**Version Control System**

Table of Contents

[Features and Capabilities 3](#_Toc83393755)

[Vocabulary 4](#_Toc83393756)

[Branching 6](#_Toc83393757)

[Merging Branches 6](#_Toc83393758)

[Versioning Models 8](#_Toc83393759)

[Centralized Version Control 8](#_Toc83393760)

[Distributed Version Control 9](#_Toc83393761)

[Lock-Modify-Unlock 10](#_Toc83393762)

[Copy-Modify-Merge 10](#_Toc83393763)

[File Comparison and Merge Tools 11](#_Toc83393764)

[Benefits 12](#_Toc83393765)

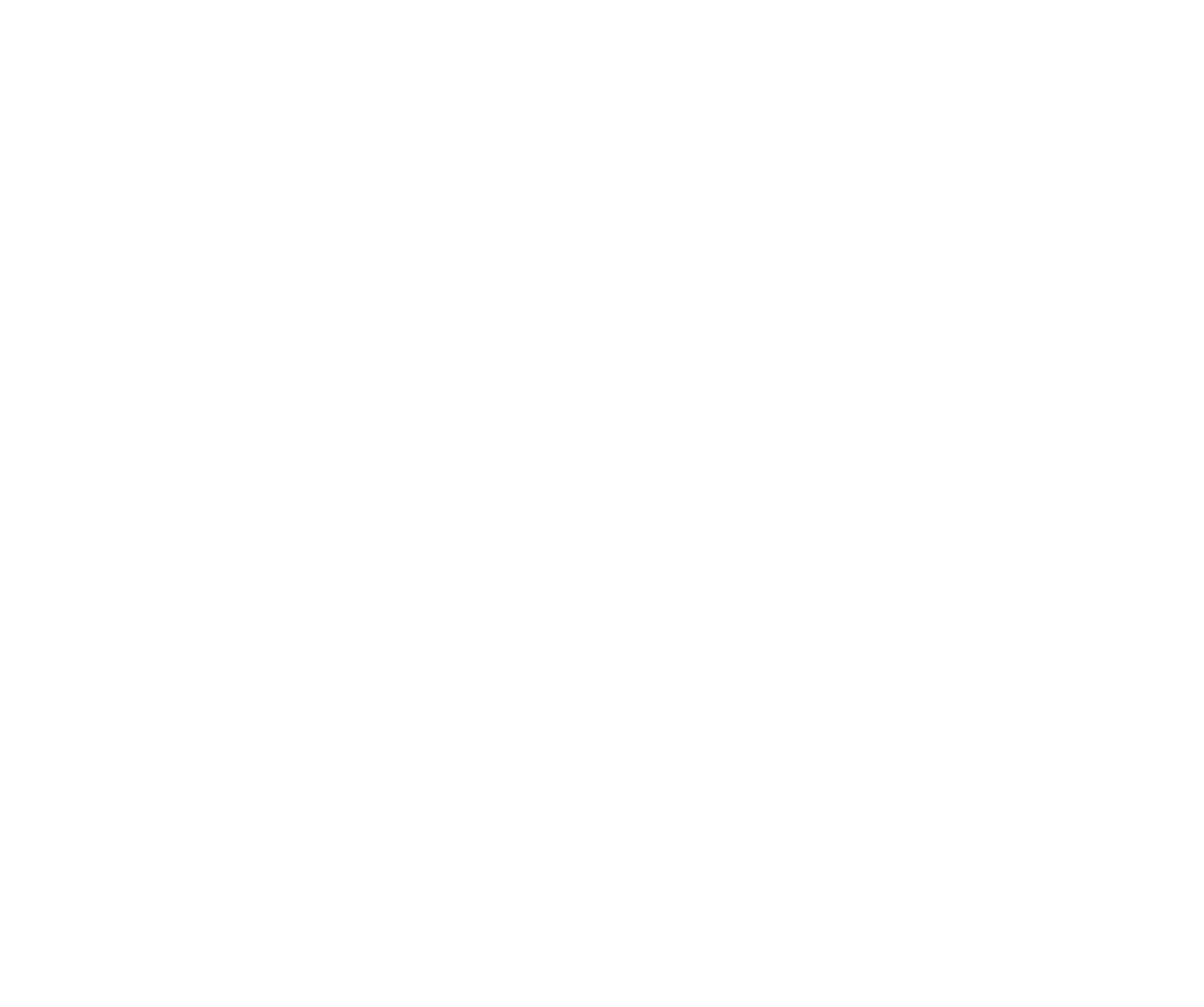
[Choosing the Right VCS 13](#_Toc83393766)

[Best Practices 14](#_Toc83393767)

A **Version Control System** (VCS), also called **Software Configuration Management** (SCM), helps organizations systematically manage, organize and control the **changes** in documents, code and other entities during the Software Development Life Cycle. They help keep track of **what is happening** in the project over time and help solve **conflicts** in the changes.

The primary goal is to increase productivity with minimal mistakes.

Although VCSs are generally associated with code, they are actually involved at every stage of the software development life cycle, working with different documents at each stage.



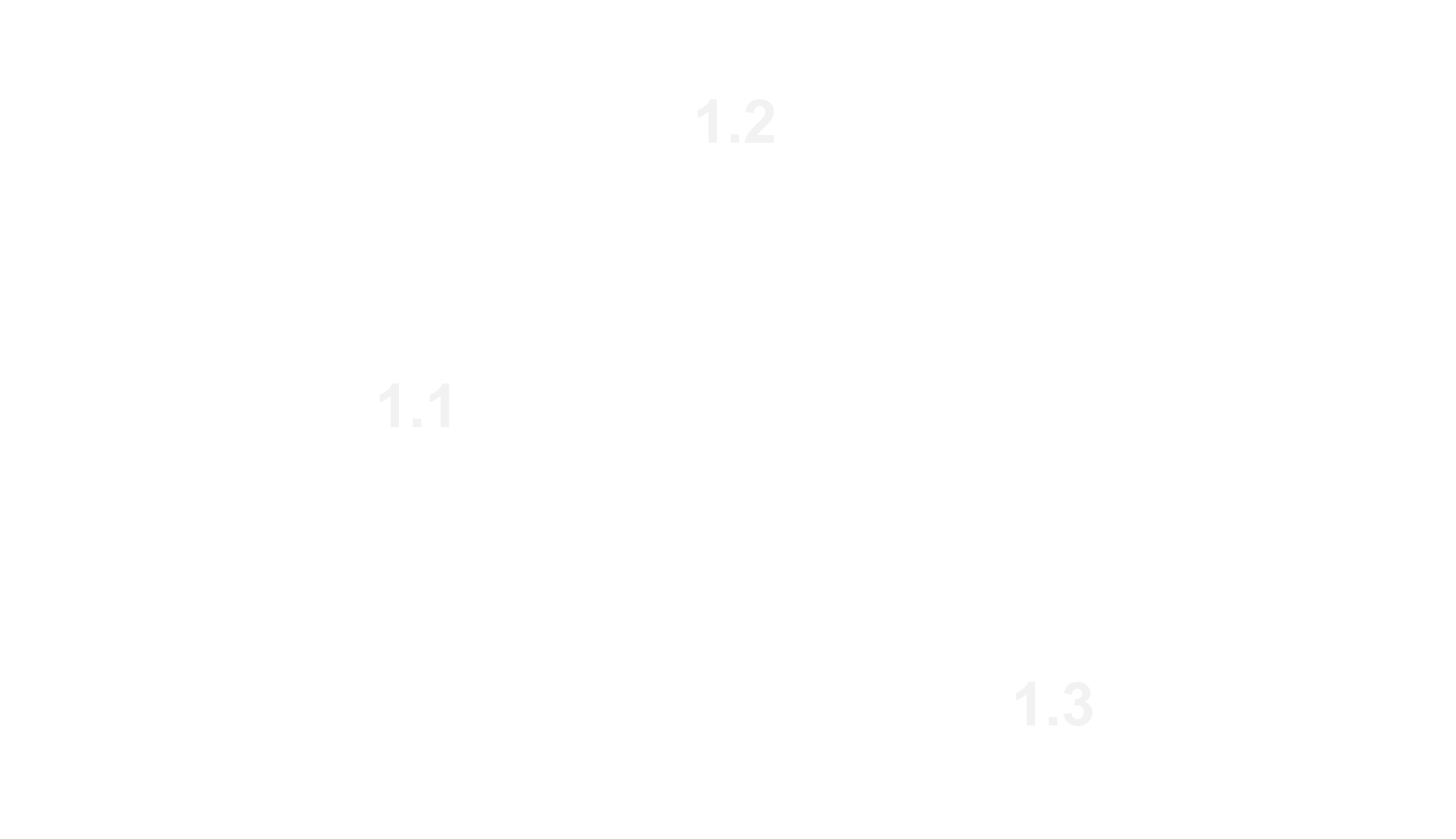
In summary, a VCS manages **different versions** of the **same document**. Some well-known VCS products are CVS, SVN, Git, Mercurial, Perforce and Microsoft TFS.

## Features and Capabilities

* File versions control
* Merge and differences search
* Branch creation and deletion
* Compare and merge version branches
* Revert code to previous versions
* File locking
* Code version management (e.g. conflict resolution)
* Parallel development streams & branches
* Console and GUI clients
* Constantly used in software engineering (during software development and while working with documents)
* Changes are identified using version numbers (1.0, 2.0, 2.17, etc.)
* Complete change logs are maintained, with the time of the change, the author and the files changed
* Old versions of files can be retrieved, examined and compared
* It is possible to revert to an old version

## Vocabulary

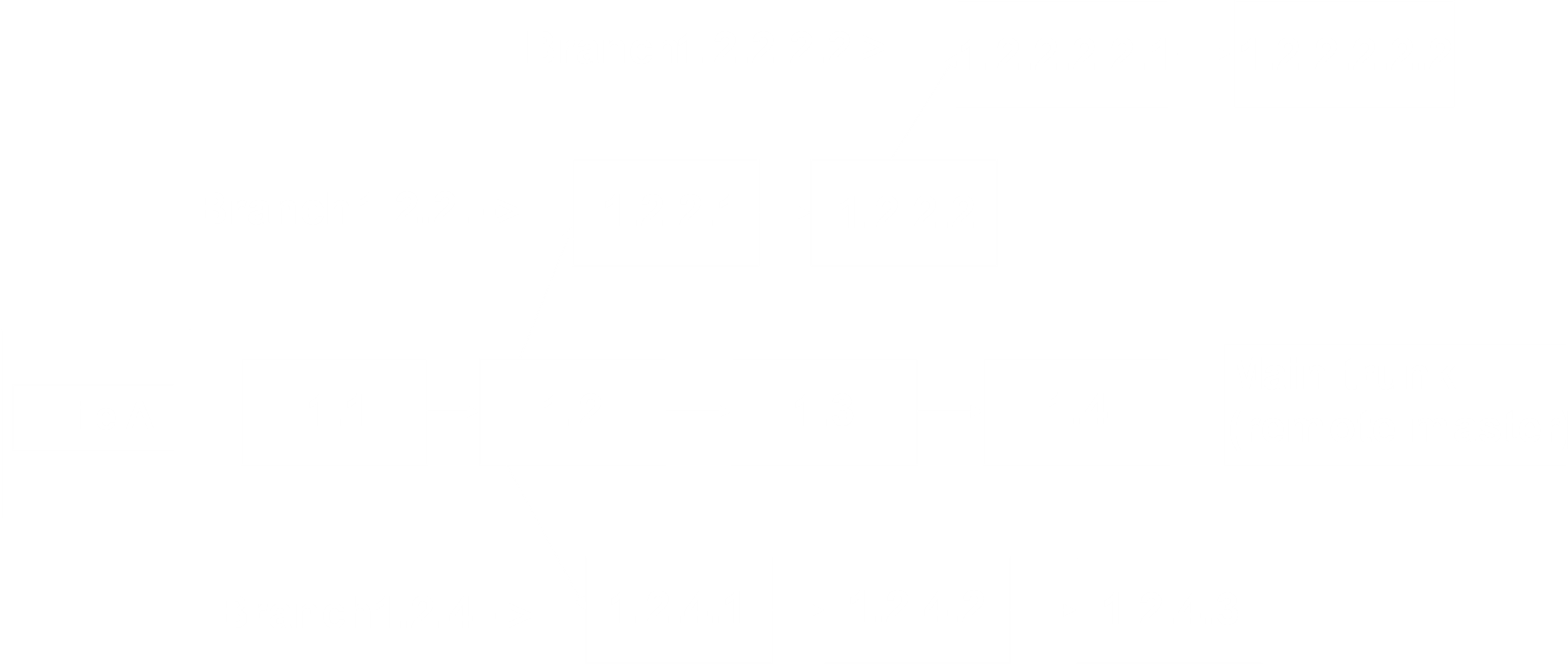
* **Repository** – This is a server that stores files and keeps a change log.
* **Revision, Version** – This is a specific version of a document, which may be the result of multiple changes.
* **Check-Out, Clone** – This is the process of retrieving a copy of the files from the remote repository into a local directory. It is possible to lock the files.
* **Change** – This is a modification of a local file that is under version control.
* **Change Set, Change List** – This is a set of changes to multiple files that will be committed at the same time.
* **Commit, Check-In** – This is the process of submitting the changes made from the local working copy to the repository. This automatically creates a new version. Conflicts may occur at this stage.
* **Conflict** – Simultaneous changes to the same file by multiple users can cause conflicts. These may be resolved automatically or manually.
* **Update, Get Latest Version, Fetch, Pull** – This is the process of downloading the latest version of the files from the repository to a local working directory. It also involves resolving merge conflicts.
* **Undo Check-Out, Revert, Undo Changes** – This is the process of cancelling local changes and restoring the state of the files from the repository.
* **Merge** – This is the process of combining the changes to a file made locally with the copy in the repository. This is automated in most cases.
* **Label, Tag** – These are used to mark specific versions of files, for example for release.



* **Branching** – This is the process of dividing a repository into a separate workflow.

## Branching

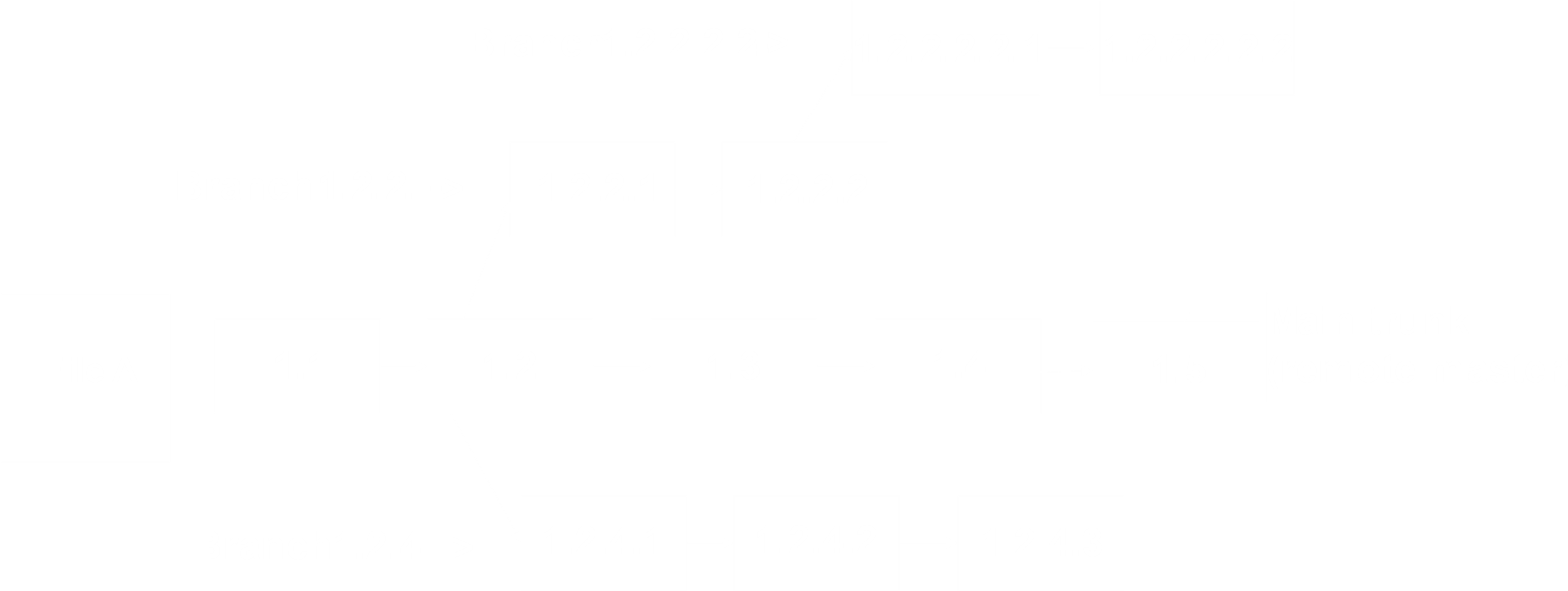
A branch is essentially a copy of the repository that can be made to do work that we might not necessarily want on the main repository just yet. It may be the case that different developers are working in different branches, or that different features are being developed on different branches.



Branching is appropriate for the development of **new features or fixes**. The changes made in the branch, even if commits are made, are **not visible** on the main development line until the branch is merged. The **version** of the repository being worked on in a branch may also be different.

### Merging Branches

When working on a branch, changes made are committed to that branch, not to the main development line. However, when all the work on the branch is complete and has been tested, it needs to be **merged** into the main development line.



## Versioning Models

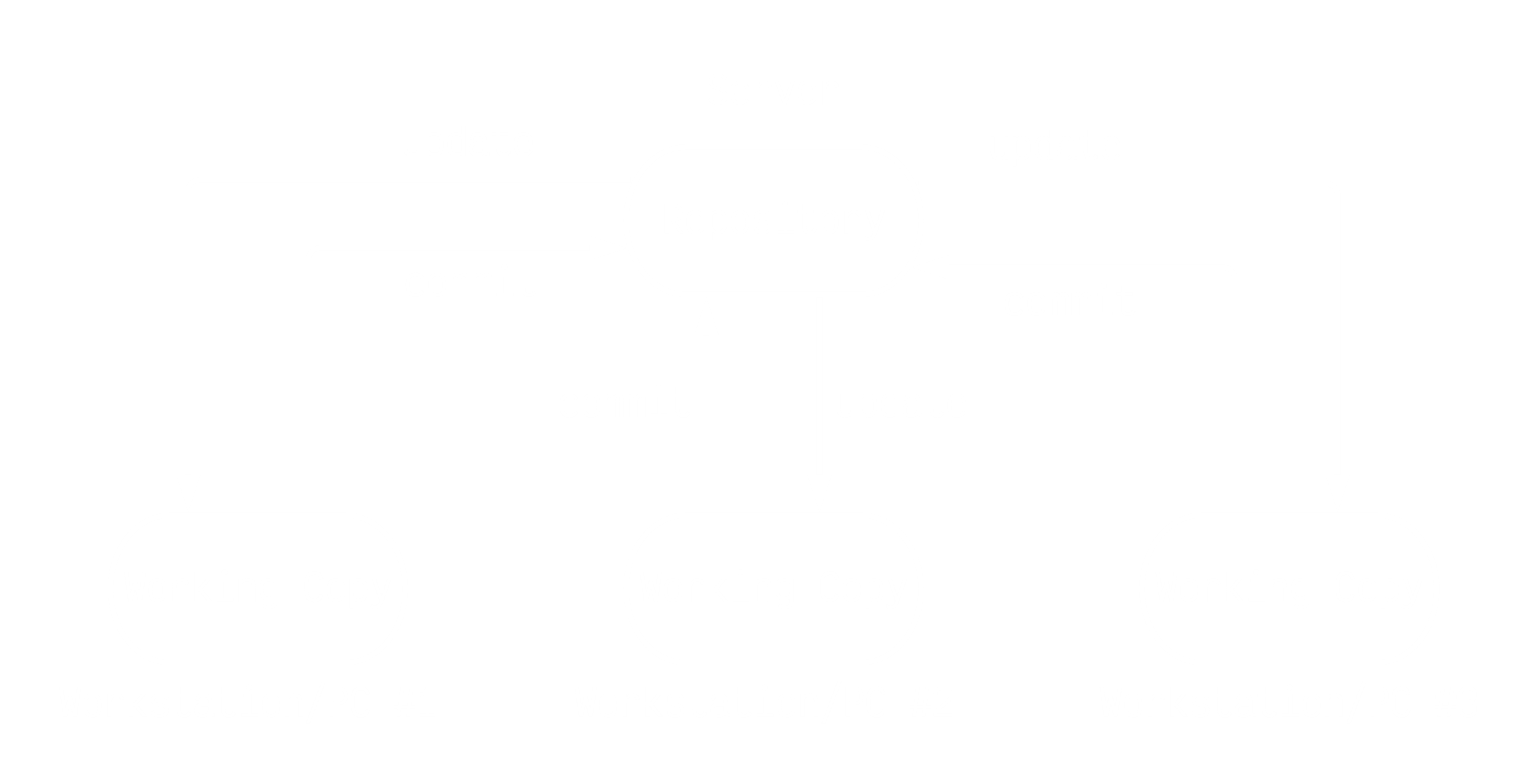
VCS tools are primarily of two types:

* Centralized
* Distributed

Each of these types can use either the Lock-Modify-Unlock method or the Copy-Modify-Merge method.

### Centralized Version Control

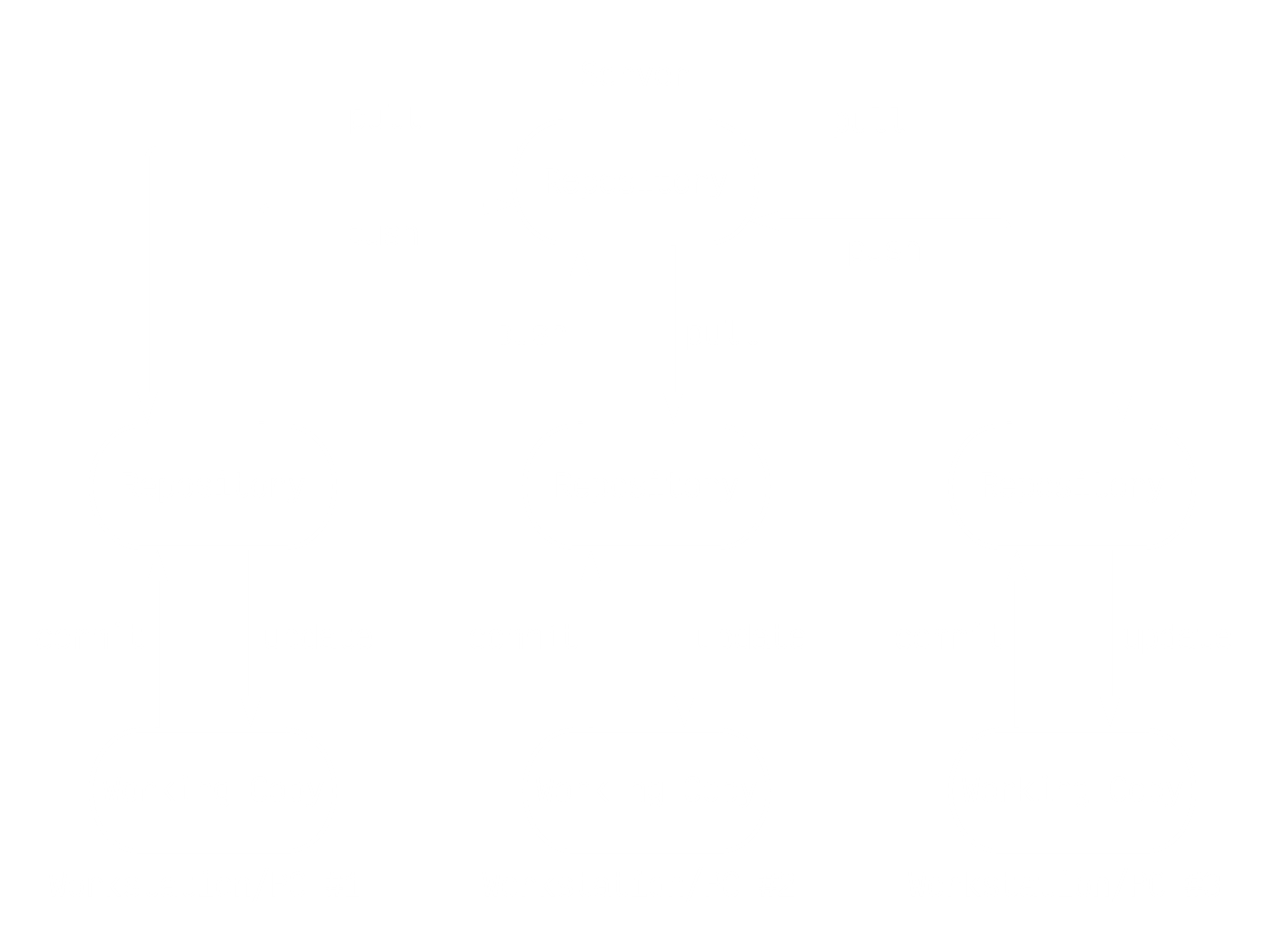
**Centralized Version Control** is based on the idea of having a **single central copy** of the project to which changes are committed.



The main benefits of such as system is that it is easier to understand and use and can grant access level control at the directory level.

### Distributed Version Control

In **Distributed Version Control**, each user makes a **copy** of the project and has the full history of the project on their local hard drive. Changes are made on the copy and later merged with the main project.



The main benefits of such a system are improved performance, easier branching and merging and the lack of a need to constantly be connected to the main project.

For Distributed Version Control, users are working on their own repositories. This uses the **Lock-Modify-Unlock** mechanism. The changes made are **locally committed**, which means at this point, no concurrency or conflicts can occur. Afterwards, the local repository is **pushed** to the central repository. At this point, conflicts can occur and merges occur frequently.

Thus, the only difference between Distributed Version Control and Centralized Version Control is that a copy of the entire repository exists with each user and users can commit directly to the local copy instead of just copying specific files that they will modify and commit back to the main repository.

### Lock-Modify-Unlock

In this model, only **one user** works on a particular file at a time. **No conflicts** can occur under this model. Other users must wait for the file to be unlocked. This only works for small development teams and is rarely used in the real world, due to the huge number of potential issues.

Someone may forget to unlock a file, time is wasted while waiting for locked files and entire files are locked, which may be unnecessary. For example, if one user is working at the start of one file and another is working at the end of the file, they could work simultaneously without causing conflicts, but are unable to do so.

Note that the **checkout** process can be done without locking the files, i.e. a copy of the repository can be made by different developers without locking the files. It is just that the file needs to be locked before it can be modified.

### Copy-Modify-Merge

In this model, users make changes **parallelly** to their own working copies. This can lead to **conflicts** if multiple users edit the same file. Conflicts are **merged** to produce a final version, which can be done automatically or manually.

The merging process is where things get difficult. Although it can be automated, this is not always possible. Developers need to be coordinated and responsible. They need to commit changes as soon as they are done and not commit code that does not compile or blocks the work of others. They should leave meaningful comments for each commit.

### File Comparison and Merge Tools

For manual merges, file comparison is important. Some famous visual comparison and merge tools are:

* TortoiseMerge
* WinDiff
* AraxisMerge
* WinMerge
* BeyondCompare
* CompareIt

## Benefits

* The system manages the directories, files, and individual changes made over time.
* Allows users to find root causes for mistakes or bugs, or revert to an earlier version.
* Improves team productivity and enables collaboration.
* Enhances team communication with a reliable solution.
* Reduces development errors and conflicts.
* Improves customer satisfaction with reliable software versions.

## Choosing the Right VCS

The things that modern development teams look for in a VCS are:

* Concurrent Development
* Automation
* Team Collaboration
* Tracked Changes — Who, What, When, Why
* High Availability/Disaster Recovery

## Best Practices

* Save changes and back-up code consistently and frequently.
* Use a comprehensible naming convention.
* Manage appropriate controls to allow proper, productive access.
* Ensure developers can easily pull up necessary documents.
* Define team, individual, and read only access to streamline document availability and allow users to perform their functions in a timely manner.
* Use a descriptive commit message.
* Incorporate others' changes frequently
* Share your changes frequently
* Understand your merge tool