How open-source projects work

This page answers: *your project is built on other people's open-source work, and is open source itself, so what does that let you do, what does it ask of you, and how do you work with the people whose code you depend on?*

A Tiny Tapeout repository is a small project sitting on top of several large ones. The template is Tiny Tapeout's; the `gds` job calls Tiny Tapeout's actions, which fetch Tiny Tapeout's support tools; those run LibreLane, cocotb, Icarus Verilog, Magic, KLayout and an open PDK, each maintained by someone else. The projects your work comes from are your **upstream**, and you're **downstream** of them. Most of the time you don't think about them, but when something breaks or you want to reuse someone's code, knowing how open-source projects work saves a great deal of time.

Licences

Both Tiny Tapeout templates ship a `LICENSE` file containing the Apache License 2.0, and Tiny Tapeout's terms require designs to be released under terms compatible with Apache-2.0. A licence is the permission that lets other people use, change and share your work, and it sets the conditions for doing so. The Apache licence's conditions are mild: roughly, keep the licence and the copyright notices with the code, and say what you changed.

Two habits follow. Leave the `LICENSE` file in your repository, because it's what makes your project open source. And when you copy code from another project, its licence comes with it: check that it's compatible before you copy, and keep the licence file next to the copied code.

Copying someone else's code

There are two ways to bring another project's code into yours. You can **vendor** it, which means copying the files into a folder of your repository, or you can add it as a **git submodule**, which records the other repository's address and one of its commits, and has git fetch the files from there. Tiny Tapeout's workflows check out submodules (`submodules: recursive` in every checkout step), so either way works on the runner.

`hart-1` vendors TinyQV, an open-source RISC-V core, into `src/tinyQV/`, and it does it well. The folder keeps TinyQV's own `LICENSE` (Apache-2.0), and it adds a file called `UPSTREAM.md`. Its first line names the project, its author and its licence, and the rest reads:

```
- Repo: https://github.com/MichaelBell/tinyQV
- Branch: ttsky25a (the version taped out on Tiny Tapeout SKY25a)
- Commit: baf7d4dc6fbdd7d348fa146c6bea1d6f6e03e664 (3 Sep 2025)

Copied: cpu/ (all RTL used by the core; latch_reg.v and qspi_flash.v left out as unused),
test/ (cocotb testbenches, minus the legacy qspi_flash test), verify/ (formal + lint).

Local changes:
- verify/lint.sh: dropped ../peri/*/*.v (peripherals not copied).
```

Those few lines answer the questions people will later have about the copy. A reviewer can compare the folder with that exact commit and confirm it's what it claims to be, which is how a 7,408-line pull request becomes reviewable, as Pull requests and review describes. Whoever updates TinyQV next knows which version they're starting from and which files were left out on purpose. And the one local change is written down, so it won't be lost or mistaken for an upstream bug.

A submodule records the address and commit automatically, but it makes local changes awkward, because the files belong to another repository. Vendoring keeps everything in one place and makes local changes easy, at the cost of having to write the provenance down yourself. Either is fine, as long as someone reading your repository can tell what came from where.

Working with an upstream project

When something in Tiny Tapeout's workflows behaves strangely, the answer is often already written down upstream, in the project's README, its FAQ, its issues or its commit history. The viewer re-run failure described in Reading a workflow file is an example. Its explanation is the message of Tiny Tapeout's own fix, commit `16c727e` in `tt-gds-action`:

```
fix(viewer): use a unique pages artifact name per run attempt

Re-running a failed viewer job keeps the github-pages artifacts of the previous attempts, and
deploy-pages then fails with "Multiple artifacts named github-pages were unexpectedly found
for this workflow run".
```

Reading upstream's recent commits took a few minutes and explained the failure, and comparing the commit's date with the dates of the tags showed which tags have the fix. That's a good order in which to look: your own log first, then upstream's documentation, then its history and issues, and only then ask.

When you do need to ask, ask where the people who can answer will see it. For a problem in a Tiny Tapeout template or action, that's the issues page of the repository concerned, or Tiny Tapeout's community, which the template's README links under "Join the community". For anything about the team's projects, ask in the team's Discord. Either way, include what you ran, the exact first error line, a link to the run, and the tag and commit you're on, so that someone can reproduce it.

If you find something to fix upstream, the route is the one from Templates, forks and clones: fork the repository, make the change on a branch, and open a pull request back to it. It's also the route you'll take into your partner's repository in Lab 04. Follow the project's own conventions. Tiny Tapeout's commit messages, for instance, start with a type and the part they touch, as in `fix(viewer): …`, `fix(custom_gds): …` or `feat: ihp-sg13cmos5l PDK support`, and a pull request that looks like the project's own work is easier for its maintainers to accept. Keep it small, and say in the description what problem it solves and how you tested it, as you would for a reviewer on your own team.

Upstream projects keep changing after you start from them. Tiny Tapeout updates its templates and actions for every shuttle, and your repository stays on the tag it started with until you change it, as Templates, forks and clones explains. When you report a problem, the tag and the commit it resolved to (both in your log) are what tell upstream which version you mean.

Working in public

Everything in a public repository is public: the code, the whole history, the commit addresses described in Git in one page, the issues, the pull request discussions, and the workflow runs. Run summaries can be read by anyone, and the full logs by anyone with a GitHub account. That has a few practical consequences:

- **What you put in `info.yaml` is published.** The `author` field, and the optional `discord` field ("Your discord username, for communication and automatically assigning you a Tapeout role (optional)", in the template's own words), go into the repository and into the datasheet the docs workflow builds. Put there what you're happy for anyone to read, and leave the optional field empty if you'd rather.
- **Nothing secret goes in a repository.** That includes passwords and access tokens, and also private data that has found its way into a test file. A file deleted in a later commit is still in the history.
- **Write for strangers.** Commit messages, issues and reviews stay readable long after the conversation is over, by people who weren't there. Write them so that they make sense to that reader, and so that you'd be comfortable with anyone reading them, including whoever you next apply to for a job.

Organization repositories also publish their GitHub Pages sites under the team's own domain, for example `https://uoftasic.com/simproc/` for `simproc`, because GitHub places project sites under an organization's custom domain when it has one. A repository in your personal account publishes under your account instead.

Check yourself

1. You copy a small UART module from another open-source project into `src/uart/` of your Tiny Tapeout repository. What should you add alongside it, and what should you check first?

Answer: A1, in Answers at the end.

2. Who can read what you put in the `discord` field of `info.yaml`?

Answer: A2, in Answers at the end.

3. Tiny Tapeout's `viewer` job does something you don't understand. Where do you look before asking anyone, and what do you include if you do ask?

Answer: A3, in Answers at the end.

4. Why does a reviewer care whether `UPSTREAM.md` names a commit and not just a branch?

Answer: A4, in Answers at the end.

Next: Lab 04, where you take a change of your own through review.

Answers

A1. Check yourself

Check that its licence is compatible with Apache-2.0, since Tiny Tapeout requires that. Then keep its licence file in `src/uart/`, and add a provenance note like `hart-1`'s `UPSTREAM.md`: the repository, the branch and exact commit you copied from, which files you took or left out, and any changes you made.

A2. Check yourself

Anyone. The field is in a public repository, and the docs workflow puts it in the datasheet. It's optional, so you can leave it empty.

A3. Check yourself

Your own log first, then the action's README and Tiny Tapeout's FAQ, then the recent commits and issues of `tt-gds-action`. If you still need to ask, include the exact first error line, a link to the run, and the action's tag and the commit hash your log says it resolved to.

A4. Check yourself

A branch moves as people commit to it, so "the `ttsky25a` branch" can mean different code next month. A commit hash names one exact version, so the reviewer can compare the copied files with it and confirm nothing was changed except what the file lists.