Studienarbeit Herbstsemester 2025

Version: 2025-12-19

GitLab Time-Report

Generierung von Statistiken und Diagrammen über geleistete Arbeitszeit auf GitLab

Projektteam: **Nina Grässli**
Jannis Tschan

Betreuer: **Mirko Stocker**
Dominic Klinger

Abstract

Für Semester- und Bachelorarbeiten an der OST müssen Studierende ihre Arbeitszeit erfassen. Sie sind dabei frei in der Wahl ihrer Methodik, doch empfohlen werden Jira und YouTrack. Diese Tools sind jedoch schwerfällig und haben eine steile Lernkurve. Als Alternative kann das Zeiterfassungs-Feature von GitLab verwendet werden, das sich sehr einfach bedienen lässt. GitLab bietet jedoch keinerlei Funktionalitäten zur Auswertung der Zeiteinträge an. Es besteht lediglich die Möglichkeit, die Einträge über eine API auszulesen. Es gibt mehrere Programme, die versuchen, diese Lücke zu schliessen, indem sie die Zeiteinträge von GitLab über die API abrufen und weiterverarbeiten. Diese sind jedoch teils veraltet, fehleranfällig, unflexibel oder erfordern viel manuelle Nacharbeit.
Genau hier setzt unser Projekt «GitLab Time-Report» an: Ein funktionales und wartbares Programm, das die Zeiteinträge von GitLab extrahiert, für die Studierenden aufbereitet und dabei perfekt auf deren Bedürfnisse zugeschnitten ist.

Die Business-Logik von «GitLab Time-Report» ist in einer Rust-Library umgesetzt, die über ein CLI-Frontend angesteuert wird.
Die Applikation bereitet die von der GitLab GraphQL-API bezogenen Zeiteinträge in verschiedenen Diagramm-Typen und Tabellen auf und unterstützt diverse Filtermöglichkeiten und eine Validierung der Einträge. Um einen Überblick über die geleisteten Arbeitsstunden zu erhalten, werden die Diagramme und Tabellen in ein Dashboard eingebettet, das sich beispielsweise über GitLab Pages veröffentlichen lässt. Zusätzlich können die Diagramme auch über CI/CD in die Projekt-Dokumentation (z.B. LaTeX, Typst, Markdown etc.) übernommen werden.
Die Qualität des Programms wird durch Unit- und Integration-Tests mit hoher Testabdeckung und einem strikt konfigurierten Linter sichergestellt. Diese Massnahmen werden automatisiert durch die CI/CD Pipeline ausgeführt, welche auch Builds für Windows und Linux erstellt. Damit unser Programm leicht bedienbar ist, wurden Usability Tests durchgeführt.
Für die Sicherstellung der Wartbarkeit haben wir eine umfangreiche Code-Dokumentation geschrieben, die es anderen Personen einfach macht, die Applikation weiterzuentwickeln.

Mit «GitLab Time-Report» erreichen wir das Ziel, ein praxistaugliches und robustes Werkzeug für die Projekt-Planung via GitLab bereitzustellen. Es erfüllt die am Anfang des Projektes definierten Anforderungen, wird von Usern als intuitiv bedienbar wahrgenommen und lässt sich in bestehende Workflows integrieren.
Durch die Wartbarkeit, die wir durch die umfangreiche Dokumentation und die saubere Architektur sichergestellt haben, kann die Applikation in Zukunft ohne Probleme durch weitere Features ergänzt werden.

Nach Abschluss des Projektes werden wir «GitLab Time-Report» unter der GPLv3-Lizenz öffentlich zugänglich machen, damit neben den OST-Studierenden auch die breite Öffentlichkeit von diesem Programm profitieren kann.

Management Summary

Ausgangslage

Für das SE Project, die Studien- und die Bachelorarbeit an der OST wird eine verbindliche Zeiterfassung verlangt. Viele Studierende arbeiten sich dafür in Jira oder YouTrack ein. Da jedoch gleichzeitig auch eine Versionskontrolle des Codes gefordert ist und diese in der Praxis meist über GitLab erfolgt, bietet sich das Erfassen der Aufwände direkt in GitLab Issues und Merge Requests als naheliegende Alternative an.

GitLab erlaubt zwar das Eintragen von Zeitschätzungen und das Erfassen von Aufwänden, bietet jedoch keinerlei Auswertungs- oder Filtermöglichkeiten. Die Daten können lediglich über eine API ausgelesen werden, die Weiterverarbeitung muss eigenhändig erledigt werden. Dieser Prozess verursacht zusätzlichen Aufwand und führt häufig zu inkonsistenten oder unvollständigen Reports.

Es existieren bereits verschiedene Programme, welche Zeiteinträge aus GitLab abrufen und aufbereiten. Diese Lösungen sind jedoch je nach Tool veraltet, fehleranfällig oder unvollständig und damit nur eingeschränkt für den Einsatz im unserem Kontext geeignet, wie die Tabelle 1 zeigt. Eine genauere Beschreibung der einzelnen Applikationen befindet sich in Abschnitt 2.2.

	gitlab-time-tracker	gitlab-time-tracker (Fork)	gtt-charts	gitlab-timelogs
Fehlerfreier Abruf der Zeiteinträge	✗	✗	✗	✓
Aktive Weiterentwicklung	✗	✓	✓	✓
Keine bekannten Bugs	✗	✓	✗	✓
Verwendbar mit der OST-GitLab-Instanz	✓	✓	✓	✗
Daten-Export	✓	✓	✗	✓
Erstellen von Diagrammen	✗	✗	✓	✗
Übersicht über alle Daten als Dashboard	✗	✗	✗	✗

Tabelle 1: Vergleich von bestehenden Programmen

Ziele der Arbeit

Das Ziel dieser Arbeit war die Entwicklung eines funktionalen und wartbaren Programmes, das Zeiteinträge aus GitLab automatisiert extrahiert, validiert, filtert und für Semester- und Abschlussarbeiten als Tabellen und Diagramme aufbereitet. Die Ergebnisse sollen zusätzlich in einem Dashboard dargestellt werden können, das sich beispielsweise über GitLab Pages veröffentlichen lässt.

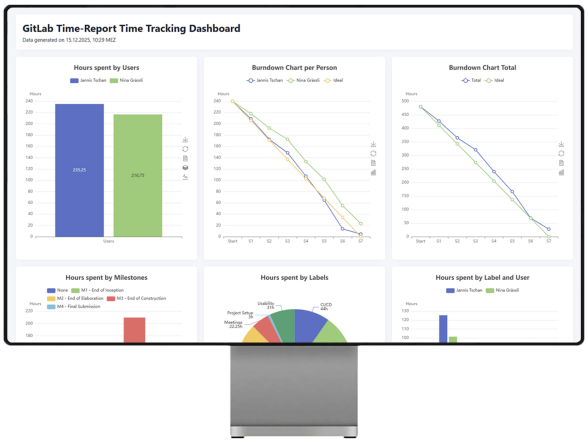


Abbildung 1: Dashboard im Light Mode

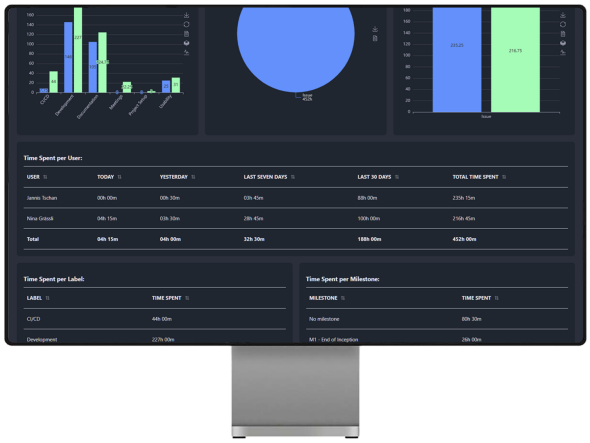


Abbildung 2: Dashboard im Dark Mode

Vorgehen und Technologien

In einem ersten Schritt haben wir bestehende Tools analysiert, Stärken und Schwächen dokumentiert und daraus die Anforderungen an GitLab Time-Report abgeleitet. Darauf aufbauend haben wir das Design und die Architektur des Programmes erarbeitet. Ein zentrales Architekturprinzip war die strikte Trennung zwischen Business-Logik und Frontend, um die Wartbarkeit und Testbarkeit sicherzustellen.

Die Umsetzung erfolgte in Rust. Die Kernfunktionen (Datenbeschaffung, Validierung, Aufbereitung) sind in einer Library gekapselt. Bedient wird das Programm über ein Kommandozeilen-Interface, das basierend auf User-Inputs die passenden Library-Funktionen aufruft und den Output dem User zur Verfügung stellt.

```
gitlab-time-report exports time logs from Issues and Merge Requests of a GitLab repository to create statistics and charts of your working hours. Use --help on commands to get more information.
If no subcommands are used, statistics will be printed directly to your console.


Usage: gitlab-time-report-cli.exe [OPTIONS] <URL> [COMMAND]

Commands:
  export     Export the time logs to a CSV file
  charts     Generate charts from your time logs. This function creates the following charts: Time spent per user, per milestone, per user/label and burndown charts (total and per user)
  dashboard  Generate an HTML file with charts and statistics about the time logs
  help       Print this message or the help of the given subcommand(s)

Arguments:
  <URL>  The URL of the GitLab repository you want to export time logs from [env: GITLAB_URL=]

Options:
  -t, --token <TOKEN>
          A GitLab access token. Needed if the repository is not public. Can be a personal, group or repository access token [env: GITLAB_TOKEN=]
  -l, --labels <LABELS>
          Comma-separated list of GitLab labels to filter by. All other labels will be grouped under "Others". If your labels contain spaces, delimit them with `"`: `--labels "Epic 1",Documentat
ion`. The labels are case-sensitive. If not set, all labels are shown [env: GITLAB_LABELS=]
      --validation-details
          Displays the problems with the time logs found during validation
      --validation-max-hours <VALIDATION_MAX_HOURS>
          When validating time logs, the maximum number of hours a time log can have [default: 10]
  -h, --help
          Print help
  -V, --version
          Print version

By default, validation of time logs is enabled. It checks for excessive hours, future dates, duplicate time logs and missing summaries.
If a time log violates any of these rules, it is printed to the console. No automatic correction is performed. You can display the full validation results with --validation-details.
```

Abbildung 3: Übersicht über die Funktionen unseres Programmes

Die Zeiteinträge werden von der Library über die GitLab GraphQL-API bezogen und als Tabellen, Diagramme und in einem übersichtlichen, publizierbarem Dashboard aufbereitet.

Zusätzlich können die Diagramme in bestehende Workflows integriert werden. Beispielsweise haben wir in unserer Pipeline die von GitLab Time-Report generierten Diagramme direkt in unsere Typst-Dokumentation eingebunden.

Um die Robustheit unseres Programmes zu gewährleisten, haben wir verschiedene Massnahmen zur Qualitätssicherung ergriffen. Dazu zählen Unit-, Integration- und Usability-Tests, ein strikt konfigurierter Linter sowie eine CI/CD-Pipeline, die automatisch Windows- und Linux-Builds erstellt und die Test-Suite sowie den Linter ausführt.

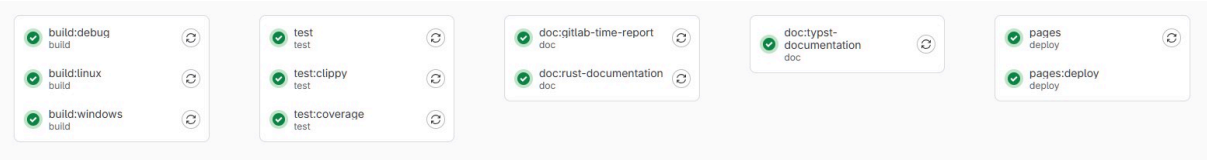


Abbildung 4: Einbindung von GitLab Time-Report in die CI/CD Pipeline

Fazit und Ausblick

Mit «Gitlab Time-Report» ist es uns gelungen, ein praxistaugliches Werkzeug zu schaffen, dass die Zeiterfassung in GitLab um Funktionen ergänzt, die nicht nur im OST-Kontext für die Projektplanung relevant sind: Validierung, Filterung, Auswertung und Publikation.

Das Ergebnis erfüllt die zu Projektbeginn definierten Anforderungen, lässt sich in bestehende GitLab-Workflows integrieren und wird von Usern als intuitiv bedienbar wahrgenommen. Dank sauberer Architektur, automatisierter Qualitätssicherung und umfassender Dokumentation ist das Projekt wartbar und kann künftig ohne grosse Architektur-Änderungen erweitert werden.

Nach Abschluss des Projektes wird GitLab Time-Report unter der GPLv3-Lizenz öffentlich zugänglich gemacht. Damit können neben OST-Studierenden auch externe Teams und andere Open-Source-Projekte von der Lösung profitieren.

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Für die Module SE Project (SEProj), Studienarbeit (SA) und Bachelorarbeit (BA) an der OST (Ostschweizer Fachhochschule) ist eine Zeiterfassung der geleisteten Arbeit erforderlich. Dabei sind die Studierenden frei bei der Wahl ihrer Methodik, es werden aber die Projektmanagement-Tools Jira und YouTrack empfohlen. Diese sind allerdings unserer Meinung nach relativ komplex und benötigen vor allem beim erstmaligen Verwenden einige Einarbeitungszeit.

Als Alternative dazu dient GitLab: Die Studierenden sind damit durch bereits besuchte Module schon vertraut und der Code der Arbeiten wird häufig auf der OST GitLab-Instanz gehostet. GitLab bietet ein Zeiterfassungs-Feature, welches erlaubt, bei Issues und Merge Requests (MR) den geschätzten Zeitaufwand und die tatsächlich benötigte Zeit zu erfassen. Eine simple Auflistung aller eingetragenen Zeiten ist möglich, allerdings zum Zeitpunkt dieser Arbeit von GitLab noch als «Experiment» gekennzeichnet [1]. Eine Filterung der Einträge in GitLab selbst ist nicht möglich.

Hier setzen bereits bestehende Tools wie gitlab-time-tracker [2] oder gtt-charts [3] an. Sie extrahieren die Zeitdaten mithilfe der GitLab-API und fassen diese nach User oder Issue/MR zusammen. Jedoch sind diese Tools entweder nicht mehr gewartet oder funktionieren nicht mehr zuverlässig. Zudem sind sie in der Bedienung umständlich und erfordern einen fehleranfälligen, zeitintensiven Workflow mit manuellen Schritten, um die Daten für die Projektdokumentation aufzubereiten.

1.2 Aufgabenstellung

Das Ziel dieser Arbeit ist es, ein wartbares und robustes Programm zu entwickeln, welches die Daten aus GitLab extrahiert und für die Verwendung in Semesterarbeiten aufbereitet. Dazu gehören das automatisierte Erstellen von Diagrammen (Zeit pro Teammitglied, Burndown-Chart) und das Filtern der Arbeitszeiten nach verschiedenen Kriterien. Die Funktionalitäten sollen sowohl über eine Kommandozeilen-Schnittstelle (CLI), als auch über ein grafisches Frontend zur Verfügung gestellt werden. Damit wird gewährleistet, dass das Tool sowohl in einer CI/CD-Pipeline als auch durch die User manuell auf die individuellen Anforderungen zugeschnitten werden kann.

Die Entwicklung von GitLab-Time-Report erfolgt in der Programmiersprache Rust. Damit es auch von anderen Personen weiterentwickelt werden kann, wird als Teil der Dokumentation eine Einführung in Rust geschrieben. Diese soll auch Einsteigern die Verwaltung des Tools möglich machen. Unser Ziel ist es, dass Studierende unser Programm für ihre Semesterarbeiten verwenden und bei Bedarf weiterentwickeln können.

Da diese Funktionalitäten auch für Personen ausserhalb der OST-Studienarbeiten interessant sein können, planen wir nach dem Abschluss dieser Studienarbeit, das Programm zu veröffentlichen.

1.3 Rahmenbedingungen

Dieses Projekt wird als Semesterarbeit durchgeführt. Der Schwerpunkt liegt auf der Konzeption, Entwicklung und Evaluation eines Programmes, welches das Zeitmanagement eines Projektes mit GitLab erleichtern soll. Das Projekt umfasst die Planung, Implementierung sowie die Erstellung einer Dokumentation über Rust und das Programm selbst.

- **Arbeitstyp:** Semesterarbeit (240 Stunden)
- **Gesamtarbeitsaufwand:** 480 Stunden
- **ECTS-Punkte:** 8 ECTS pro Person

2 Analyse

2.1 Anforderungen

In diesem Kapitel werden die Anforderungen an das entwickelte Produkt dargelegt. Die Functional Requirements werden mit User Stories abgebildet, welche die Funktionen des Tools modellieren. Die Non-Functional Requirements dienen hingegen dazu, die Qualitätsmerkmale des Produkts zu beschreiben.

2.1.1 Functional Requirements

Unser Produkt hat nur eine Art von Usern: Die Personen, welche das Tool nutzen, um ihre Zeiten von GitLab zu extrahieren und Diagramme zu erstellen. In unserem Fall sind das mehrheitlich Informatik-Studierende an der OST. Die Funktional Requirements werden als User Story nach der Struktur «Als User möchte <Aktion> so dass <Effekt>» definiert [4].

Die User Stories 1 bis 5 bilden die Features unseres MVP ab, diese sind zusätzlich auch mit Akzeptanzkriterien genauer ausformuliert. Die Beschreibung bezüglich Umsetzung der User Stories befindet sich in Abschnitt 6.1.1.

2.1.1.1 User Story 1 – Verwendung via CLI

Als User möchte ich das Produkt via CLI verwenden können, um die Funktionalitäten auf effiziente Art und Weise nutzen zu können.

Akzeptanzkriterien

- ☐ Eine vollständig dokumentierte Anleitung ist vorhanden
- ☐ Die einzugebenden Befehle sind logisch benannt und möglichst kurz gehalten
- ☐ `--help` zeigt eine Übersicht über die angebotenen Befehle und Argumente an
- ☐ Autocompletion mittels Tab wird angeboten

2.1.1.2 User Story 2 – Zeiten anzeigen

Als User möchte ich die extrahierten und gefilterten Zeiten tabellarisch angezeigt bekommen, um schnell einen Überblick über die Daten zu erhalten.

Akzeptanzkriterien

- ☐ Die Zeiten werden im CLI tabellarisch angezeigt
- ☐ Die Zeiten werden im Dashboard (vgl. Abschnitt 2.1.1.5) angezeigt
- ☐ Die Übersicht über die Zeiten ist logisch strukturiert und verständlich
- ☐ Eine heuristische Datenüberprüfung stellt sicher, dass alle erfassten Zeiten und Schätzungen, egal ob über Issues oder Merge Request eingetragen, korrekt berücksichtigt werden
- ☐ Gelöschte Zeiten werden korrekt verarbeitet
- ☐ Bei nicht plausiblen Daten (Minus-Stunden, zu grosse Unterschiede zwischen den Personen) wird der User informiert

2.1.1.3 User Story 3 – Diagramme und Statistiken generieren

Als User möchte ich verschiedene Diagramme und Tabellen erstellen können, um eine schnelle visuelle Übersicht über verschiedene Projektstatistiken zu erhalten und diese für die Projektdokumentation nutzen zu können.

Akzeptanzkriterien

- ☐ Die Daten werden in verschiedenen tabellarischen Ansichten strukturiert dargestellt
- ☐ Ein Diagramm für Gesamtzeit pro Contributor kann erstellt werden
- ☐ Ein Burndown-Diagramm für die Gesamtzeit und pro Person kann erstellt werden
- ☐ Ein Diagramm für den Vergleich von Schätzungen und effektiv geleisteten Stunden kann erstellt werden
- ☐ Ein Diagramm für Stunden pro Label und Person kann erstellt werden
- ☐ Ein Diagramm pro Zuweisung (Issue oder MR) und Person kann erstellt werden
- ☐ Die Diagramme sind als Bild exportierbar
- ☐ Die Darstellung der Diagramme ist übersichtlich, einheitlich und Labels sind lesbar

2.1.1.4 User Story 4 – Zeiten und Statistiken filtern

Als User möchte ich die Zeiten und die Statistiken nach verschiedenen Kriterien filtern können, um für mich relevante Daten analysieren und weiterverwenden zu können.

Akzeptanzkriterien

- ☐ Filterung ist nach Person, Zeit, Datumsbereich, Meilenstein, Label oder Zuweisung (Issue oder MR) möglich
- ☐ Die Filter können kombiniert werden (z.B. nach Zeit und Person)

2.1.1.5 User Story 5 – Übersicht generieren

Als User möchte ich ein Dashboard mit einer Übersicht über die Daten und Diagramme erhalten, um schnell einen vollständigen Überblick über den Stand des Projektes zu bekommen und diesen auch bei Sitzungen präsentieren zu können.

Akzeptanzkriterien

- ☐ Das Dashboard kann als HTML exportiert werden (z.B. für Weiterverwendung mit GitLab Pages)
- ☐ Das Dashboard kann optional über eine GUI-App angezeigt werden (vgl. Abschnitt 2.1.1.6)
- ☐ Das Dashboard enthält sowohl eine tabellarische Übersicht über die Daten als auch die erstellten Diagramme

2.1.1.6 User Stories über mögliche Erweiterungen

In diesem Abschnitt sind die User Stories über die möglichen Erweiterungen definiert, jedoch ohne Akzeptanzkriterien. Eine komplette Liste der Features befindet sich in der Aufgabenstellung

User Story 6 – Integration in CI/CD Workflows
Als User möchte ich das Tool in meine CI/CD-Pipeline integrieren, um bei jedem Build automatisch einen aktuellen Zeit-Report und Diagramme zu generieren und ggf. über GitLab Pages bereitzustellen.

User Story 7 – GUI App für Konfiguration
Als User möchte ich über eine grafische Oberfläche meine Daten und Diagramme nach meinen Bedürfnissen filtern und konfigurieren, um dies nicht über die Kommandozeile machen zu müssen.

User Story 8 – Kategorien erstellen
Als User möchte ich Projekte und Repositories zu Kategorien zusammenfassen können, um Daten für grössere Projekte übersichtlich darzustellen.

User Story 9 – Lokale Stoppuhr-Funktion
Aus User möchte ich meine Zeit automatisch lokal erfassen und mit GitLab synchronisieren können, um das Eintragen von Zeiten zu erleichtern.

User Story 10 – Unterstützung von Sprints
Als User möchte ich Sprints angeben können, um die Daten auf Sprints aufgeteilt darstellen zu können.

User Story 11 – Erstellung von Gantt-Charts
Als User möchte ich aus meinen Meilensteinen und Issues ein Gantt-Chart generieren können, um die Projektplanung automatisiert darstellen zu können.

2.1.1.7 Aufteilung auf EPICs

Wir haben die User Stories in mehrere EPICs gruppiert und somit auch priorisiert. Die User Story 1 (Verwendung via CLI) ist EPIC-übergreifend und kommt in der Tabelle deshalb mehrmals vor. Die EPICS 2 und 3 sind überlappend, da nicht nur die Zeiteinträge, sondern auch die Diagramme gefiltert werden sollen.

EPIC	**User Stories**
EPIC 1: Extraktion und Strukturierung	– US 1: Verwendung via CLI – US 2: Zeiten anzeigen
EPIC 2: Filterung	– US 1: Verwendung via CLI – US 4: Zeiten und Statistiken filtern
EPIC 3: Statistik	– US 3: Diagramme und Statistiken generieren
EPIC 4: Dashboard	– US 5: Übersicht generieren
EPIC 5: Erweiterungen	– US 6 bis 11

Tabelle 2: Liste der EPICs

In Abschnitt A.2 ist ersichtlich, wann die Umsetzung von welchen EPICs eingeplant ist.

2.1.2 Non-Functional Requirements

In den nicht-funktionalen Anforderungen werden die Qualitätsmerkmale des Produktes definiert. Sie beeinflussen, wie die in den funktionalen Anforderungen spezifizierten Features implementiert werden.

Für unser Projekt haben wir uns an den in ISO/IEC 25010:2023 spezifizierten Qualitätsmerkmalen [5], [6] orientiert und uns für folgende NFRs entschieden:

ID	1
Thema	Die Code-Qualität entspricht den im Code-Linter spezifizierten Anforderungen.
Anforderungen	Wartbarkeit – Analysierbarkeit
Massnahmen	Der Linter wurde so konfiguriert, dass er in der IDE und in unserer Pipeline automatisch ausgeführt wird. Vor jedem Commit wird in der IDE auf Warnungen überprüft. Die Pipeline schlägt fehl, wenn der Linter eine Warnung ausgibt. Genaueres dazu befindet sich in Abschnitt 5.1.2.1 und Abschnitt 5.3.
Priorität	mittel
Erstelldatum	25.09.2025
Fälligkeitsdatum	M3 - KW49: Die Linter-Regel für `unwrap()` wird aktiviert. M4 - KW59: Der Linter gibt keine Warnungen aus.

Tabelle 3: Beschreibung NFR 1 «Wartbarkeit – Analysierbarkeit»

ID	2
Thema	Die Testabdeckung beträgt mindestens 80%.
Anforderungen	Wartbarkeit – Testbarkeit
Massnahmen	Unsere CI/CD Pipeline führt bei jedem Push alle Tests aus und generiert danach einen Code Coverage Report. In jedem Merge Request wird die aktuelle Code Coverage angezeigt. Sollte die Test Coverage in einem Merge Request um mehr als 3% fallen, darf dieser nicht gemerged werden.
Priorität	hoch
Erstelldatum	25.09.2025
Fälligkeitsdatum	M3 - KW49 M4 - KW59

Tabelle 4: Beschreibung NFR 2 «Wartbarkeit – Testbarkeit»

ID	3
Thema	Das Programm ist leicht zu bedienen.
Anforderungen	Interaktionsfähigkeit – Lernbarkeit
Massnahmen	Es wird ein Usability Test durchgeführt. Dabei soll mindestens eine Durchschnittsbewertung von 4 auf einer Skala von 1 bis 6 mit mindestens 5 Befragten erreicht werden.
Priorität	mittel
Erstelldatum	25.09.2025
Fälligkeitsdatum	M3 - KW49

Tabelle 5: Beschreibung NFR 3 «Interaktionsfähigkeit – Lernbarkeit»

ID	4
Thema	Die Funktionen sollen von unterschiedlichen Frontends angesprochen werden können.
Anforderungen	Wartbarkeit – Wiederverwendbarkeit
Massnahmen	Die Funktionen des Programms werden in einer separaten Library platziert und sind über eine öffentliche API von Consumern ansprechbar.
Priorität	hoch
Erstelldatum	25.09.2025
Fälligkeitsdatum	M2 - KW43

Tabelle 6: Beschreibung NFR 4 «Wartbarkeit – Wiederverwendbarkeit»

ID	5
Thema	Das Programm soll auf unterschiedlichen Betriebssystemen und Umgebungen (z.B. Docker Container) laufen.
Anforderungen	Flexibilität – Adaptierbarkeit
Massnahmen	Um sicherzustellen, dass unser Code auf jeder Plattform läuft, kompiliert unsere Pipeline das Programm automatisch für mehrere Plattformen. Genaueres dazu befindet sich in Abschnitt 5.3. Sollte sich herausstellen, dass die Portabilität für eine bestimmte Plattform unverhältnismässig aufwändig ist, werden wir diese aus diesem NFR ausklammern.
Priorität	mittel
Erstelldatum	30.09.2025
Fälligkeitsdatum	M3 - KW49

Tabelle 7: Beschreibung NFR 5 «Flexibilität – Adaptierbarkeit»

ID	6
Thema	Es soll eine Anleitung für Rust-Einsteiger erstellt werden, um die Wartung des Projektes zu erleichtern.
Anforderungen	Wartbarkeit – Modifizierbarkeit
Massnahmen	Wir möchten, dass andere Personen nach Abschluss unserer Studienarbeit (SA) das Programm «GitLab Time-Report» weiterentwickeln können. Um dies zu erleichtern, möchten wir eine Anleitung schreiben, welche Einsteigern in Rust die wichtigsten im Tool verwendeten Konzepte erklärt.
Priorität	niedrig
Erstelldatum	05.10.2025
Fälligkeitsdatum	M4 - KW59

Tabelle 8: Beschreibung NFR 6 «Wartbarkeit – Modifizierbarkeit»

2.2 Marktanalyse

Im Rahmen unserer Marktanalyse, die wir mithilfe einer Internet-Recherche durchgeführt haben, konnten wir folgende Produkte mit ähnlicher Funktionalität wie dieser unseres Produktes identifizieren.

2.2.1 gitlab-time-tracker (Original)

gitlab-time-tracker [2] ist ein in JavaScript implementiertes CLI-Programm, welche alle eingetragenen Zeiten aus Issues und Merge Request extrahiert und in einem vom User gewünschten Format ausgibt (ASCII-Tabelle, Markdown, CSV, PDF, Excel-Tabelle). Es können Zeiteinträge für einzelne Repositories oder ganze GitLab-Gruppen abgerufen werden. Die Zeiteinträge können nach Datum, GitLab-Milestone, Typ (Issue oder MR), Status des Issue/MRs (open/closed), User und Label gefiltert werden [7].

Zusätzlich unterstützt es die automatische lokale Zeiterfassung: Der User kann zu Beginn ein Issue oder Merge Request angeben, dann wird die Zeit lokal gemessen, bis der User diese stoppt. Die erfasste Zeit kann dann direkt mit GitLab synchronisiert werden [7].

Wir haben mit diesem Programm bereits im Modul SEProject gearbeitet, wo wir die Zeiten jeweils pro Datumsbereich der Sprints abgerufen und als CSV gespeichert haben. Diese Daten haben wir dann manuell in eine Excel-Datei übernommen und dort die Diagramme für die Dokumentation generiert. Da gitlab-time-tracker die Zeiten aber in Stunden und Minuten angibt, Excel aber nur mit dezimalen Nachkommastellen umgehen kann, haben wir die Zeiten manuell konvertiert. Dieser Prozess stellte sich als fehleranfällig heraus und kostete zusammen mit dem Konfigurieren und Erstellen der Diagramme viel Zeit.

Ebenfalls hatten wir mit einem Bug zu kämpfen: gitlab-time-tracker greift auf der GitLab-API nicht direkt auf die gespeicherten Zeiteinträge zu, sondern auf die Verlaufsmeldungen, welche im jeweiligen Issue/MR unter «Aktivität» sichtbar sind (z.B. Person als Reviewer hinzugefügt, Titel/Beschreibung geändert etc.). Es filtert dann nach denjenigen Meldungen, die für das Zeit-Management relevant sind und addiert diese dann zusammen. Leider hat sich seit der Veröffentlichung der Text für die «Zeiteintrag gelöscht»-Meldung geändert. gitlab-time-tracker hat also gelöschte Einträge nicht aus der Gesamtzeit entfernt. Aufgefallen ist uns das erst, als ein Teammitglied versehentlich 45h anstatt 45min eingetragen hatte und diese Zeit auch nach der Korrektur immer noch in den finalen Daten vorlag. Wir haben dann gitlab-time-tracker selbst repariert mithilfe eines Patches, welcher zum damaligen Zeitpunkt als offener Pull Request im gitlab-time-tracker-Repository vorlag [8]. Danach wurden die Zeiten korrekt erfasst.

Aufgrund der oben genannten Probleme und der Archivierung des Repositories am 22. Mai 2025 können wir die weitere Verwendung des Tools nicht empfehlen.

2.2.2 gitlab-time-tracker (Fork)

Die Weiterentwicklung des Original gitlab-time-tracker wurde eingestellt: Der letzte Code-Commit wurde am 07. Juni 2020 getätigt [9], bevor das Repository am 22. Mai 2025 vom Autor archiviert wurde. Vom Autor wurde aber ein Fork des GitHub-Nutzers «ndu2» als offizieller Nachfolger designiert [10]. Der Bug, welchen wir bei der Original-Variante beschrieben haben, wird behoben [11] und das Tool fügt zusätzlich ein Feature zum Generieren von Offerten als PDF hinzu.

Der Fork hat bei einem kurzen Funktionstest ohne Probleme funktioniert und kann als Ersatz anstelle des Originals verwendet werden. Da die Kritik des Originals weiterhin bestehen bleibt, können wir aber auch diesen nur eingeschränkt empfehlen.

2.2.3 gtt-charts

gtt-charts ist ein .NET 5-basiertes CLI-Programm, welches von Moritz Schiesser neben seiner SE Project-Arbeit im FS 2021 entwickelt wurde [3], [12]. Dieses verwendet eine auf gitlab-time-tracker basierende Parsinglogik für die Zeiteinträge und hat deswegen die selbe Problematik wie diese Tools. Die Benutzung ist relativ simpel: Man erstellt eine `gttchartsettings.json` mit Einträgen für die GitLab-Instanz, Repository und (falls nötig) Access Token. Das Programm extrahiert dann die Zeiteinträge und generiert aus diesen Daten neun verschiedene Diagramme [13].

Wir haben dieses Programm auch testweise im SE Project verwendet, mussten aber schnell feststellen, dass viele der Zeiteinträge nicht richtig erfasst oder zugeordnet werden konnten. Schon beim Parsen der Zeiteinträge der GitLab-API gibt das Programm häufig «`could not parse Date`»-Fehler aus. Die Diagramme enthalten dann schlussendlich völlig abstruse Daten (beispielsweise soll laut `gtt-charts` ein Teammitglied in unserem SE Project total nur 8 Stunden gearbeitet haben).

Ein weiteres Problem ergibt sich bei grösseren Projekten mit vielen Zeiteinträgen: In den Diagrammen werden dadurch viele Labels generiert, welche sich überlappen und dadurch teils unleserlich werden.

2.2.4 gitlab-timelogs

gitlab-timelogs ist ein in Rust geschriebenes CLI-Programm, welches wie gitlab-time-tracker die Zeiteinträge aus GitLab extrahiert und eine Zusammenfassung der Zeiten liefert [14]. Im Unterschied zu gitlab-time-tracker kann gitlab-timelogs dies pro Durchlauf nur für einen einzelnen User, um die Zeiten für ein Team zu erhalten muss das Programm mehrmals ausgeführt werden. Als einziges der verwendeten Programme parst gitlab-timelogs nicht die Verlaufsmeldungen der Issues/MRs, sondern nutzt GitLabs GraphQL-API, um die Zeiten direkt abzufragen. Diese Methodik vermeidet fehleranfälliges Parsen von Text und greift stattdessen direkt auf die gewünschten Daten zu.

Leider kann gitlab-timelogs Stand Beginn dieser Studienarbeit nicht mit der OST GitLab-Instanz verwendet werden, da es zusätzlich zu den Zeiten noch GitLab Epics abruft [15]. Diese sind in der von der OST verwendeten GitLab Community Edition nicht verfügbar; gitlab-timelogs ist also für Arbeiten mit dieser nicht verwendbar.

gitlab-timelogs bietet ebenfalls keine Möglichkeit, die erhaltenen Daten in Diagramme umzuwandeln, womit auch diese Kritikpunkte von gitlab-time-tracker bestehen bleiben.

2.2.5 Fazit

Es gibt bereits einige Lösungen in diesem Bereich, welche aber alle für unsere Zwecke ungenügend sind und weiterhin Handarbeit benötigen, um aus den eingetragen Zeiten in GitLab passende Diagramme zu generieren. Hier wollen wir mit GitLab Time-Report ansetzen.

3 Design und Architektur

3.1 Systemkontext und Domäne

In diesem Kapitel werden die externen Schnittstellen und verwendete Infrastruktur von GitLab Time-Report beschrieben.

3.1.1 GitLab-API

Die wichtigste externe Schnittstelle ist die GitLab-API. Über sie rufen wir alle benötigten Informationen über die Repositories ab (Zeiteinträge, Labels, Issues/Merge Requests usw.). Die GitLab-API lässt sich via REST und GraphQL ansprechen. Diese beiden Schnittstellen haben jedoch nicht denselben Umfang. Beispielsweise lassen sich die Zeiteinträge nur über die GraphQL-API abrufen. Mit der GraphQL-API lassen sich auch flexiblere Abfragen erstellen, deswegen verwenden wir in diesem Projekt ausschliesslich die GraphQL-API.

3.1.1.1 Access Token

Ist ein Repository nicht öffentlich (Sichtbarkeit «Intern» oder «Privat»), muss für den Zugriff auf dieses Repository über die API ein Access Token erstellt werden. Dieser funktioniert wie ein Passwort und sollte auch dementsprechend behandelt werden. GitLab bietet drei verschiedene Arten von Access Token an: Projekt-, Gruppen- und Persönliche Access Token. Diese unterscheiden sich in ihrer Funktionalität nicht, sie bestimmen lediglich, auf welche Repositories ein Token Zugriff hat. Mit einem Projekt-Access Token kann beispielsweise nur auf dieses Repository zugegriffen werden, während mit einem Persönlichen Access Token auf alle Repositories des Users zugegriffen werden kann [16], [17], [18].

Die Berechtigungen eines Access Tokens können bei der Erstellung festgelegt werden. Für unsere Zwecke muss ein Access Token die «`read_api`»-Berechtigung besitzen. Bei Gruppen- und Projekt-Access Token kann zusätzlich noch eine Rolle gesetzt werden. Um die von uns benötigten Daten zu erhalten, muss der Token mindestens die «Reporter»-Rolle besitzen [19].

3.1.1.2 GraphQL

Die GraphQL-API bietet flexiblen Zugang auf die von uns benötigten Daten. Dabei wird zuerst ein Root-Element definiert, von welchem alle anderen Informationen abgerufen werden. Wird beispielsweise wie in Listing 1 `timelogs` (Zeiteinträge) als Root-Element gesetzt, wird als Parameter das Repository, von welchem die Zeiteinträge abgerufen werden sollen, mitgegeben. Dann können in der gleichen Abfrage weitere Elemente abgerufen werden, wie das dazugehörige Issue oder Meilenstein. In Listing 1 und Listing 2 wird eine GraphQL-Anfrage und die dazugehörige Antwort gezeigt.

```
query {
  timelogs(projectId: "gid://gitlab/Project/13891") {
    nodes {
      spentAt,
      timeSpent,
      summary,
      user {
        name
      }
      issue {
        title,
        labels {
          nodes{
            title
          }
        }
      }
    }
    ...
```

Listing 1: Beispiel für eine GraphQL-Query auf der GitLab-API

```
{
  "data": {
    "timelogs": {
      "nodes": [
        {
          "spentAt": "2025-09-15T12:00:00+02:00",
          "timeSpent": 3600,
          "summary": "Brainstorming before first meeting",
          "user": {
            "name": "Jannis Tschan"
          },
          "issue": {
            "title": "Project Setup",
            "labels": {
              "nodes": [
                {
                  "title": "Project Management"
                }
              ]
            }
          }
        },
        ...
```

Listing 2: Beispiel für eine GraphQL-Response auf der GitLab-API

3.1.2 Gerät des Users

Da unsere Zielgruppe, wie in Abschnitt 2.1.1 beschrieben, aus Informatik-Studierenden an der OST besteht, soll unser Programm auf den gängigen Betriebssystemen Windows, MacOS und Linux sowie innerhalb eines Docker-Containers laufen. Dazu soll der Code möglichst platformneutral sein. Um dies zu überprüfen, wird innerhalb der Pipeline der Code jeweils für Windows und Linux kompiliert, aus technischen Gründen aber nicht für MacOS (Details dazu befinden sich in Tabelle 7 und Abschnitt 5.3).

3.2 Domain Modell

In diesem Kapitel wird unsere Problemdomäne mit folgenden konzeptionellen Klassen modelliert.

- **GitLab Repository:** Das Repository, in welchem sich die Issues und Merge Requests befinden. Ein User muss Mitglied in einem Repository sein, um dort Zeiten eintragen zu können.
- **User:** Die Personen, welche Zeiten eintragen.
- **Time Log:** Einzelne Zeiteinträge. Diese gehören jeweils zu einem User und zu einem Trackable Item. Hierbei ist anzumerken, dass GitLab auch negative Zeiten erlaubt, um Zeit von einem Trackable Item abzuziehen. Bei der Implementation müssen deswegen auch negative Zeiten korrekt verarbeitet werden können.
- **Trackable Item:** Auf einem Trackable Item können Zeiteinträge vorgenommen werden. Es besitzt beliebig viele Label und maximal einen Meilenstein.
- **Work Item:** Subklasse von Trackable Item. Kann entweder ein Issue, ein Task oder ein Incident sein. Definiert meist eine bestimmte Aufgabe oder ein zu implementierendes Feature.
- **Merge Request:** Merge Requests werden erstellt, um einen Branch in einen anderen Branch zu mergen. Auch dies ist eine Subklasse von Trackable Item.
- **Label:** Auf GitLab können Labels erstellt werden, um Trackable Items zu kategorisieren. Jeder Issue und jeder Merge Request kann mehrere Labels besitzen.
- **Meilenstein:** Meilensteine werden gebraucht, um Datumsbereiche zu definieren, die für das Projekt relevant sind. Trackable Items können jeweils maximal einem Meilenstein zugewiesen werden.

Das folgende Diagramm bildet die Beziehungen zwischen den Klassen ab und dient somit als Grundlage für die technische Umsetzung. In der GraphQL-API von GitHub sind alle konzeptionellen Klassen abrufbar.

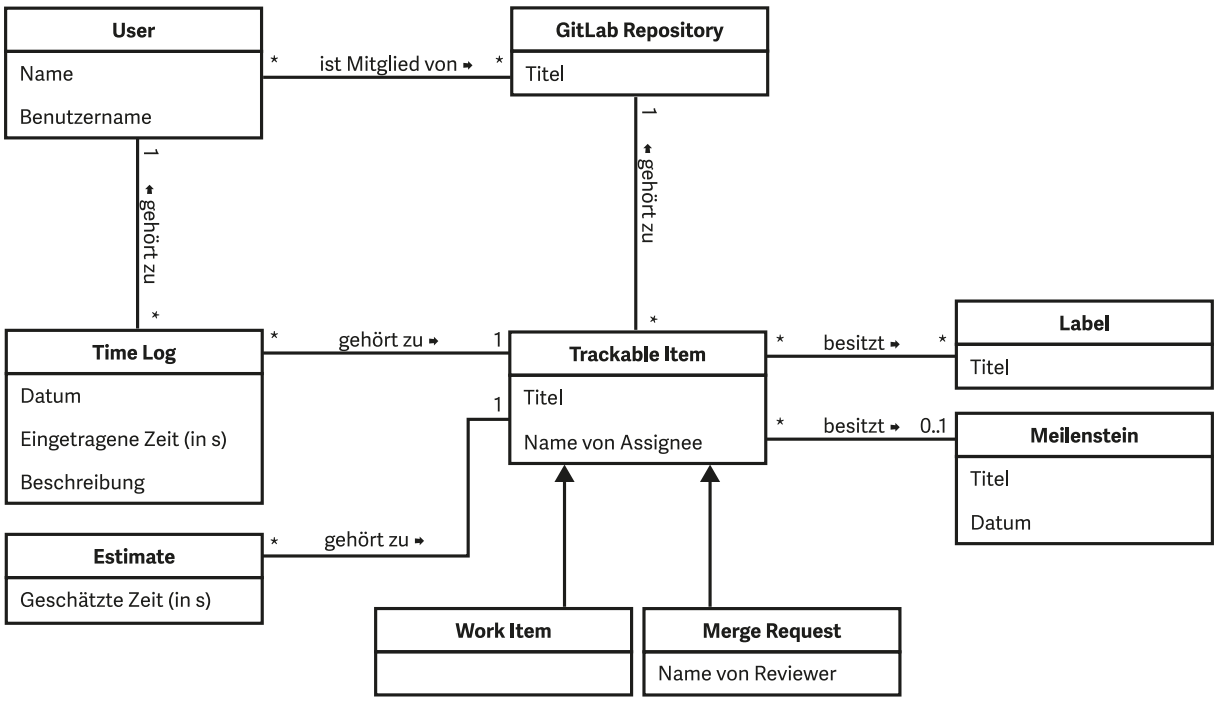


Abbildung 5: Domain Modell

3.3 C4-Modell

Zur Visualisierung der Architektur haben wir Diagramme nach dem C4-Modell erstellt [20]. Diese zeigen die Struktur der Applikation in verschiedenen Kontexten an.

3.3.1 System-Kontext

Das System-Kontext-Diagramm zeigt die Gesamtübersicht der Applikation inklusive der Interaktion mit dem externen System der GitLab API auf. Wie in Abschnitt 2.1.1 beschrieben, existiert nur ein einziger Akteur. Dieser interagiert einerseits mit seiner GitLab-Instanz, auf welcher er CRUD-Aktionen der Zeiteinträge ausführt, andererseits mit GitLab Time-Report-Frontends, wo er seine Einstellungen für das Programm vornimmt.

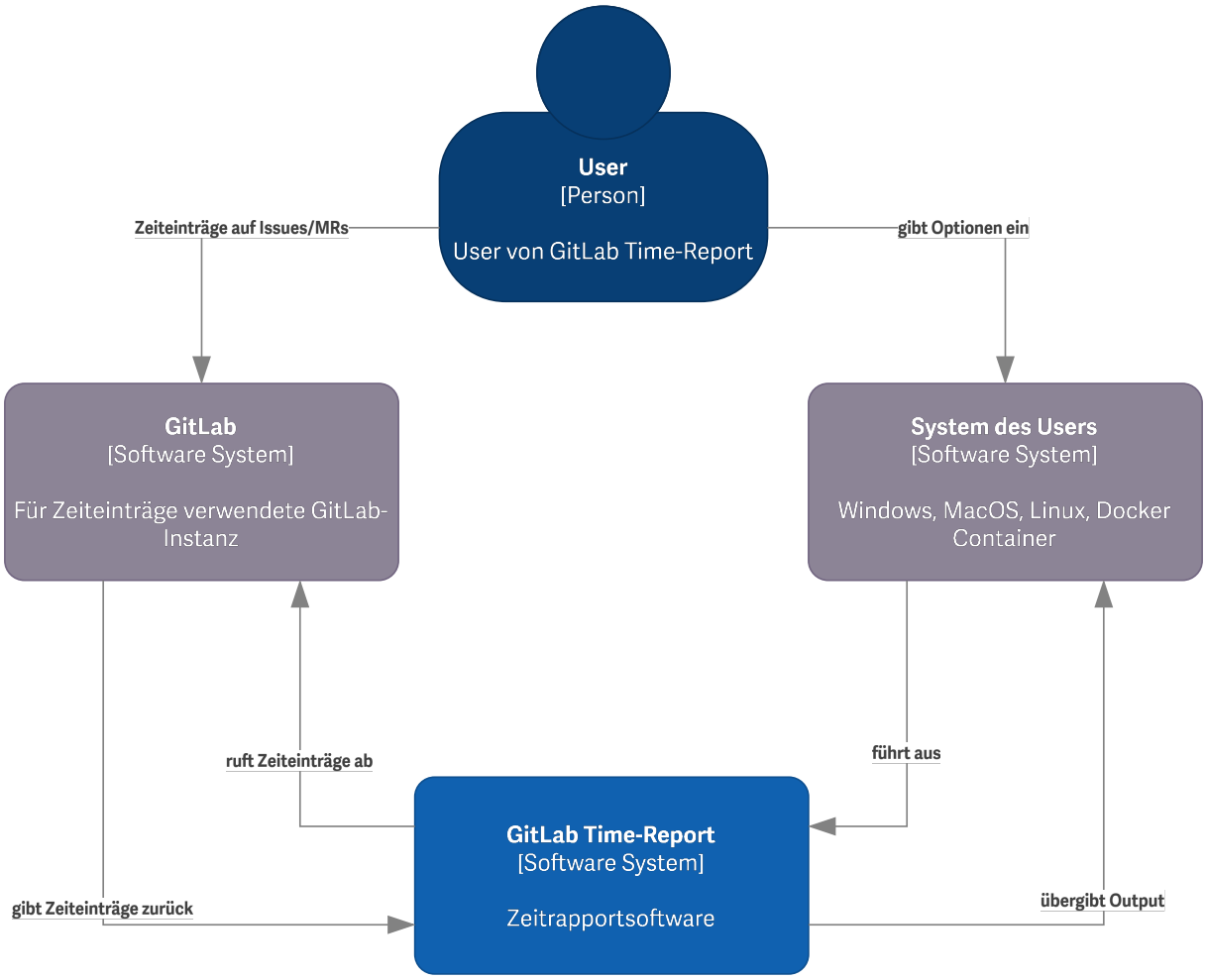


Abbildung 6: C4 System-Kontext-Diagramm

3.3.2 Software-System

In der zweiten Abstraktionsstufe des C4-Diagramms werden die Hauptkomponenten unseres Systems veranschaulicht. Die Applikation gliedert sich in drei Komponenten: Das CLI-Frontend, das GUI-Frontend und die Library, welche die Businesslogik beinhaltet und mit der GitLab-API kommuniziert. Das GUI-Frontend war als Erweiterung geplant, wurde aber aufgrund Zeitmangels nicht mehr implementiert.

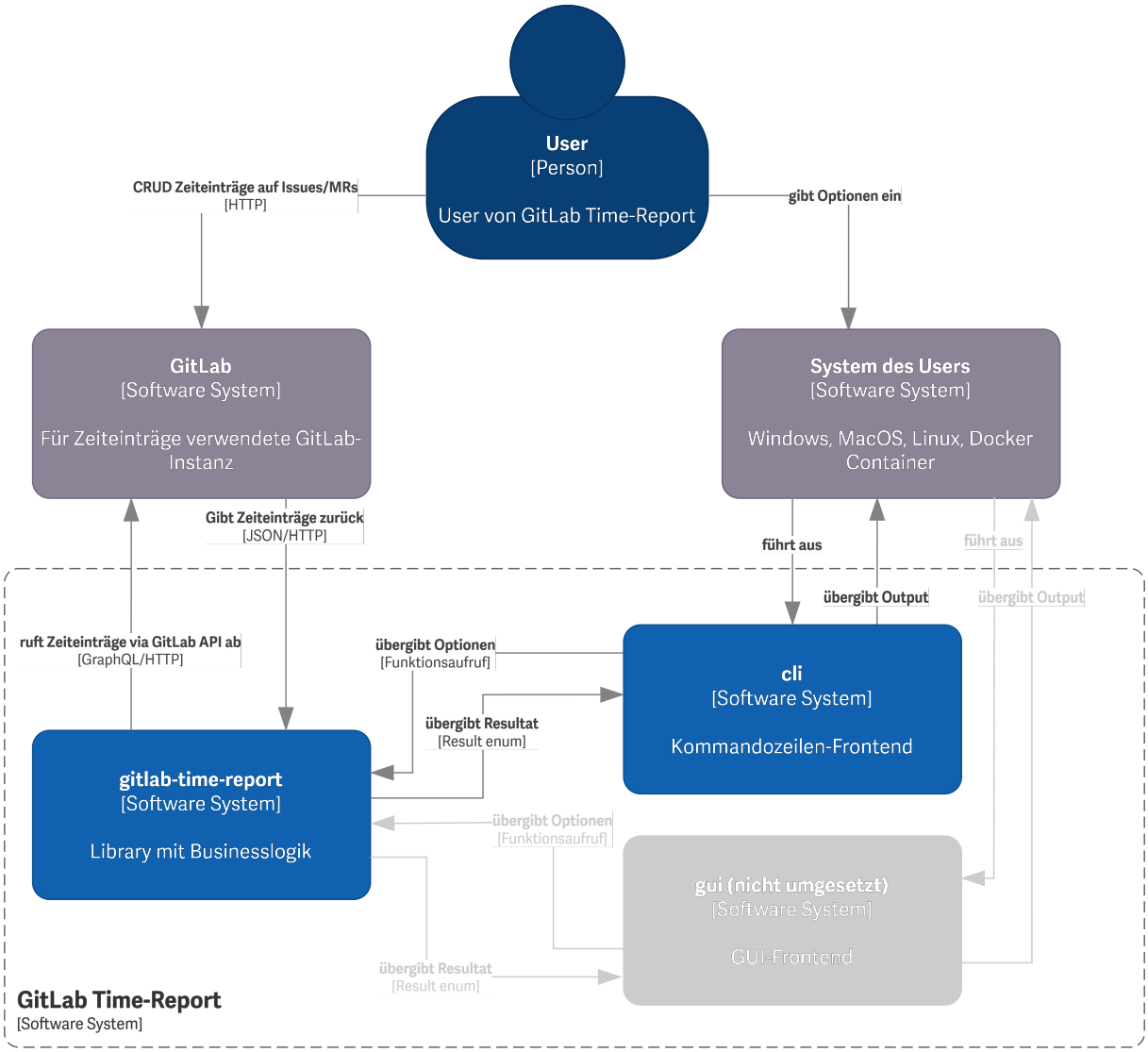


Abbildung 7: C4 Software-System-Diagramm

3.3.3 Komponenten

In der dritten Stufe des C4-Diagramms werden die einzelnen Komponenten des Systems dargestellt, in unserem Fall die Rust-Module und ihre Interaktion untereinander. Die Module sind nach Funktionalität aufgegliedert.

3.3.3.1 CLI

Im der folgenden Abbildung sind die Module unserer CLI-Crate und ihre Funktion aufgelistet. Details dazu befinden sich in Abschnitt 4.2.

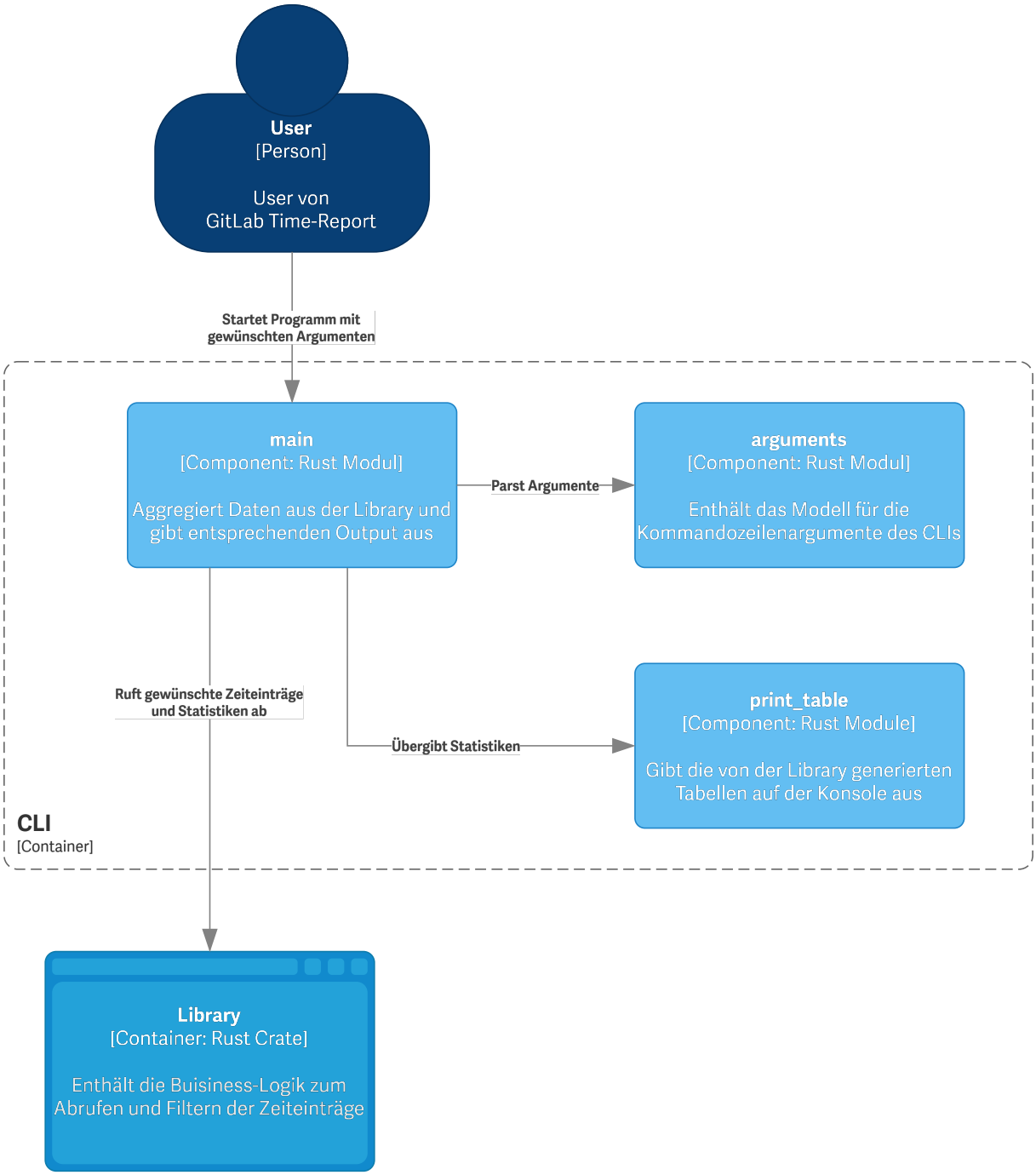


Abbildung 8: C4 Diagramm der CLI-Komponenten

3.3.3.2 Library

Die Abbildung 9 zeigt unsere Library-Module und Ihre Funktionen. Die Implementation dieser ist in Abschnitt 4.1 genauer beschrieben.

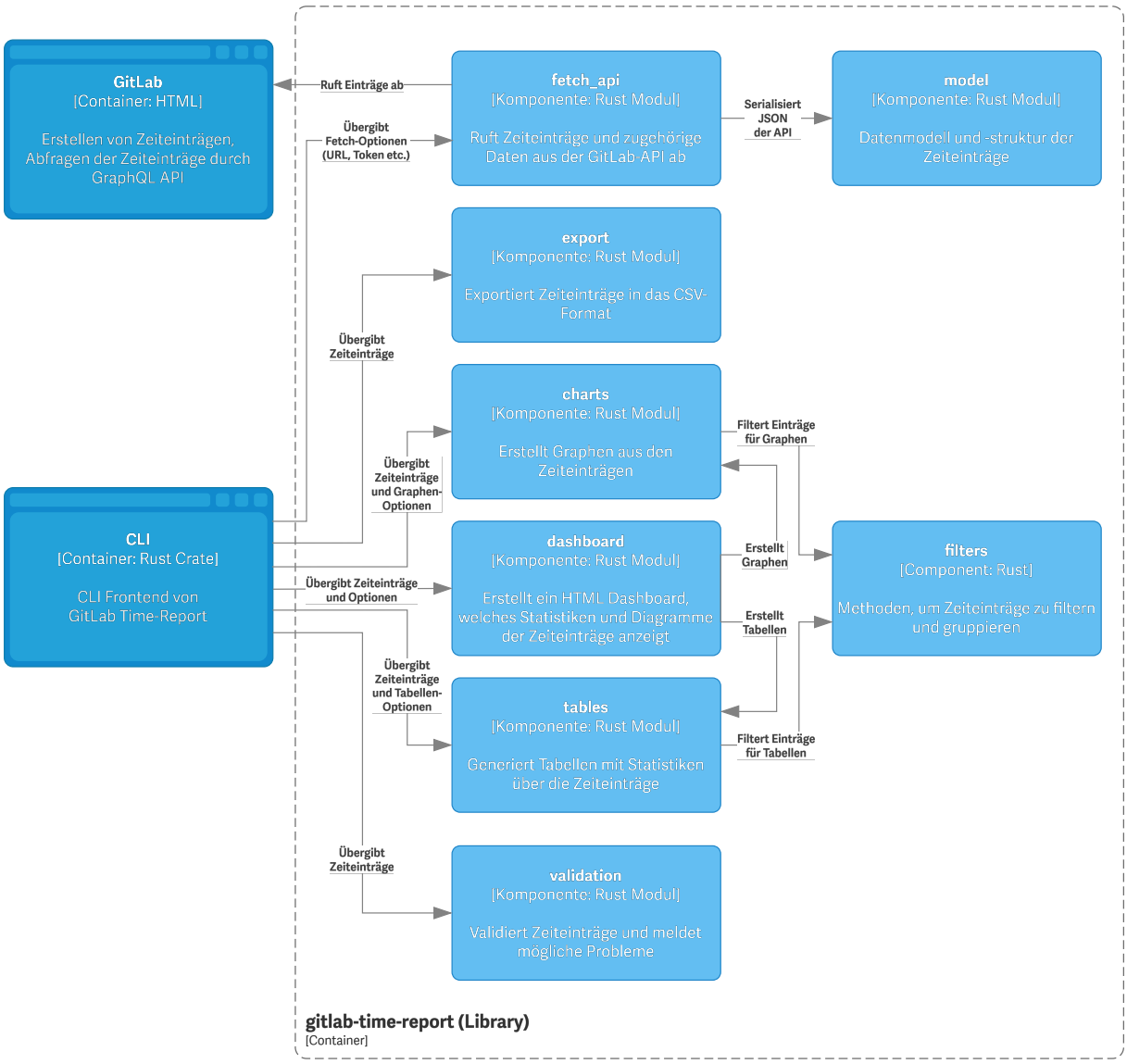


Abbildung 9: C4 Diagramm der Library-Komponenten

4 Implementation

4.1 Library

Wir haben die gesamte Businesslogik in eine Library Crate ausgelagert, damit diese von unterschiedlichen Frontends aus verwendbar ist. Die Library ist in verschiedene Module unterteilt, welche nach Funktionalitäten gegliedert sind.

Eine grafische Übersicht über die Library befindet sich in Abbildung 9.

4.1.1 fetch_api

Das `fetch_api`-Modul kümmert sich um die Kommunikation mit der GitLab-API. Es wird zuerst eine GraphQL-Query für das gewünschte Repository vorbereitet und in JSON umgewandelt. Die `reqwest` Crate [21] sendet dann dieses JSON durch einen POST-Request an die GraphQL-API der spezifizierten GitLab-Instanz. Durch die `serde` und `serde_json` Crates [22], [23] wird das erhaltene JSON in die Structs des Models deserialisiert und dem Client übergeben.

Wir haben uns entschieden, die nicht-asynchrone «Blocking»-Variante des `reqwest`-HTTP-Clients zu verwenden. Der Einsatz von async-Funktionen in Rust benötigt eine Async-Runtime wie beispielsweise `tokio` [24], welche Overhead bei der Laufzeit und beim Debugging mit sich bringt. Da die API-Calls die einzigen Async-Funktionen in der Library wären und der API-Call nicht lange dauert (~5.2s bei 487 Einträgen in unserem SEProj Repo), haben wir den sequenziellen HTTP-Client verwendet.

GraphQL beschränkt die Anzahl Elemente, die mit einem Request abgerufen werden können, auf 100. Dies dient dazu, die API von übermässigen Anfragen zu schützen [25].
Jeder Request beinhaltet das Feld `PageInfo` mit einem Boolean `hasNextPage` und einem `endCursor`, welches den Cursor für das letzte Element auf der Seite beinhaltet. `hasNextPage` ist `true`, so lange es noch eine nächste Seite gibt. Der `endCursor` wird auf der nächsten Seite als `after`-Parameter mitgegeben, damit definiert ist, bei welchem Eintrag diese Seite beginnt [26].

Mithilfe dieser Informationen konnten wir unseren Request so erstellen, dass solange `hasNextPage` auf `true` ist, eine weitere Anfrage mit dem korrekten `after`-Parameter gestellt wird und die zurückgegebenen Daten aneinandergekettet werden. Somit werden erfolgreich alle Zeiteinträge zurückgegeben.

Um die Testbarkeit zu gewährleisten, haben wir für die HTTP-Kommunikation einen eigenen Trait `HttpFetcher` implementiert, welcher den POST-Request der `reqwest` Crate kapselt. Dadurch kann mithilfe der `mockall` Crate [27] in den Unit-Tests ein Mock-Objekt erstellt werden, welches ebenfalls `HttpFetcher` implementiert. Beim Aufruf der Trait-Funktion in den Tests wird der POST-Request simuliert und ein von uns festgelegtes JSON zurückgegeben.

4.1.2 model

Im `model`-Modul wird das in Abschnitt 3.2 beschriebene Domain Model abgebildet. Dies folgt grösstenteils der Struktur, welche von der GitLab-API vorgegeben wurde. Bei der Implementation von `TrackableItem` offenbarte sich aber ein Problem: Da Rust keine Vererbung unterstützt, konnten wir nicht einfach eine Basisklasse erstellen und für Issue und Merge Request davon erben. Zur Umsetzung haben wir deshalb verschiedene Varianten evaluiert.

1. Duplikation aller geteilten Felder in den `Issue` und `MergeRequest` Structs. `TrackableItem` wird als Enum mit Varianten für beide Typen abgebildet, welche den jeweiligen Struct enthalten:

```
enum TrackableItem {
  Issue(Issue),
  MergeRequest(MergeRequest)
}

struct Issue {
    title: String,
    time_estimate: Duration,
    total_time_spent: Duration,
    // ...
}

struct MergeRequest {
    reviewers: UserNodes, // MR specific field
    title: String,
    time_estimate: Duration,
    total_time_spent: Duration,
    // ..
}
```

Listing 3: Variante 1 zur Modellierung von `TrackableItem`: Duplikation

2. Die geteilten Felder in einen eigenen Struct auslagern und diesen als Feld zu `Issue` und `mergeRequest` hinzufügen. `TrackableItem` wird ebenfalls als Enum abgebildet:

```
struct TrackableItemFields {
    title: String,
    time_estimate: Duration,
    total_time_spent: Duration,
    // ...
}

struct MergeRequest {
    reviewers: UserNodes,
    merge_request: TrackableItemFields,
}

struct Issue {
    issue: TrackableItemFields,
}

enum TrackableItem {
  Issue(Issue),
  MergeRequest(MergeRequest)
}
```

Listing 4: Variante 2 zur Modellierung von `TrackableItem`: Auslagerung als eigenes Feld

3. Im `TrackableItem`-Struct ein `kind`-Feld hinzufügen, in welchem ein Enum mit dem entsprechenden Typ und den Struct-spezifischen Feldern abgespeichert wird:

```
struct TrackableItem {
    common: TrackableItemFields,
    kind: TrackableItemKind,
}

enum TrackableItemKind {
    Issue(Issue),
    MergeRequest(MergeRequest),
}

struct Issue {}

struct MergeRequest {
  reviewers: UserNodes,
}
```

Listing 5: Variante 3 zur Modellierung von `TrackableItem`: Typ in separates Feld speichern

Die Varianten 1 und 2 haben den Nachteil, dass bei jedem Zugriff auf ein geteiltes Feld im Code unterschieden werden muss, ob das Objekt ein Issue oder ein Merge Request ist. Somit kann das Liskovsche Substitutionsprinzip nicht angewendet werden (zumindest soweit Rust dieses ohne Vererbung zulässt).

```
let title_of_item = match item {
  TrackableItem::Issue(issue) => issue.issue.title,
  TrackableItem::MergeRequest(mr) => mr.merge_request.title,
};
```

Listing 6: Zugriff auf gemeinsame Felder mit den Varianten 1 und 2

Es wäre zwar möglich, dieses Match-Statement in eine Instanzmethode auszulagern und dann die Felder über einen Funktionsaufruf abzufragen. Diese Methode war uns aber zu unergonomisch.

```
impl TrackableItem {
  pub fn common(&self) -> &TrackableItemFields {
    match self {
      Self::Issue(issue) => &issue.common,
      Self::MergeRequest(mr) => &mr.common,
    }
  }
}
// Accessing title of item:
item.common().title;
```

Listing 7: Instanzmethode für Zugriff auf gemeinsame Felder mit den Varianten 1 und 2

Variante 3 hat dieses Problem nicht, hier kann ohne Typenunterscheidung auf die geteilten Felder zugegriffen werden. Allerdings muss der `Deserialize` Trait für `TrackableItem` selber implementiert werden, weil die Datenstruktur im Programm nicht mehr mit der Struktur des JSONs der GitLab-API übereinstimmt.

Aufgrund der oben genannten Vorteile und einer von uns gestarteten Diskussion im Rust User Forum [28] haben wir uns schlussendlich für Variante 3 entschieden und selber einen Deserializer implementiert. Dies ist Abschnitt 4.1.3 genauer beschrieben.

Wir haben noch versucht, die Felder die in das `TrackableItemFields`-Struct ausgelagert wurden, direkt in `TrackableItem` zu verschieben. Dadurch konnten wir aber die `serde`-Attribute, welche für das Zuordnen und Konvertieren der Feldnamen und Datentypen verwendet werden, nicht mehr verwenden. Diese benötigen jeweils eine derived Implementation von Deserialize, und da wir dieses Struct selbst deserialisieren, war die Verwendung der Attribute nicht mehr möglich.

4.1.3 deserializer

Während der Grossteil der API-Antwort mithilfe der `serde` Crate-Familie durch das `#[derive(Deserialize)]`-Attribut simpel in die Rust-Structs deserialisiert werden kann, mussten wir aufgrund der Entscheidung aus Abschnitt 4.1.2 den Deserializer für `TrackableItem` selbst implementieren. Unser Deserializer führt drei Schritte durch:

1. Es werden innerhalb der `deserialize()`-Funktion Structs definiert, die die Struktur der Issues und Merge Requests abbilden, so wie sie in der Antwort der GitLab-API modelliert sind. Sie werden dann mit dem `#[derive(Deserialize)]`-Attribut versehen.
2. Das JSON wird in die Structs deserialisiert. Eine valide Antwort enthält entweder ein Issue oder einen Merge Request, der andere Eintrag ist jeweils `null` (im JSON) bzw. `None` (in Rust).
3. Aus den Daten dieser Structs kann nun ein «richtiges» `TrackableItem`-Objekt erstellt werden. Das `kind`-Feld beinhaltet die Klassifizierung, ob es sich um ein Issue oder einen MR handelt, sowie die typ-spezifischen Werte. Die gemeinsamen Elemente im werden `common`-Feld gesetzt.

```rust
impl<'de> Deserialize<'de> for TrackableItem {
fn deserialize<D: Deserializer<'de>>(deserializer: D) -> Result<Self, D::Error> {
    // Step 1: Create temporary structs to deserialize the JSON
    #[derive(Deserialize)]
    #[serde(rename_all = "camelCase")]
    struct TrackableItemDeserialize {
        issue: Option<IssueDeserialize>,
        merge_request: Option<MergeRequestDeserialize>,
    }

    #[derive(Deserialize)]
    struct IssueDeserialize {
        #[serde(flatten)]
        common: TrackableItemFields,
    }

    #[derive(Deserialize)]
    struct MergeRequestDeserialize {
        #[serde(flatten)]
        common: TrackableItemFields,
        reviewers: UserNodes,
    }

    // Step 2: Deserialize the trackable item type with the derived deserializer
    let api_trackable_item = TrackableItemDeserialize::deserialize(deserializer)?;

    // Step 3: Create the real trackable item based on which field is non-null
    let real_trackable_item = match (
        api_trackable_item.issue,
        api_trackable_item.merge_request,
    ) {
        (Some(issue), None) => TrackableItem {
            common: issue.common,
            kind: TrackableItemKind::Issue(Issue {}),
        },
        (None, Some(mr)) => TrackableItem {
            common: mr.common,
            kind: TrackableItemKind::MergeRequest(MergeRequest {
                reviewers: mr.reviewers,
            }),
        }, // Checks for invalid combinations removed for brevity
    };
    Ok(real_trackable_item)
  }
}
```

Listing 8: Eigener Deserializer für `TrackableItem`

4.1.4 filters

Das `filters`-Modul enthält Funktionen zum Gruppieren und Filtern von Zeiteinträgen. Die Funktionen nehmen jeweils einen `TimeLog`-Slice entgegen und geben einen Iterator über die Gruppierung beziehungsweise die gefilterten Zeiteinträge zurück.

Wir haben für die Gruppierung eine generische Methode implementiert, welche von den public-Funktionen mit den entsprechenden Parameter aufgerufen werden kann.

```rust
fn group_by_key<'a, T, I, F>(
  time_logs: I,
  predicate: F
) -> impl Iterator<Item = (T, Vec<&'a TimeLog>)>
where
    T: Ord,
    I: IntoIterator<Item = &'a TimeLog>,
    F: Fn(&'a TimeLog) -> T
```

Listing 9: Signatur der generischen Gruppierungsmethode

```rust
pub fn group_by_user(time_logs: &[TimeLog]
) -> impl Iterator<Item = (User, Vec<&TimeLog>)> {
    group_by_key(time_logs, |time_log| time_log.user.clone())
}
```

Listing 10: Aufruf der generischen Gruppierungsmethode mit User als Key

Da die Anzahl der `TimeLogs` sehr gross werden kann, werden diese als Referenz und nicht als Wert übergeben, um unnötiges Kopieren der Daten zu vermeiden.

Die Zeiteinträge sind vom Typ `I`, welcher eine Collection von Referenzen zu mehreren `TimeLog`-Elementen darstellt, die in einen Iterator konvertiert werden kann (`IntoIterator`). Die Gruppierung wird durch eine Prädikatsfunktion `F` generiert, welche als Parameter eine Referenz auf einen `TimeLog` entgegen nimmt und den gewünschten Gruppierungs-Key `T` zurückgibt. Damit verschiedene `T`'s miteinander verglichen werden können, muss `T` den `Ord`-Trait implementieren. `group_by_key()` gibt einen Iterator über ein Tupel von `T` und einen Vector von Referenzen auf `TimeLog` zurück.

In der Methodensignatur sehen wir auch einen expliziten Lifetime Specifier `'a`. In der Signatur sind Referenzen in den Parametern als auch im Rückgabetyp vorhanden, deswegen müssen wir dem Rust-Compiler angeben, in welcher Beziehung diese zueinander stehen. Da alle Referenzen sich auf die selben `TimeLogs` beziehen, haben sie den gleichen Lifetime Specifier.

4.1.5 validation

Das `validation`-Modul überprüft die erhaltenen Zeiteinträge auf ihre Konsistenz. Die Validierung wird über das «Chain of Responsibility»-Design-Pattern vorgenommen [29]. Auf den Struct `TimeLogValidator` können mit der Funktion `with_validator()` Validatoren hinzugefügt werden, welche den `Validator` Trait implementieren. Beim Ausführen von `validate()` wird jeder Zeiteintrag von den Validatoren auf ihr spezifisches Problem überprüft und im `ValidationResult` Struct gespeichert.

```rust
let time_logs = [
    TimeLog{ summary: None, ..Default::default() },
    TimeLog{ summary: Some("Code Review"), ..Default::default() }
];

let mut validator = TimeLogValidator::new()
    .with_validator(ExcessiveHoursValidator::new(10))
    .with_validator(HasSummaryValidator);
let results = validator.validate(&time_logs);

for result in results {
    if result.is_valid() { continue; }
    for problem in &result.problems {
        match problem {
            ValidationProblem::ExcessiveHours { max_hours }
              => println!("Time spent exceeds maximum of {max_hours} hours"),
            ValidationProblem::MissingSummary
              => println!("No summary was entered"),
            _ => {}
        }
    }
}
```

Listing 11: Beispiel zur Verwendung der Validierung

Wir überprüfen die Zeiteinträge auf folgende Probleme:
- Eingetragene Zeit oberhalb von selbst definierbarem Limit
- Fehlende Beschreibung
- Datum liegt vor dem selbst definierten Startdatum oder in der Zukunft
- Duplizierte Zeiteinträge (gleicher Autor, gleiches Datum, gleiche Beschreibung, gleiche eingetragene Zeit)

Die Validierung wird im CLI ausgeführt, bevor die Zeiteinträge weiter verwertet werden.

4.1.6 export

Das `export`-Modul bietet eine Export-Funktion der Zeiteinträge als CSV über das `csv` Modul an. Dazu verwenden wir die `csv`-crate [30]. Sie serialisiert die Daten, welche wir im `TimeLogCsvRow` angeben über die `serde` Crate ins CSV-Format.

Sollte in Zukunft ein weiteres Export-Format angeboten werden, kann ganz einfach ein weiteres Sub-Modul in `export` erstellt werden.

Um zu definieren, welche Inhalte der Zeiteinträge im CSV angezeigt werden sollen, haben wir ein Struct `TimeLogCsvRow` definiert. Folgende Felder möchten wir speziell hervorheben:
- `spent_at` übersetzt das von GraphQL zurückgegebene Format `DateTime<Local>` in ein `NaiveDate`, da wir nur den Tag und nicht die Uhrzeit anzeigen wollen, die von GraphQL immer auf 12:00:00 gesetzt wird.
- `summary` ist ein optionales Feld, wenn dieses nicht gesetzt ist, bleibt die Spalte im CSV leer.
- `trackable_item_type` zeigt an, ob der Eintrag bei einem Issue oder einem Merge request getätigt wurde.

Die Funktion `create_csv_with_writer` nimmt eine Referenz auf einen Vektor von `Timelogs`, einen Pfad und einen definierten Writer entgegen und schreibt für jeden Eintrag die im `struct` definierten Werte in einen Buffer, inklusive Header-Zeile, welche von der `csv`-Crate standardmässig ergänzt wird. Daraus wird dann am vom User definierten Pfad oder, falls dieser nicht angegeben wurde, direkt im aktuellen Arbeitsverzeichnis eine CSV-Datei mit dem Namen `timelogs.csv` erstellt.

Für ein sauberes Error-Handling haben wir einen eigenen `CsvError` mit Varianten für IO, CSV, Finalisierung und UTF-8 erstellt. Die Testbarkeit wird analog zum Modul `fetch_api` (Abschnitt 4.1.1) mit einem Mock-Objekt gewährleistet, welches das Schreiben der CSV-Datei simuliert.

4.1.7 tables

Das `tables`-Modul beinhaltet die Logik für das Erstellen der Tabellen-Inhalte, welche dann sowohl für die Ausgabe in der Konsole als auch für das HTML-Dashboard weiterverarbeitet werden.

Die Funkionen in diesem Modul nehmen jeweils die Zeiteinträge entgegen und geben ein Tupel mit dem Tabellen-Inhalt und dem Tabellen-Header zurück.

Hier wird beispielsweise der Inhalt einer Tabelle erstellt, die pro User die eingetragene Zeit von heute, von gestern, der letzten sieben Tage, der letzten 30 Tage und insgesamt anzeigt.

4.1.8 charts

Das `charts`-Modul generiert aus den Zeiteinträgen verschiedene Diagramme als SVG und HTML. Dazu verwenden wir die `charming`-Crate [31], welche ein Rust-Wrapper um die Apache Echarts JavaScript Library ist [32].

Um Diagramme der Art «Zeit pro X» zu erstellen, wird die Funktion `create_bar_chart()` aufgerufen. Sie ist ein Wrapper um die Funktionalität der `charming`-Crate, welche wir wie mit `request` in `fetch_api` in einen Trait abstrahiert haben. Die Daten werden in die `Series`-Datenstruktur von `charming` umgewandelt, das Diagramm wird erstellt und dann als SVG und HTML gerendert.

Das Generieren der Burndown-Diagramme ist etwas aufwändiger, weswegen wir diese Funktionalität in ein eigenes Submodul ausgegliedert haben. Der Einstiegspunkt ist `create_burndown_chart()`, bei welchem die Zeiteinträge, der Burndown-Typ (Total oder pro Person) und die Optionen für den Burndown-Diagramme übergeben. Details zu den Optionen können in Abschnitt 4.2.1.3 eingesehen werden. Zuerst berechnen wir die gesamt benötigten Stunden und die geleisteten Stunden in jedem Sprint. Daraus erstellen wir die Zeit-Linie pro Person oder total im Projekt. Danach wird die Idealline berechnet, die Beschriftung der X-Achse generiert und das Diagramm gerendert.

Bei der Evaluation der verfügbaren Crates zur Diagramm-Erstellung haben wir uns ursprünglich für die `charts_rs` Library entschieden [33]. Diese ist zwar weniger populär als `charming`, hat aber keine Dependencies auf andere Sprachen – `charming` benötigt die Deno JavaScript Runtime [31]. Wie sich aber bei der Benutzung herausstellte, war der Umgang mit der Crate eher unergonomisch und wir stiessen schnell auf mehrere Bugs, welche wir dem Projekt gemeldet haben [34], [35], [36]. Sie wurden zwar allesamt innerhalb von nützlicher Frist behoben, wir befürchteten aber, dass wir auf weitere Probleme stossen würden.

Aufgrund der oben genannten Probleme haben wir die `charming` Crate nochmals evaluiert. Sie unterstützt neben der Ausgabe als SVG auch die Generierung als HTML. Dieses Format bietet Animationen, Interaktionen und Tooltips für die Diagramme, was ein deutlicher Pluspunkt für das Dashboard darstellt. Auch sind die Diagramme unserer Meinung nach visuell ansprechender.

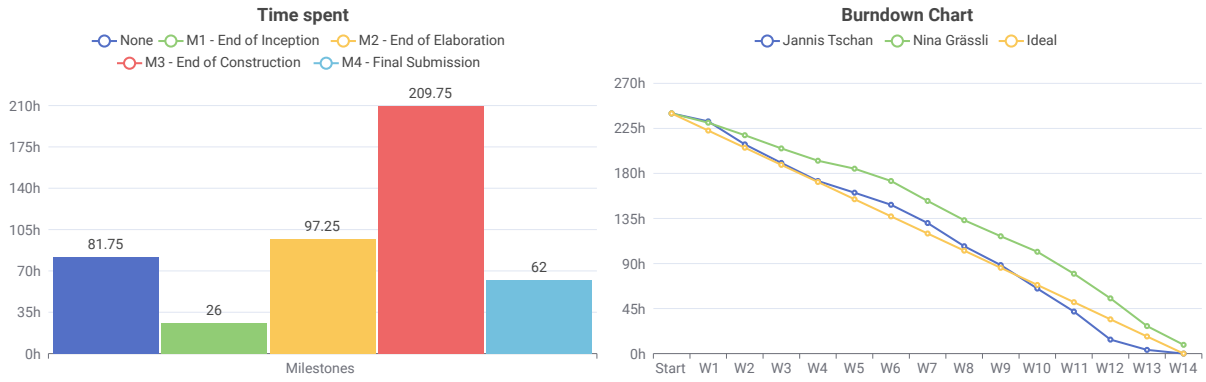


Abbildung 10: Mit `charts_rs` erstellte Diagramme

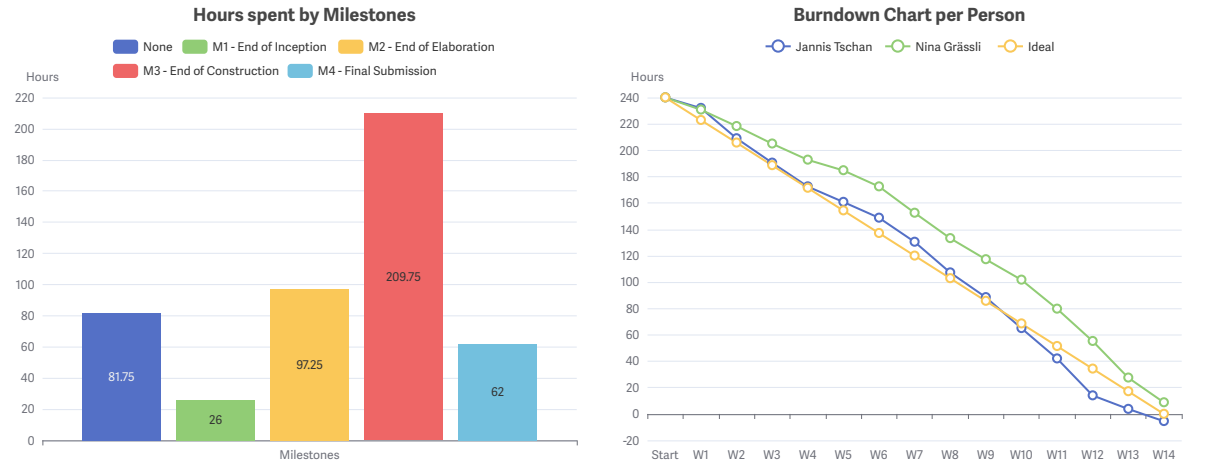


Abbildung 11: Mit `charming` erstellte Diagramme

Ein weiterer Vorteil von `charming` ist, dass die User auf der Apache Echarts-Webseite ein eigenes Theme für ihre Diagramme erstellen können [37]. Die Einstellungen können anschliessend als JSON heruntergeladen und der Pfad darauf GitLab Time-Report mitgegeben werden. Das Theme wird im generierten HTML mithilfe `charming_theme_loader.js` als Theme in Apache Echarts registriert und geladen. So können die User die Diagramme dem Design ihres Projektes anpassen.

Leider gibt es zum Zeitpunkt unserer Arbeit einen Bug in der `charming` Crate: Das Design wird auf die SVG-Dateien angewendet, auf die HTML-Dateien aber nicht. Wir haben dazu ein Issue im Repository der Crate erstellt; ein Fix wurde zwar angekündigt, aber zum Ende dieses Projektes noch nicht veröffentlicht [38]. Als temporärer Workaround injizieren wir das Theme selber in das generierte HTML:

```rust
/// Renders the chart into an HTTP with the theme.
/// If `custom_theme` is `None`, [`Theme::Default`] is used.
fn render_html(&self, width: u64, height: u64, custom_theme: Option<&str>)
→ Result<String, ChartSettingError> {
    let mut html = charming::HtmlRenderer::new("Chart", width, height)
        //.theme(load_theme(custom_theme)) // Broken in charming 0.6.0
        .render(self)?;

    // Workaround to inject theme into HTML due to upstream bug.
    html = html.replace("'chart'))", "'chart'), echartsTheme)");
    let theme_name = match custom_theme {
        Some(_) ⇒ "'custom'",
        None ⇒ "''",
    };

    // Inject setColorTheme() from dashboard, add a fallback for individual HTML chart files
    html = html.replace(
        r#"<script type="text/javascript">"#,
        &format!(
            "<script type=\"text/javascript\">
            const echartsTheme = typeof setColorTheme ⩵ 'function'
              ? setColorTheme({theme_name}) : {theme_name};"
        ),
    );

    let Some(theme) = custom_theme else {
        return Ok(html); // If no theme is specified, return here
    };

    // Inject the JS Theme loader with the theme file contents into the HTML
    let loader = include_str!("charming_theme_loader.js").replace("JSON", theme);
    html = html.replace(r"'custom';", &format!("'custom';\n{loader}"));

    Ok(html)
}
```

Listing 12: Temporärer Workaround für den Upstream-Bug der `charming` Crate

Da wir den `charts_rs`-spezifischen Code in ein eigenes Modul ausgelagert hatten, waren für die Umstellung der Library nur kleine Änderungen in der `mod.rs` nötig.

4.1.9 dashboard

Dieses Modul erstellt eine HTML-Datei `dashboard.html`, welches sowohl die Diagramme des `charts`-Moduls, als auch die Tabellen des `tables`-Moduls anzeigt.

Um die Inhalte im HTML anzeigen zu können, verwenden wir die Crate `build_html`. Diese Crate bietet die Funktionalität, direkt im Rust Code HTML Strings zu generieren und diese zu einer kompletten HTML-Datei zusammenzusetzen [39].

Damit nicht die gesamte HTML-Datei via Rust geschrieben werden muss, haben wir eine Datei `base.html` erstellt, welche die Grundlage für das Dashboard liefert. Das gesamte Styling und die verwendeten JavaScript Funktionen sind in dieser Datei bereits hinterlegt. Auch Platzhalter für die dynamischen Inhalte sind in der Datei vorhanden (Abbildung 12).

Wir haben dabei keine Template-Engine verwendet, da wir nur wenige Stellen einmalig ersetzen und uns deshalb eine weitere Abhängigkeit ersparen wollten.

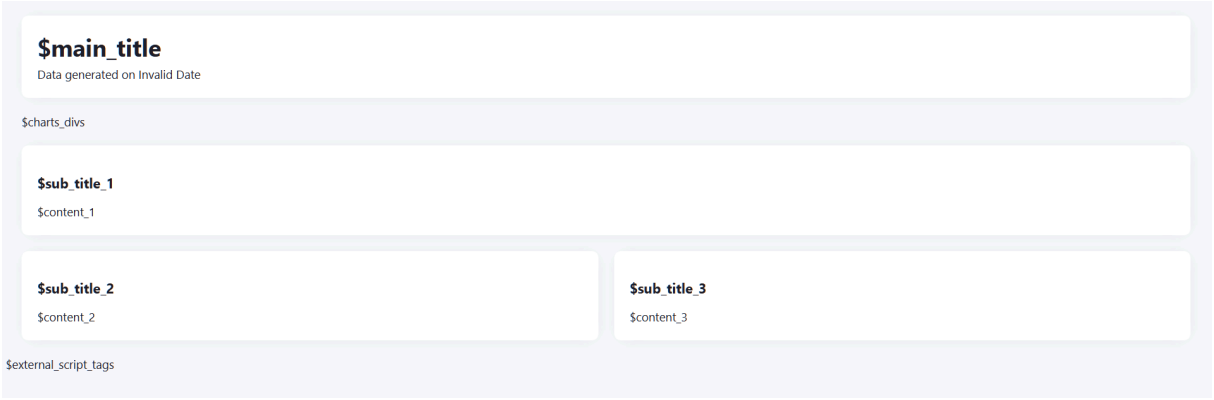


Abbildung 12: Ansicht des Dashboards, bevor der Inhalt eingefügt wird

4.1.9.1 Tabellen

Die Tabellen, die vom Modul `tables` vorbereitet werden, können mit dieser Crate relativ einfach in HTML-Tabellen umgewandelt werden:

```
let (table_data, table_header) = populate_table_timelogs_in_timeframes_by_user(time_logs);
let table = Table::from(table_data).with_header_row(table_header);
```

Listing 13: Umwandlung der bestehenden Tabellen in HTML-Tabellen

Damit die Tabellen sortierbar sind, haben wir selber eine JavaScript-Funktion geschrieben, um nicht von einer weiteren Library abhängig zu sein. Bei der Tabelle, die die Zeiten pro Person anzeigt, mussten wir einen Spezialfall berücksichtigen: Die Zeile mit den aufsummierten Zeiten, sollte immer ganz unten bleiben. Dazu mussten wir die relevanten Daten aus der von `populate_table_timelogs_in_timeframes_by_user()` zurückgegebenen `table_data` extrahieren und in den Tabellen-Footer einfügen, damit sie von der Sortierung nicht betroffen sind.

Die Tabellen werden nach dem Erstellen in die Platzhalter `$content_1` bis `$content_3` eingefügt. Auch die Tabellenüberschriften werden dynamisch ergänzt.

4.1.9.2 Diagramme

Um die Diagramme für das Dashboard aufzubereiten, ist etwas mehr Aufwand nötig. Wir erhalten diese vom Modul `charts` als SVG und als HTML Dateien. Da die HTML-Dateien interaktiv sind, verwenden wir diese für das Dashboard. Jedoch handelt es sich um eigenständige HTML-Dateien, welche direkt im Browser angezeigt werden können. Um komplette HTML-Dateien einzubinden, würde sich auf den ersten Blick der `iframe`-Tag anbieten. In unserem Fall ist diese Methode aber keine gute Lösung: Die Höhe des eingebetteten Dokuments lässt sich nur schlecht dynamisch an den Inhalt anpassen, was zu einem schlechtem Nutzererlebnis führt. Also mussten wir eine andere Lösung finden. Dafür haben wir uns den Aufbau der Diagramme etwas genauer angeschaut.

```
...
<div class="container">
  <div class="item" id="chart"></div>
</div>
<script type="text/javascript">
    var chart = echarts.init(document.getElementById('chart'));
    var option = { /* Chart is defined here */ }
    chart.setOption(option);
</script>
...
```

Listing 14: Aufbau eines HTML-Diagramms der `charming`-Library

Im Listing 14 ist der relevante Part der Datei ersichtlich. Die ganze Logik des Diagramms is in einem `script`-Tag definiert. Wir benötigen also nur diesen Tag, wobei wir nur die ID im `getElementById` anpassen müssen, um mehrere Diagramme in der gleichen HTML-Datei anzeigen zu können.

Das ganze haben wir automatisiert. Wir lesen alle Dateien im `charts`-Verzeichnis mit einer `html`-Extension ein und speichern alle vorhandenen Pfade in einem Vector. Anschliessend sortieren wir die Pfade nach Dateinamen, um eine deterministische Reihenfolge sicherzustellen.

Im nächsten Schritt werden die Diagramme nummeriert und das JavaScript extrahiert. Im HTML-String werden so viele `div`-Tags ergänzt, wie es Diagramme gibt. Die Tags werden mit nummerierten IDs versehen und die `getElementById`-Funktion der Diagramme wird auf die korrekte ID angepasst. Anschliessend werden auch die `script`-Tags in das Template injiziert.

So ist es uns gelungen, die Diagramme inklusive Interaktivität in unser Dashboard einzugliedern, ohne auf `iframes` zurückgreifen zu müssen.

4.1.9.3 Dark und Light Mode

Das Dashboard übernimmt automatisch die Einstellung der Benutzer:innen bezüglich Light und Dark Mode. Dies lässt sich mithilfe von CSS-Variablen und dem `prefers-color-scheme` CSS Media Feature sehr einfach umsetzen [40].

Damit sich die Diagramme ebenfalls der präferierten Einstellung bezüglich Farbschema anpassen, mussten wir beim Code der Diagramme eine kleine Änderung vornehmen:

```
...
// var chart = echarts.init(document.getElementById('chart')); // alt
var chart = echarts.init(document.getElementById('chart'), echartsTheme); // neu
...
```

Listing 15: Änderung an den Diagrammen für Light und Dark Mode

Damit wir auch den Upstream-Bug der `charming`-Library umgehen konnten, wie in Abschnitt 4.1.8 beschrieben, mussten wir diese Änderung direkt in den einzelnen HTML-Dateien der Diagramme umsetzen. Die Variable `echartsTheme` wird nun mit JavaScript auf `""` oder auf `"dark"` gestellt. Ein leerer String ist der Default-Wert und bedeutet, dass das Standard Light Theme verwendet wird. Wenn ein Nutzer ein eigenes Theme spezifiziert, wird stattdessen der Name des Themes angegeben.

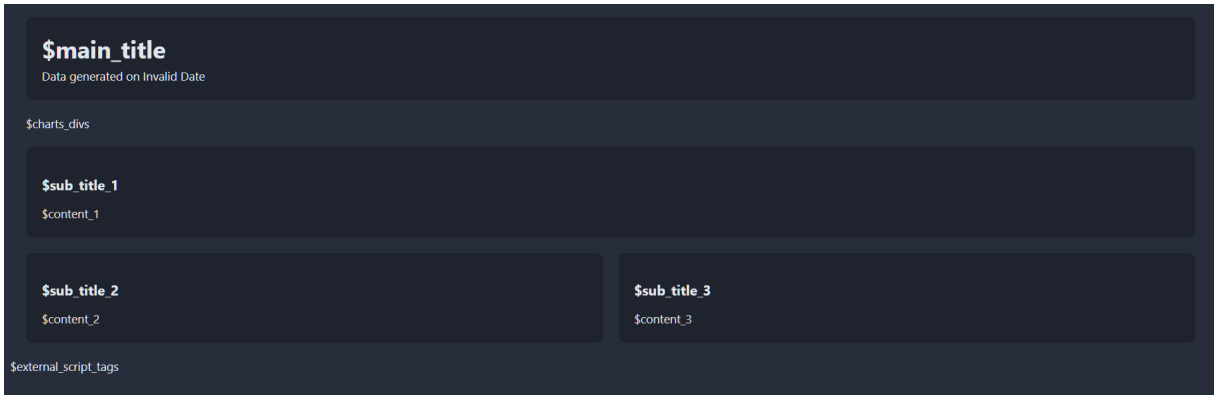


Abbildung 13: Ansicht des Dashboards im Dark Mode, bevor der Inhalt eingefügt wird

4.1.9.4 Zeitstempel

Das Dashboard beinhaltet einen Zeitstempel, der den Zeitpunkt der Dashboard-Generierung anzeigt. Dieser wird serverseitig bei der Generierung des Dashboards mit Rust gesetzt. Da jedoch Projekt-Beteiligte in unterschiedlichen Zeitzonen auf das Dashboard zugreifen können und der angezeigte Zeitpunkt dann nicht mehr ihrer lokalen Zeit entspricht, haben wir eine JavaScript-Funktion ergänzt, die das übergebene Datum mit Uhrzeit beim Laden der Seite in das entsprechende lokale Datum- und Zeitformat der jeweiligen Zeitzone anpasst.

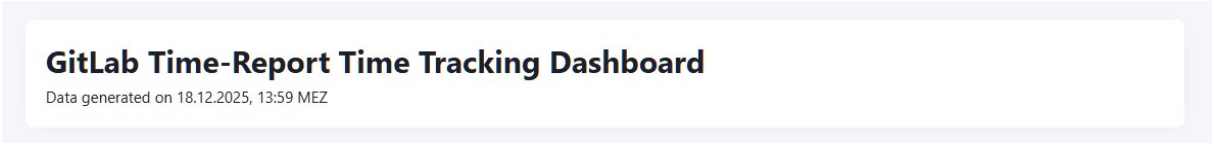


Abbildung 14: Zeitstempel bei Ansicht des Dashboards in der Schweiz

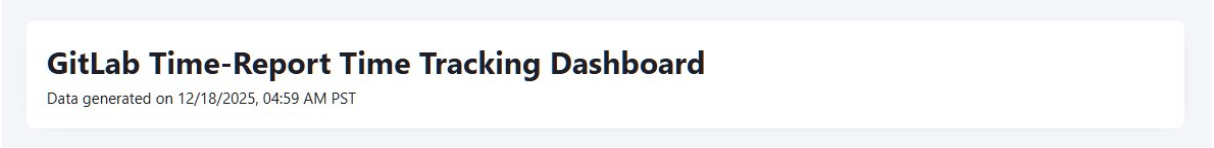


Abbildung 15: Zeitstempel bei Ansicht des Dashboards aus San Francisco

Das finale Dashboard ist in Abbildung 1, Abbildung 2 und Abbildung 21 ersichtlich.

4.2 CLI-Frontend

Das CLI-Frontend bietet eine einfache Handhabung unseres Programms über die Konsole. Dafür verwenden wir die `clap` Crate [41]. Diese Crate ermöglicht es uns, auf eine einfache Art und Weise eigene Argumente und Subcommands zu definieren.
Es gibt zwei verschiedene Möglichkeiten, `clap` zu verwenden: Die Derive-API oder die Builder-API.

Die Derive-API arbeitet mit Structs und Enums, welche von `clap` bereitgestellte `#[derive]`-Attribute verwenden. Die `clap`-Dokumentation liefert das folgende Beispiel [42]:

```rust
#[derive(Parser)]
#[command(version, about, long_about = None)]
struct Cli {
    name: Option<String>,
    /// Sets a custom config file
    #[arg(short, long, value_name = "FILE")]
    config: Option<PathBuf>,
    #[command(subcommand)]
    command: Option<Commands>,
}

#[derive(Subcommand)]
enum Commands {
    /// does testing things
    Test {
        /// lists test values
        #[arg(short, long)]
        list: bool,
    },
}

fn main() {
  let cli = Cli::parse();
  // You can check the value provided by positional arguments, or option arguments
  if let Some(name) = cli.name.as_deref() {
      println!("Value for name: {name}");
  }
  ...
}
```

Listing 16: Beispiel zur Verwendung der Derive-API von `clap`

Mit diesem Ansatz können die Argumente und Subcommands mit einem deklarativem Ansatz beschrieben werden.

Die `Builder`-API hingegen basiert auf dem imperativen Ansatz. Auch hier ein Beispiel aus den Rust-Docs [43]:

```rust
fn main() {
    let matches = command!() // requires `cargo` feature
        .arg(arg!([name] "Optional name to operate on"))
        .arg(
            arg!(-c --config <FILE> "Sets a custom config file")
            .required(false)
            .value_parser(value_parser!(PathBuf)),
        )
        .subcommand(
            Command::new("test")
                .about("does testing things")
                .arg(arg!(-l --list "lists test values").action(ArgAction::SetTrue)),
        )
        .get_matches();

    // You can check the value provided by positional arguments, or option arguments
    if let Some(name) = matches.get_one::<String>("name") {
        println!("Value for name: {name}");
    }
    ...
}
```

Listing 17: Beispiel zur Verwendung der Builder API von `clap`

Wir mussten uns zu Beginn dieser Arbeit für eine Version entscheiden. Glücklicherweise bietet `clap` eine Entscheidungshilfe in den FAQs [44].

Derive API	Builder API
– Einfacher zu lesen, schreiben und ändern – Einfacher, die Argument-Deklaration und das Lesen von Argumenten synchron zu halten – Einfacher wiederzuverwenden	– Schnellere Kompilierung, wenn noch keine andere prozedurale Makros verwendet werden – Flexibler als die Derive API

Tabelle 9: Vergleich zwischen der Derive und Builder API von `clap`

Da wir das erste Mal mit dieser Crate arbeiten und sich die Komplexität unseres Tools in Grenzen hält, haben wir uns an die Empfehlung von den Machern der Crate gehalten und die Derive API ausgewählt.

4.2.1 arguments

Die Argumente und Subcommands für unser CLI-Frontend haben wir in `arguments.rs` definiert. Die Flags `--help` und `--version` werden von Clap automatisch generiert. Ersteres zeigt eine kurze Erklärung der Commands und Argumente an, letzteres gibt die Versionsnummer aus dem `Cargo.toml` aus.

4.2.1.1 Argumente

Name	Funktion
`url: String`	URL des GitLab-Repositories, aus dem der User die Zeiten extrahieren will.
`token: Option<String>`	Falls das Repository privat ist, benötigt man für den Zugriff einen Token, wie in Abschnitt 3.1.1.1 beschrieben. Ist das Repository privat und der User gibt keinen Token an, wird eine Fehlermeldung mit Anleitung zur Erstellung eines Tokens angezeigt.
`labels: Vec<String>`	In diesem Argument können kommasepariert Labels definiert werden, die in den Tabellen und Charts angezeigt werden. Alle nicht spezifizierten Labels werden unter «Other» gruppiert. Wird kein Label angegeben, werden alle existierenden verwendet.
`validation_details: bool`	Mit Angabe dieser Flag können alle Zeit-Einträge, die bei der Validierung durchgefallen sind, auf der Konsole ausgegeben werden.
`validation_max_hours: u16`	Mit dieser Flag kann der Grenzwert für die Validierung «Eingetragene Zeit oberhalb von selbst definierbarem Limit» gesetzt werden. Der Standardwert ist 10h.

Tabelle 10: Argumente unseres CLI-Frontends

4.2.1.2 Subcommand export

Wenn der Subcommand `export` angegeben wird, werden die Einträge in eine CSV-Datei geschrieben. Es gibt dazu folgende Argumente:

Name	Funktion
`output: PathBuf`	Pfad zur CSV-Datei, die erstellt werden soll. Wird kein Pfad angegeben, wird im Projekt-Ordner eine Datei `timelogs.csv` erstellt.

Tabelle 11: Argumente des Subcommands `export`

4.2.1.3 Subcommand charts

Mit diesem Befehl kann der User verschiedene Diagramme mit Statistiken generieren. Der Befehl nimmt ein `ChartOptionsArgs` entgegen, in welchem die Argumente definiert sind. Diese Verschachtelung war notwendig, da sowohl der Befehl `charts` als auch der Befehl `dashboard` die gleichen Argumente benötigen.

4.2.1.4 Subcommand dashboard

Wird dieser Befehl angegeben, wird ein HTML-Dashboard mit Statistiken generiert. Auch dieser Befehl nimmt ein `ChartOptionsArgs` entgegen.

4.2.1.5 ChartOptionsArgs für Dashboard und Charts

Die Diagramme werden sowohl mit dem Befehl `charts` als auch mit dem Befehl `dashboard` generiert. Die dafür verwendbaren Argumente sind die folgenden:

Name	Funktion
`width: u16`	Die Breite der Diagramme. Wenn keine Angabe gemacht wird, werden 800 Pixel verwendet.
`height: u16`	Die Höhe der Diagramme. Wenn keine Angabe gemacht wird, werden 480 Pixel verwendet.
`theme_json: Option<PathBuf>`	Die User können, wie in Abschnitt 4.1.8 beschrieben, ein eigenes Theme für ihre Diagramme erstellen und als JSON speichern. Den Pfad zu dieser Datei kann diesem Argument mitgegeben werden.
`output: PathBuf`	Standardmässig werden die Diagramme im Folder `charts` im Projekt-Ordner abgelegt. Mit diesem Argument kann der User einen alternativen Pfad angeben, an welchem die Diagramme gespeichert werden.
`sprints: u16`	Mit diesem Argument muss angegeben werden, in wie viele Sprints das Projekt aufgeteilt ist. Hat ein Projekt keine Sprints, muss hier die Anzahl Wochen angegeben werden. Dieses Argument muss angegeben werden, da sonst kein Burndown-Diagramm erstellt werden kann.
`weeks_per_sprint: u16`	Mit diesem Argument muss angegeben werden, wie lange ein Sprint dauert.
`hours_per_person: f32`	Hier muss definiert werden, wie viele Stunden jede Person in das Projekt insgesamt investieren sollte. Auch dieser Wert ist für das Burndown-Diagramm relevant und muss deshalb zwingend angegeben werden.
`start_date: Option<NaiveDate>`	Dieser Wert wird ebenfalls für das Burndown-Diagramm benötigt. Wird kein Start-Datum angegeben, wird das Datum des ersten Zeit-Eintrags verwendet.

Tabelle 12: Argumente der Subcommands `charts` und `dashboard`

4.2.2 print_table

Im `print_table`-Modul werden die in der Library erstellten Tabellen-Inhalte (vgl. Abschnitt 4.1.7) mit der Crate `cli_table` zu Tabellen weiterverarbeitet, welche dann auf der Konsole ausgegeben werden, wenn kein Subcommand angegeben wurde.

4.2.3 main

Die Datei `main.rs` enthält den Einstiegspunkt der Applikation. Sie ruft die `parse()`-Funktion von `clap` auf, um an die Argumente und Subcommands zu gelangen. Die ersten Schritte sind für alle Subcommands gleich: Zuerst werden die Zeiteinträge abgerufen und die Validierung durchgeführt. Die Validierungs-Resultate werden für die Ausgabe auf der Konsole formatiert und anschliessend ausgegeben.

Tritt beim Abrufen der Zeiteinträge ein `ProjectNotFound`-Fehler auf, gibt das Programm eine eigene Fehlermeldung aus, welche auf einen möglicherweise benötigten Access Token hinweist.

Anschliessend werden je nach Subcommand die Optionen-Structs mit den Argumenten erstellt und die entsprechenden Library-Funktionen aufgerufen. Auf der Konsole werden entsprechend der Ausführung Fehler- beziehungsweise Erfolgsnachrichten ausgegeben.

5 Qualitätssicherung

5.1 Guidelines

Die Guidelines definieren, welche Workflows und Bewertungskriterien wir bei der Arbeit am Projekt angewendet haben.

5.1.1 Versionskontrolle

Für Versionskontrolle wird Git mit GitLab als remote Repository verwendet. Sämtliche Code- und Dokumentationsbeiträge zum Projekt müssen mit Git verwaltet werden und auf das GitLab-Repo gepusht werden. Die Commit Messages werden nach dem «Conventional Commits»-Standard verfasst [45]. Damit wird ein klarer Überblick geboten, welcher Teil des Projekts und was daran geändert wurde. Falls nötig, kann eine mehrzeilige Commit Message verwendet werden, in welcher die Änderung genauer spezifiziert wird, auf bekannte Probleme aufmerksam gemacht wird oder ähnliches.

5.1.1.1 Issues

Sämtliche User Stories und Tasks werden mithilfe eines GitLab Issues getrackt. Sie enthalten alle wichtigen Informationen in der Issue-Beschreibung und nicht in den Kommentaren. Diese dürfen aber für Diskussionen und Updates über das Issue verwendet werden. Die Issues erhalten jeweils ein Kategorisierungs-, ein Progress- und ein EPIC-Label. Die Kategorien beziehen sich auf die Art des Tasks (z.B. Projektplanung, CI/CD, Feature), das Progress-Label auf den aktuellen Status (In Planning, In Progress, Ready to Merge) und das EPIC-Label zeigt, zu welchem EPIC das Issue gehört.

Die Zeiterfassung des Projektes erfolgt über Issues. Aus Übersichtsgründen sollte deshalb auch die Zeit, die für das Erstellen und Überprüfen eines Merge Requests benötigt wird, beim Issue und nicht direkt beim MR eingetragen werden.

5.1.1.2 Definition of Ready (DoR)

Issues werden im GitLab Issue Board von «Sprint Progress» nach «Sprint Review» verschoben, wenn der dazugehörende Merge Request erstellt wird. Der MR wird erst erstellt und dem anderen Team-Mitglied zum Review zugewiesen, wenn die folgende Checkliste abgearbeitet wurde. Für die Dokumentation muss nur der erste Punkt erfüllt sein.

- ☐ Alle zum Issue gehörenden Tasks wurden erledigt
- ☐ Tests für neue Funktionalitäten wurden geschrieben
- ☐ Alle Tests laufen durch, inklusive den neuen

Sollen Änderungen bereits vor Erfüllung dieser Kriterien geprüft werden, soll der Merge Request als Draft erstellt werden. Nachdem die Kriterien erfüllt wurden, kann der Merge Request als «ready» markiert werden und folgt dann dem normalen Vorgehen.

5.1.1.3 Definition of Done (DoD)

Issues werden im GitLab Issue Board von «Sprint Review» nach «Closed» verschoben, wenn alle Punkte auf den folgenden Checklisten abgehakt werden können und der dazugehörende Merge Request somit bewilligt wird.

DoD bei Dokumentations-Änderungen

- ☐ Die Änderungen im Source-Branch behandeln die im Issue genannten Themen
- ☐ Die Dokumentation kompiliert
- ☐ Die neuen Inhalte sind klar strukturiert und ohne Schreibfehler
- ☐ Der Source-Branch wurde nach den Richtlinien in Abschnitt 5.1.1.4 auf den Target-Branch gerebased
- ☐ Die Pipeline im Merge Request ist erfolgreich

DoD bei der Entwicklung

- ☐ Die Änderungen im Source-Branch machen das, was sie laut Issue-Beschreibung machen sollen
- ☐ Alle Tests werden erfolgreich ausgeführt
- ☐ Eine kurze manuelle Überprüfung des Tools zeigt kein Fehlverhalten
- ☐ Die Code-Qualität entspricht den Guidelines
- ☐ Die Test-Coverage fällt durch das Mergen der Änderungen um maximal 3%
- ☐ Der Source-Branch wurde nach den Richtlinien in Abschnitt 5.1.1.4 auf den Target-Branch gerebased
- ☐ Die Pipeline im Merge Request ist erfolgreich

Die Checklisten wurden im GitLab als Merge Request Template hinterlegt, sodass diese bei jedem Merge Request ersichtlich sind und die Punkte abgehakt werden können. Das erfolgreiche Durchlaufen der CI/CD-Pipeline im Merge Request wird automatisch durch GitLab überprüft, da wir die Option «Pipelines must succeed» aktiviert haben. Deswegen haben wir diesen Punkt nicht separat im Template aufgeführt.

5.1.1.4 Git-Strategie

Im Projekt wird eine «Git Feature Branch»-Strategie mit Rebasing und Squashing verfolgt [46]. Wir haben uns für diesen Workflow entschieden, da wir in unserem Projekt keine regelmässigen Releases oder Deployments durchführen und damit ein eigener `develop`-Branch, wie er beispielsweise bei «Git Flow» [47] verwendet wird, überflüssig ist.

Für jede Änderung am Code oder der Dokumentation wird ein neuer Branch aus `main` erstellt, der «Feature-Branch». Es werden nur auf diesem Branch Commits erstellt. Sind die Änderungen bereit zum Review nach den Kriterien in Abschnitt 5.1.1.2, wird ein Merge Request (MR) erstellt und die andere Person als Reviewer hinzugefügt. Der Reviewer führt dann ein Code-Review durch und überprüft, ob die Kriterien in Abschnitt 5.1.1.3 erfüllt sind.

Genehmigt der Reviewer die Änderung, kann der MR in den `main`-Branch gemerged werden. Davor soll der zu mergende Branch auf `main` rebased werden. Dadurch wird sichergestellt, dass in der Git History nur fast-forward Merges stattfinden. Um Rebases zu erzwingen, haben wir in GitLab die Option «Merge commit with semi-linear history» aktiviert. Dies erlaubt keinen Merge, wenn vorher kein Rebase ausgeführt wird. Durch das Rebasing entspricht die Git-History nicht immer der Realität, dafür ist sie aber äusserst linear, übersichtlich, nachvollziehbar und verläuft stets vorwärts. Zudem können Merge Konflikte dadurch bereits beim Rebasing gelöst werden.

Beim Mergen sollen die Commits gesquasht werden. Auch dies dient der Übersichtlichkeit der Git History auf dem `main`-Branch: Da in einem MR häufig viele Commits durchgeführt werden, kann die Git History ohne Squashing schnell unübersichtlich werden. Beim Squashing werden alle Commits in einem MR beim Mergen zu einem einzigen Commit zusammengefügt. Dafür haben wir in GitLab die Option «Encourage Squash commits when merging» aktiviert, damit Squashing bei jedem MR automatisch aktiviert ist. Auch nach dem Merge kann beim entsprechenden MR die genaue Commit History eingesehen werden.

5.1.1.5 Merge Requests

Folgende Punkte sollten in einem Merge Request enthalten sein:

- **Assignee:** Die Person, welcher die Änderung im Sprint Planning zugeteilt wurde.
- **Reviewer:** Die Person, welche das Code-Review durchführt.
- **Issue ID:** Das Issue, welches zu diesem MR gehört, wird in der MR-Beschreibung verlinkt. Dies erleichtert die Auffindbarkeit der entsprechenden Issues/MRs im Repo. Wird mit einem Merge ein Issue abgeschlossen, kann ein Closing Pattern verwendet werden, damit das Issue automatisch geschlossen wird (z.B. «Closes #1»).
- **Issue Template:** Beim Erstellen soll das Issue-Template ausgewählt werden, welches die entsprechenden DoD-Kriterien enthält.

5.1.2 Code

Wir orientieren uns beim Entwickeln an den SOLID-Prinzipien. Diese wurden ursprünglich für die Verwendung in OOP-basierten Sprachen entworfen. Da Rust einige OOP-Konzepte wie Vererbung nicht besitzt, lassen sich die Konzepte nicht 1:1 auf Rust-Code anwenden. Die Eckpunkte der SOLID-Prinzipien verwenden wir dennoch:

- **Single Responsibility Principle:** Jede eigenständige Funktionalität wird in ein eigenes Modul ausgegliedert. Innerhalb eines Moduls befinden sich nur Methoden, welche sich mit dieser Funktionalität befassen. Wird das Modul zu gross, können einzelne Teile in Submodule ausgegliedert werden. Innerhalb der Module sind einzelne Funktionen so aufgebaut, dass diese genau eine Aufgabe erfüllen.
- **Open-Closed Principle:** Per Default sind Funktionen, Structs und deren Felder `private`, was keine externen Modifikationen an den Klassen erlaubt. Grundsätzlich sind Methoden so zu gestalten, dass bei Ergänzungen keine Änderungen an anderen Stellen notwendig sind. Das Verwenden von `match`-Statements, welche jede Möglichkeit abdecken müssen, erschwert diese Tatsache aber. Unserer Meinung nach überwiegt deren Nutzen aber, weswegen wir Verletzungen des Prinzips in Kauf nehmen.
- **Liskov Substitution Principle:** Da Rust keine Vererbung wie aus anderen OOP-Sprachen bekannt besitzt, kann dieses Prinzip nicht direkt angewendet werden. Wir wenden jedoch Traits an. Sie funktionieren wie Interfaces in anderen Programmiersprachen. Jedes Element, welches einen Trait implementiert, kann durch ein anderes Objekt substituiert werden, welches diesen Trait ebenfalls implementiert.
- **Interface Segregation Principle:** Traits sollen nur so viele Funktionen wie nötig anbieten. Wenn ein implementierender Typ eine Trait-Methode nicht benötigt, wird der Trait aufgeteilt.
- **Dependency Inversion Principle:** Bei Dependencies auf externe Ressourcen wie Netzwerk-Verbindungen oder Dateisysteme wird ein Trait implementiert, welcher den Zugriff darauf abstrahiert. In API-Funktionen der Crate wird jeweils die Dependency initialisiert und dann eine `impl`-Variante der Funktion aufgerufen, welche die Dependency als zusätzlichen Parameter entgegen nimmt und die eigentliche Funktionalität ausführt. So lässt sich die Dependency für Unit-Tests mocken.

5.1.2.1 Linter

Wie in Tabelle 3 beschrieben, setzen wir den in der Rust-Toolchain integrierten Linter «Clippy» ein [48]. Er überprüft Code auf mögliche Ungereimtheiten und warnt den User davor. Die Lints teilen sich dabei in verschiedene Kategorien auf, zum Beispiel `clippy::correctness` (Code, der völlig falsch oder nutzlos ist), `clippy::style` (Code, der idiomatischer geschrieben werden sollte) oder `clippy::performance` (Code, der so geschrieben werden kann, dass er schneller ausgeführt wird). Dadurch werden bereits viele Code-Probleme verhindert.

Wir haben zusätzlich noch das `clippy::pedantic`-Regelset aktiviert [49]. Diese Regeln prüfen den Code noch genauer auf Probleme, z.B. Values, die als Referenzen übergeben werden könnten, mögliche Casting-Probleme oder `if-not-else` Statements. Wir haben diese Lints im Sprint 2 auf den bestehenden Code angewendet und die Regel im Sprint 3 permanent aktiviert.

Generell folgen wir den Empfehlungen von Clippy. Zwei Lints haben wir jedoch projektweit deaktiviert, da sie in unserem Kontext keinen Mehwert bieten: `match_bool` und `implicit_hasher`.

Der Lint `match_bool` empfiehlt, `match`-Cases über `true/false` durch eine `if/else`-Struktur zu ersetzen [50]. Wir empfinden die `match`-Variante jedoch als besser lesbar, deshalb behalten wir diese Schreibweise bei.

Der Lint `implicit_hasher` warnt, wenn bei `HashMap` oder `HashSet` keine explizite Hash-Funktion angegeben wird [51]. Da der Standard-Hashing-Algorithmus für unsere Anforderungen ausreicht und die Funktionssignatur bei einer expliziten Angabe deutlich komplizierter wird (Listing 18), verzichten wir auf diese.

```rust
// No explicit hash function
impl<K: Hash + Eq, V> Serialize for HashMap<K, V> { }
pub fn foo(map: &mut HashMap<i32, i32>) { }

// Explicit hash function given
impl<K: Hash + Eq, V, S: BuildHasher> Serialize for HashMap<K, V, S> { }
pub fn foo<S: BuildHasher>(map: &mut HashMap<i32, i32, S>) { }
```

Listing 18: Beispiel Funktionssignatur mit und ohne expliziter Angabe des Hashing-Algorithmus

Ebenfalls gibt es Code-Stellen, bei denen wir Regelverstösse erwarten: Beispielsweise überprüfen wir in einem Unit-Test, ob eine Zeitdauer von 5.75 Stunden zurückgegeben wird, was gegen die Regel `clippy::float_cmp` (Direkter Vergleich von Kommazahlen) verstösst. Da wir in diesem Test-Case exakt auf diesen Wert prüfen, haben wir den Test mit der `#[expect(clippy::float_cmp)]`-Annotation versehen, um die Warnung auszuschalten. Im Gegensatz zu der `#[allow(clippy::float_cmp)]`-Annotation warnt sie auch, wenn der Regelverstoss nicht (mehr) stattfindet, z.B. nach einem Refactoring.

5.1.3 Dokumentation

Wir verfolgen in diesem Projekt zwei Arten von Dokumentation: Der Bericht der Studienarbeit in Typst und die Code-Dokumentation in den Codedateien.

5.1.3.1 Studienarbeit-Bericht

Der Bericht folgt in der Strukturierung den Vorgaben der Orientierungshilfe und dem Leitfaden für SA/BA [52], [53], während die feinere Struktur aus der SE Project LaTeX Vorlage [54] stammt. Das Dokument wird in Typst verfasst und basiert auf der Vorlage von Dario Glasl [55]. Der Inhalt folgt den in Abschnitt 5.1.1.3 definierten Kriterien.

Um eine konsistente Formatierung sicherzustellen, setzen wir den in Tinymist integrierten `typstyle`-Linter ein. Mithilfe unserer VS Code Workspace-Konfiguration wird bei jedem Speichern einer Typst-Datei der Code automatisch formatiert und der Text nach ca. 120 Zeichen automatisch umgebrochen. So entstehen keine überlangen Zeilen, welche auf GitLab nicht immer automatisch umgebrochen werden.

5.1.3.2 Code-Dokumentation

In Rust können Elemente mithilfe von Documentation Comments annotiert werden. Mithilfe des Rust-Paketmanagers Cargo können diese in HTML exportiert werden und dient so als Handbuch für den Code. Ebenfalls können Beispiel-Codefragmente in Dokumentation Comments eingefügt werden. Sie werden beim Ausführen der regulären Unit-Tests ebenfalls ausgeführt, um inkorrekte Codebeispiele innerhalb der Dokumentation zu verhindern. Eine genauere Erläuterung dazu befindet sich in Abschnitt 5.2.2.

5.2 Testing

Um die Funktionalität des Codes bestmöglichst garantieren zu können, arbeiten wir mit drei Arten von Code-Tests: Unit-Tests, Documentation-Tests und Integration-Tests. Diese Arten werden von Rust nativ unterstützt und können mit `cargo test` ausgeführt werden. So können nahtlos Tests geschrieben werden, ohne dass externe Testing-Frameworks verwendet werden müssen.

Zusätzlich zu den Code-Tests führen wir auch Usability-Tests durch, um die Benutzerfreundlichkeit unseres Programms sicherzustellen.

5.2.1 Unit-Tests

In Rust ist es üblich, Unit-Tests in der gleichen Datei zu schreiben wie der zu testende Code. Dazu wird am Ende der Datei ein neues Modul erstellt, welches meist «`tests`» genannt wird. Das Modul wird dann mit der `#[cfg(test)]`-Annotation versehen, was bedeutet, dass dieses Modul nur im «test»-Profil kompiliert wird. Die Unit-Tests befinden sich also nicht im Debug- oder Release-Binary.

Da sich die Tests in einem eigenen Modul befinden, können wir nicht direkt auf die zu testenden Funktionen im Parent-Modul zugreifen. Deswegen nutzen wir `use super::*;` um alle überliegenden Symbole in den Scope zu bringen. Weil «tests» ein Sub-Modul des zu testenden Moduls ist, kann es so auch auf private Funktionen zugreifen – es ist also auch problemlos möglich, Unit-Tests für private Funktionen zu schreiben.

Die einzelnen Test-Funktionen werden mit der `#[test]`-Annotation markiert. Die Ergebnisse eines Tests können mit den Makros der `assert!()`-Familie überprüft werden: `assert!()` prüft eine Boolesche Bedingung, während `assert_eq!()` und `assert_ne!()` zwei Elemente auf (Un-)gleichheit überprüfen. Ersteres wird oft mit dem `matches!()`-Makro kombiniert, damit das gleiche Pattern Matching wie in einem `match`-Statement verwendet werden kann. Wird ein Assert nicht erfüllt, wird eine Panic ausgelöst und der Test schlägt fehl.

```rust
struct HasSummaryValidator;
impl Validator for HasSummaryValidator {
  fn validate_single(&mut self, time_log: &TimeLog) -> Vec<ValidationProblem> {
    match time_log.summary.is_none() {
        true => vec![ValidationProblem::MissingSummary],
        false => Vec::new(),
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    const NUMBER_OF_LOGS: usize = 7;
    fn get_time_logs() -> [TimeLog; NUMBER_OF_LOGS] { /* ... */ }

    #[test]
    fn test_has_summary_validator() {
        const EXPECTED_PROBLEM: ValidationProblem = ValidationProblem::MissingSummary;
        let time_logs = get_time_logs();

        let mut validator = TimeLogValidator::new()
          .with_validator(HasSummaryValidator);

        let results = validator.validate(&time_logs);
        assert_eq!(results.len(), NUMBER_OF_LOGS);
        for (i, result) in results.iter().enumerate() {
            match i {
                2 | 5 => assert!(result.has_problems(&EXPECTED_PROBLEM)),
                _ => assert!(result.is_valid()),
            }
        }
    }
}
```

Listing 19: Unit-Test aus dem `validation`-Modul

Wir haben bei unserem Projekt 70 Unit-Tests geschrieben und damit eine Testabdeckung von 84.92% erreicht. Zunächst haben wir uns auf Unit-Tests für die «Happy Paths» konzentriert, um sicherzustellen, dass die Funktionen den erwarteten Output ausgeben.

Mithilfe unseres Code Coverage-Tools konnten wir anschliessend sehen, welche Szenarien wir noch nicht abgedeckt haben (z.B. wenn eine Validierung fehlschlägt und wir einen `Error` zurückgeben) und auch für diese Unit-Tests schreiben.

Zur Nachvollziehbarkeit arbeiten wir bei vielen Unit-Tests mit Kompilierzeit-Konstanten (`const`), um «Magic Numbers» in den Asserts zu vermeiden.

Wie in Tabelle 3 beschrieben, überprüfen wir die Effektivität unserer Tests mit einem Code Coverage Report. Ursprünglich haben wir `cargo-llvm-cov` verwendet [56], da RustRover es intern ebenfalls für die Test Coverage innerhalb der IDE verwendet. Das Tool hatte jedoch zwei Probleme: Es konnte nur Code aus der Coverage ausschliessen, wenn der Nightly-Rust-Compiler verwendet wurde und der Code aus Unit-Tests wurde ebenfalls zu der Test-Abdeckung dazugezählt. Aufgrund dieser Probleme haben wir zu `cargo-tarpaulin` [57] gewechselt, welches die genannten Mängel behebt.

Einen Nachteil hatte der Wechsel jedoch: Ursprünglich haben wir Region Coverage als Metrik verwendet, aber da `cargo-tarpaulin` nur Line Coverage unterstützt, mussten wir auf diese etwas ungenauere Metrik umsteigen.

5.2.2 Documentation Tests

Rust ermöglicht innerhalb der Documentation-Comments das Schreiben von Rust-Code, welcher beim Ausführen von `cargo test` kompiliert und ausgeführt wird. Diese sogenannten Doctests (Documentation Tests) ermöglichen das einfache Einfügen von Beispielen in die Code-Dokumentation. Durch die Kompilierung werden diese immer auf ihre Funktionalität überprüft. Im Gegensatz zu Unit-Tests können allerdings nur öffentliche Funktionen auf diese Art getestet werden, da die Doctests als eigene Crate kompiliert werden und deshalb nur auf die öffentliche API zugreifen können. Bei privaten Funktionen muss die Kompilierung mithilfe des `norun`-Attributes abgeschaltet werden. Da wir es als wenig sinnvoll erachteten, nicht-kompilierende Beispiele für private Funktionen zu schreiben, haben wir dies in unserem Projekt nicht verwendet.

Für die wichtigsten Public-API-Funktionen haben wir für Consumer der Library Doctests geschrieben, um die Verwendung dieser zu demonstrieren. Insgesamt haben wir 6 Doctests für die Code-Dokumentation erstellt.

5.2.3 Integration-Tests

In der Library arbeiten wir mit Unit-Tests, um sicherzustellen, dass die Business-Logik wie gewünscht funktioniert. Im CLI rufen wir die Funktionen der Library auf und führen nur minimale Verarbeitungen durch, um die Daten für die Ausgabe aufzubereiten. Aus diesem Grund haben wir im CLI auf Unit-Tests verzichtet und den Code der CLI-Crate aus der Code Coverage ausgeschlossen. Um trotzdem überprüfen zu können, dass das Programm die gewünschte Ausgabe erstellt, wenden wir Integration-Tests an.

Integration-Tests werden in Rust in einem «`tests`»-Ordner neben dem `Cargo.toml` der Crate erstellt. Jede `.rs`-Datei wird beim Ausführen von `cargo test` als eigenes Executable kompiliert. Wir verwenden die `assert_cmd` Crate [58], um das CLI zu starten und die entsprechenden Argumente mitzugeben. Die Crate sendet anschliessend einen HTTP Request an einen von uns mit der `mockito` Crate [59] erstellten HTTP Server, welcher die API von GitLab simuliert.

Der Output des ausgeführten Befehls wird stichprobenmässig überprüft. Beispielsweise überprüfen wir beim Chart-Command, ob die korrekte Anzahl an Dateien erstellt wurde und ob die erste Zeile mit «`<!DOCTYPE html>`» beziehungsweise «`<svg`» beginnt.

```rust
#[test]
fn test_charts_command_writes_charts_to_disk() {
    let mut server = Server::new();
    let mock = create_successful_response_mock(&mut server)
    let url = format!("{}/test-user/test-repo", server.url());

    let dir = tempfile::tempdir().unwrap();
    let output_path = dir.path().join("test-charts");

    cargo_bin_cmd!() // Executes the CLI
        .args([&url, "charts", "--output", output_path.to_str().unwrap(),
          "--sprints", "4"])
        .assert().success(); // Checks for a successful exit code

    mock.assert(); // Check if the simulated server has sent the specified response
    assert!(output_path.exists());
    assert_charts(&output_path);
}

fn create_successful_response_mock(server: &mut Server) -> Mock {
  server
      .mock("POST", "/api/graphql")
      .with_header("Content-Type", "application/json")
      .with_body_from_file("tests/fixtures/serverResponseSuccess.json")
      .create()
}

fn assert_charts(output_path: &Path) {
    const NUMBER_OF_CHARTS: usize = 9;
    const NUMBER_OF_FORMATS: usize = 2;
    const TOTAL_CHARTS: usize = NUMBER_OF_CHARTS * NUMBER_OF_FORMATS;

    let files = output_path.read_dir().unwrap();
    assert_eq!(files.count(), TOTAL_CHARTS);
    for file in output_path.read_dir().unwrap() {
        assert!(file.is_ok());
        let file = file.unwrap();
        let file_path = file.path();
        let file_ext = file_path.extension().unwrap().to_str().unwrap();

        let expected_first_line = match file_ext {
            "html" => "<!DOCTYPE html>",
            "svg" => "<svg",
            _ => panic!("Unexpected file extension: {file_ext}"),
        };

        let file_contents = std::fs::read_to_string(file.path()).unwrap();
        let first_line = file_contents.lines().next().unwrap();
        assert!(first_line.contains(expected_first_line));
    }
}
```

Listing 20: Integration-Test des `charts`-Befehl des CLIs

Insgesamt haben wir 7 Integration-Tests geschrieben. Diese haben wir explizit aus der Code Coverage ausgeschlossen, damit wir weiterhin sehen können, welcher Anteil des Codes durch Unit-Tests abgedeckt wird.

5.2.4 Manuelles Testing

Um die Kompatibilität sicherzustellen, haben wir GitLab Time-Report mit verschiedenen Repositories getestet. Hauptsächlich haben wir direkt mit unserem Projekt-Repository gearbeitet. Für Tests mit Merge Requests und negativen Zeiten haben wir zusätzlich ein Test-Repository erstellt und Einträge dieser Art darin erfasst.

Dank dieses Repositories haben wir einen Fehler bei der Deserialisierung von Merge Requests bemerkt und behoben. Ursprünglich hatten wir in unserem Deserializer `TrackableItem` als Enum definiert, welches ein Struct des jeweiligen Typs enthält. Allerdings wird das Feld des jeweils anderen Typs in der GraphQL-API auf `null` gesetzt, womit die `serde_json` Crate nicht umgehen konnte. Bei Issues war das kein Problem, da die API die Felder in alphabetischer Reihenfolge zurückgab und das `"mergeRequest": null` von `serde_json` einfach ignoriert wurde. Anders sah es bei Merge Requests aus: Da `"issue": null` vor `"mergeRequest": { ... }` geparst wurde, stürzte das Programm ab. Wir haben dieses Problem gelöst, indem wir das Enum in ein Struct mit jeweils einem `Option<Issue>` und `Option<MergeRequest>`-Feld umgewandelt haben.

Da die Inspiration für GitLab Time-Report während unserer Arbeit am SE Project entstanden ist und wir bereits viele verschiedene Einträge von fünf Personen in diesem Repository hatten, bot es sich an, dieses für Tests zu verwenden. Unter anderem haben wir dadurch gesehen, dass eine Person einmal anstatt 15 Minuten nur 10 Minuten eingetragen hatte und die dargestellte Stundenzahl in den Diagrammen deswegen viele Nachkommastellen beinhaltete. Wir haben die Werte daraufhin auf maximal 2 Nachkommastellen gerundet.

Wir haben auch einige öffentliche Repositories auf gitlab.com getestet, allerdings haben wir nur wenige gefunden, die das Zeiterfassungs-Feature nutzen. Ein Repository, das Einträge beinhaltet, war das Repository, in dem GitLab selbst entwickelt wird. Mit diesem haben wir vor allem bei der Deserialisierung einige Bugs aufgespürt. Beispielsweise haben wir hier festgestellt, dass die Felder `summary` und `spentAt` nicht immer in der Response gesetzt werden. Wir haben daher ein Verhalten für das Fehlen dieser in unser Programm eingebaut.

Einen weiteren spannenden Edge Case erlebten wir im selben Repository: In einem Issue wurde ein Denial-of-Service-Exploit beschrieben, bei dem mithilfe der GitLab-API Zeiteinträge mit sehr grossen Arbeitszeiten eingetragen werden konnten. Damit wurde unter anderem das Kanban-Board in GitLab funktionsunfähig gemacht [60]. Als Test wurden in diesem Issue Einträge mit 40 Jahren an Arbeitszeit eingetragen, was auch GitLab Time-Report betraf. Dieses Mal lag das Problem jedoch nicht im Deserializer, sondern in der Validierung: Im `ExcessiveHoursValidator` casteten wir die Anzahl Stunden eines Zeiteintrags zu `i16`, was für diese Zahl zu klein war und eine Panic auslöste. Dieses Problem konnten wir durch eine Anpassung der Datentypen lösen.

5.2.5 Usability Tests

Um sicherzustellen, dass unser Tool die im NFR3 (Tabelle 5) definierten Anforderungen an die Bedienbarkeit erfüllt, haben wir in der 12. Projektwoche ein Usability Testing durchgeführt.

5.2.5.1 Testablauf

Zu Beginn des Tests wurden die Teilnehmenden in ein Szenario eingeführt. Sie mussten für den Test in die Rolle einer Projektleitung schlüpfen, die mithilfe unseres Tools die Arbeitszeiten aus einem Projekt für ein bevorstehendes Review-Meeting aufbereiten sollte. Dazu haben wir kurz die Funktionsweise unseres Programms erläutert.

Ausgangsszenario
Stell dir vor, du bist die leitende Person in einem Software-Team. Für das nächste Review-Meeting willst du dir einen Überblick über die in einem bestimmten Projekt geleisteten Arbeitszeiten deines Teams verschaffen. Dazu verwendest du ein Kommandozeilen-Tool, das Zeiteinträge aus eurem GitLab Projekt ausliest.

Aufgabe 1 - Tool kennenlernen und Funktionen verstehen (US 2)
Du hast das Programm auf deinen Computer geladen und möchtest nun herausfinden, welche Funktionen es bietet. Wie gehst du vor?

Aufgabe 2 - Programm mit dem Projekt-URL starten (US 1)
Von deiner Teamkollegin hast du die URL des GitLab-Repositories des Projektes erhalten, um das es im bevorstehenden Review-Meeting geht. Starte das Programm mit dieser URL und finde heraus, was noch fehlt.

Aufgabe 3 - Zeiteinträge im CLI anzeigen (US 2)
Als erstes möchtest du dir direkt im Terminal einen Überblick über die eingetragenen Zeiten verschaffen. Finde heraus, wie viele Stunden bis jetzt insgesamt für das Projekt aufgewendet wurden.

Aufgabe 4 - CLI-Output anpassen (US 4)
Du möchtest die Ausgabe in der Konsole etwas aufräumen. Finde heraus, was du dafür für Möglichkeiten hast.

Aufgabe 5 - Zeiteinträge in CSV exportieren (US 2)
Für die Buchhaltung benötigst du die Zeiteinträge in einer CSV-Datei. Exportiere die Zeiteinträge in das gewünschte Format. Finde heraus, ob du für den Export Einstellungen vornehmen kannst.

Aufgabe 6 - Diagramme generieren (US 3)
Für deine Präsentation am Meeting möchtest du die Zeiteinträge in Diagrammen visualisiert haben. Finde heraus, was du dafür für Möglichkeiten hast.

Aufgabe 7 - Dashboard anzeigen (US 5)
Eigentlich wäre es auch ganz praktisch, wenn du zur Übersicht über das Projekt ein Dashboard hättest, das dir diverse Statistiken anzeigt. Finde heraus, wie du so ein Dashboard erstellen kannst.

Die Ergebnisse des Usability-Tests befinden sich in Abschnitt 6.1.2.3.

5.3 CI/CD Pipeline

Eine CI/CD Pipeline ist für die Qualitätssicherung unerlässlich. Sie wird bei jedem Push auf das GitLab Repository ausgeführt und benachrichtigt uns, falls die Kompilation des Codes oder Unit-Tests fehlschlagen. Eine Übersicht über alle Pipeline-Jobs befindet sich in Abbildung 4.

5.3.1 Jobs für Rust Code

Für den Umgang mit dem Programmcode verwenden wir in der Pipeline den offiziellen Rust-Container in der Version 1.91 [61]. Um die identischen Optionen nicht bei jedem Rust-Job erneut spezifizieren zu müssen, haben wir zwei Templates erstellt:

5.3.1.1 Job-Templates

- **`.rust-template`:** Das Grund-Template. Setzt den verwendeten Container, Variablen für den Cargo-Paketmanager und das Caching fest (vgl. Abschnitt 5.3.1.4). Vor jedem Job wird geprüft, ob der Job die Variablen `APT_PACKAGES` oder `CARGO_CRATES` setzt: Dann werden die darin spezifizierten Pakete via `apt-get` oder `cargo-binstall` installiert [62]. Wir haben uns für `cargo-binstall` entschieden, da der Standard-Befehl `cargo install` die Crates immer neu kompiliert und die Laufzeit der Pipeline dadurch unnötigerweise verlängert wird.

  Auf dem `main`-Branch sollen immer alle Jobs ausgeführt werden, damit das Deployment auf GitLab Pages reibungslos funktioniert (vgl. Abschnitt 5.3.3). Auf allen anderen Branches werden die Jobs, welche dieses Template verwenden, nur ausgeführt, wenn Rust-Dateien geändert wurden.

- **`.rust-build-release`:** Erweitert das Grund-Template für das Kompilieren von Release Builds, bei welchen der Compiler Optimierungen am Binary vornimmt. Die Jobs müssen die `TARGET`-Variable setzen, in welcher sie die gewünschte Rust-Toolchain angeben. Diese wird dann installiert und das Programm dafür kompiliert. Wenn gewünscht, kann über die `CARGO_SUBCOMMAND` Variable ein zusätzlicher Subcommand von Cargo ausgeführt werden – das verwendet bei uns der `build:windows`-Job. Das Release-Template überschreibt auch die von `.rust-template` gesetzten Regeln, sodass Release-Builds nur auf dem main-Branch ausgeführt werden.

Die meisten Jobs verwenden `.rust-template`, während die beiden Release-Build-Jobs das `.rust-build-release-template` verwenden. Folgende Jobs haben wir definiert:

5.3.1.2 Build-Jobs

- **`build:debug`:** Kompiliert den gesamten Rust-Code im Projekt in der `test` Konfiguration. Der Vorteil der `test`-Konfiguration gegenüber der standardmässig verwendeten `dev`-Konfiguration ist, dass Unit-Tests ebenfalls kompiliert werden. Schlägt dieser Job fehl, werden die nachfolgenden Jobs übersprungen.
- **`build:linux`:** Installiert die `x86_64-unknown-linux-gnu` Rust-Toolchain und kompiliert zusammen mit dem `glibc`-Linker einen Release-Build für das OS auf der x86_64 Architektur. Das resultierende Binary ist nur mit Linux-Distributionen kompatibel, welche `glibc` verwenden. Für `musl`-basierende Distros wie Alpine Linux müsste der Code mit einer separaten Toolchain kompiliert werden. Wir haben darauf aber verzichtet, da wir uns auf einen Build pro OS eingeschränkt haben.
- **`build:windows`:** Der Build für Windows ist etwas komplexer. Für Windows gibt es zwei verschiedene Rust-Toolchains: Die MSVC und GNU Toolchain. Lokal verwenden wir die MSVC Toolchain, da sie die von Rust empfohlene Windows Toolchain ist [63]. Sie benötigt aber die Microsoft Visual C++ Build Tools (MSVC), damit sie deren Linker und Libraries verwenden kann. Da die Installation von MSVC im Container recht umständlich ist, haben wir die GNU Toolchain verwendet. Diese hat in der Pipeline lediglich eine Installation von `mingw-w64` benötigt, damit sie einen funktionierenden Linker hat.

Mit dem Wechsel zu der Charming-Library standen wir aber vor einem Problem: Die Crate besitzt eine Dependency auf die V8-JavaScript-Engine. Da das Projekt, welches Rust-Bindings für die Engine anbietet, keine vorkompilierte Version für die Windows-GNU-Toolchain anbietet [64], mussten wir V8 komplett neu kompilieren, was etwa 45 Minuten dauerte. Bei der Suche nach einer Lösung für dieses Problem sind wir auf `cargo-xwin` gestossen [65]: Diese Crate lädt die MSVC Build Tools herunter und installiert sie, damit sie von Clang und LLVM zur Kompilation verwendet werden können. Dieses Setup funktionierte nun ohne Probleme. Wir haben lediglich eigenen Cache für MSVC im Job eingerichtet, um den etwa 12 Minuten andauernden Download zu umgehen. Damit konnten wir die Laufzeit der Pipeline wieder auf etwa 4 Minuten reduzieren.

5.3.1.3 Test-Jobs

- **`test:clippy`:** Führt den Clippy-Linter für alle Targets (`dev`, `test` und `release`) aus. Damit wird auch der Code in Unit-Tests gelintet. Der Job schlägt bei Fehler oder Warnungen von Clippy fehl. Genaueres zu den Linter-Regeln befindet sich in Abschnitt 5.1.2.1.
- **`test`:** Führt alle Unit-Tests im Projekt aus. Wenn ein Test fehlschlägt, schlägt auch der Job fehl. Zusätzlich werden JUnit-Reports generiert, welche dann in GitLab die Ergebnisse der Tests anzeigen. Dafür werden die in `cargo test` instabilen Features `--format json` und `--report-time` benötigt. Sie werden durch `-Z unstable-options` aktiviert, welche normalerweise nur in Nightly-Builds des Rust Compilers verfügbar sind. Um dies zu umgehen, verwenden wir die `RUSTC_BOOTSTRAP=1` Variable, welche diese Limitation deaktiviert. Dieses Vorgehen wird zwar von den Rust-Entwicklern nicht empfohlen [66], ist aber effizienter, als sich für diesen einen Schritt den gesamten Nightly-Build des Compilers herunterzuladen und wird auch in der Dokumentation von `junitify` beschrieben.

  Der `cargo test`-Befehl gibt dann die Testresultate in JSON aus, welche anschliessend von `junitify` in das JUnit Format umgewandelt werden.
- **`test:coverage`:** Erstellt Code Coverage-Reports, welche zeigen, wie viel Code unsere Unit-Tests abdecken. Der Report wird als HTML und als Cobertura-Report generiert. Das HTML wird auf GitLab Pages gehostet, während der Cobertura-Report dazu dient, die Coverage in Merge Requests anzuzeigen.

  Ursprünglich hatten wir in der Pipeline `cargo-llvm-cov` verwendet [56], da RustRover es intern ebenfalls für die Test Coverage in der IDE verwendet. Wir haben jedoch aus den in Abschnitt 5.2.1 genannten Gründen zu `cargo-tarpaulin` gewechselt.
- **`doc:rust-documentation`:** Erstellt mit dem `cargo doc`-Befehl aus den Documentation Comments im Code HTML-Seiten, welche als Handbuch für die Verwendung des Programms dienen.

Wir haben keinen eigenen Job zur Kompilierung eines MacOS-Builds, da der dafür benötigte Linker nur über das Apple SDK bezogen werden kann. Die Einbindung davon in die Pipeline haben wir als zu komplex angesehen und darauf verzichtet, stattdessen gehen wir bei MacOS wie in Tabelle 7 beschrieben vor.

5.3.1.4 Caching

Der Rust-Compiler benötigt für die Code-Kompilierung im Vergleich zu anderen Sprachen relativ lange. Das hat mehrere Gründe, beispielsweise die vielen Compile-Time-Checks (u.a. Borrow Checking), aber auch weil Dependencies als Quellcode heruntergeladen werden und bei der ersten Verwendung ebenfalls zuerst kompiliert werden müssen.

Da die verwendeten Befehle in den Jobs eine erneute Kompilierung auslösen, haben wir den Output sowie die benötigten Dependencies mit Caching innerhalb der Pipeline zwischengespeichert. So werden Dependencies nicht erneut heruntergeladen und die bereits kompilierten Binaries können weiterverwendet werden.

In GitLab wird das Caching über einen Key gesteuert: Wird in Jobs derselbe Key verwendet, wird auch derselbe Cache benutzt. Indem wir den Hash der `Cargo.lock`-Datei als Key verwenden, in welcher alle (transitiven) Dependencies mit Versionsnummer abgebildet sind, können wir die Kompilate branch-übergreifend cachen. Erst wenn sich eine (transitive) Dependency und damit der Cache-Key ändert, wird der Cache neu erstellt.

Leider kann der Cache nicht ganz verhindern, dass in den einzelnen Jobs neu kompiliert wird. Beispielsweise erzwingt `cargo-tarpaulin` einen Rebuild, um falschen Resultaten vorzubeugen [67]. Für den Download von MSVC im `build:windows` Job haben wir ebenfalls einen eigenen Cache erstellt, dies ist in Abschnitt 5.3.1.2 genauer beschrieben.

5.3.2 Jobs für den Studienarbeit-Bericht

Der Bericht wird ebenfalls in der Pipeline mithilfe des offiziellen Typst-Containers kompiliert.

- **`doc:gitlab-time-report`:** Dieser Job generiert die Diagramme und das Dashboard für die im Projekt eingetragenen Arbeitsstunden. Dafür wird der neueste Build unseres in dieser Arbeit erstellten Produkts «GitLab Time-Report» benötigt. Wenn in der aktuellen Pipeline ein Debug-Build erstellt wurde, wird dieser verwendet. Ansonsten wird der Linux-Release-Build aus der letzten erfolgreichen Pipeline des `main`-Branches heruntergeladen.
- **`doc:typst-documentation`:** Kompiliert die Hauptdatei `main.typ` ohne Kommentare als PDF. Es lädt die Artefakte aus `doc:gitlab-time-report` herunter und überschreibt die Platzhalter-Bilder, welche wir für die lokale Kompilierung verwenden, mit den aktuellen Zeiterfassungs-Diagrammen. Das PDF der Dokumentation wird als Artefakt gespeichert.

5.3.3 Deployment-Jobs

Das Deployment der Dokumentationen auf GitLab Pages wurde in einen eigenen Job ausgelagert, damit dies nur ausgeführt wird, nachdem ein Merge Request auf den `main`-Branch gemerged wird.

- **`pages`**: Deployed den Studienarbeit-Bericht, den Code Coverage Report, die Code-Dokumentation und das Dashboard von GitLab Time-Report auf GitLab Pages. Die Links zu diesen Dokumenten befinden sich im README.

Hier mussten wir einen kleinen Workaround implementieren: Wenn ein Job nicht ausgeführt wurde (vgl. Abschnitt 5.3.4), schlug der `pages`-Job fehl, weil `mv` die Dateien nicht fand. Auch wird der Inhalt von GitLab Pages bei jedem Deployment vorher gelöscht. Um diese Problematiken zu beheben, lassen wir *alle* Jobs nach dem Merge in `main` noch einmal laufen. Also werden gewisse Jobs doch teilweise «unnötigerweise» noch einmal ausgeführt (da sich evtl. bei diesen Dokumenten nichts geändert hat). Doch dies schien uns ein guter Mittelweg zu sein, da auf den Feature Branches so die Pipeline immer noch effizienter läuft als vorher.

Eine etwas elegantere Methode wäre mit dem `needs:project` Keyword möglich gewesen: Damit könnte man die Artefakte der vorherigen Ausführung eines Jobs herunterladen [68]. Leider ist dieses Feature in der GitLab Community-Edition, welche die OST verwendet, nicht verfügbar.

5.3.4 Merge Request-Jobs

Aus Effizienzgründen haben wir die Rust- und Typst-Jobs so konfiguriert, dass sie nur bei Änderungen ausgeführt werden. Das führte zu Problemen, wenn wir eine Datei geändert haben, die nicht in eine dieser Kategorien fiel (z.B. das README oder die Pipeline-Konfiguration). Beim Merge Request wurde dann keine Pipeline ausgeführt, weshalb der Merge nicht durchgeführt wurde. Um dies zu umgehen, haben wir einen Dummy-Job hinzugefügt.

- **`dummy-merge-request-job`:** Wird bei jedem Merge Request-Event ausgeführt, damit immer eine Pipeline generiert wird. Führt einen `echo`-Befehl aus.

5.4 Externe Code-Reviews

Während der Studienarbeit haben wir zwei Code-Reviews von Olivier Lischer vom Institut für Software an der OST beantragt. Einige Punkte davon haben wir direkt umgesetzt, bei anderen entschieden wir uns bewusst dagegen. In diesem Kapitel sind alle umgesetzten und nicht umgesetzten Punkte inklusive Begründung aufgelistet.

Feedback	**Umsetzung**
`#[expect(lint)]` anstatt `#[allow(lint)]` verwenden	Die Allow-Annotation erlaubt Linter-Verstösse im nachfolgenden Block. Die Expect-Annotation funktioniert genauso, gibt aber auch eine Warnung aus, wenn der Linter-Verstoss nicht mehr auftritt. Das kann beim Refactoring nützlich sein, um zu erkennen, ob sich das Verhalten unabsichtlich geändert hat oder die Ausnahme nicht mehr nötig ist.
Unnötige Instanzierung von `Client` entfernen	Wir haben auf `reqwest::Client` einen Extension-Trait implementiert, welcher als Parameter `self` (also den Client selbst) entgegennahm. Zusätzlich instanzierten wir in der Funktion unnötigerweise noch einmal einen neuen Client. Dies haben wir nach dem Review entfernt.
`max_hours` in `ExcessiveHoursValidator` ist signed, unsigned wäre aber sinnvoller	Da ein Zeiteintrag in GitLab auch negativ sein kann, haben wir `max_hours` als `i16` (signed integer) implementiert, damit kein Casting nötig ist. Da eine negative maximale Anzahl Stunden jedoch nicht sinnvoll ist, haben wir den Datentyp auf `u16` umgestellt, um diesen «ungültigen» Zustand unrepräsentierbar zu machen.
Entfernen der `except()`-Funktion in `ExcessiveHoursValidator`	In der Validierung für die Überschreitung der maximalen Stunden in einem Zeiteintrag haben wir die im Zeiteintrag angegebenen Stunden von `i32` nach `i16` konvertiert. Da ein `i16` Werte bis zu 32′767 speichern kann, haben wir in der Erwartung, dass kein grösserer Wert angegeben wird, bei Fehlschlagen der Konvertierung ein `expect()` verwendet. Mit der Änderung von `max_hours` konnten wir auch das `expect`-Statement entfernen und somit mögliche Panics vermeiden.
Die Visibility von `group_by_filter()` auf Public ändern	Wir haben im Modul `filters` eine generische Funktion `group_by_filter()` geschrieben, welche von anderen Funktionen des Moduls, wie `group_by_user()` oder `group_by_milestone()` aufgerufen wird. Ursprünglich hatten wir diese Funktion als private deklariert, da wir nicht damit rechneten, dass sie für externe User von Nutzen sein könnte. Nach dem Code-Review haben wir sie aber auch öffentlich gemacht, damit auch andere Consumer der Library davon profitieren können.
Umbennenung von `to_readable_string()`	Im Modul `chrono_extensions` haben wir eine Funktion geschrieben, die uns ein `TimeDelta` (und dessen Alias `Duration`) als String zur Anzeige im CLI und Dashboard ausgibt. Im Rahmen des Code-Reviews haben wir sie in `to_hm_string()` umbenannt, um die Funktionsweise im Namen klarer zu spezifizieren.

Tabelle 13: Umgesetztes Feedback aus den externen Code-Reviews

Feedback	Begründung für die Nicht-Umsetzung
Felder aus Struct `TrackableItemFields` direkt in `TrackableItem` anlegen	Dies wäre grundsätzlich ergonomischer. Da wir jedoch einen eigenen Deserializer für `TrackableItem` verwenden, können wir keine `#[serde]` und `#[serde_as]`-Attribute verwenden, um beispielsweise die Anzahl Sekunden von `total_time_spent` direkt in den `chrono::Duration`-Typ zu konvertieren. Diese hätten wir ebenfalls in unseren eigenen Deserializer einbauen müssen. Der zusätzliche Aufwand, uns in diese Thematik einzuarbeiten, sie zu implementieren und Tests dafür zu schreiben, war uns im Vergleich zu der zu gewinnenden Ergonomie zu gross.
Verwendung der `chrono`-internen Formatierungsfunktionen	In der `to_hm_string`-Funktion haben wir uns die nötigen Informationen aus `TimeDelta` manuell berechnet, um den String für die Anzeige zu erhalten. Den Vorschlag, dafür die von `chrono` bereitgestellte `format()`-Funktion zu verwenden, konnten wir nicht umsetzen, da `TimeDelta` diese nicht anbietet.
Snapshot-Testing in den Integration Tests verwenden	Im Code-Review wurde uns empfohlen, Snapshot Testing durchzuführen. Dabei wird der Output eines Programmes mit einem Snapshot verglichen, der einen korrekten Output darstellt. Da wir jedoch gemeinsam mit den Betreuern beschlossen haben, den Output nur stichprobenartig zu überprüfen, da die Unit-Tests bereits die inhaltliche Korrektheit garantieren sollen, haben wir uns gegen Snapshot-Testing entschieden.
Eine HTML Templating Engine verwenden	Im Code-Review wurde nachgefragt, wieso wir keine HTML Templating Engine verwendet haben. Die Gründe dafür sind in Abschnitt 4.1.9 dargelegt.
Frage nach der Verwendung des «older style»-Naming von Modulen.	Rust-Module können auf zwei Arten benannt werden: Einmal der «older style», auch «directory module» genannt. Dabei wird ein Ordner und darin eine `mod.rs`-Datei erstellt. Im «newer style», auch «file module» genannt, wird direkt eine neue Rust-Datei erstellt. Wir haben zuerst mit File Modules begonnen und sind bei einzelnen Modulen auf Directory Modules umgestiegen, um Code in andere Dateien ausgliedern zu können und das Modul somit sowohl logisch (Erstellung eines Submoduls) als auch physisch (Verschiedene Dateien) zu unterteilen. Bei Modulen, in denen sich alles in der gleichen Datei befindet, haben wir den «newer style» belassen.
Anderes Code Coverage-Tool verwenden, damit nicht so viel Code manuell ausgeschlossen werden muss	Wir haben bestimmte Abschnitte unseres Codes von der Code Coverage ausgeschlossen, wenn Unit-Tests nicht sinnvoll waren. Ein Beispiel sind die Extensions-Module, die lediglich Funktionen von externen Libraries aufrufen und deshalb keine eigenständige, testbare Logik beinhalten. Es gibt zwar, wie im Code-Review erwähnt, einige `#![cfg(not(tarpaulin_include))]`-Annotationen. Doch da unsere Bedürfnisse durch Tarpaulin jedoch erfüllt sind, sahen wir keinen Grund, das Code Coverage Tool so spät im Projekt nochmals zu ändern.

Tabelle 14: Nicht umgesetztes Feedback aus den externen Code-Reviews

6 Ergebnisse

6.1 Resultat der Arbeit

Im Rahmen dieser Studienarbeit wurde mit «GitLab Time-Report» ein Programm umgesetzt, dass Zeiteinträge aus GitLab automatisiert ausliest, validiert und als Tabellen, Diagramme sowie als HTML-Dashboard aufbereitet.

Damit wird die Lücke geschlossen, die GitLab selbst offen lässt: Zeiterfassung ist dort zwar möglich, aber Auswertung und Filterung der Zeiten ist nicht möglich.

6.1.1 Erfüllung der Functional Requirements

Die funktionalen Anforderungen wurden wie in Abschnitt 2.1.1 definiert als User Stories (US) mit Akzeptanzkriterien formuliert. Die Kernfeatures des MVPs und eine Erweiterung wurden umgesetzt, nur zwei Akzeptanzkriterien blieben offen.

6.1.1.1 User Story 1 – Verwendung via CLI

Das Programm ist über ein CLI nutzbar und die Befehle sind, wie im Usability-Test überprüft, logisch benannt. Die Flag `--help` bietet sowohl für das ganze Programm als auch für jeden einzelnen Befehl eine Übersicht über die Funktionalitäten. Auch eine vollständig dokumentiert Anleitung ist vorhanden: Im Readme befindet sich eine detaillierte Anleitung zur Nutzung des Tools.

Status	Akzeptanzkriterium
✓	Eine vollständig dokumentierte Anleitung ist vorhanden.
✓	Die einzugebenden Befehle sind logisch benannt und möglichst kurz gehalten.
✓	`--help` zeigt eine Übersicht über die angebotenen Befehle und Argumente an.
✗	Autocompletion mittels Tab wird angeboten.

Tabelle 15: Status der Akzeptanzkriterien der User Story 1

Damit sind alle Akzeptanzkriterien erreicht, ausser das Kriterium «Autocompletion mittels Tab wird angeboten». Wir konnten keine Library finden, die diese Erweiterung für die `clap`-Crate anbietet. Eine manuelle Umsetzung hat sich als zu zeitintensiv herausgestellt, da die Autocomplete-Unterstützung von der verwendeten Shell abhängt. Um eine breite Unterstützung zu gewährleisten, hätten wir für jede Shell einzeln eine Unterstützung anbieten müssen. Aus diesem Grund haben wir die Umsetzung dieses Akzeptanzkriteriums fallen gelassen.

Der Output des `--help`-Arguments ohne Subcommands ist in Abbildung 3 ersichtlich.

6.1.1.2 User Story 2 – Zeiten anzeigen

Von der User Story 2 haben wir alle Akzeptanzkriterien erfüllt, ausser dass wir Minus-Stunden und zu grosse Unterschiede zwischen den Personen nicht validieren. Aus Zeitgründen haben wir uns dazu entschieden, nur jeden Zeiteintrag separat zu validieren. Diese beiden Features würden jedoch eine Validierung von einer Gruppe von Zeiteinträgen (hier pro Person) erfordern. Es wäre möglich, solche Validierungen in unser Programm einzubauen, in dem ein neuer Trait im `validation`-Modul erstellt wird, welche dann diese Funktionalität analog zu `validate_single()` anbietet. Dafür hat die Zeit jedoch nicht mehr gereicht.

Die Validierungen für einzelne Zeiteinträge umfassen:
- Die für einen Zeiteintrag eingetragene Zeit ist unterhalb einem definierte Maximum
- Der Zeiteintrag hat einen Beschreibungstext (Summary)
- Der Zeiteintrag ist nach dem Startdatum des Projektes und liegt nicht in der Zukunft
- Prüfung auf duplizierte Einträge (gleicher User, gleiches Datum, gleiches Issue/Merge Request, gleiche Zeit und gleicher Beschreibungstext)

Status	Akzeptanzkriterium
✓	Die Zeiten werden im CLI tabellarisch angezeigt.
✓	Die Zeiten werden im Dashboard (vgl. Abschnitt 2.1.1.5) angezeigt.
✓	Die Zeiten werden in eine Datei in einem gängigen Format exportiert.
✓	Die Übersicht über die Zeiten ist logisch strukturiert und verständlich.
✓	Eine heuristische Datenüberprüfung stellt sicher, dass alle erfassten Zeiten und Schätzungen, egal ob über Issues oder Merge Request eingetragen, korrekt berücksichtigt werden.
✓	Gelöschte Zeiten werden korrekt verarbeitet.
✗	Bei nicht plausiblen Daten (Minus-Stunden, zu grosse Unterschiede zwischen den Personen) wird der User informiert.

Tabelle 16: Status der Akzeptanzkriterien der User Story 2

Standardmässig wird nur ein Hinweis auf die Validierung angezeigt, wie in Abbildung 16 ersichtlich.

```
Fetching time logs from 'https://gitlab.ost.ch/gitlab-time-report/test'...

5 problems found in the time logs of the project. To see the problems, run with --validation-details
```

Abbildung 16: Hinweis auf Validierung im CLI-Output

In Abbildung 17 wird gezeigt, wie die Validierungsresultate präsentiert werden, wenn im CLI das `--validation-details`-Argument gesetzt wird. Nach einem Titel werden alle Einträge aufgelistet, die in der Validierung aufgefallen sind. Dabei wird der Typ, die ID und der Titel des Trackable Item, sowie das Datum, die angegebene Zeit und die Person, welche den Eintrag erstellt hat, ausgegeben. Darunter wird angemerkt, welche Validierung auf diesem Eintrag ausgeöst wurde. Nachdem alle Problemdetails angezeigt wurden, wird zu unterst die Anzahl Probleme ausgegeben.

Die Validierung dient lediglich der Information der User. Es werden keine Daten herausgefiltert oder in GitLab korrigiert. Standardmässig werden alle Validierungen durchgeführt, ohne dass die User etwas daran ändern können. Die einzige Einstellungsmöglichkeit ist die Maximalzeit pro Eintrag, die auf zehn Stunden voreingestellt ist.

Hier gibt es also noch Erweiterungspotenzial: Man könnte die User selber einzelne Validierungen ein- oder ausschalten lassen.

```
Fetching time logs from 'https://gitlab.ost.ch/gitlab-time-report/test'...
============================================
      Time Logs with validation problems
============================================
Issue #1: Parent Issue 1
2025-10-09, (00h 15m), Nina Grässli:
No summary was entered

Issue #2: Child Task 1
2025-10-25, (01h 00m), Jannis Tschan:
No summary was entered

Issue #1: Parent Issue 1
2025-12-15, (00h 15m), Nina Grässli: test
Duplicate entry

Issue #4: Doc Issue 1
2025-12-16, (12h 00m), Nina Grässli: test
Time spent exceeds maximum of 10 hours

Issue #2: Child Task 1
2025-12-18, (01h 00m), Nina Grässli: test
Date is in the future

============================================
Total Problems found: 5
============================================
```

Abbildung 17: Validierung im CLI-Output

6.1.1.3 User Story 3 – Diagramme und Statistiken generieren

In der User Story 3 konnten wir alle Akzeptanzkriterien erfüllen. Die Daten werden in drei verschiedenen Tabellen sowohl im CLI als auch im Dashboard angezeigt.

Status	Akzeptanzkriterium
✓	Die Daten werden in verschiedenen tabellarischen Ansichten strukturiert dargestellt.
✓	Ein Diagramm für Gesamtzeit pro Contributor kann erstellt werden
✓	Ein Burndown-Diagramm für die Gesamtzeit und pro Person kann erstellt werden
✓	Ein Diagramm für den Vergleich von Schätzungen und effektiv geleisteten Stunden kann erstellt werden
✓	Ein Diagramm für Stunden pro Label und Person kann erstellt werden
✓	Ein Diagramm pro Zuweisung (Issue oder MR) und Person kann erstellt werden
✓	Die Diagramme sind als Bild exportierbar
✓	Die Darstellung der Diagramme ist übersichtlich, einheitlich und Labels sind lesbar.

Tabelle 17: Status der Akzeptanzkriterien der User Story 3

In einer Tabelle werden die Zeiten pro Person von heute, gestern, den letzten sieben und den letzten 30 Tagen sowie die insgesamt von dieser Person eingetragenen Zeiten angezeigt. In einer Zeile werden ausserdem die aufsummierten Daten für alle Metriken angezeigt.

```
Time Spent per User:
+--------------+---------+-----------+-----------------+-------------+------------------+
| User         | Today   | Yesterday | Last seven days | Last 30 days | Total time spent |
+--------------+---------+-----------+-----------------+-------------+------------------+
| Jannis Tschan | 00h 30m |   00h 30m |         03h 15m |      84h 45m |         236h 15m |
+--------------+---------+-----------+-----------------+-------------+------------------+
| Nina Grässli  | 00h 45m |   09h 15m |         31h 45m |      99h 45m |         222h 30m |
+--------------+---------+-----------+-----------------+-------------+------------------+
| Total         | 01h 15m |   09h 45m |         35h 00m |     184h 30m |         458h 45m |
+--------------+---------+-----------+-----------------+-------------+------------------+
```

Abbildung 18: Tabelle mit Zeiten pro User im CLI-Output

Zwei weitere Tabellen zeigen die insgesamt eingetragene Zeit pro Label und die Zeit pro Meilenstein an.

Time Spent per Label:

LABEL	TIME SPENT
CI/CD	44h 00m
Development	227h 00m
Documentation	130h 30m
Meetings	23h 15m
Project Setup	03h 00m
Usability	31h 00m

Time Spent per Milestone:

MILESTONE	TIME SPENT
No milestone	81h 45m
M1 - End of Inception	26h 00m
M2 - End of Elaboration	97h 15m
M3 - End of Construction	209h 45m
M4 - Final Submission	44h 00m

Abbildung 19: Tabelle mit Zeiten pro Label und Milestone im Dashboard

Alle geplanten Diagramme werden sowohl als HTML als auch als SVG-Dateien generiert. Sie werden oberhalb der Tabellen im Dashboard angezeigt. Auf der rechten Seite jedes Diagrammes befindet sich eine Toolbox. Mithilfe dieser ist es möglich, das Diagramm als PNG herunterzuladen und die zugrundeliegenden Daten anzuzeigen. Je nach Typ sind noch weitere Funktionen vorhanden: Bei Balkendiagrammen ist es beispielsweise möglich, ein gestapeltes Balkendiagramm anzuzeigen. Auch lassen sich über die Legende einzelne Elemente der Diagramme ein- und ausblenden, womit eine genauere Filterung durch den User möglich ist.

Abbildung 20: Alle Diagramme, angezeigt im Dashboard.

Wir haben versucht, die Labels im Diagramm und in der Legende so gut wie möglich lesbar zu gestalten. Bei zu vielen oder zu langen Einträgen in der Legende kommt es jedoch immer noch zu Überlappungen, die wir nicht vermeiden konnten. Solange dieses Limit aber nicht erreicht ist, bleibt der Graph gut lesbar.

6.1.1.4 User Story 4 – Zeiten und Statistiken filtern

Auch von der User Story 4 konnten wir alle Akzeptanzkriterien erfüllen. Die Filterung ist möglich und die Filter können kombiniert werden. Jedoch geschieht das automatisch, der Nutzer kann die Filter nicht selber definieren. Das wäre aber eine mögliche Erweiterung des Projektes.

Status	Akzeptanzkriterium
✓	Filterung ist nach Person, Zeit, Datumsbereich, Meilenstein, Label oder Zuweisung (Issue oder MR) möglich.
✓	Die Filter können kombiniert werden (z.B. nach Zeit und Person).

Tabelle 18: Status der Akzeptanzkriterien der User Story 4

6.1.1.5 User Story 5 – Übersicht generieren

Sowohl die Tabellen als auch die Diagramme werden in einem Dashboard angezeigt. Das Dashboard besteht aus einem einzigen HTML-File, was die Weiterverwendung beispielsweise mit GitLab Pages einfach macht.

Das von GitLab Time-Report generierte Dashboard ist als HTML-Dateianhang in diesem PDF enthalten.

Status	Akzeptanzkriterium
✓	Das Dashboard kann als HTML exportiert werden (z.B. für Weiterverwendung mit GitLab Pages)
✗	~~Das Dashboard kann optional über eine GUI-App angezeigt werden (vgl. Abschnitt 2.1.1.6)~~
✓	Das Dashboard enthält sowohl eine tabellarische Übersicht über die Daten als auch die erstellten Diagramme

Tabelle 19: Status der Akzeptanzkriterien der User Story 5

Somit sind auch hier die Akzeptanzkriterien erfüllt. Da wir keine GUI-App erstellt haben, fällt das zweite Kriterium als «nicht umgesetzt» weg.

Die User können für das Design der Diagramme ein eigenes Theme verwenden, wie in Abschnitt 4.1.8 beschrieben. Je nach Theme sieht das Dashboard entsprechend unterschiedlich aus, wie in Abbildung 21 ersichtlich.

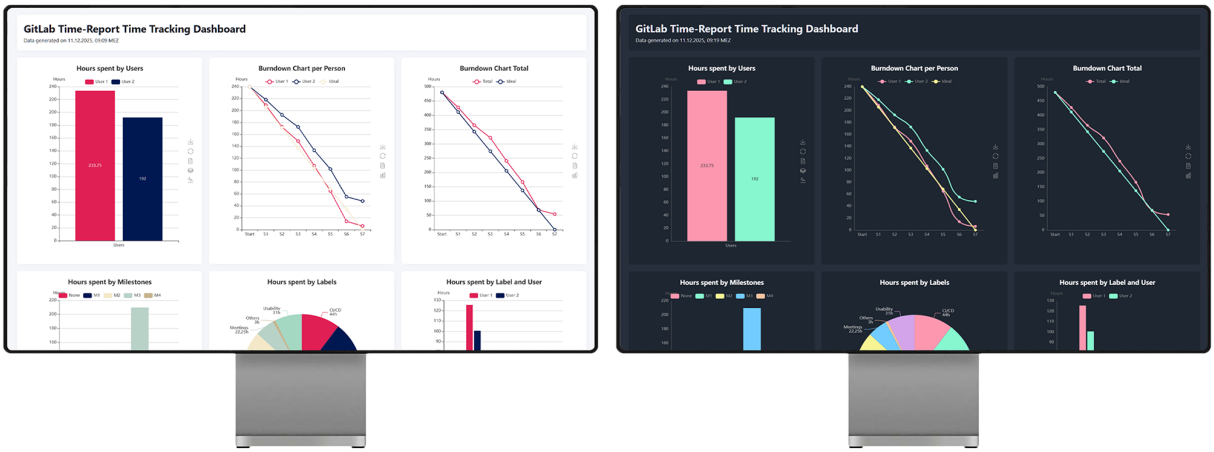


Abbildung 21: Dashboard mit unterschiedlichen Themes

Das Aussehen des Dashboards ohne eigenes Theme wird in Abbildung 1 und Abbildung 2 aufgezeigt. Ebenfalls ist die HTML-Datei als Dateianhang an dieses PDF angefügt.

6.1.1.6 User Story 6 – Integration in CI/CD Workflows

Zusätzlich zu den User Stories, die unseren MVP abbilden, haben wir auch noch die User Story 6, die in den Erweiterungen definiert war, erfüllt. Wir haben unsere eigenen Reports mit unserem eigenen Programm direkt in der Pipeline generieren lassen, das Dashboard auf GitLab Pages veröffentlicht und die Bilder direkt hier eingebunden.

Damit sich die Dokumentation (z.B. LaTeX oder Typst) lokal kompilieren lässt, haben wir Platzhalter-Bilder für verschiedene Diagramm-Typen definiert (Abbildung 22). Sie können anstelle der von GitLab Time-Report generierten Diagramme in der Dokumentation verwendet und dann in der Pipeline durch aktuelle Diagramme von GitLab Time Report ersetzt werden. Dazu haben wir im Readme zwei minimale Pipeline-Konfigurationen definiert, welche die User in ihrem Projekt verwenden können. Die Platzhalter-Bilder stehen den Usern ebenfalls zur Verfügung.

Abbildung 22: Platzhalter für die Diagramme

```yaml
gitlab-time-report:
  image: debian:latest
  before_script:
    # Install wget and current root CA certificates to download gitlab-time-report
    - apt-get -qq update && apt-get -qq install wget ca-certificates
  script:
    - wget --no-verbose -O gitlab-time-report-cli https://gitlab.ost.ch/gitlab-time-report/gitlab-time-report/jobs/artifacts/main/raw/target/x86_64-unknown-linux-gnu/release/gitlab-time-report-cli?job=build:linux
    - chmod +x gitlab-time-report-cli
    # Creates the dashboard. Make sure to create the GITLAB_TOKEN variable in your repository settings
    - ./gitlab-time-report-cli $CI_PROJECT_URL dashboard --sprints 5 --weeks-per-sprint 3 --hours-per-person 360

documentation:
  image:
    name: ghcr.io/typst/typst:latest
    entrypoint: [ "" ]
  needs: [ "gitlab-time-report" ]
  script:
    # Move charts into documentation. Make sure to change the path
    - mv -f charts/*.svg my/image/path
    # Move dashboard into documentation. See Typst snippet below on how to insert it.
    - mv -f dashboard.html my/attachment/path
    - typst compile main.typ
  artifacts:
    paths:
      - main.pdf

pages:
  needs: [ "gitlab-time-report" ]
  pages: true
  script:
    # Create Pages folder if it doesn't exist
    - mkdir -p public
    # Deploy dashboard
    - mv dashboard.html public/dashboard.html
    - echo "Time Tracking Dashboard available on GitLab Pages at $CI_PAGES_URL/dashboard.html"
  rules:
    # Only deploy when pushing to main
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Listing 21: Beispiel-Pipeline aus dem Readme zur Generierung von Diagrammen für Typst

```
#pdf.attach(
  "path/to/dashboard.html",
  relationship: "supplement",
  mime-type: "text/html",
  description: "The dashboard containing the worked hours on the project",
)
```

Listing 22: Beispiel-Code-Snippet aus dem Readme zur Einbindung des Dashboards in Typst

6.1.1.7 Nicht umgesetzte User Stories

Aus Zeitmangel haben wir davon abgesehen, folgende User Stories aus den Erweiterungen einzuplanen:

- **US 7 – GUI App für Konfiguration**
- **US 8 – Kategorien erstellen**
- **US 9 – Lokale Stoppuhr-Funktion**
- **US 10 – Unterstützung von Sprints**
- **US 11 – Erstellung von Gantt-Charts**

Anzumerken ist, dass zwar in Burndown-Diagrammen mit Sprints gearbeitet werden kann, in allen anderen Diagrammen und Tabellen allerdings nicht. Deswegen gilt US 10 als nicht umgesetzt.

6.1.2 Erfüllung der Non-Functional Requirements

Die Überprüfung der NFRs wurde im Projekt laufend in Form eines Changelogs dokumentiert.

6.1.2.1 NFR 1: Wartbarkeit – Analysierbarkeit

«Die Code-Qualität entspricht den im Code-Linter spezifizierten Anforderungen.»

- **30.09.2025:** Zusätzlich zum automatischen Linter in der IDE haben wir Clippy in der Pipeline hinzugefügt.
- **06.10.2025:** Wir wollen stabiles Error Handling in unser Projekt einbauen. Dafür aktivieren wir ab M3 im Linter Warnungen, wenn unwrap() verwendet wird. Diese Funktion sollte nur zum Erstellen von Prototypen verwendet werden, da ein Programm bei einem Fehler sofort beendet wird.
- **13.11.2025:** Nachdem wir bereits in der Vergangenheit die Vorschläge aus dem `clippy::pedantic`-Regelsatz angewendet haben, wenden wir nun das Regelset permanent auf unseren Code an. Das heisst, die Pipeline schlägt neu auch fehl, wenn gegen diese genaueren Regeln verstossen wird.
- **24.11.2025:** Lint für unwrap() wurde aktiviert.

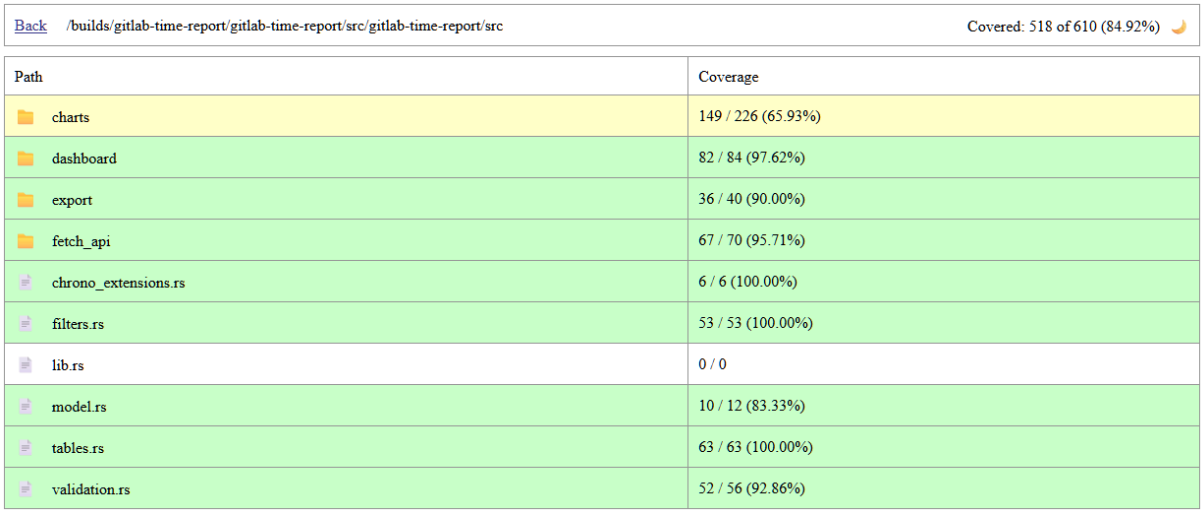
Da wir den Linter mit einem strengen Regelset konfiguriert haben und dieser die Pipeline bei Warnungen fehlschlagen lässt, haben wir sichergestellt, dass die Code-Qualität den spezifizierten Anforderungen gerecht wird.

6.1.2.2 NFR 2: Wartbarkeit – Testbarkeit

«Die Testabdeckung beträgt mindestens 80%.»

- **30.09.2025:** In unserer Pipeline generiert `llvm-cov` einfache Code Coverage Metriken. Da wir für die Code-Qualität zusätzlich noch den Clippy-Linter verwenden, haben wir auf tiefergreifende Analysetools wie SonarQube verzichtet.
- **23.10.2025:** Die Region-Coverage ist aktuell auf 90.11%.
- **09.11.2025:** Wir haben das Code-Coverage Tool zu `tarpaulin` gewechselt. Genaueres dazu befindet sich in Abschnitt 5.2.1. Mit dem neuen Tool messen wir statt der Region-Coverage die Line-Coverage.
- **05.12.2025:** Die Line-Coverage ist aktuell auf 85.97%.

Wir haben für unser Programm sowohl Unit- als auch Integration-Tests geschrieben, die in der Pipeline automatisch ausgeführt werden. Schlägt ein Test fehl, schlägt auch die Pipeline fehl. Mehr Details zu der Testabdeckung befinden sich in Abschnitt 5.2.



Back /builds/gitlab-time-report/gitlab-time-report/src/gitlab-time-report/src — Covered: 518 of 610 (84.92%)

Path	Coverage
charts	149 / 226 (65.93%)
dashboard	82 / 84 (97.62%)
export	36 / 40 (90.00%)
fetch_api	67 / 70 (95.71%)
chrono_extensions.rs	6 / 6 (100.00%)
filters.rs	53 / 53 (100.00%)
lib.rs	0 / 0
model.rs	10 / 12 (83.33%)
tables.rs	63 / 63 (100.00%)
validation.rs	52 / 56 (92.86%)

Abbildung 23: Testabdeckung von GitLab Time-Report

6.1.2.3 NFR 3: Interaktionsfähigkeit – Lernbarkeit

«Das Programm ist leicht zu bedienen.»

Ergebnisse des Usability-Tests
Wir haben das Testing am 04.12.2025 mit insgesamt sechs Probanden durchgeführt und von ihnen auf einer Skala von 1-6 im Durchschnitt eine Bewertung von 5.375 erhalten.

Die vom Feedback abgeleiteten Änderungen an unserem Programm lassen sich in zwei Kategorien einteilen. Zum einen die Features, die wir relativ einfach umsetzen konnten und dies auch getan haben, zum anderen die Rückmeldungen, die umständlicher umzusetzen wären und den Scope unseres Projektes sprengen. Diese sind in die Kategorie «Mögliche Erweiterungen» eingegliedert.

Umgesetztes Feedback	– Der Output der Validierung ist nun besser strukturiert und hat einen Titel. Ausserdem wird standardmässig nur die Anzahl der Probleme angezeigt, erst mit Angabe einer Flag wird detailliert aufgelistet, welcher Eintrag welche Probleme aufweist. – Die Flag `weeks-per-sprint` ist nun nicht mehr optional – Der Konsolen-Output ist besser strukturiert und die Texte wurden zum Teil überarbeitet – Sowohl der Token als auch die Chart-Settings können neu auch über Environment-Variablen gesetzt werden
Mögliche Erweiterungen	– Unterstützung für Poly-Repositories (Zeiten für mehrere Repositories zusammenfassen) – Autocomplete der Flags und Commands – Verlinkung der generierten Dateien im CLI – Nicht-Alphabetische Sortierung der Tabellen, damit «Others» und «No Label» immer zu unterst angezeigt werden – CLI-Output farbig gestalten – Labels neben dem expliziten Inkludieren auch explizit ausschliessen können – Die verschiedenen Validierungen einzeln ein- und ausschalten können – Einzelne Diagramme erstellen können

Tabelle 20: Rückmeldungen des Usability-Tests

Durch die erreichte Note und den Umstand, dass wir durch das Testing einige kleinere Optimierungen an unserem Produkt vornehmen konnten, kann das NFR als erfüllt angesehen werden.

Die genauen Protokolle des Usability-Tests befinden sich im Anhang im Abschnitt C.

6.1.2.4 NFR 4: Wartbarkeit - Wiederverwendbarkeit

«Die Funktionen sollen von unterschiedlichen Frontends angesprochen werden können.»

– **23.10.2025:** Unsere Architektur haben wir wie im Abschnitt 3 beschrieben aufgebaut. Die Business-Logik befindet sich komplett in der Library und das CLI-Frontend ruft die entsprechenden Funktionen auf.

Aus Zeitgründen konnten wir nur das CLI-Frontend implementieren und mussten auf ein GUI verzichten. Aufgrund der strikten Trennung von Business-Logik und Frontend wäre jedoch ein zweites Frontend gut in die bestehende Codebasis integrierbar gewesen. Somit kann dieses NFR als erfüllt betrachtet werden.

6.1.2.5 NFR 5: Flexibilität – Adaptierbarkeit

«Das Programm soll auf unterschiedlichen Betriebssystemen und Umgebungen (z.B. Docker Container) laufen»

- **30.09.2025:** Das automatisierte Kompilieren von MacOS-Builds in einer Pipeline ist sehr umständlich, da der benötigte Linker nur mit einer Apple Developer-Lizenz erhältlich ist und dementsprechend schwierig in die Pipeline integrierbar ist. Es sollte zwar weiterhin möglich sein, den Code für MacOS zu kompilieren, da wir aber keine Macs besitzen, können wir dies nicht selbst testen.
- **15.10.2025:** Wir konnten uns erfolgreich einen 2011 Mac mini mit MacOS 10.13 High Sierra beschaffen. Das Betriebssystem ist zwar von Apple nicht mehr unterstützt, dennoch lässt sich darauf ein x86_64 MacOS Build von GitLab Time-Report erstellen und ausführen. Ein ARM64 Build für Apples M-Prozessoren ist damit leider nicht möglich. Wir werden in Zukunft den Code in unregelmässigen Abständen auf dem Mac mini kompilieren und kurze manuelle Tests durchführen.
- **27.11.2025:** Leider können wir unser Projekt nicht mehr auf dem Mac mini kompilieren, da die von `charming` verwendete `v8`-JavaScript-Engine nicht mit einer so alten Version von MacOS kompatibel ist. Grundsätzlich sollte ein MacOS-Build aber auf neueren Versionen des Betriebssystems weiterhin möglich sein, wir können das aber nicht mehr überprüfen.
- **09.12.2025:** Wir konnten unser Programm auf dem MacBook eines Kommilitonen mit macOS Tahoe erfolgreich kompilieren und alle Funktionen funktionierten wie erwartetet.

Dieses NFR erachten wir ebenfalls als erfüllt, da unser Programm sowohl auf Linux, Windows als auch Mac kompiliert werden kann.

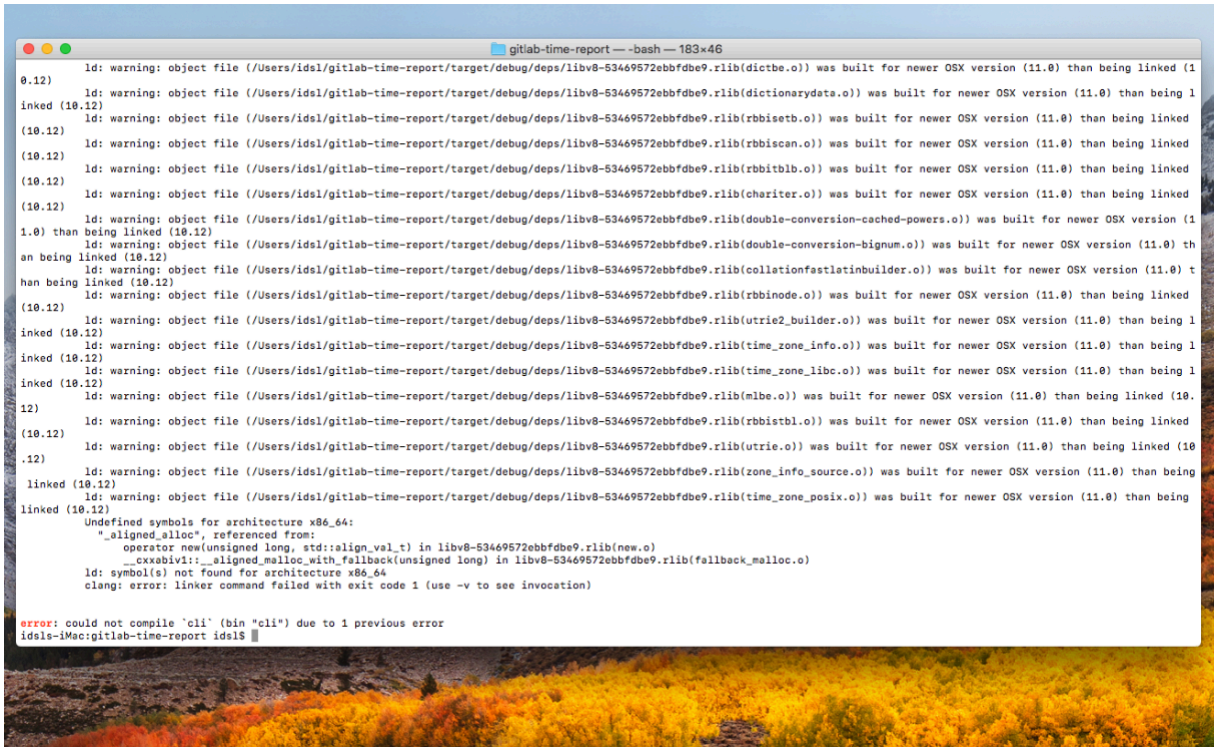


Abbildung 24: Kompilierfehler beim Erstellen der `v8`-Crate auf MacOS 10.13 High Sierra

6.1.2.6 NFR 6: Wartbarkeit – Modifizierbarkeit

«Es soll eine Anleitung für Rust-Einsteiger erstellt werden, um die Wartung des Projekts zu erleichtern.»

– **24.11.2025:** Da wir die verbleibende Zeit dazu nutzen möchten, den Code und die Dokumentation sauber abzuschliessen, haben wir zusammen mit unseren Betreuern beschlossen, die Anleitung nicht zu schreiben. Stattdessen geben wir mit dem Programm die Rust-Zusammenfassung ab, welche Jannis als Vorbereitung für die SA für sich selbst verfasst hat. Diese zählt nicht zur Bewertung der Arbeit dazu. Ebenfalls wollen wir die Doc-Comments ausführlicher gestalten und dort an den passenden Stellen entsprechende Rust-Konzepte erklären oder zu den entsprechenden Ressourcen verlinken.

Die Anleitung ist zwar in Form einer Zusammenfassung vorhanden, ist jedoch nach Absprache mit den Betreuern aus dem Scope des Projekts entfernt worden. Wir haben die Zusammenfassung «Eine_Einführung_in_Rust.pdf» für interessierte Personen ins Repository hinzugefügt.

6.2 Schlussfolgerung und Ausblick

Das zentrale Ziel, den Workflow mit dem Zeiterfassungs-Feature von GitLab zu verbessern, wurde erreicht.

Das Produkt ist unserer Meinung nach damit praxistauglich und sowohl im OST-Kontext als auch darüber hinaus gut einsetzbar, vor allem im Vergleich zu den bisher existierenden Tools.

	gitlab-time-tracker	gitlab-time-tracker (Fork)	gtt-charts	gitlab-timelogs	gitlab-time-report
Fehlerfreier Abruf der Zeiteinträge	✗	✗	✗	✓	✓
Aktive Weiterentwicklung	✗	✓	✓	✓	✓
Keine bekannten Bugs	✗	✓	✗	✓	✓
Verwendbar mit der OST-GitLab-Instanz	✓	✓	✓	✗	✓
Daten-Export	✓	✓	✗	✓	✓
Erstellen von Diagrammen	✗	✗	✓	✗	✓
Übersicht über alle Daten als Dashboard	✗	✗	✗	✗	✓

Tabelle 21: Vergleich von bestehenden Programmen und unserem Programm

Zum Abschluss dieses Projektes hat das Produkt qualitativ und bezüglich der angebotenen Features einen guten Stand erreicht. Es sollte jedoch ständig weiterentwickelt werden, um auch langfristig nützlich und aktuell zu bleiben. Durch unsere Qualitätssicherungsmassnahmen und die umfangreiche Dokumentation haben wir die nötigen Schritte unternommen, damit das Programm in Zukunft mit nur wenig Einarbeitungsaufwand von anderen Personen weiterentwickelt werden kann.

Zu diesem Zweck werden wir GitLab Time-Report nach Abschluss des Projektes unter der GPLv3-Lizenz veröffentlichen.

6.2.1 Mögliche Erweiterungen

Alle in den Functional Requirements als Erweiterungen aufgelisteten Punkte (Abschnitt 2.1.1.6), die noch nicht umgesetzt sind, wären mögliche Erweiterungen für das Projekt. Dazu zählen auch die Rückmeldungen vom Usability-Test, die ausserhalb des Projekt-Scopes lagen (Tabelle 20), sowie die Akzeptanzkriterien der functional Requirements, die wir im Projekt nicht erreicht haben, vor allem Autocomplete beim CLI.

A Projektplanung

A.1 Projektmanagement

A.1.1 Vorgehen

Als Projektmanagement-Methode wird «Scrumban» [69] verwendet, also eine Mischform von Scrum und Kanban. Von Scrum übernehmen wir die Sprints mit fixer Länge (in unserem Fall zwei Wochen), aber keine Rollenverteilung mit Product Owner, Scrum Master, etc.
Anstelle von Product Backlog und Sprint Backlog verwenden wir ein einfaches Scrumban Board (Open, In Planning, In Progress, Ready to Merge, Closed). Mit dieser Methode bleiben wir flexibel, ohne dass wir zu viel Zeit in die Verwaltung des Projektes investieren müssen.

Damit wir auch in der Langzeitplanung einen Überblick behalten, verwenden wir die Phasen von RUP (Rational Unified Process) [70]:

- **Inception:** In dieser Phase wird das ungefähre Projektziel und der Scope definiert. Zusätzlich wird eine grobe Zeitplanung vorgenommen.
- **Elaboration:** Hier werden die Requirements identifiziert und die Kernarchitektur implementiert. Die höchsten Risiken sollten in dieser Phase beseitigt und die Zeitschätzungen realistischer werden.
- **Construction:** In dieser Phase werden die Funktionen implementiert und die geringen Risiken eliminiert. Das Deployment wird vorbereitet.
- **Transition:** In dieser Phase wird das Projekt abgeschlossen. Die letzten Details werden fertiggestellt und das Produkt wird deployed.

A.1.2 Meetings

Grundsätzlich wird jede Woche ein Meeting mit den Betreuern durchgeführt, in dem der aktuelle Stand gezeigt und offene Fragen geklärt werden können. Sollten keine Fragen bestehen, kann das Meeting auch abgesagt werden.

Offizielle Team-interne Meetings sind keine geplant, da beide Team-Mitglieder vor Ort arbeiten und deshalb offene Punkte direkt angesprochen werden können.

A.2 Projektplan

In der Grafik unten ist unser Projektplan ersichtlich. Wir haben das Projekt in die zwei Arbeitsbereiche Dokumentation und Entwicklung aufgeteilt. In der Grafik ist auch die Aufteilung in die vier Phasen nach RUP und in unsere Sprints zu sehen.

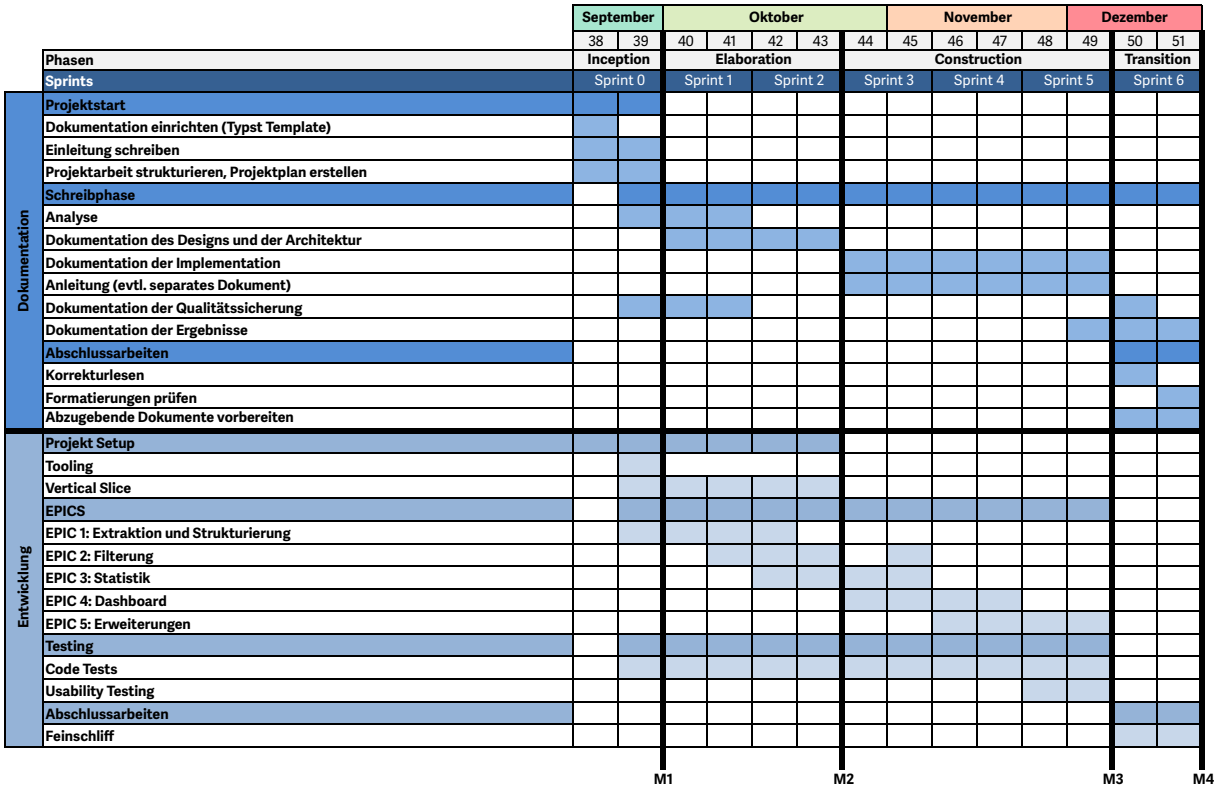


Abbildung 25: Grafischer Projektplan

A.2.1 Meilensteine

Die folgenden Meilensteine dienen als Orientierungshilfe. Welche EPICs welche Features beinhalten, wird in Abschnitt 2.1.1.7 beschrieben.

M1: End of Inception (KW39)
- Tooling und Projekt-Setup ist abgeschlossen
- Einleitung der Dokumentation ist geschrieben

M2: End of Elaboration (KW43)
- Architektur ist in der Dokumentation beschrieben
- Vertical Slice ist abgeschlossen
- EPIC 1 ist abgeschlossen

M3: End of Construction (KW49)
- Die Dokumentation ist abgesehen von finalen Anpassungen und den Ergebnissen fertig geschrieben
- EPICs 2 - 4 sind abgeschlossen

M4: Abgabe (KW51)
- Die Dokumentation ist fertig gestellt
- Das Produkt ist fertig entwickelt
- Der Abstract und alle weiteren abzugebenden Dokumente sind fertig gestellt

A.3 Risiko Management

Für das Projekt werden verschiedene Risiken identifiziert, die in der folgenden Matrix kategorisiert werden. Links ist die Eintrittswahrscheinlichkeit und oben die Auswirkung aufgeführt. Anschliessend werden die einzelnen Risiken beschrieben, sowie die Massnahmen, die ergriffen werden, um das Risiko zu minimieren. Nach jedem Sprint wird ausgewertet, ob die Risiken eingetreten sind und sie sich in der Matrix verschoben haben.

	Niedrig	**Mittel**	**Hoch**	**Sehr hoch**	**Kritisch**
Unmöglich					
Unwahrscheinlich		R1, R4		R3	
Möglich		~~R1~~, R2, R6			
Wahrscheinlich		R5			
Sehr wahrscheinlich		~~R5~~			

Tabelle 22: Risikomatrix des Projektes

Nummer	R1
Risiko	Zu steile Lernkurve
Beschreibung	Die Anwendung von Technologien, mit denen beide Team-Mitglieder noch weniger Erfahrung haben, wie zum Beispiel Rust, kann zu Problemen führen.
Massnahmen	Es wird möglichst früh im Projekt ein Prototyp erstellt. Sollte sich die Arbeit während der Erstellung des Prototyps als zu mühsam herausstellen, wird mit den Betreuern eine Abänderung der Aufgabenstellung besprochen.
Changelog	**23.10.2025:** Das Risiko hat sich als unbegründet herausgestellt. Wir konnten uns sehr gut in die Arbeit mit Rust einleben und hatten bis anhin keine grösseren Probleme oder Roadblocks. Das Risiko wurde deshalb von «möglich» zu «unwahrscheinlich» geschoben.

Tabelle 23: Beschreibung R1 «Zu steile Lernkurve»

Nummer	R2
Risiko	Ungeeignete Libraries
Beschreibung	Es lassen sich keine Libraries finden, die für unsere Zwecke geeignet sind, zum Beispiel für die Erstellung von den gewünschten Diagrammen.
Massnahmen	Das Projekt ist in einen MVP und Erweiterungen aufgeteilt. Sollte sich ein Thema als deutlich aufwändiger als geplant herausstellen, können Features nach Absprache mit dem Betreuer verkleinert oder vom MVP in die Erweiterungen geschoben werden.
Changelog	**23.10.2025:** Für das Parsen von CLI-Argumenten, für die Deserialisierung, für HTTP-Requests, für Mocking und für das Erstellen von CSV-Dateien haben wir sehr gut dokumentierte und funktionable Libraries gefunden. Bis jetzt ist dieses Risiko also nicht eingetroffen. Da jedoch noch grössere Funktionalitäten anstehen, die ebenfalls von Libraries abhängen, bleibt das Risiko weiterhin bestehen.

Tabelle 24: Beschreibung R2 «Ungeeignete Libraries»

Nummer	R3
Risiko	Änderungen an der GitLab TimeTracking API
Beschreibung	Es gibt ein Update der GitLab Instanz und die TimeTracking API funktioniert plötzlich anders.
Massnahmen	Gegen dieses Risiko kann nichts unternommen werden. Auf der Webseite mit den geplanten GitLab Releases sind jedoch keine Hinweise sichtbar, dass in der nächsten Zeit hier grössere Änderungen geplant sind [71].
Changelog	

Tabelle 25: Beschreibung R3 «Änderungen an der GitLab TimeTracking API»

Nummer	R4
Risiko	Ausfall von GitLab
Beschreibung	Es kommt zu einem kurz- oder längerfristigen Ausfall von der OST GitLab Instanz.
Massnahmen	Dieses Risiko kann nicht verhindert werden. Im schlimmsten Fall können für die Zeit des Ausfalls keine Arbeitszeiten extrahiert werden, dann wird in dieser Zeit an anderen Bereichen der SA weitergearbeitet oder mit einem Backup der Zeiten gearbeitet.
Changelog	**27.09.2025:** Zur Minimierung dieses Risikos haben wir das Mirroring des Repos mit GitLab.com aufgesetzt. Bei jedem Push auf das OST-Repository werden die Commits, Tags und Branches mit GitLab.com synchronisiert. Die Issues, Merge Requests und Labels allerdings nicht, deswegen können auf dem Mirror auch keine Zeiteinträge vorgenommen werden. Durch den Import des gesamten bestehenden Repos sind jedoch bereits einige Zeiteinträge vorhanden, mit welchen bei einem Ausfall weitergearbeitet werden könnte.

Tabelle 26: Beschreibung R4 «Ausfall von GitLab»

Nummer	R5
Risiko	Falsche Aufwand-Schätzung
Beschreibung	Es wird zu wenig Zeit für die Arbeit an den Features eingeplant.
Massnahmen	Bei jedem Sprint-Planning kann evaluiert werden, ob das Zeitbudget eingehalten wird und falls nötig können Features weggelassen werden. Die Aufteilung des Produktes in einen MVP und in Erweiterungen, welche in der Aufgabenstellung definiert wurde, bietet die dazu nötige Flexibilität.
Changelog	**23.10.2025:** Wir sind etwas hinter dem Zeitplan, aber nicht übermässig. Da wir am Ende des Projektes Zeit für die Erweiterungen eingeplant haben, sind wir noch flexibel genug und es ist nicht nötig, irgendwelche Features zu streichen. Die einzige nötige Anpassung am Zeitplan war, das erste Usability-Testing zu streichen, da dies zum aktuellen Zeitpunkt noch keinen Sinn ergibt. Das Risiko wurde von «sehr wahrscheinlich» auf «wahrscheinlich» geschoben.

Tabelle 27: Beschreibung R5 «Falsche Aufwand-Schätzung»

Nummer	R6
Risiko	Ausfall eines Team-Mitgliedes
Beschreibung	Es fällt ein Team-Mitglied durch beispielsweise Krankheit oder Unfall aus.
Massnahmen	Eine saubere Dokumentation erleichtert die Übernahme einer Arbeit durch das andere Team-Mitglied. Falls nötig, können auch Features gestrichen werden.
Changelog	**23.10.2025:** Um dieses Risiko zu minimieren, wird die Code-Dokumentation ständig aktuell gehalten.

Tabelle 28: Beschreibung R6 «Ausfall eines Team-Mitgliedes»

A.4 Tooling

In diesem Kapitel werden die von uns verwendeten Tools aufgelistet.

A.4.1 Code

Zum Schreiben von `gitlab-time-report` wurde Rust mit folgenden Tools verwendet:
- Rust Toolchains `stable-x86_64-pc-windows-msvc`, `stable-x86_64-apple-darwin` (lokal auf dem Mac mini) und `stable-x86_64-unknown-linux-gnu` 1.91.0 in der Rust 2024 Edition [72], [73]
- rustfmt und clippy (Rust Linter) [48], [74]
- JetBrains RustRover (IDE) [75]

A.4.2 Dokumentation

Die Dokumentation wird in der Typesetting-Sprache Typst verfasst. Um die Arbeit damit zu erleichtern, wurden folgende Tools verwendet:
- Visual Studio Code [76]
- Typst Compiler 0.14.1 [77]
- Tinymist (VS Code-Extension zur Preview von Typst Dokumenten) [78]
- typstyle (Typst Linter, in Tinymist integriert) [79]
- Code Spell Checker und Swiss German - Code Spell Checker [80], [81]

A.4.3 Pipeline

Innerhalb der GitLab CI/CD Pipeline [82] werden folgende Tools verwendet:
- Docker-Container des Rust-Compilers [61]
- Docker-Container des Typst-Compilers [83]
- cargo-binstall (Lädt vorkompilierte Rust-Crates herunter und installiert sie) [62]
- cargo-xwin (Lädt die MSVC Build Tools herunter und installiert sie im Linux-Container) [65]
- cargo-tarpaulin (Generiert Code Coverage Report) [57]
- junitify (Wandelt den Output von `cargo test` in das JUnit-Format um) [84]

Während der Arbeit wurden die folgenden Tools für eine gewisse Zeit verwendet, aber aus den in Abschnitt 5.3.1.2 und Abschnitt 5.3.1.3 beschriebenen Gründen ersetzt:
- cargo-llvm-cov (Generiert Code Coverage Reports) [56]
- jq (Parst das von cargo-llvm-cov generierte JSON zur Ausgabe der Code Coverage für GitLab) [85]
- mingw-w64 (Linker für die Windows-Builds mit der `stable-x86_64-pc-windows-gnu` Rust-Toolchain) [86]

A.4.4 KI-Tools

Im Rahmen dieser SA wurden KI-Tools zur Optimierung der Qualität eingesetzt. Sämtlicher Output von KI-Tools wurde vom Team nochmals manuell auf ihre Korrektheit überprüft und wenn nötig angepasst. Die Einsatzbereiche umfassen:

- Strukturieren und Formulieren von Text in der Dokumentation
- Unterstützung bei der Fehlersuche und Optimierung von Programmcode
- Evaluierung und Verbesserung von Architektur-, Design- und Frameworkentscheidungen

Die nachfolgenden KI-Tools wurden verwendet:
- OpenAI ChatGPT [87]
- DeepL Write [88]
- JetBrains AI Assistant [89]
- Anthropic Claude [90]

B Externe Code-Reviews

Nachfolgend befinden sich die vollständigen Code-Reviews von Olivier Lischer, durchgeführt am 06. November und 07. Dezember 2025.

B.1 Code-Review 1

1 Allgemein

Verwendet `#![expect(dead_code)]` anstelle von `#![allow(dead_code)]`. `expect` wirft ein Fehler, falls beim linten, der Fehler nicht auftritt.

Doc-Comments werden super verwendet.

Bezüglich linting: `cargo clippy --workspace -- -D warnings` ist schon sehr gut. Evtl. wollt ihr noch `-D clippy::pedantic` ergänzen (Usage - Clippy Documentation)

2 CLI

So weit so gut.

3 Report

3.1 http_requests.rs

Warum wird ein neuer `Client` erstellt? Warum nicht `self.post(url).json(&payload);`?

```
impl HttpFetcher for Client {
    fn http_post_request(
        &self,
        url: &str,
        payload: serde_json::Value,
        options: &FetchOptions,
    ) -> Result<String, NetworkError> {
        let http_client = Client::new();
        let mut request = http_client.post(url).json(&payload);

        // Add the access token to the query if specified
        if let Some(token) = &options.token {
            request = request.bearer_auth(token);
        }

        Ok(request.send()?.error_for_status()?.text()?)
    }
}
```

3.2 model.rs

```
/// A trackable item is either an issue or a merge request.
#[derive(Debug, PartialEq, Default, Eq, Hash, Clone)]
pub struct TrackableItem {
    /// Fields shared by all trackable items.
    pub common: TrackableItemFields,
    /// The type of the trackable item. Contains the fields that are only available for the
    specific type.
    pub kind: TrackableItemKind,
}
```

Ich hätte die Felder in `TrackableItemFields` direkt im `TrackableItem` benutzt. Ist aber auch gut so.

Unterscheidung mit `TrackableItemKind` ist so genau so wie man solche "Ich bruache Vererbung!"-Problem in Rust und ML-Sprachen löst. Sehr gut!

3.3 validation.rs

```
pub struct ExcessiveHoursValidator {
    max_hours: i16,
}
```

Verwendet hier ein `u16` oder evtl. sogar `u64` oder `usize`. Eine negative Anzahl Stunden macht hier kein Sinn.

```
impl Validator for ExcessiveHoursValidator {
    fn validate_single(&mut self, time_log: &TimeLog) -> Vec<ValidationProblem> {
        let hours = i16::try_from(time_log.time_spent.num_hours())
            .expect("Time spent should be within i16 range");

        match hours > self.max_hours {
            true => vec![ValidationProblem::ExcessiveHours {
                max_hours: self.max_hours,
            }],
            false => Vec::new(),
        }
    }
}
```

Verwendet hier nicht `expect`. `expect` sollte nur dann verwendet werden, wenn dieser case nicht passieren kann, und falls doch auf ein Bug in **eurem** Code hinweist. Hier ist es aber möglich, als User es crates, ein `TimeLog` mit einer `time_spent` $> 2\hat{}16$ (manuel) zu erstellen. In diesem Fall würde euer Programm panicen.

3.4 filters.rs

```
fn group_by_filter<'a, T, I, F>(
    nodes: I,
    filter: F,
) -> impl Iterator<Item = (&'a T, Vec<&'a TimeLog>)>
where
    T: Ord + 'a,
    I: IntoIterator<Item = &'a TimeLog>,
    F: Fn(&'a TimeLog) -> &'a T,
{
    let mut grouped = BTreeMap::<&'a T, Vec<&'a TimeLog>>::new();
    for node in nodes {
        let key = filter(node);
        grouped.entry(key).or_default().push(node);
    }
    grouped.into_iter()
}
```

Wollt ihr diese Funktion nicht auch public machen? Dann könne die Crate-Benutzer auch eigene Filters / Grouping definieren.

3.5 chrono_extensions.rs

```
fn to_readable_string(&self) -> String {
    let hours = self.num_hours();
    let minutes = self.num_minutes() % 60;
    let seconds = self.num_seconds() % 60;
    format!("{hours:02}:{minutes:02}:{seconds:02}")
}
```

Ich würde die Funktion anders bennen (z.B. `to_hms_string()` / `to_hms()`). Evtl. auch besser die Formatting-Funktionen verwenden (chrono - Rust).

```
/// Get the total minutes and hours of a `TimeDelta`. Seconds are discarded.
fn total_hours(&self) -> f32 {
    self.as_seconds_f32() / 3600.0
}
```

Funktions name und Dokumentation stimmen nicht ganz überein. Die Funktion gibt die Stunden als Kommazahl aus, Minuten und Sekunden sind als Nachkommazahl representiert. Sekunden werden also explizit **nicht** discarded (Kommazahl):

```
#[test]
fn test_total_hours_2() {
    let delta = TimeDelta::hours(5) + TimeDelta::minutes(45) + TimeDelta::seconds(10);
    assert_ne!(delta.total_hours(), 5.75);
}
```

3.6 charts_rs_extensions.rs

B.2 Code-Review 2

December 7, 2025

1 Integration tests

Viele eurer Tests basierend auf dem Predicate `contains`. Ich gehe davon aus, dass nicht der ganze Output verglichen wird, da Fehleranfällig und viel Handarbeit nötigt. Für solche fälle bietet sich Snapshot Testing an, z.B. mit insta - Rust. Ihr müsst aber nicht alle Tests auf insta umstellen.

Die Tests, welche Charts erstellen und testen, würde ich umbedingt auf insta umstellen. Nur weill die erwarteten String im HTML / SVG vorkommen, ist der Output noch lange nicht richtig. Mit insta könntet ihr auch das HTML / SVG relativ einfach testen und bei Änderungen, diese entweder übernehmen oder ihr habt einen Bug gefunden.

Evtl. könnt ihr auch noch mehr success Server Response erstellen und für jede Success Fixture den selben Tests ausführen.

2 src/main.rs

- Unnötige Type Annotation: https://gitlab.ost.ch/gitlab-time-report/gitlab-time-report/-/blob/main/src/cli/src/main.rs?ref_type=heads#L21
- Irgendwie stört es mich, dass es in diesem Branch kein `println!` hat. Ist ein rein symmetrisches Problem zu den anderen beiden Branches. https://gitlab.ost.ch/gitlab-time-report/gitlab-time-report/-/blob/main/src/cli/src/main.rs?ref_type=heads#L40-48

3 src/dashboard/html.rs

Tolle Arbeit. Ich habe die Dokumentation nicht gelesen, aber ich frage mich weshalb ihr euch gegen eine Template Engine wie Tera, askama entschieden habt.

4 Modules

FYI: Die `mod.rs` gelten mitlerweile als ”older style”. Es wird empfohlen die neue Variante zu verwenden. Der einzige Nachteil, der alten Variante ist, dass man potenziell viele `mod.rs`-Datei hat (https://doc.rust-lang.org/stable/book/ch07-05-separating-modules-into-different-files.html). Daher sehe ich kein Grund was zu ändern.

5 Code Coverage

Mit tarpaulin habt ihr doch an einigen Stellen ein `#![cfg(not(tarpaulin))]`. Für ein zukünftiges Projekt könnt ihr euch vieleicht auch mal noch eine Alternative anschauen.

- https://doc.rust-lang.org/rustc/instrument-coverage.html
- https://blog.rng0.io/how-to-do-code-coverage-in-rust/

C Usability Testing Protokolle

Das Usability Testing wurde am 04.12.2025 mit 6 Probanden durchgeführt. Hier sind die einzelnen Feedbacks der Teilnehmenden aufgelistet.

Name	Daniel
Feedback	– Informationen zur Token-Erstellung sollten zusätzlich und klar sichtbar im README dokumentiert sein. – Die Validierung wurde nicht als solche erkannt und sollte mit einem eindeutigen Titel gekennzeichnet werden. – Die `weeks-per-sprint`-Flag sollte nicht optional sein. – Die `help`-Flag sollte konsistent entweder bei allen oder bei keinem Subcommand aufgeführt werden. – Status-Nachrichten im CLI-Output sollten vereinheitlicht werden – Die Fehlermeldung «File containing chart settings not found» sollte «Theme path not found» heissen
Note	5.5

Tabelle 29: Rückmeldungen vom Usability Testing 1

Name	Valentino
Feedback	– Der Help-Text des `Charts`-Commands ist bezüglich verfügbaren Diagramm-Typen nicht mehr aktuell und sollte angepasst werden – Dass standardmässig Statistiken auf der Konsole ausgegeben werden, ist zwar im Help-Text erwähnt, geht aber leicht unter und sollte prominenter platziert werden – Die Validierung wurde nicht als solche erkannt. Sie sollte standardmässig nur als Zusammenfassung erscheinen und bei Bedarf mit einer Flag umfangreich angezeigt werden können – Status-Nachrichten im CLI-Output sollten vereinheitlicht werden – Wenn Dateien erstellt werden, sollten diese direkt im CLI verlinkt sein – Mit einer `--show`-Flag sollte festgelegt werden können, welche Tabellen im CLI angezeigt werden – Der Help-Text sollte besser strukturiert werden – Es ist nicht klar ersichtlich, welche Flags required sind. Dies sollte eindeutiger gekennzeichnet werden – Im Dashboard wurde nicht erkannt, dass man scrollen muss, um weitere Inhalte zu sehen – Support für Polyrepos wurde gewünscht
Note	5.5

Tabelle 30: Rückmeldungen vom Usability Testing 2

Name	Andreas
Feedback	– Die Reihenfolge von Subcommands und URL wird als verwirrend empfunden und sollte klarer gestaltet werden – Autocomplete für die Commands und Flags wäre hilfreich, um die Bedienung effizienter und fehlertoleranter zu gestalten.
Note	5.0

Tabelle 31: Rückmeldungen vom Usability Testing 3

Name	Joel
Feedback	– Im Help-Text steht der Satz «The subcommands of gitlab time report» an einer unpassenden Stelle und sollte korrekt platziert oder entfernt werden – Bei der `--labels`-Flag sollte explizit erwähnt werden, dass nach dem Komma kein Leerzeichen erlaubt ist – Tabellen sind alphabetisch sortiert, womit das Label `others` nicht immer am Ende steht. – Die `help`-Flag sollte entweder bei allen oder bei keinem der Subcommands erwähnt werden – Eine Environment-Variable für die Chart-Settings wäre hilfreich, damit die gleichen Daten nicht bei jedem Aufruf erneut eingetippt werden müssen – Beim Erstellen von Dateien sollte im CLI ein klarer Hinweis stehen, wo die Datei abgelegt wurde.
Note	5.5

Tabelle 32: Rückmeldungen vom Usability Testing 4

Name	Timo
Feedback	– Dass standardmässig Statistiken auf der Konsole ausgegeben werden, ist zwar im Help-Text erwähnt, geht aber leicht unter und sollte prominenter platziert werden – Farben im CLI-Output wären für die visuelle Unterscheidung von Informationen und Warnungen hilfreich – Die Reihenfolge von Subcommands und URL wird als verwirrend empfunden und sollte klarer kommuniziert werden – Es ist nicht klar ersichtlich, welche Flags required sind. Dies sollte eindeutiger gekennzeichnet werden – Es wäre praktisch, wenn man mit der `--labels`-Flag neben expliziten Einschlüssen auch Labels ausschliessen könnte
Note	5.25

Tabelle 33: Rückmeldungen vom Usability Testing 5

Name	Sven
Feedback	– Die Validierung wurde nicht als solche erkannt. – Es wäre nützlich, einzelne Validierungen separat ein- und ausschalten zu können – Der Help-Text der `--labels`-Flag ist nicht einfach verständlich und sollte klarer formuliert werden – Nachkommastellen bei Stunden in den Diagrammen sollten gerundet werden, um die Lesbarkeit zu erhöhen – Die Flags des `Chart`-Commands sind teilweise verwirrend, vor allem ist nicht klar, was Pflicht ist und was nicht. Die Werte werden nur für die Burndown-Charts benötigt, trotzdem kann man die anderen Charts ohne diese auch nicht generieren. – Die Legende mancher Charts überlagert Teile des Inhalts
Note	5.5

Tabelle 34: Rückmeldungen vom Usability Testing 6

D Test Coverage Übersicht

Die Test Coverage der Unit-Tests des Projektes beträgt 84.92%.
Wie in Abschnitt 5.2.3 beschrieben, haben wir die Integration-Tests aus der Testabdeckung entfernt.
Der komplette Coverage Report ist diesem PDF als Dateianhang beigelegt.

Back /builds/gitlab-time-report/gitlab-time-report/src/gitlab-time-report/src Covered: 518 of 610 (84.92%)

Path	Coverage
charts	149 / 226 (65.93%)
dashboard	82 / 84 (97.62%)
export	36 / 40 (90.00%)
fetch_api	67 / 70 (95.71%)
chrono_extensions.rs	6 / 6 (100.00%)
filters.rs	53 / 53 (100.00%)
lib.rs	0 / 0
model.rs	10 / 12 (83.33%)
tables.rs	63 / 63 (100.00%)
validation.rs	52 / 56 (92.86%)

Abbildung 26: Die Code-Abdeckung der Library-Module

E Glossar und Abkürzungsverzeichnis

E.1 Glossar

Hier befinden sich Erklärungen zu projektspezifischen Begriffen.

Access Token Ein Access Token kann wie ein Passwort verwendet werden. Beim Erstellen eines Access Tokens kann festgelegt werden, auf welche Ressourcen dieser Zugriff hat und wann dieser abläuft. In GitLab kann ein Access Token für User, Gruppen oder Repositories erstellt werden. Für Zugriff auf nicht-öffentliche Repositories zwingend nötig.

Crate Der gesamte Rust-Code wird zu einer Crate zusammengefasst. Eine Crate ist die kleinste Einheit, die der Compiler berücksichtigt. Sie wird entweder zu einem Binary oder einer Library kompiliert. Externe Libraries werden oft auch als Crate bezeichnet.

CRUD Create, Read, Update, Delete. Beschreibt das Erstellen, Lesen, Bearbeiten und Löschen von Einträgen, meist in einer Datenbank.

Epic Eine grössere Anforderung an das System (zum Beispiel ein grösseres Feature), die in mehrere User Stories aufgeteilt werden kann.

Happy Path Ein Happy Path stellt den normalen Programmflow ohne Fehler oder Ausnahmen dar.

HTML Template-Engine Eine Template-Engine ist eine Software, die Platzhalter in einer Vorlage mit aktuellen Inhalten ersetzt und daraus ein fertiges Dokument erzeugt. Eine HTML Template-Engine wird dazu verwendet, um HTML-Templates mit dynamischen Inhalten zu befüllen.

Library Eine Library ist eine Sammlung von wiederverwendbarem Programmcode, die bestimmte Funktionen zur Verwendung bereitstellt. Im Rust-Kontext wird eine Library als Crate kompiliert.

Meilenstein Definierter Zeitpunkt in einem Projekt, an welchem bestimmte Ziele oder Ergebnisse erreicht werden sollen. In GitLab können Meilensteine verwendet werden, um Issues und Merge Requests zu gruppieren und den Fortschritt eines Projektes zu verfolgen.

Modul Logische Einheit in einem Rust-Projekt. Ist eine Sammlung von Funktionen, Structs, Traits, etc. Existiert entweder innerhalb einer Datei, als eigene Modul-Datei oder als ein Verzeichnis, welches eine `mod.rs`-Datei beinhaltet.

Panic Stösst ein Rust-Programm auf einen nicht behandelten Fehler, löst es eine Panic aus. Das Programm terminiert und zeigt auf der Konsole eine Fehlermeldung an, wieso die Panic aufgetreten ist. Vergleichbar mit C/C++ SEGFAULT.

Repository (Repo) Eine Datei- und Ordnerstruktur, welche von einem Versionskontrollsystem wie Git verwaltet wird. Ein Projekt auf einer Code Forge-Website wie GitHub oder GitLab wird als Repository bezeichnet. Während bei Git jeder User ein Repository lokal auf seinem Gerät besitzt, wird mit dem Begriff häufig das auf der Code Forge liegende Repository bezeichnet.

Subcommand Ein Befehl, der in unserem CLI-Frontend beim Ausführen des Programms angegeben werden kann, damit eine bestimmte Funktionalität des Programms genutzt wird.

Toolchain Eine spezifische Installation des Rust Compilers und verwandten Tools wie der Paketmanager Cargo. Der Name einer Toolchain setzt sich aus dem Channel (z.B. `stable` oder `nightly`) und dem Host-Triplet zusammen. Letzteres besteht aus der Prozessor-Architektur (z.B. `x86_64` oder `aarch64`), dem Vendor (z.B. `apple`, `pc` oder `unknown`) und dem OS und dem Linker (z.B. `windows-msvc` oder `linux-gnu`).

User Story Eine Anforderung an das System aus Sicht der User. Beschreibt, was die User tun möchten und welchen Nutzen sie daraus ziehen.

E.2 Abkürzungsverzeichnis

In diesem Kapitel werden die in der Arbeit verwendeten Akronyme ausgeschrieben.

BA Bachelorarbeit

CI/CD Continuous Integration / Continuous Delivery

CLI Command Line Interface, Kommandozeilen-Schnittstelle

KW Kalenderwoche

M Meilenstein

MR Merge Request

OS Operating System / Betriebssystem

OST Ostschweizer Fachhochschule

RUP Rational Unified Process

SA Studienarbeit

SEProj SE Project / Software Engineering Project

US User Story

F Tabellenverzeichnis

G Abbildungsverzeichnis

H Quellcodeverzeichnis

I Bibliographie

[1] «Time Tracking | GitLab Docs». GitLab. Zugegriffen: 18. September 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/user/project/time_tracking/

[2] kriskbx, *gitlab-time-tracker: A command line interface for GitLab's time tracking feature.* Zugegriffen: 18. September 2025. [Online]. Verfügbar unter: https://github.com/kriskbx/gitlab-time-tracker

[3] Moritz "moschi" Schiesser, *gtt-charts*. Zugegriffen: 18. September 2025. [Online]. Verfügbar unter: https://github.com/moschi/gtt-charts

[4] «What is User Story Template?». Zugegriffen: 27. September 2025. [Online]. Verfügbar unter: https://agilealliance.org/glossary/user-story-template/

[5] ISO Committee, «ISO/IEC 25010:2023 -- Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model», Nov. 2023. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://www.iso.org/standard/78176.html

[6] «Update on ISO 25010, version 2023». Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://quality.arc42.org/articles/iso-25010-update-2023

[7] kriskbx, «gitlab-time-tracker Documentation». Zugegriffen: 23. September 2025. [Online]. Verfügbar unter: https://github.com/kriskbx/gitlab-time-tracker/blob/master/documentation.md

[8] danielh186, «Fix detection of deleted time, Pull Request #159, gitlab-time-tracker». Zugegriffen: 23. September 2025. [Online]. Verfügbar unter: https://github.com/kriskbx/gitlab-time-tracker/pull/159

[9] jdecaron, jedecaron, und cgdobre, «Fix rate limit of 600 requests per minutes on Gitlab.com (#106), commit f377108, gitlab-time-tracker». 7. Juni 2020. Zugegriffen: 23. September 2025. [Online]. Verfügbar unter: https://github.com/kriskbx/gitlab-time-tracker/commit/f3771088cf6915142ad81536c53142ca6eff9a6b

[10] Andreas "ndu2" Müller, *gitlab-time-tracker [Forked from kriskbx]: A command line interface for GitLab's time tracking feature.* Zugegriffen: 23. September 2025. [Online]. Verfügbar unter: https://github.com/ndu2/gitlab-time-tracker

[11] Andreas "ndu2" Müller, «support new gitlab version with can "delete" time notes, commit 715569a, gitlab-time-tracker [Fork]». 28. September 2022. Zugegriffen: 23. September 2025. [Online]. Verfügbar unter: https://github.com/ndu2/gitlab-time-tracker/commit/715569a

[12] Farhad D. Mehta und Moritz Schiesser, «Integrate gtt-charts into the documentation in case it works well enough., Issue #1, SE Project/GitLab Time Tracking Tutorial». Zugegriffen: 23. September 2025. [Online]. Verfügbar unter: https://gitlab.ost.ch/SEProj/GitLab_Time_Tracking_Tutorial/-/issues/1

[13] Moritz "moschi" Schiesser, «gtt-charts Example Time Report». 21. März 2021. Zugegriffen: 24. September 2025. [Online]. Verfügbar unter: https://github.com/moschi/gtt-charts/blob/main/example/Timereport.md

[14] phip1611, *gitlab-timelogs: CLI utility to support you with your time logs in GitLab.* Zugegriffen: 24. September 2025. [Online]. Verfügbar unter: https://github.com/phip1611/gitlab-timelogs

[15] phip1611, «Fetch Epics only when server supports · Issue #39 · phip1611/gitlab-timelogs». Zugegriffen: 24. September 2025. [Online]. Verfügbar unter: https://github.com/phip1611/gitlab-timelogs/issues/39

[16] «Personal access tokens | GitLab Docs». GitLab. Zugegriffen: 5. Oktober 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/user/profile/personal_access_tokens

[17] «Group access tokens | GitLab Docs». GitLab. Zugegriffen: 5. Oktober 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/user/group/settings/group_access_tokens/

[18] «Project access tokens | GitLab Docs». GitLab. Zugegriffen: 5. Oktober 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/user/project/settings/project_access_tokens

[19] «Roles and Permissions: Project Planning | GitLab Docs». GitLab. Zugegriffen: 5. Oktober 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/user/permissions/#project-planning

[20] S. Brown, «The C4 Model for Software Architecture», 25. Juni 2018, *InfoQ*. Zugegriffen: 7. Oktober 2025. [Online]. Verfügbar unter: https://www.infoq.com/articles/C4-architecture-model/

[21] «reqwest Documentation». docs.rs. Zugegriffen: 22. Oktober 2025. [Online]. Verfügbar unter: https://docs.rs/reqwest/latest/reqwest/

[22] «serde Documentation». docs.rs. Zugegriffen: 22. Oktober 2025. [Online]. Verfügbar unter: https://docs.rs/serde/latest/serde/

[23] «serde_json Documentation». docs.rs. Zugegriffen: 22. Oktober 2025. [Online]. Verfügbar unter: https://docs.rs/serde_json/latest/serde_json/

[24] «tokio Documentation». docs.rs. Zugegriffen: 22. Oktober 2025. [Online]. Verfügbar unter: https://docs.rs/tokio/latest/tokio/

[25] «Verwendung der Paginierung in der GraphQL-API». Zugegriffen: 26. Oktober 2025. [Online]. Verfügbar unter: https://docs.github.com/de/enterprise-cloud@latest/graphql/guides/using-pagination-in-the-graphql-api

[26] «GraphQL API resources - PageInfo». Zugegriffen: 26. Oktober 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/api/graphql/reference/#pageinfo

[27] «mockall Documentation». docs.rs. Zugegriffen: 22. Oktober 2025. [Online]. Verfügbar unter: https://docs.rs/mockall/latest/mockall/

[28] TheJanzap, «How do you deal with modeling types with lots of shared fields?». Zugegriffen: 27. Oktober 2025. [Online]. Verfügbar unter: https://users.rust-lang.org/t/how-do-you-deal-with-modeling-types-with-lots-of-shared-fields/134612

[29] «Chain of Responsibility», *Refactoring Guru*. Zugegriffen: 21. Oktober 2025. [Online]. Verfügbar unter: https://refactoring.guru/design-patterns/chain-of-responsibility

[30] «csv Documentation». docs.rs. Zugegriffen: 28. Oktober 2025. [Online]. Verfügbar unter: https://docs.rs/csv/latest/csv/

[31] «charming Documentation». docs.rs. Zugegriffen: 13. November 2025. [Online]. Verfügbar unter: https://docs.rs/charming/latest/charming/

[32] «Apache ECharts». Zugegriffen: 13. November 2025. [Online]. Verfügbar unter: https://echarts.apache.org/en/index.html

[33] «charts_rs Documentation». Zugegriffen: 13. November 2025. [Online]. Verfügbar unter: https://docs.rs/charts_rs/latest/charts_rs/

[34] TheJanzap, «X-Axis label disappear if text is too long, even if they would not overlap · Issue #34 · vicanso/charts-rs». Zugegriffen: 13. November 2025. [Online]. Verfügbar unter: https://github.com/vicanso/charts-rs/issues/34

[35] TheJanzap, «Data points get rounded when converted to SVG · Issue #36 · vicanso/charts-rs». Zugegriffen: 13. November 2025. [Online]. Verfügbar unter: https://github.com/vicanso/charts-rs/issues/36

[36] TheJanzap, «Legend alignment breaks when legend is too long · Issue #35 · vicanso/charts-rs». Zugegriffen: 13. November 2025. [Online]. Verfügbar unter: https://github.com/vicanso/charts-rs/issues/34

[37] «Theme Builder - Apache ECharts». Zugegriffen: 27. November 2025. [Online]. Verfügbar unter: https://echarts.apache.org/en/theme-builder.html

[38] TheJanzap, «Unable to set theme on HTML renderer · Issue #207 · yuankunzhang/charming». Zugegriffen: 11. Dezember 2025. [Online]. Verfügbar unter: https://github.com/yuankunzhang/charming/issues/207

[39] «build_html Documentation». docs.rs. Zugegriffen: 11. Dezember 2025. [Online]. Verfügbar unter: https://docs.rs/build_html/latest/build_html/

[40] «prefers-color-prefersColorScheme». MDN Web Docs - Mozilla. Zugegriffen: 11. Dezember 2025. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/CSS/Reference/At-rules/@media/prefers-color-scheme

[41] «clap Documentation». docs.rs. Zugegriffen: 9. November 2025. [Online]. Verfügbar unter: https://docs.rs/clap/latest/clap/

[42] «Tutorial for the Derive API - clap Tutorial». docs.rs. Zugegriffen: 9. November 2025. [Online]. Verfügbar unter: https://docs.rs/clap/latest/clap/_derive/_tutorial/index.html

[43] «Tutorial for the Builder API - clap Tutorial». docs.rs. Zugegriffen: 9. November 2025. [Online]. Verfügbar unter: https://docs.rs/clap/latest/clap/_tutorial/index.html

[44] «When should i use the builder vs derive APIs? - clap FAQ». docs.rs. Zugegriffen: 9. November 2025. [Online]. Verfügbar unter: https://docs.rs/clap/latest/clap/_faq/index.html#when-should-i-use-the-builder-vs-derive-apis

[45] Conventional Commits contributors, «Conventional Commits: A specification for adding human and machine readable meaning to commit messages». Zugegriffen: 19. September 2025. [Online]. Verfügbar unter: https://www.conventionalcommits.org/en/v1.0.0/

[46] «Git feature branch workflow», *Atlassian*. Zugegriffen: 19. September 2025. [Online]. Verfügbar unter: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

[47] «Gitflow workflow», *Atlassian*. Zugegriffen: 19. September 2025. [Online]. Verfügbar unter: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[48] The Rust Project Developers, *clippy: A bunch of lints to catch common mistakes and improve your Rust code.* Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://github.com/rust-lang/rust-clippy

[49] Clippy contributors, «Pedantic: Clippy's Lints - Clippy Documentation». Zugegriffen: 21. November 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/clippy/lints.html#pedantic

[50] Clippy contributors, «Clippy Lints - match_bool». Zugegriffen: 14. Dezember 2025. [Online]. Verfügbar unter: https://rust-lang.github.io/rust-clippy/master/index.html#match_bool

[51] Clippy contributors, «Clippy Lints - implicit_hasher». Zugegriffen: 14. Dezember 2025. [Online]. Verfügbar unter: https://rust-lang.github.io/rust-clippy/master/index.html#implicit_hasher

[52] T. Erhart und C. Furrer, «Orientierungshilfe für die Dokumentation einer Studien- oder Bachelorarbeit», OST - Ostschweizer Fachhochschule, Apr. 2025. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://ostch.sharepoint.com/:b:/r/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten/Informationen/Orientierungshilfe%20SA%20BA%20Dokumentation.pdf?csf=1&web=1&e=sbjUEz

[53] S. Richter und M. Stocker, «Leitfaden für Bachelor- und Studienarbeiten», OST - Ostschweizer Fachhochschule, Jan. 2025. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://ostch.sharepoint.com/:b:/r/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten/Leitfaden%20BA%20SA%20v.1.3.pdf?csf=1&web=1&e=tRcv9M

[54] F. D. Mehta und T. Kählin, *SE Project LaTeX Documentation Template*. OST - Ostschweizer Fachhochschule. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://gitlab.ost.ch/SEProj/latex-documentation-template

[55] D. Glasl, *Arbeiten-Vorlage-Typst: Vorlage für Arbeiten an der Fachhochschule OST mit Typst*. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://github.com/darioglasl/Arbeiten-Vorlage-Typst

[56] taiki-e, *cargo-llvm-cov: Cargo subcommand to easily use LLVM source-based code coverage (-C instrument-coverage)*. Zugegriffen: 28. September 2025. [Online]. Verfügbar unter: https://github.com/taiki-e/cargo-llvm-cov

[57] xd009642, *cargo-tarpaulin: A code coverage tool for Rust projects*. Zugegriffen: 17. November 2025. [Online]. Verfügbar unter: https://github.com/xd009642/tarpaulin

[58] «assert_cmd documentation». docs.rs. Zugegriffen: 27. November 2025. [Online]. Verfügbar unter: https://docs.rs/assert_cmd/latest/assert_cmd

[59] «mockito documentation». docs.rs. Zugegriffen: 27. November 2025. [Online]. Verfügbar unter: https://docs.rs/mockito/latest/mockito

[60] toukakirishma, «Unvalidated `timeSpent` value in POST /api/graphql `createTimelog` operation leads to unable to load issues on Issue board (#418226) · Issue · gitlab-org/gitlab». Zugegriffen: 17. Dezember 2025. [Online]. Verfügbar unter: https://gitlab.com/gitlab-org/gitlab/-/issues/418226

[61] The Rust Project Developers, «Image Layer Details - rust:1.91 | Docker Hub». Zugegriffen: 3. November 2025. [Online]. Verfügbar unter: https://hub.docker.com/layers/library/rust/1.91/images/sha256-0f36b8c5b45a1403f31ae3e1f2db1be6126a1a8d3d89157dfbf1a9caf8364a3d

[62] cargo-bins organization, *cargo-binstall: Binary installation for rust projects*. Zugegriffen: 17. November 2025. [Online]. Verfügbar unter: https://github.com/cargo-bins/cargo-binstall

[63] The rustup contributors, «Windows - The rustup book». Zugegriffen: 18. November 2025. [Online]. Verfügbar unter: https://rust-lang.github.io/rustup/installation/windows.html

[64] «404 when building v137.3.0 with `x86_64-pc-windows-gnu` toolchain · Issue #1877 · denoland/rusty_v8». Zugegriffen: 18. November 2025. [Online]. Verfügbar unter: https://github.com/denoland/rusty_v8/issues/1877#issuecomment-3527745467

[65] rust-cross organization, *cargo-xwin: Cross compile Cargo project to Windows MSVC target with ease*. Zugegriffen: 17. November 2025. [Online]. Verfügbar unter: https://github.com/rust-cross/cargo-xwin

[66] The Rust Unstable Book contributors, «RUSTC_BOOTSTRAP - The Rust Unstable Book». Zugegriffen: 18. Dezember 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/beta/unstable-book/compiler-environment-variables/RUSTC_BOOTSTRAP.html

[67] «cargo-tarpaulin: README.md -- Ignoring files in code». Zugegriffen: 18. November 2025. [Online]. Verfügbar unter: https://github.com/xd009642/tarpaulin?tab=readme-ov-file#ignoring-code-in-files

[68] «needs:project - CI/CD YAML syntax reference | GitLab Docs». GitLab. Zugegriffen: 30. September 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/ci/yaml/#needsproject

[69] «Scrumban: Mastering two Agile methodologies», *Atlassian*. Zugegriffen: 23. September 2025. [Online]. Verfügbar unter: https://www.atlassian.com/agile/project-management/scrumban

[70] S. Kapferer und T. Kälin, «SE Practices 2: 01 - Project Planning», OST - Ostschweizer Fachhochschule, Feb. 2025.

[71] «GitLab upcoming releases», *GitLab*. Zugegriffen: 18. September 2025. [Online]. Verfügbar unter: https://about.gitlab.com/upcoming-releases/

[72] The rustc Book contributors, «The rustc Book: Platform Support». Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/stable/rustc/platform-support.html

[73] The Rust Edition Guide contributors, «The Rust Edition Guide: What are Editions?». Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/edition-guide/editions/index.html

[74] rustfmt contributors, *rustfmt: Format Rust code*. Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://github.com/rust-lang/rustfmt

[75] «RustRover: Die Rust-IDE von JetBrains». Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://www.jetbrains.com/de-de/rust/

[76] Microsoft, «Visual Studio Code - Code Editing, Redefined.». Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://code.visualstudio.com/

[77] Typst contributors, *Typst: A new markup-based typesetting system that is powerful and easy to learn. - Version 0.14.1 (December 3, 2025)*. Zugegriffen: 5. Dezember 2025. [Online]. Verfügbar unter: https://github.com/typst/typst/releases/tag/v0.14.1

[78] Myriad-Dreamin et al., *tinymist: Tinymist [ˈtaɪni mɪst] is an integrated language service for Typst [taɪpst].* Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://github.com/Myriad-Dreamin/tinymist

[79] typstyle contributors, *typstyle: Beautiful and reliable typst code formatter.* Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://github.com/typstyle-rs/typstyle

[80] Street Side Software, *vscode-spell-checker: A simple source code spell checker for code*. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://github.com/streetsidesoftware/vscode-spell-checker

[81] Street Side Software, «Swiss German - Code Spell Checker: Swiss German dictionary extension for VS code». Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://marketplace.visualstudio.com/items?itemName=streetsidesoftware.code-spell-checker-swiss-german

[82] «CI/CD pipelines | GitLab Docs». GitLab. Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://docs.gitlab.com/ci/pipelines/

[83] Typst contributors, «Typst Container 0.14.1». Zugegriffen: 5. Dezember 2025. [Online]. Verfügbar unter: https://github.com/typst/typst/pkgs/container/typst/598388272?tag=0.14.1

[84] Kores, *junitify: Takes cargo test JSON and transform to JUnit XML*. Zugegriffen: 5. Oktober 2025. [Online]. Verfügbar unter: https://gitlab.com/Kores/junitify

[85] jqlang organization, *jq: Command-line JSON processor*. Zugegriffen: 28. September 2025. [Online]. Verfügbar unter: https://github.com/jqlang/jq

[86] mingw-w64 contributors, «mingw-w64: A complete runtime environment for GCC & LLVM for 32-bit (x86), 64-bit (x64), and ARM64 Windows». Zugegriffen: 5. Oktober 2025. [Online]. Verfügbar unter: https://www.mingw-w64.org/

[87] «ChatGPT». Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://chatgpt.com/

[88] «DeepL Write». Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://www.deepl.com/de/write

[89] «AI Assistant features». Zugegriffen: 22. September 2025. [Online]. Verfügbar unter: https://www.jetbrains.com/ai-assistant/

[90] «Claude AI». Zugegriffen: 17. November 2025. [Online]. Verfügbar unter: https://claude.ai/