Git intro

Obolenskiy Arseniy, Nesterov Alexander

ITLab

November 15, 2024

Contents

1. What are version control systems?
2. What version control systems exist? History and evolution
3. What is Git?
4. Basic Git commands
5. Git workflows overview

Version control systems

A version control system (VCS) is a software tool that helps individuals and teams manage changes to source code, documents, and other collections of information over time.

- It is a persistent data structure (data structure that maintains its previous versions after modifications, allowing access to both current and past versions of the data)
- This is in contrast to ephemeral data structures, where changes overwrite the current state, losing prior versions

Early Days of Computing (1950s - 1970s). SCCS

Before the development of dedicated version control systems, programmers used manual processes to manage changes. Early systems, like the Source Code Control System (SCCS), developed by Marc Rochkind at Bell Labs in 1972, were among the first tools created to automate version control. SCCS stored multiple versions of code and helped manage modifications through change sets, allowing programmers to revert to earlier versions if necessary.

SCCS subcommands

admin -i file.f s.file.f - Put subs under SCCS control.
get s.file.f - Retrieve, read only.
get -e s.file.f - Retrieve, read/write (e = edit).
get -p s.file.f - Retrieve, just peak.
delta s.file.f - Store changes.
prs s.file.f - List revisions.

Source:
https://sites.science.oregonstate.edu/ landaur/nacphy/coping-with-unix/node169.html

Centralized Version Control Systems (1980s - 1990s)

Centralized version control systems (CVCS) came to prominence during the 1980s. These systems required a single, central server where all files and version histories were stored. Developers had to be connected to this server to commit changes or retrieve updates.

- RCS (Revision Control System): Developed by Walter Tichy in the early 1980s, RCS is a more advanced system than SCCS. It introduced features like automated version numbering and handling of concurrent edits, but it was still a single-user system.
- CVS (Concurrent Versions System): Introduced in 1990, CVS extended RCS with support for multiple developers working on the same project. It allowed distributed teams to collaborate more effectively, although merging changes was often difficult.
- SVN (Subversion): developed as an improvement over older systems like CVS. It was created by CollabNet in 2000 and later became an Apache project.

Distributed Version Control Systems (2000s)

The limitations of centralized systems (reliance on a single server), led to the development of distributed version control systems (DVCS) in the early 2000s. In DVCS, every developer has a complete copy of the project, including its entire history, on their local machine.

- BitKeeper (1998): BitKeeper was one of the first DVCS, created by Larry McVoy. It gained fame for being used by the Linux kernel project until a licensing dispute in 2005 led to its discontinuation in favor of free alternatives.
- Git (2005): Linus Torvalds, creator of Linux, developed Git after the fallout with BitKeeper. Git was designed with performance, flexibility, and speed in mind, especially for large projects like the Linux kernel. Git introduced powerful features such as branching, merging, and decentralized collaboration.
- Mercurial (2005): Developed around the same time as Git, Mercurial is another DVCS designed for speed and scalability. While it shares many similarities with Git, it is known for being more user-friendly and consistent in behavior.

What is Git?

- Git is a distributed version control system (DVCS) for tracking changes in source code during software development.
- Created by Linus Torvalds in 2005.
- Allows multiple developers to efficiently work on the same project and creates and environment for effective collaboration on software project.
- Key features:
  - Distributed architecture
  - Free and open source
  - Wide usage in the industry
  - Speed and efficiency
  - Data integrity
  - Support for non-linear development (parallel work on different features)
  - History and blame tracking

Key concepts

- Version control system
- Repository
- Commit
- Branch

Download Git

1. Navigate to the official Git website: `https://git-scm.com/`
2. Click on the **Downloads** section.
3. Select the appropriate version for your operating system (Windows, macOS, Linux).
4. Follow the installation instructions provided.

Creating a New Repository

- Initialize a new Git repository in an existing directory:

Command

```
git init
```

- Clone an existing repository:

Command

```
git clone <repository-url>
```

This could be the repository hosted anywhere (GitHub, GitLab or other platforms) or local repository

Commit

In Git, a commit is a fundamental unit of change. Commit has a unique identifier (commit hash) and it holds a number of options:

- Hash: unique identifier (SHA-1 checksum). This allows Git to uniquely identify each commit in the history of the project.
- Message: description (what was done in this change)
- Author (name, e-mail)

Commit message

Generally consists of two parts:

- Subject line (summary)
  - Usually is short (up to 50 characters)
  - Imperative is used (e.g., "Fix bug in user login" or "Add tests for API endpoint").
  - Avoid periods at the end of the line
- Body (optional)
  - Wrap lines at 72 characters
  - Explain why the change was made, rather than just what was done (the code diff itself explains the "what").

Commit message example

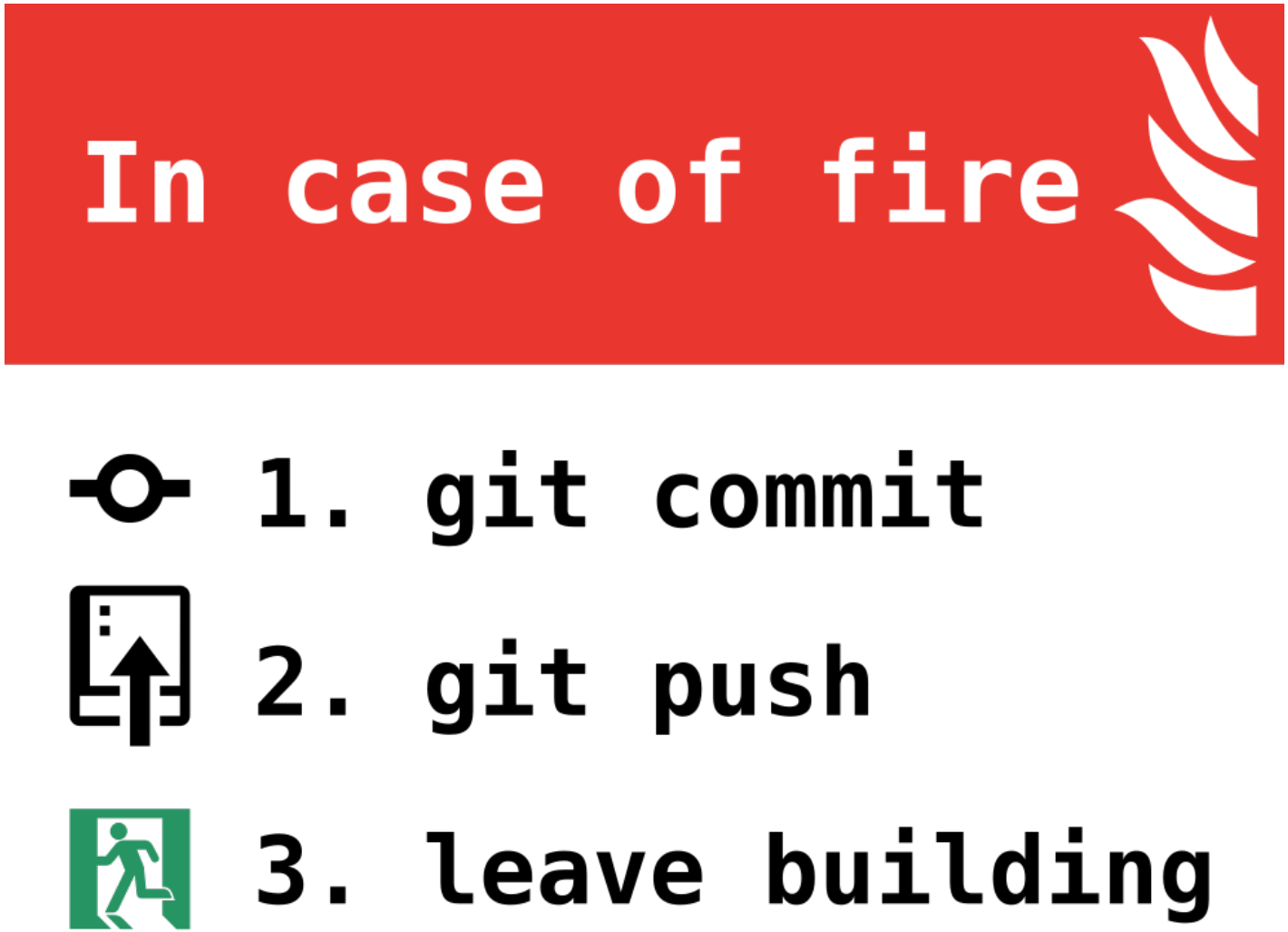
Add caching for user profile data

This improves the performance of loading user profiles by caching the data in memory. Previously, each request would query the database, which caused a significant slowdown.

Commit best practices

- Write meaningful commit messages: Follow the rules from previous slide. This helps others (and future you) understand the purpose of each commit.
- Make small, logical commits: Each commit should represent a single logical change. Avoid lumping multiple unrelated changes into one commit.
  - If you can split your commit into two in many cases it is better to do this.
  - If your commit message contains the word "and" this might be a signal that commit can be split
- Write meaningful commit messages: This helps others (and future you) understand the purpose of each commit.
- Commit often: Regular commits allow you to track progress and makes it easier to revert to a stable state if something goes wrong.

Commit importance



Source: https://github.com/hendrixroa/in-case-of-fire

First Commit

- Check the status of your repository:

Command

```
git status
```

- Stage files for commit:
  Before you make a commit, changes must be added to the **staging area** using `git add`. This allows you to carefully select which changes to include in a commit. For example, you might only want to commit changes to one file, even if you've modified several others.

Command

```
git add <file>
```

- Commit changes:

Command

```
git commit -m "Commit message"
```

Branches in Git

In Git, a branch represents an independent line of development, enabling you to work on different features, fixes, or experiments without affecting the main line of the project.
A branch is a movable pointer to a commit. It allows users to develop different project directions in parallel.
e.g. several developers are working on several different independant features

- Create a new branch:

Command

```
git branch <branch-name>
```

- Switch to a branch:

Command

```
git checkout <branch-name>
```

- Create and switch to a new branch:

Command

```
git checkout -b <branch-name>
```

Viewing Commit History

- View the commit history:

Command

```
git log
```

- View a summarized commit history (one line per commit):

Command

```
git log --oneline
```

- View graphical representation of branches:

Command

```
git log --graph --oneline --all
```

Push to Master (Linear history approach)

- All changes are made directly to the `master` branch.
- Simplest workflow, suitable for small projects.
- Potential issues:
  - Conflicts when multiple developers push simultaneously.
  - No isolation for new features or bug fixes.

Using Feature Branches

- Create separate branches for new features or bug fixes.
- Benefits:
  - Isolates development work.
  - Facilitates code reviews.
  - Safe integration into `master` after testing.
- Workflow:
  1. Create a new branch: `git checkout -b <feature-branch>`
  2. Develop and commit changes on the feature branch.
  3. Merge back into `master` when ready.

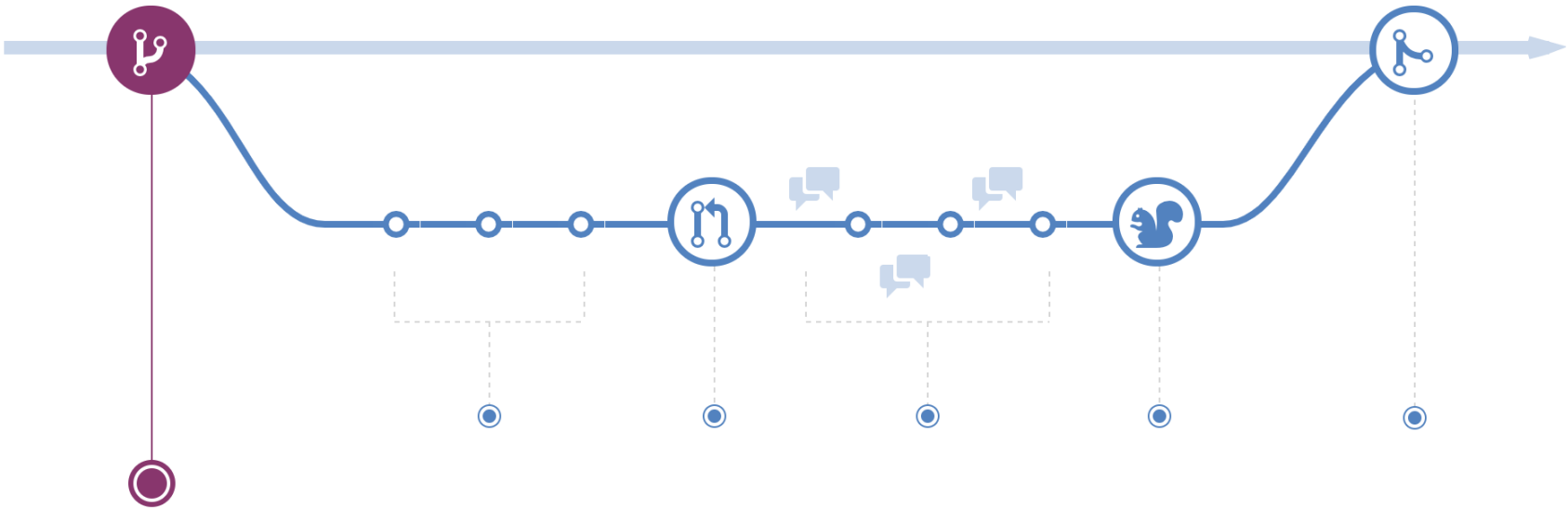
Trunk-based development

- Developers commit directly to the `trunk` (main branch), avoiding long-lived branches
- Encourages continuous integration by integrating small changes frequently
- If branches are used, they are short-lived (usually less than a day) and merged back quickly
- Use of feature flags is encouraged and allows incomplete features to be safely included in the main codebase
- Benefits:
  - Reduces merge conflicts and integration problems
  - Simplifies version control management
  - Facilitates rapid release cycles and continuous deployment
- Drawbacks:
  - Requires high discipline from developers to commit stable, working code frequently
  - Feature flags can become complex to manage, especially if there are many incomplete features
  - Approach is not suitable for big teams and teams working on long-term

GitHub Flow

- Lightweight, branch-based workflow suitable for continuous deployment.
- Steps:
  1. **Create a branch** for your work.
  2. **Commit** changes to your branch.
  3. **Open a Pull Request** when your work is ready.
  4. **Discuss and review** your code.
  5. **Merge** the Pull Request once approved.
  6. **Deploy** to production.
- Benefits:
  - Emphasizes collaboration and code quality.
  - Organized workflow for medium projects (a team or several teams that consist of several people).

GitHub Flow

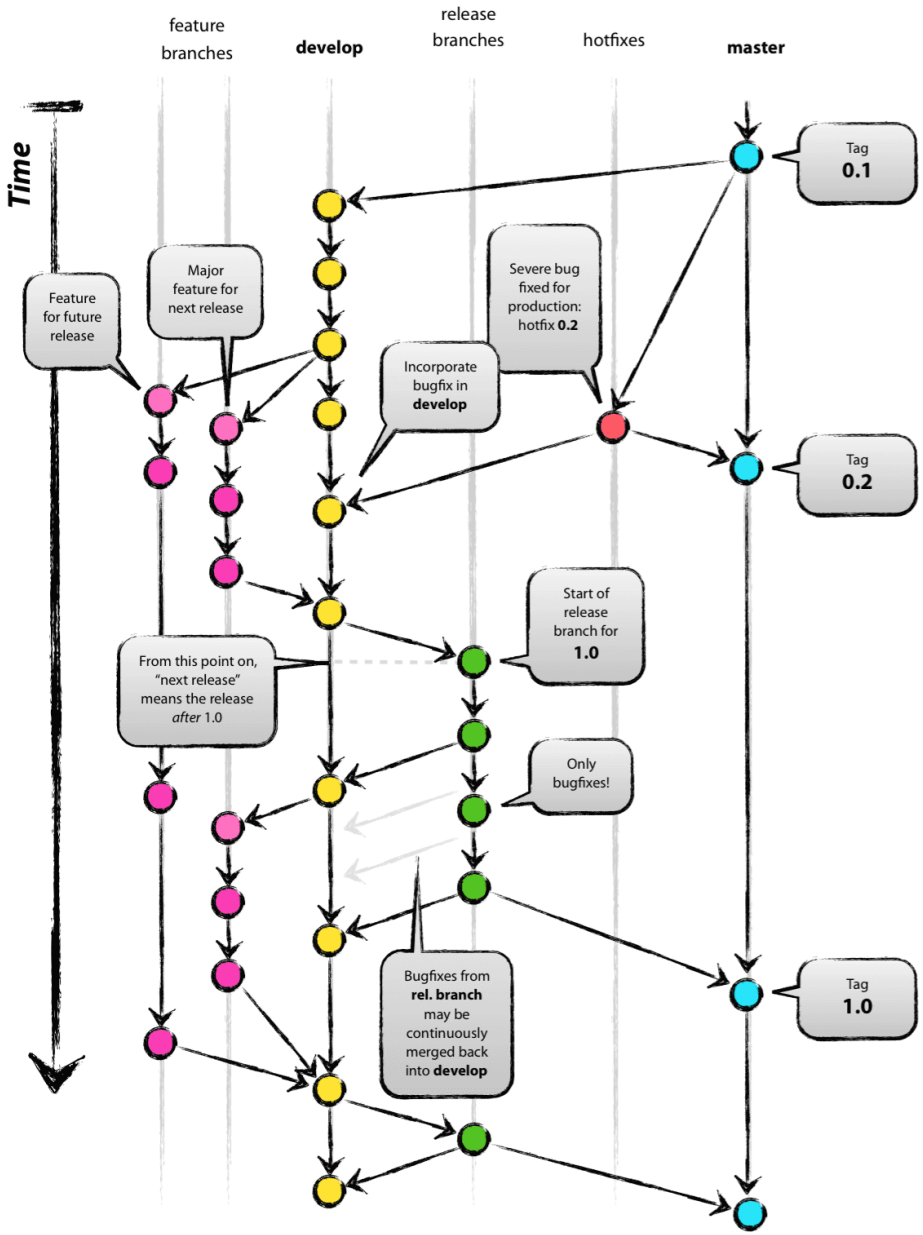


Source: https://github.com/a-a-ron/Github-Flow

Git Flow

- A robust branching model for managing releases.
- Defines specific branches:
  - **master** - contains production-ready code.
  - **develop** - integration branch for features.
  - **feature branches** - for new features.
  - **release branches** - prepare for a new production release.
  - **hotfix branches** - quick fixes for production.
- Benefits:
  - Organized workflow for large projects.
  - Clear separation of different types of work.

Git Flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Thank You!

References

1. Git Download https://git-scm.com/downloads
2. Git Book https://git-scm.com/book/en/v2
3. Git Flow Tutorial by Atlassian https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
4. Git Flow cheatsheet https://danielkummer.github.io/git-flow-cheatsheet/
5. GitHub Flow Tutorial by GitHub https://docs.github.com/en/get-started/using-github/github-flow