prodockit
User Guide

A docs-as-code workflow for professional and academic documentation, built on Zensical and Markdown.

Release: 1.13.0

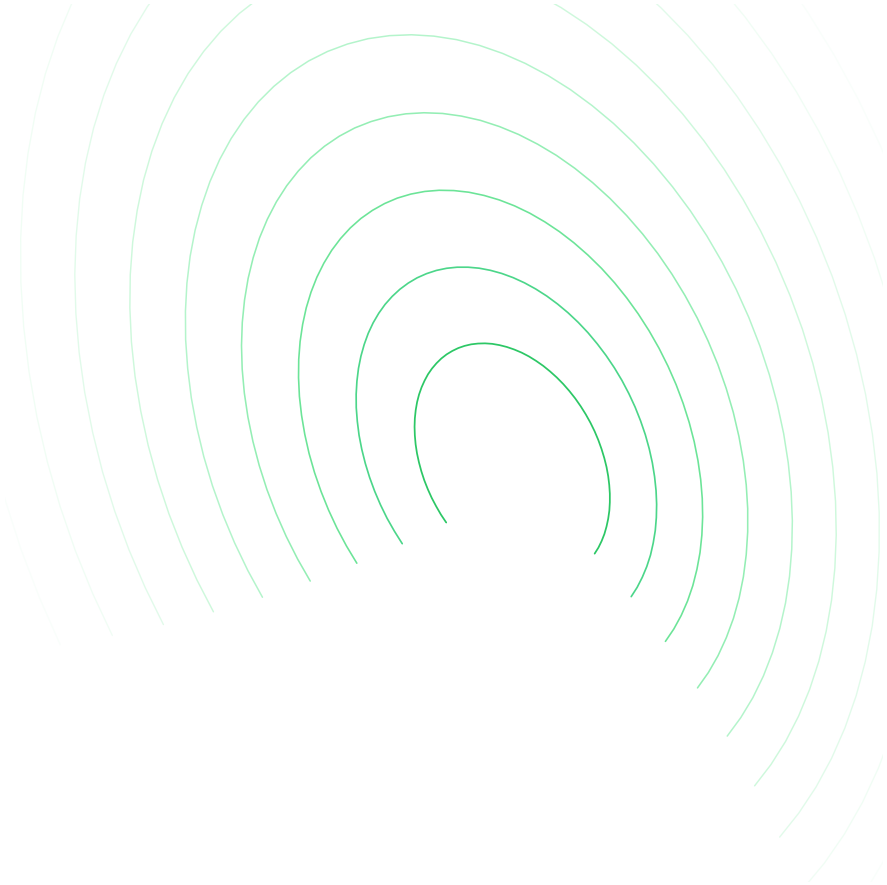


Table of Contents

1. About this guide

A docs-as-code workflow makes it easy to publish documentation on a static website. Markdown is a markup language you use to write documentation in text files, which you then store in a Git repository. Hosted Git services such as GitLab or GitHub, together with tooling such as Visual Studio Code, make it easy to maintain and host documentation.

Adopting a docs-as-code workflow transforms documentation from a chore into an engineering process. By treating your written content with the same rigour as code, you enable a collaborative approach to documentation.

This site is the full setup, authoring, customisation, and testing guide for prodockit-template and other Zensical projects built on the prodockit package. It's hosted independently of any individual fork of those templates, so it can be kept current without every existing fork being stuck with whatever it looked like the day it was forked.

1.1 The docs-as-code philosophy

Docs-as-code means using the same tools and workflows for documentation as you do for software development. This creates a unified environment where writers and developers use the same tools and development workflow.

`Markdown as the Source of Truth`

> Markdown is a lightweight markup language that's simple to read in its raw form and consistently renders on the web. You write your documentation in it, then convert that documentation into HTML for web publishing.

`Version control via Git`

> Storing files in a Git repository (such as GitHub, GitLab, or Bitbucket) enables the tracking of all changes to the documentation. There will be a complete history of "who changed what and why," making it easy to undo errors and audit changes.

`Collaborative Reviews`

> Instead of emailing Word docs back and forth, teams use Pull Requests (PRs) or Merge Requests (MRs). This enables peer reviews, automated linting, and transparent discussions before any content goes live.

1.2 The docs-as-code stack

To move from using a word processor or simple website to a scalable and effective documentation tool set needs a stack of tools, as shown in the

diagram below. The stack is made up of three layers: the authoring tool, the docs-as-code builder, and the code repository and management system.

`Authoring tool`

Effective documentation depends on tools that help with content editing and automate quality checks, including spelling, grammar, style and formatting consistency. While there are numerous text editors available, VS Code stands out as a favoured option due to its extensive ecosystem of extensions. By using Visual Studio Code, writers can take advantage of a variety of extensions that offer real-time quality assurance. Some of the extensions available for VS Code include spell checkers, grammar checkers, and linters that enforce style guides. These tools help ensure that the documentation is clear, consistent, and professional. We are suggest using Python, Zensical Studio, Even better TOML and LTeX+.

`Docs-as-code builder`

Markdown files can serve as a foundation for a static website. However, they often need enhanced formatting options through additional themes and styling. A docs-as-code builder then transforms the Markdown and supplementary instructions into HTML, applying a theme to generate a professional-looking website. Zensical is a fast and reliable docs-as-code builder that processes Markdown files and creates a static documentation website. It lets you view the website locally before publishing, so you can confirm the final output meets your quality standards. Additional extensions, such as the prodockit package, can be added to Zensical to provide extra features like heading numbering, a references page and a pdf document with an index.

`Code Repository and Management`

By connecting directly to a Git repository, this integrated environment establishes a secure, centralised vault that tracks the history of project files. It records each modification as a distinct commit, letting users audit changes or revert to earlier versions if errors arise. Before finalising any work, a pull request triggers a peer-review process, where collaborators comment on, test, and approve the updates. This workflow ensures that only vetted, high-quality content reaches final publication. GitLab and GitHub are popular platforms that provide these capabilities, along with additional features like issue tracking, project management, and continuous integration.

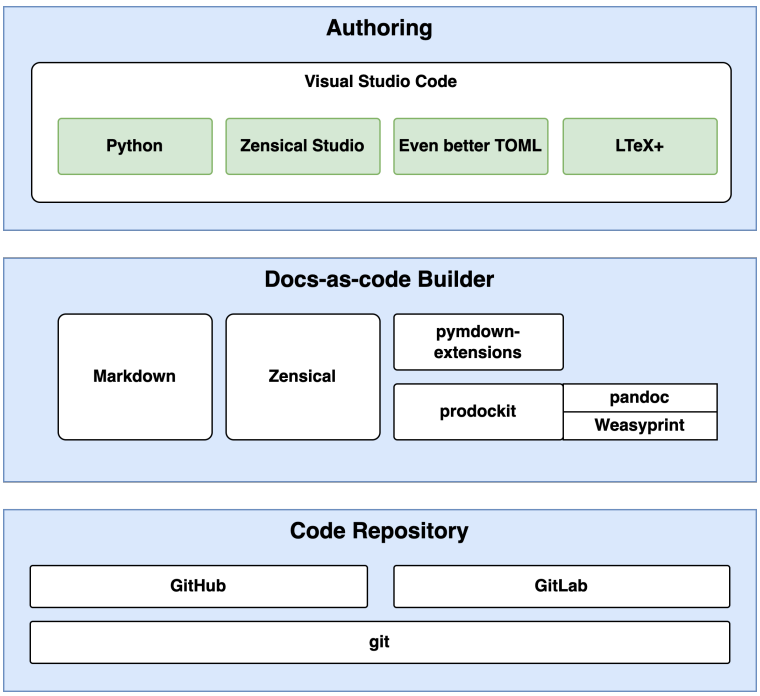


Figure 1.1. Docs as code stack

Why not LaTeX?

LaTeX is a typesetting system widely used in academia and for specialised industrial documentation that requires precise formatting. Although it is not built specifically for web publishing, external tools can convert LaTeX source files into HTML for use on static websites. Markdown is often preferred for general documentation in industry because it integrates more naturally with modern web-based development workflows.

1.3 Docs-as-code in production

As a student, you'll follow a simplified workflow, since you won't be handling documentation that spans thousands of pages and needs a large development team to maintain it. Nevertheless, understanding the approach used at scale can help you appreciate the value of the skills you'll develop. GitLab provides a video that outlines the entire process for their documentation and highlights the importance of the skills you gain through a docs-as-code methodology.

Introduction to using GitLab as a technical writing team Youtube video

Watch Video

1.4 Zensical for docs-as-code

Zensical provides the themes and tools necessary to draft professional documentation in Markdown with instant local previews. Once finalised, you can publish your site by uploading the files to a Git repository. From there, automated pipelines build and deploy the content into a live website.

Zensical, written for speed and reliability using the Rust programming language and Python, publishes documentation as a website.

We have provided an additional extension, the prodockit package, which adds extra features to Zensical enabling you to create a professional-looking documentation website, together with a PDF version of the report.

prodockit-template is a documentation template built on Zensical and prodockit, available for you to fork and clone into a project that you can use to write your own document or report.

> **Tip**
>
> Zensical processes these Markdown-formatted instructions into this site. To view the structure of this site by examining the Markdown files, go to this Git repository, linked at the top right of this page.

1.5 Guide structure

This guide continues by presenting the following sections, in the order you'll need them for the development of your documentation.

`Install tooling`

: The *Install tooling* section describes how to install the core prerequisite tools and create a fork of the document template. By the end of this section you will have an environment ready to develop your own professional or academic documentation. The instructions are available for macOS, Windows 11, and Ubuntu/Debian Linux.

`Start editing`

: The *Start editing* section describes how to edit the documentation and view the changes locally before publishing to a Git repository. It also describes how to synchronise your changes with the Git repository.

`Markdown basics`

: The *Markdown basics* section describes the principles of Markdown and how to use it to write your documentation.

`Zensical basics`

The *Zensical basics* section describes the principles of Zensical and how to use it to create and manage your documentation - the same general-purpose features available on any Zensical site.

`Customisation`

The *Customisation* section discusses how to configure prodockit-template to give it a different style and layout to meet your specific needs - features `macros.py` and the prodockit package add on top of Zensical (heading numbering, the references page pattern, Surrey/generic branding, and more) that don't exist in a stock Zensical project.

`Additional tooling`

The *Additional tooling* section describes additional tools you can install to help you create quality docs-as-code documents. They will take additional effort to install and configure.

`Shell commands`

The *Shell commands* section describes the shell commands used in this guide. It's intended as a reference for you to use when writing your own documentation.

Continue to the next section to get started with the installation of the core tools and creating a fork of prodockit-template.

2. Install tooling

This section takes you through the core installation steps for the tools needed to author your document as a static website and PDF file. The instructions are for macOS, Windows 11, and Linux (Ubuntu/Debian). If you are using a different operating system, please refer to the official documentation for that operating system.

The install and configuration starts with the setup of Visual Studio Code.

2.1 Install Visual Studio Code

Visual Studio Code (VS Code) is the editor we have chosen for developing the documentation using Zensical. You can use other editors, but the availability of many plugins in Visual Studio Code will help you edit your documentation more efficiently.

The steps below will help you install VS Code and some essential plugins to edit your documentation. If you have already installed VS Code, check through the steps so you have the plugins installed.

2.1.1 Install Visual Studio Code

Start with installing Visual Studio Code. Instructions for macOS, Windows 11, and Linux (Ubuntu/Debian) are below.

Install Visual Studio Code

macOS

1. Open the **Terminal** application.
2. You are likely to already have Homebrew installed, but if not, follow the instructions on brew.sh to install it. **Close and reopen your Terminal after installing it.** As the installer adds `brew` to your `PATH`, and a session that was already open won't pick that up.
3. Use the Homebrew package manager to install Visual Studio Code in your Terminal:

```
brew update
brew install --cask visual-studio-code
```

Windows

1. Download the VS Code User setup for Windows from the official website.
2. Run the installer, `VSCodeUserSetup-{version}.exe`. By default the User setup installs Visual Studio Code to your user profile directory. You can change the install location if you want to install it for all users.

Linux (Ubuntu)

1. Download the `.deb` package from the official website.
2. Open a terminal and navigate to the directory where you downloaded the `.deb` package.
3. Run the following command to install Visual Studio Code:

   ```
   sudo apt install ./<file>.deb
   ```

   Replace `<file>` with the name of the downloaded `.deb` file.

Further installation instructions are available on the Visual Studio Code website.

2.2 Install Git with Visual Studio Code

Git is a version control system that enables you to track changes to your code and collaborate with others. You will be using Git to manage your documentation website and push your changes to your **GitLab** or **GitHub** cloud repository.

Next, install the `git` command and configure it for Visual Studio Code. The instructions below are for use with both *GitLab* and *GitHub*.

2.2.1 Install and configure Git

Start by installing Git and configuring it for Visual Studio Code. The instructions below are for macOS, Windows 11, and Linux (Ubuntu/Debian).

1. As a start, you need to install the `git` command. Follow the instructions below to install or update `git` to the latest stable version.

Install Git

macOS

Use the Homebrew package manager to install or update `git` to the latest stable version:

```
brew install git
```

Windows

Open up a **PowerShell** Administrator window and install `git` using the command, or you can download and install the official git installer from git-scm.com.

```
winget install Git.Git
```

If you just require an updated version of `git`, you can run the following command in **PowerShell**:

```
winget upgrade Git.Git
```

Close down PowerShell and reopen it after installing or updating `git` to ensure that the new version is available in your `PATH`. Check the version of `git` installed by running the following command in **PowerShell**:

```
git --version
```

Linux (Ubuntu)

Open a terminal and run the following command to install or update `git` to the latest stable version:

```
sudo apt update
sudo apt install git
```

2. Before connecting to any cloud provider, open your terminal (Terminal on macOS/Debian, Git Bash or PowerShell on Windows 11) and set your global username. This is the identity stamped onto your commits.

```
git config --global user.name "Your Name"
```

Then set the email address to go with it. Make sure it's the same one you used to register for your GitLab or GitHub account.

```
git config --global user.email "your.email@example.com"
```

> **Already use Git for other projects?**
>
> `--global` applies everywhere, on this project and every other one on your machine - the only option available right now, since you haven't cloned anything yet to scope it to. If you already have a Git identity set up for your own projects, run both commands again with `--local` instead once you've cloned the template below, so this project's commits use these details without changing your identity anywhere else.

3. Register for an account on the public **GitLab** or **GitHub** cloud instance you will use. If you have already registered, you can skip this step.

2.2.2 Generate and configure ssh keys for Git

Now generate the ssh keys to use for authentication with your GitLab or GitHub account and configure your ssh settings to use these keys.

1. Follow the instructions below to generate a new SSH key pair and add it to your account. It's best practice to use a modern, secure `ed25519` key.

Generate SSH keys

macOS

1. Open the **Terminal** application.
2. Generate the key for **GitHub**. Only the email address needs changing - the rest of the command is complete as written:

```
ssh-keygen -t ed25519 -C
"your.github.email@example.com" -f ~/.ssh/
id_ed25519_github
```

3. Then generate a **separate** key for **GitLab**:

```
ssh-keygen -t ed25519 -C
"your.gitlab.email@example.com" -f ~/.ssh/
id_ed25519_gitlab
```

4. When prompted, type a strong passphrase. You are asked once per key, so this happens twice.

Windows

1. Open the **PowerShell** application.
2. Create the `.ssh` folder, if it doesn't already exist:

```
mkdir $env:USERPROFILE\.ssh -Force
```

3. Generate the key for **GitHub**. Only the email address needs changing - the rest of the command is complete as written:

```
ssh-keygen -t ed25519 -C
"your.github.email@example.com" -f
$env:USERPROFILE\.ssh\id_ed25519_github
```

4. Then generate a **separate** key for **GitLab**:

```
ssh-keygen -t ed25519 -C
"your.gitlab.email@example.com" -f
$env:USERPROFILE\.ssh\id_ed25519_gitlab
```

5. When prompted, type a strong passphrase. You are asked once per key, so this happens twice.

Linux (Ubuntu)

1. Open the **Terminal** application.
2. Generate the key for **GitHub**. Only the email address needs changing - the rest of the command is complete as written:

```
ssh-keygen -t ed25519 -C
"your.github.email@example.com" -f ~/.ssh/
id_ed25519_github
```

3. Then generate a **separate** key for **GitLab**:

```
ssh-keygen -t ed25519 -C
"your.gitlab.email@example.com" -f ~/.ssh/
id_ed25519_gitlab
```

4. When prompted, type a strong passphrase. You are asked once per key, so this happens twice.

`gitxxx` in the steps that follow

You now have two key files, `id_ed25519_github` and `id_ed25519_gitlab`. The remaining steps are written once, with

> `gitxxx` standing for whichever of the two you are working on - so run them twice, substituting `github` and then `gitlab`.

2. Then configure the SSH config file to use the correct key for each service.

Edit the SSH config file

macOS

Open the file in your preferred text editor (create it if it doesn't exist) - for example with `nano`:

```
nano ~/.ssh/config
```

Paste in the configuration below, then save and close (`Ctrl+O` to save, `Ctrl+X` to exit, in nano).

Windows

Create the file from PowerShell first, then open it - creating it directly inside an editor risks Notepad naming it `config.txt` instead of `config`:

```
New-Item -ItemType File -Path
$env:USERPROFILE\.ssh\config -Force
code $env:USERPROFILE\.ssh\config
```

(Use `notepad` in place of `code` if you'd rather not use VS Code.) Paste in the configuration below, then save.

> **⚠ The file must be called `config`, with no extension**
>
> Notepad silently appends `.txt` unless you prevent it, and Windows hides known extensions in File Explorer, so

> `config.txt` looks identical to `config`. SSH reads only a file named exactly `config` - a misnamed one is ignored entirely, and `git clone` falls back to asking for a password that will never be accepted. Creating the file with `New-Item` first avoids this. To check, and fix it if needed:
>
> ```
> Get-ChildItem $env:USERPROFILE\.ssh
> Rename-Item $env:USERPROFILE\.ssh\config.txt config
> # only if the first command lists config.txt
> ```

Linux (Ubuntu)

Open the file in your preferred text editor (create it if it doesn't exist) - for example with `nano`:

```
nano ~/.ssh/config
```

Paste in the configuration below, then save and close (`Ctrl+O` to save, `Ctrl+X` to exit, in nano).

The configuration to paste in:

```
# GitLab
Host gitlab.com
    HostName gitlab.com
    User git
    IdentityFile ~/.ssh/id_ed25519_gitlab
    AddKeysToAgent yes

# GitHub
Host github.com
    HostName github.com
    User git
    IdentityFile ~/.ssh/id_ed25519_github
    AddKeysToAgent yes
```

Make sure to replace the paths with the correct paths to your SSH keys if you used different names or locations. `AddKeysToAgent yes` is what makes the key-loading step below self-healing - without it, the key you

add to the agent today is gone the next time the agent restarts (a reboot, a logout, on some setups just time), and SSH fails with a permission error that looks like a rejected key rather than a missing one, since the *public* half still authenticates fine and only the signing step - which needs the private half - actually fails.

> **On macOS, add one more line**
>
> Add `UseKeychain yes` too, in each `Host` block above, so macOS can supply the passphrase from your login keychain instead of asking every time - paired with `--apple-use-keychain` on `ssh-add` below. This directive is Apple-specific: **don't** add it on Windows or Linux, where it isn't recognised and breaks every `ssh` command that reads this file with `Bad configuration option: usekeychain`.

> **Tip**
>
> Separate keys per account are safer, but if you reuse one, add its public key to each account separately in Integrate Visual Studio Code with Git below.

3. Set the correct permissions for the SSH config file and the private key(s) to ensure they're secure. If you are using macOS or Linux, run the following commands in your terminal, substituting `gitxxx` and paths to your SSH keys if you used different names or locations:

   ```
   chmod 600 ~/.ssh/config
   chmod 600 ~/.ssh/id_ed25519_gitxxx
   ```

   Windows handles permissions differently and are normally set to only allow access to the user, but ensure that the private key(s) aren't accessible to other users.

4. You've set a passphrase for the SSH keys, so you'll need to enter it every time you use a key. To avoid this, you can use an SSH agent to cache

your passphrase. Follow the instructions below to start the SSH agent and add your keys.

Adding SSH keys

macOS

1. macOS normally starts an SSH agent for you automatically. Add your SSH private keys to it, substituting `gitxxx` with either `github` or `gitlab` depending on which service you are adding the key for - `--apple-use-keychain` stores the passphrase in your login keychain, so the key survives a reboot instead of silently dropping out of the agent:

   ```
   ssh-add --apple-use-keychain ~/.ssh/id_ed25519_gitxxx
   ```

   If this fails with an error about not being able to connect to the agent, start one first, then repeat the command above:

   ```
   eval "$(ssh-agent -s)"
   ```

Windows

1. Set the SSH agent to start automatically with Windows, and then start it. Run these in a PowerShell window opened **as Administrator** (right-click the Start menu, or search for PowerShell, then select **Run as administrator**):

   ```
   Set-Service -Name ssh-agent -StartupType Automatic
   Start-Service ssh-agent
   ```

   > **Run in that order, in an Administrator window**
   >
   > Windows ships this service **disabled**, so `Set-Service` has to take it out of that state before `Start-Service` has anything it's allowed to start - reversed, the first command fails with `Cannot start service ssh-agent`. Both commands also need elevation: an ordinary

> window fails with `Access is denied`, which then makes the second command fail too, for the same underlying reason.
>
> An Administrator PowerShell opens in `C:\WINDOWS\system32` (an ordinary one opens in `C:\Users\yourname`), and its title bar says *Administrator*.

Check it worked before moving on:

```
Get-Service ssh-agent
```

The **Status** column should read `Running`. If it still says `Stopped`, confirm the PowerShell window really is running as Administrator - the title bar says *Administrator* when it is.

2. Back in your normal (non-administrator) PowerShell window, add your SSH private keys to the agent, substituting `gitxxx` with either `github` or `gitlab` depending on which service you are adding the key for:

```
ssh-add $env:USERPROFILE\.ssh\id_ed25519_gitxxx
```

Linux (Ubuntu)

1. Add your SSH private keys to the running SSH agent, substituting `gitxxx` with either `github` or `gitlab` depending on which service you are adding the key for:

```
ssh-add ~/.ssh/id_ed25519_gitxxx
```

Unlike macOS, Linux doesn't always start an SSH agent automatically. If the command above fails with an error about not being able to connect to the agent, start one first, then repeat the command above:

```
eval "$(ssh-agent -s)"
```

5. Display your **public** key, so you can copy it - the next section needs it

pasted into your GitLab and GitHub accounts. Only the public key goes there; never paste the private one (the file with no `.pub` extension).

Display the public key

macOS

```
cat ~/.ssh/id_ed25519_gitxxx.pub
```

Windows

```
Get-Content $env:USERPROFILE\.ssh\id_ed25519_gitxxx.pub
```

Linux (Ubuntu)

```
cat ~/.ssh/id_ed25519_gitxxx.pub
```

Substitute `gitxxx` as before, and run it once for each key you generated. Select the entire line it prints - starting with `ssh-ed25519` and ending with the email address you gave it - and copy it.

2.2.3 Integrate Visual Studio Code with Git

1. Now that you've generated your keys and finished the configuration,

add them to your GitHub and GitLab accounts using the instructions below.

Integrate Visual Studio Code with Git

GitLab

1. Log in to your **GitLab** account in a web browser.
2. In the top-right corner, click on your **profile avatar** and select **Edit profile**.
3. On the left-hand sidebar, select **Access > SSH Keys**.
4. Click **Add new key** and fill out the following details:
    - **Title:** Give it a clear name (e.g., VS Code Extension).
    - **Key:** Paste the contents of your public SSH key file (e.g., `~/.ssh/id_ed25519_gitlab.pub`).
    - **Expiration date:** GitLab fills this in for you, one year ahead, and will not let you leave it empty. Set it well into the future - the end of your course or project, say - or you will be locked out mid-way through and have to generate and register a new key.
5. Click **Add key** to save the key.

> ⚠ **An expired key fails confusingly**
>
> When the date passes, `git push` and `git pull` stop working with a permission error that looks like a misconfigured key rather than an expired one. If pushing suddenly fails having worked for months, check this date first.

GitHub

1. Log in to your **GitHub** account in a web browser.
2. In the top-right corner, click on your **profile avatar** and select **Settings**.
3. On the left-hand sidebar, select **SSH and GPG keys**.
4. Click **New SSH key** and fill out the following details:
    - **Title:** Give it a clear name (e.g., VS Code Extension).
    - **Key:** Paste the contents of your public SSH key file (e.g., `~/.ssh/id_ed25519_github.pub`).
5. Click **Add SSH key** to save the key.

> **No expiry date to set here**
>
> Unlike GitLab, GitHub SSH keys have no expiration field - the key stays valid until you delete it, so there is nothing to set.

2. Test the SSH connection to GitHub and GitLab to ensure that the keys are working correctly. Run the following commands in your terminal:

```
ssh -T git@gitxxx.com
```

If successful, you will see greetings like:

```
Hi username! You've successfully authenticated, but GitHub does
not provide shell access.
Welcome to GitLab, @username!
```

2.3 Cloning the prodockit-template

Cloning the documentation template creates a local copy of the template on your computer. You will then be able to edit the template locally in Visual Studio Code and publish your own documentation website.

2.3.1 Clone the prodockit-template

Start by cloning the template into your own local device.

1. If you don't have a directory for your git repositories, create one for all your *GitLab* or *GitHub* projects on your local desktop. For example, create a directory called 'GitLab' in your home directory.

2. Open a terminal (Terminal on macOS/Debian, Git Bash or PowerShell on Windows 11) and navigate to the directory you created in the previous step.

3. Then run the following command to clone the documentation template into your local directory. Use whichever tab matches the host you use.

Clone the template

GitLab

```
git clone git@github.com:buckwem/prodockit-template.git
```

The template itself lives on GitHub, so that is where you clone it from even if you intend to publish to GitLab.

GitHub

```
git clone git@github.com:buckwem/prodockit-template.git
```

Tip

You can find your `username` by logging into your GitLab or GitHub account and clicking on your profile picture at the top right corner of the page. On **GitLab** your username is **below** your name in the dropdown menu. On **GitHub** your username is **above** your name in the dropdown menu.

2.3.2 Point your clone at your own repository

If you have cloned a repository that has been given to you to work on and is not a direct copy of the template, you can skip this section.

Cloning gives you the template's *files*, but the clone still points at the template's *repository*. This section repoints it at a repository of your own - and is not a step to skip or leave for later, whichever of the two ways it goes wrong if you do:

Skipping this doesn't fail safely for everyone

If you don't have write access to the template, `git push` fails with a permission error - confusing the first time you see it, but harmless, and the fix is this section.

If you do have write access - a maintainer, a contributor, anyone who has ever been given push rights - `git push` **succeeds**, silently, straight into the template repository itself. That repository is public, and every future reader clones it. There is no error to notice and no prompt to confirm; the only protection is doing this section before your first commit.

The steps below have to happen **in this order**, not just as a numbered convention: step 3 (reset the history) deletes the repository's entire `.git` directory, which takes *every* remote with it - including `origin`, however it's currently set. Repoint `origin` at your own repository before resetting, and the reset undoes the repoint along with everything else, leaving you back at square one with no warning that it happened. Reset first, then repoint, as the steps below do.

1. Rename the directory to something meaningful for your own report. Cloning leaves you with a folder called `prodockit-template`, which says nothing about whose work it holds - and if you clone a second project later, you will not be able to tell them apart.

Rename the project folder

macOS

```
mv prodockit-template report-az1234
```

Windows

```
Rename-Item prodockit-template report-az1234
```

Linux (Ubuntu)

```
mv prodockit-template report-az1234
```

Replace `report-az1234` with a name that identifies your own work - your username, your coursework code, or whatever your course tutor specifies.

> **Renaming the folder doesn't rename the repository**
>
> This changes only the folder on your own machine. The project's name on GitLab or GitHub is unaffected, and so is the `origin` remote inside it - `git push` and `git pull` carry on working exactly as before.

2. Check what your clone currently points at:

Check the current remote

macOS

```
cd report-az1234
git remote -v
```

Windows

```
cd report-az1234
git remote -v
```

Linux (Ubuntu)

```
cd report-az1234
git remote -v
```

A fresh clone has exactly one remote, `origin`, pointing at the template:

```
origin  git@github.com:buckwem/prodockit-template.git (fetch)
origin  git@github.com:buckwem/prodockit-template.git (push)
```

Figure 2.1. Right after cloning: `origin` points at the template

If you added any others of your own - a `gitlab` mirror, say - they will be listed here too. You do not need to remove any of them by hand: the next step deletes the repository's entire `.git` directory, which takes every remote with it.

3. Start with a fresh commit history. This is your own independent project, so carrying the template's entire commit log and branches from the template into it serves little purpose.

Reset the repository history

macOS

```
rm -rf .git
git init -b main
git config core.fileMode false
```

`rm -rf .git` cannot be undone

This permanently deletes the repository's history from your machine - every commit, branch and tag. There is no undo, and nothing to recover from, because the deleted history is the thing that would have recovered it. Make sure you are in the right directory (`pwd`) and that you have pushed anything you care about somewhere else first.

Windows

```
Remove-Item -Recurse -Force .git
git init -b main
git config core.fileMode false
```

`Remove-Item -Recurse -Force .git` cannot be undone

This permanently deletes the repository's history from your machine - every commit, branch and tag. There is no undo, and nothing to recover from, because the deleted history is the thing that would have recovered it. Make sure you are in the right directory (`pwd`) and that you have pushed anything you care about somewhere else first.

Linux (Ubuntu)

```
rm -rf .git
git init -b main
git config core.fileMode false
```

`rm -rf .git` cannot be undone

> This permanently deletes the repository's history from your machine - every commit, branch and tag. There is no undo, and nothing to recover from, because the deleted history is the thing that would have recovered it. Make sure you are in the right directory (`pwd`) and that you have pushed anything you care about somewhere else first.

Why `core.fileMode false`

Git records whether a file is executable, and treats a change to that bit as a change to the file. Cloud-sync clients - OneDrive in particular - rewrite those bits as they sync, so a folder that syncs can show every file in the project as modified when not one byte of content has changed. Turning it off tells git to ignore the executable bit entirely.

It is set per repository and is not committed, so it has to be repeated on each machine and after each fresh clone. If you keep all your work in a synced folder, `git config --global core.fileMode false` saves repeating it - though `git init` and `git clone` each write their own local setting, which still wins.

`rm -rf .git` / `Remove-Item -Recurse -Force .git` deletes the whole repository, remotes included, and `git init` starts a brand new one with none at all:

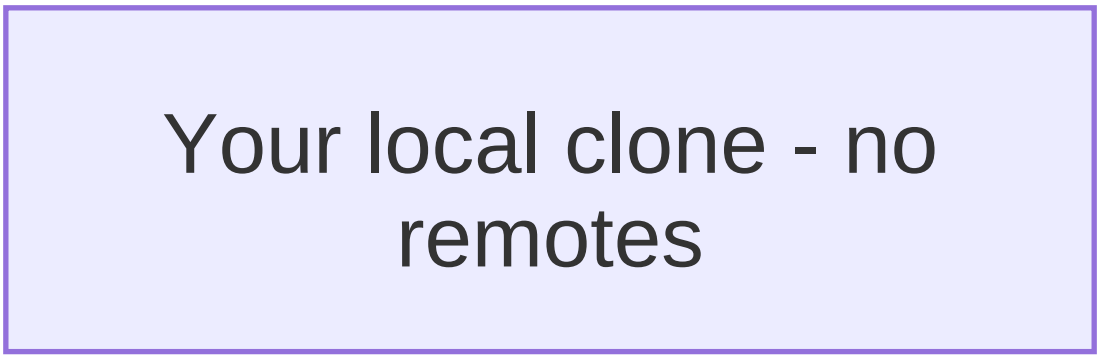


Figure 2.2. After resetting the history: no remotes at all

Run `git remote -v` at this point and it prints nothing.

4. Create the new, **empty** repository on the host you are publishing to. Do **not** add a README, `.gitignore` or licence - the template brings its own, and an initial commit on the host side collides with what you are about to push.

Create the empty repository

GitLab

On the GitLab website, click **New project > Create blank project**. Name it, set **Visibility Level** to **Private**, and untick **Initialize repository with a README**.

GitHub

On the GitHub website, click **New repository**. Name it, set it to **Private**, and leave every **Initialize this repository with** option unticked.

5. Point your clone at your own repository, using the tab matching your host:

> **There is nothing to remove first**
>
> Deleting `.git` in step 3 took the template's `origin` with it, along with any other remotes you saw in step 2 - `git init` starts a repository with none at all. If you run `git remote remove origin` out of habit, Git tells you so:
>
> ```
> error: No such remote: 'origin'
> ```
>
> That message means the previous step did its job, not that anything is wrong.

Point the clone at your repository

GitLab

```
git remote add origin git@gitlab.com:your-namespace/your-new-directory-name.git
```

Replace `gitlab.com` with your own GitLab instance if it is self-hosted.

GitHub

```
git remote add origin git@github.com:your-username/your-new-directory-name.git
```

Confirm it took:

```
git remote -v
```

Figure 2.3. After repointing: `origin` points at your own repository

`origin` now points at your own repository rather than the template's - compare this against the first diagram in this section, where it pointed at `prodockit-template` instead.

> **Don't commit or push yet**
>
> Step 3 left you with an empty repository - the template's files are all still there, but Git is tracking none of them yet.
>
> Resist committing them now. The template's files still name the *template's* repository in several places, and `prodockit sync-repo` in the next section is what repoints them at yours. Committing first would put those stale references into your project's very first commit, and you would then be correcting them in the second.
>
> Install Python and Zensical ends with the `git add`, `git commit` and `git push` that publish everything in one go, once there is something correct to publish.

We are not complete with the setup yet, but you now have a local copy of the template that is connected to your own repository on GitLab or GitHub. Do not start editing yet, as the next section installs Python, Zensical and prodockit, which are needed to build your documentation.

2.4 Install Python and Zensical

Here are brief instructions for installing Python are below for macOS, Windows 11, and Linux (Ubuntu/Debian). However, it's recommended to refer to the official Python installation documentation for your operating system.

> **Note**
>
> You may need to use 'python3' and 'pip3' instead of 'python' and 'pip' depending on your system configuration.

The instructions below are for installing Python 3.12 or later. If you have an older version, please update to Python 3.12 or later.

1. Follow the instructions below to install Python, create a virtual environment, and install Zensical inside it for your operating system.

Install Python, Zensical and prodockit

macOS

1. If you use the Homebrew package manager, run this command in your Terminal to install Python. If you don't have Homebrew installed, you can install it by following the instructions on the Homebrew website.

   ```
   brew install python3
   ```

2. Install Pango, which is not a Python package, so `pip` cannot install it for you:

   ```
   brew install pango
   ```

3. Install Pandoc at the version this project builds with. Homebrew always installs the newest release, which is why it is not used here - see Which pandoc version below:

   ```
   curl -fsSL -o /tmp/pandoc.pkg "https://github.com/
   jgm/pandoc/releases/download/3.10.1/pandoc-3.10.1-
   arm64-macOS.pkg"
   sudo installer -pkg /tmp/pandoc.pkg -target /
   ```

   On an Intel Mac, use `pandoc-3.10.1-x86_64-macOS.pkg` instead.

   > **Why Pango but a fixed Pandoc**
   >
   > `prodockit pdf` shells out to `pandoc`, which hands the result to WeasyPrint to lay out the pages - and WeasyPrint draws text through Pango, so `pango` alone is enough (glib, HarfBuzz and fontconfig come along as its dependencies). Skipping either still looks fine right

> up until `prodockit pdf`, which then fails with `pandoc exited with status 43` - see WeasyPrint cannot start (status 43) if that happens.

4. Install the desktop font files this template's PDF uses by default - **Inter** and **JetBrains Mono**:

```
brew install --cask font-inter font-jetbrains-mono
```

> **Why this early**
>
> The website loads its fonts from a CDN at view time, but the PDF has no such fallback - WeasyPrint has to embed the actual font files, and silently substitutes a fallback font instead of erroring if they are missing. See Fonts for the full picture, including how to check the right fonts actually made it into a built PDF.

5. Open **Terminal** in your project folder and run the following commands to create a virtual environment and install Zensical inside it:

```
# 1. Create the virtual environment
/opt/homebrew/bin/python3 -m venv .venv

# 2. Activate it
source .venv/bin/activate
```

Your prompt gains a `(.venv)` prefix, which is how you know the virtual environment is active:

```
(.venv) yourname@Mac your-project %
```

It disappears when you close the terminal, and every new one needs activating again - or let VS Code do it, which the Python extension below handles for you.

> **Why the full path to Python**
>
> macOS ships its own older Python, and a plain `python3`

> may well find that one instead of Homebrew's. Naming `/opt/homebrew/bin/python3` explicitly builds the virtual environment from the version you just installed. On an Intel Mac, Homebrew installs to `/usr/local` instead, so use `/usr/local/bin/python3`.

Windows

1. Download and run the official Python installer from python.org.

   > **Three things to get right during install**
   >
   > - Check **Add python.exe to PATH** on the first screen. This is what lets you run `python` from the command line at all, and also puts `pip` and every command it installs on your `PATH`.
   > - Once installation finishes, a final screen offers **Disable path length limit** - click it. Windows historically caps a full file path at 260 characters, and this project's own dependencies nest deep enough (`.venv\Lib\site-packages\...`, `tools\mermaid\node_modules\...`) to hit that limit without it.
   > - Make sure you are running the installer you just downloaded, not Windows' own placeholder. Typing `python` in a terminal with no real Python installed opens the Microsoft Store instead of running anything - if that still happens *after* installing, search **Manage app execution aliases** and turn off the **App Installer** entries for `python.exe` / `python3.exe`, which take priority over the one you just installed.

2. Next install pandoc, which is not a Python package, so `pip` cannot install it for you. Open **PowerShell** and run the following command:

   ```
   winget install --id JohnMacFarlane.Pandoc --version
   3.10.1
   ```

> **The package is under its author's name, not `Pandoc`**
>
> winget identifies packages as `Publisher.Package`, and Pandoc's publisher is its author, John MacFarlane. There is no `Pandoc.Pandoc`, so guessing that gives:
>
> ```
> No package found matching input criteria.
> ```
>
> `winget search pandoc` lists the real identifier if you ever need to check it.

3. Install the graphics libraries WeasyPrint needs. Pandoc hands your document to WeasyPrint to lay out the pages, and WeasyPrint is not pure Python - it draws text through Pango, which on Windows comes from MSYS2. Install MSYS2 first:

   ```
   winget install --id MSYS2.MSYS2
   ```

   Then open the **MSYS2 MINGW64** shell from your Start menu (not PowerShell) and run:

   ```
   pacman -S mingw-w64-x86_64-pango
   ```

   If that fails partway through with a download error, just run it again - MSYS2's mirrors are occasionally flaky, and a second attempt usually goes through cleanly.

   Finally, add `C:\msys64\mingw64\bin` to your user `PATH`, the same way you added Python: search for *Edit the system environment variables*, click **Environment Variables**, select **Path** under *User variables*, and add that folder. Close and reopen PowerShell afterwards so the change takes effect - the next two steps continue in that new window.

   > **No virtual environment to reactivate yet**
   >
   > Unlike other "close and reopen PowerShell" steps on this page, there's nothing to `cd` back to or activate here. The virtual environment isn't created until step 5

> below - reopening PowerShell now just gets the `PATH` change into a fresh window before continuing.

> **Why this is needed**
>
> That folder is where WeasyPrint finds `libgobject-2.0-0.dll`, `libpango-1.0-0.dll`, `libharfbuzz-0.dll` and `libfontconfig-1.dll` - installing `pango` brings all four in. Skipping this still looks fine until `prodockit pdf`, which then fails with `pandoc exited with status 43` - see WeasyPrint cannot start (status 43) if that happens.

4. Install the desktop font files this template's PDF uses by default - **Inter** and **JetBrains Mono**. Download the desktop (`.ttf` / `.otf`) files for each - Inter, JetBrains Mono - then select them all, right-click, and choose **Install for all users**.

   > **Why this early**
   >
   > The website loads its fonts from a CDN at view time, but the PDF has no such fallback - WeasyPrint has to embed the actual font files, and silently substitutes a fallback font instead of erroring if they are missing. See Fonts for the full picture, including why a `.woff` / `.woff2` download will not do, and how to check the right fonts actually made it into a built PDF.

5. Allow PowerShell to run scripts. Windows blocks all of them by default, and activating a virtual environment *is* a script, so this has to be done once before the next step will work:

   ```
   Set-ExecutionPolicy -Scope CurrentUser -
   ExecutionPolicy RemoteSigned
   ```

   Depending on your PowerShell version it may ask you to confirm the change; answer `Y` if it does. Often it simply returns to the prompt, which means it worked.

> **What this changes**
>
> Without it, activating the venv fails with `... cannot be loaded because running scripts is disabled on this system`. `RemoteSigned` allows locally-written scripts while still requiring signed ones from the internet; `-Scope CurrentUser` limits that to your account, so it needs no Administrator window and is a one-time, per-account change.
>
> Would rather not change it at all? Use **classic CMD** instead of PowerShell and run `.\.venv\Scripts\activate.bat` in the next step - `.bat` files aren't covered by execution policy.

6. Change into your project folder, then create a virtual environment and install Zensical inside it:

```
cd C:\path\to\your-project
```

> **Check where you are first**
>
> The steps above will have moved you. The SSH agent needed an Administrator window, which opens in `C:\WINDOWS\system32`, and every "close and reopen PowerShell" leaves you in your home directory, `C:\Users\yourname`.
>
> `python -m venv .venv` does not object to either. It creates a perfectly good virtual environment in the wrong place, and the mistake only shows up a step later when `pip install -r requirements.txt` cannot find a file that is sitting in your project folder all along.
>
> `pwd` prints where you are.

```
# 1. Create the virtual environment
python -m venv .venv

# 2. Activate it (choose the line matching your terminal)
.\.venv\Scripts\Activate.ps1     # <-- Use this if you are in PowerShell
```

```
.\.venv\Scripts\activate.bat     # <-- Use this if
you are in classic CMD
```

Your prompt gains a `(.venv)` prefix, which is how you know the virtual environment is active:

```
(.venv) PS C:\path\to\your-project>
```

It disappears when you close the terminal, and every new one needs activating again - or let VS Code do it, which the Python extension below handles for you.

Linux (Ubuntu)

1. Open a terminal and run the following command to install Python, the `venv` module, pandoc, the graphics libraries WeasyPrint needs, and the fonts this template's PDF uses by default. None of these is a Python package, so `pip` cannot install them for you:

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
curl \
  libpango-1.0-0 libpangoft2-1.0-0 libharfbuzz-
subset0 \
  fonts-inter fonts-jetbrains-mono
```

Ubuntu's own `pandoc` package is several major versions behind, far enough to change how the PDF renders, so install the pinned release directly - see Which pandoc version below. This picks the `amd64` or `arm64` package to match your CPU, so it also works on Ubuntu running under an Apple Silicon Mac:

```
curl -fsSL -o /tmp/pandoc.deb "https://github.com/
jgm/pandoc/releases/download/3.10.1/pandoc-3.10.1-1-$
(dpkg --print-architecture).deb"
sudo apt install -y /tmp/pandoc.deb
```

> **Why the three library packages**

> Pandoc hands the result to WeasyPrint, which draws text through Pango and won't start without it. `libharfbuzz-subset0` is easy to miss - on Debian it's a *separate* package from `libharfbuzz0b`, and WeasyPrint needs this one specifically (glib and fontconfig aren't listed, since `libpango-1.0-0` already depends on them). Skipping this still looks fine until `prodockit pdf`, which then fails with `pandoc exited with status 43` - see WeasyPrint cannot start (status 43) if that happens.

> **Debian 12 or Ubuntu 22.04 and newer**
>
> `libharfbuzz-subset0` does not exist on older releases. On those, upgrade the distribution rather than hunting for a substitute package.

> **Why the fonts, this early**
>
> The website loads its fonts from a CDN at view time, but the PDF has no such fallback - WeasyPrint has to embed the actual font files, and silently substitutes a fallback font instead of erroring if they are missing. Skipping this still looks fine right up until your first `prodockit pdf`, whose test suite (if you run one - see Testing) then fails with `No 'Inter' font found anywhere in the compiled PDF`. See Fonts for the full picture, including how to check the right fonts actually made it into a built PDF.

2. Navigate to your project folder and run the following commands to create a virtual environment and install Zensical inside it:

```
# 1. Create the virtual environment
python3 -m venv .venv

# 2. Activate it
source .venv/bin/activate
```

Your prompt gains a `(.venv)` prefix, which is how you know the virtual environment is active:

```
(.venv) yourname@host:~/your-project$
```

It disappears when you close the terminal, and every new one needs activating again - or let VS Code do it, which the Python extension below handles for you.

2.4.1 Which pandoc version

Every command above installs pandoc **3.10.1** specifically, rather than whatever your package manager considers current. This project is built and tested against that version, and pandoc is not always compatible with itself across releases: 3.10 changed how it reads highlighted code in HTML, in a way that broke every fenced code block in the PDF while the build still reported success - see Pandoc version drift for what that looked like.

Confirm which version you actually have:

```
pandoc --version
```

The first line should read `pandoc 3.10.1`. If it doesn't - a Homebrew upgrade, a distribution update, or `winget upgrade` run without thinking about it will all move this - repeat the pandoc install step above for your platform to bring it back.

2. Install Zensical and prodockit inside the virtual environment. The `requirements.txt` file in the template lists the required packages, so you can install them all with a single command (use `pip` if `pip3` is not available):

```
pip3 install -r requirements.txt
```

3. Check that the `prodockit` command actually resolves to the one you just installed:

```
prodockit --version
```

`pip` exiting without an error only means the package landed in `.venv` - it doesn't prove your shell finds it there first. An older, separately-installed `prodockit` earlier on your `PATH` shadows it silently, and every command in this guide from here on would run against that instead.

4. Check that WeasyPrint can find its graphics libraries. This is the one part of the setup `pip` cannot verify for you, so it is worth confirming now rather than at your first PDF build:

   ```
   python3 -c "import weasyprint; print(weasyprint.__version__)"
   ```

   A version number means everything is in place. If instead you get a long error ending in `cannot load library`, the libraries from the step above are missing or cannot be found - go back and install them.

5. Fetch the citation style your first build needs. The template enables `prodockit.bibliography` by default, pointing `csl_style` at `harvard-cite-them-right.csl` - but that file isn't part of the clone, so `zensical serve` / `zensical build` / `prodockit pdf` all fail outright until it's in place. Fetch it once, from your project root:

   ### Fetch the citation style

   macOS

   ```
   curl -fsSL -o harvard-cite-them-right.csl "https://
   www.zotero.org/styles/harvard-cite-them-right"
   ```

   Windows

   ```
   Invoke-WebRequest -Uri "https://www.zotero.org/styles/
   harvard-cite-them-right" -OutFile harvard-cite-them-
   right.csl
   ```

   Linux (Ubuntu)

   ```
   curl -fsSL -o harvard-cite-them-right.csl "https://
   www.zotero.org/styles/harvard-cite-them-right"
   ```

   See An alternative: prodockit.bibliography for what this feature does, and how to fetch a different CSL style instead.

6. Sync the repository's own self-references to match. The template's files still name the *template's* repository in several places, and nothing about changing a Git remote updates them:

   ```
   prodockit sync-repo
   ```

   It reports what it changed:

   ```
   Detected GitLab remote (https://gitlab.surrey.ac.uk/az1234/
   report-az1234); updated: repo_url, repo_name, theme.icon.repo,
   README badges
   ```

   This rewrites `repo_url`, `repo_name`, `theme.icon.repo` and `edit_uri` in `zensical.toml`, plus the badge row in your `README.md`, to match the `origin` you just set - so your built site and PDF link to your own repository rather than the template's. Note `theme.icon.repo` in that list: moving from a GitHub template to a GitLab project switches the header's brand icon to match, which is easy to miss by hand. Only the values that actually needed changing are listed, so the set you see may be smaller.

   > **Check it any time**
   >
   > `prodockit sync-repo --check` writes nothing and exits non-zero if these have drifted from your remote - useful after any later change of host. See Checks worth having.

7. Lets now commit the changes to your own repository. Run the following commands to commit and push the changes:

   ```
   git add .
   git commit -m "Initial commit with Zensical and prodockit
   installed"
   git push -u origin main
   ```

2.4.2 Install Zensical Studio and other plugins

Now we'll install the Zensical Studio plugin for Visual Studio Code, which provides a set of tools to help you work with Zensical projects, including commands to build and preview your site. Then we'll install a couple of other useful plugins for working with Markdown and TOML files.

1. Start by opening Visual Studio Code and navigating to the Extensions view by clicking on the Extensions icon in the Activity Bar on the side of the window or pressing `Ctrl+Shift+X` / `Cmd+Shift+X`.

2. Install the **Python** extension (published by Microsoft) by searching for

"Python" in the Extensions view and clicking **Install**. As well as Python support, this is what makes VS Code notice the `.venv` folder in your project and activate the virtual environment automatically in every new Terminal in VS Code - so you don't have to run `source .venv/bin/activate` by hand each time you open one.

> **Tip**
>
> Check it worked by opening a new terminal (**Terminal > New Terminal**) - the prompt should start with `(.venv)`. If it doesn't, reopen VS Code in the project folder, then choose **Python: Select Interpreter** from the Command Palette (`Ctrl+Shift+P` / `Cmd+Shift+P`) and pick the one inside `.venv`.

3. Install the **Zensical Studio** extension by searching for "Zensical Studio" in the Extensions view and clicking **Install** and then **Trust Publisher and Install** when prompted. This extension provides a set of tools to help you work with Zensical projects, including commands to build and preview your site.

4. Follow the instructions on the Zensical Studio plugin page to configure the extension. You may find the configuration aleady exists but if it's not there add to the `.vscode/settings.json` file in your project directory the following lines:

```
{
  "files.associations": {
   "*.md": "python-markdown"
  }
}
```

5. Install the **Even Better TOML** extensiuon for Visual Studio Code by searching for "Even Better TOML" in the Extensions view and clicking **Install** and then **Trust Publisher and Install** when prompted. This extension provides syntax highlighting and other features for working with TOML files, which are used for configuration in Zensical projects.

6. Install the **LTeX+ – LanguageTool grammar/spell checking** plugin for Visual Studio Code by searching for "LTeX+" in the Extensions view and clicking **Install** and then **Trust Publisher and Install** to enable spelling and grammar checking for Markdown. Configure the plugin's *language* setting to whichever English (or other language LTeX+ supports) you're actually writing in.

> **Get this right, or corrections are confidently wrong**

> Set to the wrong variety, LTeX+ still checks every sentence - it just checks it against the wrong rules, and offers "corrections" for perfectly correct spelling and phrasing in the variety you're actually using. That's worse than no checking at all, since a wrong suggestion looks exactly as confident as a right one.

There are many other extensions available for Visual Studio Code that can help you with your documentation. You can explore the Visual Studio Code Marketplace to find more extensions that suit your needs.

2.5 Install the diagram and maths tooling

Two things your document can contain - diagrams and mathematical notation - need tooling that none of the steps so far has installed.

On the website they look after themselves: the reader's browser draws them as the page loads. A PDF has no browser, so `prodockit pdf` converts both into images *before* building the document, using two Node.js programs to do it.

> **Without these, the PDF is wrong rather than missing**
>
> `prodockit pdf` does **not** fail when they are absent. It leaves the content as it found it, so instead of a flowchart your PDF shows the diagram's own definition text - the `graph LR` line and every node written out beneath it - and instead of a typeset equation, raw LaTeX with all its backslashes and braces.
>
> Meanwhile the website renders both perfectly. So nothing looks wrong until somebody opens the PDF, which may be well after you have written the document.

2.5.1 Install Node.js

The two tools are Node.js programs, so install Node.js first. Version 22 or newer - that is what the automated builds use.

Install Node.js

macOS

```
brew install node
```

Windows

```
winget install OpenJS.NodeJS.LTS
```

Close and reopen PowerShell afterwards, so it picks up the new `PATH`. The new window starts in `C:\Users\yourname` with the virtual environment inactive, so get back to where you were before continuing:

```
cd C:\path\to\your-project
.\.venv\Scripts\Activate.ps1
```

Check the prompt starts with `(.venv)` again. The next step's `npm ci` commands are relative to your project folder, and every `prodockit` command after it lives inside the virtual environment - outside it, PowerShell reports `The term 'prodockit' is not recognized`.

Linux (Ubuntu)

Ubuntu's own `nodejs` package is often several versions behind. Use NodeSource's repository to get a current release:

```
sudo apt update
sudo apt install -y curl

curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E
```

```
bash -
sudo apt install -y nodejs
```

> ⚠ **Don't skip the `curl` line**
>
> A clean Ubuntu install does not necessarily have `curl`. Without it the first command fails with `Command 'curl' not found` - and the failure does not stop there, because the `apt install` on the next line still **succeeds**, quietly fitting Ubuntu's own older Node.js instead of NodeSource's.
>
> You then have a `node` that looks installed but no `npm` at all, and the toolchain commands in the next section fail for what appears to be an unrelated reason. If you have already hit this, install `curl` and run the two NodeSource lines again - the correct package replaces the wrong one.

Check it worked - **both** commands, not just the first:

```
node --version
npm --version
```

You should get two version numbers, with `node` at 22 or above:

```
v22.14.0
10.9.2
```

> **`node` answers but `npm` is not found**
>
> That is the signature of the NodeSource step not having run - the most likely cause on Linux being the missing `curl` described above. Node came from your distribution's own package instead, which does not always bring `npm` with it.
>
> Fix the earlier step and run it again rather than installing `npm` separately, so both come from the same source and stay in step.

2.5.2 Install the two toolchains

Your cloned template already contains the manifests and lockfiles for both

tools, in `tools/mermaid` and `tools/mathjax` - so you only need to install them.

If you're on Linux, install a native Chromium and point Puppeteer at it **before** running `npm ci` below, rather than letting `tools/mermaid`'s own `npm ci` download one for you - Puppeteer's download is not guaranteed to match your CPU's architecture. This matters most on ARM64 machines (an Apple Silicon Linux VM, an AWS Graviton instance, a Raspberry Pi), where `npm ci` would otherwise silently fetch an x86_64 Chrome build it can never run, but it costs nothing to do on any Ubuntu install:

```
sudo apt update
sudo apt install -y chromium-browser
which chromium-browser || which chromium
```

The second command should print a path such as `/usr/bin/chromium-browser` or `/usr/bin/chromium` - that's what the next step needs. Point Puppeteer at it, and skip its own download entirely, for this session, then make both permanent so every future session picks them up too:

```
export PUPPETEER_EXECUTABLE_PATH=$(which chromium-browser || which
chromium)
export PUPPETEER_SKIP_DOWNLOAD=true
echo 'export PUPPETEER_EXECUTABLE_PATH=$(which chromium-browser ||
which chromium)' >> ~/.bashrc
echo 'export PUPPETEER_SKIP_DOWNLOAD=true' >> ~/.bashrc
source ~/.bashrc
```

Open a new terminal for this step (it picks up the exports above from `~/.bashrc` automatically), and make sure it's actually sitting in your project's root directory first - the `--prefix` paths below are relative to wherever you run them from:

```
cd path/to/your-project
```

Then install both:

```
npm ci --prefix tools/mermaid
npm ci --prefix tools/mathjax
```

`npm ci` installs the exact versions recorded in each lockfile, which is what the automated builds use too - so your PDF is rendered by the same versions as the published one.

This creates a `node_modules` folder inside each, which is deliberately not committed (see `.gitignore`). Run these two commands again if you ever re-clone the project.

Install the MathJax bundle the *website* itself needs, from the `tools/mathjax`

install you just ran. It isn't committed - it's third-party code, and a repository is redistribution - so this step is what makes formulas render at all until you run it, on every machine that needs to see them, including CI (see Diagrams and maths):

```
mkdir -p docs/javascripts/vendor/mathjax
cp tools/mathjax/node_modules/mathjax-full/es5/tex-svg-full.js docs/
javascripts/vendor/mathjax/
cp tools/mathjax/node_modules/mathjax-full/LICENSE docs/javascripts/
vendor/mathjax/
```

Then write the config MathJax needs to actually process the formulas your document contains - without it, every equation renders as raw TeX, with nothing in the build to say why:

```
cat > docs/javascripts/mathjax.js <<'MATHJAX'
window.MathJax = {
  tex: {
    processEscapes: true,
    processEnvironments: true,
  },
  options: {
    ignoreHtmlClass: ".*|",
    processHtmlClass: "arithmatex",
  },
};
MATHJAX
```

If npm reports vulnerabilities or an `allow-scripts` warning

Both are normal here, not a sign anything went wrong:

```
Run `npm audit` for details.
npm warn allow-scripts 1 package has install scripts not yet
covered by allowScripts:
npm warn allow-scripts   puppeteer@25.3.0 (postinstall: node
install.mjs)
```

The vulnerability count comes from `npm audit` scanning the whole dependency tree Puppeteer pulls in for known advisories, most of which don't apply to how this project uses it - a locally-run PDF build, not a public-facing server. There's nothing to fix here; running `npm audit fix` is more likely to break the pinned versions the lockfile records than to help.

The `allow-scripts` warning is different: recent npm versions skip Puppeteer's own setup step, which downloads the headless browser

Mermaid draws diagrams with. The install still succeeds - if a later PDF build reports it cannot find a browser, approve the step and reinstall:

```
npm approve-scripts puppeteer --prefix tools/mermaid
npm ci --prefix tools/mermaid
```

Starting a project that isn't from the template?

Then you have no `tools/` directory to install from, and need `prodockit init-tools` first to create it. Running it on a copy of the template is harmless but pointless - it will just report `Kept existing tools/mermaid/package.json` for each file it finds. See Diagrams and maths for the full picture.

Test the whole thing by building the PDF - see Generate the Source and PDF documents in the next section. If a diagram appears as an image rather than as text, everything is set up correctly. None of this is required if your document has no diagrams or formulas, but setting it up now costs nothing and means the trap above can't catch you later when you add your first one.

2.6 Where to go next

You now have Visual Studio Code, Git, Zensical and the diagram and maths tooling installed, and your own copy of the documentation template cloned locally. Continue to Start editing to preview your changes locally and publish them to GitLab or GitHub.

3. Start editing

This page covers the day-to-day cycle of working on your document: previewing your changes locally, syncing them to GitLab or GitHub, viewing the published website, building the PDF, working with branches and issues, troubleshooting common problems, and finally releasing your report. It assumes no prior experience with the command line, Git, or software development - each step includes the exact commands to type. For one-time setup (installing Python, Git, and Zensical itself), see Install tooling first if you haven't already.

3.1 Viewing documentation locally

Zensical renders your Markdown into a live, locally hosted website as you write, without you needing to push anything - so you can check headings, links, images, diagrams, and PDF-only/web-only content render correctly before anyone else sees them.

3.1.1 Open a terminal

A terminal (also called a command line, console, or shell) is a text-based way to give your computer instructions by typing commands, instead of clicking buttons. It can look intimidating at first, but this whole workflow only needs a handful of commands, and they're all given below.

The easiest way to open one is Visual Studio Code's own integrated terminal:

1. Open your project folder in Visual Studio Code, if it isn't already open (**File** > **Open Folder...**).
2. Open the integrated terminal, whichever way is quickest for you:
   - Menu: **View** > **Terminal**.
   - Keyboard shortcut: `Ctrl+`` on Windows/Linux, `Cmd+`` on macOS.
   - Command Palette (`Ctrl+Shift+P` / `Cmd+Shift+P`) > **View: Toggle Terminal**.
3. A panel opens at the bottom of the window, already sitting in your project folder - defaulting to PowerShell on Windows, or your shell of choice (bash/zsh) on macOS and Linux.

This integrated terminal also activates your Python virtual environment automatically (a self-contained folder holding just this project's Python packages, kept separate from everything else on your computer), as long as you've selected the `.venv` interpreter once - see Install Python and Zensical. That's the recommended path, since it needs no further steps below.

If you'd rather use your system's own terminal application instead of Visual

Studio Code's, you need to navigate to your project folder and activate the virtual environment yourself:

Activate the virtual environment manually

macOS

1. Open a terminal application: press `Cmd+Space` to open Spotlight, type `Terminal`, and press `Enter`.
2. Navigate to your project folder using the `cd` (change directory) command - replace the path below with wherever you cloned your project:

   ```
   cd path/to/your/project
   ```

3. Activate the virtual environment:

   ```
   source .venv/bin/activate
   ```

   Your prompt now starts with `(.venv)`, confirming it's active.

Windows

1. Open PowerShell: press the `Windows` key, type `PowerShell`, and press `Enter`.
2. Navigate to your project folder using the `cd` command:

   ```
   cd path\to\your\project
   ```

3. Activate the virtual environment:

   ```
   .\.venv\Scripts\Activate.ps1
   ```

   Your prompt now starts with `(.venv)`, confirming it's active.

Linux (Ubuntu)

1. Open a terminal application: look for **Terminal** in your applications menu.
2. Navigate to your project folder using the `cd` (change directory) command - replace the path below with wherever you cloned your project:

   ```
   cd path/to/your/project
   ```

3. Activate the virtual environment:

   ```
   source .venv/bin/activate
   ```

   Your prompt now starts with `(.venv)`, confirming it's active.

3.1.2 Start the preview server

1. In your terminal (with the virtual environment active), start the local preview server:

   ```
   zensical serve
   ```

2. Wait for it to finish starting - you'll see some log messages ending with a local web address.
3. Open that address (typically http://127.0.0.1:8000) in your browser to view your documentation.

Leave `zensical serve` running in its terminal while you write - it watches your files and automatically rebuilds and refreshes the browser whenever you save a change, so you don't need to restart it after every edit. To stop it, click back into its terminal and press `Ctrl+C`.

> **Tip**
>
> `zensical serve` only builds the website - it doesn't touch `docs/site_documentation.pdf`. See Build the PDF below to preview the PDF output.

3.1.3 Generate the Source and PDF documents

You may notice there is no Source or PDF document for download. The two buttons are on the front page, but clicking either one gets you nothing - the documents they point at are generated rather than stored, so a fresh clone doesn't have them yet. They are also deliberately excluded from Git (see `.gitignore`), which keeps large binary files out of your repository and stops every rebuild showing up as a change.

You can generate these two documents by running the following commands:

```
prodockit pdf              # the report itself
prodockit source-bundle    # your Markdown content and
zensical.toml, one file per page
```

`prodockit pdf` writes the report to `docs/site_documentation.pdf`, which the **PDF** button already points at. `prodockit source-bundle` writes `docs/source_bundle.pdf` the same way, which the **Source** button points at - both land directly in `docs/`, so there's nothing to copy anywhere before the preview can serve them.

> **Only your Markdown and config, not your whole repository**
>
> `prodockit source-bundle` bundles every `.md` file plus `zensical.toml` - your documentation's own source, not the template's build tooling. See Source-code bundling if your submission needs the whole repository bundled instead.

Refresh your browser and both buttons will work.

> **Re-run it after changes you want reflected**
>
> Neither document rebuilds itself as you write - unlike the website, which `zensical serve` refreshes automatically. Run `prodockit pdf` again whenever you want the PDF to catch up. See Build the PDF for more on the command, including what it needs installed and how the automated builds use it.

3.2 Synchronise your updates

Whenever you've made a change you want to keep, there are three things to do: **save** the file, **commit** it (record a labelled snapshot of the change in your project's history), and **push** it (upload that snapshot to GitLab or GitHub, where it's backed up and, once it reaches your default branch, published). You can do all three either through Visual Studio Code's Source Control view,

or by typing Git commands directly - both do exactly the same thing, so use whichever feels more comfortable.

Commit and push your changes

Visual Studio Code

1. Make sure you've saved your changed files (a filled circle next to a file name in the Explorer tab means it has unsaved changes - select the file and press `Ctrl+S` / `Cmd+S`).

2. Click the **Source Control** icon in the left-hand sidebar. You'll see a list of every changed and new file.

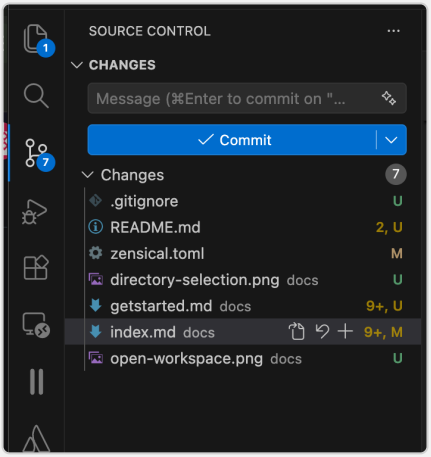


Figure 3.1. Initial commit

3. Type a short, descriptive message in the message box (for example, "Add section 2 draft") - this is the label future-you (or a marker) will see when looking back through the history.

4. Press the **Commit** button and select **Save All and Commit Changes**. This records the snapshot on your computer only - you haven't sent anything anywhere yet.

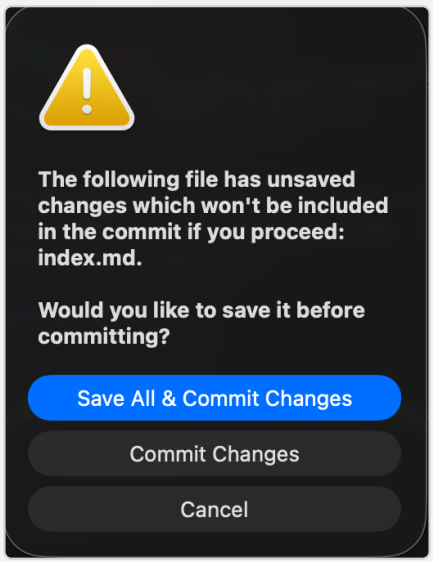


Figure 3.2. Commit changes

5. Press **Sync Changes** to push your commit to GitLab or GitHub (and pull down anyone else's changes too).

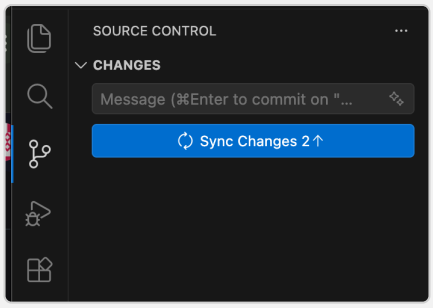


Figure 3.3. Sync changes

Command line

1. Check what's changed - this lists every file you've added, edited, or deleted since your last commit:

```
git status
```

2. Stage the files you want to commit - "staging" means marking them so Git includes them in the next commit (use `git add .` to stage everything shown by `git status` in one go):

```
git add docs/section1.md
```

3. Commit the staged changes with a short, descriptive message:

   ```
   git commit -m "Add section 2 draft"
   ```

   This records the snapshot on your computer only - you haven't sent anything anywhere yet.

4. Push your commit to your GitLab or GitHub remote, uploading it so it's backed up and visible online:

   ```
   git push
   ```

> **Note**
>
> Commit little and often. Small, clearly described commits are easier to review, easier to revert if something goes wrong, and give you a much more useful history to look back on than one huge commit at the deadline.

3.3 Viewing online website

Once your commit reaches the default branch, the CI/CD pipeline rebuilds and republishes the website (and the PDF) automatically - there's nothing extra to trigger.

> **The first build takes longer than you'd expect**
>
> Every build installs the whole toolchain from scratch - Node.js, Chrome, Pandoc, the Python environment - so even a routine rebuild takes several minutes, and the very first one on a fresh project can easily run into the mid-teens. A blank page or a 404 on your first visit almost always means the build simply hasn't finished yet, not that something is broken.
>
> Check first, rather than refreshing a page that hasn't been built yet: **Build > Pipelines** in the sidebar on GitLab, or the **Actions** tab on

GitHub. A running pipeline or workflow shows a spinner or a yellow dot; wait for it to turn green.

Find your published site

GitLab

The simplest way to find your site is from the project itself, rather than working out the URL by hand: open your project on the GitLab website and look for the **GitLab Pages** link, shown on the project overview page once Pages has deployed at least once (also always available under **Deploy > Pages** in the sidebar). Click it.

1. The first time you visit, GitLab prompts you to authorise GitLab Pages access to your project:

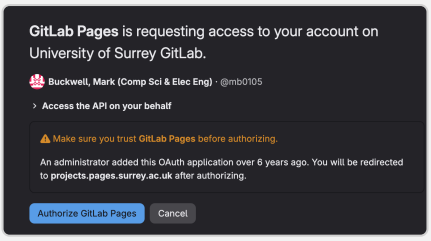


Figure 3.4. Authorise GitLab Pages

2. Your browser redirects to a URL with an extra, unique key added, such as https://prodockit-template-4f75ad.pages.surrey.ac.uk/. This confirms that you (specifically, someone with access to the underlying GitLab project) can view the page - GitLab Pages sites aren't public by default.

This works the same way on the University of Surrey GitLab and on gitlab.com or another self-hosted instance.

Working out the address yourself

If you'd rather not click through, most GitLab Pages addresses follow the form `https://` *namespace* `.gitlab.io/` *repository-name*, though a self-hosted instance may use its own domain - check **Settings > Pages** on your project for the exact one.

GitHub

1. Go to your GitHub Pages address, in the form `https://` *username* `.github.io/` *repository-name*. This template's own site is at https://buckwem.github.io/prodockit-template.
2. Unlike GitLab Pages, GitHub Pages sites are publicly accessible by default, even when the source repository is private - so no separate authorisation step is normally needed to view a GitHub Pages site once it's built.
3. If your organisation has restricted Pages visibility (available on GitHub Enterprise), GitHub will ask you to sign in with an account that has access to the repository before the site loads.

3.4 Build the PDF

`docs/site_documentation.pdf` isn't built by `zensical serve` or `zensical build` - it has its own build command, `prodockit pdf`, provided by the prodockit package on top of Zensical (see Customise PDF generation for how to customise its output, and Customise build for how the two builds fit together; this section just covers running it).

3.4.1 Building it manually

1. Make sure you've installed the PDF build's dependencies - see Install Python and Zensical for `requirements.txt`, and Additional tooling if your document uses Mermaid diagrams or maths and you need the optional `tools/mermaid` / `tools/mathjax` Node packages too.
2. Open a terminal with your virtual environment active, in your project's root directory.
3. Run the build command:

   ```
   prodockit pdf
   ```

   This can take a little while, especially the first time - it's converting every page into a single PDF, rendering any diagrams and maths along the way.
4. If your project wants a source bundle too - see Source-code bundling - build it with a second command:

   ```
   prodockit source-bundle
   ```

   This writes `docs/source_bundle.pdf` directly - nothing to copy anywhere, and refreshing your browser is enough for the website's own **Source** download button to find it in your local preview.

5. Once it finishes, open `docs/site_documentation.pdf` (in the `docs` folder) to check the result - and `docs/source_bundle.pdf`, if you built one.

Run this again after any change you want reflected in the PDF - it always rebuilds the whole document from scratch, so there's no separate "clean" step needed for it, unlike the website build below.

3.4.2 Automated builds

Both `.gitlab-ci.yml` and `.github/workflows/docs.yml` run this exact sequence automatically on every push to your default branch:

```
prodockit pdf
zensical build --clean --strict
```

`prodockit pdf` runs first so `docs/site_documentation.pdf` exists before Zensical builds the site - that's what makes the "Download PDF" button on the cover page work, since the published website includes the PDF as part of itself. `zensical build --clean --strict` then builds the site into the `public/` directory (set by `site_dir` in `zensical.toml`), which GitLab Pages or GitHub Pages then publishes. `--strict` turns validation warnings such as broken internal links or missing anchors into build failures, so a faulty site is not published. See Clean build in Troubleshooting if you need to force a fresh website build locally.

3.5 Managing branches and issues

Once you're comfortable with the basic commit-and-push cycle, branches and issues give you two extra tools for organising bigger or shared pieces of work: a branch isolates a change until it's ready, and an issue records what needs doing, with the two linked together.

3.5.1 Working with branches

A branch is a parallel, isolated copy of your files where you can work without affecting the "real", published version until you're ready. For anything more than a small tweak - a new section, a bigger restructure - it's worth developing it on its own branch rather than directly on your default branch (usually `main`). That keeps `main` (and therefore the published website and

PDF) stable while you're mid-change, and makes an unfinished idea easy to abandon without cleaning up half-done edits.

Create a new branch

Visual Studio Code

1. Click the branch name in the bottom-left of the status bar (it normally reads `main`).
2. Select **Create new branch...** and give it a short, descriptive name (for example, `add-section-3`).
3. Visual Studio Code switches you onto the new branch. Edit, save, and commit as usual (see Synchronise your updates) - your commits go onto this branch, not `main`.
4. The first time you press **Sync Changes**, Visual Studio Code offers to **Publish Branch** instead - accept this to push the new branch to GitLab or GitHub.

Command line

1. Create and switch to a new branch in one step:

   ```
   git switch -c add-section-3
   ```

2. Commit as usual (see Synchronise your updates) - your commits go onto this branch, not `main`.

3. Push it, telling Git to track this new branch on the remote the first time:

   ```
   git push -u origin add-section-3
   ```

   After that first push, a plain `git push` is enough.

3.5.2 Merging your branch back

Once you're happy with the branch, bring it into your default branch so it's published.

Merge your branch

Merge/pull request (recommended)

1. Open your project on GitLab or GitHub in a browser.
2. Open a merge request (GitLab) or pull request (GitHub) from your branch into `main` - both platforms show a prompt for this as soon as you push a new branch, or you can start one from the **Merge requests**/**Pull requests** section of the sidebar.
3. This gives you, or a collaborator, a chance to review the diff before it goes live.
4. Once you're happy, click the **Merge** button on the merge/pull request page - GitLab or GitHub does the rest.

Merge locally

If you're working alone and don't need a review step first:

1. Switch to your default branch:

```
git switch main
```

2. Pull down the latest version, in case anything's changed since you branched:

```
git pull
```

3. Merge your branch into it:

```
git merge add-section-3
```

4. Push the result:

```
git push
```

Either way, once the merge reaches `main`, the CI/CD pipeline rebuilds and

republishes the website and PDF automatically, the same as any other push to `main`.

> **Tip**
>
> Delete the branch once you've merged it - neither GitLab nor GitHub need it anymore, and it keeps your branch list tidy. Both offer a **Delete branch** button right after you merge a merge request or pull request.

3.5.3 Recording issues and linking them to a branch

Issues are GitLab's and GitHub's built-in way to track things to do - a missing section, a diagram to add, a typo to fix - separately from the writing itself. They're especially useful once more than one person is working on the same report, or if you just want a running to-do list attached to the project instead of a separate document.

1. Open the **Issues** section in the left-hand sidebar of your project on the website, and select **New issue**.
2. Give it a short title (for example, "Add diagram to section 2") and, optionally, a longer description of what's needed.

Both platforms let you create a branch directly from an issue, which links the two together from the start:

- On GitLab, open the issue and use the **Create merge request** button (or the dropdown next to it, for **Create branch** only). This creates a branch named after the issue (for example `12-add-diagram-to-section-2`) and links it back to the issue automatically.
- On GitHub, open the issue and, in the right-hand sidebar under **Development**, select **Create a branch**. This creates a branch linked to the issue, and offers to check it out for you.

If you've already created your branch by hand instead (see Working with branches), you can still link it to an issue by mentioning the issue number in a commit message:

```
git commit -m "Add diagram to section 2 (#12)"
```

Using `Closes #12`, `Fixes #12`, or `Resolves #12` instead of just `#12` - in the commit message, or in the merge/pull request description - automatically closes that issue as soon as the commit reaches your default branch.

3.6 Trouble shooting

Common problems you might hit while working on your document, and how to fix them.

3.6.1 `prodockit` or `zensical` is not recognized

```
prodockit : The term 'prodockit' is not recognized as the name of a
cmdlet,
function, script file, or operable program.
```

Both commands are installed *inside* the virtual environment, not system-wide, so they only exist in a terminal where it is active. A new terminal window never has it - activation lasts for that window only.

Activate the virtual environment

macOS

```
cd path/to/your-project
source .venv/bin/activate
```

Windows

```
cd C:\path\to\your-project
.\.venv\Scripts\Activate.ps1
```

Linux (Ubuntu)

```
cd path/to/your-project
source .venv/bin/activate
```

The prompt gains a `(.venv)` prefix when it works. This bites most often after a step that told you to close and reopen your terminal to pick up a `PATH` change - the new window has lost the virtual environment as well.

If activating makes no difference, the virtual environment itself may be in the wrong place: `.venv` created somewhere other than your project folder still activates perfectly happily. `pwd` tells you where you are.

3.6.2 Local preview isn't updating

If you save a change and the browser doesn't refresh, or the page looks stuck:

1. Do a hard refresh in the browser first (`Ctrl+Shift+R` on Windows/Linux, `Cmd+Shift+R` on macOS) - this bypasses the browser's own cache, which is a more common culprit than Zensical itself.
2. If that doesn't help, stop the server (`Ctrl+C` in its terminal) and start it again:

   ```
   zensical serve
   ```

3. Still stuck? Check the terminal `zensical serve` is running in - a build error there (for example, invalid TOML in `zensical.toml`, or a broken link) stops it rebuilding, and it'll usually tell you exactly which file and line to look at.

3.6.3 Clean build

The `--clean` flag on `zensical build --clean --strict` deletes the previous contents of `public/` before rebuilding, so pages you've since renamed or removed don't linger in the published site. `--strict` also makes validation warnings fail the build. Both CI pipelines use both flags.

To do the same locally - useful if the local `public/` folder looks out of date or you suspect a stale file is causing an issue - delete it yourself first:

```
rm -rf public
prodockit pdf
zensical build --clean --strict
```

3.6.4 Numbered lists reset to "1."

If a numbered list in your Markdown restarts at "1." partway through instead of continuing (for example after a code block, admonition, or tab), it's almost always an indentation problem - Zensical (and Pandoc, for the PDF) only treat content as *continuing* the list item if it's indented to match. See Lists within lists for the exact rule to follow.

3.6.5 PDF build fails

If `prodockit pdf` errors out or produces a PDF missing content:

1. Check the error message in the terminal - it usually names the file and the problem directly, and anything the underlying tool printed appears beneath it.

2. Make sure you've installed the dependencies from `requirements.txt` in your active virtual environment (see Install Python and Zensical).
3. If your document uses Mermaid diagrams or maths, make sure you've installed the optional Node tooling too (see Additional tooling) - without it, the build silently skips those elements rather than raising an error.

3.6.6 WeasyPrint cannot start (status 43)

A specific case of the above, worth its own entry because the number is the only clue and it points somewhere unexpected:

```
Building PDF from zensical.toml...
Error: pandoc exited with status 43 building 'docs/
site_documentation.pdf' (only pass)
```

Status 43 means Pandoc ran perfectly well, then handed your document to WeasyPrint, which could not start. **It is not a problem with your document.** Nothing you write in Markdown causes it, and no amount of editing your content will clear it.

WeasyPrint is not a pure Python package. It draws every page through Pango and a few related graphics libraries, and `pip install -r requirements.txt` cannot install those - they belong to your operating system rather than to Python. If they are missing, WeasyPrint stops before laying out a single page.

The give-away is in the output beneath the error, which contains a line like:

```
OSError: cannot load library 'libgobject-2.0-0'
```

Install the libraries for your platform, following the per-OS instructions already in Install Python and Zensical - `brew install pango` plus the pinned pandoc download on macOS, the MSYS2 `pacman` step on Windows, or the `apt install` line plus the pinned pandoc `.deb` on Linux. If you followed that page when setting up, you already ran this and something else is missing; re-running it is harmless either way.

Then check before building again:

```
python3 -c "import weasyprint; print(weasyprint.__version__)"
```

A version number means you are ready. The same long error means the libraries still are not being found.

> **On macOS, if it still fails after installing Pango**
>
> macOS only looks in Homebrew's folder for libraries if the Python you are using was itself installed by Homebrew. That is why Install tooling

> has you create the virtual environment with the full path `/opt/homebrew/bin/python3` rather than a plain `python3`. If you created it another way, the quickest fix is to delete `.venv` and make it again, following those steps exactly.

3.6.7 Mermaid render fails with a browser process error

Another specific case of "PDF build fails" above, this one on Linux:

```
    Mermaid render failed for diagram 1:
Error: Failed to launch the browser process:  Code: 2

stderr:
.../chrome-headless-shell: 1: ELF: not found
.../chrome-headless-shell: 3: Syntax error: Unterminated quoted
string
```

That garbled output is the giveaway: it's a binary being misread as a shell script. `npm ci --prefix tools/mermaid` downloads Chrome for whichever architecture Puppeteer defaults to, and on some Linux setups - most often ARM64 (an Apple Silicon Linux VM, an AWS Graviton instance, a Raspberry Pi) - that does not match your CPU. Linux can't execute a binary built for the wrong architecture, falls back to interpreting it as a script, and the first few bytes of a Chrome binary aren't valid shell - hence `ELF: not found` and an "unterminated quoted string" a few lines further down.

Install a native Chromium and point Puppeteer at it instead of its own download, following the Linux step in Install the two toolchains. If you followed that page when setting up, you already ran this and something else is wrong; re-running it is harmless either way. Then run `prodockit pdf` again.

3.6.8 Published site shows old content or a 404

1. Check the pipeline (GitLab **CI/CD > Pipelines**) or workflow (GitHub **Actions** tab) actually ran, and succeeded, for your latest commit - if it's still running, or failed, the old version stays published.
2. Confirm your change actually reached the default branch (`main`) - a commit sitting on a feature branch, or a merge/pull request you haven't merged yet, never triggers a rebuild. See Managing branches and issues.
3. Hard refresh the published page (`Ctrl+Shift+R` / `Cmd+Shift+R`) - your browser can cache the old version just as easily as it caches the local preview.
4. On GitHub specifically, if the workflow fails with `Get Pages site failed... Not Found`, GitHub Pages hasn't been switched on for the repository yet. Go to **Settings > Pages** and change **Build and deployment > Source** from **Deploy from a branch** to **GitHub Actions**, then re-run the failed workflow. This is a one-off step after

cloning a fresh copy of the repository into a new GitHub account - see the GitHub Pages step in Cloning the prodockit-template in Install tooling.

5. On GitLab specifically, if the pipeline succeeds but no Pages site ever appears, check that the **Pages** feature itself hasn't been disabled for the project: **Settings > General > Visibility, project features, permissions**, and make sure **Pages** is toggled on. Unlike GitHub, GitLab doesn't need a separate "source" setting - Pages deploys automatically from the `pages` job in `.gitlab-ci.yml` once the feature is enabled, which it is by default.

3.7 Release your report

Before you submit your report, remove the "Start Here" stub page so it isn't part of what you hand in - see Start here in your own copy of the template for exactly what to comment out in `zensical.toml` and what to delete.

> **Info**
>
> Once you've removed the stub from your own report, you can still come back to this guidance any time on the independent prodockit User Guide site.

3.8 Where to go next

Continue to Markdown basics and Zensical basics to learn the syntax you'll actually use to write your document. Once you're comfortable writing content, come back to these later chapters when you need them:

- Customisation - branding, the cover page, PDF layout, and the document's directory structure.
- Additional tooling - GitLab/GitHub VS Code extensions, and Vale for spelling/grammar/style checking.

4. Markdown basics

Markdown is a lightweight markup language that enables you to format plain text using a simple syntax. It's easy to read and easy to write, eventually converting into structurally valid HTML, and is widely used for documentation, readme files, and content management systems. It enables you to focus on writing without worrying about complex formatting, making it an ideal choice for collaborative documentation projects.

It's text-based, meaning you can use a text editor to create and edit Markdown files. This makes it highly portable and compatible with version control systems like Git, enabling collaborative editing and tracking of changes over time. Plain text files with the `.md` extension store the Markdown. You can then share, version, and convert these files into HTML for web publishing.

Below is a summary of the most common formatting elements you'll use in a `.md` file.

> **Zensical Markdown**
>
> Zensical uses a flavour of Markdown called Python Markdown with some extensions. This ensures that your Markdown files are compatible with a wide range of tools and platforms, while also providing additional features for enhanced formatting and functionality. There are some differences between Zensical Markdown and other flavours of Markdown, so it's important to refer to the Zensical documentation for details.

> **Markdown Live Preview**
>
> You can use the Markdown Live Preview website to see how your Markdown will look when rendered. This is a great way to check your formatting and make adjustments as needed.

4.1 Headings

Add hash signs (#) before your text to create a heading. The number of hashes corresponds to the heading level.

```
# H1 Heading
## H2 Heading
### H3 Heading
#### H4 Heading
```

```
##### H5 Heading
###### H6 Heading
```

The `toc` extension automatically turns every heading into a linkable anchor (the `¶` symbol you can see next to each heading on this page), and generates the sidebar and table of contents from them. The `attr_list` extension lets you override the generated anchor or add a CSS class, by adding an attribute block after the heading text, for example `## Heading {: #custom-id }`. This lets you attr_list attach IDs, classes, and other HTML attributes almost anywhere in your Markdown.

> **Warning**
>
> As covered in Navigation structure in Customisation, each page in this template can contain only one heading 1 (`#`) - it's what drives the automatic chapter/section numbering (e.g. "9.1") across the whole document. Start a new page instead of adding a second heading 1 to this one.

4.2 Text formatting

You can make text bold, italic, or both to add emphasis without needing complex menus.

```
**bold text**
*italic text*
***bold and italic***
~~strikethrough~~
`inline code`
```

The `pymdownx.betterem` extension handles bold and italic emphasis, and is more consistent about nested and mixed emphasis (for example `**bold _and italic_**`) than plain Python Markdown. Strikethrough isn't part of core Markdown at all - the `pymdownx.tilde` extension provides it here. That same extension also enables subscript (`H~2~O`), and the paired `pymdownx.caret` extension enables superscript (`A^T^A`) and underline (`^^text^^`). See Formatting in Zensical basics for these and other extended styles, such as highlighting text and keyboard keys.

4.3 Links and images

The syntax for these is similar. Just add an exclamation mark at the beginning for an image.

```
[Link text](https://example.com)
[Link with title](https://example.com "Hover title")
```

```
[Reference-style link][example-ref]
![Alt text](image.jpg)
![Image with title](image.jpg "Image title")

[example-ref]: https://example.com "Hover title"
```

The `attr_list` extension lets you attach HTML attributes to a link or image by adding a `{: ... }` block straight after it, with no space:

```
[Link text](https://example.com){: .external-link }
```

This template uses that same syntax with a `target="_blank"` attribute throughout, to make external links (like the ones on this page) open in a new browser tab on the website. the PDF build strips those attributes back out, since "open in a new tab" has no meaning in a printed document.

The `pymdownx.magiclink` extension also auto-links bare URLs (`https://example.com` becomes a clickable link with no `[]()` needed) and recognises shorthand references to GitHub/GitLab issues, pull requests, and commits.

4.4 Lists

Markdown handles ordered (numbered), unordered (bulleted), and definition lists.

4.4.1 Unordered lists

Use a minus sign (-), asterisk (*), or plus sign (+).

```
- Item 1
- Item 2
    - Nested item
```

4.4.2 Ordered lists

Simply use numbers followed by a period. Python Markdown renumbers the list for you based on the *first* number used, so `1.` for every item (or repeating the same number) is a common way to avoid manually renumbering items as you edit.

```
1. First item
2. Second item
3. Third item
```

4.4.3 Definition lists

The `def_list` extension adds definition lists: a term on its own line, followed by one or more indented lines starting with a colon.

```
Term
:   Definition of the term, indented under it.

Second term
:   First definition.
:   A second definition for the same term.
```

4.5 Code blocks

Markdown is a favourite for developers because of how it handles code snippets. This template uses the `pymdownx.superfences` extension for fenced code blocks (in place of Python Markdown's more limited built-in `fenced_code`), together with `pymdownx.highlight` and `pymdownx.inlinehilite` for syntax highlighting.

- Inline code: wrap text in backticks: `code`.
- Fenced code blocks: wrap multiple lines in "fences" using three backticks (```). Add the language name straight after the opening fence for syntax highlighting.

````
```javascript
function hello() {
  console.log("Hello, world!");
}
```
````

`pymdownx.superfences` also lets you nest fenced code blocks inside other Markdown structures such as lists and admonitions, and supports custom fence types - the Mermaid diagrams in Diagrams are a fenced ```` ```mermaid ```` block rather than plain code. For line highlighting, titled code blocks, and inline code with syntax highlighting, see Code blocks in Zensical basics.

4.6 Tables

Use pipes `|` and hyphens `-` to create tables. The `tables` extension is what enables this - it isn't part of core Python Markdown. Add colons to the separator row to control column alignment.

```
| Left-aligned | Centred | Right-aligned |
|:-------------|:-------:|--------------:|
| Row 1        | Data    | Data          |
| Row 2        | Data    | Data          |
```

4.7 Horizontal rule

Put three or more hyphens, asterisks, or underscores on their own line, surrounded by blank lines, to create a thematic break:

```
---
```

```
***
```

```
___
```

4.8 Task lists

Task lists aren't part of core Python Markdown either - they're enabled by the `pymdownx.tasklist` extension, configured in this template to render as clickable checkboxes rather than plain `[x]` / `[ ]` text.

```
- [x] Completed task
- [ ] Incomplete task
- [ ] Another task
```

4.9 Blockquotes

Use the `>` symbol before the text to create a callout or quote. Nest additional `>` symbols to quote within a quote.

```
> This is a blockquote
> Multiple lines
>> Nested quote
```

For structured callouts with an icon and title (notes, warnings, tips), use an admonition instead - see Admonitions in Zensical basics.

4.10 Quick tips

- **Line breaks:** To create a line break without starting a new paragraph, end a line with two or more spaces before hitting enter.
- **Escaping characters:** If you want to show a literal character (like a `*`) without it formatting the text, use a backslash: `\*`.
- **Attributes on any element:** The `attr_list` extension used for links, images, and headings above works on most other Markdown elements too - for example adding a CSS class to a paragraph or list item with `{: .my-class }` directly after it.

4.11 Where to go next

The syntax on this page works in any Markdown file, including plain README files on GitLab or GitHub. Continue to Zensical basics for the extensions that

only work in this template's own Zensical-built pages - admonitions, content tabs, diagrams, and more.

5. Zensical basics

Zensical is the static site generator that powers this template: it turns the Markdown files under `docs/` into the website you're reading now, and (via `prodockit pdf`) into the single-file PDF version of your document. It reads its configuration from `zensical.toml`, extends Markdown with the authoring features shown below (admonitions, tabs, diagrams, maths, and more), and lets you preview your changes locally with `zensical serve` before publishing.

This page is a quick reference for the Markdown extensions you're most likely to use while writing your document, each with a live example. For the underlying, general-purpose Markdown syntax these extensions build on (headings, links, bold/italic text, and so on), see Markdown basics. For full documentation on Zensical itself, visit zensical.org.

5.1 Commands

Zensical provides a command line interface (CLI) to create, build, and serve your documentation. The following commands are available:

- `zensical new` - Create a new project
- `zensical serve` - Start local web server
- `zensical build` - Build your site

5.2 Examples

Some examples of Zensical syntax are below. For full documentation visit zensical.org.

5.2.1 Lists within lists

Markdown supports nested lists by indenting the inner list by four spaces. This is an implementation-specific feature of Python Markdown used by Zensical, and isn't part of the original Markdown specification.

> **⚠ The Four Space Rule**
>
> If you are nesting Tabs, Admonitions, or Code Blocks inside a list, you must indent by exactly 4 spaces. If your ordered list numbering resets to "1", check your indentation! Further background information is on the Zensical authoring section.

5.2.2 Admonitions

Zensical supports admonitions, that highlight blocks of content to draw

attention to important information. Admonitions are available for notes, warnings, tips, and more. For further details, go to the admonitions documentation.

> **Note**
>
> This is a **note** admonition. Use it to provide helpful information.

> **Warning**
>
> This is a **warning** admonition. Be careful!

5.2.3 Details

Zensical supports collapsible blocks using the `???` syntax. This is useful for hiding content until the user clicks to expand it. For further details, go to the admonitions collapsible blocks documentation.

Click to expand for more info

This content is hidden until you click to expand it. Great for FAQs or long explanations.

5.3 Code blocks

Zensical supports fenced code blocks with syntax highlighting. You can specify the language for syntax highlighting by adding the language name after the opening backticks. For further details, go to the code blocks documentation.

Code blocks

```
def greet(name):
    printf("Hello, {name}!") # (1)!

greet("Python")
```

1. Go to code annotations documentation

   Code annotations enable attaching of notes to lines of code.

You can also highlight code inline: `print("Hello, Python!")`.

5.4 Content tabs

Zensical supports content tabs, which enables you to present different content in the same space. This is useful for showing code examples in multiple programming languages. For further details, go to the content tabs documentation.

Python

```
print("Hello from Python!")
```

Rust

```
println!("Hello from Rust!");
```

5.5 Images

Zensical supports Markdown image syntax using the `<figure>` tag to add captions. For further details, go to the images documentation.

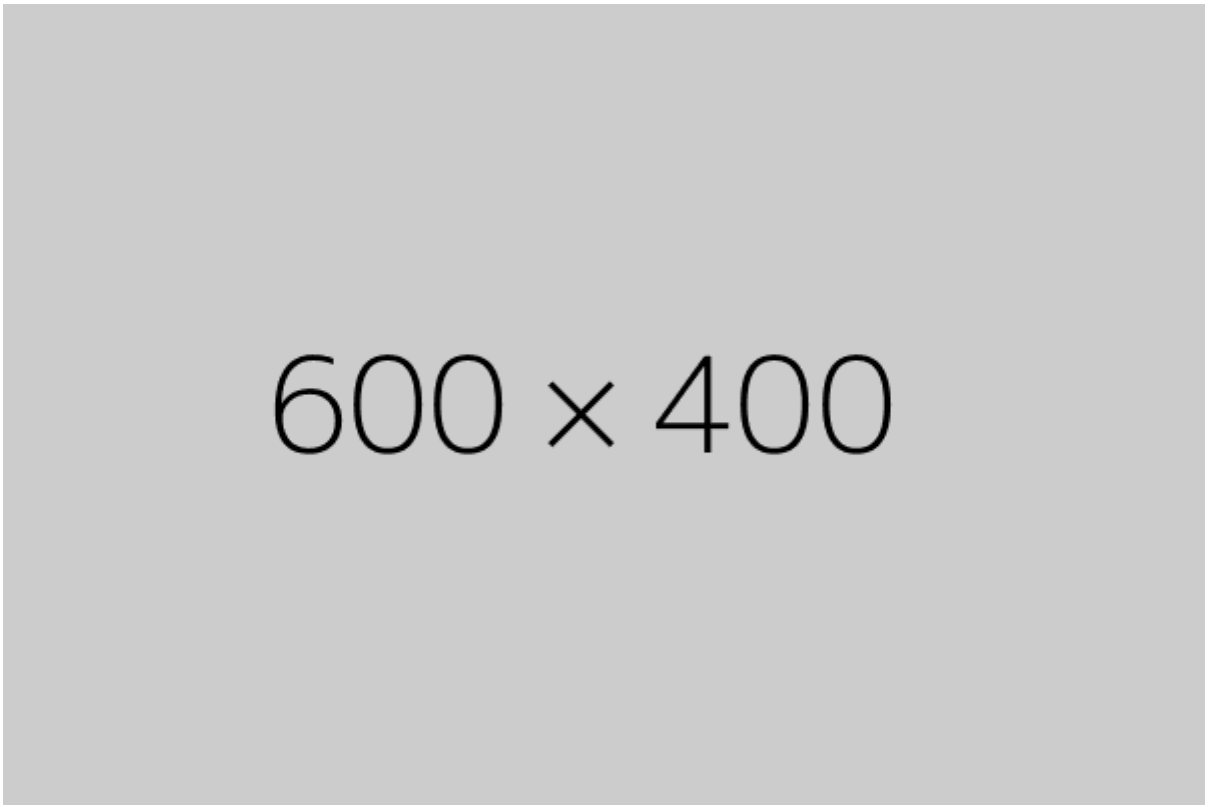


Image caption

This template also enables the `pymdownx.blocks.caption` extension, an alternative way to caption an image using a `/// caption ... ///` block straight after it, rather than wrapping it in `<figure>`/`<figcaption>` tags:

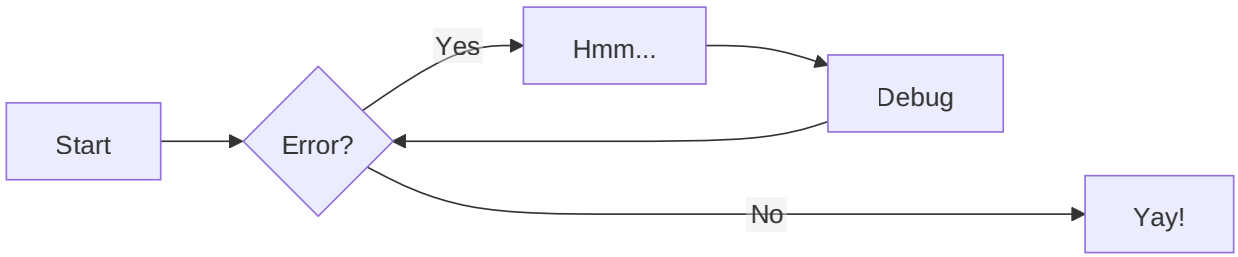
```
![Image title](https://dummyimage.com/600x400/){ width="300" }
/// caption
Image caption
///
```

See Captions in Customise document content for more detail, including how this same syntax also captions **tables**, and how this template handles it in the PDF.

5.6 Diagrams

Zensical supports Mermaid diagrams. You can create flowcharts, sequence diagrams, and more. For further details, go to the diagrams documentation.

> **Note**
>
> If you're on the COMM058 Architectural Thinking for Security module, it doesn't use any of these documentation types. It's better that you use draw.io to create your diagrams and export them as images to include in your documentation. Use the downloadable version of draw.io, not the web version, as it's much easier to edit. Also, if you use draw.io in your working life, your company may have a policy against using cloud services unless they're a paid, approved service for hosting confidential company data.



5.7 Footnotes

Zensical supports footnotes, which enables you to add references or additional information without cluttering the main text. You can create a footnote by using the `[^1]` syntax. For further details, go to the footnotes documentation.

Here's a sentence with a footnote.[1]

1. This is the footnote.

Hover it, to see a tooltip.

5.8 Formatting

Zensical supports various formatting options, including bold, italics, and strikethrough. You can also create headings, blockquotes, and horizontal rules. For further details, go to the formatting documentation.

- This was marked (highlight)
- This was inserted (underline)
- ~~This was deleted (strikethrough)~~
- H_2O
- A^TA
- `Ctrl`+`Alt`+`Del`

5.9 Icons, emojis

Zensical supports icons and emojis. You can use the `:icon-name:` syntax to add icons from the Lucide icon set, or use standard emoji codes. For further details, go to the icons and emojis documentation.

- ✨ `:sparkles:`
- 🚀 `:rocket:`
- 🎉 `:tada:`
- 📝 `:memo:`
- 👀 `:eyes:`

5.10 Maths

Zensical supports mathematical notation using MathJax. You can write inline math using the `$...$` syntax, and display math using the `$$...$$` syntax. For further details, go to the math documentation.

$$\cos x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} x^{2k}$$

Available on every page

MathJax is loaded site-wide from `extra_javascript` in `zensical.toml` - see Extra CSS and JavaScript - so you can write a formula on any page without adding anything to it. The copy it loads is installed for this project rather than fetched from a CDN, so formulas render with no external request and work offline - see Install the diagram and maths tooling for the one-time setup this needs.

5.11 Task lists

Zensical supports task lists, which allow you to create checklists with checkboxes. You can create a task list by using the `- [ ]` syntax for an unchecked item and `- [x]` for a checked item. For further details, go to the task lists documentation.

- [x] Install Zensical
- [x] Configure `zensical.toml`
- [x] Write amazing documentation
- [ ] Deploy anywhere

5.12 Tooltips

Zensical supports tooltips, which allow you to add additional information that appears when the user hovers over a specific element. You can create a tooltip by using the `[text][example]` syntax. For further details, go to the tooltips documentation.

Hover over this text

5.13 Where to go next

Continue to Customisation to change this template's branding, restructure your document's pages, and customise the cover page and PDF layout.

6. Customisation

A small number of files control almost every visible part of this template - the website, the cover page, and the generated PDF: `zensical.toml` for configuration, `macros.py` for build-time logic, and `docs/stylesheets/extra.css` / `print.css` for appearance. This section walks through each of these in turn: customising the website's branding and behaviour, changing your document's page structure, customising the cover page, and adjusting the PDF's page layout. It ends with a full map of the template's directory structure, so you know where everything lives.

> **prodockit-specific features**
>
> Where Zensical basics is a quick reference for Zensical's own general-purpose Markdown extensions (the same ones you'd find on any Zensical site), everything on this page is specific to this template: features `macros.py` and the prodockit package add on top of Zensical (Surrey/generic branding, PDF-only/web-only content, and so on) that only exist here, not in a stock Zensical project. For the prodockit extensions that number, cross-reference, cite, and index your document's actual *content*, see Customise document content instead.

6.1 Customise the web site

Most website-wide settings live in `zensical.toml`, in the `[project]` and `[project.theme]` sections. The sections below describe the most commonly customised ones; see the Zensical setup documentation for the full reference.

6.1.1 Site logo

If the documentation website is part of the university's GitLab service, or the website's location falls under the University of Surrey domain, the build automatically changes the site logo to the University of Surrey logo. Otherwise, the site logo uses the default logos in the `docs/assets/` directory. You can change the default logo by replacing the existing default logo files with your own logo files named `logo_default_black.png` and `logo_default_white.png`.

Every build, and `zensical serve`, copies either the Surrey pair or your default pair over `docs/assets/logo_black.png` / `logo_white.png` - the two files `extra.css` actually references for the light/dark logo swap. Don't edit `logo_black.png` / `logo_white.png` directly, since the next build overwrites them.

6.1.2 Site metadata

`site_name`, `site_description`, `site_author`, and `site_url` (all in `[project]` in `zensical.toml`) set the browser tab title, the HTML description used by search engines, the HTML author metadata, and the canonical site URL. `site_name` is also shown on the cover page.

6.1.3 Copyright

`copyright` in `[project]` sets the text shown in the website's footer. It can contain an HTML fragment, for example an `&copy;` entity:

```
copyright = """
Copyright &copy; 2026 Your Name
"""
```

The PDF build reuses this same setting for its own running footer - see Page footer.

Set `pdf_copyright` in `[project.extra]` to override this text for the PDF's own footer only, leaving the website's footer untouched:

```
[project.extra]
pdf_copyright = "Copyright &copy; 2026 Your Name.<br>Made with real,
clickable links here too."
```

Unlike `copyright`, this accepts a real HTML fragment in the PDF - a literal `<br>` forces a line break, and `<a>` links render as actual clickable links rather than flattened plain text. Leave it unset and the PDF simply reuses `copyright` as normal.

6.1.4 Repository link

`repo_url` and `repo_name` in `[project]` show a link to your repository, with an icon and short name, near the top of the sidebar. For example, on GitHub:

```
repo_url = "https://github.com/buckwem/prodockit-template"
repo_name = "prodockit-template"
```

Or on the University of Surrey GitLab:

```
repo_url = "https://gitlab.surrey.ac.uk/mb0105/prodockit-template"
repo_name = "prodockit-template"
```

You set the icon shown next to it separately - see `theme.icon.repo` under Icons.

> **Note**
>
> This is unrelated to the `{{ repo_url }}` macro variable used on the cover page (see Word count and repository link). The build computes that value independently from your local Git remote, rather than reading it from this `repo_url` setting, though in practice they'll usually point to the same place.

6.1.5 Favicon

Set `favicon` in `[project.theme]` to a path (relative to `docs_dir`) for your own browser-tab icon:

```
[project.theme]
favicon = "images/favicon.png"
```

Left unset, Zensical uses its default favicon.

6.1.6 Colour Scheme

The two `[[project.theme.palette]]` blocks in `zensical.toml` configure the light and dark themes and the toggle button between them:

```
[[project.theme.palette]]
media = "(prefers-color-scheme: light)"
scheme = "default"
toggle.icon = "lucide/sun"
toggle.name = "Switch to dark mode"

[[project.theme.palette]]
media = "(prefers-color-scheme: dark)"
scheme = "slate"
toggle.icon = "lucide/moon"
toggle.name = "Switch to light mode"
```

`scheme` selects Zensical's built-in `default` (light) or `slate` (dark) palette; `toggle.icon` / `toggle.name` set the icon and tooltip for the button used to switch between them.

6.1.7 Page Heading

The website's header background image swaps between light and dark mode too, in `docs/stylesheets/extra.css`:

```
[data-md-color-scheme="default"] .md-header {
  background: ... url("../assets/header-background.jpg");
}
```

```
[data-md-color-scheme="slate"] .md-header {
  background: ... url("../assets/header-background-dark.jpg");
}
```

Replace `header-background.jpg` / `header-background-dark.jpg` in `docs/assets/` with your own images, or switch to a plain colour gradient instead - `extra.css` has a commented-out example of this directly below.

6.1.8 Fonts

`[project.theme.font]` in `zensical.toml` sets the fonts loaded from Google Fonts, used across the website:

```
[project.theme.font]
text = "Inter"
code = "Jetbrains Mono"
```

Use `text` for body copy and headings, and `code` for code blocks and inline code. Both default to Inter and JetBrains Mono if you leave this section unset. The PDF build reuses this same setting - see Customise PDF generation.

6.1.9 Icons

`[project.theme.icon]` sets the icons used for the edit/view/repository buttons in the header, and `[project.theme.icon.admonition]` sets the icon shown for each admonition type (`note`, `warning`, `tip`, and so on):

```
[project.theme.icon]
edit = "lucide/pencil"
view = "lucide/eye"
repo = "fontawesome/brands/gitlab"

[project.theme.icon.admonition]
note = "fontawesome/solid/note-sticky"
warning = "fontawesome/solid/triangle-exclamation"
```

You can use any Lucide or FontAwesome icon name.

6.1.10 Navigation and feature toggles

The `features` list in `[project.theme]` turns individual website behaviours on or off - instant navigation, sticky tabs, search highlighting, the back-to-top button, and around twenty others. `zensical.toml` lists each one with a link to its own documentation in a comment directly above it; comment a line out to disable that feature, or uncomment one of the already-listed-but-disabled options to enable it.

6.1.11 Extra CSS and JavaScript

`extra_css` and `extra_javascript` in `[project]` list additional stylesheets and scripts to load, with paths relative to `docs_dir`:

```
extra_css = ["stylesheets/extra.css"]
extra_javascript = [
  "javascripts/mathjax.js",                    # config - must
load first
  "javascripts/vendor/mathjax/tex-svg-full.js",   # the library
itself
  "javascripts/extra.js"
]
```

The config has to come before the library it configures: MathJax reads `window.MathJax` once, at startup, so a config file listed after it is ignored - see Maths for what happens without one.

> **Prefer an installed copy over a CDN link**
>
> An entry can be a full URL as well as a local path, which makes loading a library from a public CDN a one-line change. It costs more than it looks: the version usually floats, so the library can change under you with nothing recorded in your repository; the page depends on someone else's uptime; it won't work offline; and every reader's browser makes a request to a third party. This project installs its own copy of MathJax for exactly those reasons - see Maths - but doesn't *commit* it: the bundle is third-party code, and a repository is redistribution. `prodockit bootstrap` installs it on a developer's machine; CI installs it the same way for a published build.

`docs/stylesheets/extra.css` is where most of this template's own customisations live (the logo swap, header image, cover page title styles, and the `.pdf-only` / `.web-only` markers).

Set `pdf_extra_css` in `[project.extra]` to load additional stylesheets for the PDF only, in the same shape as `extra_css` above:

```
[project.extra]
pdf_extra_css = ["stylesheets/print.css"]
```

Use it for a rule that would look wrong on the live website, or one that needs to override something `extra_css` itself sets - `pdf_extra_css` stylesheets load after `extra_css`, so they win the cascade.

6.1.12 Social links

Add icons linking to your social profiles or other sites in the footer, by uncommenting and repeating this block in `[project.extra]` in `zensical.toml`:

```
[[project.extra.social]]
icon = "fontawesome/brands/github"
link = "https://github.com/user/repo"
```

6.2 Navigation structure

The `nav` list in `zensical.toml` controls how your document is broken into pages, and the order they appear in. It applies identically to the website's sidebar and the generated PDF.

`nav` (under `[project]` in `zensical.toml`) lists, in order, every page in your document and how they're grouped. Here's the template's own `nav` as delivered, for reference:

```
nav = [
  {"Cover" = [
    "index.md",
  ]},
  {"Originality" = [
    {"1. Originality & AI Use" = "originality.md"}
  ]},
  {"Assignment" = [
    {"2. Section" = "section1.md"},
    {"3. Section" = "section2.md"},
    {"4. Section" = "section3.md"},
    {"5. Section" = "section4.md"}
  ]},
  # Comment out the START HERE section before releasing your report,
as it contains instructions for the author and is not meant for the
reader of the report.
  {"START HERE" = [
    {"6. Start Here" = "starthere.md"}
  ]},
  {"Appendixes" = [
    {"Appendix A. Acronyms" = "acronyms.md"},
    {"Appendix B. Glossary" = "glossary.md"},
    {"Appendix C. References" = "references.md"}
  ]}
]
```

Each entry is either a plain path to a markdown file, or a `{"Group name" = [...]}` block nesting further entries - top-level groups become tabs, and nested groups become collapsible sections in the sidebar. This same `nav` list, walked in this same order, is also what `prodockit pdf` uses to decide which files go into the PDF and in what order - so reordering, adding, or removing an entry here changes both outputs at once.

To add a new page: create the markdown file under `docs/`, then add its path to `nav` wherever you want it to appear.

> **Warning**
>
> Each markdown file can contain only one heading 1 (`#`). Zensical numbers headings sequentially across the whole document in `nav` order, starting a new top-level number at each heading 1 - a second heading 1 in the same file breaks that numbering and confuses the table of contents. If you need another top-level heading, create a new markdown file for it and add it to `nav` instead. See Changing heading numbering in Customise document content for how that numbering itself works.

6.3 Customise front page

The cover page (`docs/index.md`) consists of a few independently customisable pieces, described below.

6.3.1 Institution branding

A Jinja conditional block wraps the cover page's logo, colours, and introductory text:

```
{% if is_surrey %}
... Surrey-branded logo and text ...
{% else %}
... your own branding and text ...
{% endif %}
```

`is_surrey` is a boolean computed once per build in `macros.py`, set to `true` if *any* of the following match:

- The build is running in GitLab CI/CD with `CI_SERVER_HOST` set to `surrey.ac.uk`.
- Your local Git repository's `origin` remote contains `surrey.ac.uk`.
- Zensical's own config (e.g. the site URL) contains `surrey.ac.uk`.

If you're not from the University of Surrey, the `{% else %}` branch is where you customise your own institution or company branding: replace `Crested Eagle Labs`, `University of the World`, and `Research programmes in Cyber Security` with your own text, and point the two `![]()` image lines at your own logo files (see Site logo above for the light/dark logo swap).

> **Tip**
>
> `is_surrey` isn't only used on the cover page - `repo_url` / `repo_name` in Repository link above switch the same way. Install tooling and the other

> guide pages show both the GitLab and GitHub paths side by side instead, since a centrally-hosted guide can't detect which one applies to any given reader.

6.3.2 PDF-only and web-only content

`.pdf-only` and `.web-only` are two general-purpose CSS marker classes. Add either to any element on any page, not just the cover page, to show it in only one output:

- `.pdf-only` - shown in the generated PDF, hidden on the live website.
- `.web-only` - shown on the live website, hidden in the generated PDF.

For static content, just add the relevant class - it looks identical either way, so hiding it from the other output is all that's needed. Computed values are different: the PDF and the website fill them in using two separate mechanisms - a `{MARKER}` placeholder substituted only during the PDF build, and a `{{ macro_variable }}` Jinja variable evaluated only by the live website - so each only works paired with its own class. The word count and repository link below are examples of this.

6.3.3 Site name

The cover page also shows your project's `site_name` (from `zensical.toml`), using `{{ site_name }}`. Unlike the marker-restricted values below, this one doesn't need a `.pdf-only` / `.web-only` pair: `prodockit pdf` substitutes that exact same text directly during the PDF build (rather than a separate `{MARKER}`), so a single line works correctly in both outputs. It appears twice in `docs/index.md`, once in each half of the `is_surrey` block, styled the same way as `module_id - module_name`:

```
<p class="title-ctr-b4">{{ site_name }}</p>
```

Delete both lines if you don't want the site name shown on the cover page.

6.3.4 Word count and repository link

Four elements on the cover page use marker classes out of the box: the automated word count, the repository link, the latest release number, and the "Download PDF" button.

Word count: `.pdf-only`, shows an automated word count of your document's content (excluding the cover page itself and the Table of Contents). To remove it from the PDF, open `docs/index.md` and delete the following line:

```
<p class="pdf-only">Word count: {WORDCOUNT}</p>
```

The PDF build replaces the `{WORDCOUNT}` marker with the actual count. If you delete the line, the PDF simply builds without a word count - you don't need to change anything else.

The count also skips any page whose front matter sets `exclude_from_word_count: true` - already set on `references.md`, `acronyms.md`, `glossary.md`, and `originality.md`, matching the common academic convention that a bibliography, acronym list, glossary, and originality/AI-use declaration don't count toward a submission's word limit. Add the same line to a page's own front matter (alongside `icon:`) to exclude any other page the same way:

```
---
icon: lucide/book-open
exclude_from_word_count: true
---
```

Repository link: `.pdf-only`, shows the fully qualified URL of your project's Git repository. To remove it from the PDF, open `docs/index.md` and delete the following line:

```
<p class="pdf-only">Repo: {REPOURL}</p>
```

The PDF build replaces the `{REPOURL}` marker with your repository's `origin` remote URL. If you delete the line, the PDF simply builds without a repository link - you don't need to change anything else.

Release number: `.pdf-only`, shows the tag of your repository's latest published GitHub or GitLab release (e.g. `v0.0.11`), so a distributed PDF can be traced back to the exact version of the source it was built from. To remove it from the PDF, open `docs/index.md` and delete the following line:

```
<p class="pdf-only">Release: {RELEASE}</p>
```

The PDF build fetches the latest release from your repository host's API and replaces the `{RELEASE}` marker with its tag. This only happens if a release has actually been published - most forks of this template never publish one, so by default the whole line is dropped rather than showing an empty "Release:" label.

To add the word count, repository link, or release tag to the website, add a line like one of the following to any page, for example next to the lines you just deleted on the cover page:

```
Word count: {{ word_count }}
Repo: {{ repo_url }}
Release: {{ release }}
```

`{{ word_count }}`, `{{ repo_url }}`, and `{{ release }}` are macro variables that Zensical makes available on every page, so you can drop any of them into any markdown file, not just the cover page. This template's own `docs/index.md` already uses `{{ release }}` for the cover page's own "Release:" line.

> **Note**
>
> The PDF and the website calculate the word count slightly differently, so it may not always match exactly. The PDF count reflects the final, built PDF content. The website count is a rough estimate across the pages that `nav` lists in `zensical.toml` (excluding the cover page).
>
> The website and the PDF also resolve the release tag differently, and can genuinely disagree: `{{ release }}` reads the latest tag reachable from your local Git history at build time, while `{RELEASE}` queries your repository host's API for the latest *published* release - a tag pushed but not yet turned into a GitHub/GitLab release shows up on the website but not in the PDF.

6.3.5 Download PDF button

`.web-only`, links to the generated PDF so website visitors can download it. It isn't shown inside the PDF itself, since that would be circular. To remove it from the website, open `docs/index.md` and delete the following line:

```
[:material-file-pdf-box: PDF](site_documentation.pdf){ .md-button
target="_blank" style="float: right; margin-left: 15px;" .web-only}
```

6.4 Customise PDF generation

Zensical only builds the website, so the PDF comes from a separate command, `prodockit pdf`, that turns the same `docs/` content into a single-file PDF via Pandoc and WeasyPrint. This used to be a `build_pdf.py` script owned by the template; it now lives in the prodockit package, so there is no per-project build code to maintain. It renders every page through the same Zensical/prodockit pipeline the website uses, then hands Pandoc the resulting HTML directly - so `\citeref{}` / `\gls{}` / `\ref{}`, admonitions, tabs, and captions all resolve exactly the same way in both outputs, with no separate PDF-side translation for any of them. It reads the same `zensical.toml` your website does, so most website customisations (site name, copyright, fonts,

and so on) apply to the PDF automatically - the sections below cover the handful of things that are PDF-specific.

For how to actually run it as part of your day-to-day writing - installing its dependencies, the `prodockit pdf` command itself, and troubleshooting a failed build - see Build the PDF in *Start editing* and Customise build; this section is about customising its output once it's already working.

`prodockit pdf` controls most of the generated PDF's page layout - the running header, the footer, the page size, and the fonts - either from `zensical.toml` settings you already use for the website, or (for page size and margins) their own PDF-only `zensical.toml` settings.

6.4.1 Page header

Every page except the cover shows a running header: your project's `site_name` (from `zensical.toml` - see Site name), left-aligned, with a divider line underneath. There's no separate PDF setting for it - editing `site_name` in `zensical.toml` updates the header everywhere, including the website.

The header also shows the current chapter title, right-aligned - starting from the first numbered heading 1, so it's blank on the cover page and the Table of Contents. It is computed automatically from each page's heading 1 (including its chapter number), so there's nothing to configure here either.

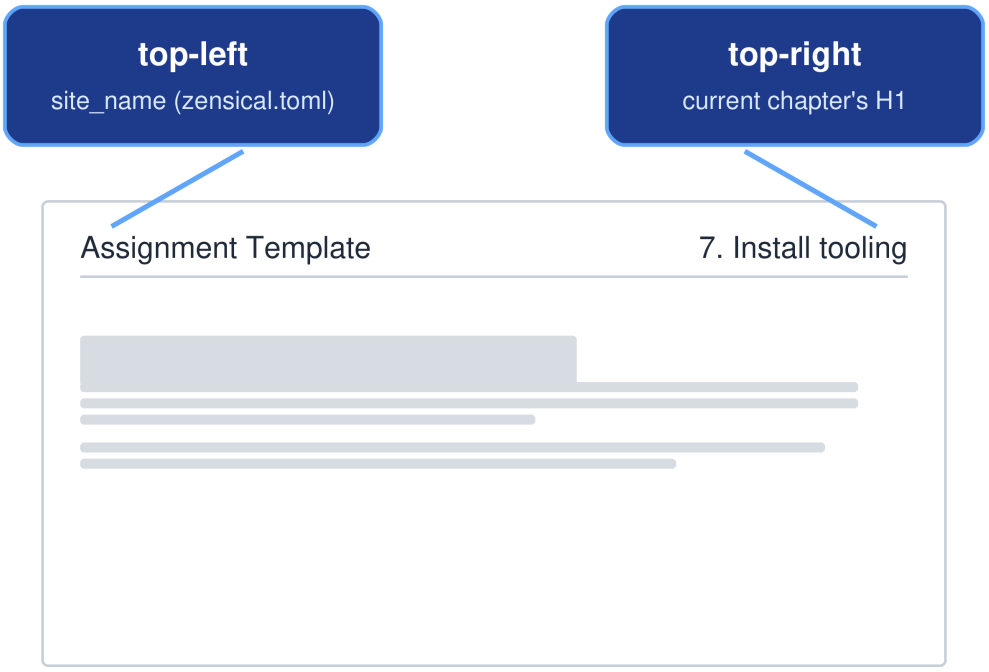


Figure 6.1. PDF page header layout

6.4.2 Page footer

Every page except the cover also shows a running footer: your `copyright` text (left-aligned - see Copyright) and a "Page X of Y" counter (right-aligned).

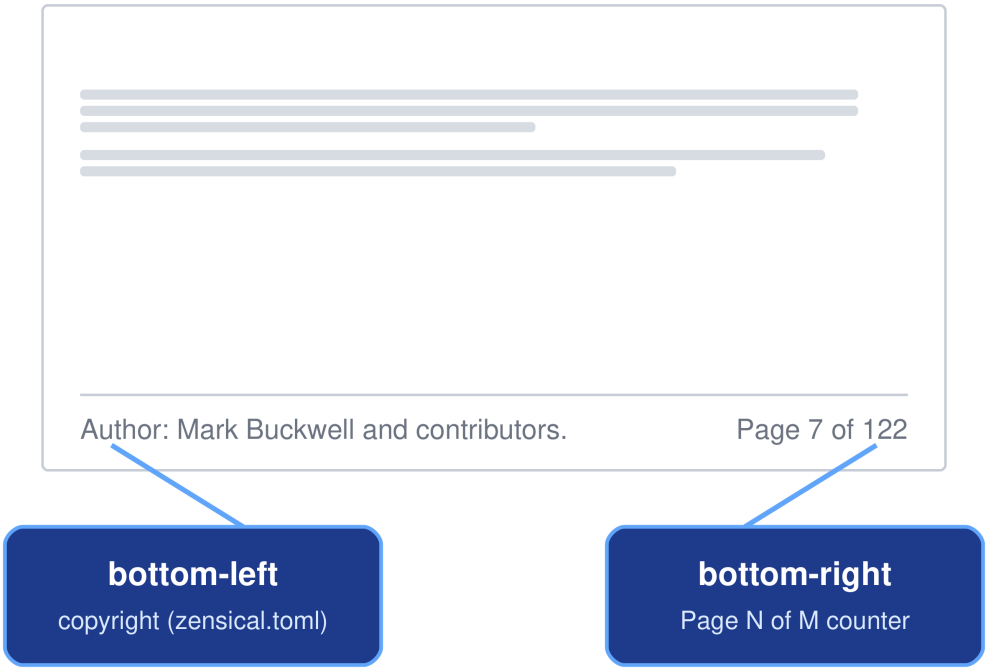


Figure 6.2. PDF page footer layout

The cover page (`docs/index.md`) never shows this header or footer at all - see the note at the end of Page size and margins below.

Set `project.extra.pdf_header_footer_font_size`, `pdf_header_footer_color`, and `pdf_header_footer_divider_color` in `zensical.toml` to change the header/footer text's font size, its colour, and the divider line's colour:

```
[project.extra]
pdf_header_footer_font_size = "10pt"
pdf_header_footer_color = "#555555"
pdf_header_footer_divider_color = "#e2e8f0"
```

Each accepts any valid CSS value (e.g. `9pt`, `rgb(85, 85, 85)`). One setting each applies to the header and the footer together - there's no separate setting per corner, since nothing else about their appearance differs. All three default to the values shown above if left unset.
`pdf_header_footer_divider_color` is independent of `pdf_header_footer_color` - changing the text colour doesn't automatically lighten or darken the divider line, so set both if you want them to match.

6.4.3 Page size and margins

Set `project.extra.pdf_page_size` and `project.extra.pdf_margin_top` / `pdf_margin_right` / `pdf_margin_bottom` / `pdf_margin_left` in `zensical.toml`:

```
[project.extra]
pdf_page_size = "A4"
pdf_margin_top = "2cm"
pdf_margin_right = "2cm"
pdf_margin_bottom = "2cm"
pdf_margin_left = "2cm"
```

`pdf_page_size` accepts any standard CSS page size (e.g. `letter`, `legal`, `A3`) or explicit dimensions (e.g. `21cm 29.7cm`), optionally followed by `landscape`. Each `pdf_margin_*` setting accepts any valid CSS length (e.g. `2cm`, `0.75in`) and can differ from the others - useful for, say, extra left margin for binding, or matching an institution's own asymmetric submission template. The header and footer live inside this margin, so shrinking a side also narrows the space available to them there. All five default to the values shown above if left unset, and none of them affect the live website.

The PDF also reuses your website's theme fonts (body copy, headings, and the header/footer) - see Fonts above for the `zensical.toml` setting.

> **Note**
>
> The cover page (`docs/index.md`) never shows the running header or footer, and heading numbering (e.g. "11.4") is a separate setting - see Changing heading numbering in Customise document content.

6.4.4 Double-sided (duplex) printing

Set `pdf_double_sided = true` in `[project.extra]` to mirror the header, footer, and margins between recto (right-hand) and verso (left-hand) pages, and start every chapter on its own recto page - matching the convention of a professionally bound, printed document:

```
[project.extra]
pdf_double_sided = true
pdf_margin_inner = "2.5cm"
pdf_margin_outer = "1.5cm"
```

`pdf_margin_inner` / `pdf_margin_outer` replace the plain `pdf_margin_left` / `pdf_margin_right` above once duplex printing is on - "inner" is the spine-side margin (left on a recto page, right on a verso page), "outer" is the fore-edge on the opposite side - so one pair of settings covers both without you needing to track which physical side is which for any given page. Every numbered chapter also starts its own recto page automatically, inserting a blank page where needed, just like a printed book. Add `recto_title: "Short Title"` to

a page's own front matter to shorten its running-header title from the *next* page onward, for a chapter title too long to comfortably fit.

6.4.5 Source-code bundling

Run `prodockit source-bundle` alongside `prodockit pdf` to also produce a `source_bundle.pdf` - a separate document containing your Markdown content and `zensical.toml`, one file per page:

```
prodockit source-bundle
```

Useful for a submission that requires the underlying source alongside the report itself. A separate command rather than a `zensical.toml` setting, so a project that wants only the rendered document doesn't build the source bundle on every run too.

Written into `docs_dir`, alongside `site_documentation.pdf`, so the website's own **Source** download button finds it with no extra step.

> **Only your Markdown and config, not your whole repository**
>
> Earlier versions of this feature bundled every one of your repository's own tracked (and not-ignored) files - Python, CSS, tests, everything. `prodockit source-bundle` bundles your documentation's own source: every `.md` file, plus `zensical.toml`. For most coursework this is what you actually want - your own written content, not the template's build tooling.
>
> If your submission needs the *whole* repository bundled - for example, an originality declaration covering custom code you wrote yourself, not just prose - that's still possible, just not from this command. See `prodockit.pdf.source_bundle` in the prodockit-extensions docs for calling `build_source_bundle()` directly.

6.4.6 Screenshots

Every screenshot of an application or website - as opposed to a logo, icon, or diagram - must have both `figure-caption` *and* the `.screenshot` class, which frames it with a subtle border, rounded corners, and a light shadow so it reads as "a picture of your screen" rather than blending into the body text. Add `.screenshot` as an extra attribute alongside `width`:

```
![Initial commit](images/initial-commit.png)
{ width="40%" .screenshot }
/// figure-caption
Initial commit
///
```

`.screenshot` works the same way in both outputs - the underlying CSS rule lives in `docs/stylesheets/extra.css` for the website and the equivalent compiled block generated by `prodockit.pdf` for the PDF, matching every other class this template applies to both.

6.5 Directory structure

Now that you've customised the website, the document structure, the cover page, and the PDF layout, it's worth knowing where everything you've just changed actually lives. The listing below is the template as delivered. Use it as a reference when you're looking for a file mentioned earlier in this section, or deciding where to add a new page.

- docs — Your report's Markdown source
  - index.md — The cover page
  - originality.md — Your declaration of originality and AI use, for you to complete
  - section1.md — The first section, with worked citation, acronym and glossary examples
  - section2.md — The second section, with a worked cross-reference example
  - section3.md — The third section, with a worked figure-caption example
  - section4.md — The fourth section, with a worked table-caption example
  - acronyms.md — Your acronym list, for you to complete
  - glossary.md — Your glossary of key terms, for you to complete
  - references.md — Your bibliography, for you to complete
  - bibliography.md — The bibliography page
  - assets — Images, logos and header backgrounds
  - javascripts — Site JavaScript, including the MathJax bundle CI installs
  - stylesheets — CSS for both outputs
    - extra.css — Website customisations
    - print.css — PDF-only styles, loaded by the `pdf_extra_css` setting
- overrides — Theme partials this template replaces
- tools — Node tooling used only by the PDF build
  - mermaid — `mermaid-cli`, for rendering diagrams to images
  - mathjax — `mathjax-full`, for rendering maths to images
- test — The test suite that checks the built website and PDF
- .github — GitHub configuration
  - workflows — The pipelines that publish the site and PDF
    - docs.yml — Builds and publishes to GitHub Pages
    - drift.yml — Reports when a pinned build input falls behind
    - release-redeploy.yml — Rebuilds the site after a release is published
  - ISSUE_TEMPLATE — Issue forms for this repository
- .vscode — Editor settings and the LTeX dictionary
- zensical.toml — Site configuration and navigation
- macros.py — This template's own build-time logic
- references.bib — Your bibliography source
- bibliography.bib — A second bibliography source, if you keep them apart
- requirements.txt — The Python dependencies
- testrequirements.txt — The test suite's own dependencies
- .python-version — The Python version CI and your shell both read

- .vale.ini — Configuration for Vale, a prose style checker
- .gitlab-ci.yml — Builds and publishes to GitLab Pages
- .gitignore — Files and directories Git should ignore
- README.md — The repository's front page, for you to rewrite before submitting
- CONTRIBUTING.md — How to contribute to the template itself
- CHANGELOG.md — What has changed in the template
- LICENSE — The MIT licence this template is published under

Some of these are covered in more detail elsewhere: References and bibliography, Acronyms and abbreviations and Glossary in Customise document content; Extra CSS and JavaScript above for `print.css`; Diagrams and maths for `tools/`; and Testing for `test/`.

`harvard-cite-them-right.csl` is not in the list because it is not committed - every build fetches it, and `prodockit bootstrap` does it for you.

6.6 Where to go next

Continue to Customise document content for the prodockit extensions that number, cross-reference, cite, and index your document's actual content - or skip ahead to Customise build for how your document is built and published.

7. Customise document content

Customisation covers your website's branding, layout, and cover page. This page covers the prodockit extensions that number, cross-reference, cite, define, and index your document's actual *content* - the things you reach for while writing, not while setting the project up. Every one of them works identically on the website and in the PDF, with no separate PDF-side translation needed.

7.1 Changing heading numbering

By default, this documentation template enables heading numbering. If you want to disable heading numbering, you can do so by adding the following line to the `[project.extra]` section of the `zensical.toml` file:

```
heading_numbering = false
```

This will also disable heading numbering in the generated PDF output. If you want to enable heading numbering again, simply set the value to `true`:

```
heading_numbering = true
```

The top level heading numbering shown in the sidebar isn't generated automatically - it's typed directly into each entry's title in `nav`, matching the pattern of the ones already there, for example:

```
{"6. Case Study" = "casestudy.md"}
```

Keep the numbers in each title sequential as you add, remove, or reorder chapters - inserting a new entry partway through (as above, right after "5. Section") means renumbering every entry after it, since (unlike the in-page heading numbers) `nav` doesn't renumber these for you.

> **Note**
>
> Appendix pages are the one exception - see Appendixes below - since they're lettered rather than numbered, and don't take a number from this sequence at all. The front matter flag that marks a page as an appendix is `is_appendix` by default, but its name itself is configurable (`appendix_attr` in `prodockit.headings`) if you'd rather use a different key.

7.2 Section cross-references

This template uses `prodockit.refs` (from the same prodockit package as citations/glossary below) for cross-references: give a heading an id, then reference it from anywhere with `\ref{id}` - it resolves to that heading's current section number, similar in spirit to LaTeX's `\ref`.

> **How the PDF handles this**
>
> Same as citations/glossary below - `prodockit pdf` renders this page through the real Zensical/prodockit pipeline, so `\ref{id}` resolves the same way in both outputs with no separate PDF-side translation.

1. Every heading already has an id, the same slugified-from-its-text id `toc`'s permalinks use (this page's own "## Changing heading numbering" heading above got `changing-heading-numbering` automatically, with no extra markup needed). Give it an explicit id instead with attr_list syntax when you want a short, stable id that won't change if you reword the heading later, or to avoid a collision with another heading elsewhere in the document that slugifies to the same text:

   ```
   ## SubSection {: #citations-example }
   ```

2. Reference it from anywhere in the document with `\ref{id}`:

   ```
   As covered in \ref{changing-heading-numbering}, ...
   ```

   Which renders as: As covered in 7.1 Changing heading numbering¶, ...

   No need to track down the section's current number, or update it by hand if the target moves - `\ref{id}` re-resolves on every build. This template's own `docs/section1.md` - `docs/section4.md` cross-reference each other's citation/acronym/glossary/caption examples this way, each using an explicit attr_list id since "SubSection" repeats several times per page.

> **Note**
>
> A reference to a heading that doesn't exist (a typo in the id, or a heading in a page not yet processed) falls back to `??`, the same way an undefined LaTeX `\ref` shows `??` until a later compilation pass - a quick visual signal something needs fixing. It renders with a `prodockit-ref-unresolved` CSS class, so you can style it more prominently (e.g. a

> warning colour) in `extra.css` if `??` alone isn't visible enough while drafting.

7.3 References and bibliography

This template uses `prodockit.citations` (from the prodockit package, already installed and enabled in `zensical.toml` - see prodockit-template#25) for citations: define a source once, cite it by key anywhere with `\citeref{id}`.

> **How the PDF handles this**
>
> `prodockit pdf` renders every page through the same Zensical/prodockit pipeline the website uses, so `\citeref{id}` resolves to the same linked citation in both outputs automatically - no separate PDF-side translation needed, and no manual HTML or per-output link either.

1. Create a page for your sources (this template includes one at `docs/references.md`). List each source as a paragraph, and give it a short, unique id plus a short display text using attr_list syntax on the line directly below it (no heading needed - attr_list works on plain paragraphs too):

   ```
   Skoulikari, A. (2023) *Learning Git: A Hands-On and Visual Guide
   to the Basics of Git*. Sebastopol, CA: O'Reilly Media.
   {: #skou2023 .reference data-cite-text="Skoulikari, 2023" }
   ```

   Each entry needs a blank line before and after it - attr_list only recognises `{: ... }` as an id (rather than literal visible text) when it's the last line of its own paragraph. Removing the blank lines to save space merges entries into one paragraph and breaks both outputs.

2. Add the page to `nav` in `zensical.toml` so it appears in the sidebar - as a regular numbered chapter, or as a lettered appendix (see Appendixes below). This template ships it as an appendix by default.

3. Cite the source in-text with `\citeref{id}`:

   ```
   Git is a tool used to manage version control.\citeref{skou2023}
   ```

   Which renders as: Git is a tool used to manage version control. [Skoulikari, 2023]

   No relative path to work out, regardless of which page cites it - unlike a hand-typed Markdown link, `\citeref{id}` resolves the same way from

any page, and the `data-cite-text` you set once is reused everywhere the source is cited. Cite more than one source in the same place with a comma: `\citeref{skou2023,chacon2014}` renders \citeref{skou2023,chacon2014}.

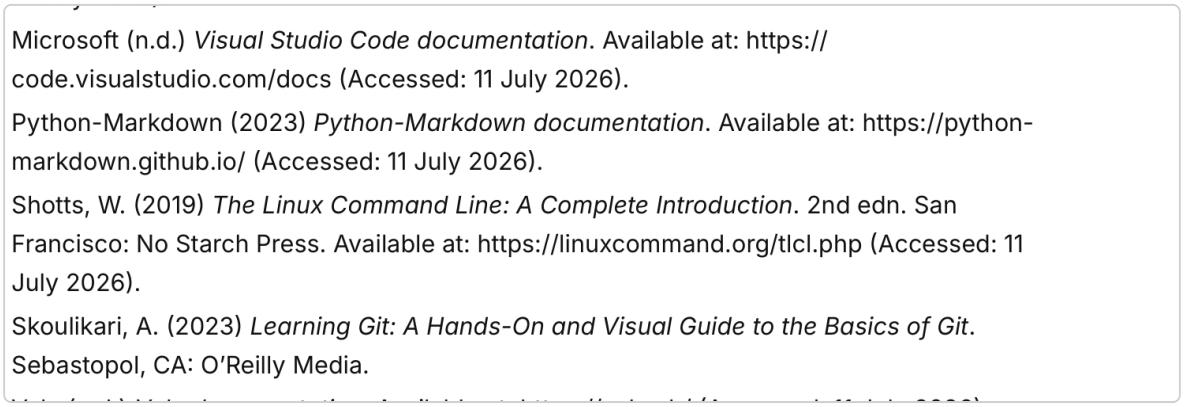
This in-text citation resolves correctly in both outputs - on the website, and as an internal cross-page link jumping straight to the cited entry within the built PDF.

4. Consecutive entries get the browser's normal spacing between paragraphs by default - noticeably looser than a typical bibliography. Give each entry's attr_list line a `.reference` class alongside its id and `data-cite-text` (as shown in the code block above) so the template's layout rules - described next - can target them.

5. Set `project.extra.reference_style` in `zensical.toml` to control how `.reference` entries are laid out, on both the website and the PDF build:

```
[project.extra]
reference_style = "european" # or "global"
```

`"european"` (the default) - single line spacing, no indent, entries close together:



Microsoft (n.d.) *Visual Studio Code documentation*. Available at: https://code.visualstudio.com/docs (Accessed: 11 July 2026).

Python-Markdown (2023) *Python-Markdown documentation*. Available at: https://python-markdown.github.io/ (Accessed: 11 July 2026).

Shotts, W. (2019) *The Linux Command Line: A Complete Introduction*. 2nd edn. San Francisco: No Starch Press. Available at: https://linuxcommand.org/tlcl.php (Accessed: 11 July 2026).

Skoulikari, A. (2023) *Learning Git: A Hands-On and Visual Guide to the Basics of Git*. Sebastopol, CA: O'Reilly Media.

Figure 7.1. European reference style

`"global"` - double spacing between entries, with a 0.5in/1.27cm hanging indent on wrapped lines (the common APA/MLA/Chicago style):

Mermaid (n.d.) *Mermaid documentation*. Available at: https://mermaid.js.org/ (Accessed: 11 July 2026).

Microsoft (n.d.) *Visual Studio Code documentation*. Available at: https://code.visualstudio.com/docs (Accessed: 11 July 2026).

Python-Markdown (2023) *Python-Markdown documentation*. Available at: https://python-markdown.github.io/ (Accessed: 11 July 2026).

Shotts, W. (2019) *The Linux Command Line: A Complete Introduction*. 2nd edn. San Francisco: No Starch Press. Available at: https://linuxcommand.org/tlcl.php (Accessed: 11 July 2026).

Skoulikari, A. (2023) *Learning Git: A Hands-On and Visual Guide to the Basics of Git*. Sebastopol, CA: O'Reilly Media.

Figure 7.2. Global reference style

Set `project.extra.reference_spacing_european`, `reference_indent_global`, and `reference_spacing_global` in `zensical.toml` to change the spacing/indent values themselves, on both the website and the PDF build:

```
[project.extra]
reference_spacing_european = "-0.8em"  # gap between entries,
"european" style
reference_indent_global = "1.27cm"     # hanging indent on
wrapped lines, "global" style
reference_spacing_global = "2em"       # gap between entries,
"global" style
```

Each accepts any valid CSS length and defaults to the value shown above if left unset. `reference_spacing_european` also controls the Acronyms and Glossary pages' own list spacing, which share the same tight "european" look but have no "global"-style alternative to switch to.

Tip

Keep ids short and stable (e.g. `skou2023`, author surname plus year) so citations keep working even if you reorder entries on the references page later. Unlike a hand-typed link, `\citeref{id}` needs no adjustment when citing from a page nested in a subdirectory.

> **Note**
>
> An unresolved `\citeref{id}` (a typo in the key, or a source not yet added) renders `?` instead of a linked citation, with a `prodockit-cite-unresolved` CSS class for styling it distinctly.

7.3.1 An alternative: prodockit.bibliography

This template also enables `prodockit.bibliography` in `zensical.toml`, a different way to manage sources - not currently used in this guide's own content, but available if you'd rather work this way instead of (or alongside) `prodockit.citations` above.

Where `prodockit.citations` is a hand-typed reference list you write and format yourself, `prodockit.bibliography` generates one automatically from a BibTeX/BibLaTeX `.bib` file, in any Citation Style Language (CSL) style - APA, IEEE, Harvard, and hundreds more:

```
[project.markdown_extensions."prodockit.bibliography"]
bib_file = "references.bib"
csl_style = "harvard-cite-them-right.csl"
```

The template enables this by default, pointing at `harvard-cite-them-right.csl` - but that file isn't part of the clone, so it's fetched rather than committed. If you followed Install tooling when setting up, you already did this; if not, or if you'd rather use a different CSL style, fetch one from the Zotero Style Repository into your project root and point `csl_style` at its filename:

```
curl -fsSL -o harvard-cite-them-right.csl "https://www.zotero.org/
styles/harvard-cite-them-right"
```

Cite a source with `\cite{id}` (note the different marker, distinct from `\citeref{id}` above), and put a bare `\bibliography` marker on its own paragraph wherever you want the formatted reference list to appear:

```
Git is a distributed version control system \cite{chacon2014}.
```

It's a longer-term trade for a shorter one: `prodockit.bibliography` needs Pandoc installed even for a website-only build with no PDF, but in return gives you an automatically generated, correctly styled reference list you never hand-format yourself. See prodockit.bibliography's own docs for the full comparison between the two.

7.4 Acronyms and abbreviations

This template uses `prodockit.glossary` (from the same prodockit package as citations above - see prodockit-template#87) for acronyms: define a term once, insert it by id with `\gls{id}` - it expands to the term's own text, linked to its definition.

> **How the PDF handles this**
>
> Same as citations above - `prodockit pdf` renders this page through the real Zensical/prodockit pipeline, so `\gls{id}` resolves the same way in both outputs with no separate PDF-side translation.

1. Create a page for your acronyms (this template includes one at `docs/acronyms.md`). List each acronym as a short paragraph, and give it an id plus a `data-term` attribute (the acronym's own text) using attr_list syntax on the line directly below it:

   ```
   **CSS** - Cascading Style Sheets
   {: #css .acronym data-term="CSS" }
   ```

   Each entry needs a blank line before and after it, and the `.acronym` class is what keeps consecutive entries close together rather than using the browser's normal, looser paragraph spacing.

2. Add the page to `nav` in `zensical.toml` so it appears in the sidebar - as a regular numbered chapter, or as a lettered appendix (see Appendixes below). This template ships it as an appendix by default.

3. Insert the acronym the first time you use it in a page with `\gls{id}`:

   ```
   This template uses \gls{css} to control the website's
   appearance.
   ```

   Which renders as: This template uses CSS to control the website's appearance.

> **Tip**
>
> Keep ids short and lowercase (e.g. `css`, `pdf`) so `\gls{id}` keeps working even if you reorder entries on the acronyms page later.

> **Note**
>
> An unresolved `\gls{id}` renders `?` instead of the term's expansion, with a `prodockit-gls-unresolved` CSS class for styling it distinctly.

7.5 Glossary

You can build a glossary of key terms the same way, in its own page - this template includes one at `docs/glossary.md`, right after the acronyms page in `nav`. Acronym entries and glossary entries share the same `prodockit.glossary` registry - they're the same kind of thing, an id with a short display text - so a `\gls{id}` works identically whichever page defines it.

> **How the PDF handles this**
>
> Same as acronyms above - resolved automatically, no separate PDF-side translation.

1. Create a page for your glossary (this template includes one at `docs/glossary.md`). List each term as a short paragraph, and give it an id plus a `data-term` attribute using attr_list syntax, the same as an acronym entry:

   ```
   **Markdown** - A lightweight markup language for formatting
   plain text...
   {: #markdown-def .glossary data-term="Markdown" }
   ```

   Give glossary entries their own ids, distinct from any acronym ids for the same concept (for example `css-def` rather than `css`) - `prodockit.glossary` shares one id namespace across every page, so two entries sharing an id anywhere in the site would collide.

2. Add the page to `nav` in `zensical.toml` so it appears in the sidebar - as a regular numbered chapter, or as a lettered appendix (see Appendixes below). This template ships it as an appendix by default.

3. Insert the term the first time you use it in a page with `\gls{id}`, the same way as an acronym:

   ```
   This document is written in \gls{markdown-def}.
   ```

   Which renders as: This document is written in Markdown.

4. Cross-link an acronym to its own glossary entry (and vice versa) with a

plain Markdown link, **not** `\gls{id}`. A "see also" reference needs to say something like "see the glossary", not repeat the term itself - `\gls{id}` always inserts the term's own registered text instead, so `\gls{css-def}` would read "See Cascading Style Sheets for the expansion" rather than "See the glossary...":

```
**CSS** - Cascading Style Sheets. See the [glossary]
(glossary.md#css-def) for what this means in practice.
{: #css .acronym data-term="CSS" }
```

This template's own `docs/acronyms.md` / `docs/glossary.md` cross-link every entry that has a counterpart on the other page this way - see prodockit.glossary's own docs for the full rule of thumb: `\gls{id}` when the term's own name belongs in the sentence, a plain link when the link text needs to say something else entirely.

> **Tip**
>
> If a term is also one of your acronyms, cross-link the two entries as shown above rather than duplicating the explanation on both pages.

7.6 Appendixes

Set `is_appendix: true` in a page's front matter to give its heading letter-based numbering - "Appendix A", "Appendix B", ... - instead of continuing the document's normal numbered sequence, matching the usual academic convention for appendixes. Sub-headings within an appendix page number the same way numbered sections do, just using the letter instead of a chapter number - "A.1", "A.1.1", and so on.

```
---
icon: lucide/book-open
is_appendix: true
---
```

Appendix pages are lettered in `nav` order - the first `is_appendix: true` page becomes Appendix A, the second becomes Appendix B, and so on - regardless of how many numbered chapters come before them, and without taking a number away from that sequence (see the note in Changing heading numbering above). This template ships `docs/acronyms.md`, `docs/glossary.md`, and `docs/references.md` as appendixes by default, grouped under their own "Appendixes" tab in `nav`.

> **Note**

Like the numbered chapter titles in `nav` (see Changing heading numbering), the "Appendix A"/"Appendix B" prefix shown in the sidebar isn't generated automatically - type it directly into each entry's title in `nav`, matching the pattern already there:

```
{"Appendix A. Acronyms" = "acronyms.md"}
```

Tip

Appendixes conventionally don't count toward a submission's word limit either - pair `is_appendix: true` with `exclude_from_word_count: true` (see Word count and repository link in Customisation), as this template's own appendix pages already do.

7.7 Back-of-book index

This template also enables `prodockit.index` for a back-of-book index: a traditional, alphabetised, PDF-only index at the end of your document, listing every term you've marked and the page(s) it appears on - the same kind of index/back matter every printed technical book has.

The shipped userguide already enables the extension **and** generates the index, so you normally only need to mark the terms you want to include. There is no separate setup step.

PDF-only

There's no website equivalent - a reader of the live site uses Zensical's own search instead. Marking a term has no visible effect on the website at all; it only ever becomes an index entry once the PDF is built.

Mark a term wherever you actually discuss it with `\index{Term}` - it displays inline exactly as written, and is marked for the index in one go, no separate definition step needed:

```
A \index{merge conflict} happens when Git can't automatically
combine two changes.
```

Which renders as: A merge conflict happens when Git can't automatically combine two changes.

Marking the same term more than once across the document merges into a single index entry, with every page it appears on listed together.

Nest a term under another with `Parent!Child` - only the last segment displays inline, the earlier segments only ever shape how the generated index groups related entries:

```
Now generate the \index{Git!ssh keys} to use for authentication.
```

Which renders as: Now generate the ssh keys to use for authentication.

Wrap the last segment in backticks to mark a command or other code term, both inline and in the generated index entry:

```
Run \index{`git commit`} to save your changes.
```

Which renders as: Run `git commit` to save your changes.

The existing extension table in `zensical.toml` contains the generation settings:

```
[project.markdown_extensions."prodockit.index"]
include = true
title = "Index"
```

`include = true` appends the generated index to the PDF. The `title` line is an optional customisation: change it if your document needs a different heading, or omit it to use the default `"Index"`.

> **Tip**
>
> Generating page numbers for the index needs a second pass over the whole PDF, so it's a little slower than a build without one. The shipped userguide already has `include` switched on.

7.8 Captions

The attribute list-based `<figure>`/`<figcaption>` pattern in Zensical basics works for images, but this template also enables `pymdownx.blocks.caption`, a `/// caption ... ///` block that captions *either* an image *or* a table, auto-numbers itself, and - unlike the `<figure>` approach - works correctly in the PDF too.

> **How the PDF handles this**
>
> `prodockit pdf` renders this page through the real Zensical/pymdownx pipeline, so `pymdownx.blocks.caption`'s own per-page auto-number is already correct by the time Pandoc sees it - a Lua filter (`Figure()`, generated by `prodockit.pdf.lua`) just prepends the current chapter number/appendix letter in front of it (e.g. "1." → "8.1."), matching the same `<chapter>.<n>` numbers the website shows via CSS.

This template configures three caption types under `[project.markdown_extensions.pymdownx.blocks.caption]` in `zensical.toml`:

1. **`caption`** - plain and unnumbered, for an image that doesn't need a "Figure N" label - a decorative image or an institution logo, rather than a screenshot or diagram that's part of the document's actual content:

   ```
   ![Institution logo](images/logo.png)
   /// caption
   Institution logo
   ///
   ```

2. **`figure-caption`** - auto-numbered "Figure `<chapter>.<n>`" (e.g. "Figure 9.1"), attached to the image immediately before it. `<chapter>` is wherever this page ends up in `nav`; `<n>` auto-increments per page - reordering chapters, or adding another figure to the page, never needs a manual renumber:

   ```
   ![GitLab fork project](images/gitlab-fork-project.png)
   { width=70% .screenshot }
   /// figure-caption
   GitLab fork project
   ///
   ```

3. **`table-caption`** - the same auto-numbering, but for a table, shown *below* it by default - just like a figure. Add `| <` after the type name to show it *above* the table instead, genuinely repositioned in both outputs rather than just a CSS visual trick:

   ```
   | Feature | Fork | Clone |
   |----|----|---|
   | ... |
   /// table-caption | <
   Fork and Clone Comparison at a Glance
   ///
   ```

> **⚠ Always add `| <` to `table-caption`**
>
> `table-caption` has no setting that makes it default to top-positioned - `| <` isn't optional here, it's part of the syntax every single `table-caption` block needs. This template shows every table caption of its own above its table (see Fork and cloning the prodockit-template for a real example); a `table-caption` block missing `| <` silently falls back to *below* the table instead, breaking that consistency without any warning. There's no `zensical.toml` setting to make this the default and skip typing `| <` each time - see issue #68 if you want to help change that.

The caption block always comes *after* the image or table it captions, regardless of where it's actually shown - `pymdownx.blocks.caption` attaches to whichever element immediately precedes it.

> **Tip**
>
> Force a specific number instead of the auto-incrementing one with `| 5` (later auto-numbers on the same page continue counting up from there, never going backwards); give a caption a stable custom id instead of the auto-generated one with `| #my-id`; add an extra CSS class with `| #my-id.my-class`. Combine modifiers with spaces, e.g. `/// table-caption | < 5 #my-id`.

> **Caption every image, diagram, and table**
>
> Every screenshot, diagram, or other image that's actually part of the document's content gets `figure-caption`, and every table gets `table-caption` - so a reader can cite "Figure 7.2" or "Table 3.1" and mean something specific. Reserve the plain `caption` type for decorative images that aren't part of the content itself, like an institution logo (see the `caption` example above).

7.8.1 Table column widths

Give any table's header cell a `width` attribute with attr_list syntax to control how much horizontal space that column takes, in either output:

```
| Name {: width="25%" } | Description {: width="50%" } | Due {:
width="25%" } |
|---|---|---|
```

```
| Headings | Heading ids and section numbers | Q1 |
| Refs | Cross-references, resolved by number | Q2 |
```

Which renders as:

Name	Description	Due
Headings	Heading ids and section numbers	Q1
Refs	Cross-references, resolved by number	Q2

Use a CSS length (e.g. `120px`) instead of a percentage for a column that should stay a fixed size regardless of the table's own width. Leave a column unannotated and it takes whatever space is left over, shared evenly with any other unannotated column - only a table with at least one `width` gets this treatment at all.

7.8.2 Dense tables

A table with many short columns comes out wider than it needs to be. The theme gives every header cell a minimum width of `5rem` and pads each cell generously, so a column holding `H` takes as much room as one holding a sentence - and a wide table overflows whatever it actually contains.

Mark any header cell `{: .compact }` to turn both off:

```
| Threat {: .compact } | Likelihood | Impact | Risk |
|---|---|---|---|
| Credential theft | H | H | H |
```

Which renders as:

Threat	Likelihood	Impact	Risk
Credential theft	H	H	H

The marker describes the whole table, not the column it's written on - it can go on any header cell. It applies to the website and the PDF alike, and combines with `width`, which answers a different question: how wide one column is, rather than how tightly every cell is set.

It's deliberately opt-in. A table that reads well at its default keeps it.

7.8.3 A header of more than one row

A Markdown table has exactly one header row. A heading that needs two lines has to be written as a second body row - and that row then stops repeating when the table breaks across pages, because only the real header repeats.

Mark it `{: .header }` and it becomes part of the header:

```
| Target {: rowspan=2 } | Measured {: colspan=2 } | | Note {:
rowspan=2 } |
|---|---|---|---|
| | Before {: .header } | After | |
| Widget | 1 | 2 | ok |
```

Which renders as:

Target	Measured		Note
	Before	After	
Widget	1	2	ok

Both lines now repeat on every page the table reaches.

`colspan` and `rowspan` merge cells in the usual way. Because a pipe table has to keep its columns to parse at all, a merged cell is written with empty cells after it - those are removed, so the row isn't left wider than its header.

> **Put the marker on a cell that has text**
>
> `attr_list` needs something to attach to, so `{: .header }` won't work in an empty cell. In the example above the first cell of the second row is blank - covered by the `rowspan` above it - so the marker goes on `Before` instead. Any cell in the row will do.
>
> Only rows at the top are promoted. A `{: .header }` further down the table stays where it is, rather than the table being quietly re-ordered around it.
>
> An empty placeholder cell is removed; one with text in it is kept, on the assumption that it's your content.

7.8.4 Rotated headings

A wide table is often wide because of its headings, not its data. Turn them on their side:

```
| Item | Availability {: rotate=270 width="2em" height="90pt" } |
Confidentiality {: rotate=270 width="2em" height="90pt" } |
|---|---|---|
| Investment data | H | H |
```

Which renders as:

Item	Availability	Confidentiality
Investment data	H	H

`270` reads bottom-to-top and `90` top-to-bottom; no other angle is accepted, because it would give a heading nobody can read and a row height nobody can predict.

All three parts are needed together, and a `rotate` without a `width` is refused rather than rendered:

> ⚠ **The width is what saves the space**
>
> Rotating text doesn't make a column narrower - a rotated heading still occupies the room it would have taken lying flat. **The `width` is what narrows the column; the rotation is what keeps the heading readable once it is narrow.** That's why the two have to be given together: a rotated heading in a full-width column looks exactly like the feature working.
>
> `height` sets how tall the header row is, and is what a long heading wraps against - the rotated text reserves no height of its own.

7.8.5 Landscape Table or Diagram

A table or diagram too wide for a portrait page can have its own landscape page instead - wrap it (and its own caption) in a `<div class="landscape-page" markdown="1">` block, using `md_in_html` (the `markdown="1"` is required):

```
<div class="landscape-page" markdown="1">

| ID {: width="15%" } | Description {: width="70%" } | Due {:
width="15%" } |
|---|---|---|
| 1 | ... | Q1 |
/// table-caption | <
A wide reference table
///

</div>
```

The caption block goes **after** the table it describes - that is what attaches the

two together. Put it before and the caption becomes a standalone item with the table left loose beneath it.

A diagram works the same way:

```
<div class="landscape-page" markdown="1">

![Architecture overview](assets/images/architecture.png)
/// figure-caption
Architecture overview
///

</div>
```

A table longer than one page carries on across further landscape pages, repeating its header on each one exactly as it would on a portrait page - including a two-row header marked with `{: .header }`.

This is PDF-only - the same table or diagram renders completely normally on the live website. A page break is always forced immediately before and after the block, so it never shares a page with anything else. A document mixing portrait and landscape pages prints without any special handling.

7.9 Finalising your document

Before you release your document, work through the following step.

7.9.1 Remove the Originality warning

Delete the first Warning admonition box in `originality.md` - it's a note for you as the author, explaining what to do on that page, and isn't part of your declaration itself.

> **Note**
>
> Earlier versions of this template shipped a "START HERE" nav entry and stub page (`docs/starthere.md`) that had to be removed before submitting. The template no longer ships one at all - this User Guide is the only copy of this guidance, so there's nothing left in your own fork to comment out of `nav` or delete.

7.10 Where to go next

Continue to Customise build for how your document is built and published - the two build commands, the tooling that diagrams and maths need in the PDF, and the settings that make publishing behave.

8. Customise build

Customisation covers changing your *document*. This page covers changing how it gets **built and published** - the two build commands, the optional tooling some features need, and the settings that make publishing behave correctly.

> **Why this page exists**
>
> Almost everything that goes wrong in a build goes wrong *quietly*. A missing font is substituted rather than reported. A missing diagram renderer leaves the raw diagram source in your PDF. A shallow clone makes your release number disappear. In each case the build succeeds and publishes something subtly wrong, so there is no error message to search for - you have to know the failure exists.

8.1 The two build commands

Your project produces two outputs from the same source, and each has its own command:

```
prodockit pdf      # builds the PDF
zensical build     # builds the website
```

Both read the same `zensical.toml`, so your zensical.toml nav, fonts, page size and settings apply to both. The website is rendered by Zensical directly; the PDF goes through Pandoc and WeasyPrint instead.

That second path is the source of most of what follows. **WeasyPrint has no JavaScript engine**, so anything your website renders in the browser - diagrams, mathematical notation - has to be turned into a static image *before* the PDF is built.

> **Build the outputs before running the tests**
>
> The Testing suite checks the *built* website and PDF, not the build process. Run both commands first, or the tests will fail on artifacts that simply aren't there yet.

8.2 Diagrams and maths

Diagrams work on the website with no setup: Mermaid.js is part of Zensical's

own bundle, and runs in the reader's browser. Maths does not - MathJax is installed, not committed (it's third-party code, and a repository is redistribution), so a formula renders as raw TeX on the website too until it's been installed once. Neither works in the PDF at all: WeasyPrint has no JavaScript engine, so `prodockit pdf` pre-renders both to images using two Node.js tools.

Set them up once. Which commands you need depends on where your project came from:

From the template, the `tools/mermaid` and `tools/mathjax` manifests and lockfiles are already tracked, so you only install them:

```
npm ci --prefix tools/mermaid
npm ci --prefix tools/mathjax
```

This is the case for most readers, and Install the diagram and maths tooling walks through it alongside installing Node.js itself - including the extra step Maths needs afterwards, to install the MathJax bundle and config the *website* uses (not just the PDF pre-render above).

Starting a project from scratch, you have no `tools/` directory yet, so create it first:

```
prodockit init-tools
npm ci --prefix tools/mermaid
npm ci --prefix tools/mathjax
```

`prodockit init-tools` writes the `tools/mermaid` and `tools/mathjax` directories, then prints the install commands and the settings a CI pipeline needs. Commit the manifests and lockfiles it creates; the `node_modules/` directories are ignored. Still install the MathJax bundle itself afterwards, the same way.

> **Tip**
>
> Running `init-tools` on a copy of the template does no harm - it reports `Kept existing tools/mermaid/package.json` for each file already there and changes nothing - but it has nothing to do, so it is easy to mistake that message for an error.

> **CI needs this too, and starts from nothing every run**
>
> Neither the MathJax bundle nor its config is committed, so a CI pipeline that builds the website - not just the PDF - needs the same install step

this project's own `.github/workflows/docs.yml` and `.gitlab-ci.yml` run, right after `npm ci --prefix tools/mathjax`. Skip it and the pipeline succeeds while publishing a site where every formula is raw TeX - the same silent failure the box below describes, just reaching the website instead of stopping at the PDF.

This is the quiet one

If these tools are missing, `prodockit pdf` does **not** fail. It leaves the content exactly as it found it - so instead of a flowchart, your PDF shows the diagram's own definition text, the `graph LR` line and every node and connector written out beneath it. Instead of a typeset equation, it shows the raw LaTeX, backslashes and braces and all. The website renders diagrams perfectly regardless - Mermaid.js needs nothing from `tools/`, only Zensical itself - but a formula is only as reliable there as the MathJax install above; skip it and the website shows exactly the same raw TeX the PDF would.

That is the right default for a document using neither feature - nobody should have to install Node.js to build a PDF with no diagrams in it. It is a trap for a document that *does* use them. Since prodockit 0.12.0 the build prints a warning naming the missing tool, and this project's test suite fails if either reaches the PDF unrendered.

A project using neither diagrams nor maths can skip this section entirely.

8.3 Fonts

Your website loads its fonts from a CDN when a reader opens the page. The PDF cannot: WeasyPrint has to embed the actual font files, so they must be installed on whatever machine builds the PDF.

If they are missing, **WeasyPrint substitutes a default font without warning**. The PDF builds, publishes, and simply looks wrong.

The fonts to install are whichever ones `zensical.toml` names - see Fonts for that setting. Left unset, they are Inter **Inter** for body text and JetBrains Mono **JetBrains Mono** for code, which is what the examples below use.

8.3.1 What format to install

Install the **desktop** font files - `.ttf` (TrueType) or `.otf` (OpenType). Both work equally well.

> ⚠ **`.woff` / `.woff2` will not do**
>
> Those are web delivery formats. A browser reads them over HTTP, but they are not what your operating system's font system indexes, so WeasyPrint will not find them and will substitute silently - the exact failure this section exists to avoid. If a font is offered as a "web font" download, look for the desktop or "static" download instead.

8.3.2 Where to install them

Install the document fonts

macOS

```
brew install --cask font-inter font-jetbrains-mono
```

Homebrew puts them in `~/Library/Fonts`, which is the per-user font folder - no further step needed. To install a font you downloaded yourself instead, double-click the file and click **Install Font**, or copy the `.otf` / `.ttf` files into `~/Library/Fonts` directly.

Windows

Download the desktop font files, then select them all, right-click, and choose **Install for all users**.

Installing for all users matters if a scheduled task or service builds your PDF, since those do not run as you - a font installed only for your own account is invisible to them.

Linux (Ubuntu)

Where a distribution packages the font, use that - it handles the font cache for you:

```
sudo apt-get install -y fonts-inter fonts-jetbrains-mono
```

For a font with no package, copy the files in by hand and rebuild the cache:

```
mkdir -p ~/.local/share/fonts
cp *.ttf *.otf ~/.local/share/fonts/
fc-cache -f
```

Use `/usr/share/fonts` instead of `~/.local/share/fonts` to install for every user on the machine.

Your CI runner needs them too, and starts from a bare image every run - see the `fonts-inter fonts-jetbrains-mono` line in Publishing below.

8.3.3 Check they were actually used

Since a missing font produces a PDF rather than an error, the only reliable check is to look at what the finished document actually embedded:

```
python -c "
import pymupdf
d = pymupdf.open('docs/site_documentation.pdf')
print(sorted({f[3].split('+')[-1] for p in d for f in
p.get_fonts()}))
"
```

`pymupdf` is already installed - it comes with the `[index]` extra in `requirements.txt`. Run against this guide's own PDF, it prints:

```
['Inter', 'Inter-Bold', 'Inter-Italic', 'Inter-Ultra-Bold',
 'JetBrains-Mono', 'JetBrains-Mono-Bold', 'Trebuchet-MS']
```

The configured fonts, in whichever weights the document happens to use. A name you did not configure appearing there is the substitution happening - a PDF built without Inter installed shows a default serif or sans face in its place instead.

> **The `ABCDEF+` prefix**
>
> Embedded fonts normally carry a random six-letter tag, as in `LNXIQZ+Inter`, marking the file as a *subset* containing only the glyphs

this document actually uses. The `.split('+')[-1]` above strips it so the names read cleanly; it says nothing about whether the font is correct.

Mermaid diagrams bring their own font

You will also see `Trebuchet-MS` if your document contains a diagrams Mermaid diagram. That is not a substitution: Mermaid renders its diagrams to SVG with its own stylesheet, which asks for Trebuchet MS, and that travels into the PDF with the image. It applies only to text *inside* diagrams, and needs no action - your body text and code are unaffected.

8.4 Pinning build inputs

Your document has more inputs than its own content. Zensical renders the site, Pandoc and WeasyPrint lay out the PDF, and whatever runs the pipeline carries its own OS image. Left unpinned, an upgrade to any of them doesn't fail the build - it just quietly publishes a slightly different document, with nothing to indicate anything changed.

`prodockit pins` keeps these declarations - a version in `requirements.txt`, a `PANDOC_VERSION` in a workflow file, a runner label - in step with each other, wherever they're written down:

```
prodockit pins          # prompt per package; Enter takes the newest
release
prodockit pins --check  # behind, or files disagreeing with each
other? exit non-zero
```

With no `-p` flags, both commands cover five packages by default: `zensical`, `weasyprint`, `markdown`, `pymdown-extensions`, and **pandoc** - Pandoc is the fourth build input alongside Zensical and WeasyPrint, and the one that has actually caught this project family out. Pandoc 3.10 stopped treating a syntax-highlighted `<pre><code>` as a code block; on that version every fenced code block in the PDF lost its preformatting and reflowed as justified prose, while the build reported success. The published PDFs stayed correct only because CI happened to be installing an older pandoc than anyone's laptop - exactly the silent drift this section exists to catch. Pandoc version drift covers the how, and prodockit's limitations page records the episode in full.

This project pins `zensical`/`weasyprint` in `requirements.txt`, and pandoc as `PANDOC_VERSION` in both `.github/workflows/docs.yml` and `.gitlab-ci.yml`:

```
$ prodockit pins --check
zensical
  requirements.txt:12  zensical==0.0.53
  newest on PyPI: 0.0.53
...
pandoc
  .gitlab-ci.yml:52  PANDOC_VERSION: "3.10.1
  .gitlab-ci.yml:139  PANDOC_VERSION: "3.10.1
  .github/workflows/docs.yml:50  PANDOC_VERSION: "3.10.1
  not on PyPI - set the version yourself
Every managed package is current and consistent.
```

"Not on PyPI" is the one way pandoc's entry differs from the rest: it isn't a Python package, so there's no release index to check against or take "newest" from - `prodockit pins` (no `--check`) still prompts for it like any other package, but you type the version rather than pressing Enter for the newest. `--set` works exactly the same as anywhere else:

```
prodockit pins --set pandoc=3.10.1
```

This project also pins the runner/image used to build it - `ubuntu-24.04` in `.github/workflows/docs.yml`, `python:3.13` in `.gitlab-ci.yml` - since a runner label or image tag has no package index either, only `--set` to a version you choose yourself, e.g. `prodockit pins -p ubuntu --set ubuntu=24.04`.

`prodockit` itself uses a minimum version rather than an exact pin: this userguide requires 0.41.0 or newer because that release introduced the extension-owned back-of-book index configuration. It isn't one of `prodockit pins`' five default packages, so managing or checking that floor needs naming explicitly:

```
prodockit pins --check -p zensical -p weasyprint -p prodockit
```

> **Tip**
>
> `prodockit pins --check` only reports what's already pinned - it doesn't run on every push here, since a pin going out of date on PyPI is expected over time, not something that should fail an unrelated change's own build. Watching for a newer release *actually mattering* is the next section's job instead.

> **Needs prodockit 0.17.5 or newer to see this line at all**
>
> `prodockit[index]>=0.41.0` has an extras bracket between the name

> and the version - a shape `prodockit pins` couldn't parse before 0.17.5, silently reporting "not declared anywhere" rather than failing to parse (prodockit-extensions#156). Run the check with an older `prodockit` installed and it passes for the wrong reason: it never saw the declaration to disagree with.

8.4.1 Watching for drift

A newer release existing isn't interesting on its own - PyPI already answers that. Whether taking it would *change what gets published* is the thing worth someone looking at, and the only way to answer that is a real build: with the pinned versions, then again with the newest ones, diffed byte for byte.

Both `.github/workflows/drift.yml` and the `drift` job in `.gitlab-ci.yml` do exactly this, weekly:

```
build (pinned) → build (newest) → diff the PDF and website → report
```

It reports rather than fails - `allow_failure: true` on GitLab, and the GitHub job skips its own report step entirely once nothing actually changed - and keeps a single open issue updated in place rather than filing a fresh one every week. A build order flip here (`zensical build` after `prodockit pdf`, not before) matters more than it looks: `zensical build` copies the PDF into the published site, so building it first would make every comparison a false positive that looks exactly like drift.

All three pinned packages are watched, not just Zensical and WeasyPrint - `prodockit` is upgraded and reported alongside them, since it renders content directly and can change output the same way.

> **The GitLab job needs two things set up once**
>
> A weekly pipeline schedule pointed at this project (the job's own `rules:` only stop it running on a normal push, they don't create the schedule), and a `DRIFT_TOKEN` CI/CD variable - a project access token with the `api` scope, masked and protected - so the job can open or update an issue. The GitHub workflow needs neither: `schedule:` in the workflow file is the trigger, and the built-in `github.token` already has enough access to open an issue in the same repository.

8.4.2 Taking an upgrade

When a drift issue reports something worth having:

```
prodockit pins -p zensical -p weasyprint -p prodockit  # accept the
suggested version, or type one
prodockit pdf                                          # rebuild -
PDF first ...
zensical build --clean --strict                        # ... then
the site
```

Then diff the built output against the previous version before committing - the same comparison the drift job already made, just with a person deciding rather than reading a report. Repeat for the runner image if that's what moved (`prodockit pins -p ubuntu`, or `-p python` on GitLab), since it isn't upgraded by the same command.

See prodockit-extensions' own Taking an upgrade and Version pinning and drift sections for the full reasoning behind each step - this project's two files are that same recipe, adapted from a `pyproject.toml`-based Python package to a `requirements.txt`-based Zensical site.

8.5 Publishing

This project publishes from two pipelines, kept deliberately in step:

- `.github/workflows/docs.yml` — GitHub Actions, publishing to GitHub Pages.
- `.gitlab-ci.yml` — GitLab CI/CD, publishing to GitLab Pages, for a mirrored copy.

Both install the same tooling and run the same commands, in the same order:

```
install dependencies → prodockit pdf → zensical build --clean --
strict → run tests → publish
```

The tests run *after* both builds because they check the built output - a diagram that reached the PDF as raw source fails the pipeline rather than being published.

The GitHub pipeline has one further step the GitLab one does not: after publishing, it polls the live site and fails the run if the page it fetches doesn't match what was just deployed. A successful deploy is not proof the site is actually serving it - GitHub Pages has, more than once, reported success and quietly kept serving the previous build.

8.5.1 Four settings that are easy to get wrong

Each of these is invisible when wrong: nothing fails, the site just publishes something incorrect.

Full git history. The cover page's release number comes from `git describe`

`--tags`. Both GitHub Actions and GitLab CI clone *shallowly* by default, which fetches **no tags at all**, so the release line silently disappears - while working perfectly on your own machine, where you have the full history.

```
# GitHub Actions
- uses: actions/checkout@v5
  with:
    fetch-depth: 0

# GitLab CI
variables:
  GIT_DEPTH: "0"
```

Rebuild when a release is published - but not from the tag. A release is normally tagged *after* the commit is pushed, and that push is what triggers the deploy, so the first deploy after a release still shows the *previous* version. Something has to rebuild once the tag exists.

The obvious answer - trigger the deploy on the tag as well - is the wrong one, and fails quietly on both hosts. On GitHub, a Pages deployment carrying a tag ref is accepted, reported successful, and then never served (issue #43). On GitLab it *is* served, but it publishes whatever the tag points at, which is older than `main` whenever a release has been overtaken by newer commits (issue #58).

So neither pipeline builds tags. GitHub rebuilds against `main` from a separate `redeploy-after-release.yml` workflow; the GitLab mirror gets the same result from pushing tags before the branch, as in Mirroring to a second host below.

One deploy at a time. Publishing a release shortly after merging its version bump starts two deploys at once, and they race. The one that finishes last does not necessarily win - so the site can end up serving the older build, with nothing reporting a problem. GitHub uses a `concurrency` group; GitLab uses `resource_group`.

The Puppeteer variable. `mermaid-cli` drives Chrome through Puppeteer to draw diagrams. Set `PUPPETEER_SKIP_DOWNLOAD`, **not** the older `PUPPETEER_SKIP_CHROMIUM_DOWNLOAD` - Puppeteer renamed it, and the current version ignores the old name, downloading a full Chrome build on every run before discarding it.

> **Copy a working pipeline rather than assembling one**
>
> prodockit's own Continuous integration page has complete, working recipes for both GitHub Actions and GitLab CI, with the reasoning behind each step. This project's two pipeline files are the same recipe in use.

8.6 Mirroring to a second host

Both pipelines above assume your code already exists on both hosts. Getting it there in the first place - and keeping it there as you keep writing - is a separate, one-off step: a plain remote operation, nothing platform-specific. A remote is just a name pointing at a URL; a repository can have as many as it likes, on as many different hosts as it likes, regardless of which one you originally cloned from.

Add the second host as a new remote (once):

```
git remote add mirror <url-of-the-second-host's-copy-of-this-repo>
```

Then, whenever you want to bring the mirror up to date with whatever you've pushed to your primary host:

```
git pull origin main    # bring your local main up to date with your
primary host
git push mirror --tags  # release tags first, so the build below can
see them
git push mirror main    # then the branch - this is the push that
publishes
```

That last push is what triggers the mirror's own CI/CD pipeline - see Publishing above. There's nothing to automate here unless you want to: pushing a fresh copy up before or after a release is a manual step, the same size as any other Git command.

> ⚠ **Push the tags first, not last**
>
> The order matters, and getting it wrong is invisible. The cover page's release number comes from `git describe --tags`, which can only find a tag that is already on the mirror when the build runs. Push the branch first and its build sees the *previous* tag, so the mirror publishes with a release number one behind - correctly built, quietly wrong.
>
> Tags first, and the branch build resolves the new release by itself.

> 📣 **Fetch tags before you push them**
>
> A release created through the host's web interface or `gh release create` makes the tag *on the server*, not in your local clone - so `git push mirror --tags` has nothing new to send and silently pushes nothing. Refresh first:

```
git fetch origin --tags
git describe --tags --abbrev=0   # should show the release you
just made
```

Both hosts need their own SSH key

Use the SSH URL for the mirror remote, not HTTPS, matching Generate and configure ssh keys for Git in Install tooling - the same guide already covers setting up a separate key per host and picking the right one automatically via `~/.ssh/config`.

8.6.1 GitHub to GitHub

Mirroring between two GitHub-hosted copies - your own account and an organisation's, say, or two separate GitHub instances - looks like:

```
git remote add mirror git@github.com:your-org/your-repo.git
git pull origin main
git fetch origin --tags
git push mirror --tags
git push mirror main
```

8.6.2 GitHub to GitLab

Mirroring to a GitLab instance - an institution's own self-hosted instance, or GitLab.com - is exactly the same pattern, just a different host in the URL:

```
git remote add mirror git@gitlab.example.org:your-namespace/your-
repo.git
git pull origin main
git fetch origin --tags
git push mirror --tags
git push mirror main
```

Confirm both remotes are set up correctly at any point with `git remote -v`.

Why the GitLab pipeline doesn't build tags

`.gitlab-ci.yml`'s `pages` job runs on the default branch only. A tag build would publish whatever the tag points at, which is older than `main` whenever a release has been overtaken by newer commits - so syncing after such a release would republish the older site over the newer one,

> successfully and silently. With tags pushed first, the branch build already resolves the right release number, so building tags separately gains nothing. See prodockit-userguide#58.

8.7 Checks worth having

Three commands turn the quiet failures above into loud ones:

```
prodockit sync-repo --check   # config drifted from your git remote?
prodockit pins --check        # build inputs behind PyPI, or pinned
inconsistently?
python -m pytest test/        # diagrams, maths and links in the
built output
```

`prodockit sync-repo --check` writes nothing and exits non-zero if your repository links, header icon or README badges no longer match the remote you are actually publishing from - useful after forking or moving a project. `prodockit pins --check` does the same for the pins in Pinning build inputs above. See Testing for the third.

8.8 Where to go next

Continue to Additional tooling for optional extras - VS Code extensions, GitLens, and Vale - or see Testing for the checks that guard the failures described above.

9. Additional tooling

This page covers optional tooling you can add on top of the core Zensical workflow covered in Install tooling and Start editing: Visual Studio Code extensions that connect the editor directly to GitLab or GitHub, view Git history, and check your writing; converting existing Word, PowerPoint, or PDF content into Markdown; and keeping your document's images small. You don't need any of this to write or publish your document - add whichever pieces are useful to you, and skip the rest. Each section below assumes no prior Linux or command-line experience, and spells out every step.

9.1 Installing Visual Studio Code extensions

An extension is a small add-on that adds extra features to Visual Studio Code. The extensions covered here let you manage GitLab or GitHub directly from the editor, view your Git history inline, and check your writing for spelling, grammar, and style issues as you type.

9.1.1 Installing GitLab or GitHub extensions

VS Code doesn't require any extensions to work with GitLab or GitHub, but installing the relevant extension can make it easier to manage your documentation without leaving the editor. Some features of these extensions include:

- Viewing issues and merge requests directly in VS Code.
- Creating and managing issues and merge requests directly in VS Code.
- Viewing and managing your GitLab or GitHub repositories directly in VS Code.
- Viewing and managing your GitLab or GitHub CI/CD pipelines directly in VS Code.

Install the extension

GitLab

1. Open VS Code.
2. Click the **Extensions** icon in the left-hand Activity Bar (or press `Ctrl+Shift+X` on Windows/Linux, `Cmd+Shift+X` on macOS).
3. In the search box at the top of the Extensions view, type `GitLab`.
4. Find **GitLab** (published by GitLab) in the results and select **Install**.

5. Wait for the installation to finish - VS Code shows a notification once it's ready.

GitHub

1. Open VS Code.
2. Click the **Extensions** icon in the left-hand Activity Bar (or press `Ctrl+Shift+X` on Windows/Linux, `Cmd+Shift+X` on macOS).
3. In the search box at the top of the Extensions view, type `GitHub Pull Requests and Issues`.
4. Find **GitHub Pull Requests and Issues** (published by GitHub) in the results and select **Install**.
5. Wait for the installation to finish - VS Code shows a notification once it's ready.

9.1.2 Configuring GitLab or GitHub extensions

Once you've installed the extension, it needs permission to access your GitLab or GitHub account. The most reliable way to do this is with a personal access token (PAT) - a long, randomly generated code that works like a password, but you can limit it to just this purpose, give it an expiry date, and revoke it at any time without changing your main account password.

1. Create a personal access token (PAT) for your account:

Create a Personal Access Token (PAT)

GitLab

1. Log in to **GitLab** (either gitlab.com or your organisation's self-managed instance) in a web browser.
2. In the top-right corner, click on your **profile avatar** and select **Edit profile**.
3. On the left-hand sidebar, select **Access > Personal access tokens**.
4. Click **Add new token** and fill out the following details:
   - **Token name:** Give it a clear name (for example, `VS Code Extension`).
   - **Expiration date:** (Optional) Set an expiration date according to your team's security policy. You can click on the date and select the last date available.
5. Under **Select scopes**, check the **api** scope.

6. Click **Create token** and copy the token string.

> **Warning**
>
> Copy the token string **immediately**. GitLab will only show it to you once; if you refresh or leave the page, it is gone forever.

GitHub

VS Code's built-in GitHub integration normally signs you in through your browser (OAuth) rather than a token, so you usually won't need a PAT for GitHub. If you use GitHub Enterprise Server and the browser sign-in doesn't work, create a personal access token instead:

1. Log in to **GitHub** (either github.com or your organisation's self-managed instance).
2. In the top-right corner, click on your **profile avatar** and select **Settings**.
3. On the left-hand sidebar, select **Developer settings > Personal access tokens > Fine-grained tokens**.

   > **Note**
   >
   > GitHub may ask you to reauthenticate before you can proceed to the next step.

4. Click **Generate new token** and fill out the following details:
   - **Token name:** Give it a clear name (for example, `VS Code Extension`).
   - **Expiration date:** (Optional) Set an expiration date according to your team's security policy. You can click on the date and select the last date available.
5. Select the **Repository access** level for the token. Choose **All repositories** if you want the token to have access to all your repositories, or choose **Only select repositories** and specify the repositories you want to grant access to.

6. Under **Permissions**, select **+ Add permissions** and add the following:
    - **Read and Write** access to **Contents** and **Pull Requests**.
    - **Read-only** access to **Metadata**.
7. Click **Generate token** and copy the token string.

> ⚠ **Warning**
>
> Copy the token string **immediately**. GitHub will only show it to you once; if you refresh or leave the page, it is gone forever.

2. Configure the extension to use the token you just created:

Configure VS Code for Git repo access

GitLab

GitLab's browser sign-in (OAuth) may not work in some environments, so we'll use the personal access token (PAT) instead.

1. Open VS Code and open the **Command Palette**:
    - For **macOS**, press `Cmd+Shift+P`.
    - For **Windows** or **Linux**, press `Ctrl+Shift+P`.
2. Type `GitLab: Authenticate` and press `Enter`.
3. Choose your GitLab instance:
    - Select **GitLab.com** if you use a public cloud instance.
    - Select **Add new instance URL** if you use a self-hosted instance, and type your full domain (for example, `https://gitlab.yourorganisation.com`).
4. Select **Enter an existing token**.
5. Paste in the personal access token (PAT) you created earlier and press `Enter`.

The extension instantly validates the token. If successful, your

GitLab status updates in the bottom status bar, and your GitLab sidebar panel populates with your issues and merge requests.

GitHub

The built-in GitHub Authentication provider in VS Code uses a browser sign-in (OAuth) by default.

1. Click the **Accounts** icon (the profile silhouette) in the bottom-left corner of VS Code.
2. Select **Sign in with GitHub** (this may appear under a Copilot or Settings Sync prompt, depending on your VS Code version).
3. Click **Allow** when asked to open the external website.
4. Your browser opens GitHub to authorize the app. Click **Authorize Visual Studio Code**.
5. **If the browser doesn't redirect back to VS Code:** GitHub displays a page saying "*If your browser does not redirect you...*" alongside a **blue box containing an authorization token.** Copy that token.
6. Return to VS Code. Look at the very bottom **Status Bar**; it says `Signing in to github.com....`.
7. Click that status bar text. An input box opens at the top of your editor.
8. Paste the token you copied from the browser page and press `Enter`.

> **Note for GitHub Enterprise Server users**
>
> If your company hosts its own private GitHub Enterprise Server, the browser flow won't always work out of the box. To use a PAT directly, start the sign-in prompt, click **Cancel** on the browser authorization pop-ups, and VS Code automatically changes its prompt to a direct text field asking you to paste your Enterprise PAT.

9.1.3 Installing GitLens for commit history and blame

GitLens is a VS Code extension that adds inline "blame" annotations - showing who last changed each line, and when - directly above your text, along with a visual commit graph and richer history browsing. It's especially

useful once you're using the branches and issues workflow from Managing branches and issues.

1. Click the **Extensions** icon in the left-hand Activity Bar (or press `Ctrl+Shift+X` on Windows/Linux, `Cmd+Shift+X` on macOS).
2. In the search box, type `GitLens`.
3. Find **GitLens — Git supercharged** (published by GitKraken) in the results and select **Install**.
4. Once installed, a small annotation appears above the line your cursor is on, showing the last commit that changed it. Hover over it for full details, or select the **GitLens** icon in the Activity Bar for the commit graph and history views.

9.1.4 Installing Code Spell Checker for lightweight spell checking

If Vale below feels like more setup than you need right now, Code Spell Checker is a lighter alternative (or a useful complement to it) that underlines misspelled words directly in the editor as you type, with no configuration files required.

1. Click the **Extensions** icon in the left-hand Activity Bar (or press `Ctrl+Shift+X` on Windows/Linux, `Cmd+Shift+X` on macOS).
2. In the search box, type `Code Spell Checker`.
3. Find **Code Spell Checker** (published by Street Side Software) in the results and select **Install**.
4. Misspelled words are now underlined in the editor as you type. Right-click an underlined word for suggested corrections, or to add it to your personal dictionary if it's a term you use often (such as a project name or acronym).

9.1.5 Install vale to check for grammar, spelling, and style issues

Vale is a syntax and style checker for writing. You can use it to check your documentation for grammar, spelling, and style issues.

1. Install Vale for your operating system. Open a terminal if you don't already have one open, then follow the steps below:

Install Vale

macOS

```
brew install vale
```

Windows

1. Open PowerShell as an administrator: press the `Windows` key, type `PowerShell`, then either press `Ctrl+Shift+Enter`, or right-click **Windows PowerShell** in the results and select **Run as administrator**.
2. Use the Microsoft Windows Package Manager (winget) to install Vale:

   ```
   winget install --id errata-ai.Vale
   ```

Linux (Ubuntu)

1. If you don't already have `snapd` installed, install it:

   ```
   sudo apt update
   sudo apt install snapd
   ```

2. Once you've installed `snapd`, install Vale:

   ```
   sudo snap install vale
   ```

See the Vale installation page for other Linux distributions.

2. Create a `.vale.ini` file in the top-level directory of your project (the same folder that contains `zensical.toml`). This file configures Vale and specifies the rules and styles used to check your documentation.

   In VS Code, right-click your project's root folder in the Explorer pane and select **New File...**. Name the file `.vale.ini` (including the leading dot), then paste in the following content as a starting point:

   ```
   StylesPath = styles
   MinAlertLevel = suggestion
   Packages = Microsoft, Readability, proselint

   [*.{md,rst,asciidoc,html}]
   BasedOnStyles = Vale, Microsoft, Readability, proselint

   Vale.Terms = YES
   Vale.Avoid = YES
   Vale.Spelling = YES

   Microsoft.OxfordComma = YES
   Microsoft.Passive = YES
   Microsoft.Dashes = YES
   Microsoft.Spacing = YES
   Microsoft.Wordiness = YES
   Microsoft.We = YES
   ```

3. Create a `styles` directory in the top-level directory of your project. In VS Code, right-click your project's root folder in the Explorer pane and select **New Folder...**, then name it `styles`.

   Then synchronise the styles specified in `.vale.ini`. Open a new terminal (or PowerShell) for this, and make sure it's actually sitting in your project's root directory first:

   ```
   cd path/to/your-project
   ```

   Then run:

   ```
   vale sync
   ```

4. Install the Vale VS Code extension so Vale's suggestions appear directly in the editor:

   1. Click the **Extensions** icon in the left-hand Activity Bar (or press `Ctrl+Shift+X` on Windows/Linux, `Cmd+Shift+X` on macOS).
   2. In the search box, type `Vale`.
   3. Find **Vale** (published by Chris Chinchilla) in the results and select **Install**.
   4. Restart Visual Studio Code once the installation finishes.

When you open a Markdown file in Visual Studio Code, the Vale extension automatically checks it for grammar, spelling, and style issues based on the rules and styles you configured. View the results in the **Problems** panel: **View > Problems**, or the keyboard shortcut `Ctrl+Shift+M` (`Cmd+Shift+M` on macOS).

Vale generates a large number of suggestions, some of which aren't relevant to your documentation. You can ignore these and focus on the suggestions that are relevant to your writing style and the requirements of your documentation.

5. One of the most prominent suggestions is to change from passive voice to active voice. This is a good suggestion and recommended in business writing, but it can take time to change all instances of passive voice to active voice.

   If you need some help with this, here are a few websites that can help you understand how to change from passive voice to active voice:

   - BBC Bitesize
   - Communication for Professionals
   - University of York

   You can also change the `Microsoft.Passive` rule in the `.vale.ini` file to `NO` if you find it too difficult to change all instances of passive voice to active voice.

9.2 Converting existing documents to Markdown

If you're starting from content you've already written elsewhere - a Word report, PowerPoint slides, an Excel table, a PDF - you don't have to retype it. anydoc converts Word document conversionWord, PowerPoint, Excel, OpenDocument, RTF, EPUB, CSV, and PDF files into clean Markdown, close enough to what this template already expects that you can paste the result straight into a page and clean up from there.

The simplest option, needing no installation, is anydoc's own browser demo - it runs the conversion locally as WebAssembly, so your document never leaves your machine. Drag a file in, and copy the Markdown it produces.

If you'd rather convert from the command line - useful for several files, or if you want to redo the conversion after editing the original - run it with `npx`, which downloads the tool the first time and reuses it afterwards. You already have Node.js installed from Install the diagram and maths tooling:

```
npx @firecrawl/anydoc report.docx -o report.md
```

Replace `report.docx` with the file you're converting, and `report.md` with

wherever you want the Markdown written. Leave off `-o report.md` to print the result to the terminal instead.

> **Check the result before you rely on it**
>
> Automatic conversion is a starting point, not a finished page - tables, footnotes, and anything with complex formatting are the most likely to need a manual fix afterwards. Review the Markdown against the original before you build on it, the same way you would proofread text you'd typed yourself.

9.3 Optimising images before committing

When you commit screenshots and diagrams to the Git repository, the PDF build also embeds them in the generated PDF, so a handful of large, uncompressed images can noticeably slow down cloning the repository and increase the size of the published PDF. Optimising (compressing) an image before you commit it usually shrinks it considerably with no visible loss of quality.

The simplest option, needing no installation, is Squoosh - a free, browser-based image compressor from Google. Drag your screenshot into the page, choose a format and quality, and download the smaller result to use in place of the original.

If you'd rather compress images from your desktop without opening a browser each time, install a dedicated tool instead:

Install an image optimiser

macOS

```
brew install --cask imageoptim
```

Once installed, drag image files onto the ImageOptim window (or its Dock icon) to compress them in place.

Windows

Download and run the installer for FileOptimizer - there's no reliable single winget package for it. Once installed, drag image files onto the FileOptimizer window to compress them in place.

Linux (Ubuntu)

```
sudo apt update
sudo apt install pngquant jpegoptim
```

Then compress an image from the terminal, for example:

```
pngquant --ext .png --force docs/starthere/images/example.png
jpegoptim docs/starthere/images/example.jpg
```

9.4 Where to go next

This is the last of the step-by-step "Start Here" chapters. Shell commands is a standalone reference you can return to any time you need a reminder of a terminal command - it isn't something you need to read in order.

10. Shell commands

You will be using Shell commands if you are operating on Linux or macOS to write your documentation. This section serves as a refresher on the essential commands for navigating and managing your system using either bash or zsh.

> **Tip**
>
> You can find a more detailed reference to the shell command line on linuxcommand.org.

10.1 Navigation and basic info

Knowing where you are and how to move is the first step to understanding bash/zsh.

Table 10.1. Basic navigation commands

Command	Action
`pwd`	Print Working Directory: Shows exactly where you are.
`ls`	List: Shows files in the current folder.
`ls -la`	List All: Shows hidden files and detailed info (sizes, dates).
`cd [dir]`	Change Directory: Move to a folder (For example, cd Documents).
`cd ..`	Go Up: Moves one level up to the parent folder.
`cd ~`	Home: Takes you back to your user folder.
`clear`	Cleans up the terminal screen (Shortcut: Cmd + K).

10.2 File and folder operations

Next, here are the shell commands to manage directories and files. You will be using these commands to create, copy, move, and delete files and folders.

> **Warning**

> A shell doesn't have a "Trash" bin. When you delete something here, it's usually gone for good.

Table 10.2. File and folder operations

Command	Action
`mkdir [name]`	Make Directory: Creates a new folder.
`touch [file]`	Creates a new empty file.
`cp [src] [dest]`	Copy: Duplicates a file. Use cp -r for folders.
`mv [src] [dest]`	Move: Also used to rename files.
`rm [file]`	Remove: Deletes a file.
`rm -rf [dir]`	Force Remove: Deletes a folder and everything inside (Use with care!).
`cat [file]`	Displays the entire contents of a file in the terminal.
`less [file]`	Opens a file for reading (press q to exit).

10.3 Editing files

You will need to edit files in the terminal. There are many editors available, each with its own strengths and weaknesses. Here are some popular options:

Table 10.3. Terminal text editors

Editor	Style	Best For
`nano`	Lightweight / Modeless	Quick, beginner-friendly configuration edits.
`vim` / `neovim`	Modal / Highly Keyboard-Driven	Rapid editing, coding, and heavy terminal workflows.
`micro`	Modern / Modeless	A modern terminal editor with intuitive Ctrl+C/ Ctrl+V shortcuts and mouse support.
`helix`	Modal / Modern	A modern, fast terminal editor with built-in language server support and multiple cursors.
`emacs`	Extendable Environment	Users who want an entire operating system of utilities inside their editor.

If you are new to terminal editors, start with `nano` or `micro`. If you want to learn a more powerful editor, try `vim` or `helix`.

10.4 Administrative and permissions

Often the permissions of files and folders will need changing. Here are some commands to help you manage permissions and ownership.

> **Note**
>
> If you get a "Permission Denied" error, you likely need sudo.

Table 10.4. Administrative and permissions commands

Command	Action
`sudo [cmd]`	SuperUser Do: Runs a command with admin privileges.
`chmod 755 [file]`	Changes permissions (755 is standard for executable scripts).
`chown [user] [file]`	Changes the owner of a file.
`history`	Shows a list of all recently used commands.

10.5 Viewing content

Handling text files with shell commands.

Table 10.5. Viewing file content

Command	Action
`cat`	Concatenate: Dumps the whole file to your screen. Best for short files.
`tac`	Reverse cat: Displays the file starting from the last line. Great for seeing the newest entries in a log first.
`nl`	Number Lines: Displays file content with line numbers. Use `nl -ba` to number every single line, including blanks.
`more`	The "classic" version. It lets you scroll down using the Spacebar. Once you reach the end, it exits automatically.
`less`	A more powerful version of 'more'. It doesn't load the whole file at once (it's faster). You can scroll up (using the arrow keys) and you can search within the text (press / then type your word).

Command	Action
`head -n 20 [file]`	Shows you the top 20 lines of a file.
`tail -n 20 [file]`	Shows you the bottom 20 lines.
`tail -f [file]`	This is the "Live" mode where the option keeps the file open and updates the screen in real-time from file additions. This is how developers watch server logs as they happen.

10.6 Searching and filtering

Essential commands for finding files and file contents.

Table 10.6. Searching and filtering commands

Command	Action
`grep "word" [file]`	Searches for a specific string inside a file.
`find . -name "*.txt"`	Finds all .txt files in the current directory and subdirectories.
`[cmd] | grep "word"`	The Pipe (|) takes the output of one command and sends it to another.

10.7 Essential keyboard shortcuts

There are some essential keyboard shortcuts to help you navigate the terminal more efficiently.

Table 10.7. Essential keyboard shortcuts

Shortcut	Action
`Tab`	Auto complete. Start typing a folder name and hit Tab. If it's unique, it will finish it for you.
`Arrow Up/ Down`	Scroll through your previous commands.
`Ctrl + C`	Cancel/Kill the currently running command.

10.8 Stream shortcuts

Programs and files pass data between themselves using the operators below.

Table 10.8. Stream redirection operators

Shortcut	Action
`>`	Overwrite: Sends output to a file, deleting whatever was there before. For example, `echo "hello" > file.txt`
`>>`	Append: Adds output to the end of a file without deleting the old stuff. For example, `echo "world" >> file.txt`
`|`	The Pipe: takes the "Standard Out" of the left command and shoves it into the "Standard In" of the right command.

10.9 Job queue control

There are commands to manage jobs running in the background or suspended in the terminal. It's important to understand that a job is a command that's running in the terminal. You can have multiple jobs running at the same time, and you can control them using these commands.

Table 10.9. Job control shortcuts

Shortcut/ Command	Action
`[cmd] &`	Start in Background: Runs the command immediately in the background so you can keep typing.
`Ctrl + Z`	Suspend: Freezes the current foreground task and sends it to the background as a "stopped" job.
`Ctrl + C`	Interrupt: Cancel/Kill the currently running command.
`Ctrl + \`	Quit: Force-quits a task and creates a "core dump" (Not often needed for casual use).
`jobs`	Lists all background/suspended jobs.
`fg %[id]`	Foreground: Brings a job back to the front so you can interact with it.
`bg %[id]`	Background: Tells a suspended job to start running again, but in the background.
`kill %[id]`	Terminates a job in your list using its job number.

10.10 Jobs and processes

There are commands available to manage processes and jobs running on your system. A process is a program that's currently running, and each process has a unique Process ID (PID).

Table 10.10. Process management commands

Command	Action
`ps`	"Process Status." Use ps aux to see every running process.
`top`	A live, updating list of processes (sorted by CPU/Memory). Press q to exit.
`htop`	A prettier, interactive version of top (install via brew install htop).
`pgrep [name]`	Finds the PID of a process by name (For example, pgrep Chrome).
`kill [PID]`	Sends a "polite" request to a process to stop (SIGTERM).
`kill -9 [PID]`	The "Executioner." Force-kills a process instantly (SIGKILL).
`killall [name]`	Kills all processes with a certain name (For example, killall Finder).

10.11 Where to go next

Continue to Testing if you're contributing to the template itself, or jump straight to Finalising your document in Customise document content once your report is ready to submit.

11. Testing

Every other page in this section explains how to *use* this template to write your report. This page is different: it documents the test suite in `test/`, which checks that this template's own prodockit-specific features (References, Acronyms, Glossary, Appendix numbering, word-count exclusions, and so on) actually work and haven't regressed - a contributor/maintainer tool, not something you need while writing your assignment.

> **Why this exists**
>
> Every feature and bug fix in this template used to be verified by hand - a one-off script, eyeballed once, then thrown away. Nothing stopped a later change from silently re-breaking something already fixed. The test suite in `test/` replaces that with checks that run automatically, in CI, on every push.

11.1 Running the test suite

The tests check the *built output* - the website in `public/` and the PDF at `docs/site_documentation.pdf` - not the build process itself, so build both first:

```
pip install -r requirements.txt -r testrequirements.txt
prodockit pdf
zensical build --clean --strict
python test/run_tests.py
```

`testrequirements.txt` is separate from `requirements.txt` - it's only needed to run the tests (locally or in CI), not to build your own report from this template.

Tests are organised into **test batches**, one per `test/test_*.py` file, each reporting its own pass/fail (see Test batches below for what each one covers):

```
python test/run_tests.py --list            # list available batches
python test/run_tests.py --batch links     # run just one batch
python test/run_tests.py -- -k caption     # extra args passed
through to pytest
```

Running just one batch is the fast path while you're actively working on a specific capability - you don't have to wait on the rest of the suite (including a couple of batches that parse the whole PDF) to check whether your change works.

11.2 Adding a new test

Most new tests belong in an existing batch - add a `test_*` function to the matching `test/test_<batch>.py` file. Start a new batch (a new `test/test_<name>.py` file) only for a genuinely new category of concern; `test/run_tests.py --list` picks up any `test_*.py` file automatically, no registration needed.

Shared fixtures live in `test/conftest.py` and cover both the failure-with-a-clear-message case (missing build artifacts) and the common lookups most tests need:

Table 11.1. Shared fixtures in test/conftest.py

Fixture	Gives you
`pdf_doc`	The built PDF, already opened with PyMuPDF (`fitz`)
`pdf_full_text`	The PDF's text, one string per page
`public_dir`	Path to the built website (`public/`)
`public_html_files`	Every built HTML file, as a sorted list of `Path`s
`macros`	The project's own `macros.py`, imported live
`nav_pages` / `docs_dir`	The current `nav` list (docs_dir-relative paths) and `docs_dir` itself
`zensical_config`	`zensical.toml`, parsed

Two different styles of test both have a place here, and most batches mix both:

- **Black-box, against the built output** - most of `test_build.py`, `test_links.py`, `test_numbering.py`, `test_word_count.py`, `test_content.py`, and `test_pdf_structure.py` work this way: build the site/PDF, then check something true about the result (no `LAUNCH`-type links, chapter numbers sequential, a flagged page's words don't count, and so on).
- **Direct unit tests against a helper function**, with a hand-written markdown snippet, no build required - `test_fences.py` works this way throughout, calling functions like `macros._count_top_level_headings()` directly. Reach for this style when you're testing a specific function's behaviour on a deliberately constructed edge case (e.g. a heading shown literally inside a fenced code example) rather than something you'd have to go hunting for in the real docs to exercise.

Where a check needs "the real thing this feature is supposed to do" as a source of truth (e.g. which pages are flagged `is_appendix: true`), reuse the actual function from `macros.py` (via the `macros` fixture) or from `prodockit`

itself rather than re-implementing the same TOML/front-matter parsing a second time in the test - that would just test the test's own parser, not catch a real regression in the production code.

11.3 Test batches

Table 11.2. Test suite batches

Batch	Checks
`build`	Both builds actually produced usable output - the PDF opens with a sane page count, the website has more than just the cover page.
`links`	No `LAUNCH`-type or local-filesystem links in the PDF (see issue #19); every internal website link and same-page fragment resolves to a real page and id; every internal PDF cross-reference resolves to a real, in-range page (see issue #71).
`numbering`	Chapter numbers are sequential with no gaps or duplicates; appendix letters run A, B, C... with no gaps and don't consume a chapter number (see issue #24); every sub-heading's number matches its actual parent chapter/appendix.
`word_count`	Pages flagged `exclude_from_word_count: true` are actually subtracted from both the website's and the PDF's word count, not just present as an unused flag (see issue #23).
`content`	No un-translated template syntax leaks into the PDF as literal, visible text - unsubstituted `{WORDCOUNT}` / `{REPOURL}` markers on the cover page, or leaked attr_list `{: ... }` syntax on the Acronyms/Glossary/References pages.
`pdf_structure`	The cover page's computed fields (word count, repo URL) and the auto-generated Table of Contents are present and look like real data.
`index`	Back-of-book index generation is enabled through `prodockit.index` rather than former PDF-wide settings, and the built PDF contains its exact `Index` section with representative plain, nested, and code-styled entries (see issue #140).
`strict_build`	Both publishing pipelines run `zensical build --clean --strict`, and the contributor guide names the same final validation check while retaining `zensical serve` for interactive preview (see issue #141).

Batch	Checks
`fences`	The fence-skipping loops in `macros.py`'s heading/word counters hold up under nested combinations of headings, admonitions, and code blocks - not just a single flat fenced block. The PDF build's own equivalent primitives were retired in prodockit-template#92: `prodockit.pdf` renders pages through Zensical's real Markdown parser, so a heading/tab/admonition shown inside a fenced code example is never mistaken for the real thing in the first place.
`captions`	The PDF's `figure-caption` / `table-caption` numbering (see Captions) is correctly prefixed with the page's chapter number/appendix letter, and no `///` block syntax leaks into the PDF as literal text - checked against the real, already-built PDF. Manual number overrides, custom id/class, and prepend/append position are `pymdownx.blocks.caption`'s own behaviour since prodockit-template#92, identical to the website's - no longer a PDF-build-specific concern to unit-test here.
`markdown_foundations`	The core Markdown syntax documented in Markdown basics - headings, text formatting, links/images, lists (including the 4-space nesting rule - see issue #70), code blocks, tables, horizontal rules, task lists, and blockquotes - rendered through this project's real `markdown.Markdown()` extension config, not a hardcoded copy of it, plus a real `pandoc` subprocess check for a genuine website/PDF nesting discrepancy.
`zensical_basics`	The Zensical-specific extensions documented in Zensical basics - admonitions (all 12 configured types, custom/empty titles, nesting, collapsible details), code blocks (line highlighting, line numbers, titles, annotations, inline highlighting), content tabs (nested, anchor ids), images (figure/figcaption, alignment, lazy loading, light/dark hash fragments), diagrams (all 5 officially-supported Mermaid types), footnotes, formatting (mark, keys), icons/emojis, maths (arithmatex wrapping), task lists, and tooltips/abbreviations - cross-checked against the upstream `zensical.org/docs/authoring/*` pages this page itself links to, not just its own worked examples, plus end-to-end checks against this page's own "live example" content in the real, already-built PDF (catching, among other things, `==mark==` / `++keys++` leaking as literal text there - see issue #72).

Batch	Checks
`customisation`	Every remaining template-specific customisation point in Customisation not already covered by a more specific batch above - site logo/metadata/copyright/repository link, colour scheme, header background, fonts, icons, feature toggles, extra CSS/JS, navigation structure, institution branding, `.pdf-only` / `.web-only` content, the Download PDF button, page header/footer content, page size/margins (see issue #51), and screenshots - reflects what `zensical.toml` actually configures, not just something plausible-looking. Where a value is duplicated by necessity between the website's CSS and the PDF's (references/acronyms/glossary spacing, `.screenshot` framing), checks both copies stay in sync with each other.

11.4 Where to go next

This is the last of the "Start Here" reference pages. Continue to Finalising your document in Customise document content once your report itself is ready to submit - removing `starthere/` (this page included) is part of that step.

Appendix A. Acronyms

The following acronyms and abbreviations are used throughout this document. `\gls{css}` is also referenced by Customisation's live example.

AI - Artificial Intelligence
API - Application Programming Interface
CI/CD - Continuous Integration/Continuous Deployment. See the glossary for what this means in practice.
CLI - Command Line Interface
CPU - Central Processing Unit
CSS - Cascading Style Sheets. See the glossary for what this means in practice.
DCO - Developer Certificate of Origin
GPG - GNU Privacy Guard
GUI - Graphical User Interface
HTML - HyperText Markup Language
HTTPS - HyperText Transfer Protocol Secure
JS - JavaScript
JSON - JavaScript Object Notation
MR - Merge Request. See the glossary for what this means in practice.
PAT - Personal Access Token. See the glossary for what this means in practice.
PDF - Portable Document Format. See the glossary for what this means in practice.
PID - Process ID
PR - Pull Request. See the glossary for what this means in practice.
SSH - Secure Shell. See the glossary for what this means in practice.
TOML - Tom's Obvious, Minimal Language
UI - User Interface
URL - Uniform Resource Locator

Appendix B. Glossary

The following key terms are used throughout this document. `\gls{markdown-def}` is also referenced by Customisation's live example.

Admonition - A labelled, coloured callout box (Note, Tip, Warning, and so on) used to highlight information set apart from the main body text, without interrupting the flow of a paragraph.
Attribute list - Markdown syntax (`{: ... }`) that attaches an id, CSS class, or other HTML attribute to the element directly above it - the mechanism the cross-reference, citation, acronym, and glossary features are all built on.
Branch - An independent line of development within a repository, letting you make changes without affecting the default branch until they're ready to be merged.
CI/CD - The practice of automatically building, testing, and publishing a project every time a change is pushed, rather than as a separate manual step. See the Acronyms entry for the expansion.
Cascading Style Sheets - The language used to control the visual appearance (colours, spacing, fonts, layout) of a web page, kept separate from its content. See the Acronyms entry for the expansion.
Clone - Creating a full local copy of a remote repository, including its entire history, so you can edit it on your own computer.
Commit - A saved snapshot of your changes in Git, together with a message describing what changed and why.
Docs-as-code - An approach to writing documentation that uses the same tools and workflow as software development - plain text files, version control, and peer review - rather than a word processor or wiki.
Fenced code block - A block of code set apart from the surrounding text by a line of three or more backticks before and after it, optionally labelled with a language name for syntax highlighting.
Fork - Your own copy of someone else's repository, created on the hosting service (GitLab or GitHub) itself, which you can then clone and edit independently of the original.
Front matter - A block of YAML metadata at the top of a Markdown file, delimited by `---` lines, that configures how that page is built or displayed (its icon, whether it's an appendix, and so on) without appearing in the rendered content.
HEAD - Git's pointer to the commit your working directory currently reflects - normally the tip of whichever branch you have checked out.
Markdown - A lightweight markup language for formatting plain text, using a simple, readable syntax that converts into HTML for web publishing.
Personal access token - A long, randomly generated code that acts as a substitute for a password, scoped to a single purpose and revocable at any time without changing your main account credentials. See the Acronyms entry for the expansion.
Portable Document Format - A fixed-layout document format that looks the same regardless of the software, hardware, or operating system used to open it. See the Acronyms entry for the expansion.

Pull request - A request to merge changes from one branch into another, reviewed and discussed by collaborators before it's accepted. See the Acronyms entry for the expansion - GitLab calls the same thing a Merge Request.
Repository - The complete collection of a project's files and their full history of changes, tracked by Git and stored on a service such as GitLab or GitHub.
Secure Shell - An encrypted network protocol used to authenticate with, and send commands to, a remote computer - most commonly used here to connect securely to GitLab or GitHub without typing a password each time. See the Acronyms entry for the expansion.
Static site generator - A tool that converts a set of source files (such as Markdown) into a complete website of plain HTML pages ahead of time, rather than generating each page on demand when a visitor requests it.
Version control - A system for recording changes to a set of files over time, so you can review the history, compare versions, and revert to an earlier one if needed.

Appendix C. References

The following sources support the content presented in this document. `\citeref{skou2023}` is also referenced by Customisation's live example.

Chacon, S. and Straub, B. (2014) *Pro Git*. 2nd edn. New York: Apress. Available at: https://git-scm.com/book (Accessed: 18 July 2026).
Markdown Guide (n.d.) *Markdown Guide*. Available at: https://www.markdownguide.org/ (Accessed: 18 July 2026).
MathJax (n.d.) *MathJax documentation*. Available at: https://www.mathjax.org/ (Accessed: 18 July 2026).
Mermaid (n.d.) *Mermaid documentation*. Available at: https://mermaid.js.org/ (Accessed: 18 July 2026).
Microsoft (n.d.) *Visual Studio Code documentation*. Available at: https://code.visualstudio.com/docs (Accessed: 18 July 2026).
Python-Markdown (2023) *Python-Markdown documentation*. Available at: https://python-markdown.github.io/ (Accessed: 18 July 2026).
Shotts, W. (2019) *The Linux Command Line: A Complete Introduction*. 2nd edn. San Francisco: No Starch Press. Available at: https://linuxcommand.org/tlcl.php (Accessed: 18 July 2026).
Skoulikari, A. (2023) *Learning Git: A Hands-On and Visual Guide to the Basics of Git*. Sebastopol, CA: O'Reilly Media.
Vale (n.d.) *Vale documentation*. Available at: https://vale.sh/ (Accessed: 18 July 2026).
Zensical (n.d.) *Zensical documentation*. Available at: https://zensical.org/docs/ (Accessed: 18 July 2026).

Index

T

V

W

Z