Creation of LaTeX documents using a cloud-based pipeline

Marei Peischl, Marcel Krüger, Oliver Kopp

Abstract

Using web-based platforms for collaborative editing of LaTeX documents is common these days. These tools focus on writing documents and not on creation of templates or packages. Using build servers with automated pipelines is common within software development but can easily be adapted for TeX & friends.

This article will show how to get started using automated workflows on platforms like GitHub, Forgejo, or GitLab for document authors as well as TeX developers.

1 Introduction

Everything has become available in or is moving towards the cloud. LaTeX is already there for about 10 years and today it's quite common to use a web editor for collaboration and local compilation became the "nerdy" way. But there also is a third variant to compile documents which can be used also to improve package development and in general the stability of LaTeX: adapting DevOps methods, like continuous integration and delivery (CI/CD) using automated workflows.

As a very rough working definition, let us consider a CI/CD workflow as a number of steps to compile a TeX document to PDF on some kind of online service that has access to the source files.

2 Why continuous integration?

Having an established workflow usually makes people avoid thinking about changing anything. So there have to be reasons why it might be worth reading this article, let alone integrating the mechanisms into projects.

Early adopters of continuous integration techniques in the TeX ecosystems tried to follow the current state of open source development and open doors for contributors in the development process. For example, the LaTeX Project is currently welcoming a lot of user interaction via their GitHub projects [16] and also takes contributions from which the whole (La)TeX community will profit.

But the advantages of these methods extend beyond that. We will focus on some cherry-picked aspects as the remainder could probably be an article by itself.

2.1 Works for me?!

Sometimes, I can successfully compile my document on my machine, but my supervisor can't on theirs. There are many reasons why a TeX compilation may succeed on one system and fail on another. Running some external continuous integration pipeline will not only illustrate the necessary steps to go from source to the full PDF, it will also help understand if problems are machine-specific or general.

2.2 Compatibility and regression testing

With CI/CD, it is possible to run workflows on multiple TeX distributions or versions. This can be used to test if there are issues with some package update before updating a machine to, e. g., the latest MiKTeX updates where downgrading may be complicated.[1]

Additionally, one can also use it to check backwards compatibility, e. g., if one collaborator is using a Debian stable version which has only some outdated version. As mentioned, the LaTeX Project Team is already using these techniques and even provides functionality for regression testing within their build system, l3build [17, 19].

Using CI/CD as a package developer enables a general interface to be used for regression testing. This allows for avoiding some of the bugs which otherwise would be published and found by a user. Furthermore, it can be used to avoid inconsistent structures, e. g., one of the authors recently found that numerous packages and files within TeX Live do not have a proper version number set within the code.

3 Structure of this tutorial

The idea is to introduce readers to the basics of setting up automated workflows on GitHub, Forgejo, and GitLab. This tutorial mainly addresses two groups of users:

1. authors focusing on typesetting actual content, including those collaborating on a document, and
2. package or template developers who provide their work to be used by the first group.

The second group obviously can also use workflows of the first, e. g., for typesetting documentation. So developers usually use an extended version of the setup provided for authors.

As all platforms covered by this article are related to the git version control system, we expect the project to be some kind of git repository. In case the reader does not yet use git, there is a little bit of information attached to this tutorial. Using that, it

[1] That's another issue to address ... but a different story.

doi.org/10.47397/tb/45-2/tb140peischl-pipeline

would be possible to use git without even noticing as it is attached to the autosave function of an editor.

Because the LaTeX project is using GitHub, we are going to start with a detailed explanation of GitHub Actions and create a matching setup on the other platforms afterwards. All workflows are available for customization in the template repositories listed in Table 1 near the end of the article. Within this article the listings are marked by icons to not confuse readers as the platform is switched multiple times to show the differences. is used to indicate the listing belongs to GitHub Actions, whereas marks the GitLab CI variant.

4 Compiling a document in a CI pipeline

4.1 First steps with GitHub Actions

To get started, we need a git repository somehow hosted on GitHub. It does not have to be public.

Thinking of the tasks needed to compile a LaTeX document in any "blank slate" environment, we have to do the following:

1. Prepare the environment and install a LaTeX distribution.
2. Run LaTeX on the document.

GitHub provides pre-configured actions which are able to combine multiple steps, e.g., the "latex-action" action [20]; this starts another container inside the action container to run `latexmk`. However, these and other actions typically limit the configuration options to simplify the interface. We are going to elaborate on two variants: Using a container image with a full TeX Live (this section) and a minimal installation (see Section 6).

The configuration for an action is done by creating a yaml file within the repositories' subdirectory `.github/workflows/hello-world.yml`. The filename may be chosen at will. The following code snippet shows a minimal configuration:

```
name: Hello World Action
on: [push]
jobs:
  action-test:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Hello world 🦁🦆"
```

name: of the workflow. This is important if a project contains multiple workflows.

on: This directive indicates conditions under which the workflow will be started. In this configuration, the jobs will be started on any push, i.e., whenever the repository on the server receives an update.

Besides attaching the trigger to some user action it is possible to use time-based settings. For all options it is worth having a look at the configuration manual [5]. The default settings differ for all platforms, and may be specific for a single instance.

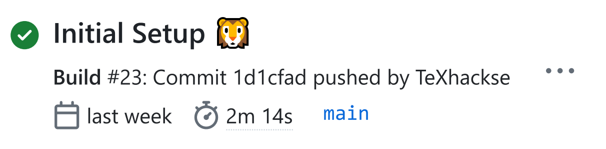
jobs: contains a list of jobs to be run one after another. For example, it is quite common to have one job for compiling a document and another one to make the PDF available. In this example, there is only one job called "action-test".

runs-on: This value corresponds to a runner setup. Runners are the systems that actually execute the jobs defined in a workflow. It does not have to be the same server as the one where the repository is hosted. In this example, "ubuntu-latest" indicates running on one of the provided runners by GitHub which is based on Ubuntu. It includes NodeJS and some tooling to simplify the work using predefined actions. A full list of the readily available runners and detailed description of the images can be found at [4].

steps: This is what the workflow should actually do. As one can see it is possible to directly enter (ba)sh code in there, and use UTF-8. This example merely echoes a string to stdout and therefore should run without any issues.

On GitHub, actions are automatically enabled for new repositories. When combined with an available runner as we do here, it is enough to add a yaml configuration file to the repository to see the effect. After pushing that configuration, the pipeline will start running on all subsequent pushes.

The current status of an action, i.e., what step it is currently running or if it has already finished, can always be checked by having a look at ⟨*repository url*⟩`/actions`; e.g., for the first of our demo repositories, this can be found at `github.com/islandoftex/tug2024-workflow-github/actions`. It looks like this:



Initial Setup 🦁

Build #23: Commit 1d1cfad pushed by TeXhackse

last week 2m 14s main

The pipeline ran successfully but did not do anything except create some shell output. Hence, we can move on to the next step: building a LaTeX document.

4.1.1 GitHub Actions using LaTeX

Actions are fundamentally based on isolated containers of software, which are run using Docker. Luckily, some of us live on the Island of TEX (IoT) and maintain images we can make use of here.[2] The setup of the images was described within [12].

The first part of the workflow will stay the same for the moment. Changes apply only after **runs-on**:

```
runs-on: ubuntu-latest
container:
  image: texlive/texlive:latest
steps:
  - name: Checkout repository
    uses: actions/checkout@v4
  - name: Run latexmk
    run: "latexmk --lualatex"
```

container: Choose the IoT "texlive-latest" image which provides a full TEX Live and some tools [11].

steps: The first step looks different from the one we had before. It's given a name, "Checkout repository", which is helpful to simplify debugging as GitHub will tell us which step failed.

uses: is a reference to another action and GitHub's way to reference pre-configured pipelines encapsulating more complex tasks. In the example, `checkout@v4` refers to a separate repository [6]. This action takes care of the authentication and some internals, so we do not have to deal with those details.

A crucial point is that only after this action are the following steps begun to be executed, within the root directory of our repository.

second step: The second step is also given a name ("Run latexmk"), and then uses the same **run:** directive as our first example, but this time we run `latexmk` [2] with the option `--lualatex` (to use the LuaLATEX engine, as you might guess). By default, `latexmk` will operate on all `*.tex` files within the root directory. So we do not even have to depend on the file name, as long as we store any files to be included within subdirectories.

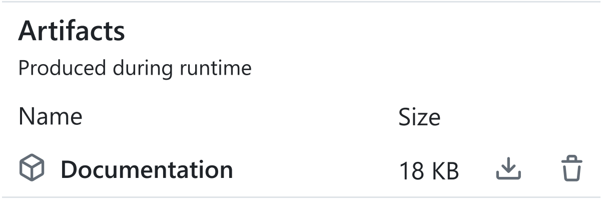
4.1.2 Where is the PDF?

If the pipeline succeeds, there will be a green checkmark, but we will fail to find the PDF somewhere. This is because GitHub cannot know which (output) files the user actually wants to see or download. Such files are called "artifacts", so now we will add another step to the action to keep the PDF and upload it to GitHub as a so-called "artifact":

[2] Special thanks to the other islanders at that point!

```
- name: Archive documentation
  uses: actions/upload-artifact@v4
  with:
  name: Documentation
    path: "./*.pdf"
```

After another (successful) run of the pipeline, it is possible to access the PDF wrapped within a `*.zip` at the run's status page. Clicking on the run on the actions overview page leads there:


Artifacts
Produced during runtime

Name	Size
Documentation	18 KB

Sadly, GitHub currently does not provide an option for individual files without a zip. Also it is not possible to have a static link pointing to the latest artifacts. (Other platforms such as GitLab do provide that feature by default.) To resolve this and make the PDF easier to access on GitHub, it is required to use additional actions. The most common way to do it is publishing the PDF to an orphan branch. Usually this mechanism is used to create web pages and is called "github-pages".

To be able to write in the repository, it is necessary to adjust the permissions. GitHub provides an interface within the yaml configuration:

```
permissions:
  contents: write
```

This has to be added to the job which should upload the PDF. Additionally, most actions which can be used for uploading artifacts to separate branches request a directory instead of a file. Consequently, all files have to be moved to a separate directory to be used with these actions.

```
- name: move pdf
   run: mkdir -p build && mv *.pdf build/.
- uses: crazy-max/ghaction-github-pages@v4
  with:
    target_branch: pdf-output
    build_dir: build
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN
     ↪  }}
```

The GitHub token on the last line is required for authentication. Otherwise, the run might be successful, but no PDF will appear on the branch.

4.2 Differences with Forgejo Actions

Forgejo [3] is another code platform which has the aim to be more open than GitHub. It can be self-hosted and provides mechanisms similar to GitHub Actions. They will never act in exactly the same way as GitHub's mechanisms, but are almost compatible.

By default, Forgejo first searches `.forgejo/workflows/` for configuration files. If none are found it will fall back to look inside the `.github/` directory. The only issue you would probably face from simply reusing a GitHub setup is that there are no hosted runners on most instances. Even if there are runners available, they will probably be different from GitHub's.

However, if you can configure a runner yourself and use the same labels as the GitHub runner uses (described in [1]), it is possible to use the same workflow configurations on both. The provided demo repositories on `codeberg.org`, which is a Forgejo instance, also explain how the runners used there are configured.

4.3 Compiling a document using GitLab CI

GitLab CI is not too different from the previous Actions setup. It is even a bit easier to set up continuous integration here as some steps are run automatically. For example, it is not necessary to check out the repository, because this is done by default.

The configuration file has to be placed within the repository's root directory and given the name `.gitlab-ci.yml`. Here's our same example:

```
runlatex:
  image: registry.gitlab.com/islandoftex/
    ↪  images/texlive:latest
  script:
    - 'latexmk --lualatex'
  artifacts:
    paths:
    - "./*.pdf"
```

Here the steps are simply a list of commands run after each other. Our list just contains one call to `latexmk`.

In contrast to GitHub, GitLab provides an API which allows creating static links to the artifacts. Thus, the `README.md` of the demo repositories there include a link of the following structure:

⟨*repository url*⟩`/-/jobs/artifacts/`⟨*branch*⟩`/browse?job=`⟨*name of job*⟩

This will list all artifacts attached to the job. For instance, using the example configuration on one of the demo projects results in this sample url:

`gitlab.com/islandoftex/texmf/tug2024-workflow-document-gitlab/-/jobs/artifacts/main/browse?job=check:[latest,lualatex]`

5 Testing with multiple versions or compilers

We promised that one can extend these setups to test using multiple TeX versions or engines. Different engines are straightforward to include by running additional steps with different commands. Alternatively, it is possible to use different engines in separate actions or even workflow files. There is no general way to structure those, as the choice always depends on whether to always run everything or save resources via conditional or sequential execution.

To use different versions of TeX Live, the Island of TeX provides historic images of all but the latest TeX Live release (which is not yet historic).

```
test-on-IoT-texlive:
  runs-on: ubuntu-22.04
  strategy:
    matrix:
      image: ["TL2022-historic",
      ↪  "TL2023-historic", "latest"]
  name: "Test on ${{ matrix.image }}"
  container:
    image: texlive/texlive:${{ matrix.image
    ↪  }}
  steps:
```

strategy: this is used to create a loop over elements. In this case, a variable called `matrix` containing a list of images is used and the content can then be accessed within other parts of the file using `${{ matrix.image }}`. Thus, the first run will use TL 2022, continue on 2023 and finally run on the latest release. A full list of the provided images can be found at [9].

Similarly, GitLab also has a matrix feature. The following example illustrates how to run multiple engines, each on multiple TeX Live releases:

```
check:
  image: registry.gitlab.com/islandoftex/
    ↪  images/texlive:$TEXLIVE_VERSION
[ ... contains script + artifacts ... ]
  parallel:
    matrix:
      - TEXLIVE_VERSION: ['TL2022-historic',
      ↪  'TL2023-historic', 'latest']
        TEX_ENGINE: ['pdflatex', 'xelatex',
        ↪  'lualatex']
```

Here the variable is more like a shell variable but can be used the same way.

6 Minimize the build container

The previous examples used a full TeX Live installation to have the most convenient setup.[3] The IoT images even ship all dependencies of additional tools such as arara or minted, which enables that all tools in TeX Live will work out of the box.

Still, sometimes you will not need all the bells and whistles and there are advantages to smaller build containers. For instance, downloading more than one gigabyte can take quite some time. Or you may only have limited disk space available on the runner server.

Side remark: If you are looking to reduce not the images themselves but the compile time because you are building many documents, have a look at last year's IoT article [11].

Back to minimizing the build environment: the most annoying part here might be discovering which packages are actually needed to compile ...

So the Island of TeX proudly presents: DEPP – The DEPendency Printer for TeX Live [7].[4]

As this article focuses on pipelines we won't go into detail on the tool itself but instead show how DEPP produces a file listing all TeX Live packages necessary for the build. For the example projects these look like:

```
# Proudly generated by the Island of TeX's...
blindtext
cm
⋮
```

6.1 GitHub

On GitHub there are multiple actions available to install TeX Live as a part of the workflow [13, 18]. Using those, it is possible to minimize the container, which will also reduce the build time.

```
- name: Install TeX Live
    uses: zauguin/install-texlive@v3
    with:
      package_file: .github/tl_packages
```

This snippet can be used within **steps:** and makes the **container:** directive obsolete, so it should be removed. The **package_file:** is the path to the DEPP output or a manually created list of packages.

[3] Not exactly full: the images we used include neither the documentation nor the source trees of TeX Live.

[4] If you are German, do not be surprised if this tool is more clever than its name suggests.

6.2 GitLab

On GitLab the simplified syntax makes running a minimized TeX Live a bit more complex. The DEPP repository [7] luckily provides a shell script to install a TeX Live based on the package file. This can be used in the pipelines to modify the container.

```
image: registry.gitlab.com/islandoftex...
before_script:
  - minimal_tl_setup.sh "tl_packages"
```

Another option would be to provide a container image which already contains the necessary packages. If a self-hosted runner is used, this is usually the best option as caching can be configured to control when the container is rebuilt or updated.

7 Pipelines for package developers

As promised, the advantages of using automated pipelines are even more significant when used within the development process of packages or templates. In this case, we expect the repository to contain a package and some kind of `l3build` configuration.[5]

7.1 GitHub

The `latex` step (calling `latexmk` in the examples) is replaced by

```
- name: Run l3build
  run: l3build check --show-log-on-error -q
  ↪ -H
```

This will automatically run all test files according to the `l3build` setup. As this is within the context of running a package's test suite, the artifacts are totally different. Now, we are not interested in a PDF but rather the test output. One of the present authors has created another action to take care of this [14]:

```
- name: Archive failed test output
  if: ${{ always() }}
  uses: zauguin/l3build-failure-artifacts@v1
  with:
    name: testfiles
    retention-days: 3
```

Apart from uploading the artifacts, this configuration illustrates another important point: In contrast to the PDF of a document, where we are only interested in the successful output, for test suites we are less interested in success than the failure output. To let GitHub know that we are in fact interested in

[5] The setup itself also works for other tools, but that is out of scope for this article.

these artifacts, `if: ${{ always() }}` has been added. This will also force the step to run even if the previous step failed.

7.2 GitLab

Again, GitLab directly supports the artifact upload without loading external extensions.

```
  - l3build check --show-log-on-error -q -H
artifacts:
  when: on_failure
  paths:
    - ./build/test/*.diff
```

This simplifies the standard setup but is less flexible. For example if you want to use different artifacts for success and failure, it is required to do that within two separate jobs within GitLab.

8 Running locally

During the tutorial at the TUG'24 conference there was a short demonstration of running the pipelines locally. This might be helpful for debugging as one can control the steps manually or to ensure the local setup matches the one on the server.

These setups usually use Docker. For the platforms which use actions there is a tool called "act" to simplify that process using the Docker API.

GitLab extends this with the option to simply have a GitLab runner installed locally. This provides the option of running

```
gitlab-runner exec
```

within a local repository containing some Gitlab CI configuration.

9 This is Continuous Development

Of course every configuration shown here is only an example and can be extended depending on the project's requirements. It is possible to run arbitrary commands, which might be necessary especially for complex setups.

For example, when creating magazines, it is possible to continuously create a print and an online version, excerpts of single articles as well as HTML/EPUB output, while the editors only have to wait for the compilation of the article they are actively working on.

The same goes for study material, where we can have rendered versions including solutions or not, or documents which share links between each other and therefore require many runs.

Integrating these structures into more advanced git usage like a useful branching concept can also help improve collaboration or simplify the contribution process within open source projects. It will reduce frustration for maintainers as some issues do not have to be checked manually.

10 Conclusion and call for action & feedback

We've explained the use of GitHub, Forgejo, and GitLab for compiling LaTeX documents and packages. We showed how different releases of TeX Live and even different compilers can be used to simplify testing across platforms. Also we took care of being able to access the PDF or the testing results in some way. To increase the stability of all parts of TeX development we hope this will help with more testing of packages and even less waste of time while compiling complex setups.

If you maintain any packages, it would be great if you could try setting up a test to check it against the latest release of TeX Live or even current TL development. This can also be a preparation for next years TeX Live pretest, as the Island of TeX is creating a Docker image for the pretests. If you are planning or attempting to do that and face any issues, we will try to help.

We would love to maintain this as a tutorial to simplify the use of automation for users of TeX & friends. So if we've left open questions, we would love to hear about it and will try to improve this tutorial as well as the examples. Table 1 summarizes the related repositories.

Contributions to this project are very welcome!

References

[1] Codeberg doc contributors. `docs.codeberg.org/ci/actions/`

[2] J. Collins. latexmk — fully automated LaTeX document generation. `ctan.org/pkg/latexmk`

[3] Forgejo. Forgejo repository. `codeberg.org/forgejo/forgejo`

[4] GitHub. Choosing GitHub-hosted runners. `docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#choosing-github-hosted-runners`

[5] GitHub. Workflow syntax for github actions. `docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions`

[6] GitHub and contributors. Checkout action. `github.com/actions/checkout/`

[7] Island of TeX. DEPP — dependency printer for TeX Live. `gitlab.com/islandoftex/texmf/depp`

Table 1: Template repositories published with this article. The naming scheme is structured as ⟨*ci-type*⟩_⟨*task*⟩, adding "_minimal" if the example is not using a pre-packaged Docker image but includes methods to install packages based on a dependency file as described in section 6.

All variants listed here have been prepared for at least the three platforms mentioned here (GitHub, GitLab, ForgeJo).

As the urls are quite long, we have published the list including links within the paper's repository `tug.org/l/peischl-cicd2024`.

Name	Platforms	Document	l3build	Testing	IoT image
latex		✓			✓
latex_minimal		✓			
latex_testing		✓		✓	✓
latex_testing_minimal		✓		✓	
l3build			✓	✓	✓
l3build_minimal			✓	✓	

[8] Island of TEX. GitHub workflow template for LATEX packages. `github.com/islandoftex/tug2024-workflow-github`

[9] Island of TEX. GitLab repository: TEX Live Docker image. `gitlab.com/islandoftex/images/texlive`

[10] Island of TEX. GitLab workflow template for LATEX documents. `gitlab.com/islandoftex/texmf/tug2024-workflow-document-gitlab`

[11] Island of TEX. Living in containers — on TEX Live (and ConTEXt) in a Docker setting. *TUGboat* 44(2):249–252, 2023. `doi.org/10.47397/tb/44-2/tb137island-docker`

[12] Island of TEX. Providing Docker images for TEX Live and ConTEXt. *TUGboat* 40(3):231, 2019. `tug.org/TUGboat/tb40-3/tb126island-docker.pdf`

[13] M. Krüger. zauguin/install-texlive repository. `github.com/zauguin/install-texlive`

[14] M. Krüger. zauguin/l3build-failure-artifacts repository. `github.com/zauguin/l3build-failure-artifacts`

[15] M. Peischl, M. Krüger, O. Kopp. Source to this paper, and links to additional resources. `tug.org/l/peischl-cicd2024`

[16] LATEX Project. GitHub organization. `github.com/latex3/`

[17] LATEX Project. l3build — a testing and building system for (LA)TEX. `ctan.org/pkg/l3build`

[18] teatimeguest/setup-texlive-action repository. `github.com/teatimeguest/setup-texlive-action`

[19] J. Wright. `l3build`: The beginner's guide. *TUGboat* 43(1):40–43, 2022. `doi.org/10.47397/tb/43-1/tb133wright-l3build`

[20] C. Xu. latex-action. GitHub action to compile LATEX documents. `github.com/xu-cheng/latex-action/`

◇ Marei Peischl
Gneisenaustr. 18
20253 Hamburg
Germany
`marei (at) peitex dot de`
`https://peitex.de`

◇ Marcel Krüger
Hamburg, Germany

◇ Oliver Kopp
Sindelfingen, Germany
ORCID 0000-0001-6962-4290