



Hochschule
München
University of
Applied Sciences

Einführung in das Versionsverwaltungssystem Git und das LRZ GitLab

Benjamin Kormann

Version: 1.3

Stand: 11. September 2022

Hochschule München
Fakultät für Elektro- und Informationstechnik

Inhaltsverzeichnis

1	Einführung in die Versionsverwaltung	1
1.1	Aufgaben der Versionsverwaltung	2
1.2	Typische Versionsverwaltungssysteme	4
2	Technische Vorbereitungen	6
2.1	Git Softwareinstallation	6
2.1.1	Installation unter Linux	6
2.1.2	Installation unter Windows	7
2.1.3	Installation unter Mac OS	7
2.2	Grundkonfiguration von Git	8
2.3	LRZ GitLab einrichten	10
3	GitLab und Git anwenden	12
3.1	Übersicht eines typischen Setups	12
3.2	Vorgehensweise in GitLab	12
3.2.1	Ein GitLab Projekt erstellen	13
3.2.1.1	Neues GitLab Projekt anlegen	13
3.2.1.2	Fork von einem GitLab Projekt erstellen	14
3.2.2	Zugriffsrechte eines Projektes	16
3.2.3	Dokumentation in GitLab	16
3.3	Einführung in gängigen Git Workflow	17
3.3.1	Projekt anlegen	19
3.3.2	Dateien im Repository verwalten	20
3.3.3	Änderungen ins Repository übertragen	21
3.3.4	Dateien von Versionierung ausschließen	22
3.3.5	Synchronisation mit zentralem Repository	23
3.3.6	Lokale Änderungen zurücksetzen	25
3.3.7	Entferntes Repository hinzufügen	25
3.3.8	Statusinformationen abrufen	26
3.3.9	Historie der Änderungstexte abrufen	27
3.3.10	Versionsunterschiede abfragen	27

3.4	Einführung in CI/CD	28
3.4.1	Konfiguration von CI/CD	30
3.4.1.1	Konfiguration GitLab Runner	30
3.4.1.2	Konfiguration CI/CD Pipeline	30
3.4.2	Anwenden von CI/CD	31
3.4.2.1	Starten einer Pipeline	31
3.4.2.2	Testergebnisse einer Pipeline	32
3.4.2.3	Artefakte einer Pipeline	33
3.5	Typische Git Use Cases	34
3.5.1	Grundlagen	34
3.5.1.1	Existierendes Repository klonen	35
3.5.1.2	Dateiveränderungen versionieren	35
3.5.1.3	Dateien ignorieren	36
3.5.2	Zentrales Repository	37
3.5.2.1	Ablauf bei der Arbeit mit zentralem Repository	38
3.5.2.2	Nebenläufiges Arbeiten, Konflikte vermeiden	38
3.5.2.3	Nebenläufiges Arbeiten, Konflikte auflösen . .	40
3.5.3	Zugriff auf frühere Versionen	42
3.5.3.1	Gelöschte Datei wiederherstellen	42
3.5.3.2	Früheren Stand einer Datei wiederherstellen	43

1 Einführung in die Versionsverwaltung

In der professionellen Softwareentwicklung sind Systeme zur Versionsverwaltung wichtige Basiswerkzeuge. Es haben sich in den letzten Jahrzehnten sowohl proprietäre als auch Open Source Lösungen entwickelt, die in der Wissenschaft, der Industrie und insbesondere in der Entwicklung von freier offener Software eingesetzt werden. Diese Systeme verwalten dabei Projekte beliebiger Größe in Bezug auf Anzahl der Entwickler und der Menge an Quelltextdateien. So besteht bspw. der Linux Kernel (Version 5.10) aus ca. 70.000 Dateien, ca. 30.000.000 Zeilen Quelltext mit der Größe von über 100.000 kB und wird weltweit kontinuierlich weiterentwickelt. Laut der Linux Foundation waren seit 2005 ca. 15.600 Entwickler aus mehr als 1.400 Unternehmen an der Linux Kernel Entwicklung beteiligt [1]. Der Kernel wächst jährlich um ca. 1.500.000 Zeilen Quelltext. Durch manuelle Verwaltung und nichtskalierende Werkzeuge wäre die Verwaltung und Organisation eines derartigen Projektes nicht machbar.



In vielen Diskussionen wird von einer Versionsverwaltung gesprochen, wenn Quelltextdateien in zip-Archive gepackt und mit einer Endung `_v2.zip` als Versionsnummer versehen werden. Das sind **keine geeigneten** Systeme zur Verwaltung von Softwareversionen. In Abschnitt 1.2 werden typische Systeme zur Versionsverwaltung vorgestellt.

Für die Linux-Entwicklung kommt das freie Versionsverwaltungssystem Git zum Einsatz, welches im Jahr 2005 von Linus Torvalds (Erfinder von Linux) initiiert wurde. Git hat sich in den letzten Jahren zu einem der beliebtesten Versionsverwaltungssysteme etabliert, da es neben den technischen Vorteilen auch durch die Unterstützung in Produkten der Softwareentwicklung eine breite Anwendergemeinde besitzt.

Der Umgang und die Anwendung der wesentlichen Funktionen eines Versionsverwaltungssystems gehören zu den Grundfähigkeiten eines Softwareentwicklers. Aus diesem Grund wird in den Lehrveranstaltungen zunehmend Git für die Bearbeitung der Programmieraufgaben eingesetzt, wodurch jeder Teilnehmern durch den praktischen Einsatz der dazugehörigen Werkzeuge Basiswissen aufbaut.

Lernziele

- Grundverständnis in der Anwendung von Git aufbauen
- GitLab als zentrale CI/CD Plattform kennenlernen
- Vorteile der Versionsverwaltung auch für kleine Softwareprojekte verstehen

1.1 Aufgaben der Versionsverwaltung

Software wird heutzutage üblicherweise im Team entwickelt. Die gesamten Entwicklungsarbeiten müssen dabei an einer zentralen Stelle zusammenlaufen, um daraus eine gültige Konfiguration und somit eine lauffähige und funktionsfähige Softwareversion erstellen zu können. Das Zusammenführen von verschiedenen Softwareartefakten (Quelltext, Konfigurationsdateien, Ressourcen etc.) in einem zentralen Repository stellt die Grundlage für eine kontinuierliche Erstellung der Software. Dazu werden meist auch sog. Continuous Integration / Continuous Delivery (CI/CD) Werkzeuge eingesetzt.

Aus Sicht des einzelnen Entwicklers besteht der Arbeitsfluss im wesentlichen aus den folgenden Schritten: aktuellen Stand des Quelltextes laden, Änderungen vornehmen, Änderungen testen, neuen Stand hochladen. Dabei kann die Ursache für diesen Workflow eine Weiterentwicklung der Software, aber auch die Fehleranalyse und anschließende Fehlerbehebung sein. In beiden Fällen werden Änderungen an der Software (dem Quelltext) vorgenommen, die zur Nachvollziehbarkeit protokolliert werden müssen. Ein Versionsverwaltungssystem bietet dazu sog. *commit*-Messages über die mit Hilfe einer aussagekräftigen Beschreibung, die Änderungen zusammengefasst werden können. Manchmal müssen verschiedene Softwarestände miteinander verglichen werden, da eine bereits vorhandene Funktionalität im Rahmen einer Weiterentwicklung plötzlich fehlerhaft geworden ist. Um die Änderungen, die möglicherweise zu diesem Verhalten geführt haben, nachvollziehen zu können, können frühere Versionsstände wiederhergestellt, mit einem weiteren Versionsstand verglichen und die Änderungen relativ einfach identifiziert werden. Diese Aufgaben fallen bereits bei kleinen Projekten (sogar bei einem Entwickler) an, so dass sich der Einsatz eines Versionsverwaltungssystems schnell auszahlt. Ein typisches Szenario stellt dabei bspw. auch die Entwicklung auf mehreren Endgeräte dar, auf denen der Entwicklungsstand aktuell und synchron gehalten werden muss.

Neben der Protokollierung von Änderungen und Wiederherstellung beliebiger Softwarestände unterstützen Versionsverwaltungssysteme auch das Entwickeln auf parallelen Entwicklungszweigen (*branch*). Dieser Mechanismus wird meist genutzt, um die Weiterentwicklung eines Softwarestandes um neue Funktionen (Features) zu organisieren. Nach der Freigabe können diese parallel entwickelten Features auf den Hauptentwicklungszweig zurückgeführt werden. Somit wird üblicherweise sichergestellt, dass der Hauptentwicklungszweig

stets einen stabilen Softwarestand darstellt.

Im Vergleich zur Softwareorganisation über Verzeichnisse und Dateien in einem Dateisystem bieten Versionsverwaltungssysteme meist ein feingranulares Authentifizierungs- und Autorisierungssystem. Darüber können die Zugriffe (lesen, schreiben uvm.) auf Benutzer- oder Gruppenebene definiert werden. Da der Quelltext meist ein wichtiges Firmen-Know-How darstellt, muss der Zugriff darauf auch gesteuert werden.

1.2 Typische Versionsverwaltungssysteme

Bei den Versionsverwaltungssystemen werden grundsätzlich zwei¹ verschiedene Arten unterschieden, nämlich zentrale und dezentrale Systeme.

Bei den zentralen Systemen existiert ein zentrales, allgemein gültiges Repository, in dem Software verwaltet und auf das durch mehrere Entwickler (Sternmuster) zugegriffen wird. Dadurch kann parallel an einer gemeinsamen Software entwickelt und mögliche Konflikte müssen bei dem Hochladen des neuen Softwarestandes aufgelöst werden. Der Konflikt entsteht meist dann, wenn durch zwei oder mehr Entwickler inkonsistente Änderungen am gleichen Softwaremodul durchgeführt wurden. Um Konflikte dieser Art zu vermeiden bedarf es einer guten Softwarestruktur (Architektur, Design) und klare Aufgabenzuweisungen (Verantwortung) auf die beteiligten Entwickler.

Im Vergleich dazu zeichnen sich dezentrale oder auch verteilte Systeme dadurch aus, dass zwar meist auch ein zentrales Repository existiert, aber jeder Entwickler eine lokale Kopie mit sämtlicher Historie besitzt. Ein wesentlicher Vorteil dabei ist, dass Entwickler durch die lokale Kopie komplett vom zentralen Repository entkoppelt entwickeln und die Änderungen

¹Beide Arten können auch lokal verwendet werden.

Art	Name	Link
Dezentral	Git♣	https://git-scm.com/
	Bazaar♣	https://bazaar.canonical.com/
	Mercurial♣	https://www.mercurial-scm.org/
Zentral	SVN♣	https://subversion.apache.org/
	Perforce	https://www.perforce.com/
	Clearcase	https://www.ibm.com/
	CVS♣	https://savannah.nongnu.org/

Tabelle 1.1: Auswahl typischer Vertreter von zentralen und dezentralen Versionsverwaltungssystemen

nachverfolgen können. Dabei wird einerseits keine permanente Netzwerkverbindung zu dem Repository benötigt und es wird die Häufigkeit der Versionierung von Änderungen gefördert, da nicht jeder Arbeitsstand sofort mit dem zentralen Haupt-Repository zusammengeführt werden muss.



Die meisten modernen Versionsverwaltungssysteme arbeiten nach dem Copy-Modify-Merge Prinzip, wodurch paralleles Arbeiten an identischen Dateien ohne Bearbeitungssperre ermöglicht wird.

In der Tabelle 1.1 sind typische Vertreter von Versionsverwaltungssystemen (♣ Open Source) aufgeführt. Diese Übersicht ist nicht vollständig, sondern stellt nur eine Auswahl dar. In vielen Unternehmen haben sich häufig gewisse Produktlandschaften etabliert, die meist aus Rahmenverträgen und somit besseren kommerziellen Konditionen resultieren. Dadurch können auch andere Versionsverwaltungssysteme zum Einsatz kommen, die nicht zu den typischen Vertretern zählen.

2 Technische Vorbereitungen

2.1 Git Softwareinstallation

Für die Interaktion mit einem Git Repository wird ein Git Client benötigt. Die Verfügbarkeit und Qualität von Clients hängt mitunter vom verwendeten Betriebssystem ab, auf dem ein Client installiert wird. Neben den Clients, die direkt über die Paketverwaltung des Betriebssystems verfügbar sind, bieten viele Entwicklungsumgebungen eigene Erweiterungen zur Kommunikation mit Versionsverwaltungssystemen. So können bspw. in Visual Studio Code, Eclipse, QtCreator etc. Git Erweiterungen hinzugefügt werden. Der Vorteil dieser Erweiterungen liegt meist darin, dass auch eine graphische Benutzeroberfläche mitgeliefert wird, was insbesondere bei dem Vergleich von Versionsständen häufig intuitiver ist als eine textuelle Ausgabe in dem Konsolen-Client.



Die Erläuterungen zur Verwendung von Git werden in diesem Dokument grundsätzlich anhand der Konsolenkommandos erläutert. Dadurch soll ein besseres Verständnis für das System erreicht werden. Der Einsatz graphischer Clients ist trotzdem möglich, jedoch sind diese unterschiedlich in der Anwendung und werden deshalb nicht näher beschrieben.

2.1.1 Installation unter Linux

Die gängigen Linux Distributionen (Ubuntu, Debian etc.) besitzen meist über die Standardinstallation bereits eine lauf-

fähige Installation des Git Clients für die Konsole. Bei den meisten Distributionen lautet das benötigte Paket `git` bzw. `gitk` für einen graphischen Client. Je nach Verfügbarkeit von Entwicklungsumgebungen empfiehlt es sich, die mitgelieferten Git Erweiterungen zu nutzen.

Alternativ sind einige graphische Benutzeroberflächen für Git unter <https://git-scm.com/download/gui/linux> verfügbar.

2.1.2 Installation unter Windows

Für die Installation eines Git Clients unter Windows stehen unter <https://git-scm.com/download/win> diverse Optionen zur Verfügung. Hier muss der Download für das entsprechende Betriebssystem (32 Bit / 64 Bit) ausgewählt werden.

Alternativ gibt es weitere graphische Benutzeroberflächen für Git unter <https://git-scm.com/download/gui/windows>.

2.1.3 Installation unter Mac OS

Die Installation von Git unter Mac OS benötigt das Homebrew Paketverwaltungssystem. Unter <https://brew.sh> ist eine Anleitung zur Installation verfügbar.

Die anschließende Installation von Git erfolgt in der Konsole über das folgende Kommando:

```
brew install git
```

Unter Mac OS wird häufig in der Entwicklungsumgebung Xcode entwickelt. In Xcode ist die Unterstützung für Git bereits als Binärpaket integriert und kann bei der Softwareentwicklung direkt verwendet werden.

Alternativ sind unter <https://git-scm.com/download/gui/mac> auch einige graphische Benutzeroberflächen für Git verfügbar.

2.2 Grundkonfiguration von Git

Vor der ersten Verwendung von Git muss eine Grundkonfiguration erstellt werden. Bei der Verwendung von graphischen Benutzeroberflächen oder Erweiterungen in Entwicklungsumgebungen können die Einstellungen von Git üblicherweise in graphischen Dialogen vorgenommen werden.

Die zwei wichtigsten Einstellungen sind der Benutzername und eine dazugehörigen E-Mail Adresse. Mit diesen persönlichen Informationen erfolgt die Interaktion mit dem Git Repository, sprich Änderungen werden mit diesen konfigurierten Benutzerdaten protokolliert. Um diese Einstellungen global verfügbar zu machen, d.h. für alle Git Repositories müssen die folgenden Kommandos in der Konsole ausgeführt werden:

```
git config --global user.name "John Doe"
git config --global user.email jd@example.com
```



Für die Lehrveranstaltungen müssen der reale Name und die E-Mail Adresse der Hochschule München verwendet werden.

Je nach Betriebssystem und verwendeter Software für die *diff* und *merge* Funktionalität (siehe Tabelle 2.1) der Versionsverwaltung muss diese noch in die Git Konfiguration aufgenommen werden. Um bspw. KDiff3 als Tool für *merge* und *diff* zu setzen müssen die folgenden Kommandos in der Konsole ausgeführt werden:

```
git config --global diff.tool kdiff3
git config --global merge.tool kdiff3
```

Unter Windows kann es erforderlich sein, den Pfad zur ausführbaren Datei des *diff/merge* Tools zu setzen. Dieser kann

Tool	Linux	Windows	Mac OS
Meld	x	x	x
Diffuse	x	x	x
KDiff3	x	x	x
FileMerge			x
Araxis Merge		x	x
TkDiff	x	x	x
Beyond Compare	x	x	x
P4Merge	x	x	x
DeltaWalker	x	x	x
Kaleidoscope			x
Code Compare		x	
WinMerge		x	

Tabelle 2.1: Mögliche Tools für merge/diff

mit den folgenden Kommandos in der Konsole gesetzt werden. Dabei muss für PFAD der tatsächliche Installationspfad eingesetzt werden, wie bspw. `"c://kdifff3/kdifff3.exe"`.

```
git config --global difftool.kdifff3.path PFAD
git config --global mergetool.kdifff3.path PFAD
```

Neben der globalen Konfiguration, die für alle Git Repositories angewendet wird können auch Konfigurationen für jedes Repository einzeln eingestellt werden. Dazu muss in der Konsole in das Hauptverzeichnis der lokalen Kopie des Git Repositories gewechselt und anschließend können die folgenden Kommandos ausgeführt werden:

```
git config user.name "John Doe"
git config user.email jd@example.com
```



Die angepasste Konfiguration je Git Repository ist dann erforderlich, wenn auf einem Rechner auf mehrere Repositories mit unterschiedlichen Identitäten (Name, E-Mail) zugegriffen werden soll.

2.3 LRZ GitLab einrichten

Das Leibniz Rechenzentrum stellt für Angehörige der Hochschule einen GitLab Zugang zur Verfügung. GitLab ist eine freie Anwendung zur Unterstützung des Entwicklungsprozesses von Software. Neben der Versionsverwaltung mit Git und vielen weiteren Modulen besitzt GitLab¹ u.a. ein System für CI/CD (Continuous Integration / Continuous Delivery) zur automatisierten Erstellung und Auslieferung von Software.

Abbildung 2.1: LRZ GitLab Login Screen

Um sich am LRZ GitLab anzumelden müssen unter <https://gitlab.lrz.de> die persönlichen Benutzerdaten der Hoch-

¹Die LRZ GitLab Installation unterstützt die CI/CD Funktionalität nicht.

schule verwendet werden. Die Abbildung 2.1 zeigt den Login Screen bei dem *LDAP (LRZ-Users)* ausgewählt werden muss. Nach Eingabe der Benutzerdaten erfolgt der Login mit einem Klick auf *Sign in*.



Beim Benutzernamen muss der Präfix **hm-** vorangestellt werden. Das Passwort entspricht dem Passwort zum Benutzernamen.

In dem persönlichen Profil können weitere Einstellungen wie Sprache, Kontaktdaten, SSH Keys etc. vorgenommen werden. Für den Zugriff auf ein Repository müssen sich Benutzer authentifizieren. Das LRZ GitLab unterstützt hierbei Benutzername/Passwort über HTTPS sowie das Public-Key-Authentifizierungsverfahren über SSH. Ein wesentlicher Vorteil über SSH besteht in der automatischen Anmeldung, d.h. ohne zusätzliche Eingabe eines Passworts oder anderer benutzerbezogenen Daten. Um das einzurichten muss auf dem Client Rechner ein SSH Schlüssel erstellt werden. Unter Linux erfolgt dies in der Konsole mit dem folgenden Kommando:

```
ssh-keygen -t rsa -b 2048
```

Der generierte öffentliche Schlüssel wird im HOME Verzeichnis des zugehörigen Benutzers unter `.ssh/id_rsa.pub` abgelegt. Um die Authentifizierung im LRZ GitLab einzurichten, muss dort unter *Settings \ SSH Keys* der Inhalt der Datei `id_rsa.pub` eingefügt und mit *Add key* aktiviert werden.

3 GitLab und Git anwenden

In diesem Kapitel werden die Grundlagen zum Arbeiten mit GitLab und Git beschrieben. Der gesamte Funktionsumfang dieser beiden Systeme wird meist erst in größeren Entwicklungsprojekten erforderlich und kann in der Anwendung je nach Anwendungsfall angepasst werden. Die in diesem Dokument beschriebenen Funktionen dienen der Einarbeitung in diese Systeme einerseits und der Anwendung im Rahmen von Lehrveranstaltungen andererseits. Weiterführende Informationen zu Git und dessen Einsatz in größeren Projekten und Teams können aus der Literatur [2] entnommen werden.

3.1 Übersicht eines typischen Setups

In einem Entwicklungsprojekt sind üblicherweise mehrere Personen und mehrere verschiedene Rollen (Entwickler, Projektleiter, Scrum Master, Product Owner uvm.) beteiligt. Je nach Vorhaben können dabei die Entwickler von verschiedenen Unternehmen, Abteilungen etc. gemeinsam an dem Softwareprojekt zusammenarbeiten. Für die Verwaltung des Projektes kommt meist ein zentrales Repository zum Einsatz, welches von jedem Entwickler lokal vorgehalten wird. Abbildung 3.1 zeigt dieses typische Setup graphisch.

3.2 Vorgehensweise in GitLab

GitLab wird für die Verwaltung von Softwareprojekten eingesetzt. Dabei kann jeder Benutzer eine beliebige Anzahl von

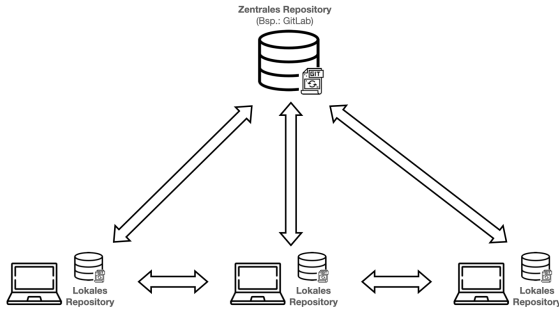


Abbildung 3.1: Ein typisches Entwicklungssetup

voneinander entkoppelten Projekten in GitLab anlegen und konfigurieren.

3.2.1 Ein GitLab Projekt erstellen

Jeder Benutzer kann mehrere eigene neue Projekte erstellen oder auch Projekte von bereits vorhandenen Projekten ableiten (Fork erstellen).

3.2.1.1 Neues GitLab Projekt anlegen

In GitLab kann mit wenigen Schritten ein neues Projekt angelegt werden. Dazu sind die folgenden Schritte erforderlich:

1. Auf *Projects* Seite im persönlichen Account wechseln.
2. Mit einem Klick auf *New Project* öffnet sich die Eingabemaske, in die die folgenden Projektdetails eingetragen werden müssen:
 - **Project name:** Kurzer und prägnanter Name des Projektes. Am besten werden hier keine länder-spezifischen Zeichen (bspw. Umlaute) verwendet.

- **Project description:** Die Beschreibung ist zwar optional, dient jedoch zur Erläuterung von Zwecks und Inhalt des angelegten Projektes.
- **Visibility Level:** Bei der Sichtbarkeit geht es darum, den Zugriff auf das Projekt grundsätzlich einzugrenzen.
 - **Private:** Projekte sind für andere GitLab Benutzer nicht sichtbar. Somit hat kein anderer Benutzer Zugriff auf das Projekt.
 - **Internal:** Jeder GitLab Benutzer kann grundsätzlich auf das Projekt zugreifen (lesen).



Die Einstellung **Private** sollte die Standardeinstellung für neue Projekte sein.

- **Initialize repository with a README:** Diese Option sollte gewählt werden, um mit der Projekterstellung ein Readme Markdown anzulegen.
3. Mit einem abschließenden Klick auf *Create project* wird das Projekt gemäß den gewählten Einstellungen angelegt.
 4. Auf der Projektseite kann über den Menüpunkt *Project Information / Members* der Zugriff auf das Projekt für andere GitLab Benutzer eingerichtet werden. Weitere Details dazu stehen in Abschnitt [3.2.2](#).

3.2.1.2 Fork von einem GitLab Projekt erstellen

Bei großen Open Source Projekten hört man häufig davon, dass ein sog. Fork eines Projektes erstellt wurde, weil sich die Entwickler nicht mehr auf eine gemeinsame Entwicklungsstrategie einigen konnten. So hat sich bspw. das relationale

Datenbankmanagementsystem MariaDB im Jahr 2009 von MySQL abgespalten. Ein Fork muss dabei nicht immer mit einer Auseinandersetzung begründet sein, sondern dient letztendlich dazu, eine Kopie des bisherigen Standes von einem existierenden Repository in ein neues Repository zu übernehmen.



In der Lehrveranstaltung wird der Fork des Kurs Repositories dazu verwendet, dass unter allen Studierenden eine einheitliche Repository-Struktur sichergestellt ist. Dadurch können Unterlagen und Programme leichter zur Verfügung gestellt werden und es wird zugleich die Grundlage für CI/CD gelegt.

GitLab bietet die Möglichkeit, von jedem verfügbaren (Sichtbarkeit vorausgesetzt) Repository einen Fork zu erstellen. Dazu muss auf der Projektseite rechts oben auf den Knopf *Fork* geklickt werden. Im nächsten Schritt muss noch der sog. *namespace* oder auch *Groups and subgroups* ausgewählt werden. Nun wurde eine Kopie von dem Ausgangsprojekt erzeugt und der *Owner* des Projektes neu gesetzt. Die Einstellungen zur Sichtbarkeit und die Zugriffssteuerung über *Members* können nun für dieses Projekt komplett losgelöst von dem ursprünglichen Projekt gesetzt werden.

Fork eines GitLab Projektes erstellen.



Als Beispielprojekt kann das Sample Projekt mit der URL <https://gitlab.lrz.de/lv/sample> verwendet werden.

Hinweis: Verwenden Sie dieses Repository nur für eigene Übungszwecke und nicht für die Lehrveranstaltung. Für die Lehrveranstaltung werden separate Repositories verwendet. Die URL können Sie beim jeweiligen Betreuer erfragen bzw. dem Moodle-Kurs entnehmen.

3.2.2 Zugriffsrechte eines Projektes

Die Steuerung der Zugriffsrechte auf ein Projekt erfolgt in GitLab auf der Basis von verschiedenen Rollendefinitionen. Alle Rechte für das Projekt werden stets dem *Owner*, d.h. dem Ersteller des Projektes eingeräumt. Darüberhinaus werden die folgenden vier weiteren Rollen definiert:

- **Maintainer:** Diese Rolle beschreibt das höchste Rechte-Level auf Projektebene. Lediglich grundlegende Einstellungen des Projektes (bspw. Sichtbarkeit, Namensänderung) bleiben dem *Owner* vorbehalten.
- **Developer:** Bei der Zusammenarbeit mit anderen Entwicklern sollten alle beteiligten Entwickler diese Rolle erhalten.
- **Reporter:** Diese Rolle erhält deutlich weniger Rechte als der Entwickler. So können Benutzer mit dieser Rolle keine Entwicklungszweige erstellen. Tester können diese Rolle erhalten.
- **Guest:** Ein Gast besitzt auf dem Projekt nur Leserechte.

Weitere Details über die Rechtestruktur können unter <https://gitlab.lrz.de/help/user/permissions> im Detail nachgelesen werden.



Für Lehrveranstaltungen müssen dem Dozenten zur Interaktion *Maintainer* Rechte auf das Projekt eingeräumt werden.

3.2.3 Dokumentation in GitLab

Es empfiehlt sich, zu jedem Projekt eine README.md Datei mit im Wurzelverzeichnis des Projektes anzulegen. Bei der Betrachtung des Projektes auf der GitLab Seite wird der Inhalt

Tool	Linux	Windows	Mac OS	Online
Typora	x	x	x	
MacDown			x	
Remarkable	x			
ghostwriter	x	x		
StackEdit				x
HackMD				x

Tabelle 3.1: Auswahl einiger Markdown Editoren

dieser Datei gemäß der Markdown¹ Syntax interpretiert und im Browser dargestellt. In dieser Datei werden üblicherweise Informationen über das Projekt, technische Abhängigkeiten, Anwendungsmöglichkeiten etc. beschrieben.

Für die Erstellung und Darstellung von Markdown Dateien bringen viele Entwicklungsumgebungen bereits eigene Erweiterungen mit. Alternativ können Markdown Dateien mit speziellen Editoren erstellt und betrachtet werden. Eine kleine Übersicht von kostenlosen Tools ist in Tabelle 3.1 dargestellt.

3.3 Einführung in gängigen Git Workflow

Die Verwendung der Git Versionsverwaltung erfolgt über den lokalen Git Client. In diesem Abschnitt werden die Funktionen der wichtigsten Git Kommandos und deren Verwendung in der Konsole näher erläutert. Die Details der Kommandos werden bei graphischen Clients (Standalone, Erweiterung in Entwicklungsumgebung) weitestgehend abstrahiert. Eine Dokumentation umfassende der Kommandos kann unter [3] nachgeschlagen werden.

¹Markdown bezeichnet eine einfache Auszeichnungssprache, um ein formatiertes Dokument zu erstellen. Mehr Informationen zur Sprache und Syntax sind unter <https://www.markdown.de> verfügbar.



Bei jedem Git Repository gibt es zwei wichtige Standard-Begrifflichkeiten. Der Name *origin* ist die Standardbezeichnung für das entfernte (zentrale) Repository, von dem bspw. ein lokaler Klon erzeugt wurde. Mit dem Namen *main* wird üblicherweise der Hauptentwicklungszweig bezeichnet.

Hinweis: Dieser Name wurde aus Gründen der Anti-Diskriminierung von *master* in *main* umbenannt. Es gibt jedoch noch weiterhin zahlreiche Repositories bei denen *master* verwendet wird.

In Abbildung 3.2 sind fünf grundsätzliche Git Workflows graphisch sowie die verschiedenen Bereiche eines Repositories dargestellt. Die Initialisierung eines Repositories erfolgt üblicherweise einmalig zu Beginn, da anschließend eine lokale Kopie des zentralen Repositories vorliegt. Während der Entwicklung werden üblicherweise die Kommandos für Aktualisierungen und Änderungen verwendet. Typische Git Use Cases werden in Abschnitt 3.5 näher beschrieben.

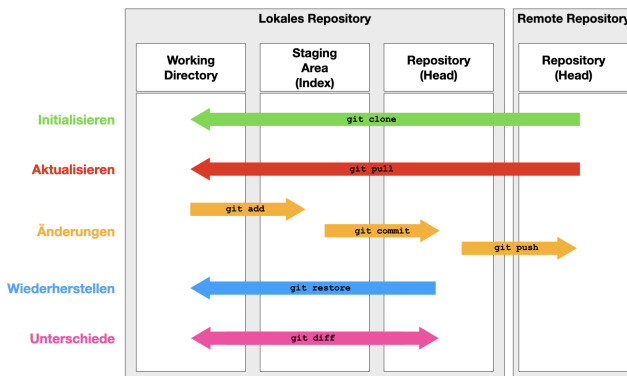


Abbildung 3.2: Übersicht verschiedener Workflows in Git

3.3.1 Projekt anlegen

Die einfachste Möglichkeit, mit einem Git Repository zu arbeiten besteht darin, einen Klon von einem bereits existierenden (meist zentralen) Repository anzulegen. Neben einer existierenden Netzwerkverbindung müssen insbesondere die technischen Voraussetzungen aus Kapitel 2 erfüllt sein.

Damit ein Klon eines Projektes erzeugt werden kann muss die Zugriffsadresse (URL) des Projektes bekannt sein. Unter GitLab kann diese Adresse auf jeder verfügbaren und sichtbaren Projektseite unter *Clone* abgerufen werden. Es wird dabei zwischen SSH und HTTPS unterschieden. Für den Zugriff mit HTTPS werden Benutzername und Passwort zu dem zugehörigen clone Kommando benötigt:

Klon eines vorhandenen Repositories erzeugen.

```
git clone https://gitlab.lrz.de/lv/sample.git
```

Mit diesem Kommando wird ein vollständiger Klon (Kopie) des entfernten Repositories auf dem lokalen Rechner erstellt. Die anschließende Ausgabe auf der Konsole lautet wie folgt:

```
Cloning into 'sample'...
Username for 'https://gitlab.lrz.de': hm-test
Password for 'https://hm-test@gitlab.lrz.de':
remote: Enumerating objects: 26, done.
remote: Counting objects: 100% (26/26), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 26 (delta 10), reused 0 (delta 0),
pack-reused 0
Unpacking objects: 100% (26/26), done.
```



Das *sample* Repository kann zum Test verwendet werden. Es ist jedoch nicht das Repository der Lehrveranstaltung.

Neues Repository anlegen

Wenn wie bisher angenommen kein Git Repository vorhanden ist, muss für die Versionierung von Dateien ein neues Repository initial erstellt werden. Dabei kann ein zentrales Repository bspw. mit Hilfe von GitLab erstellt werden, wie in Abschnitt 3.2.1 beschrieben. Der Git Client unterstützt jedoch auch die Möglichkeit, direkt ein neues (lokales) Repository anzulegen. Dazu muss in der Konsole wie folgt vorgegangen werden:

- Verzeichnis anlegen, in dem ein Git Repository angelegt werden soll.
- Im Verzeichnis das Kommando `git init` ausführen, um das Git Repository zu initialisieren.

In dem Verzeichnis wurde nun ein `.git` Unterverzeichnis angelegt und somit das Git Repository erzeugt. Von nun an können Dateien und Verzeichnisse versioniert werden.

3.3.2 Dateien im Repository verwalten

Git verwaltet seine Objekte als Schlüssel-Wert-Paare. So wird bspw. jeder Datei ein Schlüssel (Hash) zugeordnet, die intern in einer Baumstruktur als Datenmodell organisiert werden. Um Dateien in einem Repository verwalten zu können, müssen diese über den Git Client mit dem Kommando `git add <file>` dem Index hinzugefügt werden. Der Git Index bezeichnet eine sog. Staging Area zwischen dem Arbeitsverzeichnis und dem Repository. Der Index wird dazu verwendet, um eine Menge an Änderungen zusammenzufassen, die anschließend dem Repository hinzugefügt werden sollen. Die folgenden Beispiele zeigen ein paar Verwendungsmöglichkeiten des `add` Kommandos:

Dateien dem Repository hinzufügen.

```
git add file1
git add file2 file3
```

```
git add directory
git add *.c
```

Genauso wie Dateien über den Index hinzugefügt, können diese auch über den Index entfernt werden. Dazu muss das Kommando `git rm <file>` angewendet werden. Die folgenden Beispiele zeigen ein paar Verwendungsmöglichkeiten des `rm` Kommandos:

Dateien aus dem Repository entfernen.

```
git rm file3
git rm main.c
```



Wenn eine Datei nur aus dem Index gelöscht, aber im Dateisystem erhalten bleiben soll, kann dies über die Option `--cached` erreicht werden:

```
git rm --cached file2
```

Die Änderungen `add`, `rm` befinden sich in dem Git Index und noch nicht im Repository. Der Index ist eine Zwischenstufe, der die Menge an Änderungen am Repository bündelt bevor diese über einen `commit` übermittelt werden.

3.3.3 Änderungen ins Repository übertragen

Nachdem alle Änderungen im Git Index zusammengefasst wurden können diese über das `commit` Kommando ins Repository übertragen und somit eine neue Version erstellt werden. Dazu muss in der Konsole folgender Aufruf ausgeführt werden:

Inhalt einer Datei modifizieren und anschließend mit `commit` Änderungen protokollieren

```
git commit
```

Beim Git Konsolen Client öffnet sich anschließend ein Editor, in dem eine aussagekräftige Änderungsmeldung (`commit`

message) eingegeben werden muss. Anschließend muss der Editor beendet und die eingegebene Meldung gespeichert werden. Nachdem der Editor beendet wurde sendet der Git Client die Änderungen aus dem Git Index ans Repository.



Unter Linux ist als Standardeditor meist *vi* ausgewählt. Über die Git Konfiguration kann dieser jederzeit geändert werden. Der *nano* Editor kann über das Kommando:
`git config --global core.editor nano`
eingestellt werden.



Die *commit message* kann auch ohne Verwendung eines Editors, nämlich über einen Kommandozeilenparameter angegeben werden. Das dazugehörige Kommando in der Konsole lautet wie folgt:
`git commit -m "commit message"`

3.3.4 Dateien von Versionierung ausschließen

Beim Arbeiten mit der Versionsverwaltung werden grundsätzlich alle im Arbeitsverzeichnis vorhandenen Dateien und Verzeichnisse berücksichtigt. Das führt bei der täglichen Arbeit dazu, dass bei Statusabfragen oder Dateinamenexpansion (*) auch Dateien mit berücksichtigt werden, die grundsätzlich nicht unter Versionsverwaltung gestellt werden sollten. Dazu zählen u.a. temporäre Dateien (bspw. Dateien die mit ~ beginnen), Zwischenformatdateien (bspw. Object-Dateien *.o) und insbesondere Binärdateien (bspw. *.exe unter Windows).

Git bietet mit Hilfe des *.gitignore* Mechanismus eine Möglichkeit, bestimmte Dateien von der Versionierung auszuklamern. Dazu muss in dem Hauptverzeichnis der versionierten Dateien eine *.gitignore* Datei angelegt werden, in dem zeilen-

weise die zu ignorierenden Dateien beschrieben werden. Die `.gitignore` Datei muss anschließend ebenfalls zum Repository hinzugefügt werden. Eine `.gitignore` Datei kann beispielsweise wie folgt aufgebaut sein:

```
# ignore all html files
*.html
# except index.html
!index.html
# ignore object file (*.o) and archives (*.a)
*.[oa]
# Mac OS files
.DS_Store
```

Die `.gitignore` Datei muss nun zum Repository hinzugefügt werden. Dazu sind die folgenden Kommandos erforderlich:

Festlegen der zu ignorierenden Dateien

```
git add .gitignore
git commit -a -m "gitignore file added"
```

Alle bereits versionierten Dateien, die der Definition in der `.gitignore` Datei entsprechen werden nicht automatisch entfernt, sondern bleiben weiterhin unter Versionsverwaltung.



Die zu ignorierenden Dateien und Verzeichnisse hängen stark von den eingesetzten Technologien, der Programmiersprache und dem Betriebssystem ab.

3.3.5 Synchronisation mit zentralem Repository

Bei den zentralen Versionsverwaltungssystemen können Änderungen stets in einer lokalen Kopie des zentralen Repositories verwaltet werden. Je nach Zusammenarbeit mit anderen Entwicklern müssen die lokalen Änderungen immer wieder in das zentrale Repository geschoben werden. Dabei sollte der

zeitliche Abstand und die Menge der Änderungen zwischen den Synchronisationen möglichst klein gehalten werden, um auch potentielle Konflikte gering zu halten.

Änderungen vom zentralen Repository holen

Es empfiehlt sich, vor jedem Start der Entwicklungsarbeiten den lokalen Stand des Repositories zu aktualisieren. Dazu muss lediglich das *pull* Kommando ausgeführt werden. Der dazugehörige Konsolenaufruf lautet wie folgt:

Neuesten Stand vom zentralen Repository in die lokale Kopie holen

```
git pull
```

Sofern Änderungen zu dem lokalen Entwicklungsstand vorhanden sind, werden alle Abweichungen vom Server geladen und mit den lokalen Versionen verschmolzen. Dabei kann es zu Verschmelzungen (*merge*) zwischen zentralen und lokalen Dateiversionen kommen. Wenn es dabei zu keinem Konflikt kommt, erfolgt das Zusammenführen vollständig automatisch und ein sog. *Merge-Commit* wird erstellt. Andernfalls muss der Konflikt manuell aufgelöst werden. Mit Hilfe des *diff* Kommandos (Abschnitt 3.3.10) von Git können die Unterschiede analysiert und potentielle Konflikte gelöst werden.



Das *pull*-Kommando kombiniert das Holen der Änderungen (*fetch* Kommando) und das Verschmelzen.

Änderungen ins zentrale Repository schieben

Neben der Aktualisierung des lokalen Repositories stellt besonders das Bereitstellen der Änderungen im zentralen Repository eine wichtige Aufgabe im Git Workflow dar. Bei dem Abgleich werden alle lokalen versionierten Änderungen (über *commit* hinzugefügt) auf das zentrale Repository übertragen. In Analogie zum *pull* Kommando führt das *push* Kommando die Übertragung ins zentrale Repository durch. Der dazugehörige Konsolenaufruf lautet wie folgt:

Lokal durchgeführte Änderungen in das zentrale Repository übertragen

```
git push
```

Um Konflikte beim Abgleich zu vermeiden muss vor jedem *push* in das zentrale Repository ein *pull* ausgeführt werden. Der Git Client verwehrt den Abgleich, wenn vorher kein *pull* durchgeführt wurde.



Für Lehrveranstaltung müssen die finalen Entwicklungsstände stets in das zentrale Repository geschoben werden, da nur so der Betreuer das Ergebnis einsehen und bewerten kann.

3.3.6 Lokale Änderungen zurücksetzen

Bei der Entwicklung kommt es manchmal vor, dass lokal durchgeführte Änderungen an einer Datei zurückgenommen bzw. verworfen werden müssen. Es kann auch sein, dass eine Datei versehentlich lokal gelöscht wurde. Um zu erreichen, dass der letzte versionierte Stand aus dem Repository wiederhergestellt wird, kann das *restore* Kommando genutzt werden:

```
git restore main.cpp
```

3.3.7 Entferntes Repository hinzufügen

Ein weiterer Vorteil des dezentralen Git Versionsverwaltungssystems ist, dass es nicht nur mit einem zentralen Repository betrieben werden kann. Git kann mehrere, sog. *remote* Repositories integrieren. Bei einem geklonten Repository wird standardmäßig mit Hilfe des Namens *origin* auf das ursprüngliche (entfernte) Repository verwiesen. Dieses und jedes weitere hinzugefügte *remote* Repository kann mit dem folgenden Kommando abgefragt werden:

```
git remote -v
```

Um ein *remote* Repository hinzuzufügen wird lediglich die *<URL>* des entfernten Repositories benötigt. Der *<NAME>* des entfernten Repositories kann dabei frei gewählt werden.

Remote Repository dem eigenen Repository hinzufügen

```
git remote add <NAME> <URL>
```



In der Lehrveranstaltung wird zu Beginn ein sog. Fork von dem Kurs Repository erstellt. Dadurch erhalten alle Teilnehmer die identische einheitliche Struktur in dem eigenen Repository. Wenn nun jedoch in dem Kurs Repository Änderungen vorgenommen werden (Praktikumsaufgaben hinzugefügt etc.) so werden diese nicht mehr in dem studentischen Fork Repository übernommen. Um diese Aktualisierungen auch im dem studentischen Fork Repository zu erhalten, muss dem studentischen Repository das Kurs Repository als Remote Repository hinzugefügt werden. In der Lehrveranstaltung wird das Remote Repository mit dem Namen *upstream* bezeichnet.

Aktualisierungen aus dem *main* Branch können nun von jedem entfernten Repository mit dem *pull* Kommando abgeholt werden. Das dazu erforderliche Kommando lautet:

```
git pull <NAME> main
```

3.3.8 Statusinformationen abrufen

Mit zu den häufigsten Interaktionen mit der Versionsverwaltung zählt das Hinzufügen von Dateien und das Nachverfolgen von deren Änderungen. Der Status welche Dateien in der Staging-Umgebung sind, welche nicht und welche nicht verfolgt werden können mit dem folgenden Kommando abgefragt werden:

```
git status
```

Die Ausgabe listet den Status der Dateien in Bezug auf die Staging-Umgebung auf.

3.3.9 Historie der Änderungstexte abrufen

Über die Zeit der Entwicklung wächst die Historie eines Repositories entsprechend an. Der Zugriff auf die *commit* Snapshots kann mit folgendem Kommando erreicht werden:

```
git log
```

Die Ausgabe des Kommandos zeigt u.a. den *commit Hash*, den Autor, das Datum sowie die *commit* Nachricht.

3.3.10 Versionsunterschiede abfragen

Die Veränderungen an einer Datei, einer Funktion in einer Quelltextdatei werden mit dem Versionsverwaltungssystem protokolliert. Regelmäßige *commit* stellen dabei ein transparentes Abbild der Veränderungen dar. In vielen Situationen ist es erforderlich, die Veränderungen einer Datei zwischen zwei Versionen zu visualisieren. So bspw. um Konflikte aufzulösen oder eine mögliche Ursache für einen Fehler in einem Programm zu analysieren. Der Git Client kann in der Konsole über das folgende Kommando den Unterschied einer Datei in der letzten beiden Versionen abrufen:

```
git diff main.cpp
```

Die Ausgabe kann insbesondere bei längeren Dateien und mehreren Änderungen sehr schwer lesbar sein. Aus diesem Grund empfiehlt sich für das *diff* Kommando die Verwendung eines graphischen Git Clients, da diese durch Hervorhebungen und verschiedene Farbgebungen die Unterschiede sehr

gut visualisieren können. In Abbildung 3.3 ist ein Beispiel dargestellt, in dem entfernte Zeilen in rot und hinzugefügte Zeilen in grün hervorgehoben sind.



```
src/ikomponente.cpp
Hunk 1 : Lines 1-6
1 1  #include "ikomponente.h"
2 2
3 3  - IKomponente::~IKomponente() {
4 4  + IKomponente::~IKomponente()
5 5  + {
6 6  }
```

Abbildung 3.3: Unterschied zwischen zwei Dateiversionen

3.4 Einführung in CI/CD

Im Softwareentwicklungsprozess wird Continuous Integration (CI) dazu eingesetzt, eine Software an einem zentralen Ort kontinuierlich zu erstellen und automatisiert zu testen. Bei der Entwicklung von Softwaremodulen oder ganzen Applikationen gibt es mehrere Änderungen am Tag wodurch vorhandene Funktionalität negativ beeinflusst werden kann. Fehler dieser Art müssen schnell entdeckt und behoben werden bevor diese in die Produktivumgebung (zum Kunden) ausgeliefert werden. CI-Systeme sorgen dafür, dass Software mit jeder Änderung automatisiert erstellt und getestet werden. Zusätzlich können weitere Überprüfungen wie die Einhaltung von Programmierrichtlinien, Standards oder statische Codeanalysen mit in das CI System integriert werden.

Der nächste Schritte der Automatisierung wird Continuous Delivery (CD) bezeichnet und setzt auf CI auf. Dabei wird der neue Softwarestand nicht nur erstellt und getestet, sondern die Auslieferung (Delivery) in das Produktivsystem wird so-

weit vorbereitet, dass es über einen manuellen Mechanismus übertragen (Deployment) werden kann.

Die Abkürzung CD ist doppeldeutig und kann auch für Continuous Deployment stehen. Hierbei wird die Übertragung in das Produktivsystem ebenfalls automatisiert. Dieser Mechanismus kommt beispielsweise bei Facebook oder Amazon zum Einsatz. Das CI/CD System von Amazon liefert bereits im Jahr 2013 alle 11,6 Sekunden ein Deployment in die Produktivumgebung aus. Neben dem CI/CD System stellt dies natürlich auch entsprechende Anforderungen an die Softwarearchitektur und die darunterliegende Plattform. Jedoch wird daran sehr deutlich, welche Vorteile in Bezug auf Qualität und Geschwindigkeit derartige Systeme liefern können.

In der Lehrveranstaltung wird das CI/CD System von GitLab verwendet, um den Kursteilnehmern diese Technologie (nebenbei) eines zeitgemäßen Entwicklungsprozesses vorzustellen und zugleich stellt es eine Unterstützung bei der Bearbeitung von Programmieraufgaben dar. Das CI/CD System kann die Programmieraufgaben automatisiert erstellen (am Beispiel von C) und die vorhandenen Unit Tests automatisiert ausführen, so dass eine für den Kursteilnehmer eigenständige Überprüfung der Funktionsfähigkeit der Lösung möglich ist. Dieser Ansatz wird auch Test-Driven Development (TDD) genannt. Die Testergebnisse werden nach der unmittelbaren Ausführung direkt in GitLab zur Verfügung gestellt.



Die CI/CD Funktionalität ist in der LRZ GitLab Installation nicht automatisch aktiviert. In einem GitLab Projekt kann CI/CD über Settings / General / Visibility, project features, permissions aktiviert werden.

3.4.1 Konfiguration von CI/CD

In diesem Abschnitt wird die Konfiguration von CI/CD beschrieben.

3.4.1.1 Konfiguration GitLab Runner

Die GitLab Installation des LRZ stellt die CI/CD Funktionalität zur Verfügung. Für die Ausführung wird eine Laufzeitumgebung benötigt, der sog. GitLab Runner, die durch das LRZ nicht zur Verfügung gestellt wird.



Für die Lehrveranstaltung stellt die Fakultät 04 einen GitLab Runner zur Verfügung. Die Konfiguration erfolgt automatisch durch den Betreuer, d.h. der Kurs Teilnehmer muss keine eigenen Einstellungen im Projekt vornehmen.

Um CI/CD außerhalb der Lehrveranstaltung mit einem eigenen GitLab Runner zu nutzen, muss ein Runner installiert und konfiguriert werden. Unter <https://docs.gitlab.com/runner/register/> wird dies näher beschrieben.

3.4.1.2 Konfiguration CI/CD Pipeline

Eine Software kann in verschiedenen Programmiersprache entwickelt und je nach Anforderungen auch unterschiedlich strukturiert sein. Um nun den Erstell- und Testprozess in einem CI/CD System einzurichten werden sog. Pipelines verwendet. Eine Pipeline ist eine abstrakte Komponente für CI/CD und ermöglicht die Konfiguration der auszuführenden Jobs (was ausgeführt wird) und deren Strukturierung in Form von Stages (wann etwas ausgeführt wird). Dazu wird in

GitLab die `.gitlab-ci.yml`² Datei verwendet, in der die Konfiguration der Pipeline textuell erfolgt. Die Pipelines sind ein sehr mächtiges Werkzeug, die über sog. Schlüsselwörter (keywords) konfiguriert werden. Unter <https://docs.gitlab.com/ee/ci/yaml/> ist die Beschreibung der einzelnen Schlüsselwörter dokumentiert.



Für die Lehrveranstaltung übernimmt der Betreuer die Konfiguration der Pipeline, d.h. der Kursteilnehmer muss keine eigenen Konfigurationen im Projekt erstellen.

3.4.2 Anwenden von CI/CD

Es gibt verschiedene Möglichkeiten der Konfiguration, um die CI/CD Funktionalität in GitLab zu aktivieren. In diesem Abschnitt wird vorausgesetzt, dass CI/CD mit GitLab Runner in einem Projekt vollständig konfiguriert ist.

3.4.2.1 Starten einer Pipeline

Die standardmäßig aktivierte Möglichkeit, eine Pipeline zu starten ist das Hochladen eines neuen Softwarestandes über `git push`. GitLab erkennt den Softwarestand und startet automatisiert die konfigurierte Pipeline. Auf der Projektseite in GitLab wird eine laufende Pipeline über eine blaue Fortschrittsuhr dargestellt, siehe Abbildung 3.4.

Nach Ablauf der Pipeline wird der Status als Icon dargestellt. Dabei werden u.a. die folgenden Statusinformationen unterschieden:

- **passed:** Pipeline wurde erfolgreich ausgeführt.

²YAML (YAML Ain't Markup Language) ist ein einfaches Datenserialisierungsformat. Damit lassen sich Listen und einzelne Werte beschreiben.

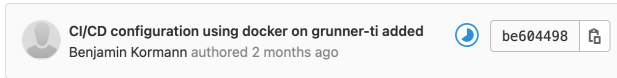


Abbildung 3.4: Pipeline wird ausgeführt

- **pending:** Pipeline wurde gestartet und wartet auf verfügbaren Runner zur Ausführung.
- **running:** Pipeline wird auf einem Runner ausgeführt.
- **failed:** Bei der Ausführung der Pipeline ist ein Job fehlgeschlagen.
- **cancelled:** Die Ausführung der Pipeline wurde (i.d.R. manuell) abgebrochen.



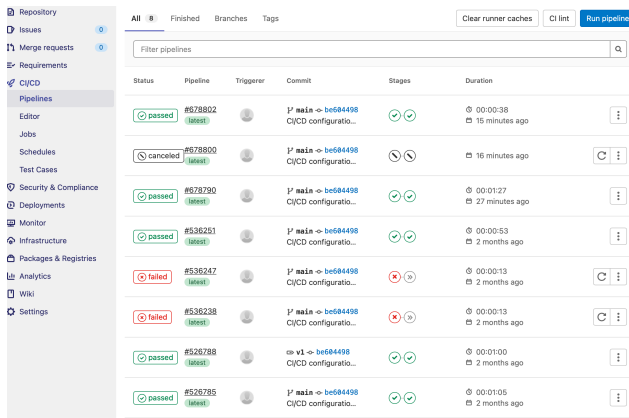
Im Rahmen der Lehrveranstaltung sollten im wesentlichen nur die Statusinformationen passed und failed eintreten. Bei passed konnte alle Schritte (Software erstellen und testen) erfolgreich durchgeführt werden. Bei failed ist min. ein Schritt fehlgeschlagen, d.h. es müssen Fehler behoben werden.

3.4.2.2 Testergebnisse einer Pipeline

GitLab archiviert alle Pipeline Ergebnisse. Meistens ist jedoch das Ergebnis des letzten Laufs von Interesse. Auf dieses Ergebnis kann über das CI/CD Menü und den Unterpunkt Pipelines zugegriffen werden, siehe Abbildung 3.5. Die Auflistung erfolgt hierbei umgekehrt chronologisch. Mit einem Klick auf das Status-Icon eines Laufs wird die Detailansicht des Laufs geöffnet. Abbildung 3.6 zeigt das Ergebnis für den letzten Lauf der Pipeline mit der ID #678802.

Abhängig vom Ergebnis des Testlaufs werden etwaige fehlgeschlagene Tests dargestellt und der Grund für den fehlge-

Die Ergebnisse der Unit Tests zu den Praktikumsaufgaben kann darüber eingesehen werden



	All	Finished	Branches	Tags	Clear runner caches	CI lint	Run pipeline
	Filter pipelines						
	Status	Pipeline	Trigger	Commit	Stages	Duration	
passed	#578802	latest	! master → be684498	CI/CD configuratio...	✓ ✓	00:00:38 15 minutes ago	
canceled	#578809	latest	! master → be684498	CI/CD configuratio...	✗ ✗	16 minutes ago	
passed	#578790	latest	! master → be684498	CI/CD configuratio...	✓ ✓	00:01:27 27 minutes ago	
passed	#536261	latest	! master → be684498	CI/CD configuratio...	✓ ✓	00:00:53 2 months ago	
failed	#536247	latest	! master → be684498	CI/CD configuratio...	✗ ✗	00:00:13 2 months ago	
failed	#536238	latest	! master → be684498	CI/CD configuratio...	✗ ✗	00:00:13 2 months ago	
passed	#526788	latest	on v1 → be684498	CI/CD configuratio...	✓ ✓	00:01:00 2 months ago	
passed	#526785	latest	! master → be684498	CI/CD configuratio...	✓ ✓	00:01:05 2 months ago	

Abbildung 3.5: Übersicht über alle Pipeline Läufe

schlagenen Test angezeigt. Mit dieser Information kann die Behebung des Fehlers gestartet werden.

3.4.2.3 Artefakte einer Pipeline

Jobs einer Pipeline können Ergebnisse (Dateien) als sog. Artefakte ablegen auf die über die Details eines Jobs in GitLab zugegriffen werden kann. Die Artefakte können in GitLab über das Menü CI/CD und dem Unterpunkt Jobs abgerufen werden. Die Auflistung zeigt alle ausgeführten Jobs der Pipeline. Mit einem Klick auf den Status des entsprechenden Jobs, bspw. Metrik öffnet die Detailansicht der Job Ausführung. Auf der rechten Seite gibt es einen Abschnitt mit dem Namen *Job artifacts*. Mit einem Klick auf den Button *Browse* kann auf alle Artefakte des Jobs zugegriffen werden.

So können die Ergebnisse der Metriken (bspw. Dokumentationsgrad) zu den Praktikumsaufgaben abgerufen werden

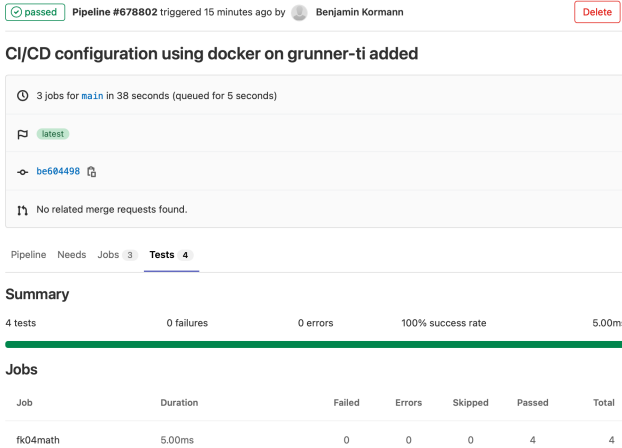


Abbildung 3.6: Testergebnisse eines Pipeline Laufs



Im Rahmen der Lehrveranstaltung kann der Dokumentationsgrad der Quelltextdateien überprüft werden. Diese Auswertungen sind über die Artefakte des Jobs verfügbar.

3.5 Typische Git Use Cases

Nachdem die Grundkonfiguration wie in Abschnitt sec:config-git beschrieben abgeschlossen ist kann mit Git Repositories gearbeitet werden. In diesem Abschnitt sind typische Git Use Cases beschrieben, die im Rahmen der Lehrveranstaltung benötigt werden.

3.5.1 Grundlagen

Um grundsätzlich mit einem Git Repository arbeiten zu können, müssen ein paar essentielle Kommandos verwendet werden. Zur besseren Einordnung dient hierbei Abbildung 3.2.

3.5.1.1 Existierendes Repository klonen

Um auf ein existierenden Git Repository in GitLab lokal zugreifen zu können muss dieses geklont werden. Der genaue Ablauf wurde bereits in Abschnitt 3.3.1 näher beschrieben. In Zeile 1 wird dem clone Kommando die URL des Repositories übergeben. Nachdem das Repository erfolgreich geklont wurde kann man wie in Zeile 10 beschrieben in das Verzeichnis wechseln und mit den Dateien arbeiten.

Die URL für das studentische Repository kann dem GitLab Projekt, wie in Abschnitt 3.3.1 beschrieben, entnommen werden

```
1 $ git clone git@gitlab.lrz.de:lv/git-use-cases.git
2 Cloning into 'git-use-cases'...
3 remote: Enumerating objects: 32, done.
4 remote: Counting objects: 100% (32/32), done.
5 remote: Compressing objects: 100% (19/19), done.
6 remote: Total 32 (delta 4), reused 0 (delta 0), pack-reused 0
7 Receiving objects: 100% (32/32), done.
8 Resolving deltas: 100% (4/4), done.
9
10 $ cd git-use-cases
11
12 $ ls
13 README.txt
```



Die Repository URL in dem `git@.....git` Format nutzt zur Authentifizierung die SSH Schlüssel. Sollten diese nicht richtig, wie in Abschnitt 2.3 beschrieben, eingerichtet worden sein, so scheitert dieser Aufruf.

3.5.1.2 Dateiveränderungen versionieren

Die häufigste Interaktion mit der Versionsverwaltung beschäftigt sich mit dem verändern, hinzufügen, entfernen und versionieren von Dateien und Verzeichnissen. In dem folgenden Beispiel werden diese Interaktionen durchgeführt. Zeile 1 verändert die Datei README.txt beispielhaft. Diese Änderung muss nur mit dem `add` Kommando der Staging Area hinzugefügt werden. Im Anschluss daran können alle Änderungen aus der Staging Area in das lokale Repository übernommen

werden. Dazu wird wie in Zeile 5 gezeigt, das *commit* Kommando mit einem Kommentar abgesetzt. Nun befindet sich diese Datei unter Versionsverwaltung.

Im weiteren Verlauf wird eine neue Datei namens *hello.txt* der Staging Area hinzugefügt (Zeile 11) und die Datei *README.txt* wird über das *rm* Kommando entfernt. Damit beide Änderungen (hinzufügen von *hello.txt* und entfernen von *README.txt*) auch in der Versionsverwaltung (lokales Repository) wirksam werden, muss dies mit einem *commit* Kommando (siehe Zeile 16) abgeschlossen werden. Die anschließende Ausgabe belegt, dass zwei Dateien verändert (*README.txt* gelöscht, *hello.txt* hinzugefügt) wurden.

```
1 $ echo "Das ist ein Text" > README.txt
2
3 $ git add README.txt
4
5 $ git commit -m "README.txt verändert"
6 [main e5f8404] README.txt verändert
7 1 file changed, 1 insertion(+), 3 deletions(-)
8
9 $ echo "Hello World" > hello.txt
10
11 $ git add hello.txt
12
13 $ git rm README.txt
14 rm 'README.txt'
15
16 $ git commit -m "Datei entfernt"
17 [main 66047e5] Datei entfernt
18 2 files changed, 1 insertion(+), 1 deletion(-)
19 delete mode 100644 README.txt
20 create mode 100644 hello.txt
```

3.5.1.3 Dateien ignorieren

In der Entwicklung geschieht es häufig, dass Zwischenformatdateien erstellt werden, die nicht in die Versionsverwaltung aufgenommen werden sollten. Der grundsätzliche Mechanismus ist bereits in Abschnitt 3.3.4 beschrieben. In dem folgenden Beispiel werden sämtlich **.o* Dateien ignoriert, indem ein Zeileneintrag in der *.gitignore* Datei hinzugefügt (Zeile 13 und 15) wird.

```
1 $ cd git-use-cases
2
3 $ git status
4 On branch main
5 Your branch is up to date with 'origin/main'.
6
7 Untracked files:
8 (use "git add <file>..." to include in what will be committed)
9 helloworld.o
10
11 nothing added to commit but untracked files present (use "git add" to track)
12
13 $ echo *.o >> .gitignore
14
15 $ git add .gitignore ; git commit -m "gitignore"
16 [main 742fb38] gitignore
17 1 file changed, 1 insertion(+)
18 create mode 100644 .gitignore
19
20 $ git status
21 On branch main
22 Your branch is up to date with 'origin/main'.
23
24 nothing to commit, working tree clean
```

3.5.2 Zentrales Repository

Die in GitLab verwalteten Projekte können als ein zentrales Repository betrachtet werden. Während der Entwicklung muss zur Bearbeitung der in Git verwalteten Dateien immer ein lokaler Klon erstellt werden. Ein wesentlicher Vorteil eines zentralen Repositorys ist es, dass man mehrere lokale Kopien (Klone) auf unterschiedlichen Rechnern haben kann oder auch im Team an einem Projekt gemeinsam arbeiten kann. Jedoch müssen die Veränderungen auch stets synchronisiert werden, d.h. vom zentralen Repository zu dem oder den lokalen Repositorys sowie umgekehrt, alle Änderungen von den lokalen zu dem zentralen Repository übertragen werden.

Mit Hilfe des *pull* Kommandos wird der aktuelle Stand aus dem zentralen Repository in das lokale Repository übertragen. Dabei kann es zu mindestens drei verschiedenen Szenarien kommen, die im Folgenden näher betrachtet werden.

3.5.2.1 Ablauf bei der Arbeit mit zentralem Repository

Sollten im zentralen Repository Änderungen zum lokalen Stand erfolgt sein, diese jedoch keinen Widerspruch zum lokalen Stand darstellen, übernimmt Git das Zusammenführen der zwei Stände automatisch mit Hilfe eines fast-forward. Dazu muss lediglich das *pull* Kommando (Zeile 1) ausgeführt werden. Nach einigen Änderungen im lokalen Repository (Zeile 14) müssen diese zurück an das zentrale Repository übertragen werden. Dies erfolgt mit Hilfe des *push* Kommandos, siehe Zeile 18. Im Anschluss daran sind der lokale und zentrale Stand identisch.

```
1 $ git pull
2 remote: Enumerating objects: 5, done.
3 remote: Counting objects: 100% (5/5), done.
4 remote: Compressing objects: 100% (2/2), done.
5 remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
6 Unpacking objects: 100% (3/3), done.
7 From gitlab.lrz.de:lv/git-use-cases
8   742fb38..ee4ab17  main       -> origin/main
9 Updating 742fb38..ee4ab17
10 Fast-forward
11   hello.txt | 1 +
12   1 file changed, 1 insertion(+)
13
14 $ echo "Hi" > hello.txt ; git add . ; git commit -m "hello.txt"
15 [main 59310e6] hello.txt
16   1 file changed, 1 insertion(+), 2 deletions(-)
17
18 $ git push
19 Enumerating objects: 5, done.
20 Counting objects: 100% (5/5), done.
21 Delta compression using up to 12 threads
22 Compressing objects: 100% (2/2), done.
23 Writing objects: 100% (3/3), 323 bytes | 323.00 KiB/s, done.
24 Total 3 (delta 0), reused 1 (delta 0)
25 To gitlab.lrz.de:lv/git-use-cases.git
26   ee4ab17..59310e6  main -> main
```

3.5.2.2 Nebenläufiges Arbeiten, Konflikte vermeiden

Bei der Verwendung einer Versionsverwaltung kann nebenläufiges Arbeiten schnell zu Dateikonflikten führen. Dabei ist es unerheblich, ob dieser Konflikt durch einen oder mehrere Benutzer entsteht. Grundlage des Konflikts sind stets mindestens zwei lokale (geklonte) Repositories eines gemeinsamen

Remote Repositories. In Abbildung 3.7 ist der Ablauf einer Interaktion zweier lokaler Repositories dargestellt, so dass es zu keinem Konflikt führt. Der Konflikt wird dadurch vermieden, da auf dem Repository B mit *git pull* die kürzlich aus Repository A hinzugefügt Änderungen ins lokale Repository übertragen werden. Ein Konflikt kann grundsätzlich nur dann entstehen, wenn in mind. zwei Repositories widersprüchliche Änderungen an ein und derselben Datei vorgenommen werden.

Im weiteren Verlauf wird dies dadurch demonstriert, indem der Inhalt der Datei *hello.txt* stets überschrieben wird. Einmal mit dem Inhalt *Hallo* und ein weiteres Mal mit dem Inhalt *Hi*.

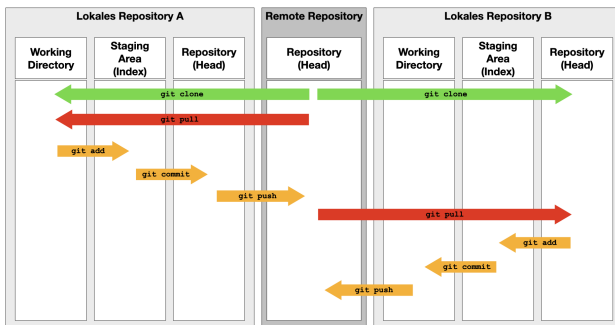


Abbildung 3.7: Repository-Interaktion ohne Konflikt

Die zu dem Ablauf gehörenden git Kommandos aus Sicht des lokalen Repositories A:

```

1 $ git pull
2 ...
3
4 $ echo "Hallo" > hello.txt
5 $ git add .
6 $ git commit -m "hello.txt"
7 ...
8
9 $ git push
10 ...

```

Die zu dem Ablauf gehörenden git Kommandos aus Sicht des lokalen Repositorys B:

```
1 $ git pull
2 ...
3
4 $ echo "Hi" > hello.txt
5 $ git add .
6 $ git commit -m "hello.txt"
7 ...
8
9 $ git push
10 ...
```

3.5.2.3 Nebenläufiges Arbeiten, Konflikte auflösen

Im Vergleich zu dem Ablauf aus dem vorherigen Abschnitt ist in Abbildung 3.8 die Entstehung eines Konflikts graphisch dargestellt. Der Konflikt entsteht hierbei dadurch, dass zeitgleich widersprüchliche Änderungen an der Datei *hello.txt* vorgenommen werden. Die Änderung aus Repository A werden etwas früher in das Remote Repository übertragen, als das aus dem Repository B heraus durchgeführt wird. Bei dem Versuch, die Änderungen aus Repository B zu übertragen kommt es zu einem Konflikt, da sich der Inhalt aus der referenzierten und geänderten Datei verändert hat.

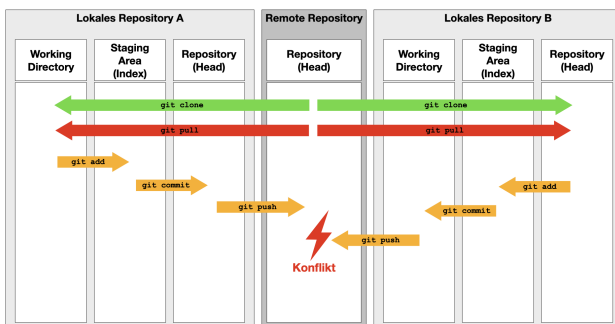


Abbildung 3.8: Repository-Interaktion mit Konflikt

Die zu dem Ablauf gehörenden git Kommandos aus Sicht des lokalen Repositorys A:

```
1 $ echo "Hallo" > hello.txt
2 $ git add .
3 $ git commit -m "hello.txt"
4 ...
5
6 $ git push
7 ...
```

Die zu dem Ablauf gehörenden git Kommandos aus Sicht des lokalen Repositorys B:

```
1 $ echo "Hi" > hello.txt
2 $ git add .
3 $ git commit -m "hello.txt"
4 ...
5
6
7 $ git push
8 ...
9 ! [rejected]          main -> main (fetch first)
10 ...
11 hint: (e.g., 'git pull ...') before pushing again.
12 hint: See the 'Note about fast-forwards' in 'git push --help' for details.
13
14 $ git pull
15 ...
16 Auto-merging hello.txt
17 CONFLICT (content): Merge conflict in hello.txt
18 Automatic merge failed; fix conflicts and then commit the result.
19
20 $ git checkout --ours hello.txt
21 ...
22 $ git add . ; git commit -m "hello.txt fixed"
23 ...
24 $ git push
25 ...
```

In diesem Beispiel meldet die Versionsverwaltung, dass das Übertragen der Änderungen abgelehnt (rejected) wurde. Deshalb muss im Anschluss ein *git pull* durchgeführt werden, um den letzten Stand aus dem Remote Repository zu übertragen. Der dadurch angestoßene Auto-Merge schlägt jedoch fehl, weil der Inhalt der Datei *hello.txt* widersprüchlich ist. Dieser Widerspruch muss nun durch den Benutzer aufgelöst werden, durch den dieser Widerspruch entstanden ist. Dazu gibt es mehrere Strategien. Die in diesem Beispiel gezeigte

verwendet die eigene Datei als die "richtige" Datei, indem das Kommando *git checkout* mit dem Parameter *–ours* und dem Dateinamen versehen wird. Alternativ könnte auch die Datei aus dem Remote Repository durch den Parameter *–theirs* verwendet werden.

3.5.3 Zugriff auf frühere Versionen

Ein wesentlicher Vorteil einer Versionsverwaltung ist es, dass jederzeit auf versionierte Stände einer Datei zurückgegriffen werden kann. Häufiges Versionieren (*commit*) hat demnach den großen Vorteil, dass jeder dieser Stände wiederhergestellt werden kann.

3.5.3.1 Gelöschte Datei wiederherstellen

Während der Entwicklung kann es vorkommen, dass eine lokale Datei verloren geht (gelöscht wird). Wenn sich diese nun unter Versionsverwaltung befand, kann sie sehr einfach wiederhergestellt werden. Das folgende Beispiel löscht in Zeile 4 die Datei README.txt. Durch das *status* Kommando wird sofort angezeigt, dass die Datei README.txt gelöscht wurde. Unter Anwendung des *restore* Kommandos (Zeile 17) wird der letzte, versionierte Stand der Datei wieder im Arbeitsverzeichnis hergestellt.

```
1 $ ls
2 README.txt      hello.txt      src
3
4 $ rm README.txt
5
6 $ git status
7 On branch main
8 Your branch is up to date with 'origin/main'.
9
10 Changes not staged for commit:
11   (use "git add/rm <file>..." to update what will be committed)
12   (use "git restore <file>..." to discard changes in working directory)
13     deleted:    README.txt
14
15 no changes added to commit (use "git add" and/or "git commit -a")
16
17 $ git restore README.txt
```

```
18 $ ls
19 README.txt      hello.txt      src
20
```

3.5.3.2 Früheren Stand einer Datei wiederherstellen

Bei längerer Projektbearbeitung kann der Bedarf entstehen, auf einen früheren Stand einer Datei zurückgreifen zu wollen. Dazu kann die Historie in der Versionsverwaltung abgefragt und auf Basis des früheren *commits* der Versionsstand der Datei wiederhergestellt werden. In Zeile 6 wird mit Hilfe der *log* Kommandos auf die Historie zugegriffen. Nachdem die Datei README.txt in Zeile 1 bewusst gelöscht wurde, soll mit Hilfe der Historie der Dateistand vom 06.11.2021 10:14:48 Uhr wiederhergestellt werden. Dazu muss der dazugehörige *commit* Hash identifiziert werden, siehe Zeile 15. Zur Wiederherstellung des dazugehörigen Versionsstands müssen dem *checkout* Kommando der Hash und der Dateiname übergeben werden, siehe Zeile 24.

```
1 $ git rm README.txt ; git commit -m "README.txt entfernt"
2 ...
3
4 $ ...
5
6 $ git log -2 --name-only
7 commit 20b45dd6b4108821456b0b71fca89174c8316438 (HEAD -> main, origin/main, origin/HEAD)
8 Author: Prof. Dr. Benjamin Kormann <benjamin.kormann@hm.edu>
9 Date: Sat Nov 6 10:30:00 2021 +0100
10
11     README.TXT entfernt
12
13     README.txt
14
15     commit b819cd306555ea1ba760994003c8717232d90072
16     Author: Prof. Dr. Benjamin Kormann <benjamin.kormann@hm.edu>
17     Date: Sat Nov 6 10:14:48 2021 +0100
18
19         some files
20
21     README.txt
22     src/exercise.c
23
24 $ git checkout b819cd306555ea1ba760994003c8717232d90072 README.txt
25 90072 README.txt
26 Updated 1 path from 046b035
27
```

Literaturverzeichnis

- [1] The Linux Foundation. 2017 Linux Kernel Report Highlights Developers' Roles and Accelerating Pace of Change, 2017.
- [2] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2020-12-01 edition, 2020.
- [3] Scott Chacon and Jason Long. Git Reference. <https://git-scm.com/docs>, 2021.