundamentals Certification

This study guide covers Astronomer Certification for Apache Airflow Fundament...

Twitter Github Instagram Linkedin