Cimiano P., Pietsch C., & Wiljes C. (Eds.) (2021). Studies in Analytical Reproducibility: the Conquaire Project. Bielefeld. https://doi.org/10.4119/unibi/2942780

2 | Conquaire Infrastructure for Continuous Quality Control

Fabian Herrmann[1], Christian Pietsch[2], Philipp Cimiano[1]

1 – Semantic Computing Group, Faculty of Technology & Cognitive Interaction Technology Excellence Center, Bielefeld University
2 – Bielefeld University Library, Bielefeld University

Abstract

In this chapter, we briefly describe the Git-based infrastructure that has been implemented as a result of the Conquaire project to support analytical reproducibility. The infrastructure implemented relies on principles of continuous integration as used in software engineering projects. Importantly, we rely on a distributed version control system (DVCS) to store computational artifacts that are key to reproducing the analytical results of an experiment. Using a DVCS has the benefit that artifacts can be versioned and each version can be uniquely addressed via a revision number. The DVCS implementation we rely on is Git, extended by GitLab as a web interface and collaboration platform. The heart of the infrastructure implemented in the Conquaire project is the so called *Conquaire server*. We assume that each research project deposits relevant data and code in a Git repository. In the background, a GitLab CI Runner on the Conquaire server is triggered by Git push events on the local GitLab server and executes a number of tests on the data and runs the code or script to reproduce a particular result. By this, we ensure that results can be reproduced independently of the original researchers on a separate machine. In this chapter, we briefly describe the infrastructure implemented and how tests are automatically executed when updates are committed to the Git repository.

2.1 Introduction

Principles of continuous integration have long been applied in software engineering to increase the quality of software artifacts and to prevent issues and failures due do integration of code developed in a distributed fashion by multiple developers in large software engineering projects. The heart of any continuous integration setup is typically a so-called *integration server* that runs a number

CC BY

of tests once updates of the software are committed and pushed, and possibly rejects the committed changes if they do not pass a number of tests. In continuous integration, software developers are encouraged to submit smaller changes in regular intervals to prevent errors and the well known *'integration hell'*.

Inspired by continuous integration principles, in Conquaire we have attempted to transfer these principles from the domain of software engineering into the domain of research data management. The starting point for any continuous integration is the availability of a repository into which data and code can be committed. Thus, a central part of the Conquaire architecture is a Distributed Version Control System (DVCS) that allows researchers to deposit their artifacts into a central repository. An important advantage of such a repository is that data and code can be versioned and each version can be uniquely referenced by a specific revision number. This allows to pinpoint and reference the exact version of code and data that was used to obtain a certain result, a central element of reproducibility.

Within Conquaire, we selected Git as a DVCS and GitLab as a web interface and collaboration platform to implement a university-wide repository allowing researchers to store their digital and computational research artifacts, code and data in particular. A key component of the Conquaire architecture is the so called *Conquaire server*, which in Conquaire acts as a continuous integration server. Upon a new commit, the GitLab CI Runner on the Conquaire server executes a number of predefined tests on code and data and runs code or scripts on data with the goal of reproducing a specific result. A central goal of Conquaire is to support the reproduction of a certain result independently on a separate machine that is out of the direct control of the original researchers.

The Conquaire server applies a number of quality checks on the data and publishes the results of these tests on a web server, sending an email to the person that committed the data to inform about the result of the test. Thus, researchers can get informed on the fact whether there are any problems with their data so that they can react early. We distinguish in Conquaire between generic tests that are specific for a certain file format (e.g. CSV or XML) and tests that are specific to the particular research projects. After tests are run, corresponding badges are generated indicating whether the tests were passed or not and rendered within a report that summarizes the results of the test. Conquaire is thus using principles from gamification to score the quality of data and thus create incentives for researchers to strive for high quality data that passes all tests.

In this chapter, we briefly describe the Conquaire approach to continuous integration as well as the core pieces and modules of the infrastructure implemented as part of the Conquaire project to verify quality of the data. In Section 2.2, we motivate our choice for Git and GitLab. In Section 2.3, we describe how we have implemented the Conquaire continuous integration infrastructure.

2.2 Why we use Git and GitLab

2.2.1 Git

One of the inspirations for this project came from the observation that GitHub had become popular not just among software developers, but also among other knowledge workers such as scientists. GitHub, as the name suggests, is built around Git (although second-class support for SVN was added later). So, of course, we looked at Git first, but we avoided committing ourselves to Git in the project proposal because a fair evaluation of all options was to be part of the project. We have to admit that we did not conduct a deep and thorough research to find alternatives. Git is the dominant versioning software today, and there is no foreseeable competitor. According to a survey by the popular question and answer website StackOverflow in 2015[1], out of 16,694 participants who answered this question, 69.3% used Git, 36.9% used SVN, 12.2% used TFS, 7.9% used Mercurial, 4.2% used CVS, 3.3% used Perforce, 5.8% used some other versioning software, and 9.3% used no versioning software at all. Other studies[2] come to similar conclusions.

Teaching researchers how to use a versioning software that is not widely used (such as Mercurial or Perforce) or is limited to one operating system (such as TFS) or is obsolete (such as CVS) was out of the question as we will not always be there to support them. Eventually, when they require help from other colleagues or their system administrator, Git will most likely be one of the versioning software they will know and provide support for. Of course, there are other criteria besides popularity that must be considered. A clear benefit of distributed versioning systems is that they can be used offline as they maintain the full history, including branches, locally. This is crucial to ensure long-term availability of data as lots of copies keep stuff safe, as the saying goes. As SVN is not a distributed versioning system, this alternative is ruled out. It goes without saying that any software used for archiving should be open source and freely licensed (FOSS). At the very least, its storage format must be documented openly. Freely available source code is a very precise way of documenting a storage format. Table 2.2 summarizes the main features that lead us to the decision to use Git to implement a university-wide distributed version control system.

We see two main disadvantages of using Git: (1) problems related learnability/usability and (2) lack of support of large files. Regarding learnability, finding out how hard it is for non-technical users to learn to use Git will be one of the outcomes of this project. Our working hypothesis is that for versioning research data, it is sufficient to learn a small subset of Git, which should not be too challenging. With respect to large files, the problem is that Git was originally not intended to be used with large files. The same is true for most versioning

[1] `https://insights.stackoverflow.com/survey/2015`
[2] `https://rhodecode.com/insights/version-control-systems-2016`

software name	popularity	actively maintained	distributed	cross-platform	FOSS
CVS	low	no	no	yes	yes
Git	high	yes	yes	yes	yes
Mercurial	low	yes	yes	yes	yes
Perforce	low	yes	no	yes	no
SVN	medium	yes	no	yes	yes
TFS	low	yes	no	no	no

Table 2.2: Features of different versioning systems

systems. They are intended for tracking changes that are caused by intellectual efforts: these rarely result in large files directly. Still, we want to include large files such as video recordings when documenting research projects. By large in this context, we mean a file larger than 50 MB. GitHub for instance warns users when pushing a file larger than 50 MB and does not accept files larger than 100 MB. Video files will often be larger than 100 MB. Fortunately, a free (MIT-licensed) and open-source extension to Git called Git Large File Storage (or Git LFS) can be used to alleviate this problem. It works around Git's size limitations by uploading large files to a separate storage area while tracking only metadata about these large files inside Git.

Using Git on the command line can be demanding. In our experience, graphical user interfaces (GUIs) that promise a more intuitive interaction style with Git often do not live up to expectations. Instead, we recommend a web interface.

2.2.2 GitLab

The web interface we use for Git is GitLab. GitLab started out as a GitHub clone, and became popular very quickly because it is available as a freely licensed community edition that includes source code needed for running a GitLab instance on premises.

Other web interfaces for Git such as Gogs and its derivatives were ruled out early on because they do not offer crucial enterprise features such as single sign-on (SSO) via LDAP or SAML2. Rolling out our source code hosting facility university-wide is part of the Conquaire project goals, so we needed to integrate with the Shibboleth-based identity management system of Bielefeld University. GitLab's SAML2 authentication method does just that.

GitLab proved to be an excellent choice because the makers of GitLab added the right features as our project progressed. For example, GitLab CI evolved from a simple continuous integration tool to a very powerful one, culminating in Auto DevOps, a feature set that provides a range of quality checks for software source code – not unlike what Conquaire provides for research data. However, Auto DevOps arrived only towards the end of the Conquaire project, so it did not influence our design decisions.

2.3 Conquaire Continuous Quality Control Infrastructure

2.3.1 Overview

A part of the Conquaire project was the development of automated data quality tests. The quality checks are integrated into the *GitLab* platform from the University of Bielefeld. The checks are written in *Python 3.6* and use the *lxml* package[3] for parsing XML files as the only external requirement. The pipeline of the quality check is shown in Figure 2.1 below. All steps are described in sections below.

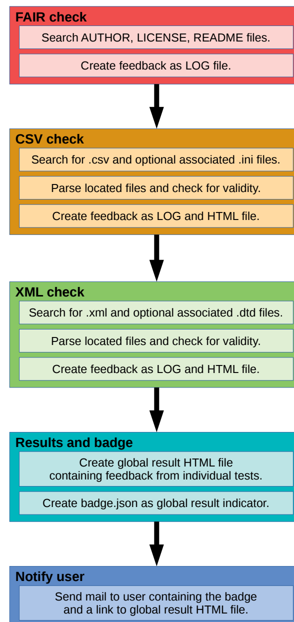


Figure 2.1: Workflow of Conquaire Quality Check.

By adding a preconfigured YAML file (in this case: *.gitlab-ci.yml*) to a repository on the GitLab instance, the checks are automatically executed via a continuous integration runner on the GitLab server.
The runner creates a docker container. As the docker image we use the python:3.6-alpine image because it is lightweight and only contains an installed version of Python 3.6. In addition to that, we install the *lxml* package and a *SMTP*[4]

[3] `https://lxml.de/`
[4] `https://wiki.debian.org/sSMTP`

instance to notify the user about the results from a check. The user is informed via email about the result of applying the test. The mail contains information about the repository and a URL to a HTML site containing the detailed feedback which can be rendered by any browser. The mail also shows the user the overall test result which is displayed as a badge icon. The same icon is displayed in PUB if the user decides to create a data publication.

2.3.2 Example of pre-configured YAML file

The pre-configured file has to be stored in the root folder of the repository. For each commit to the repository, it is automatically executed by the CI runner and performs the Conquaire quality checks for the given repository. The user only has to change the value of the `-d` parameter as it represents the local path to the data inside the repository. In the given example, a folder named *data* inside the repository contains the files which should be tested.

```
quality-check:
  # Use smallest docker python image.
  image: python:3.6-alpine
  before_script:
    # Create temporary mail configuration files.
    - mkdir /etc/ssmtp
    - echo "root=${GITLAB_USER_EMAIL}" > /etc/ssmtp/ssmtp.conf
    - echo "mailhub=conquaire.uni-bielefeld.de" >>
        /etc/ssmtp/ssmtp.conf
    - echo "hostname=gitlab-runner.conquaire.uni-bielefeld.de"
        >> /etc/ssmtp/ssmtp.conf
    # Install lxml and ssmtp package for sending feedback mail.
    - apk add py3-lxml ssmtp
  script:
    # Execute quality checking pipeline.
    - /usr/bin/python3 /opt/conquaire/quality_checks/src/main.py
        -f /var/www/html/feedback/
        -l "https://conquaire.uni-bielefeld.de/feedback/"
        -r "$(pwd)"
        -d "data"
        -gn "${GITLAB_USER_NAME}"
        -ge "${GITLAB_USER_EMAIL}"
        -gu "${CI_PROJECT_URL}"
        -gp "${CI_PROJECT_PATH}"
        -gs "${CI_COMMIT_SHA}"
  # Choose docker CI runner on gitlab server.
  tags:
    - dockerexec
```

The whole pipeline is executed in a docker container and makes use of continuous integration variables provided by GitLab. They are automatically filled with the information from the users GitLab profile.

2.3.3 Quality checks

The Conquaire Quality Check pipeline involves a variety of tests that are automatically performed on the Git repository. Each time a commit occurs, the GitLab CI runner calls our pipeline, and several scripts are executed to guarantee that the provided data is in the best possible state. The three main checks that are implemented are the FAIR check, the CSV check, and the XML check. The pipeline is designed to be very modular and flexible to make it as easy as possible to extend it with further checks, i.e., for additional file types.

Every check begins with searching the repository and generating a list of every file with the specific type using the bash `find` command. For each file that was found, the corresponding test script is called to perform the actual checks and generate a log file with errors and warnings that were observed. The details of the three specific checks are described below. In the end, an overall feedback file is created, showing the results of the checks with links to the log files, making it possible to look into the data and correct it if necessary. The contributor is informed about the results of the pipeline via email.

FAIR check

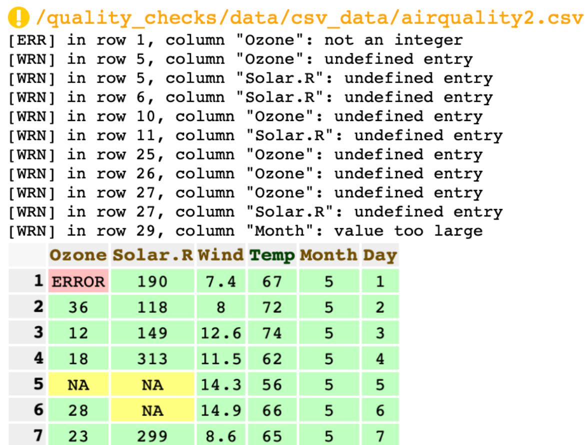
In our adaptive implementation of the FAIR metrics[5], we check if the three necessary files exist in the repository: the AUTHORS, LICENSE, and README files.
The files have to be placed in the root directory of the repository to fulfill the test condition. The files have to have either no extension, plain text (`.txt`) or markdown (`.md`).
We suggest to save the files as markdown files. The markdown file type is used as a standard in GitLab and many other websites because it has an easy to learn syntax and can be displayed in a web browser.

The AUTHORS file should contain a list of all the contributors and their emails for the possibility to contact them. The LICENSE file should describe how the data can be further used and distributed by other researchers, either by declaring one of the common licenses or providing their own. The README file should contain every other information that is related to the data and necessary or helpful to understand the research that was done, e.g., a description of the data or the experiment to obtain it.

[5] `http://fairmetrics.org/`

CSV check

In the CSV file format (`.csv`), data is organized as a table with comma separated values. The first step in the CSV check is to test whether the file can be opened and the table is well-formed, i.e., it has a header and a consistent number of rows and columns.
The researcher can provide an additional format declaration file (`.ini`) with his own specifications of the data, e.g., the type of the column and the expected range of the values. The quality check reports a warning if a required entry is missing or a value is out of range or has a wrong type, e.g., a non-numeric value in a numeric column.



	Ozone	Solar.R	Wind	Temp	Month	Day
1	ERROR	190	7.4	67	5	1
2	36	118	8	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6
7	23	299	8.6	65	5	7

Figure 2.2: Example result of the CSV check.

The log file lists all the errors and warnings that were found, and the row and column in which they occurred. In addition to that, the corresponding cell is marked in the table, allowing to conveniently find problematic entries, as seen in the example in Figure 2.2.

XML check

In the XML file format (`.xml`), data is organized as a tree structure using tag-based markup language. The first step in the XML check is to test whether the file can be opened and the document is well-formed, i.e., the syntax of the markup language tags is correct.
The researcher can provide an additional doctype definition file (`.dtd`) with his own specifications of the data, e.g., the required attributes and some restrictions to the values. The quality check reports an error if there is a mismatch of opening and closing tags, and a warning if the specifications are not fulfilled,

e.g., a value is missing.

```
/quality_checks/data/xml_data/sample.xml
[WRN] in line 8: Element target content does not follow the DTD
[WRN] in line 11: No declaration for element tes
1  <?xml version="1.0" encoding="UTF8" ?>
2  <node_description>
3      <target id="windows 32bit">
4          <graphics>nvidia_970</graphics>
5          <power_plug_type>energenie_eu</power_plug_type>
6          <test>unit test</test>
7      </target>
8      <target id="windows 64bit">
9         <graphics>nvidia_870</graphics>
10        <power_plug_type>energenie_eu</power_plug_type>
11        <tes>performance test</tes>
12     </target>
13 </node_description>
```

Figure 2.3: Example result of the XML check.

The log file lists all the errors and warnings that were found, as well as the line in which they occurred. In addition to that, the corresponding line is marked in the document, allowing to conveniently find problematic entries, as seen in the example in Figure 2.3.

After successful execution, the Conquaire quality check pipeline produces an overall result HTML file which contains visual feedback of all individual tests and links to the resulting log and optional HTML files. An example is provided in Figure 2.4. The feedback shows one of three different colors and badges. A green badge represents a successful test result, i.e., the data is valid. A yellow badge indicates well-formed data and the log files can contain some warnings. A red badge indicates not well-formed data or missing FAIR files. The user should check the log files and fix the errors before submitting a data publication. The URL of the overall result is provided to the user via email. This mail is sent automatically after every commit. In addition to that, an overall badge icon is created. This badge is equivalent to the badge of the worst individual check result. This badge is displayed in PUB[6] if the user decides to create a data publication from the repository. The badge is equal to one of the three different symbols shown in Figure 2.4.
Thus, the Conquaire quality checks are designed to help the researchers to clean up data, remove inconsistencies and make it fit for use by others.

Fulfilling the FAIR metrics is highly important for the reproducibility of the data as they are necessary to provide other researchers the information and legal

[6] https://pub.uni-bielefeld.de/

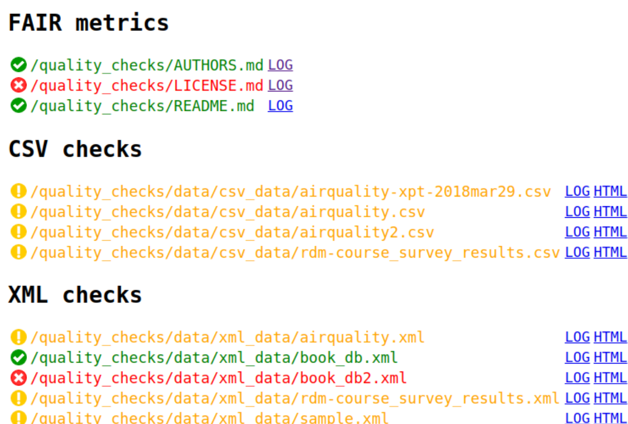


Figure 2.4: Example result of the overall result.html.

basis to use the data for their consecutive works. The file type specific checks help finding and fixing errors before releasing the data to the public. This is fundamental for reproducibility as only valid data can be used to recreate the experiment results.

2.4 Summary

In this chapter, we have briefly described how the Conquaire infrastructure implemented at Bielefeld University applies continuous integration principles to support reproduction of analytical results but also ensure high quality and valid data. The basis of the infrastructure is a distributed version control system (DVCS) that stores different versions of computational artifacts. In this chapter, we have argued why we have selected Git as a basis to implement this DVCS at Bielefeld University and why we have selected GitLab as a graphical and web-based user interface to access Git and foster collaboration. We have further described how the Conquaire infrastructure automatically runs a number of quality checks on the data once a new commit has been performed. The user merely has to add a YAML file to the root directory of the repository. This YAML file will trigger the GitLab CI runner to execute a number of standard tests on CSV and XML files to check whether the data is consistent, syntactically well-formed and complies with schema declarations. The results of each test are written into a log file and used to generate a report that is published as a website on a web server. A link to this report is sent to the user committing the data for inspection of the results of the tests, giving access to the detailed logs. Building on principles of gamification and to create incentives for committing ready-to-use-data, the Conquaire systems assigns badges to the data corresponding to

whether they passed the tests or not and visualizes these badges in the reports generated and optionally on a PUB page where the data has been published.

During the Conquaire project, we have run a number of Git workshops with all case study partners, confirming our hypothesis that the subset of Git commands that is needed to commit data into the repository can be easily learned by our target population. On the basis of our experience, we can definitely recommend Git, GitLab and our architecture for continuous integration to implement an institutional infrastructure for hosting data and checking their quality as a basis to ensure reproducibility of research results.