# IC102-D — Working in Tiny Tapeout Repos (Digital)

Make a digital Tiny Tapeout project from the template on your own GitHub account and keep its workflows passing.

**UofT ASIC Team**

# Contents


1 IC102-D — Working in Tiny Tapeout Repos (Digital) . . . . . 5

I 1. Your repository

2 Getting started . . . . . 11

3 Git in one page . . . . . 15

4 Templates, forks and clones . . . . . 19

5 Lab 01: Your repository from the template . . . . . 25

II 2. Workflows

6 What a cloud runner is . . . . . 37

7 Reading a workflow file . . . . . 41

8 The Tiny Tapeout workflows . . . . . 55

9 Lab 02: Read the red X . . . . . 59

III 3. Your design

10 RTL and gate-level tests . . . . . 67

11 Your tools and the runner's tools . . . . . 71

12 Lab 03: Make it yours . . . . . 75

IV 4. Review

13 Pull requests and review . . . . . 85

14 Branch rules . . . . . 91

15 How open-source projects work . . . . . 95

16 Lab 04: A change through review . . . . . 99

## V 5. History

17 History as evidence . . . . . 115

18 Lab 05: Find the commit that broke it . . . . . 121

# 1. IC102-D — Working in Tiny Tapeout Repos (Digital)

In this course you make a digital Tiny Tapeout project on your own GitHub account from Tiny Tapeout's Verilog template, and push to it until every workflow passes. In Lab 02 you break it in five ways and find the cause of each failure in the run's log. Later labs have you review a partner's pull request and use `git bisect` to find the commit that broke an LED dimmer's test.

IC102-D is part of the UofT ASIC Team education materials, published at https://uoftasic.com/ic102-d/. Its labs run in the workbench, the IIC-OSIC-TOOLS container you start from the `workspace` clone you set up in IC101, and on GitHub, in repositories of your own. You get the course's files with the workbench's `mod` command (`mod add ic102-d`); Lab 01, Part 1 walks you through it.

## Who this course is for

Take it after IC101 and before you push to any of the team's Tiny Tapeout projects, whether that's joining Sentry, HART-1 or the protocol emulator, or starting a project of your own. It assumes nothing beyond IC101: no Verilog and no layout. Each lab ships the design it needs: the template's adder, an 8-bit counter, a review sandbox that you and a partner each copy, and an LED dimmer's history. You practise git, GitHub's workflows, pull requests and review on them. Writing the designs themselves is what the DD courses teach.

There are two versions of IC102. Take this one, IC102-D, if your project is written in Verilog or SystemVerilog and Tiny Tapeout's `gds` workflow builds its layout. Take IC102-A if you draw your layout yourself and commit the GDS.

The course stops where Tiny Tapeout's submission would begin. You'll get a project to the state Tiny Tapeout asks for, with every check green, but you won't create a project on `app.tinytapeout.com`, pay for tiles or use a coupon.

| | |
|---|---|
| Prerequisites | IC101, and a GitHub account of your own |
| Tools | The workbench image `hpretl/iic-osic-tools:2026.08`, plus the versions Tiny Tapeout's runner uses, which Lab 01 installs next to it: Icarus Verilog 12.0 and 13.0, cocotb 2.1.0 and Python 3.11 |
| GitHub | Public repositories on your own account, with free runners and Pages, and a partner in the course for Lab 04. The course doesn't use the team's GitHub organization. |
| Total time | About 9 hours 5 minutes, self-paced, including waits for GitHub's runners but not for your Lab 04 partner. The Course path rows below add up to that. |

You may use AI tools such as Claude, ChatGPT or Copilot to help you in this course, as long as you check what they give you and say where you used them. Pull requests and review says what that means for a pull request, and Lab 04 has an optional part where you review one an AI tool wrote.

## Course path

Work down this table in order. A Read row is a set of guide pages, which are for reading only and have no commands in them. A Do row is one lab writeup, and the lab writeups are where every command in the course lives. Each lab's "Before you start" repeats which pages it expects you to have read.

| # | | What | Time |
|---|---|---|---|
| 1 | Read | Getting started · Git in one page · Templates, forks and clones | 40 min |
| 2 | Do | Lab 01: Your repository from the template. Make your own repository from Tiny Tapeout's Verilog template, fill in `info.yaml` and `docs/info.md`, turn on Pages when the viewer job fails, and push until every job is green. | 1 h |
| 3 | Read | What a cloud runner is · Reading a workflow file · The Tiny Tapeout workflows | 45 min |
| 4 | Do | Lab 02: Read the red X. Break your repository in five ways the team's repositories broke, and for each one write down the job, the step and the first error line before you fix it. | 1 h 15 min |
| 5 | Read | RTL and gate-level tests · Your tools and the runner's tools | 20 min |
| 6 | Do | Lab 03: Make it yours. Replace the template's adder with an 8-bit counter, write a test that catches a planted bug in the workbench and on the runner, and find where the `gds` job's numbers come from. | 1 h 30 min |
| 7 | Read | Pull requests and review · Branch rules · How open-source projects work | 35 min |
| 8 | Do | Lab 04: A change through review. Make a repository of your own with branch rules, claim a pin in your partner's through a pull request from a fork, review theirs and ask for a change, and | 1 h 45 min |

| # | | What | Time |
|---|---|---|---|
| | | resolve the conflict a neighbouring claim leaves. | |
| 9 | Read | History as evidence | 15 min |
| 10 | Do | Lab 05: Find the commit that broke it. Write a script that `git bisect run` can trust, find the commit that broke a project's 36-commit history, and revert it in a pull request that passes every job. | 1 h |

## Getting help

Ask in the team Discord; nobody expects you to work all of this out alone. Getting started says what to put in a question so it can be answered quickly.

# I

# 1. Your repository


2 **Getting started . . . . . 11**
2 How the course is laid out . . . . . 11
2 What you need . . . . . 11
2 Where you run git . . . . . 11
2 Public repositories . . . . . 12
2 Your tools and the runner's . . . . . 12
2 AI tools . . . . . 12
2 What the course doesn't cover . . . . . 12
2 Getting help . . . . . 12
2 Check yourself . . . . . 13
2 Answers . . . . . 13

3 **Git in one page . . . . . 15**
3 Four places a change lives . . . . . 15
3 Commit identity . . . . . 15
3 Commit messages . . . . . 16
3 Files that stay out of git . . . . . 16
3 Branches, merges and conflicts . . . . . 17
3 Undoing a commit . . . . . 18
3 Check yourself . . . . . 18
3 Answers . . . . . 18

4 **Templates, forks and clones . . . . . 19**
4 Three ways to copy a repository . . . . . 19
4 What the template gives you . . . . . 21
4 The placeholders you have to replace . . . . . 21
4 Template updates . . . . . 22
4 Check yourself . . . . . 22
4 Answers . . . . . 22

5 **Lab 01: Your repository from the template . . . . . 25**
5 Before you start . . . . . 25
5 Part 1: Load the course . . . . . 25
5 Part 2: Your identity and your token . . . . . 26
5 Part 3: Make the repository . . . . . 29
5 Part 4: Fill it in . . . . . 30
5 Part 5: Turn on Pages . . . . . 31
5 Finishing . . . . . 34
5 Answers . . . . . 34

# 2. Getting started

IC102-D is about everything around a digital Tiny Tapeout design except the design itself. You start from Tiny Tapeout's Verilog template, as every project on the team does, and by the end of the course you'll have a repository of your own with every workflow green, a pull request that a classmate reviewed and approved, and a revert you found with `git bisect` and got through every check. The Verilog comes with each lab, so you don't need to know any. This page says how the course is laid out and what you need before Lab 01, and it explains why you'll type every git command in the workbench rather than on your laptop.

## How the course is laid out

The Course path on the home page alternates Read rows and Do rows. A Read row is one or more guide pages like this one. They explain, show real output from the team's repositories and from Tiny Tapeout's, and never ask you to run anything; a "Check yourself" block at the end of a page asks a few questions you can answer on paper. A Do row is a lab writeup, and every command in the course is in one of those. Each lab is split into parts, each part opens with the question it answers and ends with what you should see, and each lab ends with a "You're done when" line that you check against your own repository.

The files a lab runs (scripts, a design, a test, a checker) are in the course repository under `labs/`, one folder per lab, and the test job's tools are in `labs/common/tt-tools`. You get all of it with `mod add ic102-d` in Lab 01, Part 1.

Five labs take about eight hours, and a good part of that is waiting: a `gds` run takes around five minutes, and a lab may need three or four of them. Each lab says when you can read ahead while a run is going.

## What you need

- **The workbench from IC101.** You'll run the course's tools in it, and every git command too, for reasons given in the next section. If you haven't set it up, do IC101 Lab 03 first.
- **A GitHub account of your own.** Every repository you make in this course goes on it, as a public repository, for three reasons: GitHub's standard runners are free and unlimited on public repositories, Pages (which the `viewer` job needs) is available on a free account only for public ones, and so are branch rules. You don't need a paid plan, and the course doesn't use the team's GitHub organization at all: Lab 04′s review happens between your account and a partner's.
- **A token for git**, which Lab 01, Part 2 has you make. GitHub doesn't accept your password from git.

## Where you run git

You'll type every git command in a terminal on the workbench desktop, in a clone under `/foss/designs/projects/`, named after its repository: `ttsky-ic102-d` for your project, and two more in Labs 04 and 05. `/foss/designs` is the `workspace` folder on your laptop, so your clones survive when the container stops, and you can open the files in an editor on your laptop if you prefer one. (On Windows, set that editor to save shell scripts with Unix line endings, because bash can't run a script whose lines end in the Windows way.) The course itself sits next to them, in `/foss/designs/modules/ic102-d`, which is where `mod` keeps courses. The alternative, cloning with git on the laptop and running the tests in the workbench, runs into two problems:

- On a Linux laptop whose user ID isn't 1000, the container's user doesn't own a clone your laptop made, and git in the container refuses to touch it with `fatal: detected dubious ownership in repository`.
- Lab 05′s `git bisect run` checks out one old commit after another and runs the test at each, so git and the test have to run on the same machine.

Working in a container changes two things about how you use git. First, anything outside `/foss/designs` is deleted when the container is replaced, and that includes the file where `git config --global` saves your settings. So the labs have you set your name, email address and credential cache with plain `git`

`config` inside each clone, which saves them in that clone's `.git/config`, on your own disk. Second, git asks for your token on the first push after each restart, because the cache that remembers it for 8 hours lives in the container's memory.

The token is a fine-grained personal access token for your own account, with access to all of your repositories and two permissions, both "Read and write":

- **Contents**, which is what pushing commits needs.
- **Workflows**, which lets a token add or change files under `.github/workflows/`. You won't edit a workflow in this course, but two of your pushes carry workflow files: the first push of your Lab 04 review repository, and the first push of Lab 05's history.

GitHub's documentation gives the same two permissions, for the same reasons, to a GitHub App that uses git (choosing permissions for Git access).

It needs all of your repositories, rather than a list, because the ones for Labs 04 and 05 don't exist yet when you make it in Lab 01. Every push in the course goes to a repository you own, including Lab 04's: your pull request to your partner comes from your own fork of their repository. GitHub also offers older "classic" tokens, whose `public_repo` scope covers every public repository your account can push to, including those of any organization you belong to. A fine-grained token reaches only your own repositories, which is all this course needs, so a token that leaks can do less harm.

## Public repositories

Everything you push to a public repository can be read by anyone: your code, your commit messages, the name and email address on every commit, and the `author` field in `info.yaml`, which also goes into the datasheet. Lab 01 has you set your commit address to the no-reply address GitHub gives every account, before your first commit, because a commit can't be changed later without changing its hash and the hash of every commit after it. How open-source projects work says more.

## Your tools and the runner's

The workbench has its own versions of the simulation tools, and GitHub's runner installs the versions Tiny Tapeout's workflows ask for. They differ: the runner's `test` job uses Icarus Verilog 12.0 and cocotb 2.1.0, and the workbench has 14.0 and 2.0.1. Lab 01 installs the runner's versions next to the workbench's, and `make test` runs your test with them, so a test that passes there passes on the runner. Your tools and the runner's tools explains what that covers and what it doesn't.

## AI tools

You may use AI tools such as Claude, ChatGPT or Copilot in this course: to explain an error, to draft a test, or to help with a pull request description. You answer for everything you push, however it was written, so check what a tool gives you, and say where you used one. A pull request description is the place for that, and Pull requests and review explains why it matters to your reviewer. Lab 04 has an optional part where you review a pull request an AI tool wrote.

## What the course doesn't cover

Tiny Tapeout puts designs on real chips, a few times a year, in shuttles with a deadline and a price per tile. This course stops just before that. You'll get a project to the state Tiny Tapeout asks for at submission, with every check green, but you won't create a project on Tiny Tapeout's site, pay for tiles or use a coupon. Hardening a design on your own machine, the FPGA workflow (which the template switches off) and writing workflows of your own are also left out.

## Getting help

Ask in the team Discord. A question about a red cross gets answered fastest when it includes the link to the run, the job and step that failed, and the first error line copied as text, not as a picture, followed by what you've already tried. Logs on GitHub are deleted after 90 days at most, so text pasted into a message also lasts longer than a link to the log.

## Check yourself

1. Why does the course have you run `git config user.email ...` in every clone, instead of once with `--global`?

*Answer: A1, in Answers at the end.*

2. Your project repository is private on a free account. Which parts of this course stop working?

*Answer: A2, in Answers at the end.*

3. A classmate made a classic token with the `public_repo` scope instead, and it works for every lab. What else can it push to, and why does that matter?

*Answer: A3, in Answers at the end.*

## Answers

**A1. Check yourself**

Because `--global` saves the setting in your home folder inside the container, and the container is replaced, with everything outside `/foss/designs`, each time you start it. Without `--global` the setting goes in the clone's own `.git/config`, which is on your disk.

**A2. Check yourself**

The runners are no longer free and unlimited (a private repository draws on a monthly allowance), Pages isn't available, so the `viewer` job can't pass, and branch rules aren't available either. That's why the course uses public repositories.

**A3. Check yourself**

It can push to every public repository the classmate's account can write to, not just their own: if they belong to an organization whose members can write to its repositories, all of those too. If the token leaked, whoever found it could push to all of them. A fine-grained token made for their own account reaches only their own repositories, which is all the course needs.

# 3. Git in one page

This page answers: *what does git keep track of, and which few ideas do you need to work in a Tiny Tapeout repository alongside other people?*

In IC101 you used git for exactly one thing, `git clone`. From Lab 01 of this course onwards you'll make commits, push them to GitHub, open pull requests and read other people's history, so you need a working picture of what git is doing. This page gives you that picture. The commands themselves are in the labs, where you'll type them, so here we only explain what they do and why the team's repositories look the way they do.

## Four places a change lives

A change you make to a file passes through four places before anyone else can see it:

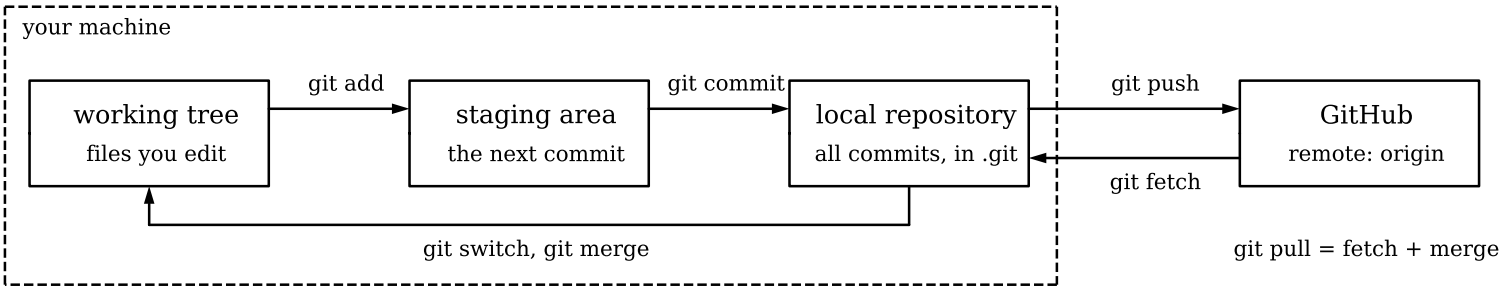



Figure 3.1: The four places a change lives, and the commands that move it between them.

The **working tree** is the folder of files you open in your editor. Git watches it but doesn't record anything until you ask. `git add` copies a file's current contents into the **staging area**, which is git's draft of the next commit, and `git commit` turns that draft into a permanent record in the **local repository**, the hidden `.git` folder at the top of your clone. None of this has left your laptop yet. `git push` sends your new commits to the **remote**, the copy on GitHub, which git calls `origin` because it's where you cloned from.

A **commit** is a snapshot of every tracked file, plus a message, an author, a date and a pointer to the commit before it (its parent). Git names each commit by a hash of all that, which is why the team's history is full of names like `b976406`: that's the first seven characters of a 40-character hash, and seven are almost always enough to pick out one commit in a repository.

A **branch** is just a name that points at a commit, and it moves forward each time you commit on it. `main` is the branch everyone shares; you'll do your work on a branch of your own and bring it into `main` with a pull request, which Pull requests and review explains. Going the other way, `git fetch` brings down commits other people pushed, and `git merge` or `git switch` brings them into your working tree. `git pull` does the fetch and the merge in one step.

Of these steps, the push matters most in a Tiny Tapeout repository, because GitHub runs the template's workflows on every push, so a commit isn't checked until you've pushed it.

## Commit identity

Every commit records an author name and an email address, taken from your git settings (`user.name` and `user.email`) at the moment you commit. Both are part of the commit itself, so they're public in a public repository, and anyone who clones it can list every address in the history with one command.

In four of the team's repositories (`sentry`, `simproc`, `crosswalk` and `asic-zoo-template`) the history shows the same person committing under two addresses: a personal one and a GitHub no-reply one. It's an easy thing to do, since every machine you commit from has its own git settings, and nobody did

anything wrong. The personal addresses are in the history for good, though, because changing an old commit changes its hash and the hash of every commit after it.

GitHub gives every account a no-reply address of the form `ID+USERNAME@users.noreply.github.com` (GitHub's email reference), and setting `user.email` to it before your first commit keeps your personal address out of every repository you work in. That's the first thing you'll do in Lab 01.




Figure 3.2: GitHub's email settings, where "Keep my email addresses private" shows your no-reply address. The ID and username in it are blurred here.

## Commit messages

A good commit does one thing, and its message says what that thing is and, when it isn't obvious, why. That matters because the history is the only record most projects keep: six weeks from now nobody will remember why a line changed, and the message will be the only place to find out.

Here are three real commits from `protocol-emulator`, all made while getting its workflows to pass:

| Commit | Message | What it changed |
|---|---|---|
| `b976406` | `docs: update info.yaml` | `info.yaml`, and the module name in `src/project.v` |
| `a5f1475` | `chore: temporarily remove assertion from test.py` | Commented out the template's only assertion |
| `7c56c54` | `chore: updated module name in testbench` | `test/tb.v`, to match the new module name |

These are ordinary messages, and most of us have written ones like them, but two of them caused problems later. The first commit renamed the design's top module, but its message mentions only `info.yaml`, so when the test workflow started failing at that commit, the message gave no hint why. The second one says what it did and not why, and "temporarily" is still true on `main` two weeks later: the assertion is commented out, so the test that passes there checks nothing. History as evidence follows this story further.

Compare `b1896da` in `ttsky-verilog-tinyCGRA`, "removed config_done assert (unreliable in GLS)". It also removes an assertion, but it says which one and gives a reason a reviewer can argue with. That's the standard to aim for. If a message needs more than a line, write a short summary line, a blank line, and then the explanation underneath; git and GitHub both show the first line on its own in lists.

## Files that stay out of git

A repository should hold the files you wrote and nothing that a tool can regenerate from them. Generated files change on every run, they're often large, and they make every diff noisy, so a reviewer can't see your real change among them. The file `.gitignore` lists the paths git should pretend aren't there. The digital template ships this one:

```
.DS_Store
.idea
*.vcd
*.fst
*.fst.hier
runs
tt_submission
src/user_config.json
src/config_merged.json
```

```
test/sim_build
test/__pycache__/
test/results.xml
test/gate_level_netlist.v
```

Every line is something a tool writes: waveforms (`*.vcd`, `*.fst`), the LibreLane run directory (`runs`), the simulator's build folder and results, and the gate-level netlist the GDS action copies into `test/`. When a project adds a new place for tools to write, it needs a new line here. `crosswalk` did that in pull request #5, "Ignore simulation output", which added `test/digital/sim_build/`, `test/digital/results.xml`, `test/digital/tb.fst`, `test/digital/__pycache__/` and `*.raw` after the repository was split into analog and digital parts.

Once a file has been committed, ignoring it later doesn't remove it from the history. `sentry` keeps two trace files in its top folder, `ata_read.csv` (1.6 MB) and `ata_write.csv` (393 KB). They were added in a commit called "first test case", reverted the same day, and added again the next day. Whether those traces belong in the repository is the project's call, and there can be good reasons to keep test data with the code. Either way the decision is permanent, because even after a revert every clone of the repository downloads them.

Two exceptions matter in this course. In IC102-A you commit GDS and LEF files on purpose, because the analog workflow reads your layout from the repository instead of building it. And nothing secret ever goes in a repository, ignored or not, because a single mistaken `git add` publishes it.

## Branches, merges and conflicts

When you work on a branch, git can later bring the two lines of work back together with a merge. Most merges need no help, because the two branches changed different files or different lines. When both branches changed the same lines, git stops and asks you to decide, and that's called a **merge conflict**.

In a Tiny Tapeout repository the most likely place for one is the pinout in `info.yaml`, because everyone who adds a block also claims pins there. Here is a real run, on a copy of the template's `info.yaml`, where one branch named `uart` gives `uo[0]` to a UART transmitter and another named `pwm` gives the same pin to a PWM output. Both are merged into `main`:

```
$ git merge uart
Updating bc18097..e3031f2
Fast-forward
 info.yaml | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git merge pwm
Auto-merging info.yaml
CONFLICT (content): Merge conflict in info.yaml
Automatic merge failed; fix conflicts and then commit the result.
```

The first merge is a **fast-forward**: `main` had no commits of its own since `uart` branched off, so git simply moved the `main` label forward. The second merge can't do that, and git writes both versions into the file, between markers:

```
  # Outputs
<<<<<<< HEAD
  uo[0]: "UART TX"
=======
  uo[0]: "PWM out"
>>>>>>> pwm
  uo[1]: ""
```

Between `<<<<<<< HEAD` and `=======` is what's on the branch you're merging into, and between `=======` and `>>>>>>> pwm` is what the other branch wants. Git can't know which is right, and here neither is: two signals can't share a pin, so one of them has to move to `uo[1]`, and that's a design decision the two authors have to make together. Resolving a conflict means editing the file to what it should say, deleting the three marker lines, and committing the result. You'll resolve a pinout conflict like this one in Lab 04.

## Undoing a commit

There are two ways to undo a commit, and which one is safe depends on whether anyone else has it.

`git revert` makes a new commit that does the opposite of an old one. The history keeps both, so anyone who already has the old commit just gets one more on top. `sentry` has one on `main`, `d12a672`, whose message git wrote automatically:

```
Revert "first test case"

This reverts commit f0804819d4a77338811fe40a080d9eae9d4e6473.
```

`git reset` moves your branch label back to an earlier commit, as if the later ones had never happened. That's fine for commits that exist only on your laptop. Once they've been pushed, though, other people may have built on them, and putting the reset on GitHub needs a **force push**, which replaces the branch there with yours. Anyone whose work was based on the commits you removed is now out of step, and the next time they pull and push, the commits you removed can come straight back.

So the rule is to revert on any branch someone else might have, and to reset only what you haven't pushed. A repository can enforce the first half of this with a branch rule that blocks force pushes. `crosswalk` and `protocol-emulator` have one on `main`, and `hart-1` has one that, as Branch rules shows, doesn't apply to any branch at all.

## Check yourself

1. You made three commits with your personal email address and pushed them to a public repository. You then set `user.email` to your no-reply address. Which of your commits show the personal address now, and why can't you simply edit them?

*Answer: A1, in Answers at the end.*

2. A teammate's commit on `main` broke the test workflow, and three other people have pulled since. Should you fix it with `git revert` or with `git reset` and a force push?

*Answer: A2, in Answers at the end.*

3. Which of these belong in a digital Tiny Tapeout repository: `src/project.v`, `test/results.xml`, `runs/`, `info.yaml`, `test/tb.fst`?

*Answer: A3, in Answers at the end.*

4. In the conflict above, what does the text between `=======` and `>>>>>>> pwm` hold?

*Answer: A4, in Answers at the end.*

Next: Templates, forks and clones.

## Answers

**A1. Check yourself**

All three still do. The setting applies only to commits you make after changing it. The address is part of each old commit, and changing a commit changes its hash and the hash of every commit after it, which on a shared branch means a force push and trouble for everyone who has the old commits. That's why you set the address before your first commit.

**A2. Check yourself**

`git revert`. It adds a new commit that undoes the broken one, so everyone else's copy stays consistent and the history shows both what went wrong and how it was undone. A reset and force push would remove a commit that three people already have.

**A3. Check yourself**

`src/project.v` and `info.yaml`, because you wrote them and no tool can recreate them. The other three are written by tools on every run (the test results, LibreLane's run folder and a waveform), and the template's `.gitignore` already lists all three.

**A4. Check yourself**

The `pwm` branch's version of the line, `uo[0]: "PWM out"`. The part above `=======` is the version already on the branch you're merging into (`HEAD`, which here is `main` with `uart` merged in).

# 4. Templates, forks and clones

This page answers: *what do you actually get when you make a repository from Tiny Tapeout's template, how is that different from a fork or a clone, and what is still missing afterwards?*

Every Tiny Tapeout project starts the same way. Someone presses **Use this template** on one of Tiny Tapeout's template repositories, gets a repository of their own, and starts filling it in. It looks like a copy of the template, but it isn't quite one, and the differences explain several of the red crosses you'll see in the team's repositories. You'll make your own repository this way in Lab 01.

## Three ways to copy a repository

GitHub and git give you three different kinds of copy, and each takes a different amount from the original:

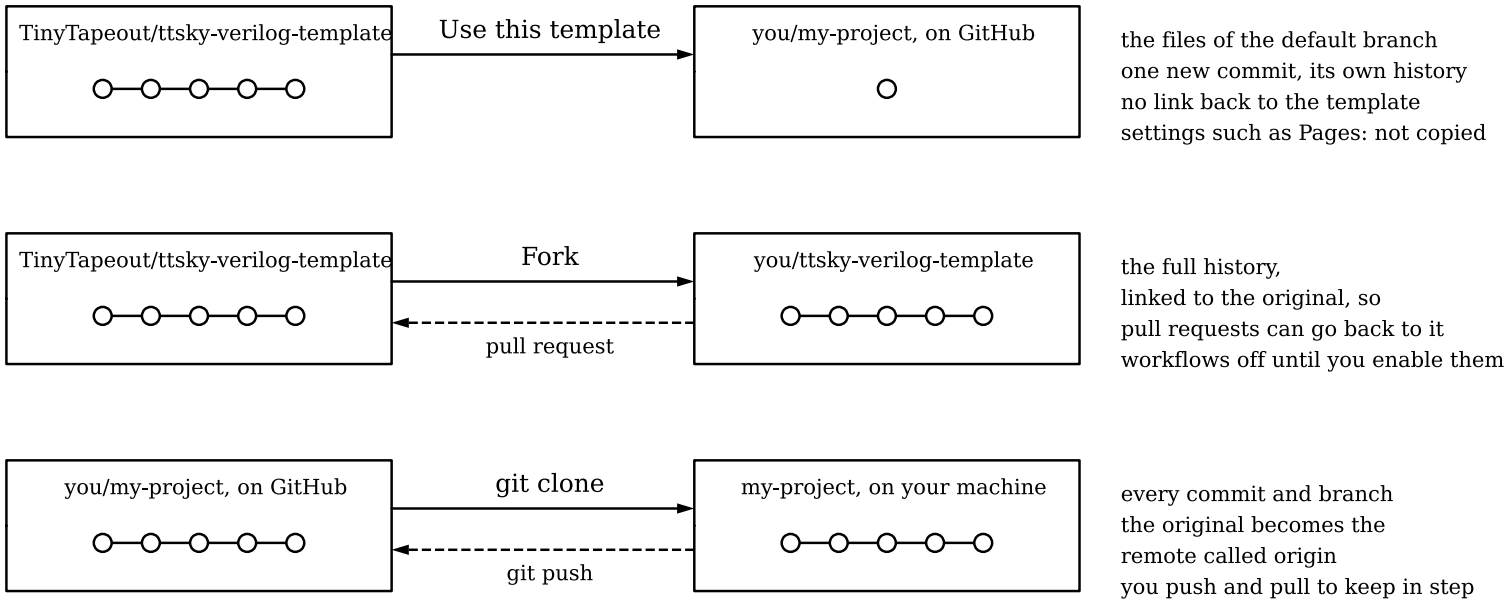



Figure 4.1: What a template, a fork and a clone each take from the repository they start from.

**Use this template** makes a new repository that starts with the template's files and nothing else. GitHub's documentation puts it this way: the new repository gets "the same directory structure, branches, and files", by default only the default branch, and "a repository created from a template starts with a single commit", with a history unrelated to the template's (creating a repository from a template). Your project's history starts at that commit, and nothing in it links back to the template's history.

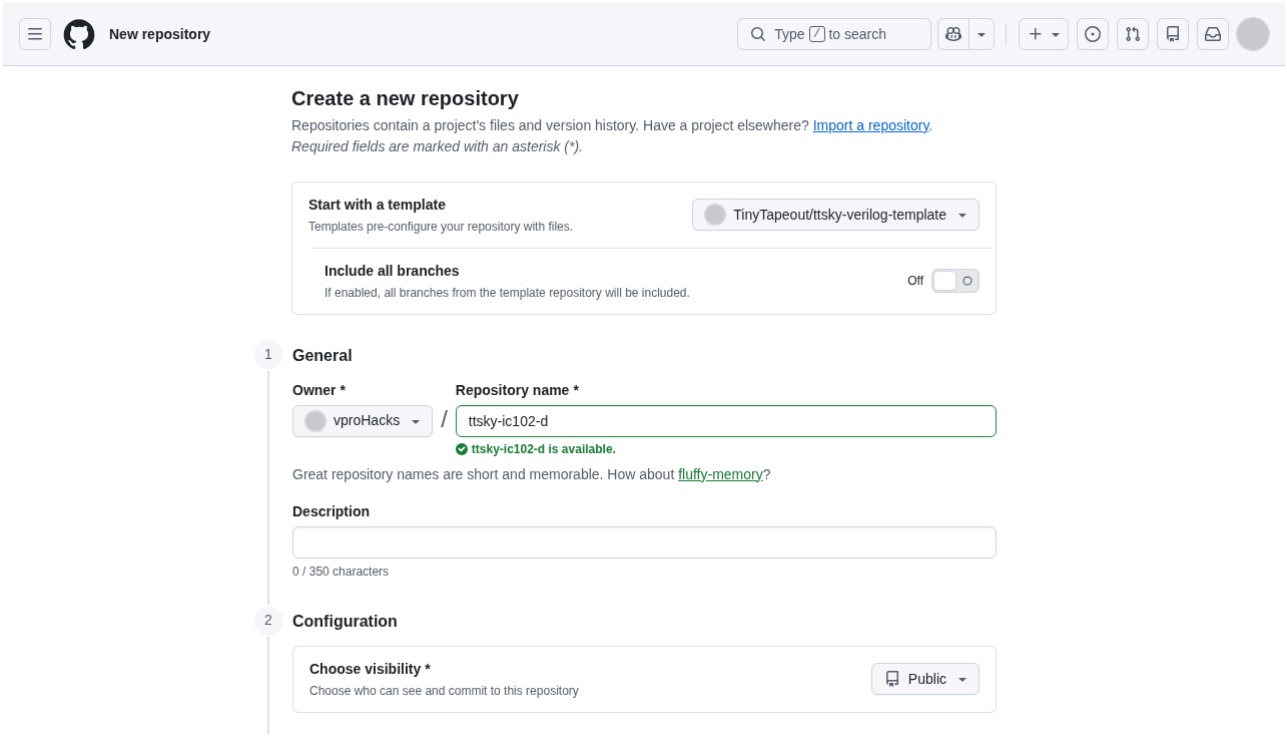



Figure 4.2: Creating a repository from the template. "Include all branches" is off by default, and the course uses a public repository.

The template itself has a much longer past. This is Tiny Tapeout's digital template on Nov 15, 2026, seen without signing in:

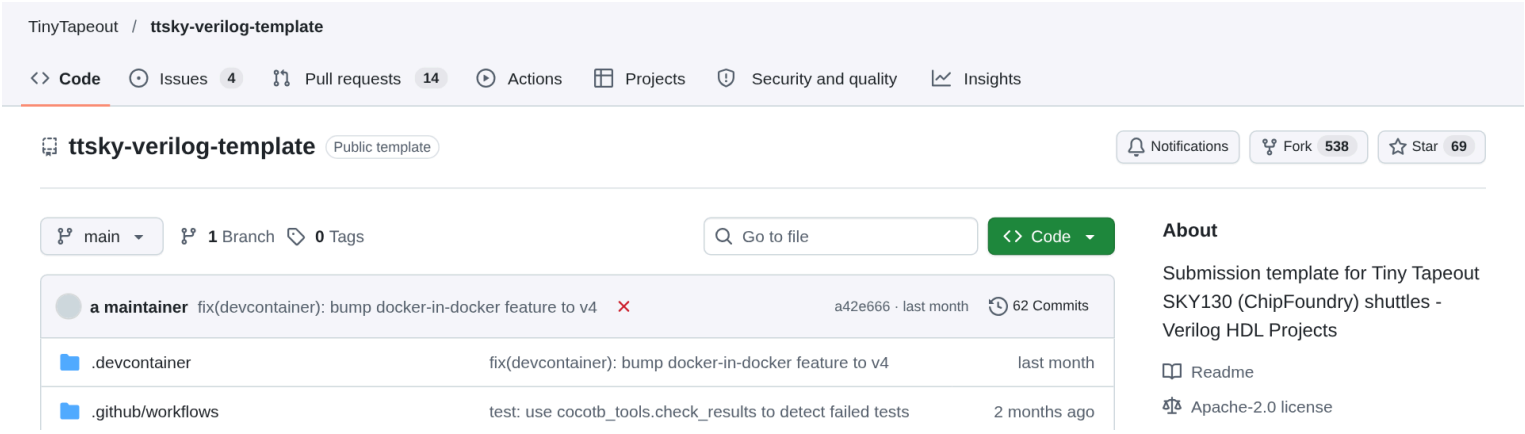



Figure 4.3: Tiny Tapeout's digital template on Nov 15, 2026. The maintainer's name is replaced with a role.

None of its 62 commits come with it when you use it; your repository starts with one commit of its own.

A **fork** is a copy that stays linked to the original. It keeps the history, and because GitHub remembers the link, you can open a pull request from your fork back to the original repository. That's how you contribute to a project you can't push to, such as Tiny Tapeout's own template, or, in Lab 04, your partner's repository. A fork doesn't run workflows straight away, though. GitHub's documentation says that "Workflows don't run in forked repositories by default" (events that trigger workflows), so a fork of a Tiny Tapeout repository won't check anything until you turn Actions on in it.

A **clone** is a copy on your own machine, made with `git clone`. It has every commit and branch, and the repository you cloned from becomes the remote called `origin`, which is where your pushes go. You'll clone every repository you work on, whichever way it was created.

For your own project, you use the template. You fork when you want to propose a change to somebody else's repository, and you clone whenever you want to work on a repository at all.

## What the template gives you

The digital template, `TinyTapeout/ttsky-verilog-template`, gives you these files:

| Path | What it's for |
|---|---|
| `info.yaml` | Your project's title, author, top module, tile size, clock and pinout |
| `docs/info.md` | The text of your page in the datasheet |
| `src/project.v`, `src/config.json` | An example design (an adder) and the LibreLane settings |
| `test/` | A cocotb testbench (`tb.v`, `test.py`), its `Makefile` and `requirements.txt` |
| `.github/workflows/` | Four workflows: `docs`, `test`, `gds` and `fpga` (`fpga` is switched off with `branches: none`) |
| `.devcontainer/` | A container definition, with the template's tools, for VS Code's dev containers |
| `LICENSE` | Apache-2.0 |
| `README.md` | Badges for the workflows, and links to Tiny Tapeout's guides |

The analog template, `TinyTapeout/ttsky-analog-template`, is smaller. It has only two workflows, `docs` and `gds`, and its `src/project.v` is an empty module that declares the power and signal pins your layout must provide. There's no example design, because the design is the layout you draw, and the `gds` workflow expects to find it already in the repository. If you're joining the team's ASIC Zoo, you'll start from `asic-zoo-template`, which was itself made from Tiny Tapeout's analog template and is marked as a template in its own right.

The template doesn't give you its settings, though. Every `gds` workflow ends with a job called `viewer`, which publishes your layout through GitHub Pages so that Tiny Tapeout's 3D viewer can show it, and Pages is switched off in a new repository. When we checked on Nov 15, 2026, Pages was off in `protocol-emulator`, `hart-1`, `crosswalk` and `asic-zoo-template`, all of them made from a template, and the viewer job had failed in every `gds` run they'd made. (Pages has since been turned on in all four.) GitHub's documentation doesn't say which settings a template passes on, but these four repositories show that Pages isn't one of them. Reading a workflow file shows what the failure looks like and how it's fixed.

## The placeholders you have to replace

The template is meant to be filled in, and its checks fail until you do. The two files that matter most are `info.yaml` and `docs/info.md`.

`info.yaml` arrives with its title, author and description empty, its pinout blank and its top module set to `tt_um_example`. Tiny Tapeout's validator, `project_info.py` in `tt-support-tools`, reads it on every push and rejects, among other things, an empty title, author, description or language, a top module whose name doesn't start with `tt_um_`, a missing or non-integer `clock_hz`, `analog_pins` outside 0 to 6, `uses_vapwr` set without an analog pin, an empty pinout, and any pin added to or deleted from the pinout. The comments in the template's own `info.yaml` say the same things in plainer words ("DO NOT delete or add any pins"), and they're worth reading line by line.

`docs/info.md` arrives with two sections, "How it works" and "How to test", each holding a sentence of placeholder text. The docs check (`project_checks.py`) doesn't look for the headings. It looks for the placeholder sentences themselves, `Explain how your project works` and `Explain how to use your project`, under those headings, and reports the section as missing if it finds them.

`hart-1` shows both checks at work. Its `info.yaml` still has the template's empty fields, and its `docs/info.md` is identical to the template's. Here is the end of the docs job from its run on Nov 15, 2026 (run 43752267314):

```
## Errors in info.yaml
- Project title cannot be empty
- Project author cannot be empty
- Project description cannot be empty
- Please fill in the 'pinout' section

## Errors in docs/info.md
- Missing 'How it works' section in docs/info.md
- Missing 'How to test' section in docs/info.md
```

Both sections exist in that file; they're reported as missing because they still hold the template's sentence. The `docs` and `gds` workflows in `hart-1` have failed on every push since its first commit on Oct 18, 2026, while `test` has passed every time, because its test is still the template's own test of the template's adder.

Everything you put in `info.yaml` ends up in public: in the repository, and in the datasheet that the docs workflow builds. The `author` field and the optional `discord` field are yours to choose, and How open-source projects work says more about what that means.

## Template updates

Tiny Tapeout updates its templates for every shuttle. The digital template's history has a commit called "chore: update tags for …" for each one: TTSKY25a (Jun 27, 2025), TTSKY25b (Sep 18, 2025), TTSKY26a (Feb 27, 2026), TTSKY26c (May 26, 2026) and TTSKY26d (Sep 15, 2026). Each of those commits changes the tag at the end of every `uses: TinyTapeout/tt-gds-action/…@…` line, which picks the version of Tiny Tapeout's actions your workflows run.

Your repository doesn't get those updates, because its history is unrelated to the template's and git has no common commit to merge from. Each of the team's repositories is therefore frozen at the tag it was made with: `ttsky25b` in `simproc` and `dpll`, `ttsky26a` in `ttsky-verilog-tinyCGRA`, `ttsky26d` in `hart-1`, `sentry`, `crosswalk` and `asic-zoo-template`, and `ttihp26b` in `protocol-emulator`. That's usually what you want, since a project that worked last week keeps working. When a project does need a newer template's workflows, Tiny Tapeout's FAQ says to copy the new template's files over the old ones, and that they "should replace anything in `.github/workflows`, `.devcontainer` and `test/Makefile`". Doing that is outside this course, but it helps to know why it's a copy and not a merge.

## Check yourself

1. You make a repository from the digital template and clone it. How many commits does `git log` show, and how many does the template have?

*Answer: A1, in Answers at the end.*

2. Your docs workflow reports `Missing 'How to test' section in docs/info.md`, but you can see a `## How to test` heading in the file. What's wrong?

*Answer: A2, in Answers at the end.*

3. You've found a mistake in a comment in Tiny Tapeout's template and want to propose a fix. Do you use the template, fork it, or clone it?

*Answer: A3, in Answers at the end.*

4. Why does the `viewer` job fail on the first push to a repository made from the template, even when nothing else is wrong?

*Answer: A4, in Answers at the end.*

Next: Lab 01, where you make your own repository from the template.

## Answers

**A1. Check yourself**

One commit, the "Initial commit" GitHub made when it created your repository. The template had 62 when we looked on Nov 15, 2026, and none of them come with it, because a repository made from a template starts with a single commit and a history of its own.

**A2. Check yourself**

The placeholder sentence `Explain how to use your project` is still under the heading. The check looks for that sentence, not for the heading, so replace it with a real description of how to test your design.

**A3. Check yourself**

Fork it (and then clone your fork to edit it). A fork keeps the link to the original repository, which is what lets you open a pull request back to it. A repository made with “Use this template” has no such link.

**A4. Check yourself**

It deploys to GitHub Pages, and Pages is a repository setting. The template passes on its files but not its settings, so Pages is off in the new repository until you turn it on, and the job passes only in a run started after that.

# 5. Lab 01: Your repository from the template

You'll make a Tiny Tapeout project of your own: a public repository on your GitHub account, created from Tiny Tapeout's Verilog template. You'll fill in the two files the template leaves for you, push, read what each workflow says, turn on GitHub Pages when the last job, `viewer`, asks for it, and keep going until every job is green on a commit you made. You'll also install the test job's tools in the workbench; the later labs use them too.

Package: `labs/lab-01-your-repo-from-the-template/`

## Before you start

- Read Getting started, Git in one page and Templates, forks and clones.
- You need a GitHub account of your own. Any account works; the course doesn't use the team's GitHub organization.
- You need the workbench from IC101. Every command in this course runs in a terminal on its desktop, git included; Getting started explains why.

This lab takes about an hour, of which about 15 minutes is waiting for GitHub's runners.

## Part 1: Load the course

If you haven't set up the workbench yet, do IC101 Lab 03 first.

1. From your `workspace` clone, start the desktop (`./scripts/start_vnc.sh`, or `scripts\start_vnc.bat` on Windows), open http://localhost/, and open a terminal on the desktop.
2. Load the environment and get the course files:

```
. /foss/designs/common/.designinit   # in every new terminal
mod add ic102-d                      # first time only: clones github.com/uoftasic/
ic102-d
mod ic102-d                           # every time, including the first
```

   You should see the `ASIC-EDU: PDK=sky130A ...` banner, then this from `mod add`:

```
Cloning https://github.com/uoftasic/ic102-d.git
     into /foss/designs/modules/ic102-d
Cloning into '/foss/designs/modules/ic102-d'...
OK  module ready: /foss/designs/modules/ic102-d
    run: mod ic102-d
cwd: /foss/designs/modules/ic102-d
```

   `mod add` leaves you in the course folder, and `mod ic102-d` prints the same `cwd:` line again, at the end.
3. Install the test job's tools. GitHub's runner tests your design with Icarus Verilog 12.0 and cocotb 2.1.0, and the workbench has 14.0 and 2.0.1, so the course installs the runner's versions next to the workbench's, in the course folder:

```
cd labs/common/tt-tools
make install
make check-tools
```

`make install` took 4.7 s on a fast connection. It puts 178 MB on your own disk, in the course folder, so the tools are still there after the container restarts. You should see this table:

```
Tool (job)            Runner        Here          Workbench      Status
----------            ------        ----          ---------      ------
Python (test)         3.11.18       3.11.16       3.12.3         ok (3.11)
cocotb (test)         2.1.0         2.1.0         2.0.1          ok
pytest (test)         8.4.2         8.4.2         9.1.1          ok
Icarus (test)         12.0          12.0          14.0           ok
Icarus (gl_test)      13.0          13.0          14.0           ok
sky130A (gl_test)        8afc8346a57f    -                  026824c7969c    not
installed (optional)
LibreLane (gds)       3.0.14        -             3.1.0.dev3     not installed:
the gds job runs on GitHub

All pinned tools are installed at the runner's versions.
```

4. Run the template's own test the way the runner's test job does:

```
make selftest
```

You should see, after about 3 s, `** test.test_project PASS 100000.00` and `** TESTS=1 PASS=1 FAIL=0 SKIP=0`. Your tools and the runner's tools explains the versions, and the package README has the details.

| If you see | Do this |
|---|---|
| `make install` stops partway with a download error | Run `make install` again; it skips the parts that finished. |
| `fatal: detected dubious ownership in repository` | The clone was made on your laptop, and the container's user doesn't own it. Clone again from the workbench terminal. |
| Anything else | IC101's Troubleshooting |

## Part 2: Your identity and your token

What will your commits say about you, and how will git prove to GitHub that it's you pushing?

1. Find your GitHub no-reply address. On GitHub, open **Settings › Emails**, turn on **Keep my email addresses private**, and copy the address in the sentence under it, which looks like `ID+USERNAME@users.noreply.github.com`.

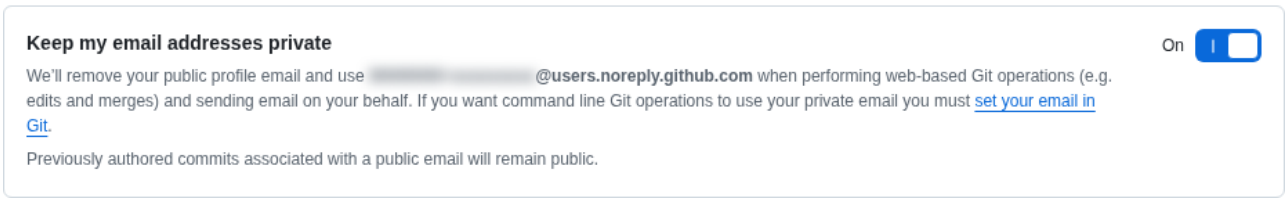



Figure 5.1: GitHub's email settings, where "Keep my email addresses private" shows your no-reply address.

Every commit records an email address, and in a public repository anyone can read it. Four of the team's repositories already show personal addresses in their history. Committing with the no-reply address keeps your personal address out of your own repositories' history. You'll set it in each repository you clone, starting in Part 3.

2. Make a token for git to push with. GitHub doesn't accept your password from git, so on GitHub open **Settings › Developer settings › Personal access tokens › Fine-grained tokens**, choose **Generate new token**, and give it:
   - the name `IC102 workbench`, and your own account as the **Resource owner**, which the form has already chosen.
   - an **Expiration** of 60 days. The menu starts at 30 days, so choose "60 days" in it.
   - **All repositories** under **Repository access**, because the repositories for Labs 04 and 05 don't exist yet. The form starts on "Public repositories", so you have to change it.
   - two repository permissions, **Contents** and **Workflows**, both "Read and write". Contents is the permission every push needs. You need Workflows as well, because it's the permission that lets a token add or change files under `.github/workflows/`, and two of your pushes, in Labs 04 and 05, do that. GitHub's documentation gives the same two permissions, for the same reasons, to a GitHub App that uses git (choosing permissions for Git access). To add them, press **Add permissions** under **Permissions** and tick both in the list it opens. Three rows appear, because GitHub adds Metadata ("Required", "Read-only") by itself. Workflows comes in as "Read and write", the only choice its menu has, but Contents comes in as "Read-only", so change it to "Read and write".

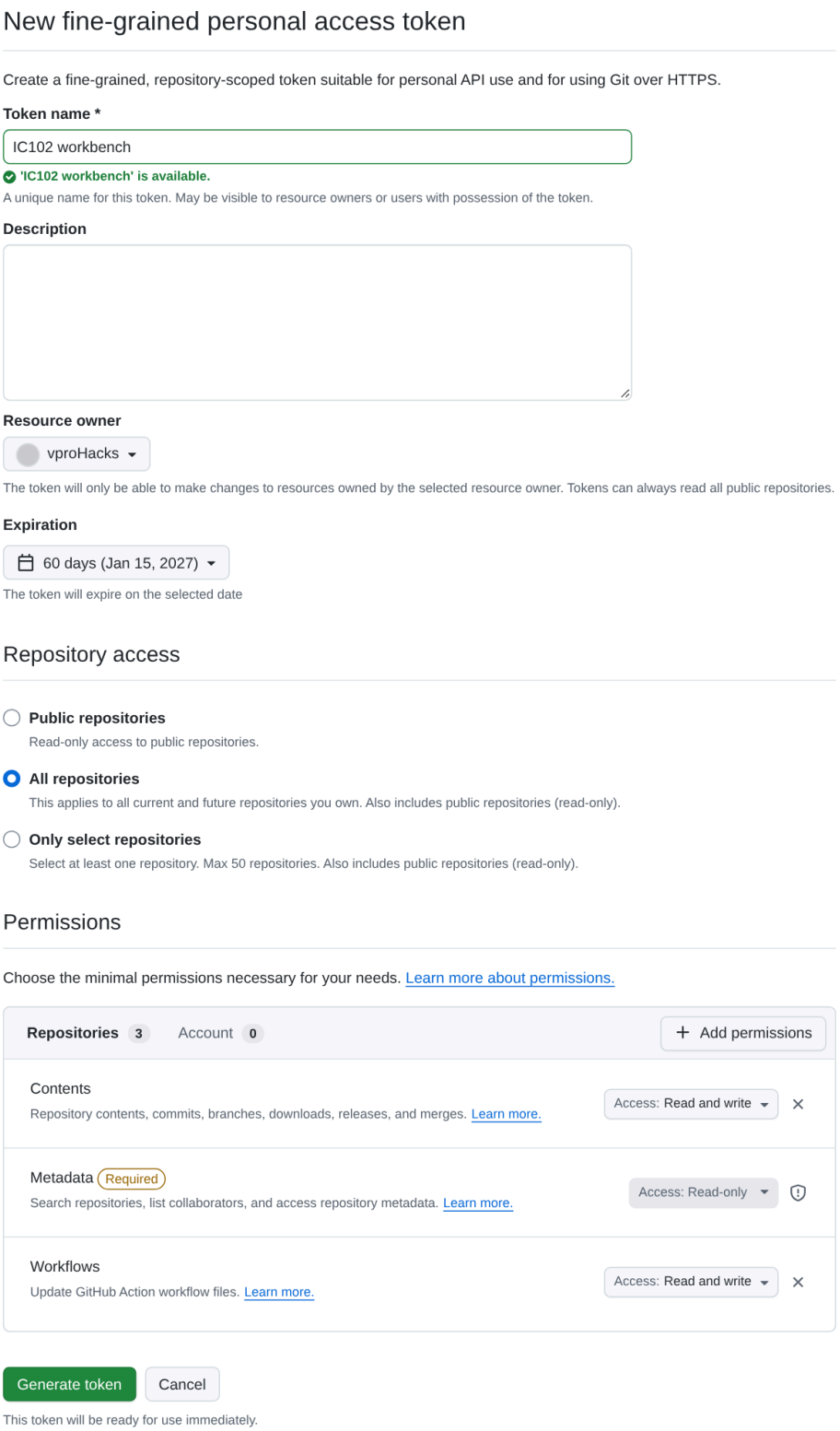



Figure 5.2: The token form filled in as this step says, in the course's practice run on Nov 16, 2026, when the 60-day expiry fell on Jan 15, 2027. The resource owner is the account the practice run used; yours shows your own username.

Press **Generate token** at the bottom of the form, and copy the token when GitHub shows it, because GitHub shows it only once. Keep it in your password manager. A file in your workspace is a bad place for it, because the workspace is a git clone and a file there can end up in a commit. Getting started says why the course uses this kind of token and not a classic one.

You should see: a no-reply address and a token, both copied somewhere safe.

## Part 3: Make the repository

What does the template give you, and what will its workflows say about it before you've changed anything?

1. Open https://github.com/TinyTapeout/ttsky-verilog-template, press **Use this template › Create a new repository**, and set the owner to your own account. Name it `ttsky-ic102-d`, choose **Public**, and leave **Include all branches** off. (The labs use that name in their commands; another name works if you change it there too.) It must be public: GitHub's runners are free on public repositories, and Pages, which the last job needs, is available on a free account only for public ones.

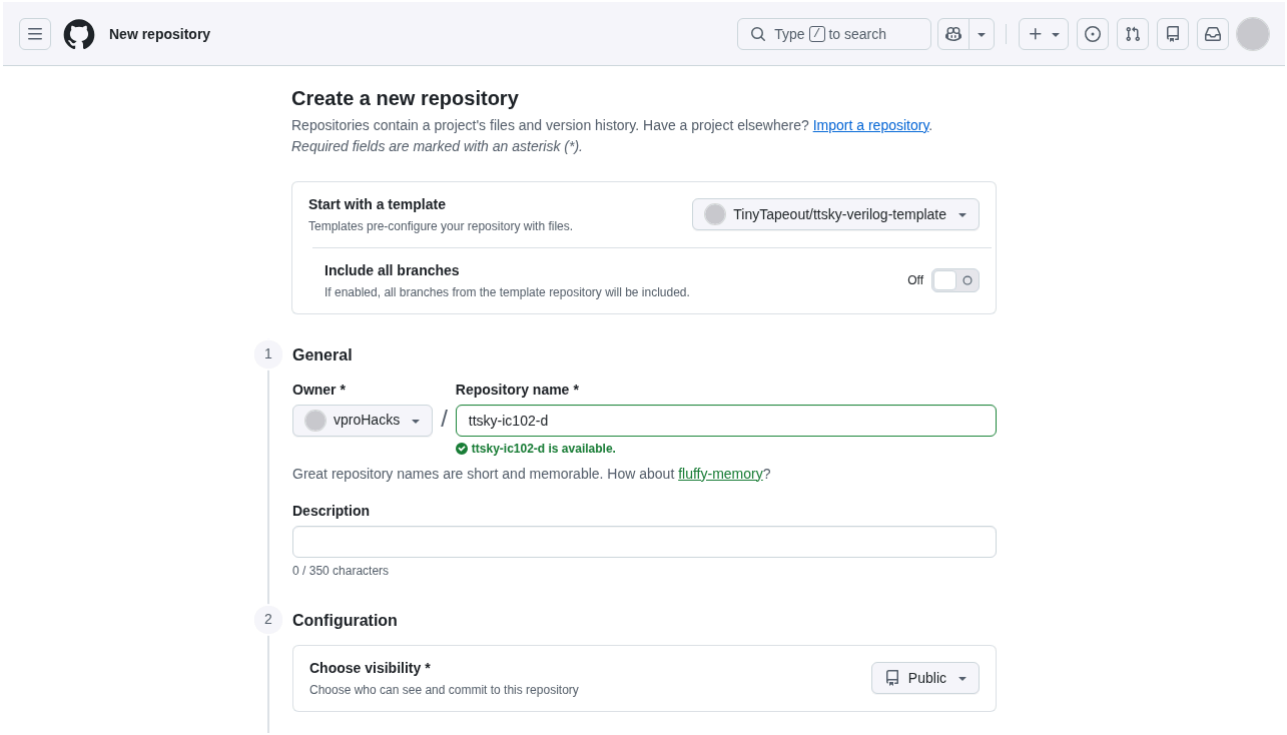



Figure 5.3: Creating a repository from the template.

2. GitHub makes the repository and commits the template's files to it, and that commit counts as a push, so three workflows have already started: `docs`, `test` and `gds`. (The fourth, `fpga`, is switched off.) Before you look, predict which of the three pass on this first commit.
3. Clone the repository into the `projects` folder of your workspace, and set your identity and token cache for it:

```
mkdir -p /foss/designs/projects && cd /foss/designs/projects
git clone https://github.com/YOUR-USERNAME/ttsky-ic102-d.git
cd ttsky-ic102-d
git config user.name "Your Name"
git config user.email "ID+USERNAME@users.noreply.github.com"
git config credential.helper 'cache --timeout=28800'
```

Put your username, your name and the no-reply address from Part 2 in place of the placeholders. These three settings go in this clone's `.git/config`, which is on your own disk. Settings made with `--global`

would be lost the next time the container starts, so you'll make the same three settings in every clone in this course.

4. Look at what came with the template:

```
git log --oneline
ls .github/workflows
grep -n "uses: TinyTapeout" .github/workflows/*.yaml
```

You should see one commit, `Initial commit`, the four workflow files `docs.yaml fpga.yaml gds.yaml test.yaml`, and six lines ending in the same tag:

```
.github/workflows/docs.yaml:17:      uses: TinyTapeout/tt-gds-action/docs@ttsky26d
.github/workflows/fpga.yaml:19:             uses:  TinyTapeout/tt-gds-action/fpga/
ice40up5k@ttsky26d
.github/workflows/gds.yaml:17:       uses: TinyTapeout/tt-gds-action@ttsky26d
.github/workflows/gds.yaml:26:                   uses:   TinyTapeout/tt-gds-action/
precheck@ttsky26d
.github/workflows/gds.yaml:38:                   uses:   TinyTapeout/tt-gds-action/
gl_test@ttsky26d
.github/workflows/gds.yaml:47:    - uses: TinyTapeout/tt-gds-action/viewer@ttsky26d
```

`ttsky26d` names the version of Tiny Tapeout's actions that your repository uses. Your workflow files keep naming `ttsky26d` whatever Tiny Tapeout later changes in its template, although Tiny Tapeout can move the tag itself to a new commit, as The Tiny Tapeout workflows explains.

5. Now open your repository's **Actions** tab and compare it with your prediction.

*Answer: A1, in Answers at the end.*

You should see: one commit in your clone, and one green and two red workflows on GitHub.

## Part 4: Fill it in

Which of the six jobs pass once `info.yaml` and `docs/info.md` are filled in?

1. Edit `info.yaml`, reading each comment as you go. Fill in `title`, `author` (whatever name you want public; it ends up in the datasheet), `description`, and the pinout. The design is still the template's adder, which adds `ui_in` and `uio_in` and puts the sum on `uo_out`, so name the pins for that, for example `ui[0]: "a[0]"`, `uio[0]: "b[0]"` and `uo[0]: "sum[0]"`. Leave `top_module`, `tiles` and `clock_hz` as they are. Don't add or delete any pins.
2. Edit `docs/info.md`. Replace the sentence under "How it works" and the one under "How to test" with a few real sentences each, because the docs check fails for as long as the template's sentences are still there.
3. Commit and push. The first push asks for your GitHub username, and then for a password: paste the token there. The cache remembers it for 8 hours.

```
git add info.yaml docs/info.md
git commit -m "Fill in info.yaml and docs/info.md"
git push
```

4. Predict the result of each job before you look: `docs`, `test`, and the four jobs of `gds`. The `gds` job builds the layout this time, which took 2 min 50 s in the course's practice repository, and the three jobs after it, which run side by side, took another 2 min 26 s.

*Answer: A2, in Answers at the end.*

5. Open the `viewer` job's log (you have to be signed in to GitHub to see it) and find the line that says what to do. In the practice repository's run 43826443768 it was:

```
##[error]Error: Failed to create deployment (status: 404) with build version 1e5f04aba0db818605ad318fb00c018d94be6d98. Request ID 9BB4:34D4D8:189992:27A7DC:6AFAB621 Ensure GitHub Pages has been enabled: https://github.com/vproHacks/ic102-test-d/settings/pages
```

The line in your log has your own commit as the build version, and your own repository's settings page at the end.

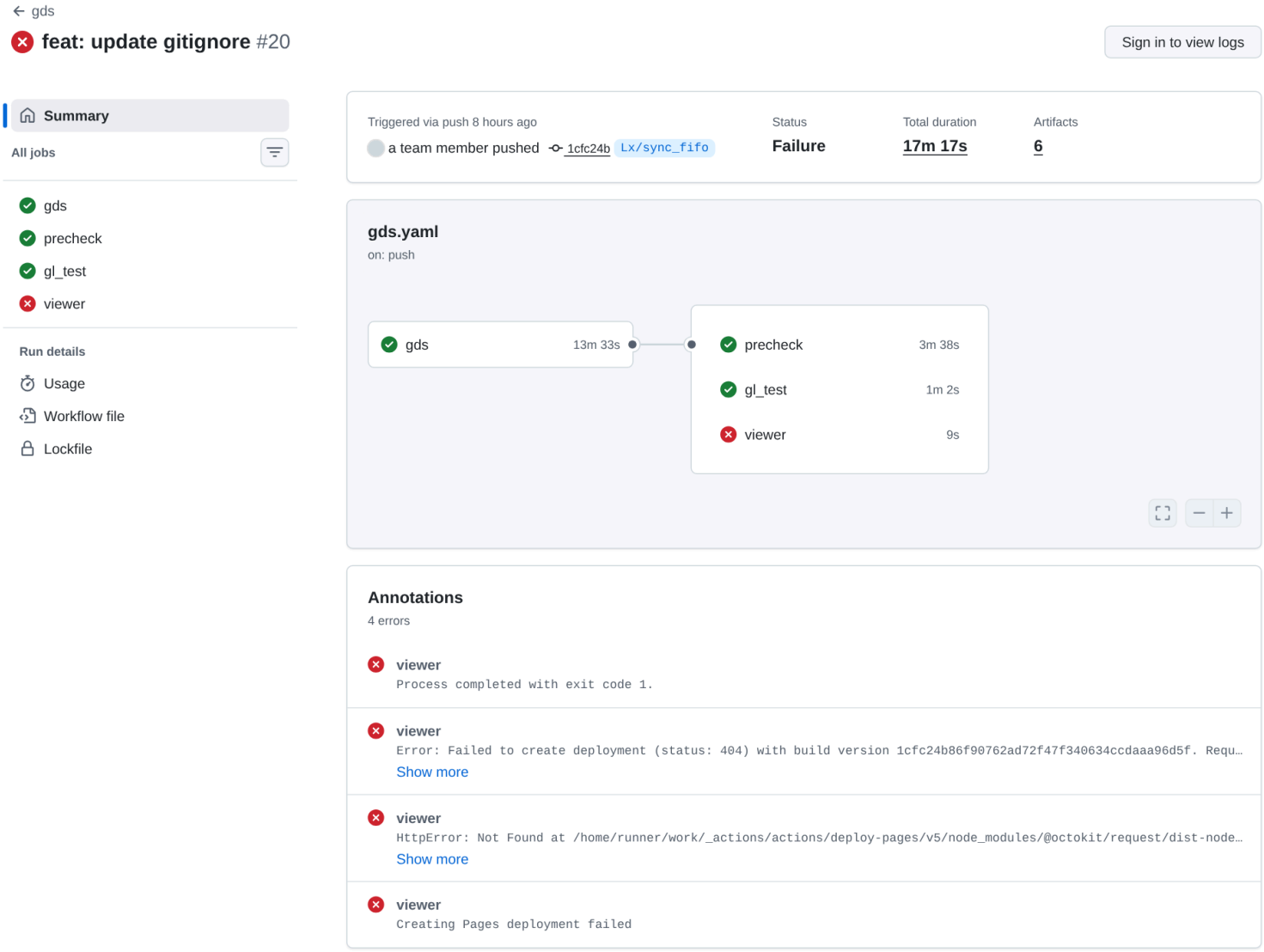



Figure 5.4: A protocol-emulator gds run where only the viewer job failed.

You should see: `docs` and `test` green, and in `gds` everything green except `viewer`, with the 404.

# Part 5: Turn on Pages

How do you fix a failure that isn't in any file?

1. On GitHub, open your repository's **Settings › Pages**, and under **Build and deployment › Source**, choose **GitHub Actions**.

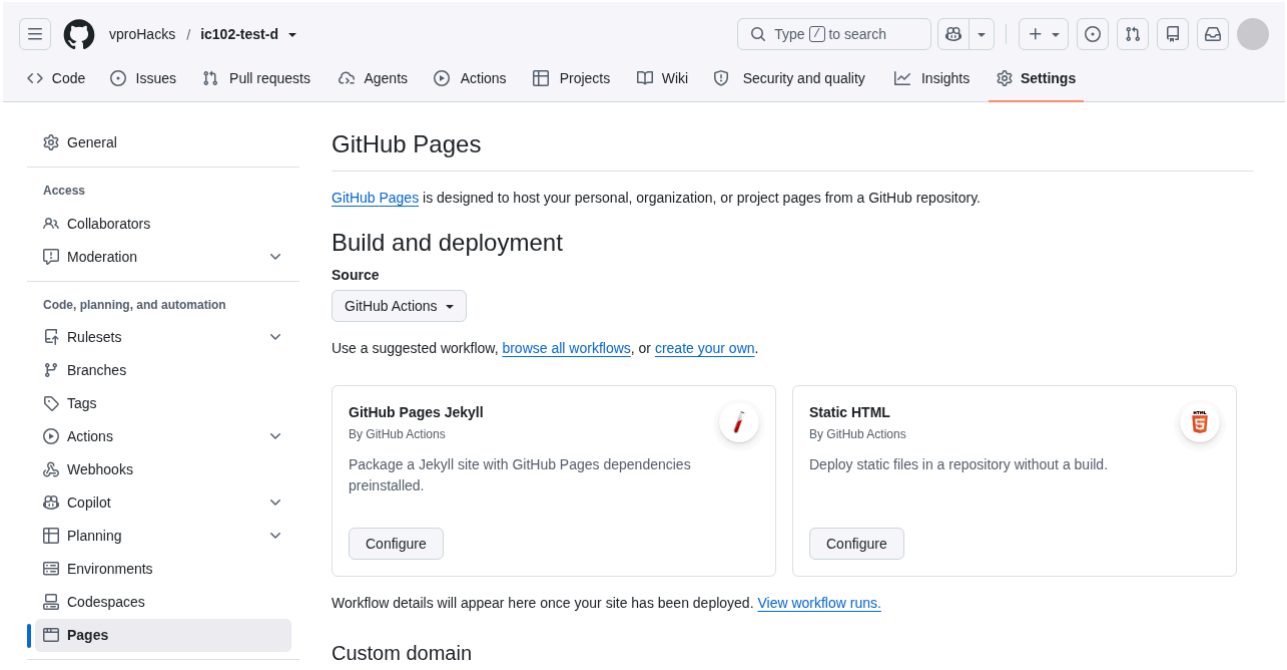



Figure 5.5: Settings › Pages, with the source set to GitHub Actions.

2. Start a new run of the `gds` workflow: on the **Actions** tab, choose **gds** in the list of workflows, then **Run workflow** on the right, and run it on `main`. (Pushing a new commit would start a new run too.)

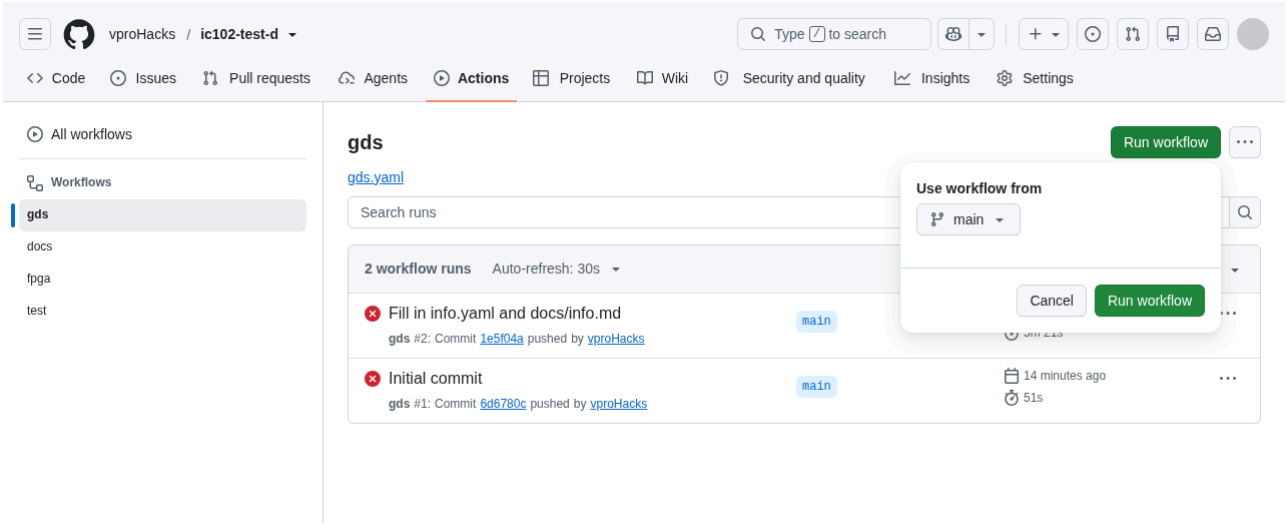



Figure 5.6: Starting a new run of the gds workflow by hand: the menu under Run workflow, with the branch set to main. The run starts only when you press the green Run workflow button inside the menu.

Start a new run rather than pressing **Re-run failed jobs** on the old one. A re-run keeps the first attempt's Pages artifact, and the `viewer` action at the `ttsky26d` tag uploads an artifact with the same name on every attempt, so the re-run fails with `Multiple artifacts named "github-pages"`. A new run starts with no artifacts, so it doesn't have this problem. `sentry`'s re-runs failed this way, and Reading a workflow file has the log.

3. When the run finishes, every job should be green. The practice repository's new run took 4 min 5 s, and looked like this run of `sentry`'s:

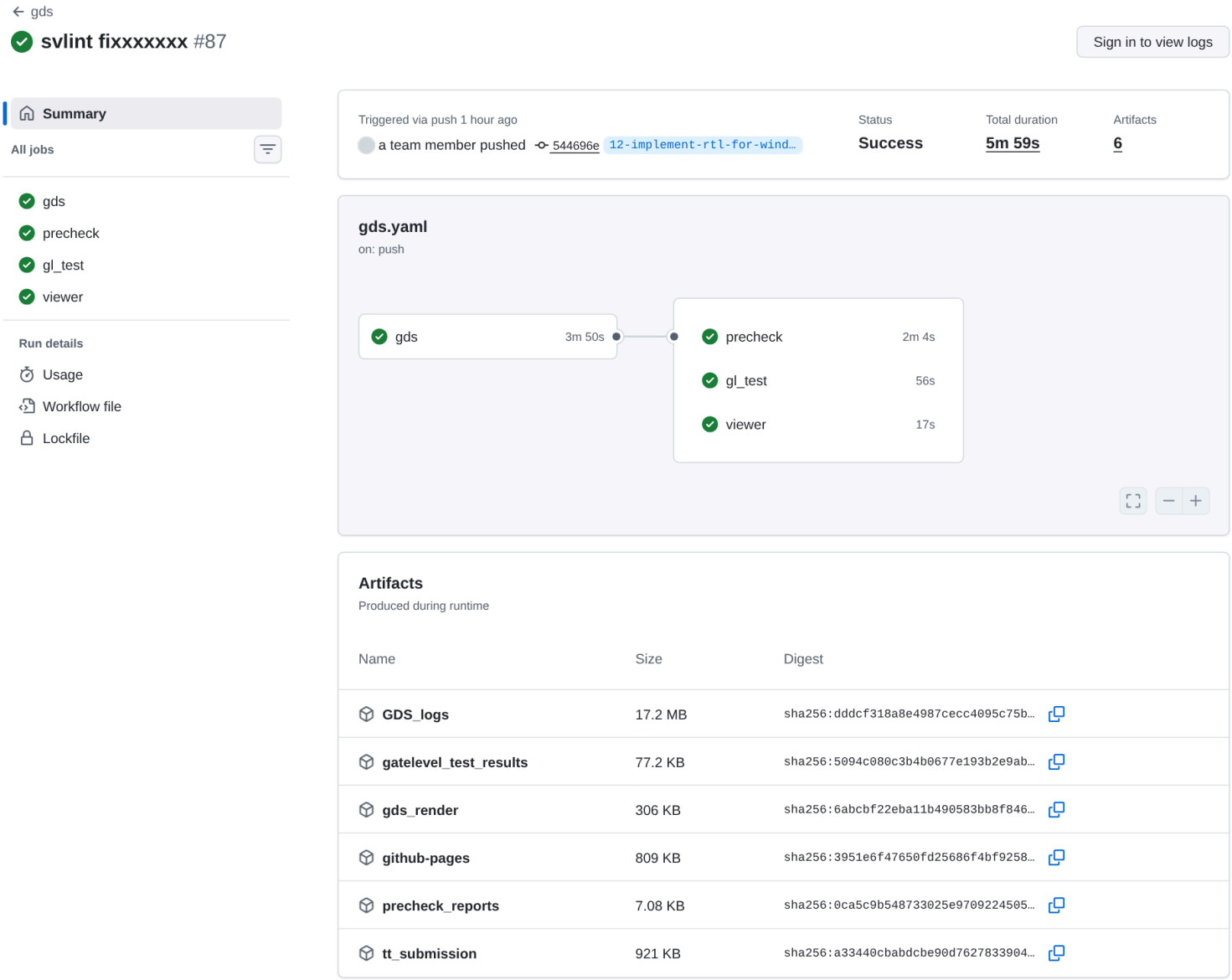



Figure 5.7: A fully green sentry gds run and its six artifacts.

4. Open your project's page; Settings › Pages now shows its address. Tiny Tapeout's 3D viewer reads your layout from that page.
5. Check your work:

```
python3     /foss/designs/modules/ic102-d/labs/lab-01-your-repo-from-the-template/
check.py
```

It checks your files and your history, then reads your commit's jobs from GitHub's public API. On the course's practice repository, `ic102-test-d`, cloned into the folder these steps use, it printed:

```
Checking /foss/designs/projects/ttsky-ic102-d

GitHub: vproHacks/ic102-test-d at 1e5f04a
  docs      success
  test      success
  gds       success
  precheck  success
```

```
  gl_test   success
  viewer    success

VERDICT: PASS
```

`gds` and the three jobs after it show the new run's results, because when a job has run more than once on the same commit, the check reads the latest.

You should see: six green jobs on your commit, and `VERDICT: PASS`.

## Finishing

You're done when `check.py` prints `VERDICT: PASS`, which it does when `info.yaml` and `docs/info.md` are filled in, HEAD is a commit of yours on top of the template's, and all six jobs passed on it.

Questions:

1. The template's own first commit passed `test`. What did that pass tell you about the design?

   *Answer: A3, in Answers at the end.*

2. Your fix in Part 5 didn't change any file. How would someone reading your history later know why the `viewer` job started passing?

   *Answer: A4, in Answers at the end.*

Submit: the link to your repository.

Next on the Course path: read What a cloud runner is, Reading a workflow file and The Tiny Tapeout workflows, then do Lab 02: Read the red X.

## Answers

**A1. Part 3: Make the repository**

`test` passes, and `docs` and `gds` fail. `test` runs the template's own test on the template's adder, and the adder is correct. `docs` and `gds` both read `info.yaml`, whose title, author and description are empty. In `gds` only the first job runs, and the other three are skipped because each one has `needs: gds`. `hart-1`'s first commit, made from the same template on Oct 18, got the same results. Here are its check runs, read with the course's `check-runs.py`:

```
uoftasic/hart-1 at 29313b4
  docs      failure
  test      success
  gds       failure
  precheck  skipped
  gl_test   skipped
  viewer    skipped
```

The course's practice repository, made from the template on Nov 16, 2026, got the same six results on its first commit, with `docs` red after 25 s and `gds` red after 48 s.

**A2. Part 4: Fill it in**

Five pass and `viewer` fails. `viewer` publishes your layout with GitHub Pages, and Pages is off in a repository made from a template. The same failure kept `protocol-emulator`'s `gds` runs red: in the two of its runs we checked, the layout built and precheck passed, and only `viewer` failed. The practice repository got the same: five green, and `viewer` red after 16 s.

**A3. Finishing**

It told you very little. The test drives one pair of inputs, 20 and 30, and checks for 50, so all it shows is that the adder gets that one sum right. A design that handled only that one case would pass it too. You'll write a stronger test in Lab 03.

**A4. Finishing**

They wouldn't, from the history alone: turning on Pages changed a setting and made no commit, and the run logs that show the 404 are deleted after 90 days. That's a reason to write the change down somewhere it lasts, for example in the README, or in an issue.

# II 2. Workflows


6 **What a cloud runner is . . . . . 37**
6 A new machine for every job . . . . . 37
6 How big it is and how long it takes . . . . . 38
6 What it costs . . . . . 38
6 Your tools and the runner's tools . . . . . 38
6 Check yourself . . . . . 39
6 Answers . . . . . 39

7 **Reading a workflow file . . . . . 41**
7 What a push starts . . . . . 41
7 The parts of a workflow file . . . . . 41
7 Composite actions . . . . . 43
7 Reading a failed run . . . . . 44
7 Job summaries and artifacts . . . . . 46
7 Re-running a run . . . . . 48
7 GitHub Pages and the viewer . . . . . 50
7 Check yourself . . . . . 52
7 Answers . . . . . 52

8 **The Tiny Tapeout workflows . . . . . 55**
8 One push, six jobs . . . . . 55
8 docs . . . . . 55
8 test . . . . . 56
8 gds . . . . . 56
8 What green means in this course . . . . . 57
8 Tags and support tools that move . . . . . 57
8 Check yourself . . . . . 57
8 Answers . . . . . 58

9 **Lab 02: Read the red X . . . . . 59**
9 Before you start . . . . . 59
9 Part 1: The diagnosis file . . . . . 59
9 Part 2: Breakage 1, step by step . . . . . 60
9 Part 3: Breakage 2 . . . . . 60
9 Part 4: Breakage 3 . . . . . 61
9 Part 5: Breakage 4 . . . . . 61
9 Part 6: Breakage 5 . . . . . 62
9 Finishing . . . . . 62
9 Answers . . . . . 63

# 6. What a cloud runner is

This page answers: *when you push to a Tiny Tapeout repository, where do the checks actually run, what kind of machine is it, how long does it take, and what does it cost?*

The green ticks and red crosses next to your commits come from programs that GitHub runs for you, on machines you never see. Those machines are called **runners**, and a few facts about them explain a lot of what you'll see in the logs: why every job starts by installing tools, why jobs hand files to each other the way they do, why a push takes minutes to come back, and why a check can pass on your laptop and fail on GitHub.

## A new machine for every job

GitHub's documentation defines runners as "the machines that execute jobs in a GitHub Actions workflow", and says that each GitHub-hosted runner "is a new virtual machine (VM) hosted by GitHub" (GitHub-hosted runners). So every job in every workflow gets a new machine that has never seen your repository. The machine runs the job's steps from top to bottom, and GitHub throws it away when the job ends.

You can see this at the top of every job's log. Here are the first lines of the `viewer` job in a `protocol-emulator` run on Nov 15, 2026 (run 43757724629), with the timestamps taken off the front of each line:

```
Current runner version: '2.338.2'
##[group]Runner Image Provisioner
Hosted Compute Agent
Version: 20261103.629
Commit: 15dbd9ccb5dbf34fae002b6ed1a2670f52df37bc
Build Date: 2026-11-03T17:26:59Z
Worker ID: {69e406aa-39b1-443e-812e-16fa60d2a9fb}
Region: centralus
Cloud: Azure
##[endgroup]
##[group]Operating System
Ubuntu
24.04.5
LTS
##[endgroup]
##[group]Runner Image
Image: ubuntu-24.04
Version: 20261111.370.1
```

So that job ran on an Ubuntu 24.04 virtual machine in a Microsoft Azure data centre, built from build `20261111.370.1` of GitHub's `ubuntu-24.04` image. The `##[group]` lines are folds in the log viewer and not part of the output. GitHub rebuilds the image from time to time, and two jobs from the same afternoon can land on different builds of it: the `test` job of a `hart-1` run on the same day reports `Version: 20261111.127.1`.

Three consequences follow from a fresh machine per job, and you'll meet all of them in the Tiny Tapeout workflows:

- **Every job installs its own tools.** The `test` job installs Icarus Verilog and cocotb, the `gds` job installs LibreLane, and the `precheck` job installs the PDK and builds Magic and KLayout. Nothing is left over from your last push.
- **Jobs pass files to each other as artifacts.** The `gds` job finishes by uploading its results as an artifact called `tt_submission`, and the `precheck` and `viewer` jobs start by downloading it, because they run on other machines and can't see the `gds` job's disk.

- **The machine has only what's in your repository.** A file you forgot to commit, a tool you installed on your laptop, or a setting in your shell doesn't exist there. When a check passes for you and fails on the runner, this is the first place to look.

## How big it is and how long it takes

A standard Linux runner for a public repository has 4 CPUs, 16 GB of RAM and a 14 GB SSD. For a private repository it has 2 CPUs and 8 GB, with the same 14 GB of disk (runner reference). That's about what a decent laptop has.

These are the times recorded in the team's own runs:

| Job | Time | Where it was measured |
|---|---|---|
| `test` workflow | 20 to 46 s | The team's repositories |
| `gds` | 207 to 230 s | `sentry` (2x2 tiles, SKY130) |
| `gds` | 510 to 573 s | `ttsky-verilog-tinyCGRA` (1x1 tile, SKY130) |
| `gds` | 791 to 1,056 s | `protocol-emulator` (6x2 tiles, IHP) |
| `precheck` | 91 to 315 s | The team's repositories |
| `gl_test` | 35 to 62 s | The team's repositories |
| `viewer` | 6 to 25 s | The team's repositories |

Tiny Tapeout's FAQ says that "a typical Tiny Tapeout GDS action often completes in about 5 minutes for a small design", and that "20-25 minutes or more" suggests the design "may be running out of available tile area". Because `precheck`, `gl_test` and `viewer` all wait for `gds`, a push to a digital repository takes somewhere between 5 and 20 minutes to come back fully green. That's long enough that you should plan around it: push, then read the next page or work on something else while you wait, rather than watching the progress bar.

How long a job ran before it failed is a clue to the cause. Very fast red usually means a setup or configuration problem, and slow red usually means a problem in the design itself. `hart-1`'s `gds` job, for example, fails in 54 seconds, because the build stops at the first step that reads `info.yaml`, long before LibreLane starts.

## What it costs

"Use of the standard GitHub-hosted runners is free and unlimited on public repositories" (runner reference). That's the main reason this course has you make your repository public.

Private repositories draw on an allowance. On the Free plan it's stated as $12 a month, which is the same as the 2,000 minutes a month that GitHub's pricing page still quotes, at the rate of $0.006 a minute for the smallest Linux runner. Usage is rounded up to the whole minute for each job (Actions billing, runner pricing). The rounding matters more than you'd think for a Tiny Tapeout repository, because a push to a digital one starts six jobs. In the `protocol-emulator` run above, the four jobs of the `gds` workflow took 13 min 33 s, 3 min 38 s, 1 min 2 s and 9 s, which round up to 14 + 4 + 2 + 1 = 21 minutes. Add a minute each for `docs` and `test`, and one push costs 23 minutes, so a private repository like it would use up the month's allowance in about 87 pushes, and probably fewer, since a private repository's runner has half the CPUs and the jobs would take longer. In a public repository the same pushes cost nothing.

A few limits apply either way. A single job can run for at most 6 hours, a Free account can run 20 jobs at once, and a run can be re-run at most 50 times (limits). You'll work in your own GitHub account, so your jobs queue against your own limit and not against anyone else's.

Logs and artifacts don't last forever. By default GitHub keeps them for 90 days, and History as evidence shows what that looks like when you need an old log and it's gone.

## Your tools and the runner's tools

The runner installs its tools fresh on every run, at whatever versions the workflow asks for, and those needn't match the tools on your laptop or in the workbench. In a `hart-1` test run on Nov 15, 2026, the `test` job installed Icarus Verilog `12.0-2build2` from Ubuntu's packages and `cocotb-2.1.0` from the

repository's `test/requirements.txt`. The `gl_test` job installs a different Icarus, Tiny Tapeout's own 13.0 package, and at the digital template's current tag (`ttsky26d`) the `gds` job installs LibreLane 3.0.14. The `precheck` job, in both templates, builds its own Magic and KLayout and installs the PDK at a fixed commit. A difference in any of these can make a check pass in one place and fail in the other.

For that reason IC102 pins the tools you run locally to the versions Tiny Tapeout's workflows use, so that your local runs and the runner's agree. Your tools and the runner's tools lists them for your version of the course, and says where the two still differ.

You might wonder whether you could run the workflows themselves on your own machine and skip the wait. There is a community tool for that, `act`, which "reads in your GitHub Actions from `.github/workflows/`" and runs them in Docker containers (nektos/act). It isn't GitHub's, though, and its own documentation says that "Default runners are intentionally incomplete". Among the features it lists as unsupported are step summaries, `job.permissions` and OIDC (not supported), and the `viewer` job needs the last two, so it can't work under `act`. In this course you run the individual tools locally to get quick answers, and treat the runner as the place where a check counts.

## Check yourself

1. The `precheck` job needs the GDS file the `gds` job just built. Why does it download an artifact instead of reading the file from the `gds` job's disk?

*Answer: A1, in Answers at the end.*

2. A push to your repository goes red in under a minute. Is the problem more likely in your design or in your configuration, and why?

*Answer: A2, in Answers at the end.*

3. A friend keeps their Tiny Tapeout repository private on a Free account and pushes about five times a day. Using the `protocol-emulator` numbers on this page, roughly how long will the month's allowance last?

*Answer: A3, in Answers at the end.*

4. Your cocotb test passes in the workbench and fails on the runner. Name two things that could differ between the two machines.

*Answer: A4, in Answers at the end.*

Next: Reading a workflow file.

## Answers

**A1. Check yourself**

Because the two jobs run on different machines. Each job gets a new virtual machine that's thrown away when the job ends, so the only way to hand a file from one job to the next is to upload it as an artifact (`tt_submission`) and download it in the later job.

**A2. Check yourself**

In your configuration. Building a layout takes several minutes, so a job that fails within a minute stopped before the build started, usually at a step that reads `info.yaml` or another setup file, as in `hart-1`'s 54-second `gds` failure.

**A3. Check yourself**

At about 23 billed minutes a push, five pushes a day use about 115 minutes a day, so 2,000 minutes last about 17 days. Made public, the same repository would cost nothing.

**A4. Check yourself**

Any two of: the simulator version (the runner's `test` job installed Icarus 12.0 from Ubuntu's packages on Nov 15, 2026), the cocotb version (whatever `test/requirements.txt` pins, which was 2.1.0 in the digital template), a file you have locally but never committed, or a setting in your own shell that the runner doesn't have.

# 7. Reading a workflow file

This page answers: *what does a workflow file tell GitHub to do, and when a run goes red, how do you find the one line that says why?*

You won't need to write a workflow from scratch in this course, because Tiny Tapeout has written them for you. You will read them, though, and read the runs they produce, because a red cross on its own tells you only that something failed. The workflow file tells you what was supposed to happen, and the run's log tells you where it stopped. With both, most failures take a minute to explain. You'll practise this on five broken commits in Lab 02.

## What a push starts

Each YAML file in `.github/workflows/` is one **workflow**. When you push, GitHub starts every workflow whose trigger includes a push, and each one becomes a separate **run** with its own entry on the Actions tab and its own tick or cross next to your commit. A run is made of **jobs**, and each job is a list of **steps**.

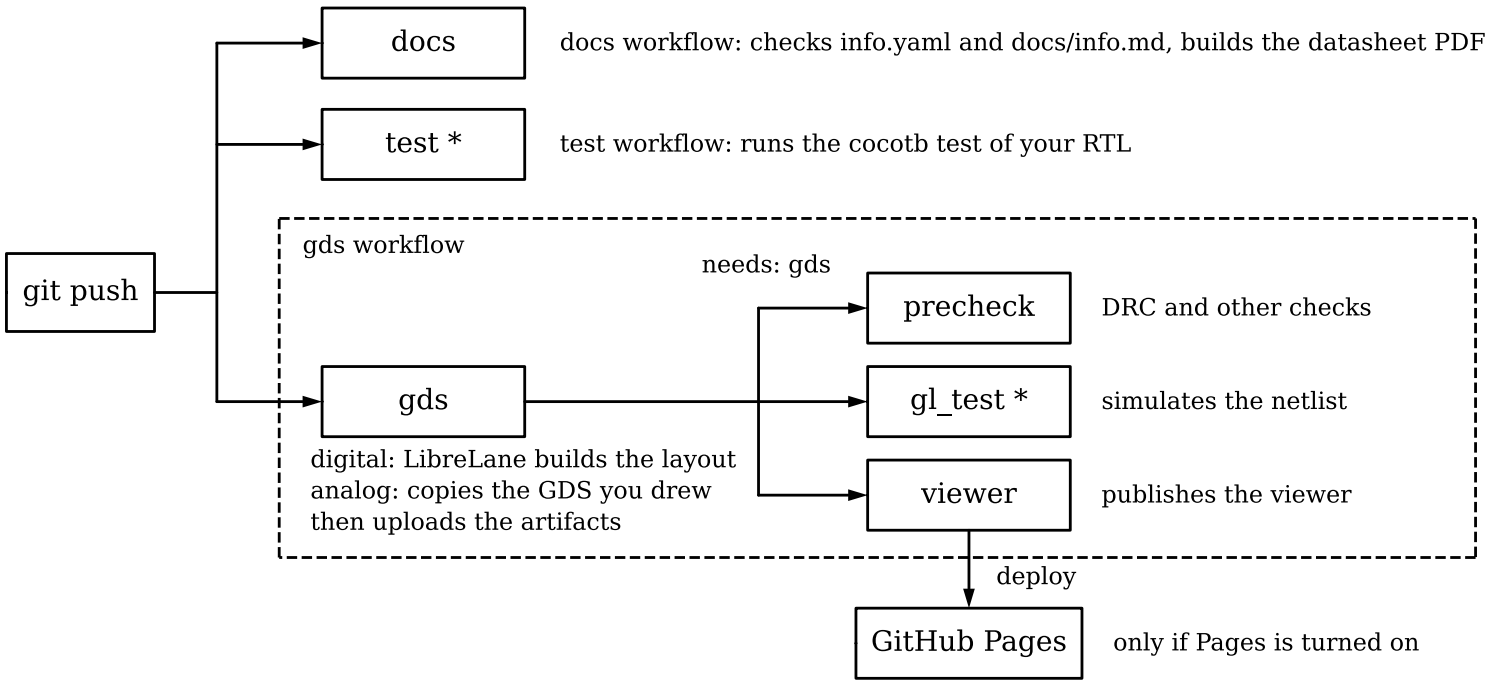



Figure 7.1: What one push starts in a Tiny Tapeout repository.

The digital template has four workflow files (`docs.yaml`, `test.yaml`, `gds.yaml` and `fpga.yaml`), but `fpga` is switched off, so a push starts three runs. The analog template has two, `docs.yaml` and `gds.yaml`. The page after this one in the course path goes through your template's workflows job by job; this page is about how to read any of them.

## The parts of a workflow file

Here is the digital template's `gds.yaml` in full, as it was at the template's commit `a42e666` (Oct 14, 2026):

```yaml
name: gds

on:
  push:
  workflow_dispatch:

jobs:
  gds:
    runs-on: ubuntu-24.04
    steps:
      - name: checkout repo
        uses: actions/checkout@v6
        with:
          submodules: recursive

      - name: Build GDS
        uses: TinyTapeout/tt-gds-action@ttsky26d
        with:
          pdk: sky130A

  precheck:
    needs: gds
    runs-on: ubuntu-24.04
    steps:
      - name: Run Tiny Tapeout Precheck
        uses: TinyTapeout/tt-gds-action/precheck@ttsky26d

  gl_test:
    needs: gds
    runs-on: ubuntu-24.04
    steps:
      - name: checkout repo
        uses: actions/checkout@v6
        with:
          submodules: recursive

      - name: GL test
        uses: TinyTapeout/tt-gds-action/gl_test@ttsky26d

  viewer:
    needs: gds
    runs-on: ubuntu-24.04
    permissions:
      pages: write      # to deploy to Pages
     id-token: write   # to verify the deployment originates from an appropriate source
    steps:
      - uses: TinyTapeout/tt-gds-action/viewer@ttsky26d
```

Reading it from the top:

- **`name`** is what the Actions tab and the badges call the workflow.
- **`on`** says what starts it. Here that's `push`, on any branch, and `workflow_dispatch`, which puts a **Run workflow** button on the Actions tab so you can start a run by hand. None of the template's workflows list `pull_request`, so opening a pull request doesn't start anything by itself; the runs you see on a pull request come from the pushes to its branch.
- **`jobs`** holds four jobs, named `gds`, `precheck`, `gl_test` and `viewer`. Each runs on its own fresh machine, chosen by **`runs-on: ubuntu-24.04`**, of the kind that What a cloud runner is describes.
- **`needs: gds`** makes a job wait until `gds` has finished, and run only if it succeeded. Jobs without `needs` start at once and in parallel, so the three jobs that need `gds` all start together when it's done.

- **steps** run in order, and a step either runs a shell command (`run:`, as in `test.yaml`) or calls an **action** (`uses:`), which is a packaged set of steps kept in another repository. The text after the `@` is the version of the action to use: a tag (`ttsky26d`), a version number (`v6`), or a commit hash.
- **with** passes inputs to an action: `submodules: recursive` tells the checkout action to fetch submodules too, and `pdk: sky130A` tells Tiny Tapeout's action which process to build for.
- **permissions** sets what the job's automatic access token may do. `viewer` asks to write to Pages and to get an identity token, because GitHub requires "a minimum of `pages: write` and `id-token: write` permissions" for a job that deploys a Pages site (custom workflows).

The analog template's `gds.yaml` has the same shape with two differences. Its `gds` job reads `top_module` from `info.yaml` and passes `gds/<top_module>.gds` and `lef/<top_module>.lef` to a different action, `tt-gds-action/custom_gds`, which copies and converts your layout instead of building one. It also has no `gl_test` job.

One step in the digital `test.yaml` is worth reading closely, because it shows how GitHub decides whether a step passed:

```
      - name: Run tests
        run: |
          cd test
          make clean
          make
          # make will return success even if the tests fail, so check results.xml for
failed tests
          python -m cocotb_tools.check_results results.xml
```

GitHub runs these lines with `bash -e` (the log prints `shell: /usr/bin/bash -e {0}` above the step), so the step stops and fails as soon as any command exits with a non-zero status, and passes if none does. That exit status is all GitHub looks at. The comment above the last line was true of cocotb 1.x, whose `make` exited with status 0 even when a test failed, which is why the template adds a command that reads the results file and exits non-zero if any test failed. The template now pins cocotb 2.1.0, and with cocotb 2.x `make` runs that same check itself at the end of the simulation, so a failing test stops the step at `make`, and the last line is a second check that only runs when `make` has passed. Either way, what turns a failing test into a red step is a command that exits non-zero. You'll need the same idea in Lab 05, where a script has to tell git whether a commit passed.

## Composite actions

The `gds` job above has two steps, and the second is one line: `uses: TinyTapeout/tt-gds-action@ttsky26d`. That line runs a **composite action**, a file called `action.yml` in the `TinyTapeout/tt-gds-action` repository whose steps run inside your job as if you'd written them yourself. At the `ttsky26d` tag it has 18 steps, beginning with "Set up environment variables" and "Install prerequisites", going through "Fetch verilog and build config", "Install LibreLane" and "Make GDS with LibreLane", and ending with "Publish tt_submission artifact", "Render PNG from GDS" and "Upload gds_render (png) artifact". In the log, all 18 appear as sections nested inside "Build GDS", so when "Build GDS" fails, you open it and look for the section that failed inside it.

A tag is only a name for a commit, and its owner can move it. Tiny Tapeout's own README for the action describes moving a tag that way, with `git tag -fa` and `git push --force`, when they update a release. The log records which commit the tag pointed to when your job ran, in a line near the top of each job. This is from a `sentry` run on Nov 6, 2026, on the tag the current SKY130 templates use:

```
Download        action        repository        'TinyTapeout/tt-gds-action@ttsky26d'
(SHA:395eedb3eeec3c2f4961c3b04ff2cf87f5525636)
```

If a run that passed last month fails today with no change on your side, compare that hash between the two logs. (The action also fetches Tiny Tapeout's support tools from their `main` branch by default, so those can change between runs even when the tag stays where it was.)

## Reading a failed run

The path to the cause is always the same: the run, then the job that failed, then the step that failed inside it, then the first error line in that step. GitHub helps with the third part: “Any failed steps are automatically expanded” when you open a job’s log (workflow run logs).

Start with the run’s summary page. Anyone can see it for a public repository, even without signing in. Here’s the `gds` run that `hart-1` made when pull request #5 was merged on Nov 15, 2026 (run 43752267356):

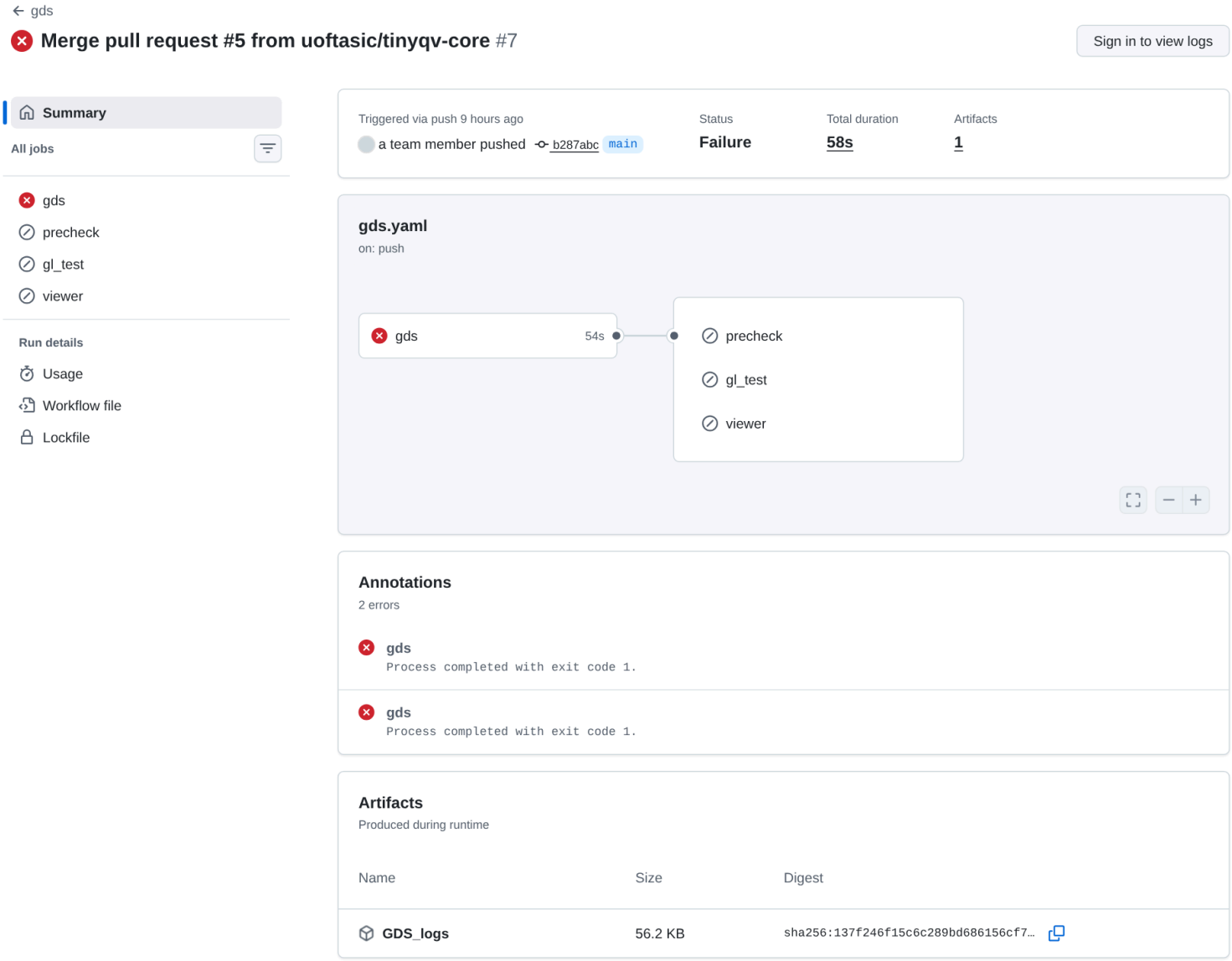



Figure 7.2: A hart-1 gds run that stopped in its first job. The person who pushed is shown as a team member.

The graph shows that the `gds` job failed after 54 seconds, which is far too short for a layout build, and that the other three jobs never ran: their grey icons mean “skipped”, because each `needs: gds`. The **annotations** box lists the errors GitHub picked out of the log, but here both just say `Process completed with exit code 1.` They tell you that two steps failed, and not why. For that you need the log itself, which GitHub shows only when you’re signed in.

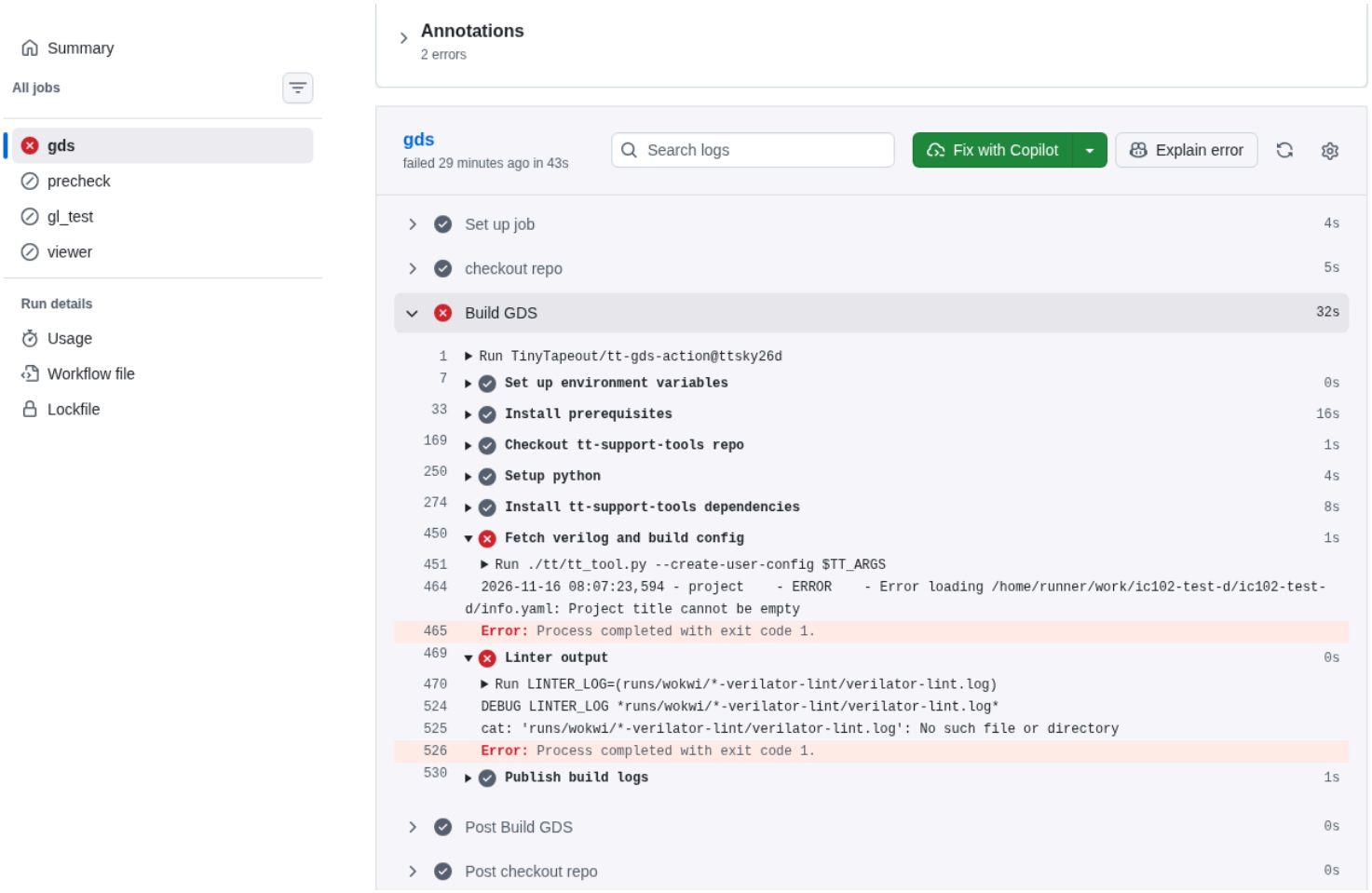



Figure 7.3: The same failure in the course's practice repository, where only the title was empty: the failed "Build GDS" step opens by itself, with its two failed sections, "Fetch verilog and build config" and "Linter output", expanded.

Inside "Build GDS", the step "Fetch verilog and build config" stops with this, which is the cause:

```
2026-11-15 19:25:32,583 - project     - ERROR     - Error loading /home/runner/work/
hart-1/hart-1/info.yaml: Project title cannot be empty, Project author cannot be empty,
Project description cannot be empty
##[error]Process completed with exit code 1.
```

Most of the steps after it are skipped, but two still run. "Publish build logs" uploads whatever logs exist as the `GDS_logs` artifact, and before it, "Linter output" prints a second error:

```
DEBUG LINTER_LOG *runs/wokwi/*-verilator-lint/verilator-lint.log*
cat: 'runs/wokwi/*-verilator-lint/verilator-lint.log': No such file or directory
##[error]Process completed with exit code 1.
```

That step is marked `if: always()` in the action, so it runs even after an earlier step has failed, and it fails because the linter it reports on never ran. It's a consequence, not a second problem, and fixing `info.yaml` makes both errors go away. That's the general rule for logs: read from the top, take the first real error as the cause, and be suspicious of anything printed by a step that runs `always()` or `if: failure()`, because those steps exist to clean up and report after something else has already gone wrong.

The same pattern shows up in the `test` workflow. When `protocol-emulator` renamed its top module in `src/project.v` but not in `test/tb.v` (run 40123836544), the "Run tests" step stopped with the cause:

```
/home/runner/work/protocol-emulator/protocol-emulator/test/tb.v:27:  error:  Unknown
module type: tt_um_example
```

```
2 error(s) during elaboration.
*** These modules were missing:
        tt_um_example referenced 1 times.
***
```

and the two steps that run afterwards regardless each added a message of their own:

```
##[error]ENOENT: no such file or directory, open 'test/results.xml'
```

```
##[warning]No files were found with the provided path: test/tb.fst
test/results.xml
test/output/*. No artifacts will be uploaded.
```

Neither of those is a separate problem. The simulation never started, so there was no results file to summarise and no waveform to upload. The fix was one line in `test/tb.v`.

## Job summaries and artifacts

A step can write Markdown to the run's summary page through a file whose path is in the variable `$GITHUB_STEP_SUMMARY`, up to 1 MiB per step (workflow commands). The GDS action uses this for its linter output, Yosys warnings, a routing summary and a cell-usage summary, so the numbers you're most likely to want after a build are on the summary page without opening any logs.

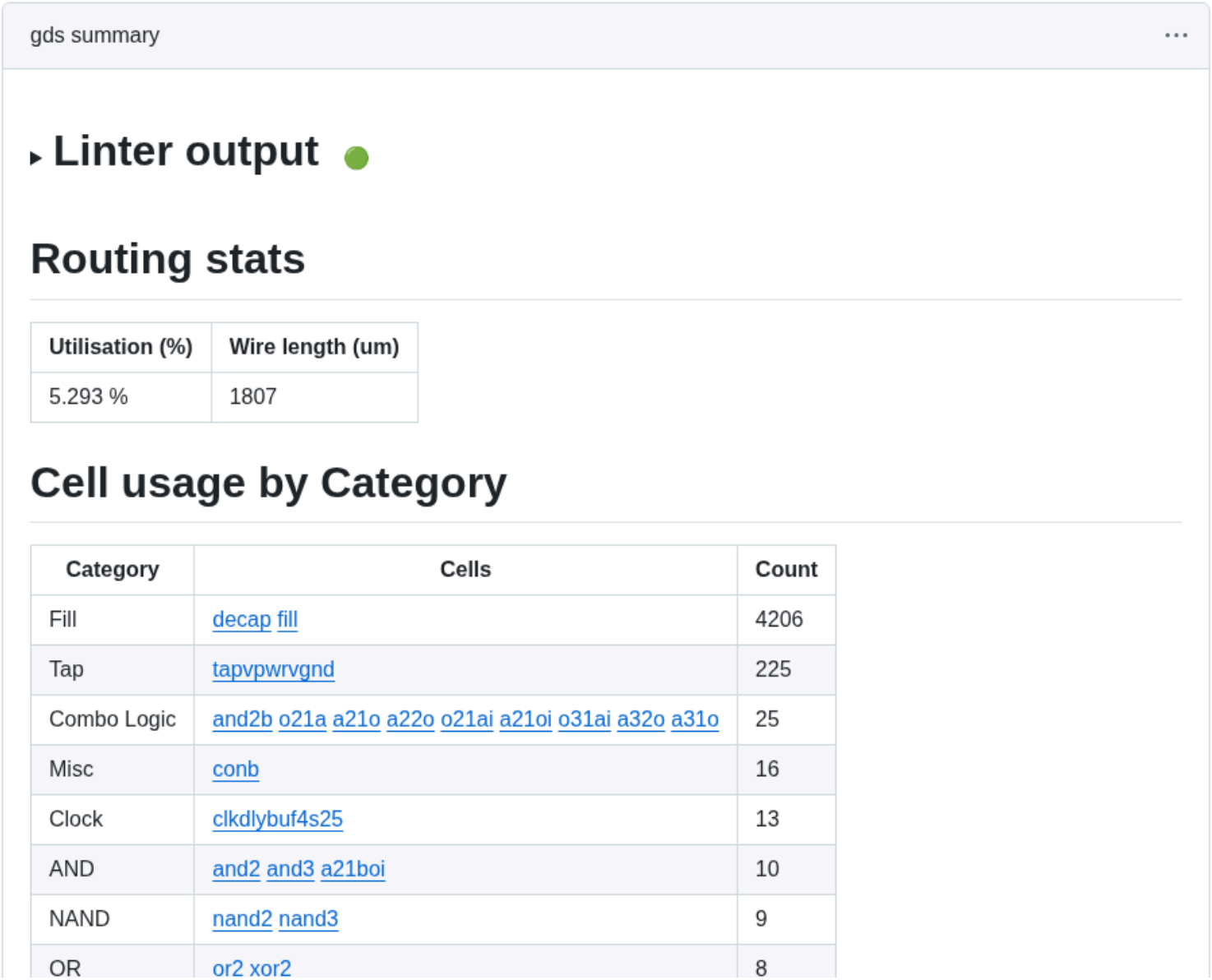



Figure 7.4: The gds job's summary on a run's Summary page, from the course's practice repository: the linter's result, the routing stats and the cell usage.

Everything else a job wants to keep, it uploads as an **artifact**: a zip file attached to the run. Here is a fully green `gds` run from `sentry` on Nov 15, 2026 (run 43796948508), with its six artifacts:

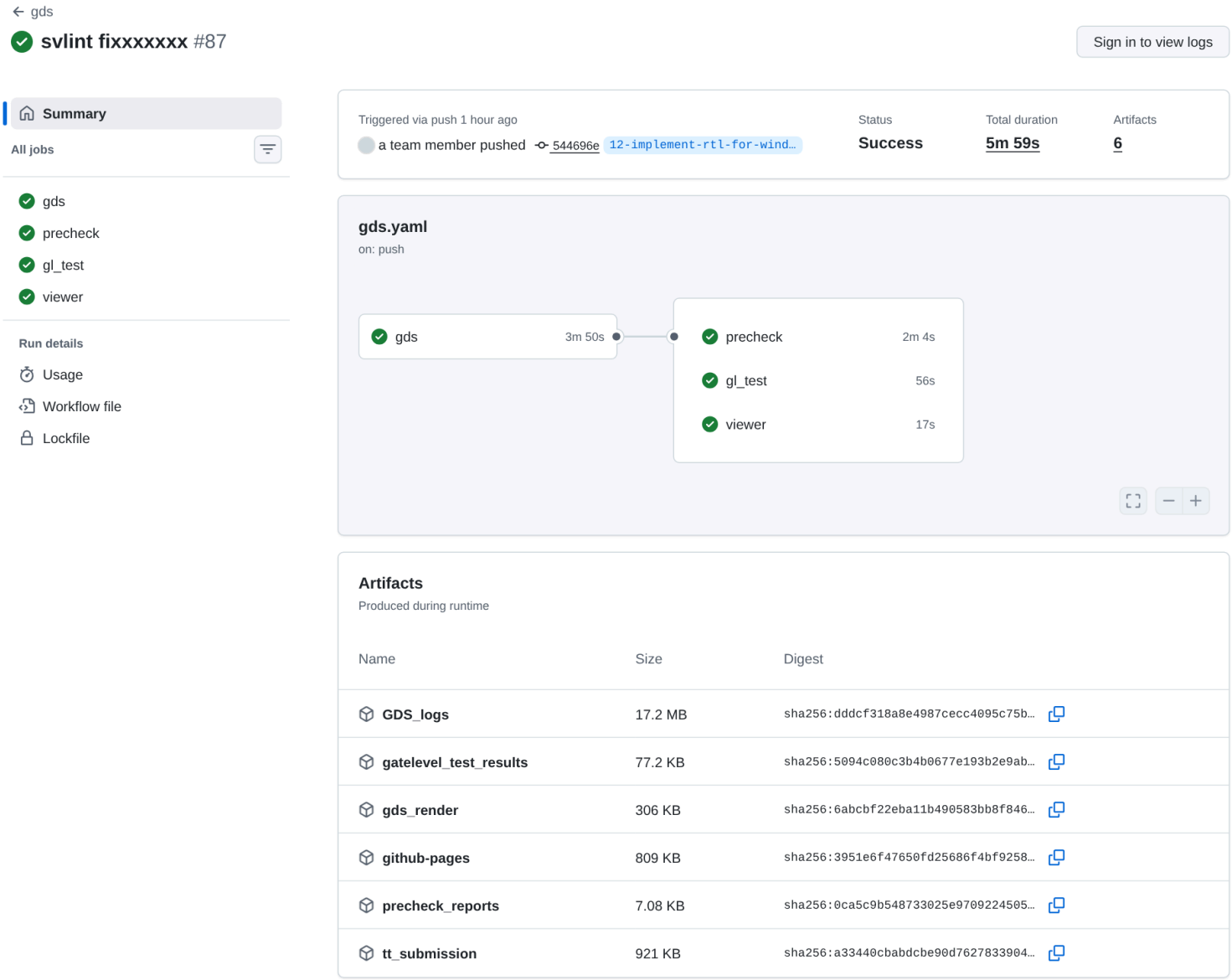



Figure 7.5: A fully green sentry gds run and its six artifacts. The person who pushed is shown as a team member.

`GDS_logs` holds your `src` folder and LibreLane's whole run folder, which is where the summary numbers come from. `tt_submission` is what the later jobs download to check and publish. `precheck_reports` and `gatelevel_test_results` are the evidence from those two jobs, and `gds_render` is a PNG picture of the layout. The `test` workflow uploads one more, `test-results`, with the waveform and the results file. Artifacts are deleted after 90 days by default, so download the ones you want to keep.

## Re-running a run

On a failed run you'll find a **Re-run jobs** menu, with a choice between re-running only the failed jobs and re-running all of them. A re-run uses the same commit as the original run ("the same `GITHUB_SHA`"), and you can re-run "up to 30 days after its initial run" (re-running workflows). So a re-run can't pick up a fix you've committed since; for that you push, or start a new run with the **Run workflow** button. A re-run is for failures that had nothing to do with your files, such as a fault on GitHub's side or a repository setting you've since changed.

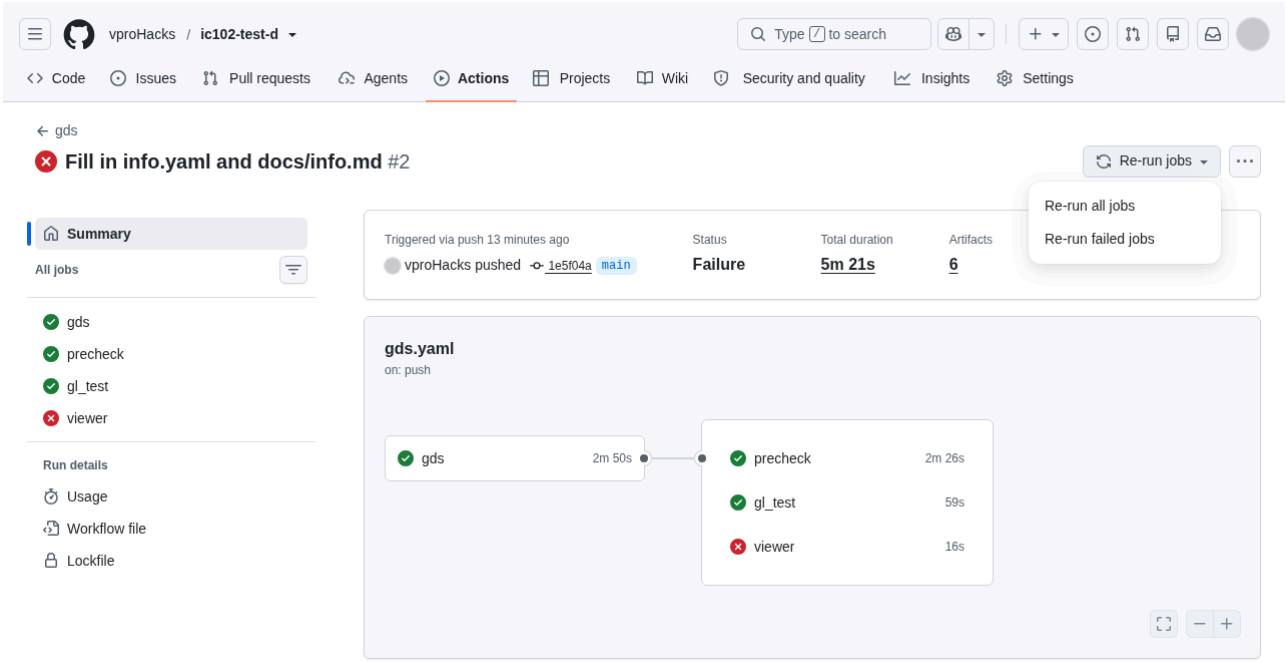



Figure 7.6: The Re-run jobs menu on a failed run.

There's a trap here for the `viewer` job, and `sentry` fell into it on Nov 6, 2026 (run 42309512218). The first attempt's `viewer` job failed with a fault on GitHub's side, and the message itself suggested a re-run:

```
##[error]Error: Failed to create deployment (status: 502) with build version 7d7cb9aff4081ae352fe3bc950122d12792039b6. Request ID 9029:3EA97E:8E9672:1CE8F8A:6AEE5E32 Server error, is githubstatus.com reporting a Pages outage? Please re-run the deployment at a later time.
```

The `viewer` job was re-run three more times in the next ten minutes, and each attempt failed for a new reason, with a count that went up by one each time:

```
##[error]Error: Multiple artifacts named "github-pages" were unexpectedly found for this workflow run. Artifact count is 2.
```

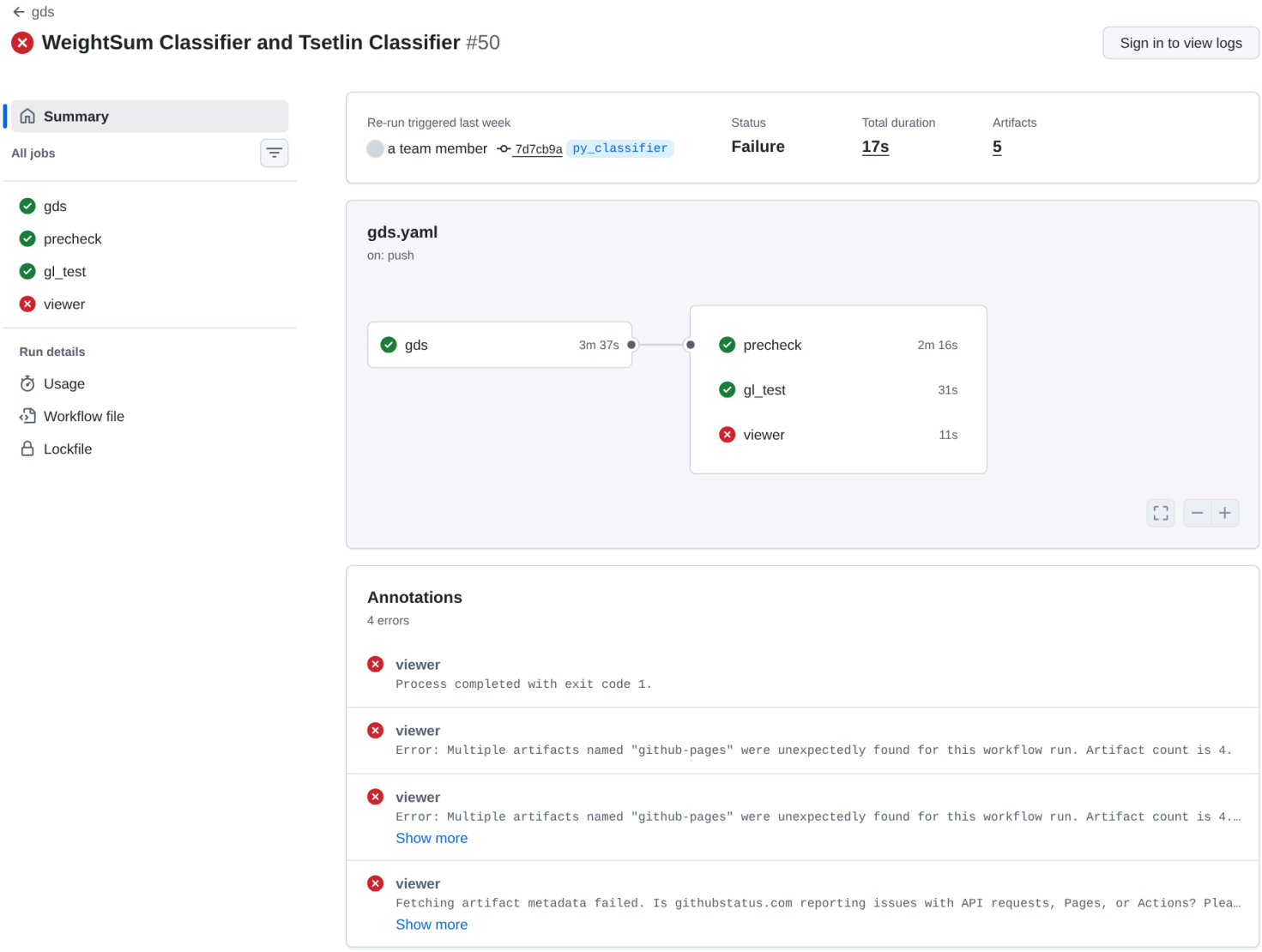



Figure 7.7: The sentry run after its fourth attempt at the viewer job. The person who re-ran it is shown as a team member.

Every attempt uploads a Pages artifact called `github-pages`, and a re-run keeps the artifacts of the attempts before it, so from the second attempt on there were two or more with the same name, and the deploy step failed instead of picking one. Tiny Tapeout fixed this in their action on Oct 28, 2026 (commit `16c727e`, "use a unique pages artifact name per run attempt"). The tags made since then, `ttihp27a` among them, have the fix, but `ttsky26d`, the tag in the current SKY130 templates, was made on Sep 18 and doesn't. The old artifact doesn't last long, though: the Pages artifact is uploaded with `retention-days: 1`, so a day later only the new one is left. That's why `protocol-emulator` could re-run the `viewer` job of a two-week-old run on Nov 15 (run 41355878948) and see it pass.

So in a repository on `ttsky26d`, when the `viewer` job fails and you've fixed the cause, don't re-run it straight away. Start a new run with **Run workflow** on the `gds` workflow (or push a commit), which begins with no old artifacts at all.

## GitHub Pages and the viewer

The `viewer` job downloads `tt_submission` and `gds_render`, writes a small web page that opens Tiny Tapeout's 3D viewer on your layout, and deploys that page with GitHub Pages. For that to work, the repository's Pages source has to be set to **GitHub Actions**, under Settings › Pages › Build and deploy-

ment › Source (publishing source). A repository made from a template starts with Pages off, as Templates, forks and clones explains, so the first `gds` run in a new repository can build and check a layout successfully and still end red. That was the state of `protocol-emulator`'s run on Nov 15, 2026 (run 43757724629):

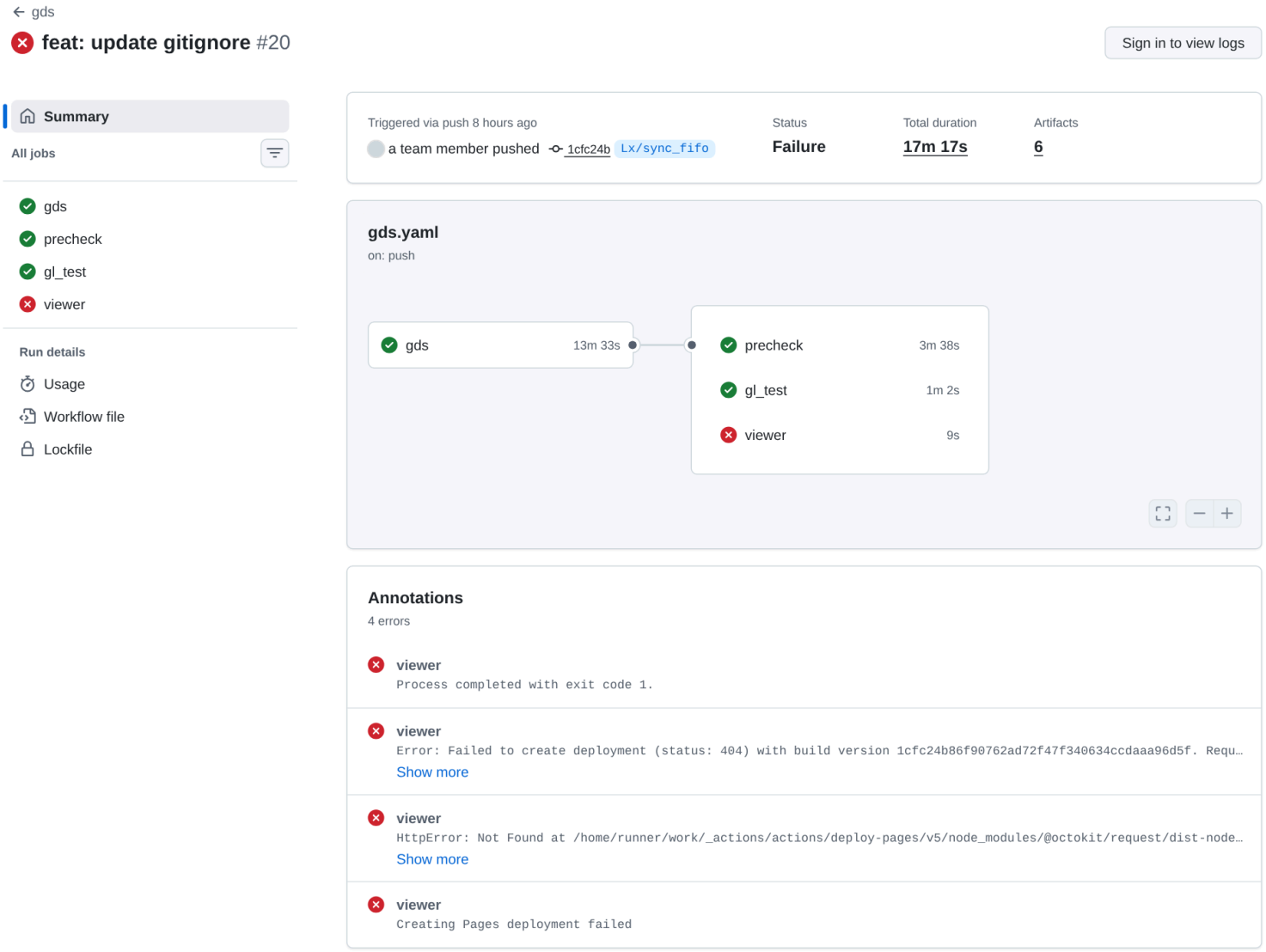



Figure 7.8: A protocol-emulator gds run where only the viewer job failed. The person who pushed is shown as a team member.

The `viewer` log says what's wrong in plain words, at the end of its third error line:

```
##[error]Creating Pages deployment failed
##[error]HttpError: Not Found
```

```
##[error]Error: Failed to create deployment (status: 404)
with build version 1cfc24b86f90762ad72f47f340634ccdaaa96d5f. Request ID
5018:223A0C:13B2E1A:3F3E2A0:6AFA17C5 Ensure GitHub Pages has been enabled: https://
github.com/uoftasic/protocol-emulator/settings/pages
```

followed by Tiny Tapeout's own pointer to the fix, from the step that runs on failure:

```
Failed to deploy to GitHub Pages, please follow the link to troubleshoot: https://
tinytapeout.com/faq/#my-github-action-is-failing-on-the-pages-part
```

That FAQ entry says to change the Pages “Source from Deploy from a branch to GitHub Actions”.

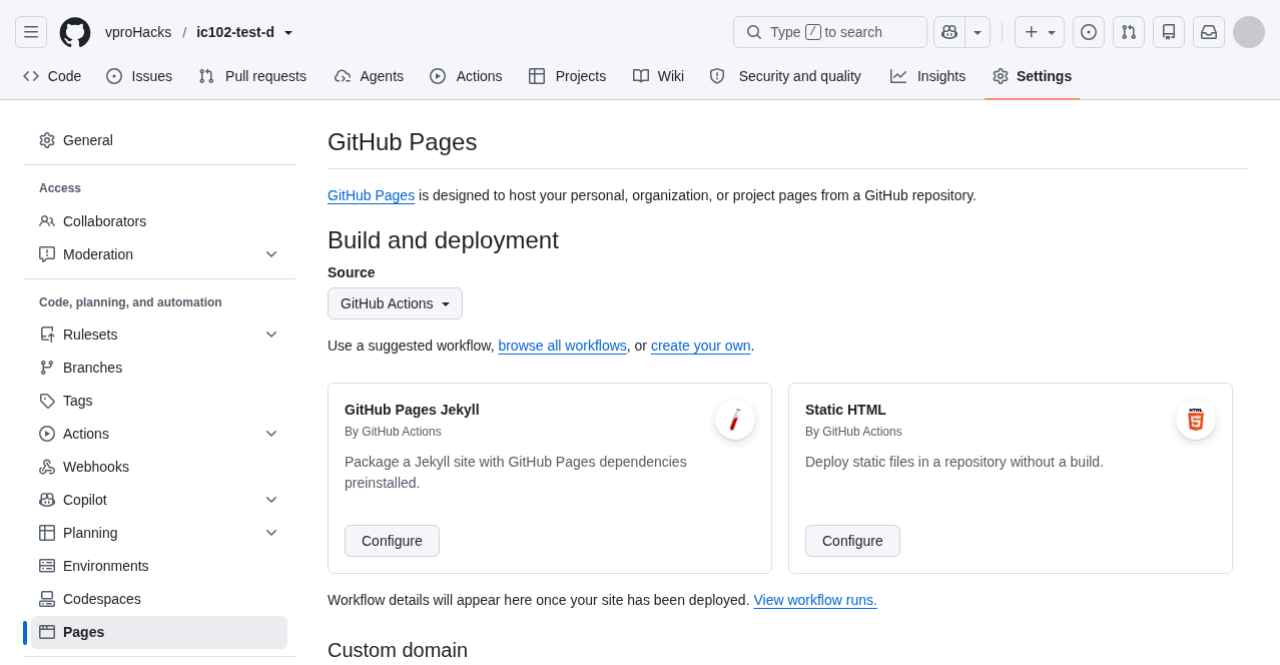



Figure 7.9: Settings › Pages, with the source set to GitHub Actions.

On GitHub’s Free plan, Pages works only in public repositories (custom workflows), which is one more reason the course has you make yours public. In this course a workflow counts as passing only when every one of its jobs is green, `viewer` included, so turning Pages on, and then starting a new run as the section above explains, is part of getting a repository working. You’ll do it first in Lab 01, and again for every repository you make in this course.

## Check yourself

1. In the `hart-1` run above, `precheck`, `gl_test` and `viewer` have grey icons. Why didn’t they run, and what line in `gds.yaml` decides that?

*Answer: A1, in Answers at the end.*

2. A `gds` log shows an `info.yaml` error in “Fetch verilog and build config” and then a `cat: ... No such file or directory` error in “Linter output”. Which one do you fix?

*Answer: A2, in Answers at the end.*

3. Your repository uses the `ttsky26d` tag. The `viewer` job failed with a 404, and five minutes ago you set the Pages source to GitHub Actions. Do you press “Re-run failed jobs” or start a new run, and why?

*Answer: A3, in Answers at the end.*

4. Why does the template’s “Run tests” step run `python -m cocotb_tools.check_results results.xml` after `make`, and does it still need to with the template’s cocotb 2.1.0?

*Answer: A4, in Answers at the end.*

Next: the page on your template’s workflows, which comes next in this row of the Course path.

## Answers

**A1. Check yourself**

Each of them has `needs: gds`, which makes a job wait for `gds` and run only if it succeeded. The `gds` job failed, so all three were skipped.

**A2. Check yourself**

The `info.yaml` error, because it comes first and stopped the build. “Linter output” runs with `if: always()`, so it ran anyway and failed because there was no linter log to show. It goes away once the build runs.

**A3. Check yourself**

Start a new run, with Run workflow on the `gds` workflow or by pushing. A re-run keeps the first attempt's `github-pages` artifact, which lasts a day, and the viewer action at `ttsky26d` uses the same artifact name every attempt, so the re-run would fail with “Multiple artifacts named”github-pages“”. A new run starts with no artifacts.

**A4. Check yourself**

GitHub decides whether a step passed from the exit status of its commands. The line was added for cocotb 1.x, whose `make` exited with 0 even when a test failed, so reading the results file and exiting non-zero was what turned a failing test into a red step. With cocotb 2.x, `make` runs the same check itself and exits non-zero, so the step already stops at `make`. The extra line now runs only after `make` has passed, and it does no harm.

# 8. The Tiny Tapeout workflows

This page answers: *what does each of the digital template's six jobs do, how long does it take, what does it leave behind, and what does it look like when it fails?*

Reading a workflow file showed how to read any workflow, using the template's `gds.yaml` as the example. This page goes through the digital template's workflows job by job, so that when one goes red you already know what it was checking. Everything here is from `TinyTapeout/ttsky-verilog-template` at `a42e666` and from Tiny Tapeout's actions at the tag its workflows name, `ttsky26d`, which pointed at commit `395eedb` when we read it on Nov 15, 2026. The times are from runs in the team's repositories.

## One push, six jobs

A push to your repository starts three workflows. Between them they run six jobs, each on its own fresh runner:

| Workflow | Job | What it checks | Time in the team's runs | What it uploads |
|---|---|---|---|---|
| `docs` | `docs` | `info.yaml` and `docs/info.md`, then builds the datasheet page | 30 s (`hart-1`) to 53 s (`sentry`) | `PDF` |
| `test` | `test` | Your cocotb test, on your RTL | 20 to 46 s | `test-results` |
| `gds` | `gds` | Builds the layout with LibreLane | 3 min 50 s (`sentry`, 2x2) to 28 min (`protocol-emulator`, 6x2 on IHP's process) | `GDS_logs`, `tt_submission`, `gds_render` |
| `gds` | `precheck` | Tiny Tapeout's checks on the layout | 91 to 315 s | `precheck_reports` |
| `gds` | `gl_test` | Your cocotb test again, on the gate-level netlist | 35 to 62 s | `gatelevel_test_results` |
| `gds` | `viewer` | Publishes a page that shows your layout in 3D | 6 to 25 s | `github-pages` |

The last three jobs each have `needs: gds`, so they wait for it and are skipped if it fails. The fourth workflow file, `fpga.yaml`, is switched off: its trigger lists only a branch called `none`.

## docs

The `docs` job has a single step of its own, "Build docs", which runs Tiny Tapeout's docs action. Inside it, the section "Check info.yaml and docs/info.md" does the checking. It runs the same validator the `gds` job uses on `info.yaml`, and then a check of `docs/info.md` that looks for the template's placeholder sentences. If either finds a problem, the job prints a list under `## Errors in info.yaml` or `## Errors in docs/info.md`, copies it into the run's summary page, and stops. Otherwise it installs Pandoc and Typst, builds your datasheet page as a PDF and uploads it as the `PDF` artifact.

Tiny Tapeout's own guide says that "projects with a failing docs action will still be included onto the shuttle", though without their datasheet page. In this course a red `docs` counts as red, because the errors it finds are usually also errors in the files every other tool reads.

You'll see its failures in Lab 01, where the template's empty fields fail it on the first commit, and in breakages 1, 2 and 5 of Lab 02.

## test

The `test` job is the only one written out in your repository rather than in Tiny Tapeout's actions, so you can read every step in `.github/workflows/test.yaml`. It installs Icarus Verilog from Ubuntu's packages (12.0 on the `ubuntu-24.04` runner), installs Python 3.11 and the packages in your `test/requirements.txt` (the template pins `cocotb==2.1.0` and `pytest==8.4.2`), and then runs the "Run tests" step:

```
      - name: Run tests
        run: |
          cd test
          make clean
          make
          # make will return success even if the tests fail, so check results.xml for
failed tests
          python -m cocotb_tools.check_results results.xml
```

As Reading a workflow file says, the comment was true of cocotb 1.x. With cocotb 2.x, `make` reads `results.xml` itself at the end of every simulation, through the same `check_results` module; you can see the line that does it in cocotb's `Makefile.icarus`, in the recipe for `results.xml`. So with the template's cocotb 2.1.0 a failing test stops the step at `make`, and the last line only runs when `make` has passed.

After "Run tests", two steps run even when it failed: "Test Summary" turns `results.xml` into a table on the run's summary page, and the last step uploads `tb.fst` and `results.xml` as the `test-results` artifact. When the simulation never started, both complain that the files are missing, as Reading a workflow file shows for `protocol-emulator`. Both complaints are about files that the failed step never wrote, so look further up the log for the cause.

A green `test` job means your test passed on your RTL with the runner's tools. It doesn't mean the test checks much: the template's test drives one pair of inputs and checks one output, and it stays green on a design that only gets that one case right.

## gds

### The gds job

"Build GDS" is one line in `gds.yaml` and 18 steps in Tiny Tapeout's action. These are the ones you need to know:

- **"Fetch verilog and build config"** runs `tt_tool.py --create-user-config`. It reads `info.yaml`, checks that every file in `source_files` exists in `src/`, and turns your tile size and top module into LibreLane's configuration. An `info.yaml` problem stops the job here, within the first minute, with a line that starts `Error loading .../info.yaml:`.
- **"Make GDS with LibreLane"** runs the whole flow: lint, synthesis with Yosys, floorplan, placement, clock tree, routing, and LibreLane's own DRC, LVS and antenna checks. At the `ttsky26d` tag the action installs LibreLane 3.0.14 and runs it in LibreLane's Docker image. A design that doesn't fit its tiles, or that has an error Yosys or the linter catches, stops here.
- **"Linter output"**, **"Yosys warnings"**, **"Routing summary"** and **"Cell usage summary"** write tables to the run's summary page, and **"Publish build logs"** uploads the whole run folder as `GDS_logs`. "Linter output" and "Publish build logs" run even after an earlier step has failed, and when the build stopped early, "Linter output" prints an error of its own about a log file that doesn't exist.
- **"Prepare tt_submission artifact"** collects the GDS, the LEF, the gate-level netlist and a `pdk.json` naming the PDK commit, which is what the next three jobs download.

The routing summary's two numbers, utilisation and wire length, come from two files in `GDS_logs`, and in Lab 03 you find them there. In `sentry`'s green run on Nov 16, the run folder also held a manufacturability report that said `Passed` for antenna, LVS and DRC, so the layout had been checked once before `precheck` started.

### precheck

`precheck` downloads `tt_submission`, installs the PDK at the commit Tiny Tapeout pins for it, builds its own Magic and KLayout with Nix, and runs `precheck.py` from Tiny Tapeout's support tools. It passed in the three runs whose logs we read (two of `protocol-emulator`'s, and `sentry`'s 43796948508), which is what you'd expect when LibreLane has already run DRC on the same layout. It matters far more to the analog version of this course, where you draw the layout yourself. Its reports are uploaded as `precheck_reports`.

### gl_test

`gl_test` checks out your repository, copies the gate-level netlist from `tt_submission` into `test/gate_level_netlist.v`, installs Tiny Tapeout's own build of Icarus Verilog 13.0 and the packages in your `test/requirements.txt`, and runs your test again with `GATES=yes make`. That switch, in `test/Makefile`, compiles the netlist and the sky130 standard-cell models instead of your RTL. The step ends with its own check:

```
test -f results.xml
! grep -E '<(failure|error)' results.xml
```

So `gl_test` fails for the same reasons `test` does, and for one more: a design that's correct in RTL simulation and wrong once it's made of real cells. RTL and gate-level tests explains how that happens. Its waveform and results go in `gatelevel_test_results`.

### viewer

`viewer` doesn't check anything about your design. It downloads `tt_submission` and `gds_render`, writes a small page that opens Tiny Tapeout's 3D viewer on your layout, and deploys it with GitHub Pages. It fails with a 404 until Pages is turned on, which is the first thing you fix in Lab 01. On the `ttsky26d` tag a re-run of a failed `viewer` fails again, for a reason Reading a workflow file explains, so after turning Pages on you start a new run instead, with **Run workflow** or a push.

## What green means in this course

Tiny Tapeout says that "Only the GDS action needs to be green to submit your design", and that for precheck "All the checks need to be green to submit to the chip". It doesn't say whether a `gds` workflow that fails only in `viewer` counts as failing. This course doesn't rely on the distinction: a workflow counts as green when every one of its jobs is, `viewer` included, and a lab's "done" means all six jobs passed on a commit of yours.

## Tags and support tools that move

Two things in these workflows can change without any commit in your repository. A tag such as `ttsky26d` is a name for one commit of Tiny Tapeout's actions, and Tiny Tapeout moves its tags with a force-push when it updates a release. And the actions fetch Tiny Tapeout's support tools, which hold the docs check, `precheck.py` and `tt_tool.py`, from their `main` branch on every run, unless a workflow asks for a particular commit. Each job's log records what it got, in a line near the top: `Download action repository 'TinyTapeout/tt-gds-action@ttsky26d' (SHA:...)`. When a run that passed last month fails today with no change on your side, compare that line between the two logs first.

## Check yourself

1. You push a commit that empties `title` in `info.yaml`. Which of the six jobs run, and which pass?

*Answer: A1, in Answers at the end.*

2. `test` is green and `gl_test` is red, on the same commit. Name two different causes.

*Answer: A2, in Answers at the end.*

3. With cocotb 2.1.0, does "Run tests" ever reach its `check_results` line when a test fails?

*Answer: A3, in Answers at the end.*

Next: Lab 02, where you break your repository in five ways and read what these jobs say.

# Answers

**A1. Check yourself**

`docs` and `gds` run and fail, both in their first minute. `test` runs and passes, because the testbench doesn't read `info.yaml`. `precheck`, `gl_test` and `viewer` don't run: they need `gds`, which failed.

**A2. Check yourself**

The design behaves differently once it's made of cells, for example because it relies on an `initial` statement where it needs a reset. Or the testbench depends on something the netlist doesn't have, such as a signal inside the design that synthesis renamed or removed.

**A3. Check yourself**

No. `make` runs the same check at the end of the simulation and stops with an error, so the step fails at `make`. The extra line runs only when `make` passed, and then it finds no failed tests.

# 9. Lab 02: Read the red X

You'll break your own repository five times, each time in a way that one of the team's repositories broke or that one of Tiny Tapeout's checks exists to catch. A course script makes each breakage as a single commit. For each one, you predict what will fail, push, find the job, the step and the first error line that explain the red X, and write them in a file called `DIAGNOSIS.md` before you fix anything. Each fix is a commit of your own, and GitHub has to run it green.

Package: `labs/lab-02-read-the-red-x/`

## Before you start

- Read What a cloud runner is, Reading a workflow file and The Tiny Tapeout workflows.
- Finish Lab 01: every job green on a commit of yours, with Pages on. This lab starts from that state.
- Every command runs in a terminal on the workbench desktop, in your repository's folder. In each new terminal, run these first:

```
. /foss/designs/common/.designinit
L=/foss/designs/modules/ic102-d/labs/lab-02-read-the-red-x
cd /foss/designs/projects/ttsky-ic102-d
```

To read a job's log you have to be signed in to GitHub, even for your own public repository. A run's summary page is public, and GitHub shows the logs behind it only to people who are signed in.

This lab takes about 1 hour 15 minutes, most of it waiting for runs. A `docs` or `test` run finishes in about a minute. A `gds` run takes about 5 minutes (4 min 30 s to 5 min 30 s in the course's practice repository), and you can read ahead in this lab while it runs.

## Part 1: The diagnosis file

1. Make sure your clone is up to date with GitHub and has nothing uncommitted, because the breakage script refuses to run otherwise:

   ```
   git pull
   git status
   ```

   You should see `nothing to commit, working tree clean`.

2. Copy the diagnosis template to the top of your repository and look at the breakages:

   ```
   cp $L/DIAGNOSIS-template.md DIAGNOSIS.md
   python3 $L/break.py --list
   ```

   You should see the five commit messages the breakages will use:

   ```
   1  info.yaml: tidy up the project section
   2  docs: start info.md again from the template's layout
   3  Rename the top module so that it's unique on the shuttle
   4  Output the difference ui_in - uio_in instead of the sum
   5  info.yaml: run the design at 50 MHz
   ```

   They read like commits a teammate might push. A commit's message doesn't always say everything the commit did: `protocol-emulator`'s `b976406 docs: update info.yaml` also renamed a module.

3. Commit `DIAGNOSIS.md` and push it, so each entry you add later travels with its fix:

```
git add DIAGNOSIS.md
git commit -m "Add DIAGNOSIS.md for Lab 02"
git push
```

## Part 2: Breakage 1, step by step

Which job tells you first that `info.yaml` is wrong, and which line in its log says how?

You'll go through the same eight steps for every breakage, and this part takes you through them once, with breakage 1.

1. Make the breakage:

   ```
   python3 $L/break.py 1
   ```

   You should see `Committed breakage 1 as <hash>: info.yaml: tidy up the project section`, then `Nothing has been pushed.` Don't look at the diff yet.

2. Predict, and write it in the **Prediction** field of the breakage's entry: which workflows go red, and which job finishes red first? You know from the commit message that `info.yaml` changed, and from The Tiny Tapeout workflows which jobs read it.

3. Push, and open the Actions tab of your repository:

   ```
   git push
   ```

4. Find the first red job. Open its run, click the red job, and the failed step opens by itself. Inside a step that runs one of Tiny Tapeout's actions, the failure is in a section nested in it; read down from the top of the failed section to the first line that names a problem.

5. Write the **Job that failed first**, the **Step** (the workflow's step, and the action's section inside it if there is one) and the **First error line**, copied without its timestamp.

6. Only now, look at what changed (`git show`), fix it the way a teammate should have, and write the **Fix** in a sentence.

7. Commit the fix together with your updated `DIAGNOSIS.md`, push, and wait until every job is green again. Then fill in **Fix commit** with the fix's hash (`git log --oneline -1`) and commit that small change too. Don't push it on its own, though. It goes up with the next breakage, in one push, and GitHub runs the workflows only on the newest commit of a push. If you pushed it by itself, the two pushes, a minute apart, would start two `gds` runs whose `viewer` jobs deploy to Pages at about the same time, and the second one would fail. In the course's practice repository it failed with:

   ```
   ##[error]HttpError: Deployment request failed for
   eda299edadf9f73ed66d97f537dae4a0840d10d6 due to in progress deployment. Please
   cancel c70f08ac35a93511ba610109124c852b4108bc57 first or wait for it to complete.
   ```

8. Check your entry so far, without asking GitHub yet:

   ```
   python3 $L/check.py --offline
   ```

   You should see `entry filled in; fix <hash> comes after breakage <hash>` under Breakage 1.

What should breakage 1's entry say? Write yours before you open this.

*Answer: A1, in Answers at the end.*

You should see: Breakage 1's entry complete, and every job green on its fix commit.

## Part 3: Breakage 2

Can a section be "missing" from a file you can see it in?

1. Run `python3 $L/break.py 2` and follow the routine from Part 2.

*Answer: A2, in Answers at the end.*

You should see: `docs` red with the two "Missing" lines, and green again after your fix.

## Part 4: Breakage 3

What does a test do when the design it tests has a new name?

1. Run `python3 $L/break.py 3` and follow the routine. This one fails in two workflows a few minutes apart, so wait for both before you write the entry.

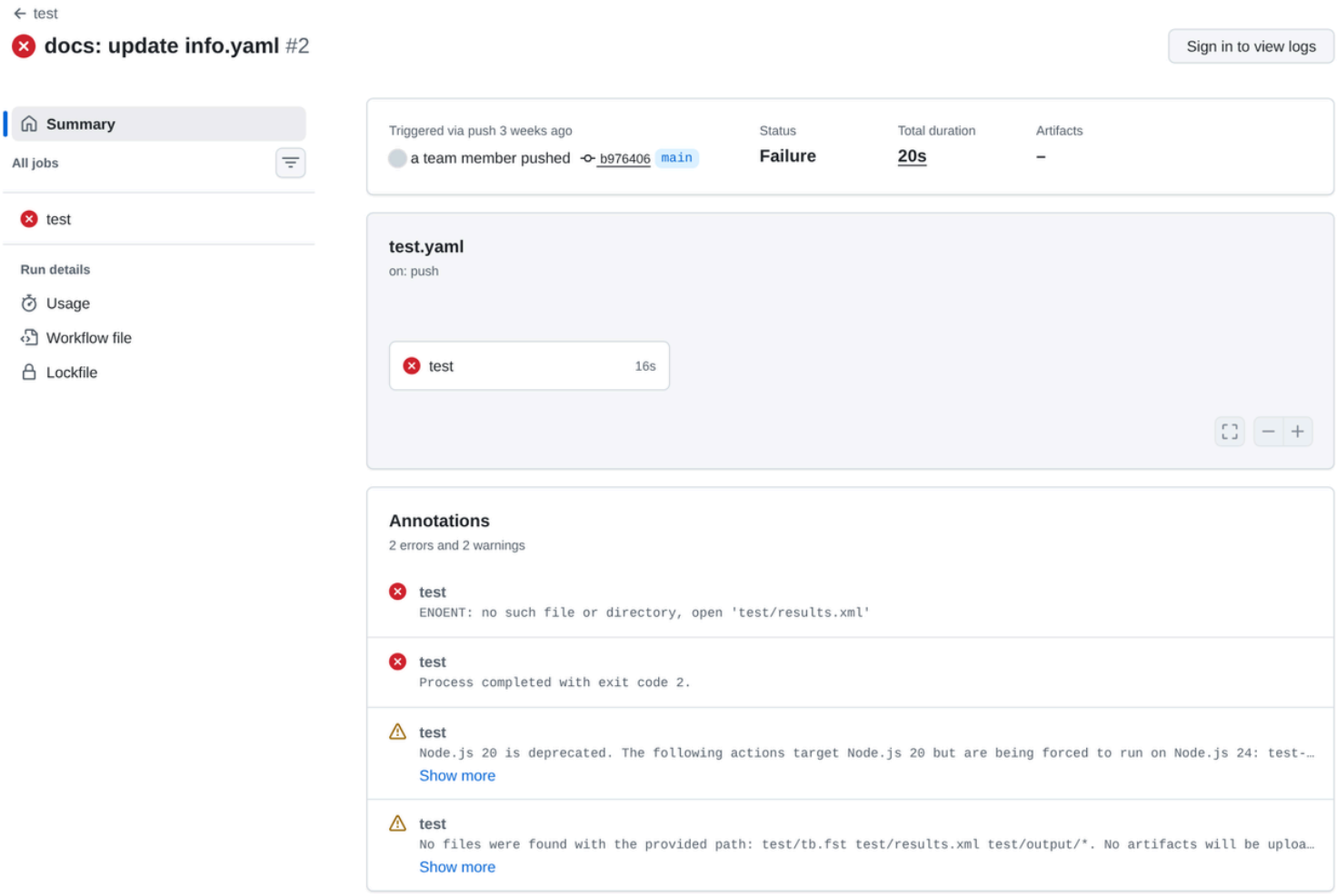



Figure 9.1: The run of protocol-emulator's test workflow after the same mistake. The annotations show what went wrong afterwards; the cause is only in the log.

That's `protocol-emulator`'s run for the same mistake, seen without signing in. None of its annotations says what's wrong.

*Answer: A3, in Answers at the end.*

You should see: `test` and `gl_test` red with `Unknown module type`, and green after the fix.

## Part 5: Breakage 4

When a test fails because the design changed on purpose, what's the right fix?

1. Run `python3 $L/break.py 4` and follow the routine, but stop before you fix it and answer the question in step 2 first.

*Answer: the error: A4, in Answers at the end.*

2. The commit says the design should output the difference now. `protocol-emulator` once had a failing assertion and commented it out (`a5f1475`, "chore: temporarily remove assertion from test.py"), and its test has passed ever since while checking nothing. What should you do instead?

   *Answer: A5, in Answers at the end.*

3. Make that fix and run the test locally before you push, then finish the routine:

```
make -C /foss/designs/modules/ic102-d/labs/common/tt-tools test PROJECT="$PWD"
```

You should see: your strengthened test passing locally, and `test` and `gds` green on GitHub.

## Part 6: Breakage 5

What does Tiny Tapeout's validator accept for `clock_hz`?

1. Run `python3 $L/break.py 5` and follow the routine.

   *Answer: A6, in Answers at the end.*

2. Fill in the last section of `DIAGNOSIS.md`, "A value no workflow catches": a value you could put in `info.yaml` that every workflow accepts and that is still wrong, and why the workflows accept it.

   *Answer: A7, in Answers at the end.*

3. There's no next breakage to carry your last `DIAGNOSIS.md` changes, so commit them and push them on their own.

You should see: five entries, each followed by a green fix, and the extra section filled in.

## Finishing

You're done when `python3 $L/check.py` prints `5 of 5 breakages diagnosed and fixed` and `VERDICT: PASS`: every entry in `DIAGNOSIS.md` is filled in, each fix commit comes after its breakage, and GitHub ran all six jobs green on every fix. On the course's practice repository it printed:

```
Breakage 1
  entry filled in; fix bf7a3fc comes after breakage b8352cc
  GitHub: all six jobs passed on the fix

Breakage 2
  entry filled in; fix 42c5311 comes after breakage eda299e
  GitHub: all six jobs passed on the fix

Breakage 3
  entry filled in; fix c81b926 comes after breakage d5a5a36
  GitHub: all six jobs passed on the fix

Breakage 4
  entry filled in; fix 9126c83 comes after breakage 757853c
  GitHub: all six jobs passed on the fix

Breakage 5
  entry filled in; fix e9f320e comes after breakage 0e0b02d
  GitHub: all six jobs passed on the fix

A value no workflow catches: filled in

5 of 5 breakages diagnosed and fixed
VERDICT: PASS
```

Questions:

1. In breakages 1 and 5, `docs` and `gds` both failed. Why does Tiny Tapeout check `info.yaml` in two workflows?

   *Answer: A8, in Answers at the end.*

2. Which breakage would have gone unnoticed by a teammate who only ever runs the test locally?

   *Answer: A9, in Answers at the end.*

Submit: the link to your repository's `DIAGNOSIS.md` on GitHub.

Next on the Course path: read RTL and gate-level tests and Your tools and the runner's tools, then do Lab 03: Make it yours.

# Answers

**A1. Part 2: Breakage 1, step by step**

Two workflows go red, `docs` and `gds`, and `docs` usually finishes first: in `hart-1`'s runs with the same mistake, the `docs` job took 30 s and the `gds` job 54 s, and in the course's practice repository 38 s and 43 s. `test` stays green, because the testbench never reads `info.yaml`.

In `docs`, the step is "Build docs", section "Check info.yaml and docs/info.md", and the error is the line under the heading:

```
## Errors in info.yaml
- Project title cannot be empty
```

That's what the practice repository's `docs` job printed for this breakage on Nov 16. When we ran tt-support-tools' docs check in the course image, on a broken copy of the template and at `9d13ff3` (the commit the runner used that day), it printed the same text. `hart-1`'s docs log (run 43752267314) has the same line in a longer list, because its title, author, description and pinout were all empty.

In `gds`, the step is "Build GDS", section "Fetch verilog and build config". In `hart-1`'s run (43752267356) it reads:

```
2026-11-15 19:25:32,583 - project     - ERROR     - Error loading /home/runner/work/
hart-1/hart-1/info.yaml: Project title cannot be empty, Project author cannot be empty,
Project description cannot be empty
```

Yours names your repository and lists only the title. The `cat: ... No such file or directory` from "Linter output" further down is a consequence of that first error, as Reading a workflow file explains. The fix is to put the title back.

**A2. Part 3: Breakage 2**

Only `docs` fails, in the same section as breakage 1, with:

```
## Errors in docs/info.md
- Missing 'How it works' section in docs/info.md
- Missing 'How to test' section in docs/info.md
```

Those are the last two lines of `hart-1`'s docs log too. The script put the template's `docs/info.md` back, so both headings are there. The check looks for the template's placeholder sentences under them ("Explain how your project works"), and reports a section as missing for as long as its placeholder is still there. `gds` passes, because only the docs check reads `docs/info.md`.

The fix is to write the two sections again, or to undo the commit: `git revert HEAD` makes a new commit that undoes the last one and leaves the history intact.

**A3. Part 4: Breakage 3**

The script renamed the top module in `src/project.v` and in `info.yaml`, to `tt_um_<your username>_adder`, and left `test/tb.v` instantiating `tt_um_example`. `test` fails first, in "Run tests". This is `protocol-emulator`'s line for the same mistake (run 40123836544):

```
/home/runner/work/protocol-emulator/protocol-emulator/test/tb.v:27:  error:  Unknown
module type: tt_um_example
```

Yours names your repository and says `tb.v:31`, the line of the current template's `tb.v` that names the module. In the course's practice repository the `test` job printed `/home/runner/work/ic102-test-d/ic102-test-d/test/tb.v:31: error: Unknown module type: tt_um_example` after 20 s, and the course's `make test` printed the same line for `/foss/designs/projects/ttsky-ic102-d/test/tb.v`. Then `gds`

goes red too, but only in `gl_test`, which runs the same testbench on the netlist: in `protocol-emulator`'s run 41345900293, and in the practice repository's, the `gds` and `precheck` jobs passed and `gl_test` stopped at the same `Unknown module type` line. The `results.xml` and `tb.fst` messages after it, and in the annotations, are consequences: the simulation never started.

The fix is to change the one line in `test/tb.v` that names the module, rather than undo the rename, because the template asks for a top module with your username in it.

**A4. Part 5: Breakage 4 (Answer: the error)**

`test` fails in "Run tests", and later `gl_test` fails in the same way. The failure reads:

```
AssertionError: assert LogicArray('11110110', Range(7, 'downto', 0)) == 50
```

The practice repository's `test` and `gl_test` jobs printed that line, and so did the course's `make test`.

The template's test drives 20 and 30 and expects their sum, 50. The design now subtracts, and $20 - 30 = -10$, which is `11110110`, or 246, in eight bits.

**A5. Part 5: Breakage 4**

Change the test to expect the new behaviour, and make it stronger while you're there. Expect 246 for 20 − 30, and add a second case the other way round (30 − 20 = 10), so that a subtraction with its operands swapped can't pass. In our run that version passed with `test.test_project PASS 110000.00`. A fix that deletes the assertion turns the cross green and leaves you with a test that can't fail.

**A6. Part 6: Breakage 5**

`docs` and `gds` fail, in the same places as breakage 1. The docs check prints:

```
## Errors in info.yaml
- Invalid value for 'clock_hz' in 'project' section, must be an integer
```

and the `gds` job's "Fetch verilog and build config" section, in the practice repository:

```
2026-11-16 09:09:02,783 - project    - ERROR    - Error loading /home/runner/work/ic102-
test-d/ic102-test-d/info.yaml: Invalid value for 'clock_hz' in 'project' section, must
be an integer
```

There both jobs took 24 s and finished in the same second, so either can be the one that failed first. `50MHz` reads as text in YAML, and the field wants a whole number of hertz. The fix is `clock_hz: 50000000`, or the number your design really runs at.

**A7. Part 6: Breakage 5**

`clock_hz: 50` is one. It's an integer, so the validator accepts it, but it says the design runs at 50 Hz, a factor of a million below the 50 MHz the commit message meant. `protocol-emulator`'s pull request #2 points out the same value. Another is a pinout that calls `uio[0]` an input while the design drives it with `uio_oe[0] = 1`.

**A8. Finishing**

They use it for different things. `docs` builds the datasheet page from it, and `gds` needs the top module, the source files and the tile size to build the layout, so each of them reads it and stops if it can't. So you see the same error twice, and fixing it once fixes both.

**A9. Finishing**

Breakages 1, 2 and 5. The local test never reads `info.yaml` or `docs/info.md`; only the runner's `docs` and `gds` jobs do, and the course doesn't reproduce either of those on your workbench. That's one reason to count a check as passed only when it has passed on the runner.

# III 3. Your design


**10 RTL and gate-level tests . . . . . 67**
10 One test, two designs . . . . . 67
10 Where they disagree . . . . . 67
10 The tinyCGRA history . . . . . 68
10 Running each one yourself . . . . . 68
10 Check yourself . . . . . 68
10 Answers . . . . . 68

**11 Your tools and the runner's tools . . . 71**
11 The versions . . . . . 71
11 What `make test` reproduces . . . . . 71
11 Where Icarus 12.0 differs . . . . . 72
11 Your `requirements.txt` . . . . . 72
11 What isn't reproduced . . . . . 72
11 When the versions change . . . . . 73
11 Check yourself . . . . . 73
11 Answers . . . . . 73

**12 Lab 03: Make it yours . . . . . 75**
12 Before you start . . . . . 75
12 Part 1: Put the counter in . . . . . 75
12 Part 2: A test that catches the bug . . . . . 76
12 Part 3: The bug on the runner . . . . . 77
12 Part 4: The numbers in the `gds` job . . . . . 78
12 Part 5: A counter with no reset (optional) . . . . . 80
12 Finishing . . . . . 80
12 Going further (optional) . . . . . 80
12 Answers . . . . . 81

# 10. RTL and gate-level tests

This page answers: *your cocotb test runs twice on every push, once in `test` and once in `gl_test`. What's different about the second run, and why can a design pass one and fail the other?*

Your Verilog describes what the design should do. The `gds` job turns it into a layout, and on the way it produces a **gate-level netlist**: a Verilog file that describes the same design as instances of SkyWater's standard cells (flip-flops, NAND gates, buffers and the like), wired together. The `test` job simulates your Verilog as you wrote it, which is called **RTL simulation**. The `gl_test` job simulates the netlist, which is **gate-level simulation**. The test is the same file both times. You'll see both runs on your own design in Lab 03.

## One test, two designs

The template's `test/Makefile` decides what to compile. Without any setting it compiles the files in `PROJECT_SOURCES` from `src/`, plus `test/tb.v`. With `GATES=yes`, which is how `gl_test` calls it, it compiles three other files instead of your sources: the netlist, which `gl_test` copies to `test/gate_level_netlist.v`, and two files of models for the sky130 cells from the PDK. It also defines a few macros for the compile:

| Macro | What it does |
|---|---|
| `GL_TEST` | Read by the template's `tb.v`, which then connects two power wires, `VPWR` (1) and `VGND` (0), to the design |
| `USE_POWER_PINS` | Gives every cell model its power pins, so the netlist's power connections have somewhere to go |
| `FUNCTIONAL` | Uses the cells' logic-only models, without the detailed timing of the real cells |
| `UNIT_DELAY=#1` | Gives every cell a delay of 1 ns, so a change ripples through the logic in steps rather than all at once |

The testbench in `tb.v` stays the same, and so does `test.py`, so one test has to work on both designs, and that puts two limits on how you write it.

**Use only the pins.** In RTL simulation, cocotb can reach inside your design and read, say, `dut.user_project.count`. In the netlist, synthesis has flattened the design into cells and renamed or removed most internal signals, so a test that reads one may find nothing there at gate level. A test that drives and reads only `ui_in`, `uo_out`, `uio_in`, `uio_out`, `uio_oe`, `clk` and `rst_n` reads the same pins in both.

**Sample away from the clock edge.** With unit delays, a flip-flop's output changes about 1 ns after the rising edge, and the logic after it settles a few nanoseconds later. A test that changes inputs and reads outputs on the falling edge, half a clock period away, gives the same answers in both simulations. Lab 03′s starter test does that, and so does the review sandbox's in Lab 04.

## Where they disagree

Tiny Tapeout's documentation warns that on a real chip "flops will have a random initial value at power on", and therefore that "Initial statements are not allowed, so you must use an explicit reset." An `initial` statement in your Verilog sets a register when the simulation starts, so in RTL simulation the register starts at the value you gave it. Synthesis can't build that into a flip-flop, though, so in the netlist the register starts with no defined value, which a simulator shows as `X`, and anything computed from an `X` stays `X`.

We tried this on Lab 03's counter. With its reset replaced by `initial count = 8'd0;`, all three of its tests passed in RTL simulation with the test job's tools. On a netlist that Yosys made from the same file with the sky130 cells (not LibreLane's netlist, but built from the same cell library), Icarus 13.0 printed:

```
AssertionError: after reset uo_out is XXXXXXXX, expected 0
AssertionError: after 10 cycles with EN high uo_out is XXXXXXXX, expected 10
```

The third test, of `LOAD`, still passed, because loading a value sets every bit at once. A failure that shows up in some tests and not others, only at gate level, is often this.

The two simulations also treat an unknown value differently, which is harder to spot. In RTL simulation, an `if` whose condition is `X` takes the `else` branch, so a design can behave as if an unknown signal were 0 without any sign of it in the results. Gates don't do that; an `X` on a gate's input usually gives `X` on its output. RTL simulation is the optimistic one, so a design that only works because of that optimism passes `test` and fails `gl_test`.

`gl_test` doesn't check timing, though. Every cell has the same made-up delay, and nothing compares the delays with the clock period. LibreLane's timing analysis in the `gds` job does that, against the clock period in `src/config.json`.

## The tinyCGRA history

`ttsky-verilog-tinyCGRA` shows how this goes wrong in a team. In May 2026, two of its `gds` runs built the layout, passed precheck and then failed at `gl_test`. The logs are gone, but the commit messages from those days survive: "commented config done (test)", "comment assert input in load_input()", "comment all user_project asserts", "commented asserts, no functional testing". The gate-level failures went away because the test stopped checking anything. The pull request that was finally merged includes a commit with a reason, "removed config_done assert (unreliable in GLS)", which at least says what was removed and why. When `gl_test` fails and `test` passes, ask what's different about the netlist. The answer is usually in the design: a missing reset, a signal read before it's defined, or a test that reaches inside.

## Running each one yourself

`make test` in the course's `labs/common/tt-tools` runs the RTL test with the test job's tools, as you'll do in Lab 03. Running the gate-level test needs the netlist, which only the `gds` job makes: the tools package can run `gl_test`'s steps on a netlist you download from a run's `tt_submission` artifact (`make gl-test`, in the package's README), with Icarus 13.0 and the sky130 cells at the PDK commit the runner used. It's optional, and most of the time reading `gl_test`'s log is quicker.

## Check yourself

1. A test reads `dut.user_project.state` to check a state machine. Which job do you expect it to pass in, and which to fail in?

*Answer: A1, in Answers at the end.*

2. Why does `tb.v` connect `VPWR` and `VGND` only when `GL_TEST` is defined?

*Answer: A2, in Answers at the end.*

3. A design passes both `test` and `gl_test`. Does that show it will run at the clock rate in `info.yaml`?

*Answer: A3, in Answers at the end.*

Next: Your tools and the runner's tools.

## Answers

**A1. Check yourself**

It can pass in `test`, where the RTL has a signal called `state`, and fail in `gl_test`, where synthesis has turned the state register into cells whose names it chose. Check the state through the pins it drives instead.

**A2. Check yourself**

Your RTL module has no power pins, because power isn't part of the logic you describe. The netlist does, because every cell needs power, and LibreLane adds `VPWR` and `VGND` ports to the top module. The testbench has to connect them in one case and mustn't in the other.

**A3. Check yourself**

No. Neither simulation models the real delays. The `gds` job's timing analysis, against `CLOCK_PERIOD` in `src/config.json`, is the check for that.

# 11. Your tools and the runner's tools

This page answers: *the workbench has its own Icarus and cocotb, and GitHub's runner installs others. Which versions does each job use, where do they differ, and what does the course do about it?*

What a cloud runner is explains why the runner's tools needn't match yours: every job installs its own, at the versions its workflow asks for, on a fresh machine. This page lists them for the digital template, and says what the course installs so that a test you run in the workbench means the same as the runner's. You install them in Lab 01, Part 1 and use them from Lab 03 on.

## The versions

We read these from Tiny Tapeout's action files at the `ttsky26d` tag (commit `395eedb`) and from runner logs of Nov 15 and 16, 2026. "Course" is what `make install` in `labs/common/tt-tools` puts next to the workbench's own tools; "Workbench" is what the image `hpretl/iic-osic-tools:2026.08` has.

| Job | Tool | Runner | Course | Workbench |
|---|---|---|---|---|
| `test` | Python | 3.11.18 | 3.11.16 | 3.12.3 |
| `test` | cocotb, pytest | 2.1.0, 8.4.2 | 2.1.0, 8.4.2 | 2.0.1, 9.1.1 |
| `test` | Icarus Verilog | 12.0 (Ubuntu's package) | 12.0, the same package | 14.0 |
| `gl_test` | Icarus Verilog | 13.0 (Tiny Tapeout's package) | 13.0, the same package | 14.0 |
| `gl_test` | sky130 standard cells | PDK commit `8afc8346` | the same, optional | PDK commit `026824c7` |
| `gds` | LibreLane | 3.0.14, in LibreLane's Docker image | not installed | 3.1.0.dev3 |
| `precheck` | Magic, KLayout | 8.3.568, 0.30.8, PDK commit `0536d02d` | not installed | 8.3.681, 0.30.11 |

The course's Python is 3.11.16 rather than the runner's 3.11.18, because 3.11.16 is the newest 3.11 that the workbench's installer offers. Both are 3.11, which is all the workflow asks for (`python-version: '3.11'`), and cocotb's package is built for 3.11 either way. Three different commits of the sky130 PDK are in play: the one LibreLane used to build your netlist, which `gl_test` uses too, the one precheck pins, and the workbench's. For a digital design only the first matters to you.

## What `make test` reproduces

`make test PROJECT=/foss/designs/projects/ttsky-ic102-d` runs the lines of the `test` job's "Run tests" step in your `test/` folder (`make clean`, `make`, and `python -m cocotb_tools.check_results results.xml`), with the runner's Python, cocotb and Icarus 12.0 ahead of the workbench's on the `PATH`. It prints the same two lines the runner's log starts the simulation with. On the runner, in `hart-1`'s test run of Nov 15:

```
Initialized cocotb v2.1.0 from /opt/hostedtoolcache/Python/3.11.18/x64/lib/python3.11/
site-packages/cocotb
Running on Icarus Verilog version 12.0 (stable)
```

and with `make test`, the second line is identical and the first names the course's folder instead. The simulated times in the summary table are the same in both: the template's test reports `100000.00` ns on the runner and with `make test`. The real-time column measures whichever computer ran it.

With cocotb 2.x, `make` fails by itself when a test fails, because cocotb runs `check_results` at the end of every simulation (in the `results.xml` recipe of its `Makefile.icarus`). The workflow's comment that `make` "will return success even if the tests fail" dates from cocotb 1.x. You'll see cocotb's `Error 1` and the course's `Error 2` in Lab 03. In Lab 05 your bisect script has to work around the fact that `make` fails the same way for a failing test and for a design that doesn't compile.

## Where Icarus 12.0 differs

On the designs we tried, the three Icarus versions agreed whenever all three accepted the code. The template's test and the test of `sentry` (at commit `544696e`) passed with the same simulated times under 12.0, 13.0 and 14.0, at RTL and on `sentry`'s gate-level netlist; so did Lab 03's counter.

They differ in what they accept. Icarus 12.0 rejects some SystemVerilog that 13.0 and 14.0 compile, so code you write and try in the workbench can stop the runner's `test` job at the compile step. When we changed the template's adder to compute its sum in a `for` loop that ends with `break`, which is legal SystemVerilog, Icarus 12.0 stopped with:

```
project.v:27: sorry: break statements not supported.
```

and `make` ended with `make: *** [Makefile:42: test] Error 2`. Icarus 13.0 and 14.0 printed `project.v:27: internal error: NetProc::nex_input not implemented`, carried on, and the test passed. Assigning a whole array at once (`q = '{8'd1, 8'd2, 8'd3, 8'd4};`) fails on 12.0 only, with `sorry: Assignment to an entire array or to an array slice is not yet supported.` Icarus says `sorry` when it recognises a construct and doesn't implement it, so a `sorry` from `make test` is one the runner's `test` job will print too.

This is the main reason `make test` uses 12.0 rather than the workbench's 14.0: the runner's `test` job will compile your code with 12.0, and only 12.0 can tell you in advance whether that compile will succeed. The course's `make test-icarus13` and `make test-workbench` run the same test with the other two versions, when you want to compare.

## Your `requirements.txt`

Both test jobs install their Python packages from your repository's `test/requirements.txt`, so the cocotb on the runner is whatever that file says, not what the template said. If you change the file, `make test` warns you before it runs. For `sentry`, which pins cocotb 2.0.1, it printed:

```
warning: the runner installs your test/requirements.txt, and it differs from this venv:
  cocotb: your test/requirements.txt pins 2.0.1, this venv has 2.1.0
  Results here may not match the runner's. See README.md, 'Your own requirements.txt'.
```

`sentry`'s `gl_test` log shows the runner following the file: `Initialized cocotb v2.0.1`, on Icarus 13.0. Leave the template's pins alone unless you have a reason, and if you do change them, the tools package's README says how to rebuild the course's copy to match.

## What isn't reproduced

- **The `gds` job.** It runs LibreLane 3.0.14 inside LibreLane's own Docker image, and Docker doesn't run inside the workbench. The workbench's own LibreLane, 3.1.0.dev3, can harden a design, with different versions of Yosys, OpenROAD and Magic, so its numbers won't match the runner's exactly. For the digital labs the runner is the place a layout gets built.
- **The `docs` job.** It checks two text files and builds a PDF; its log names the field or section that's wrong, and it takes under a minute on the runner.
- **`precheck`.** It checks the layout the `gds` job built, which only exists on the runner. The analog version of this course does run precheck locally, because there you draw the layout.
- **The runner's machine.** It's a fresh Ubuntu 24.04 virtual machine with 4 CPUs and 16 GB for a public repository; the workbench is whatever your laptop gives Docker.

## When the versions change

Tiny Tapeout updates its template for each shuttle, and can move a tag such as `ttsky26d` to a new commit. Your repository keeps the tag it was made with, but if the tag moves, the runner's tools can change under you. The course records every version it installs in one file, `labs/common/tt-tools/pins.env`, with the run or file each was read from and the steps for reading them again, so that when the runner changes, the course's copy can follow it.

## Check yourself

1. Your test passes with `make test-workbench` and fails to compile with `make test`, with a line that says `sorry`. What will the runner's `test` job do?

*Answer: A1, in Answers at the end.*

2. Which file decides the cocotb version the runner uses for your test?

*Answer: A2, in Answers at the end.*

3. Why doesn't the course install LibreLane 3.0.14?

*Answer: A3, in Answers at the end.*

Next: Lab 03, where you run your own test with these tools and on the runner.

## Answers

**A1. Check yourself**

Fail to compile in the same way, because it uses the same Icarus 12.0. Rewrite the construct in a form 12.0 accepts; the workbench's 14.0 accepting it doesn't help on the runner.

**A2. Check yourself**

Your repository's `test/requirements.txt`. Both `test` and `gl_test` install from it.

**A3. Check yourself**

The `gds` job runs it inside LibreLane's Docker image, and Docker doesn't run inside the workbench, which is itself a container. The layout is built on the runner instead, and Lab 03 reads the numbers from its artifact.

# 12. Lab 03: Make it yours

You'll replace the template's adder with an 8-bit counter that the course supplies. That means changing the five places that name a design, and getting all five to agree. Then you'll write a cocotb test that catches a one-line bug in the counter, run it with the test job's own tools in the workbench, and push the bug and its fix so the runner shows you the same failure and the same recovery. Last, you'll find where the `gds` job's routing numbers come from, in the files that the job uploads.

Package: `labs/lab-03-make-it-yours/`

## Before you start

- Read RTL and gate-level tests and Your tools and the runner's tools.
- Finish Lab 02, so your repository is green again.
- Every command runs in a terminal on the workbench desktop. In each new terminal, run these first:

```
. /foss/designs/common/.designinit
L=/foss/designs/modules/ic102-d/labs/lab-03-make-it-yours
TT=/foss/designs/modules/ic102-d/labs/common/tt-tools
cd /foss/designs/projects/ttsky-ic102-d
```

The counter is in `$L/counter.v`, and the comment at its top says what each pin does. You don't need to write Verilog in this lab, but read the file once; it's 50 lines.

This lab takes about 1 hour 30 minutes, including three `gds` runs of about 5 minutes each, plus 20 minutes for the optional part.

## Part 1: Put the counter in

Which files name your design, and what happens when one of them disagrees?

1. Copy the counter in place of the adder:

```
cp $L/counter.v src/counter.v
git rm -q src/project.v
cp $L/test_counter.py test/test.py
```

2. Make the five names agree. Tiny Tapeout wants a top module that starts with `tt_um_` and includes your GitHub username, so pick `tt_um_<username>_counter`, with any `-` in your username changed to `_` (Verilog names can't contain `-`). Then:
   - in `src/counter.v`, rename `module tt_um_ic102_counter` to your name;
   - in `info.yaml`, set `top_module` to it, and change `source_files` to list `counter.v` instead of `project.v`;
   - in `test/Makefile`, set `PROJECT_SOURCES = counter.v`;
   - in `test/tb.v`, put your module name on the line that names the module (Lab 02's breakage 3 was this line).
3. Describe the counter. In `info.yaml`, change the title and description, set `clock_hz: 10000000` (the test runs a 10 MHz clock), and fill in the pinout from the comment in `counter.v`: `ui[0]` is `EN`, `ui[1]` is `LOAD`, `ui[2]` is `DOWN`, `uo[0]` to `uo[7]` are the count, and `uio[0]` to `uio[7]` are the load value. Rewrite both sections of `docs/info.md` for the counter.
4. Check that the names agree:

```
python3 $L/check.py --offline
```

You should see all five in agreement. Ours read:

```
Names
  top_module in info.yaml    tt_um_student_counter
  source_files in info.yaml  counter.v
  PROJECT_SOURCES            counter.v
  module tb.v instantiates   tt_um_student_counter
  clock_hz                   10000000
```

followed by a History section with no bug commit yet, a `todo:` line about it, and `VERDICT: not yet`. That's expected at this stage, because you make those commits in Part 3. If a name disagrees, the check says which one, in another `todo:` line.

5. Run the test the way the test job runs it:

```
make -C "$TT" test PROJECT="$PWD"
```

You should see `Running on Icarus Verilog version 12.0 (stable)` and `Initialized cocotb v2.1.0` near the top, the same two lines the runner prints, and:

```
** test.test_reset               PASS         450.00          0.00     441401.93  **
** test.test_count_up            PASS        1450.00          0.00    2599614.96  **
** TESTS=2 PASS=2 FAIL=0 SKIP=0               1900.00          0.00     916311.41  **
```

The `SIM TIME` column is time inside the simulation, so your numbers in it should match ours exactly. The other two columns depend on how fast your computer is.

6. Commit and push, then predict which jobs will pass before you open the Actions tab:

```
git add -A
git commit -m "Replace the adder with an 8-bit counter"
git push
```

*Answer: A1, in Answers at the end.*

You should see: two passing tests locally, and six green jobs on GitHub.

## Part 2: A test that catches the bug

What does a test need to catch a one-line bug?

1. Put the course's bug into your counter, without committing it, and look at it:

```
python3 $L/bug.py apply
git diff src/counter.v
```

You should see:

```
src/counter.v: the load line now reads `count <= ui_in;` (not committed; see it with
git diff)
```

and a diff of one line:

```
-      count <= uio_in;
+      count <= ui_in;
```

`LOAD` now loads the counter from the control pins instead of the load-value pins. That's a believable slip, since the two names differ by one letter.

2. Before you run the test, predict whether the two starter tests catch it. Then run them:

```
make -C "$TT" test PROJECT="$PWD"
```

You should see `TESTS=2 PASS=2 FAIL=0`, the same as before the bug. Neither test ever raises `LOAD`, so neither can tell which bus it loads from.

3. Write a third test, `test_load`, at the end of `test/test.py`, where the comment `# Part 2: add a test of LOAD here.` is. Use the two starter tests as a model: call `start`, put a value on `uio_in`, raise `LOAD` with `dut.ui_in.value = LOAD`, let one rising edge go by with `await cycles(dut, 1)`, and assert that `uo_out` holds the value, with a message that says what you expected and what you got. Pick a value that the bug can't load by accident: with `LOAD` high, `ui_in` is `0b00000010`, so the buggy counter always loads 2.

   If an AI tool helps you write it, that's fine; you'll still have to show in step 4 that it fails on the bug, and that's what tells you the test works.

4. Run the tests with the bug still in:

   ```
   make -C "$TT" test PROJECT="$PWD"
   ```

   You should see your test fail with your own message. Ours read:

   ```
   AssertionError: after LOAD with uio_in = 0x5A, uo_out is 00000010, expected 0x5A
   ** test.test_load               FAIL        550.00          0.00    1319923.08  **
   ** TESTS=3 PASS=2 FAIL=1 SKIP=0             2450.00          0.00     658192.91  **
   ```

   then two lines from `make`:

   ```
   make[1]:    ***    [/foss/designs/modules/ic102-d/labs/common/tt-tools/.local/venv/
   lib/python3.11/site-packages/cocotb_tools/makefiles/simulators/Makefile.icarus:69:
   results.xml] Error 1
   make: *** [Makefile:42: test] Error 2
   ```

   The first is cocotb's own `make` stopping, because with cocotb 2.x it checks `results.xml` as the last step of every run and fails when a test failed. The second is the course's `make` passing that on. (The template's workflow comment says `make` "will return success even if the tests fail"; that was true of cocotb 1.x.) You'll need this in Lab 05, where a script has to tell a failing test from a design that doesn't compile.

   *Answer: one test that works: A2, in Answers at the end.*

5. Take the bug out and run the tests again:

   ```
   python3 $L/bug.py undo
   make -C "$TT" test PROJECT="$PWD"
   ```

   You should see `test.test_load PASS 550.00` (with your test's simulated time, if it differs) and `TESTS=3 PASS=3 FAIL=0`.

6. Commit your test and push it:

   ```
   git add test/test.py
   git commit -m "test: LOAD loads uio_in"
   git push
   ```

You should see: your test failing on the bug and passing without it, locally.

## Part 3: The bug on the runner

Does the runner's test job see what your workbench saw?

1. Wait for the push from Part 2 to go green. Then commit the bug, as a teammate might, and push:

```
python3 $L/bug.py commit
git push
```

You should see `Committed the bug as <hash>: counter: tidy up the load path`.

2. Before you look, predict which jobs fail. Then open the `test` run, find your assertion in the "Run tests" step, and compare it with step 4 of Part 2: the message, the `FAIL` line and its simulated time.

   *Answer: A3, in Answers at the end.*

3. Undo the bug with a new commit, and push:

```
git revert --no-edit HEAD
git push
```

4. When every job is green, run the check:

```
python3 $L/check.py
```

It finds the bug commit and the fix after it, and asks GitHub for the jobs on each. On the course's practice repository it printed:

```
Checking /foss/designs/projects/ttsky-ic102-d

Names
  top_module in info.yaml    tt_um_vprohacks_counter
  source_files in info.yaml  counter.v
  PROJECT_SOURCES            counter.v
  module tb.v instantiates   tt_um_vprohacks_counter
  clock_hz                   10000000

History
  bug commit  cd57aae
  fix commit  8f29d8f

GitHub: vproHacks/ic102-test-d
  test on the bug commit cd57aae: failure
  every job on the fix commit 8f29d8f:
    docs      success
    test      success
    gds       success
    precheck  success
    gl_test   success
    viewer    success

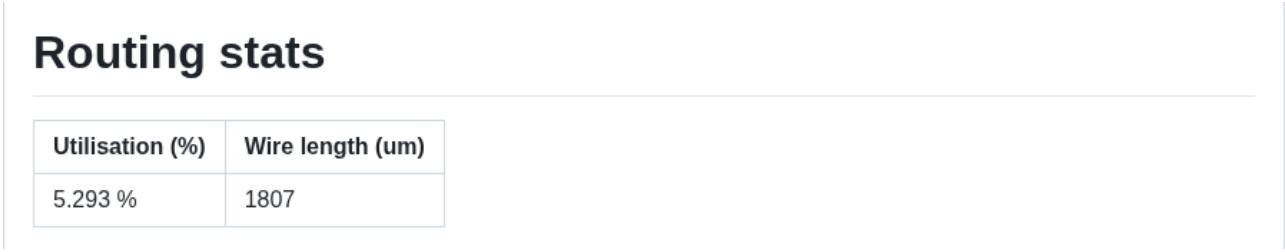
VERDICT: PASS
```

You should see: `test` red on the bug commit and green on the revert, with the same message you saw locally.

## Part 4: The numbers in the `gds` job

Where do the numbers on the run's summary page come from?

1. Open the latest `gds` run on GitHub (signed in) and scroll down its summary page to the `gds` job's summary. Under "Routing stats" there's a table with two numbers, "Utilisation (%)" and "Wire length (um)". Write both down.

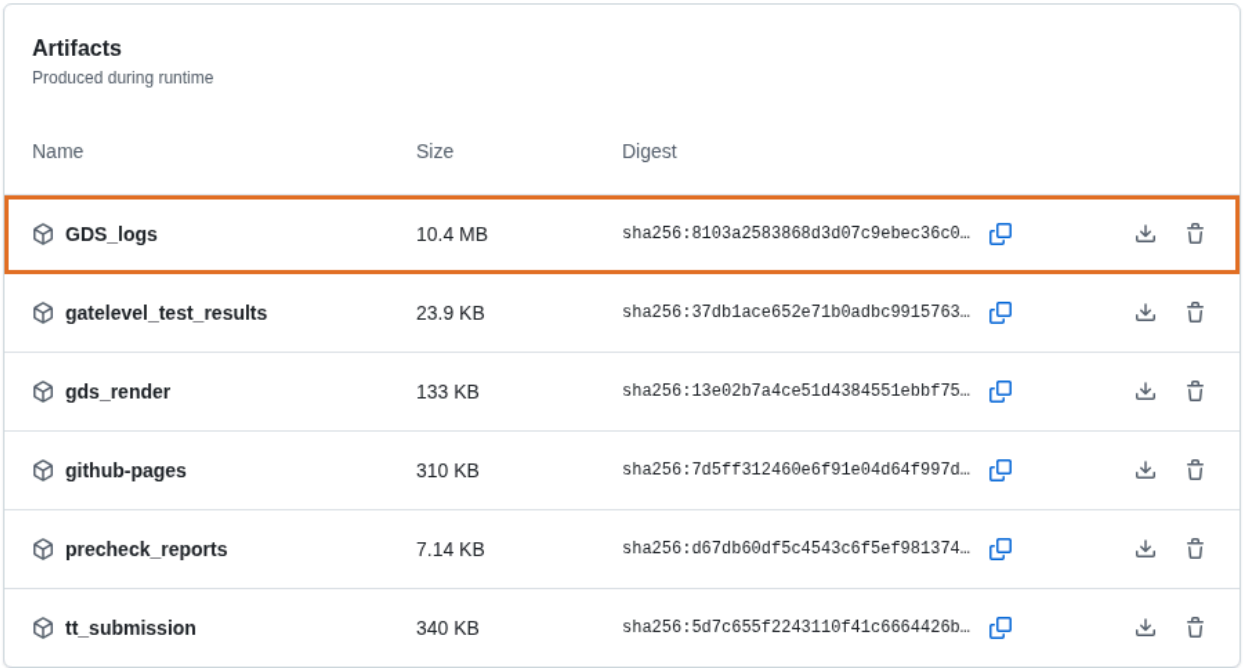


## Routing stats

| Utilisation (%) | Wire length (um) |
|---|---|
| 5.293 % | 1807 |

Figure 12.1: The routing stats in the gds job's summary, from the course's practice counter.

2. Further down the same page, under **Artifacts**, download `GDS_logs` with the download button at the end of its row. You have to be signed in to see the button. Unzip it into your workspace folder on your laptop, so the workbench sees it under `/foss/designs`:



**Artifacts**

Produced during runtime

| Name | Size | Digest |
|---|---|---|
| GDS_logs | 10.4 MB | sha256:8103a2583868d3d07c9ebec36c0… |
| gatelevel_test_results | 23.9 KB | sha256:37db1ace652e71b0adbc9915763… |
| gds_render | 133 KB | sha256:13e02b7a4ce51d4384551ebbf75… |
| github-pages | 310 KB | sha256:7d5ff312460e6f91e04d64f997d… |
| precheck_reports | 7.14 KB | sha256:d67db60df5c4543c6f5ef981374… |
| tt_submission | 340 KB | sha256:5d7c655f2243110f41c6664426b… |

Figure 12.2: The Artifacts section of a gds run's summary page, with GDS_logs highlighted.

The artifact holds `src/`, with the configuration the action built, and `runs/wokwi/`, LibreLane's whole run: one numbered folder per step, about 70 of them, and a `final/` folder.

3. Find each number's source. The utilisation is a line in the global-placement step's log, and the wire length is a row in `final/metrics.csv`. If you unzipped the artifact into a folder called `GDS_logs` at the top of your workspace:

```
cd /foss/designs/GDS_logs
grep GPL-0019 runs/wokwi/*-openroad-globalplacement/openroad-globalplacement.log
grep '^route__wirelength,' runs/wokwi/final/metrics.csv
```

In the course's practice repository, whose counter's summary is the one in the picture above (run 43839710228), they read:

```
[INFO GPL-0019] Utilization:                    5.293 %
route__wirelength,1807
```

In `sentry`'s green run 43796948508, a 2x2 design at the time, they were `0.628 %` and `919`. Yours should match the two numbers you wrote down in step 1, because the action's "Routing summary" step reads its numbers from these same two places (`print_stats` in tt-support-tools' `project.py`).

4. While you're in the run folder, read `runs/wokwi/*-misc-reportmanufacturability/manufacturability.rpt` (step 72, in `sentry`'s run and in the practice counter's). It says `Passed` under Antenna, LVS and DRC: the `gds` job checks its own layout before `precheck` checks it again.
5. Add a short "Numbers" section to your `README.md` with the two values, the run they came from, and the two file paths. Commit and push it.

You should see: the summary's two numbers and the two lines in the artifact agree.

## Part 5: A counter with no reset (optional)

About 20 minutes. Tiny Tapeout's documentation says that on an ASIC "flops will have a random initial value at power on", so "Initial statements are not allowed, so you must use an explicit reset." Find out what happens if you ignore that.

1. On a new branch, remove the reset from `counter.v`: delete the `if (!rst_n) count <= 8'd0; else` part of the `always` block, and add `initial count = 8'd0;` after `reg [7:0] count;`. Add `rst_n` to the `_unused` list.
2. Predict whether the RTL test still passes, then run `make -C "$TT" test PROJECT="$PWD"`.

   *Answer: A4, in Answers at the end.*
3. Predict which tests fail at gate level, then push the branch and wait for `gl_test`.

   *Answer: A5, in Answers at the end.*
4. Delete the branch, on GitHub and locally; `main` keeps its reset. The branch's `viewer` job passed, and it published the branch's layout to your project's page, which shows the no-reset counter until the next `gds` run on `main`.

## Finishing

You're done when `python3 $L/check.py` prints `VERDICT: PASS`: the five names agree, the runner's `test` job failed on the bug commit, and all six jobs passed on the commit that took the bug out.

Questions:

1. The starter tests passed on the buggy counter. What would you have needed to know to write `test_load` without being shown the bug?

   *Answer: A6, in Answers at the end.*
2. Your test passed with Icarus 12.0 in the workbench. Is that enough to be sure the runner's `test` job passes?

   *Answer: A7, in Answers at the end.*

Submit: the link to your repository.

## Going further (optional)

Run the same test with the other two Icarus versions:

```
make -C "$TT" test-icarus13 PROJECT="$PWD"
make -C "$TT" test-workbench PROJECT="$PWD"
```

With our counter and the two starter tests, all three versions gave the same results and simulated times (`test.test_reset PASS 450.00`, `test.test_count_up PASS 1450.00`). Your tools and the runner's tools shows a design where they don't agree.

Next on the Course path: read Pull requests and review, Branch rules and How open-source projects work, then do Lab 04: A change through review.

# Answers

**A1. Part 1: Put the counter in**

All six, if the names agree. `gds` builds a new layout from `counter.v`, `gl_test` runs the two tests on that layout's netlist, and `docs` checks the new `info.yaml`. The test reads only the pins and samples them on the falling clock edge, so it doesn't depend on any name inside the design, which the netlist doesn't keep.

**A2. Part 2: A test that catches the bug (Answer: one test that works)**

```
@cocotb.test()
async def test_load(dut):
    await start(dut)
    dut.uio_in.value = 0x5A
    dut.ui_in.value = LOAD
    await cycles(dut, 1)
     assert dut.uo_out.value == 0x5A, f"after LOAD with uio_in = 0x5A, uo_out is
{dut.uo_out.value}, expected 0x5A"
```

**A3. Part 3: The bug on the runner**

`test` fails, and so does `gl_test` in the `gds` workflow, which runs the same three tests on the gate-level netlist. `docs`, `gds`, `precheck` and `viewer` pass, because the design still builds; it just does the wrong thing. The runner's message and `SIM TIME` match yours exactly, because the runner used the same cocotb 2.1.0 and Icarus 12.0. In the course's practice repository (run 43841055248) "Run tests" printed `Initialized cocotb v2.1.0 from /opt/hostedtoolcache/Python/3.11.18/...`, `Running on Icarus Verilog version 12.0 (stable)`, and:

```
AssertionError: after LOAD with uio_in = 0x5A, uo_out is 00000010, expected 0x5A
** test.test_load                 FAIL         550.00           0.00    1698301.39  **
** TESTS=3 PASS=2 FAIL=1 SKIP=0               2450.00           0.00    1017018.65  **
```

`gl_test` printed the same three lines, with `Running on Icarus Verilog version 13.0 (stable)` and its own numbers in the last two columns.

**A4. Part 5: A counter with no reset (optional)**

Yes, all three. In simulation an `initial` statement sets the register at time 0, so the count starts at 0 whether or not reset does anything. We got `TESTS=3 PASS=3 FAIL=0`.

**A5. Part 5: A counter with no reset (optional)**

`test_reset` and `test_count_up`. Synthesis can't build an `initial` statement into a flip-flop, so in the netlist the count starts unknown, and counting up from unknown stays unknown. `test_load` passes, because loading sets every bit. In the course's practice repository (run 43843559172), `gl_test` ran the three tests on the runner's netlist with Icarus 13.0 and printed:

```
AssertionError: after reset uo_out is XXXXXXXX, expected 0
AssertionError: after 10 cycles with EN high uo_out is XXXXXXXX, expected 10
** test.test_reset                FAIL         450.00           0.01      32495.41  **
** test.test_count_up              FAIL        1450.00           0.00    2716307.02  **
** test.test_load                  PASS         550.00           0.00    1897683.79  **
** TESTS=3 PASS=1 FAIL=2 SKIP=0               2450.00           0.02     154991.26  **
```

`ttsky-verilog-tinyCGRA`'s history from May 2026 shows the same pattern: its gate-level test failed, and commits titled "comment all user_project asserts" followed.

**A6. Finishing**

Only the specification in the comment at the top of `counter.v`: `LOAD` loads the value on `uio_in`. A test of every behaviour in the specification would have included it, and would have caught the bug the moment it was committed. Writing tests from the specification, rather than from the bugs you know about, is what DD104 is about.

**A7. Finishing**

For this test, yes: the runner's job uses the same Icarus 12.0 and cocotb 2.1.0, and `make test` runs the job's steps. Two things could still differ: your `test/requirements.txt` (both jobs install from it, and `make test` warns when it differs from the course's), and anything outside the test, which is why the `gds` jobs exist. Your tools and the runner's tools goes through the gaps.

# IV 4. Review


13 **Pull requests and review . . . . . 85**
13 Issues, branches and pull requests . . . . . 85
13 What review is for . . . . . 85
13 Writing a pull request someone can review . . . 86
13 What a reviewer checks that CI can't . . . . . 87
13 Pull requests written with AI tools . . . . . 88
13 Check yourself . . . . . 88
13 Answers . . . . . 88

14 **Branch rules . . . . . 91**
14 What a ruleset is . . . . . 91
14 Three rulesets in the team's repositories . . . . . 91
14 What write access allows . . . . . 93
14 Rules that do what they say . . . . . 93
14 Check yourself . . . . . 94
14 Answers . . . . . 94

15 **How open-source projects work . . . . 95**
15 Licences . . . . . 95
15 Copying someone else's code . . . . . 95
15 Working with an upstream project . . . . . 96
15 Working in public . . . . . 96
15 Check yourself . . . . . 96
15 Answers . . . . . 97

16 **Lab 04: A change through review . . . 99**
16 Before you start . . . . . 99
16 Part 1: Your review repository . . . . . 99
16 Part 2: Claim a pin in your partner's repository . 102
16 Part 3: A neighbouring claim . . . . . 103
16 Part 4: Review your partner's pull request . . . 105
16 Part 5: Answer the review and resolve the conflict . . . . . 106
16 Part 6: A pull request written by an AI tool (optional) . . . . . 108
16 Finishing . . . . . 108
16 Going further (optional) . . . . . 108
16 Answers . . . . . 109

# 13. Pull requests and review

This page answers: *how does a change get from your branch into the `main` that everyone shares, and what is a reviewer for, when the workflows already check so much?*

A **pull request** asks for one branch to be merged into another, usually into `main`. GitHub shows the change as a diff, shows the workflow runs on the branch's latest commit, and gives everyone a place to discuss it before anything on `main` changes. Review is the discussion part: another person reads the change and either approves it or asks for changes. In Lab 04 you'll take a change of your own through review in your partner's repository, and review the one your partner opens in yours.

## Issues, branches and pull requests

Work on a project usually starts as an **issue**, a numbered note in the repository that says what needs doing. `sentry` shows the habit that keeps issues useful: its branches are named after the issue they work on (`12-implement-rtl-for-window-controller-module`, `57-rewrite-input-decoder-unit-tests-in-systemverilog`), so anyone looking at the branch list can find the discussion behind each one. Its pull request #33, "Event Decoder FSM", was linked to issue #3, "Implement Input Decoder RTL", and merging the pull request closed the issue. The same link can be made from the description: `hart-1`'s pull request #5 begins with `Closes #2`, which closed issue #2 when it was merged.

The link matters the other way round too. If an issue is never linked to the work done for it, it stays open, and anyone who reads the issue can't see that the work exists. `dpll` has three issues, "PFD - Design", "DCO - Design" and "Freq. Divider - Design", open since Oct 24, 2025. Four commits for the frequency divider, dated Nov 1 and 2, 2025, sit on a branch called `frequency-divider`, the last one titled "integrated with main". No pull request was opened, so none of that work reached `main`, which still has only its first commit, and the issue still reads as if nothing had happened.

## What review is for

The team's own record shows how rarely review happens. These are the merged pull requests in five of the team's repositories on Nov 15, 2026:

| Repository | Merged pull requests | Merged by their own author | With at least one review |
|---|---|---|---|
| `crosswalk` | 6 | 6 | 0 |
| `protocol-emulator` | 1 | 1 | 0 |
| `ttsky-verilog-tinyCGRA` | 2 | 1 | 1 |
| `hart-1` | 1 | 0 | 0 |
| `sentry` | 10 | 2 | 2 |

Seventeen of the twenty were merged without a review. That's common in small teams moving quickly, and the workflows did catch some problems along the way. The workflows can't judge whether the design is right, though, and that judgment is what a reviewer adds. `sentry`'s pull request #33 had thirteen reviews, and an early round shows what they were for:

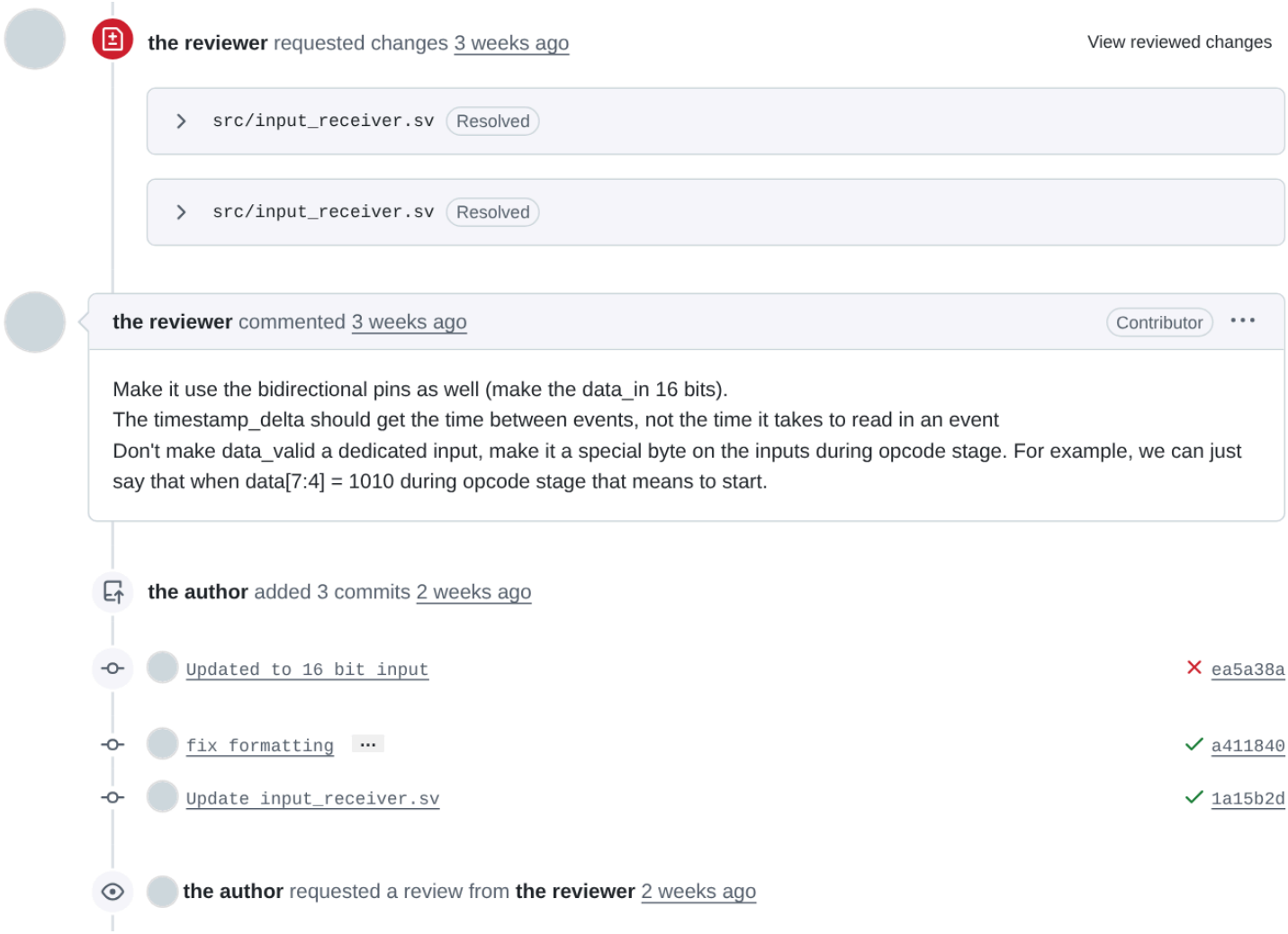



Figure 13.1: One round of review on sentry's pull request #33. Names are replaced with roles, and avatars with grey discs.

The reviewer asked for three changes: use the bidirectional pins as well, which makes `data_in` 16 bits wide; have `timestamp_delta` measure the time between events, not the time it takes to read one; and signal `data_valid` with a special byte on the inputs during the opcode stage, instead of a dedicated input. None of those three requests could come from a workflow. Each is a decision about what the design should do, made by someone who knew what it was for, and each is easier to settle before a merge than after other blocks depend on it.

## Writing a pull request someone can review

A reviewer can only review what they can understand, so the author's job is to make that easy. Two things decide it: the description and the size.

**The description** should say what the change does, why, how you tested it, and what you know is still wrong. `protocol-emulator`'s pull request #2, open since Nov 9, 2026, is a good model. It has a summary, the new pinout, a table of the tests it adds with one line each on what they check, and a section headed "Expected CI failure" that explains, before anyone asks, why one of its own tests fails:

> `test_uio_pin_directions` **fails on purpose**. `uio[0]` is read as UART RX but also driven as UART TX (`uio_oe[0] = 1`). On silicon the RX input would read back our own TX and fight an external driver. One fix: move TX to `uio_out[2]` with `uio_oe = 8'b0000_0100` and update `info.yaml`. This is left for review because it changes the pinout.

When we looked on Nov 15, the pull request was still waiting for that decision, with no reviews.

**The size** should be small enough to read in one sitting. `hart-1`'s pull request #5 added 7,408 lines in 47 files, and it was merged on Nov 15, three weeks after it was opened, without a review. Most of those lines are TinyQV, an open-source processor core copied into `src/tinyQV/`, and the description says so: "Everything outside src/tinyQV/ is our code", with a pointer to an `UPSTREAM.md` file that records exactly where the copy came from. That makes the pull request reviewable in two parts. The copied part can be checked mechanically, by comparing it with the upstream commit `UPSTREAM.md` names, and only the rest needs reading line by line. How open-source projects work says more about copying code this way. It would have been easier still as two pull requests, one for the copy and one for the team's own files.

If you want feedback before a change is finished, open the pull request as a **draft**. A draft shows the diff and runs the workflows like any other pull request, but can't be merged until you mark it ready for review, so it's safe to open early.

## What a reviewer checks that CI can't

The workflows check that `info.yaml` parses, that the layout builds and passes precheck, and that the tests pass. They can't check that the values make sense, that the tests test anything, or that the design does what it should. Those are the reviewer's job, and the team's repositories have an example of each.

**Values that pass validation and are still wrong.** The validator requires `clock_hz` to be an integer, so `clock_hz: 50` passes. `protocol-emulator`'s pull request #2 points out that it "should probably be `50000000`", which is a factor of a million that no workflow would notice.

**Pins that the tests don't look at.** A design can read a pin as an input and drive the same pin as an output, with `uio_oe` set, and every workflow will pass, because none of them checks what the pins are for. `protocol-emulator`'s `uio[0]` problem was found only because someone wrote a test specifically for pin directions.

**Tests that no longer test anything.** A deleted or commented-out assertion makes a red test green without changing the design at all. Here is the commit in `protocol-emulator` that did it. The red cross and `3 / 6` at the top are GitHub's summary of the checks on that commit: three of the six jobs passed, and `test`, `gl_test` and `viewer` failed.

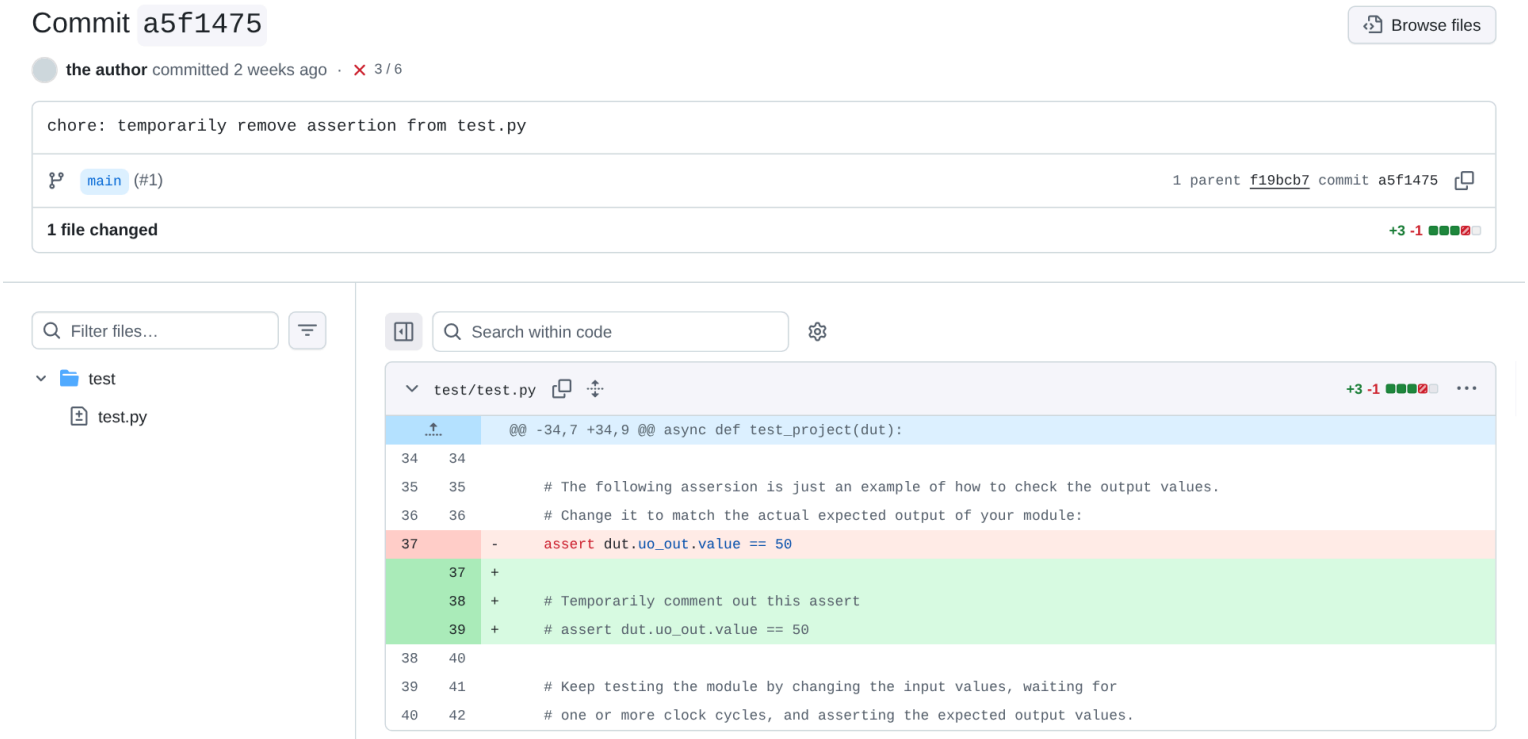



Figure 13.2: The commit that commented out protocol-emulator's only assertion. The author's name is replaced with a role.

`ttsky-verilog-tinyCGRA` has a run of the same kind of commit from May 2026, titled "comment assert input in load_input()", "comment all user_project asserts" and "commented asserts, no functional testing". Sometimes removing an assertion is the right call, and the pull request that was finally merged there includes one such commit, "removed config_done assert (unreliable in GLS)", which says which assertion and why. A reviewer's job is to notice every removed check and ask for that reason when it's missing.

**Layouts, which a diff can't show.** In IC102-A your change is a GDS file, which is binary, so the diff tells a reviewer nothing beyond "this file changed". Review a layout from the PNG render in the `gds_render` artifact and from the precheck report instead.

When you review, a short list helps:

- Do the tests still assert something, and is every removed or weakened check explained?
- Do the `info.yaml` values make sense: `clock_hz`, `tiles`, the pinout, and does the pinout match how the design drives `uio_oe`?
- Does the description match the diff? Does it claim anything the diff doesn't do, or leave out something the diff does?
- Are the runs on the branch's latest commit green, or is every red one explained?
- Has anything generated, large or secret been committed?

When you ask for a change, say what and why, as the `sentry` reviewer did, and when the author has made it, look again and approve.

# Pull requests written with AI tools

You may use AI tools, such as Claude, ChatGPT or Copilot, to help you write code and pull requests in this course, and the team's repositories already have pull requests written that way. `protocol-emulator`'s #2 ends with "Generated with Claude Code", as does `hart-1`'s #4 (closed in favour of #5), and `ttsky-verilog-tinyCGRA`'s #3 was opened by the Copilot coding agent (it happens to be the only one of that repository's pull requests with a review).

Two rules apply when you do. First, the person who opens a pull request answers for every line in it, however it was written. Say in the description which parts a tool wrote and what you checked yourself, so a reviewer knows where to look hardest. Second, as a reviewer, treat what a description says as claims to check. Pull request #2 says, for example, that "Icarus rejected the use-before-declaration, so the design didn't build", and that 6 of its 7 tests pass under "Icarus, cocotb 2.0.1". Each of those can be checked by running the tests at the pull request's latest commit, and a reviewer should, because a tool can state something fluently that turns out not to be true. Tools also make large pull requests cheap to write, without making them any cheaper to review, so the advice on size above applies with extra force, and so does the habit of looking for deleted assertions.

# Check yourself

1. A pull request changes `clock_hz: 50` to `clock_hz: 50000000`, and every workflow was already green before it. Is it worth reviewing, and why didn't any workflow complain about the old value?

*Answer: A1, in Answers at the end.*

2. A pull request adds 7,000 lines, nearly all of them an open-source core copied into one folder. How can you review it without reading every line?

*Answer: A2, in Answers at the end.*

3. Which of these would a workflow catch, and which needs a reviewer: an empty project title, a commented-out assertion, a `uio` pin that's both read and driven, a DRC error in a layout?

*Answer: A3, in Answers at the end.*

4. You used an AI tool to write the cocotb test in your pull request. What should your description say about it?

*Answer: A4, in Answers at the end.*

Next: Branch rules.

# Answers

### A1. Check yourself

Yes. The validator only checks that `clock_hz` is an integer, so 50 passes even though it means 50 Hz. Checking that the value matches the real clock (here 50 MHz, a factor of a million) is exactly the kind of thing only a person can do.

**A2. Check yourself**

Check the copied folder against the upstream commit its provenance file names (in `hart-1`, `UPSTREAM.md`), which a tool can compare for you, and read the files outside that folder in full. Better still, ask for it as two pull requests, one for the copy and one for the team's own files.

**A3. Check yourself**

The empty title (the docs and gds workflows fail) and the DRC error (precheck fails) are caught by workflows. The commented-out assertion makes the test pass, and the pin problem passes everything unless someone has written a test for it, so both need a reviewer.

**A4. Check yourself**

That the tool wrote the test, what you checked yourself (for example, that the test fails on the bug it's meant to catch and passes on the fix), and anything you weren't able to check. You're responsible for the test either way, and saying this tells your reviewer where to look hardest.

# 14. Branch rules

This page answers: *how can a repository make sure every change to `main` goes through a pull request, a review and a passing check, and why do the rules in the team's repositories stop so little?*

Pull requests and review describes the habit. A **branch rule** is how a repository enforces it: GitHub rejects a push, or blocks a merge, that breaks the rule, whoever makes it. Rules are easy to get subtly wrong, though, and the three sets of rules in the team's repositories each look protective and leave `main` open in a different way. This page goes through all three, because between them they show what each setting does.

## What a ruleset is

GitHub has two ways to set branch rules: the older "branch protection rules" and the newer **rulesets**. The team's repositories use rulesets, so this page does too. A ruleset has five parts:

- **A name**, which is only a label. A ruleset called "protect-main" protects whatever its other settings say, and nothing more.
- **An enforcement status**: active, or disabled.
- **Target branches**, the branches it applies to, given as names or patterns. The default branch (`main` in a Tiny Tapeout repository) is the usual target.
- **Rules**. The ones that matter here are *Restrict deletions* (nobody can delete the branch), *Block force pushes* (nobody can rewrite its history), *Require a pull request before merging* (nobody can push to it directly, and you can require some number of approvals), and *Require status checks to pass* (named jobs must have passed on the pull request's latest commit).
- **A bypass list**: people, teams or roles that may skip the rules. Each entry is either "Always allow", or "For pull requests only", in which case the actor "is now required to open a pull request" but may merge it without meeting the other requirements (creating rulesets).

Rulesets, like the older protection rules, are available in public repositories on GitHub's free plans, and in private repositories only on paid ones (about rulesets). The team's organization is on the free plan, so a private repository such as `dpll` can't have branch rules at all. Anyone who can read a public repository can also read its rulesets, at `github.com/<owner>/<repo>/rules`, which is how the pictures on this page were taken without signing in.

## Three rulesets in the team's repositories

**`hart-1`, "Don't Push to Main"** (created Nov 15, 2026). Its name says what it's for, but GitHub's own page for the ruleset says that it "will not be applied":

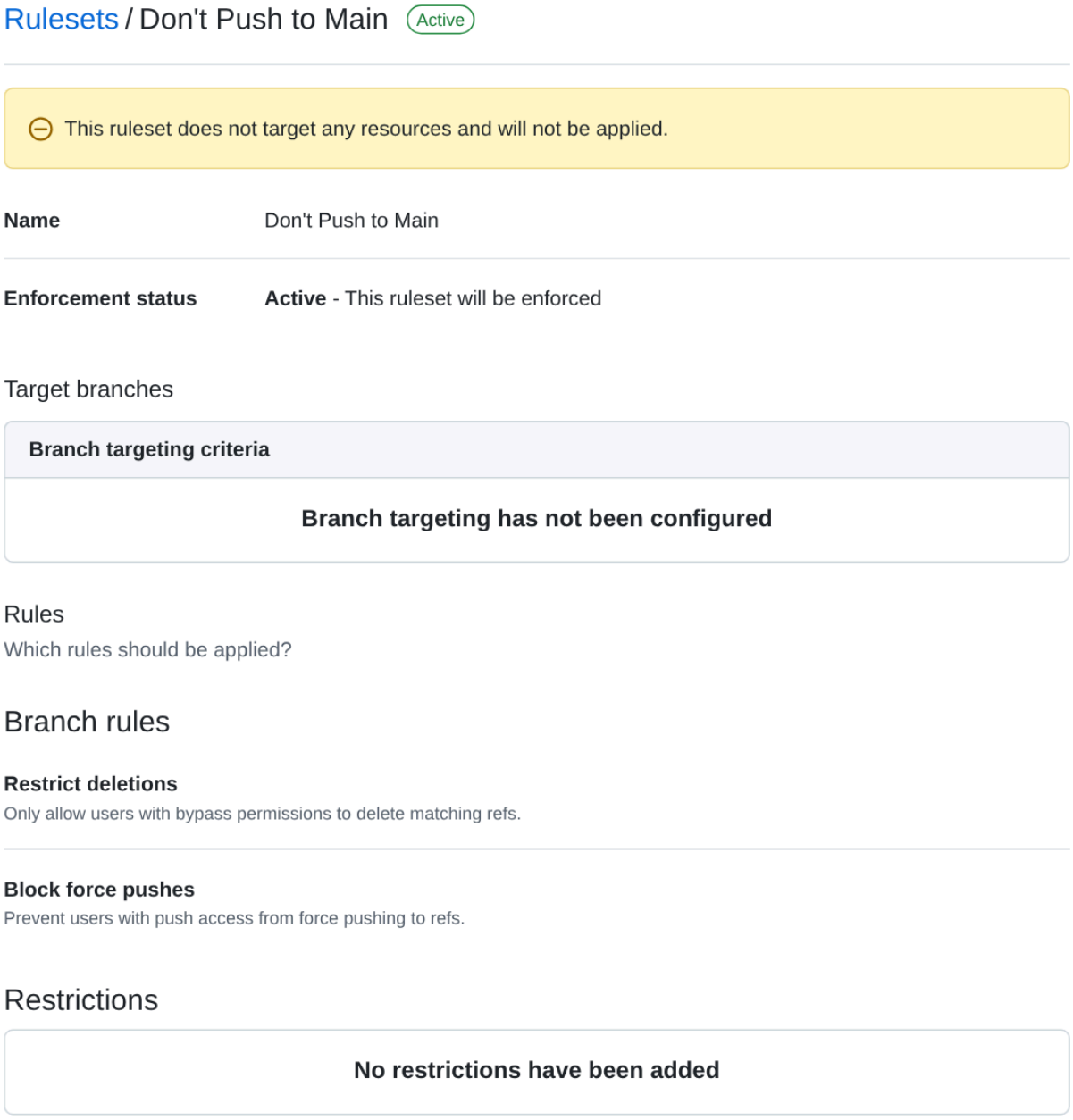



Figure 14.1: hart-1′s ruleset “Don’t Push to Main”, seen without signing in.

The ruleset is active, but its list of target branches is empty, so it applies to no branch. Suppose it did target `main`, though. Its two rules stop anyone deleting `main` or force-pushing to it, which are worth having, but neither stops an ordinary `git push` to `main`. To do what its name says, it would also need *Require a pull request before merging* and the default branch as its target.

**`crosswalk`, “protect-main”** (created Oct 31, 2026). On paper this one is set up properly: it targets the default branch, restricts deletions, blocks force pushes, and requires a pull request with one approval. Its bypass list, however, has one entry: a repository role, with “Always allow”. Since the ruleset was created, six commits have reached `main` without a pull request (`61156d4`, `3985abc`, `34cf810`, `8973a92`, `685608c` and `2f71732`; the last is a merge made locally and pushed, “Merge Sample/hold and CDAC, update interface details”), and all six pull requests merged in that time were merged by their own authors with no approval. With “Always allow”, everyone who holds the role is outside the rule for every push and every merge.

**`protocol-emulator`, “main”** (created Oct 24, 2026). This one targets `refs/heads/main`, blocks force pushes and requires a pull request, with the required number of approvals set to zero, and one person on

the bypass list with “Always allow”. Zero approvals is a reasonable choice: every change still arrives as a pull request, with its diff and its workflow runs in one place, even when nobody else is available to review it. The same repository has a second ruleset, “Branch Hierarchy (L*) Ruleset”, for branches named `L1/*`, `L2/*` and `L3/*`, which is disabled.

Rules like these are easy to end up with, because a ruleset with a reassuring name and a few rules looks as if it’s doing its job. Reading each of its five parts, as above, is how you check one.

## What write access allows

When we checked on Nov 15, 2026, the `uoftasic` organization’s base permission for members was `write`: every member can push to every one of the organization’s repositories, to any branch that no rule protects. This course doesn’t use the organization, and everything you do in it happens in your own account. If you later join one of the team’s projects as a member, though, in most of its repositories the only thing between your laptop and their `main` will be your own judgment. Use that access the way you’d want others to use theirs: push to your own branch, open a pull request, and leave `main` to the project’s own process.

## Rules that do what they say

For a shared repository such as a team project, a ruleset that matches its name looks like this:

| Part | Setting |
|---|---|
| Enforcement | Active |
| Target | The default branch |
| Rules | Restrict deletions; Block force pushes; Require a pull request before merging, with at least one approval; Require status checks to pass, naming the jobs that must be green |
| Bypass list | Empty, or project leads with “For pull requests only” |

The status checks are named after jobs (`docs`, `test`, `gds`, `precheck`, `gl_test`, `viewer` in the digital template; `docs`, `gds`, `precheck` and `viewer` in the analog one). Tiny Tapeout’s workflows run on every push, so a pull request from a branch in the same repository gets runs on its latest commit, and those are the results the rule checks. A pull request from a fork is different: its commits are pushed to the fork, not to the repository it’s aimed at, so that repository’s workflows have to run on `pull_request` as well, or the required checks never report. A bypass “For pull requests only” keeps a way for a project lead to merge an urgent fix without an approval, while still putting that fix through a pull request where everyone can see it.

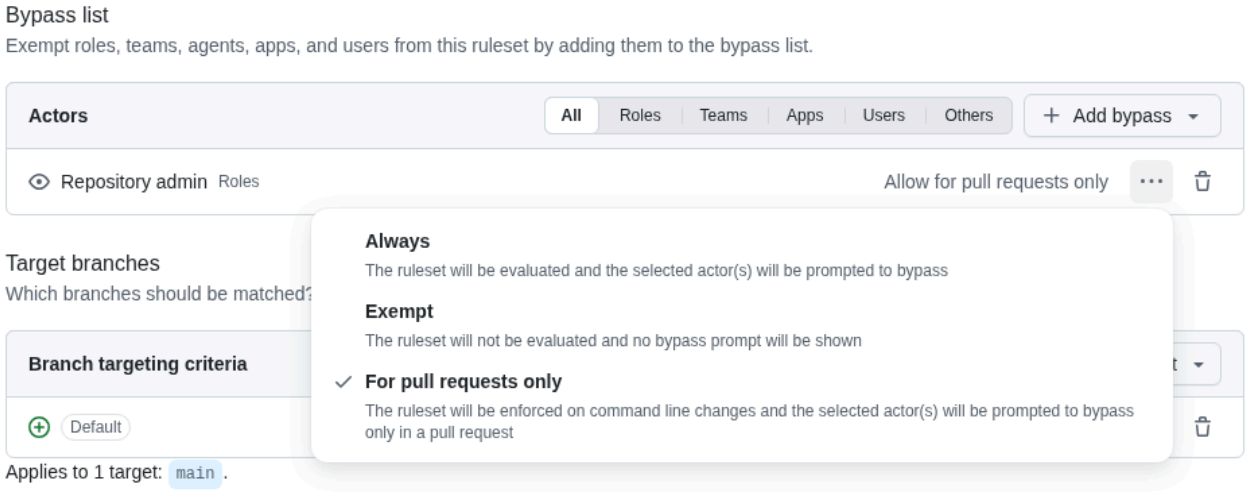



Figure 14.2: A ruleset's bypass list with its entry's menu open, offering "Always", "Exempt" and "For pull requests only". This entry is set to "For pull requests only", which the list shows as "Allow for pull requests only".

In Lab 04 you'll put rules like these on a repository of your own, with yourself on the bypass list "For pull requests only", and you'll see your partner's pull request wait until it has both your approval and green checks.

## Check yourself

1. A ruleset on `main` has two rules, Restrict deletions and Block force pushes, and it's active and targets the default branch. Can a member push a new commit straight to `main`?

*Answer: A1, in Answers at the end.*

2. `crosswalk`'s ruleset requires one approval, yet all six pull requests merged since it was created have none. How?

*Answer: A2, in Answers at the end.*

3. A friend's Tiny Tapeout repository is private, in a personal account on the free plan. Can they add a ruleset to protect `main`?

*Answer: A3, in Answers at the end.*

4. What's the difference between "Always allow" and "For pull requests only" on a bypass list?

*Answer: A4, in Answers at the end.*

Next: the next page in this row of the Course path.

## Answers

**A1. Check yourself**

Yes. Those rules stop the branch being deleted or its history being rewritten, but an ordinary push adds commits without rewriting anything. To stop direct pushes, the ruleset needs *Require a pull request before merging*.

**A2. Check yourself**

Its bypass list gives a repository role "Always allow", so anyone holding that role can merge without an approval, and push to `main` without a pull request at all, which six commits did.

**A3. Check yourself**

No. On GitHub's free plans, rulesets and branch protection are available only in public repositories. They'd need to make the repository public (which this course does anyway, for free runners and for Pages) or move to a paid plan.

**A4. Check yourself**

"Always allow" exempts the actor from every rule, so they can push directly to the branch. "For pull requests only" still requires them to open a pull request, but lets them merge it without meeting the other requirements, such as an approval, so their changes stay visible.

# 15. 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.

# 16. Lab 04: A change through review

You and a partner each make a small Tiny Tapeout project of your own, the review sandbox, and put branch rules on it. Each of its sixteen output pins is driven by a slot that starts empty. You'll claim a pin in your partner's sandbox with a pull request from a fork, and your partner has to approve it before it can merge. In your own sandbox you'll review your partner's pull request and ask for a change. You'll also claim the pin next to theirs yourself, which leaves them a conflict to resolve, and their claim in their own sandbox leaves you one.

Package: `labs/lab-04-a-change-through-review/`

## Before you start

- Read Pull requests and review, Branch rules and How open-source projects work.
- Find a partner who's doing this lab at the same time as you. You each own one repository and contribute to the other's, so you'll wait for each other a few times; agree how you'll say when you're ready (the team Discord is fine).
- You need the token from Lab 01, Part 2. Its Workflows permission matters here, because the first push of your sandbox includes the files in `.github/workflows/`.
- Every command runs in a terminal on the workbench desktop. In each new terminal, run these first; the steps use `$L` for this lab's package and `$TT` for the test job's tools:

```
. /foss/designs/common/.designinit
L=/foss/designs/modules/ic102-d/labs/lab-04-a-change-through-review
TT=/foss/designs/modules/ic102-d/labs/common/tt-tools
```

The steps write `YOUR-USERNAME` and `PARTNER-USERNAME` for your two GitHub usernames, and `N` for a pull request's number. The examples have you claim `uo[3]` for a parity block in your partner's repository, and `uo[4]` for a majority block in your own; use your own pins and blocks.

This lab takes about 1 hour 45 minutes. That includes the 5 to 10 minutes GitHub's runners take after each push, but not the time you spend waiting for your partner.

## Part 1: Your review repository

What does it take to turn the course's folder into a repository with rules on it?

1. On GitHub, create a new public repository called `ttsky-ic102-d-review` in your own account. Leave it empty: no README, no `.gitignore` and no licence, because the sandbox has its own.

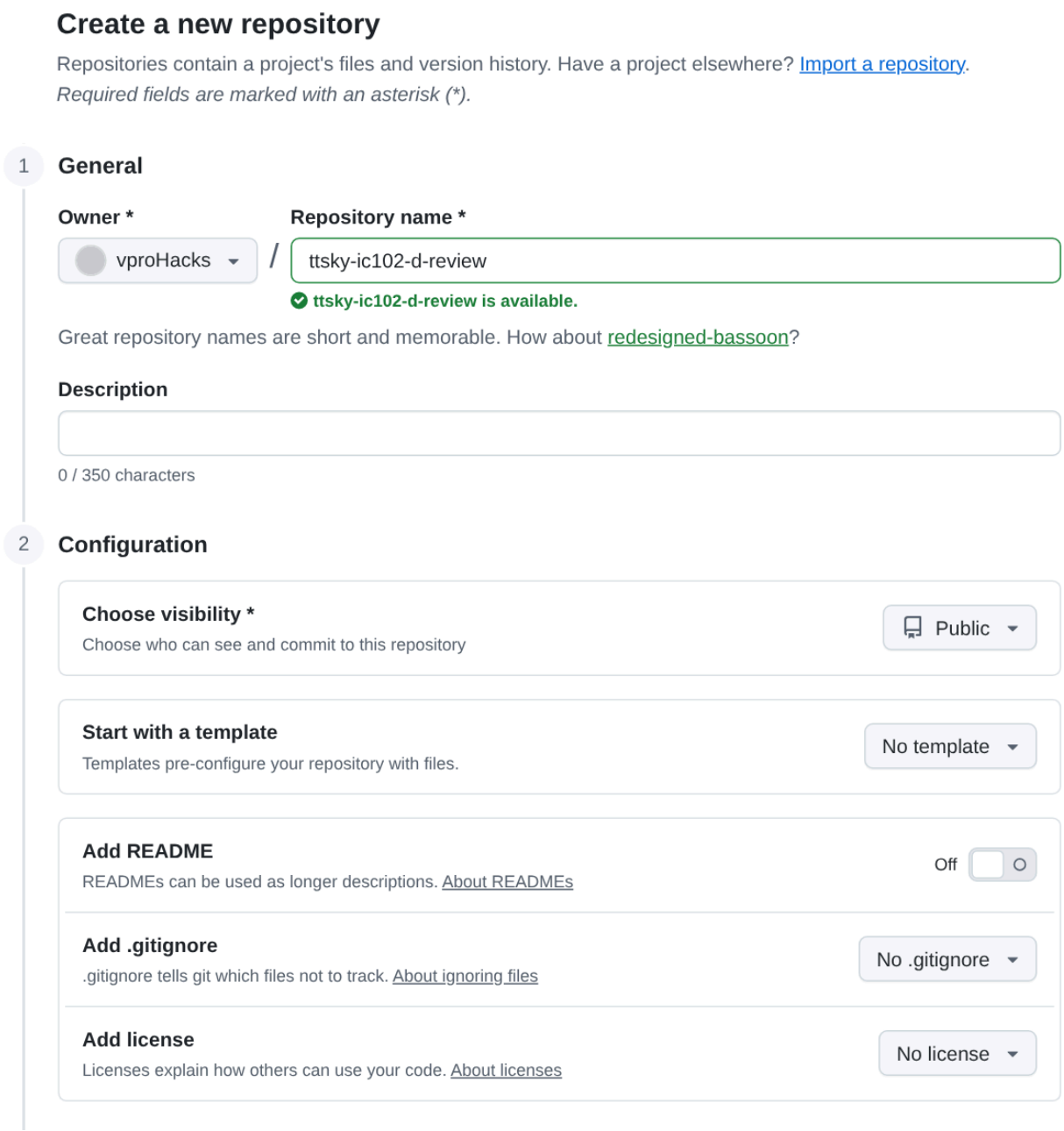



Figure 16.1: Creating the empty review repository in the course's practice run on Nov 16, 2026. The form started with Add README off and with No .gitignore and No license chosen, so none of the three had to be changed.

2. Make a repository from the sandbox folder, and push it:

```
mkdir -p /foss/designs/projects && cd /foss/designs/projects
cp -r $L/sandbox ttsky-ic102-d-review
cd ttsky-ic102-d-review
git init -b main
git config user.name "Your Name"
git config user.email "ID+USERNAME@users.noreply.github.com"
git config credential.helper 'cache --timeout=28800'
git add -A
git commit -m "Start from the IC102-D review sandbox"
```

```
git remote add origin https://github.com/YOUR-USERNAME/ttsky-ic102-d-review.git
git push -u origin main
```

You should see `50 files changed, 1497 insertions(+)` from the commit, and the push end with `* [new branch]  main -> main` and `branch 'main' set up to track 'origin/main'.`

3. The push starts the sandbox's three workflows, and `viewer` fails with the 404 from Lab 01, because Pages is off in a new repository. Turn Pages on as in Lab 01, Part 5, then start a new run of `gds` with **Run workflow** on `main`. (A re-run would fail, for the reason that part gives.)

4. Add the rules. On GitHub, open **Settings** › **Rules** › **Rulesets**, choose **New ruleset** › **Import a ruleset**, and pick `ruleset.json` from your clone. On your laptop it's in the `projects/ttsky-ic102-d-review` folder of your workspace. Before you press **Create**, check the page against `RULESET.md`, which describes the same ruleset in the five parts of Branch rules. The bypass list has one entry, the repository's admin role (which is you), set to "For pull requests only".

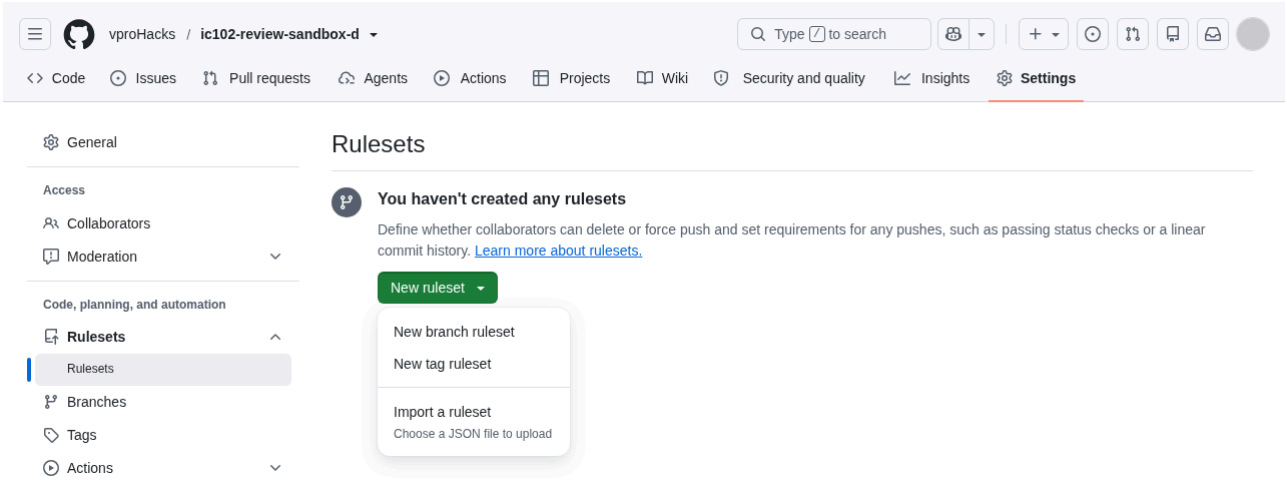



Figure 16.2: Importing the sandbox's ruleset: the New ruleset menu, with Import a ruleset at the bottom.

5. Before you try it, write down what will happen if you commit straight to `main` and push. Then try it with a commit that changes nothing, and throw the commit away afterwards:

```
git commit --allow-empty -m "Try to push to main"
git push origin main
git reset --hard origin/main
```

GitHub refuses the push and names the rules, although the repository is yours. In the course's practice sandbox the push printed:

```
remote: error: GH013: Repository rule violations found for refs/heads/main.
remote: Review all repository rules at https://github.com/vproHacks/ic102-review-
sandbox-d/rules?ref=refs%2Fheads%2Fmain
remote:
remote: - Changes must be made through a pull request.
remote:
remote: - 5 of 5 required status checks are expected.
remote:
To https://github.com/vproHacks/ic102-review-sandbox-d.git
 ! [remote rejected] main -> main (push declined due to repository rule violations)
error: failed to push some refs to 'https://github.com/vproHacks/ic102-review-
sandbox-d.git'
```

The bypass list lets you skip the approval, but only in a pull request.

You should see: all three workflows green on your `main`, `viewer` included, and a direct push refused.

## Part 2: Claim a pin in your partner's repository

What does a pull request need for your partner to review it in five minutes?

1. Fork your partner's repository: open `https://github.com/PARTNER-USERNAME/ttsky-ic102-d-review` and press **Fork**, at the top right. On the **Create a new fork** page that opens, check that **Owner** is your own account, which it should be already. **Repository name** starts out as the name of the repository you're forking, and you already have a `ttsky-ic102-d-review`, so change it to `ttsky-ic102-d-review-fork` and wait for GitHub to say that the name is available. Leave **Copy the `main` branch only** ticked, and press **Create fork** (forking a repository). The fork is yours to push to; your partner's `main` changes only through pull requests.

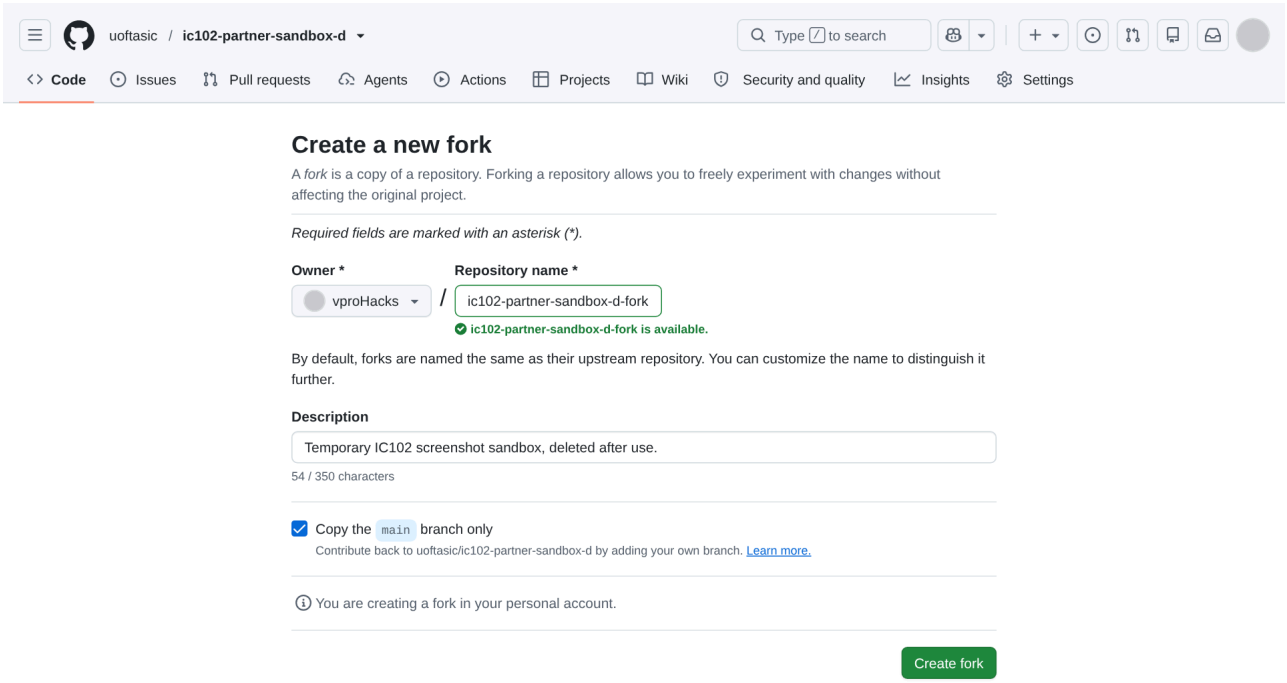



Figure 16.3: Forking your partner's repository, in the course's practice run on Nov 16, 2026, where the uoftasic organization's repository ic102-partner-sandbox-d stood in for your partner's. Owner and Copy the main branch only are as the page opened, and the name has been changed to end in -fork.

2. Clone your fork, and add your partner's repository as a second remote called `upstream`, so you can fetch what they merge:

```
cd /foss/designs/projects
git clone https://github.com/YOUR-USERNAME/ttsky-ic102-d-review-fork.git
cd ttsky-ic102-d-review-fork
git config user.name "Your Name"
git config user.email "ID+USERNAME@users.noreply.github.com"
git config credential.helper 'cache --timeout=28800'
git remote add upstream https://github.com/PARTNER-USERNAME/ttsky-ic102-d-review.
git
git remote -v
```

   You should see `origin` (your fork) and `upstream` (your partner's), each twice.

3. Pick a block. `python3 $L/claim.py --list` prints eight, from `parity` ("1 when an odd number of the eight ui_in pins are 1") to `toggle`. Pick a pin from `uo[0]` to `uo[6]`: your partner will claim the one after yours, and the optional Part 6 uses `uio[7]`. Then make a branch named after what it does, and put the block in your slot:

```
git switch -c claim-uo3-parity
python3 $L/claim.py parity uo3 --by YOUR-USERNAME
```

You should see:

```
Wrote src/slot_uo3.v and test/test_uo3.py (parity on uo[3]).
Now edit by hand: the line `uo[3]: ""` in info.yaml, and the `uo[3]` row of the
table in docs/info.md.
```

4. Make those two edits yourself. They're the claim that the datasheet and the next person read, and they're where conflicts happen. `git diff -- info.yaml docs/info.md` should show one changed line in each file, like these from a practice run:

```
-  uo[3]: ""
+  uo[3]: "parity of ui_in"
```

```
-| `uo[3]` | unclaimed | |
+| `uo[3]` | parity: 1 when an odd number of ui_in pins are 1 | student-b |
```

5. Run the tests, and read your slot's test in `test/test_uo3.py` while they run:

```
make -C "$TT" test PROJECT="$PWD"
```

You should see `test_uo3.test_uo3_parity PASS 650.00`, and `** TESTS=16 PASS=16 FAIL=0 SKIP=0`. An unclaimed slot's test checks that its pin stays 0.

6. Commit, push the branch to your fork, and open a pull request into your partner's `main`:

```
git add -A
git commit -m "Claim uo[3] for a parity block"
git push -u origin claim-uo3-parity
```

Open the link that `git push` prints, and check that the base repository is your partner's and the base branch is `main`. The description starts from the sandbox's template: fill in every section, including the test summary you just saw and whether an AI tool wrote any of it. Then tell your partner the pull request's number.

7. Before your partner does anything, predict whether this pull request could be merged once its checks pass.

*Answer: A1, in Answers at the end.*

You should see: your pull request open on your partner's repository, with its checks waiting for your partner to approve them.

# Part 3: A neighbouring claim

What happens to an open pull request when `main` changes under it?

1. Open the pull request your partner made on your repository. Its workflows haven't run, because your partner has never had a commit or pull request merged into your repository, and by default GitHub holds a first-time contributor's runs until you approve them. Their pull request runs code they wrote on your repository's runners, and it can change the workflow files themselves, so GitHub wants you to look at it first. GitHub's documentation gives the steps (approving workflow runs from forks): on the pull request's **Files changed** tab, read the changes, paying most attention to anything in `.github/workflows/`. Then press **Awaiting approval**, at the top right of the pull request, which opens the **Merge status** panel, and press **Approve workflows to run**.

GitHub decides whether to hold the runs each time workflows are started. It looks at who opened the pull request and at whose action started the runs, and with the default setting it doesn't ask about

anyone who has had a commit or pull request merged into the repository (workflows from forks). So the runs of your partner's later pushes can be held too, until their first pull request is merged. Approve them the same way whenever GitHub asks.

2. Look at **Files changed** and note the pin your partner claimed. Claim the one after it yourself, in your own clone, on a branch, the same way as in Part 2 (the two edits by hand, then the tests):

```
cd /foss/designs/projects/ttsky-ic102-d-review
git switch -c claim-uo4-majority
python3 $L/claim.py majority uo4 --by YOUR-USERNAME
```

   You should see `test_uo4.test_uo4_majority PASS 650.00` among sixteen passing tests. Commit, `git push -u origin claim-uo4-majority`, and open the pull request into your own `main`.

3. When its five checks are green, merge it. GitHub says "Review required" and "Merging is blocked", and offers you a box to tick, **Merge without waiting for requirements to be met (bypass rules)**, because the ruleset puts you on the bypass list "For pull requests only". Tick it and press **Bypass rules and merge**.

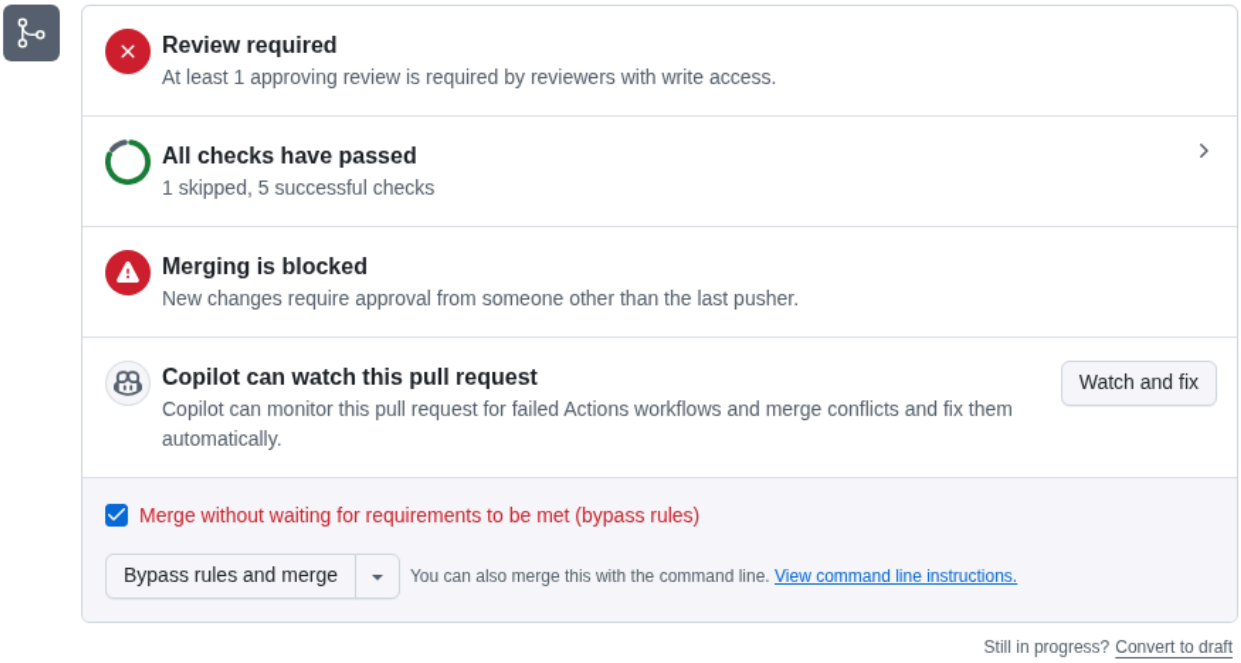



Figure 16.4: Merging your own claim with the bypass. The skipped check is viewer, which the sandbox runs only on main.

You can't approve your own pull request: GitHub greys out **Approve** and **Request changes** for its author. And your partner couldn't have approved it in a way that counts, because only approvals from people with write access count, and your partner has none.

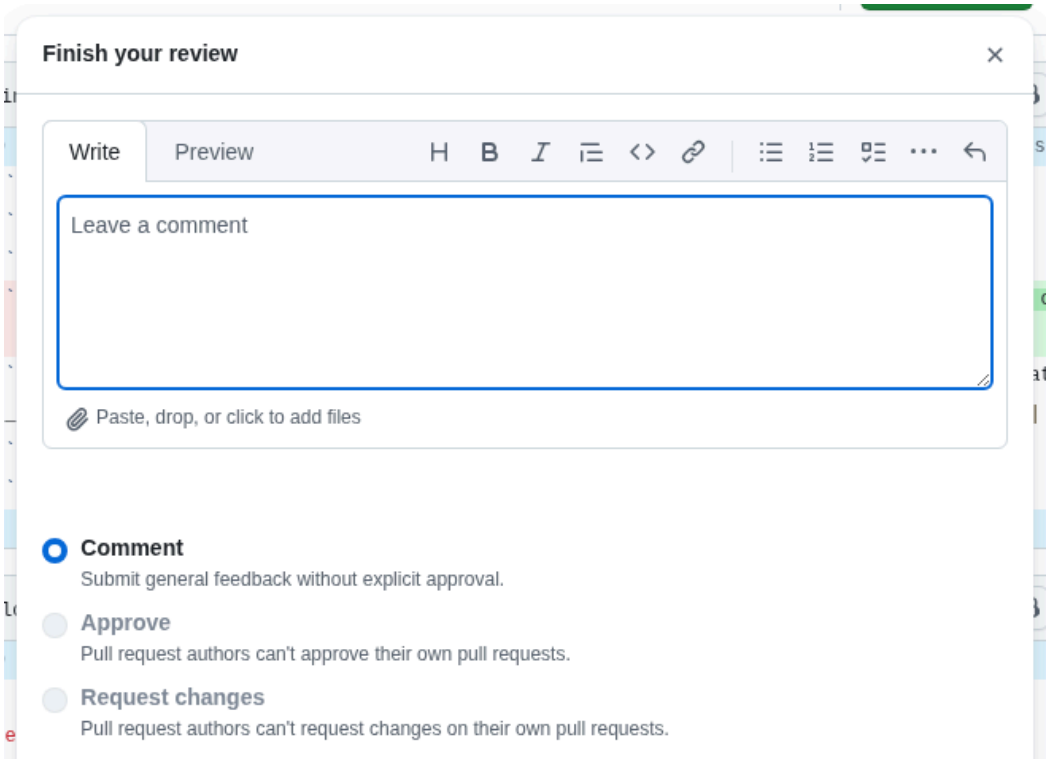



Figure 16.5: The review dialog on your own pull request, where only Comment is possible.

4. Predict what your partner's pull request says now, then look at it.

   *Answer: A2, in Answers at the end.*

You should see: your claim merged, `main`'s runs green, and a conflict on your partner's pull request.

## Part 4: Review your partner's pull request

What can you find in your partner's change that no check can?

1. Read the description first, then **Files changed**, then the checks. Go through the list in `CONTRIBUTING.md`: the test, the pin, only four files changed, the description, the checks.
2. The starter tests from `claim.py` check two input values each. Find out whether that's enough for your partner's block: check out their branch, break their block in a way that still passes those two values, and run the tests. GitHub keeps every pull request's latest commit in the repository it's aimed at, as `refs/pull/N/head`, so you can fetch it without adding your partner's fork as a remote. For a parity block, a version that looks at only half the inputs does that:

   ```
   git switch main
   git pull
   git fetch origin pull/N/head
   git switch --detach FETCH_HEAD
   sed -i 's/assign out = \^in;/assign out = ^in[3:0];/' src/slot_uo3.v
   make -C "$TT" test PROJECT="$PWD"
   ```

   You should see `* branch refs/pull/N/head -> FETCH_HEAD` from the fetch, and then `test_uo3.test_uo3_parity PASS 650.00`, although the block now ignores `ui_in[7:4]`. Put things back with `git checkout -- .` and `git switch main`.

   (Use your partner's slot and block in place of these. For another block, think about which inputs its two test values never exercise.)
3. Write your review. First add a comment on any line where you found something else: on **Files changed**, hover over the line, click the blue comment icon, type the comment, and press **Start a**

**review** (**Add review comment** for the ones after it). Those comments stay pending, and only you can see them, until you submit the review. Then press **Submit review**. It opens the **Finish your review** dialog pictured in Part 3, step 3, where **Approve** and **Request changes** were greyed out because the pull request was your own. Type a comment that says what to add and why: which input values, and which wrong block they would catch. Select **Request changes**, and press **Submit review** in the dialog (reviewing a pull request).

You should see: a review from you on your partner's pull request that requests changes.

## Part 5: Answer the review and resolve the conflict

What happens to an approval when you push again?

1. When your partner's review of your pull request arrives, make the change they asked for in your fork's clone: add the test cases, run the tests, commit, and push to the same branch. Reply to each comment and resolve it, because the ruleset won't merge with a conversation still open. In the practice run, the parity test's list of cases became `[(0x00, 0), (0x01, 1), (0x10, 1), (0x80, 1), (0xF0, 0), (0xFF, 0)]`, and the test reported `PASS 1050.00`. Ask your partner to look again. Your pull request gets no new runs yet, because GitHub doesn't run `pull_request` workflows on a pull request that has a conflict. (In the practice sandbox, this push started no runs at all.)
2. When your partner approves, the pull request still can't merge, because of the conflict from Part 3.

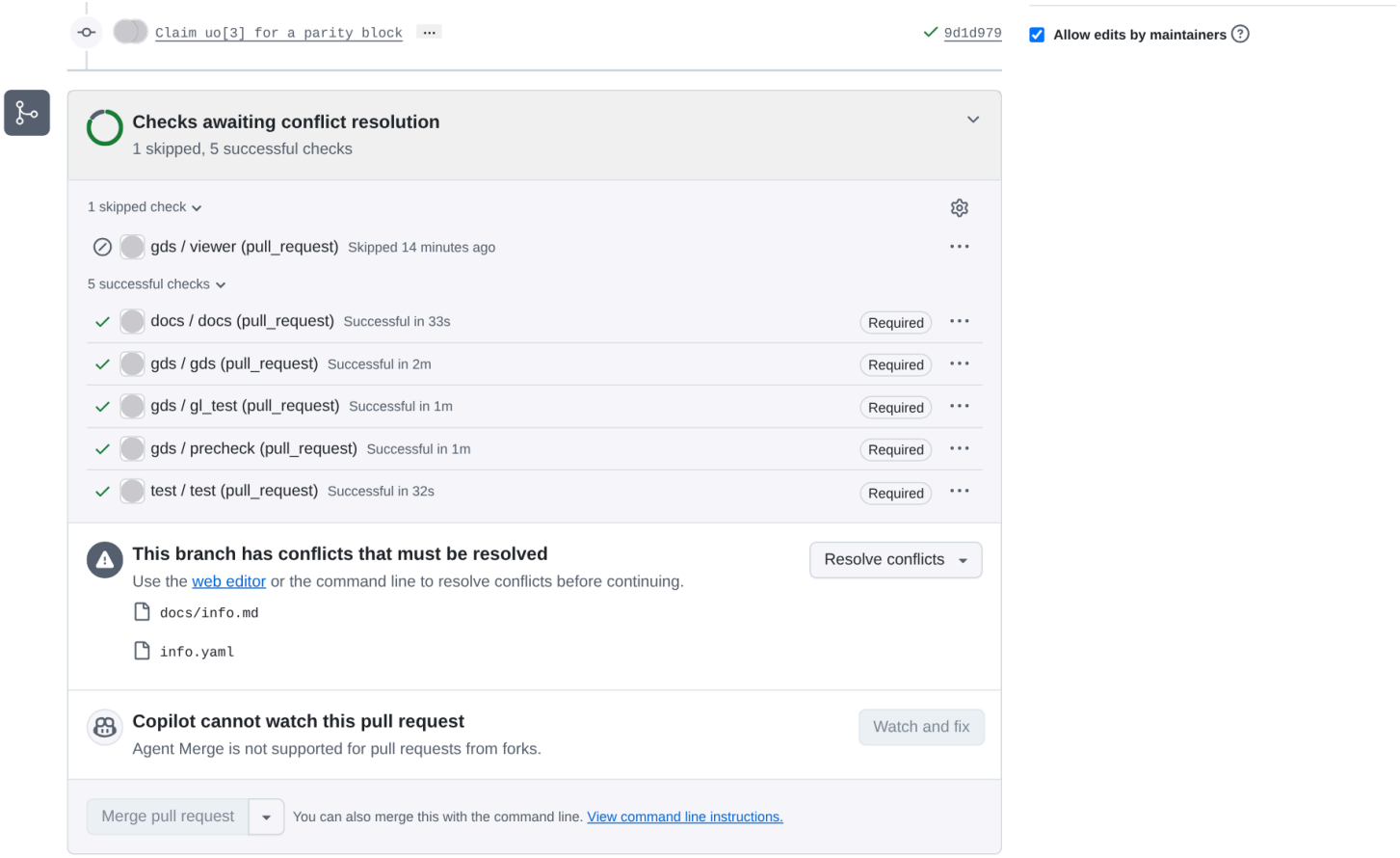



Figure 16.6: Your pull request on your partner's repository once their claim on the next pin has been merged, in the course's practice run, where the uoftasic organization's repository ic102-partner-sandbox-d stood in for your partner's. The box lists the two files that conflict, and Merge pull request is greyed out.

Bring your partner's `main` into your branch:

```
git fetch upstream
git merge upstream/main
```

In the practice run, where `uo[4]` had been merged into `main` and the branch claimed `uo[3]`, that printed:

```
Auto-merging docs/info.md
CONFLICT (content): Merge conflict in docs/info.md
Auto-merging info.yaml
CONFLICT (content): Merge conflict in info.yaml
Automatic merge failed; fix conflicts and then commit the result.
```

and `info.yaml` held:

```
<<<<<<< HEAD
  uo[3]: "parity of ui_in"
  uo[4]: ""
=======
  uo[3]: ""
  uo[4]: "majority of ui_in[2:0]"
>>>>>>> upstream/main
```

3. Resolve it: keep both claims (here, `uo[3]: "parity of ui_in"` and `uo[4]: "majority of ui_in[2:0]"`), delete the three marker lines, and do the same with the two rows in `docs/info.md`. Then finish the merge, test, and push:

   ```
   git add info.yaml docs/info.md
   git commit --no-edit
   make -C "$TT" test PROJECT="$PWD"
   git push
   ```

   You should see both claimed slots' tests pass, `test_uo3.test_uo3_parity PASS 1050.00` and `test_uo4.test_uo4_majority PASS 650.00`, with `TESTS=16 PASS=16`, and the conflict message gone from the pull request.

4. Before you look, predict whether your pull request is still approved.

   *Answer: A3, in Answers at the end.*

5. Your partner merges it, since only they can. Then run the check on your own work, with your pull request and the one you reviewed:

   ```
   python3 $L/check-pr.py PARTNER-USERNAME/ttsky-ic102-d-review#N YOUR-USERNAME/ttsky-ic102-d-review#N
   ```

   It reads both pull requests from GitHub's public API and checks two things: that yours was merged, with an approval from someone other than you that came before the merge, and that on the one you reviewed you requested changes and its author pushed at least one commit after that. It prints a `todo:` line for each one that isn't done yet, and `VERDICT: PASS` only when there are none. In the course's practice run, the `uoftasic` organization's repository `ic102-partner-sandbox-d` stood in for your partner's. One account opened both of its pull requests, #1 from a fork and #2 from a branch of the repository itself, so it could neither approve the first nor request changes on the second. There was no reviewed pull request to give the check, so it was run on #1 and #2 of that repository, and it printed:

   ```
   uoftasic/ic102-partner-sandbox-d #1: Claim uo[3] for a parity block
     author        vproHacks
     merged        2026-11-16T23:27:34Z
     approved by   nobody yet

   uoftasic/ic102-partner-sandbox-d #2: Claim uo[4] for a majority block
   ```

```
  author        vproHacks
  vproHacks hasn't requested changes on it

todo: #1 has no approval from someone else that came before the merge
todo: request a change on #2 (Part 4)
VERDICT: not yet
```

You should see: your pull request merged, and both pins' rows filled in on your partner's `main`.

## Part 6: A pull request written by an AI tool (optional)

About 15 minutes. The course has a pull request, "Claim uio[7]: heartbeat", whose code and description an AI coding tool wrote. Its description says so in its last line, and says that the course left problems in it on purpose. All sixteen tests pass on it with the course's tools (we checked), and that's part of the exercise. You each open it on the other's repository, and each review the one opened on your own.

1. In your fork's clone, put it on a branch of its own, push it, and open a pull request into your partner's `main` titled `Claim uio[7]: heartbeat`, with the text of `$L/ai-pr/pr-body.md` as its description:

   ```
   cd /foss/designs/projects/ttsky-ic102-d-review-fork
   python3 $L/ai-pr.py
   git push -u origin ai-claim-uio7-heartbeat
   ```

   You should see `Committed the AI-written pull request as ... on branch ai-claim-uio7-heartbeat.`

2. When your partner's arrives on your repository, approve its runs if GitHub asks, and review it as you reviewed their claim, but treat every statement in its description as a claim to check, as Pull requests and review suggests. Request changes, naming each problem you find with its file and line. Don't approve it and don't merge it; close it when you're done.

*Answer: A4, in Answers at the end.*

## Finishing

You're done when `check-pr.py` prints `VERDICT: PASS` for your pull request and the one you reviewed: yours was merged into your partner's repository after they approved it, and the review you wrote in yours asked for a change that its author then pushed.

Questions:

1. Your test passed before your partner asked for more cases, and it passes after. What did the new cases buy?

   *Answer: A5, in Answers at the end.*

2. You and your partner claimed `uo[3]` and `uo[4]` and got a conflict. Would claims on `uo[3]` and `uo[5]` have conflicted?

   *Answer: A6, in Answers at the end.*

Submit: the links to your pull request and to the review you wrote.

## Going further (optional)

Rules exist because without them a pull request merges whenever someone presses the button. Try `check-pr.py` on `sentry`'s pull request #33, the one with thirteen reviews in Pull requests and review, giving it as both of the pull requests it asks for:

```
python3 $L/check-pr.py uoftasic/sentry#33 uoftasic/sentry#33
```

Its first block reads as follows, with roles in place of the two people's usernames:

```
uoftasic/sentry #33: Event Decoder FSM
  author        <author>
  merged        2026-11-02T03:53:11Z
  approved by   <reviewer> at 2026-11-02T03:53:32Z after the merge
```

The approval came 21 seconds after the merge, because `sentry` has no ruleset and nothing made the merge wait for it.

The sandbox's workflows differ from the template's in their `on:` section and in one line of the `viewer` job. Compare `.github/workflows/gds.yaml` in the sandbox with your own `ttsky-ic102-d`'s, and work out why your partner's pull request from a fork got runs at all, why each workflow ran once per push to it, and why `viewer` shows as skipped on pull requests. Reading a workflow file has the parts you need.

Next on the Course path: read History as evidence, then do Lab 05: Find the commit that broke it.

# Answers

**A1. Part 2: Claim a pin in your partner's repository**

No. Your partner's ruleset needs an approval from someone with write access, which only your partner has, and only your partner can press the merge button in a repository you can't push to. The checks won't even start until your partner approves them. You've never had a commit or pull request merged into their repository, which makes you a first-time contributor there, and by default GitHub holds a first-time contributor's workflow runs until someone with write access approves them (workflows from forks).

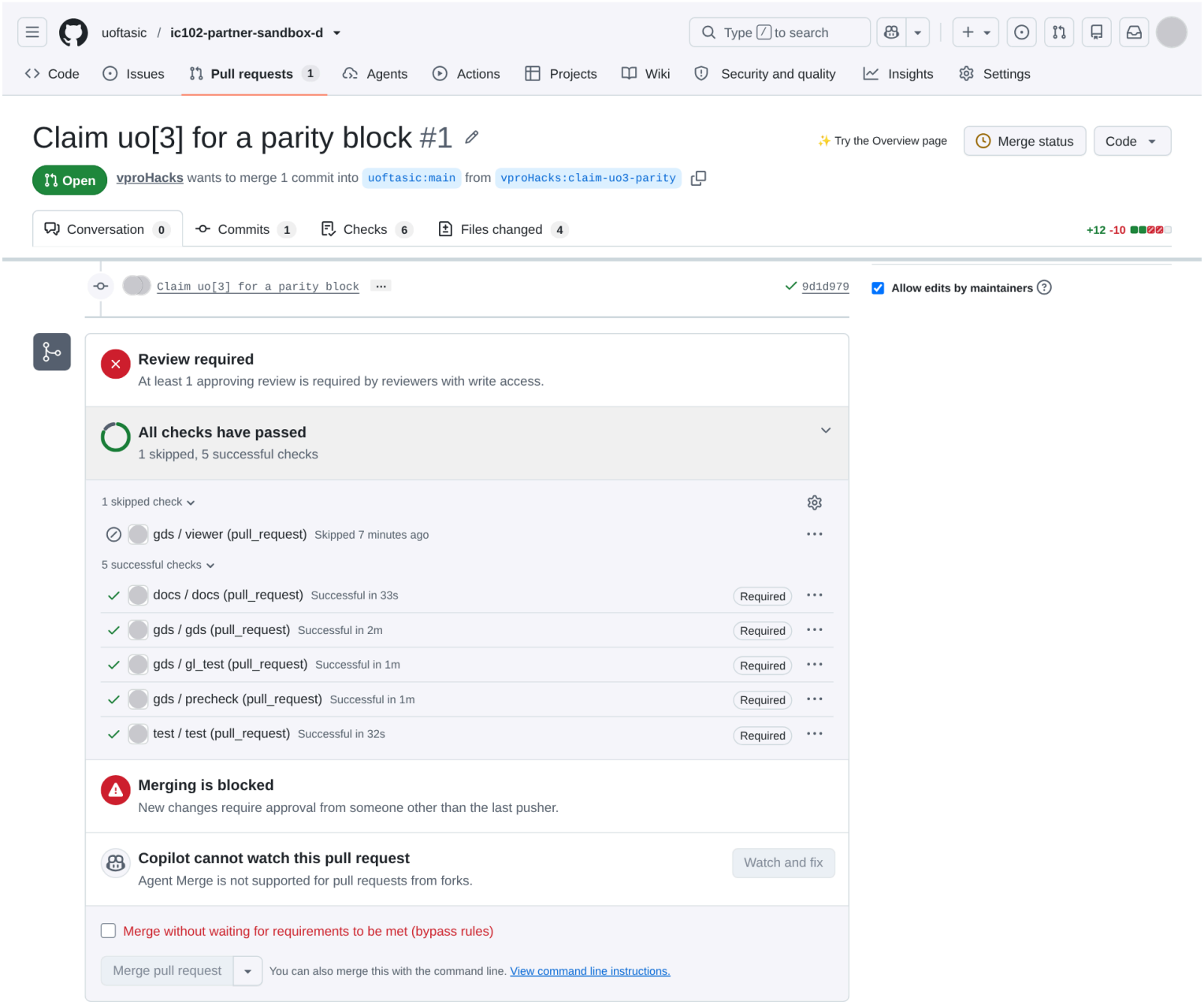



Figure 16.7: A pull request from a fork as its author sees it, in the course's practice run, where the uoftasic organization's repository ic102-partner-sandbox-d stood in for your partner's. The line under the title says where it goes, uoftasic:main, and where it comes from, the fork's claim-uo3-parity branch. The merge box says why it can't merge yet: "Review required" and "Merging is blocked".

The account that opened that pull request owns the organization, so its checks ran without waiting for anyone, and GitHub offered it the bypass box at the bottom. On your partner's repository your checks will wait for your partner, and there will be no bypass box, because you're not on their bypass list. The row "Copilot cannot watch this pull request" is there because Copilot's Agent Merge doesn't work on pull requests from forks, and you can ignore it.

**A2. Part 3: A neighbouring claim**

"This branch has conflicts that must be resolved", in `info.yaml` and `docs/info.md`. Your line and theirs are neighbours in both files, and git won't merge two changes to adjacent lines by itself, because it can't tell whether one depends on the other. Above it, the checks now say "Checks awaiting conflict resolution".

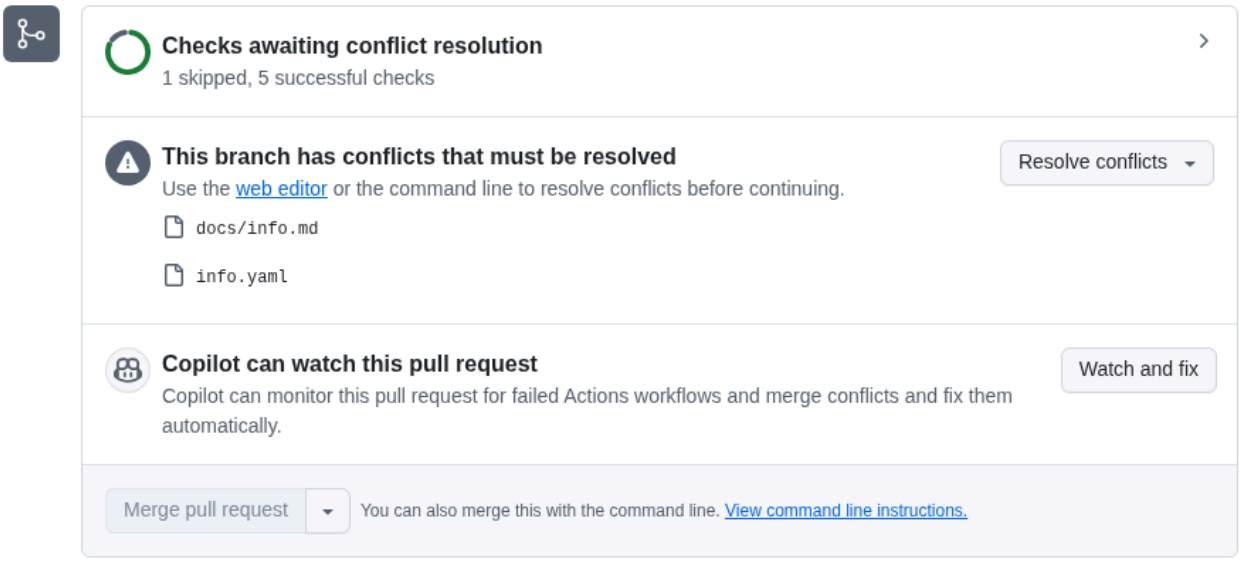



Figure 16.8: The merge box of a pull request whose branch conflicts with main, from the course's practice sandbox.

**A3. Part 5: Answer the review and resolve the conflict**

No. The ruleset dismisses an approval when new commits arrive, because the reviewer approved the commits they saw, and the merge commit you just pushed isn't one of them. It also requires the last push to be approved by someone other than the person who pushed it. Ask your partner to approve the runs and the pull request again.

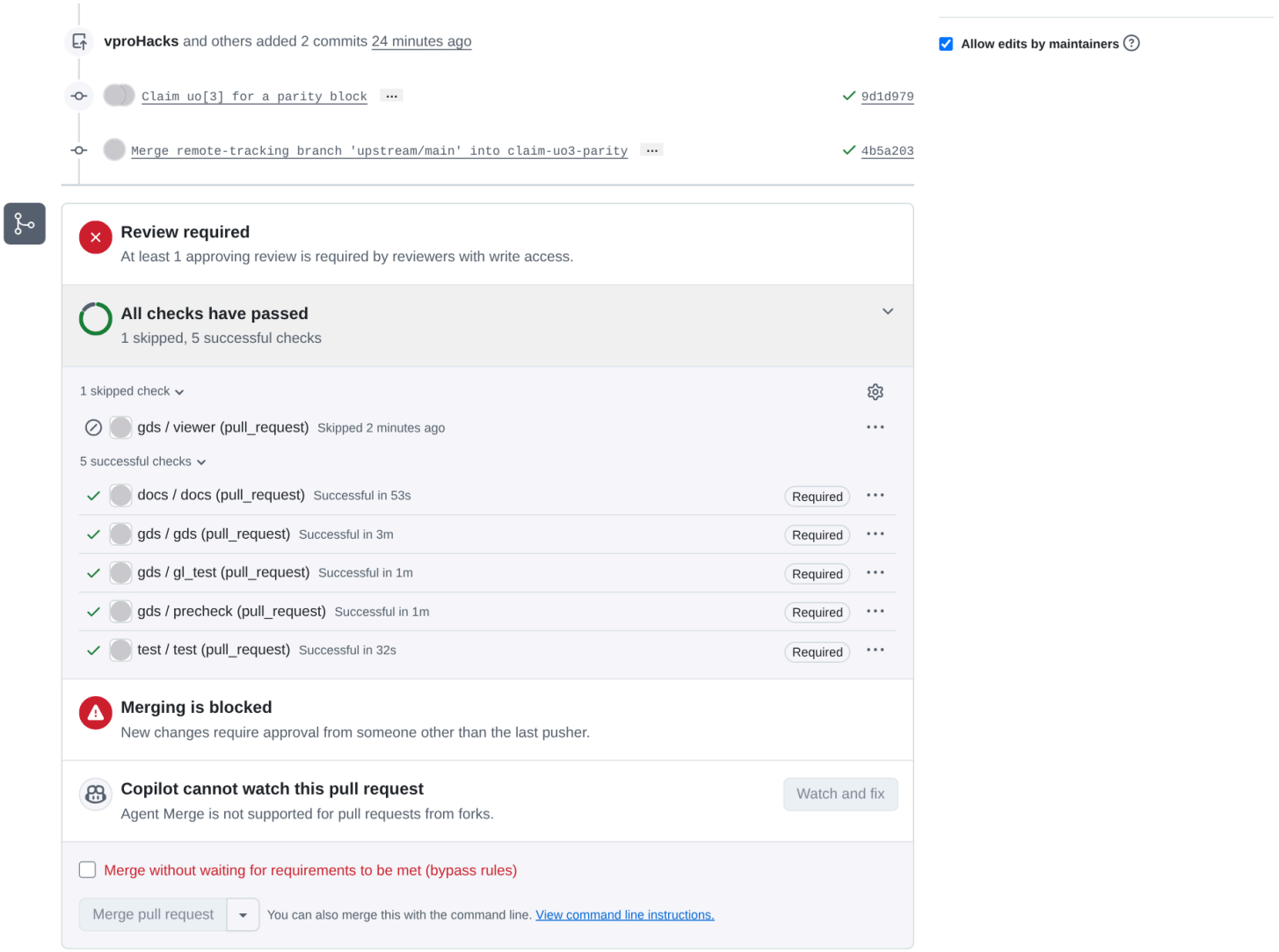



Figure 16.9: The pull request from the fork after its merge commit was pushed, in the course's practice run, where the uoftasic organization's repository ic102-partner-sandbox-d stood in for your partner's. The conflict is gone and the five required checks passed on the merge commit, but the box still says "Review required" and "Merging is blocked". Nobody had approved this pull request, so the picture can't show an approval being dismissed. The bypass box is there because the account that opened the pull request owns the organization.

**A4. Part 6: A pull request written by an AI tool (optional)**

There are three problems. `src/slot_uio7.v` sets `oe` to 0, so `uio[7]` stays an input and the heartbeat never leaves the chip, while `info.yaml` calls it an output. `test/test_uio7.py` has its only real assertion commented out ("flaky at gate level", which the pull request gives no evidence for), its last assertion (`in (0, 1)`) can't fail, and it never checks `uio_oe`. And the description says the tests pass "with cocotb 2.0.1", which is the workbench's version; the runner and the course's `make test` use 2.1.0.

**A5. Finishing**

They gave you a test that fails on blocks that are wrong in ways the old one missed, such as the half-parity block in Part 4. What a test is worth depends on which wrong designs it can catch, and the only way to find out is to try some, as you did when you reviewed your partner's pull request.

**A6. Finishing**

No. Git merges two changes cleanly when at least one unchanged line separates them, and the `uo[4]` line would sit between them. Neighbouring lines conflict because git can't tell whether the two edits depend on each other.

# V 5. History


**17 History as evidence . . . . . 115**
17 What the history keeps . . . . . 115
17 Reading a history . . . . . 115
17 Finding the commit that broke it . . . . . 116
17 Marking a commit . . . . . 118
17 Check yourself . . . . . 118
17 Answers . . . . . 118

**18 Lab 05: Find the commit that broke it . . . . . 121**
18 Before you start . . . . . 121
18 Part 1: Clone the history . . . . . 121
18 Part 2: The bisect script . . . . . 122
18 Part 3: The revert . . . . . 124
18 Part 4: The pull request . . . . . 125
18 Finishing . . . . . 126
18 Going further (optional) . . . . . 126
18 Answers . . . . . 126

# 17. History as evidence

This page answers: *when something that used to work is broken, how do you find the commit that broke it, and what can a project's history tell you, and not tell you, about what happened?*

A repository's history is a record of every change, in order, with a message and a date on each. Most of the time you only add to it. When something goes wrong, though, it's the best evidence you have, as long as you know what it keeps, what it loses, and how to search it. In Lab 05 you'll use it to find the commit that broke a project and undo that commit with a pull request.

## What the history keeps

Commits last as long as the repository does. Workflow runs don't. "By default, checks, workflow runs, commit statuses, and the artifacts and log files generated by workflows are retained for 90 days", and a public repository can set any period from 1 to 90 days (repository Actions settings).

`ttsky-verilog-tinyCGRA` shows what that means in practice. In May 2026 two of its `gds` runs built the layout, passed precheck and then failed at `gl_test`, the gate-level test. The run records still say so, but asking GitHub for the logs of one of those runs on Nov 15, 2026 gave this:

```
failed to get run log: HTTP 410: Server Error (https://api.github.com/repos/uoftasic/
ttsky-verilog-tinyCGRA/actions/runs/25332271003/logs)
```

HTTP 410 means "gone". The only account left of why the gate-level test failed is in the commit messages from those days, oldest first:

```
$ git log --oneline --reverse 21622f3..332cb3d
f42764b updated testbench (config mode mismatch)
7329711 commented config done (test)
3caef1b comment assert input in load_input()
96d508e comment assent input_reg.value in run_case()
17ef303 comment all user_project asserts
332cb3d commented asserts, no functional testing
```

followed, two days later, by the fix that was merged, which includes `b1896da` "removed config_done assert (unreliable in GLS)". These messages are useful because they say what each commit did, including that the tests had been switched off. So when a run fails and the failure matters, copy the first error line into the issue or pull request where you discuss it, because that text will outlive the log.

The history also only covers what was pushed to that repository. `simproc`'s description is "SimpleProcessor, taped out on TTSky25b", but the repository was created on Nov 10, 2025, all eight of its commits are from Nov 9 and 10, and GitHub has no workflow runs on record for it. The work that led to the taped-out design happened somewhere else, and this history doesn't show it. (Its merge messages also name the organization `ieeeuoft-asic`, because the team's organization has had another name.)

## Reading a history

Three questions come up again and again. Git answers the first two with a single command each, and for the third you can start with the Actions tab.

*Which commit last changed this line?* `git blame` prints each line of a file with the commit that last changed it. Here are lines 38 and 39 of `test/test.py` on `protocol-emulator`'s `main`, run in the course image (`-s` leaves out the author's name and the date):

```
$ git blame -s -L 38,39 test/test.py
a5f14750 38)     # Temporarily comment out this assert
a5f14750 39)     # assert dut.uo_out.value == 50
```

*Which commits added or removed lines like this?* `git log -G` searches the changes themselves, not the messages:

```
$ git log --oneline -G "^ +assert dut.uo_out" -- test/test.py
a5f1475 chore: temporarily remove assertion from test.py
04c0e8a Initial commit
```

The assertion arrived with the template in the first commit and left in `a5f1475`. Since then the test on `main` has passed without checking anything. Run on `main` in the course image, it ends with this table:

```
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*******************************************************************************************
** test.test_project              PASS      100000.00           0.00   59116335.45  **
*******************************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0             100000.00           0.00   40133039.90  **
```

*When did it start failing?* Here the Actions tab is a history of its own, because every push has a run. Here are the runs of `protocol-emulator`'s `test` workflow, one per pushed commit, oldest first:

| Commit | Pushed | `test` |
|---|---|---|
| `04c0e8a` Initial commit | Oct 18 | success |
| `b976406` docs: update info.yaml | Oct 24 | failure |
| `8f7bc9f` docs: update description | Oct 24 | failure |
| `95bdb65` chore: update pinout and docs | Oct 24 | failure |
| `f19bcb7` update info.md | Nov 1 | failure |
| `a5f1475` chore: temporarily remove assertion from test.py | Nov 1 | failure |
| `7c56c54` chore: updated module name in testbench | Nov 1 | success |

The test broke at `b976406` and was fixed at `7c56c54`, and deleting the assertion in between made no difference, because the failure had nothing to do with it. This works only while the runs are kept, and only when every commit was pushed on its own. Usually several commits arrive in one push, and only the last of them is tested, which is where bisecting comes in.

## Finding the commit that broke it

`git bisect` finds the first bad commit by binary search. You give it one commit you know is good and one you know is bad. It checks out the commit halfway between them and asks you whether that one is good or bad, and each answer halves the number of commits that could be to blame:

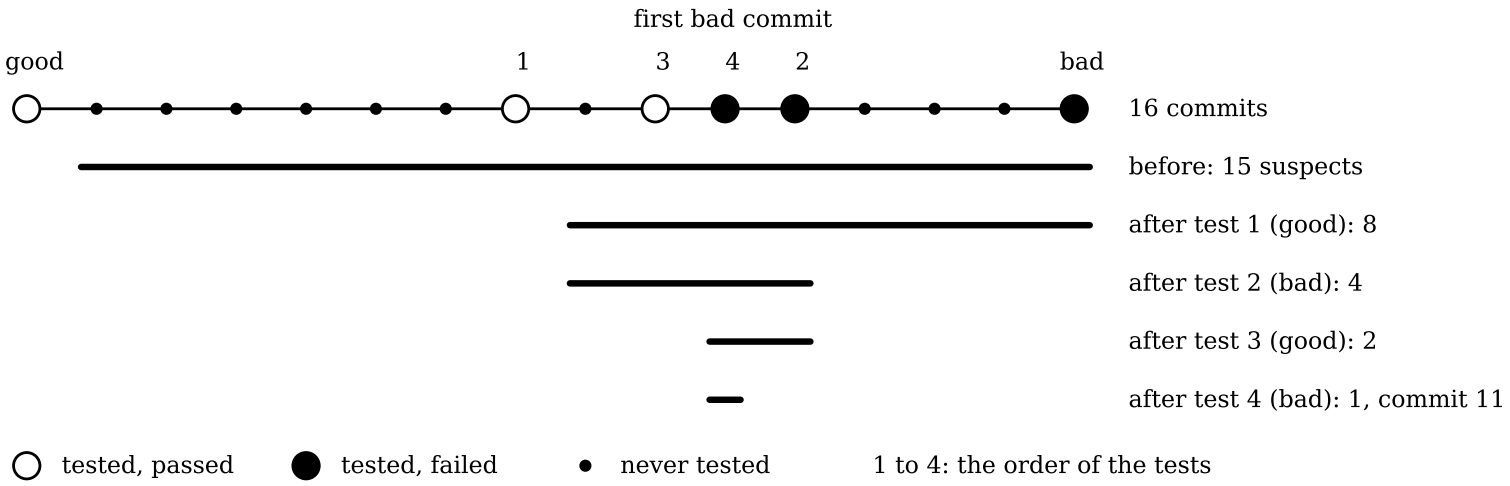



Figure 17.1: git bisect on 16 commits, where commit 1 is known to be good and commit 16 bad. Test 1 checks commit 8, which passes; test 2 checks 12, which fails; test 3 checks 10, which passes; test 4 checks 11, which fails. The suspects shrink from 15 to 8, 4, 2 and finally 1, commit 11, the first bad commit.

The number of tests grows only with the logarithm of the number of commits: 16 commits take 4 tests, and a history of 40 commits takes 6, because $2^6 = 64 \geq 40$.

You don't have to answer git's questions by hand. `git bisect run` takes a script, runs it at each commit, and reads the answer from the script's exit status. The git manual sets the rules: the script should "exit with code 0 if the current source code is good/old, and exit with a code between 1 and 127 (inclusive), except 125, if the current source code is bad/new", and "The special exit code 125 should be used when the current source code cannot be tested" (git-bisect).

Here is a script for the `protocol-emulator` history above. It runs the cocotb test and decides from the results file, not from `make`'s exit status. With cocotb 2.x, `make` does exit non-zero when a test fails, but it exits with the same status when the design doesn't compile and the simulation never starts, and the results file is what tells those two cases apart. (The comment in the template's workflow, that `make` "will return success even if the tests fail", dates from cocotb 1.x.)

```
#!/bin/bash
# For git bisect run: 0 = test passes (good), 1 = test fails (bad).
cd test || exit 125
rm -rf sim_build results.xml
make > /tmp/bisect-make.log 2>&1
if [ ! -f results.xml ]; then
    exit 1          # the simulation didn't run at all
fi
if grep -q "<failure" results.xml; then
    exit 1          # a test ran and failed
fi
exit 0
```

And here is the bisect, between the good first commit and the bad `a5f1475`, run in the course image. We've left out the author line of the result:

```
$ git bisect start a5f1475 04c0e8a
Bisecting: 2 revisions left to test after this (roughly 1 step)
[8f7bc9f7c8f48dd62fbda4dbf4c737dd515ae28d] docs: update description
$ git bisect run ../bisect-test.sh
running '../bisect-test.sh'
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[b976406293b84f285ec8d36fb7b8fd09c3f19037] docs: update info.yaml
running '../bisect-test.sh'
```

```
b976406293b84f285ec8d36fb7b8fd09c3f19037 is the first bad commit
commit b976406293b84f285ec8d36fb7b8fd09c3f19037
Date:   Sat Oct 24 15:07:01 2026 -0400

    docs: update info.yaml

 info.yaml     | 10 +++++-----
 src/project.v |  2 +-
 2 files changed, 6 insertions(+), 6 deletions(-)
bisect found first bad commit
```

Two tests were enough for five suspects. The answer agrees with the Actions history, and it shows something the commit message doesn't: a commit described as `docs: update info.yaml` also changed `src/project.v`, where it renamed the top module, and that's what broke the testbench. Bisect never looks at commit messages, only at whether the test passes, so a misleading message can't send it the wrong way.

The script above treats "the simulation didn't run" as bad, which is right for this history, because a simulation that can't start is exactly the failure being hunted. In a longer history there will be commits that can't be tested for some unrelated reason, such as a half-finished file that doesn't compile, and for those the script should exit with 125 so that git skips them instead of blaming them. Deciding which is which is the main thing you'll think about when you write your own script in Lab 05.

## Marking a commit

Bisecting needs a commit you know is good, and the easiest way to have one is to mark it when you make it. A **tag** is a permanent name for one commit, such as `v1.0` or `ready-for-review`, and on GitHub a **release** adds notes and downloads to a tag. When a commit is the one you'd stand behind, with every workflow green, tag it, and anyone can later say "it worked at that tag" and check.

A tag in a repository you control is as fixed as you keep it. A tag in someone else's repository may move: Tiny Tapeout moves its action tags with a force push when it updates a release, as Reading a workflow file describes. When you need to name an exact version as evidence, give the commit hash, which can't change.

## Check yourself

1. A teammate asks why a gate-level test failed in a run from seven months ago. Where can you still look, and where can't you?

*Answer: A1, in Answers at the end.*

2. How many tests does `git bisect` need, at most, to find the first bad commit among 64 suspects?

*Answer: A2, in Answers at the end.*

3. A bisect script for a cocotb project runs `make` and exits with `make`'s exit status. What will bisect conclude, and why?

*Answer: A3, in Answers at the end.*

4. In the middle of the range, one commit doesn't compile because a file was half-finished, and the bug you're hunting came later. What should your script exit with at that commit?

*Answer: A4, in Answers at the end.*

Next: Lab 05, where you find the commit that broke a project and revert it in a pull request.

## Answers

**A1. Check yourself**

The run's logs and artifacts are gone, since GitHub keeps them for at most 90 days, so asking for them gives HTTP 410. What's left is anything written into the repository: commit messages, and the text of issues and pull requests, which is why the first error line belongs in one of those.

**A2. Check yourself**

Six, because each test halves the suspects and $2^6 = 64$.

**A3. Check yourself**

With cocotb 2.x, `make` exits non-zero both when a test fails and when the design doesn't compile, so the script calls every commit that doesn't compile "bad" too. If a half-finished commit that doesn't compile

comes before the real bug, bisect names that commit as the first bad one, for a reason that has nothing to do with the bug. The script has to look at the results file: no results file means the commit couldn't be tested (exit 125, unless not compiling is the failure you're hunting), and a failure in it means bad. (With cocotb 1.x, whose `make` exited with 0 on a failing test, the same script would have called every commit that compiles "good".)

**A4. Check yourself**

125, which tells git the commit can't be tested, so git skips it and tests a neighbour instead. Exiting with 1 would wrongly mark it as the first bad commit.

# 18. Lab 05: Find the commit that broke it

You'll take over the history of a small Tiny Tapeout project, an LED dimmer that three teammates wrote over five weeks. Its test passed at the commit they tagged `v1.0`, and it fails now, 27 commits later. You'll write a script that tells `git bisect run` correctly whether each commit is good, bad or can't be tested, use it to find the commit that broke the test, revert that commit on a branch, and get the revert through every workflow in a pull request on a repository of your own.

Package: `labs/lab-05-find-the-commit-that-broke-it/`

## Before you start

- Read History as evidence.
- Finish Lab 03. This lab uses the test job's tools you installed in Lab 01, Part 1, your access token from Lab 01, Part 2, and the way you turned on Pages in Lab 01, Part 5.
- Every command runs in a terminal on the workbench desktop, where `git bisect run` can call the same `make test` as Lab 03. In each new terminal, run these first; the steps use `$L` for this lab's package and `$TT` for the test job's tools:

```
. /foss/designs/common/.designinit
L=/foss/designs/modules/ic102-d/labs/lab-05-find-the-commit-that-broke-it
TT=/foss/designs/modules/ic102-d/labs/common/tt-tools
```

The teammates are made up, and so is their project, which the course builds from Tiny Tapeout's template with a script. Every commit is an ordinary one, though: features, tests, documentation, two half-finished edits, a new file left out of the test's source list, and one tidy-up that went wrong.

This lab takes about an hour, including the 5 to 10 minutes each push takes on GitHub's runners.

## Part 1: Clone the history

What does the test say now, and what did it say at `v1.0`?

1. Clone the history. The course ships it as a git bundle, a single file that holds every commit and tag, and `git clone` reads it like a repository:

   ```
   mkdir -p /foss/designs/projects && cd /foss/designs/projects
   git clone $L/data/history.bundle ttsky-ic102-d-bisect
   cd ttsky-ic102-d-bisect
   git config user.name "Your Name"
   git config user.email "ID+USERNAME@users.noreply.github.com"
   git config credential.helper 'cache --timeout=28800'
   ```

   Use the same no-reply address as in Lab 01. You should see `Cloning into 'ttsky-ic102-d-bisect'...`. The folder is named after the repository you'll push it to in Part 4.

2. Look at the history:

   ```
   git log --oneline | wc -l
   git log --oneline v1.0..main | wc -l
   git log --oneline -3
   ```

   You should see `36`, then `27`, then:

```
00c3d87 test: name the LED's pin in failure messages
35f4b2e info.yaml: a longer description
14d59b6 test: run fade mode for 100 steps
```

The hashes in your clone are the same as the ones on this page, because the course built each commit with a fixed author, date and message.

3. Run the test on `main`, as the test job would:

```
make -C "$TT" test PROJECT="$PWD"
```

You should see three of seven tests fail:

```
AssertionError: with duty 0 the LED (uo_out[7]) was on for 8 of 2048 cycles, expected 0
AssertionError: with duty 64 the LED (uo_out[7]) was on for 260 of 1024 cycles,
expected 256
AssertionError: with duty 255 the LED (uo_out[7]) was on for 1024 of 1024 cycles,
expected 1020
```

and `** TESTS=7 PASS=4 FAIL=3 SKIP=0` in the summary table, followed by `make: *** [Makefile:42: test] Error 2`.

4. Before you run it, write down whether the test passes at `v1.0`. Then check:

```
git switch --detach v1.0
make -C "$TT" test PROJECT="$PWD"
git switch main
```

You should see `** TESTS=3 PASS=3 FAIL=0 SKIP=0`. The test had only three cases then, all of them about the PWM, and all three pass.

5. Somewhere in the 27 commits between `v1.0` and `main`, the PWM broke. How many tests will `git bisect` need to find the commit?

*Answer: A1, in Answers at the end.*

You should see: three PWM failures on `main`, and a pass at `v1.0`.

## Part 2: The bisect script

What should the script say about a commit whose design doesn't compile?

1. Copy the starting script to the top of your clone, and read it:

```
cp $L/bisect-template.sh bisect.sh
```

It runs the test job's steps with `run-test.sh`, the script behind Lab 03's `make test`, and then always exits 0. The comment at the top lists the three answers `git bisect run` understands: 0 for good, 1 to 127 except 125 for bad, and 125 for "can't test this one". `bisect.sh` isn't tracked by git, so checking out old commits leaves it alone.

2. Why can't the script simply exit with `run-test.sh`'s status? Try it on `main`, where a test fails, and on `cbba6a3`, a commit from the middle of the history:

```
for c in main cbba6a3; do
  git checkout -q $c; rm -f test/results.xml
  "$TT/run-test.sh" . > /dev/null 2>&1
  echo "$c: exit $?; $(ls test/results.xml 2>&1)"
done
git checkout -q main
```

You should see:

```
main: exit 2; test/results.xml
cbba6a3: exit 2; ls: cannot access 'test/results.xml': No such file or directory
```

Both exit with 2. With cocotb 2.x, `make` runs cocotb's own results check at the end of the simulation, so it stops with `Error 2` when a test fails, and it stops with the same status when the design doesn't compile. The difference shows in `test/results.xml`: cocotb writes it whenever the simulation runs, and never when the compile fails. (This is also why the script deletes `results.xml` before each run. An old one, left from the previous commit, would make a commit that never ran look as if it had.)

3. Replace the last line of `bisect.sh` (`exit 0`, under the TODO) with the rule the History as evidence script uses for `protocol-emulator`, where a missing `results.xml` counts as bad:

```
if [ ! -f test/results.xml ]; then
    exit 1
fi
if grep -q "<failure" test/results.xml; then
    exit 1
fi
exit 0
```

4. Run the bisect between the last known good commit and `main`:

```
git bisect start main v1.0
git bisect run ./bisect.sh
```

It takes about ten seconds. You should see it test five commits and end with:

```
85133d69064177106e14d05cf0e7b97a7add19f5 is the first bad commit
commit 85133d69064177106e14d05cf0e7b97a7add19f5
Author: Sam Rivera <sam.rivera@example.com>
Date:   Thu Oct 15 20:25:00 2026 -0400

    Move the seven-segment table into seg7_table.v

 info.yaml        |  1 +
 src/seg7.v       | 24 +++----------------------
 src/seg7_table.v | 35 ++++++++++++++++++++++++++++++++++++
 3 files changed, 39 insertions(+), 21 deletions(-)
 create mode 100644 src/seg7_table.v
bisect found first bad commit
```

5. Is `85133d6` the commit that broke the PWM? The script kept a log of each commit it tested; read this one, and the next commit's message:

```
grep error /tmp/bisect-85133d6.log
git log --oneline -1 ae2c625
```

*Answer: A2, in Answers at the end.*

6. End this bisect, change the first `exit 1` in `bisect.sh` (the one under `if [ ! -f test/results.xml ]`) to `exit 125`, and run it again:

```
git bisect reset
git bisect start main v1.0
git bisect run ./bisect.sh
git bisect log
```

You should see the first commit it checks out, `cbba6a3 WIP: preset brightness levels on uio_in[2:1]`, reported as skipped in the log, five commits tested in all, and a different first bad commit. Write down its hash and message before you open the answer.

*Answer: the bisect, from git bisect log: A3, in Answers at the end.*

7. Read the commit, then end the bisect:

```
git show <the first bad commit>
git bisect reset
```

Its message says it tidied up the comparator and its comments. Which line changed what the circuit does, and why does that turn the LED on for exactly 8 of 2048 cycles at duty 0?

*Answer: A4, in Answers at the end.*

8. Check your script against three commits whose answers are known:

```
python3 $L/check.py --offline
```

You should see, after a few seconds (5 s in our run):

```
Your bisect.sh, run at three commits
  4e04d77  v1.0, test passes            exit 0    ok
  cbba6a3  preset WIP, doesn't compile  exit 125  ok
  00c3d87  last commit, test fails      exit 1    ok
```

followed by `no commit made by git revert on this branch yet`. You'll make that commit in Part 3.

You should see: `git bisect` name a commit whose diff changes the PWM, and three `ok` lines from the check.

## Part 3: The revert

Can you undo that one commit without undoing the seven commits that came after it?

1. Before you run anything, look at what the seven later commits touched (`git log --stat --oneline <the first bad commit>..main`). Write down whether you expect `git revert` to apply cleanly.

*Answer: A5, in Answers at the end.*

2. Make a branch for the fix and revert the commit on it:

```
git switch -c revert-pwm-compare main
git revert --no-edit <the first bad commit>
git log -1 --format=%B
```

You should see `1 file changed, 4 insertions(+), 4 deletions(-)` and a message that ends with `This reverts commit <its full hash>.` The new commit's own hash will differ from anyone else's, because it carries your name and the time.

3. Run the test again:

```
make -C "$TT" test PROJECT="$PWD"
```

You should see all seven pass, among them `test.test_duty_0_is_off  PASS  205250.00` and `** TESTS=7 PASS=7 FAIL=0 SKIP=0`. The fade, preset and display tests added after the bad commit still pass, because the revert undid only the comparator.

4. Commit your script on the branch, so that the pull request carries the evidence:

```
git add bisect.sh
git commit -m "Add the bisect script that found the PWM regression"
```

You should see: seven passing tests on your branch, with the later features still in place.

## Part 4: The pull request

Does GitHub's runner agree with your workbench?

1. On GitHub, create a new public repository called `ttsky-ic102-d-bisect` in your own account. Leave it empty, as in Lab 04, Part 1: no README, no `.gitignore` and no licence, because the history you push already has its own first commit.
2. Point your clone at it and push `main` and the tag. The push sends files under `.github/workflows/`, which GitHub accepts only from a token with the Workflows permission you gave yours in Lab 01, Part 2:

   ```
   git remote set-url origin https://github.com/YOUR-USERNAME/ttsky-ic102-d-bisect.git
   git push -u origin main --tags
   ```

3. Before you look at the Actions tab, predict the result of each job on `main`: `docs`, `test`, `gds`, `precheck`, `gl_test` and `viewer`.

   *Answer: A6, in Answers at the end.*

   You'll also see the three workflows run a second time, on `v1.0`. Tiny Tapeout's workflows run on every push, and pushing a tag counts as one. On `v1.0` only `viewer` fails, with the same

   404.

4. Turn Pages on for `ttsky-ic102-d-bisect` the way you did in Lab 01, Part 5. You don't need **Run workflow** this time, because the push in the next step starts a new run.
5. Push your branch and open a pull request from `revert-pwm-compare` into `main`:

   ```
   git push -u origin revert-pwm-compare
   ```

   GitHub prints a link for creating the pull request. In the description, name the commit you reverted and why it was wrong, say how you found it, and paste the output of `git bisect log` from Part 2. If an AI tool helped you write the script or the description, say so and say what you checked yourself, as Pull requests and review asks.
6. Wait for the runs on your branch to finish (about 5 to 10 minutes), then run the full check:

   ```
   python3 $L/check.py
   ```

   It repeats Part 2's three runs and then reads your branch's latest commit from GitHub. In the course's practice repository it printed:

   ```
   Checking /foss/designs/projects/ttsky-ic102-d-bisect

   Your bisect.sh, run at three commits
     4e04d77  v1.0, test passes            exit 0    ok
     cbba6a3  preset WIP, doesn't compile  exit 125  ok
     00c3d87  last commit, test fails      exit 1    ok

   Your revert
     reverts d18e316, the commit that broke the test

   GitHub: vproHacks/ic102-test-d at 6b2ae2a
     docs      success
     test      success
     gds       success
     precheck  success
     gl_test   success
     viewer    success

   VERDICT: PASS
   ```

`viewer` passes on a branch too: it publishes the branch's layout to your project's page.

7. Merge the pull request. The repository is yours and has no rules, so you can merge it yourself. The merge is a push to `main`, so `main`'s workflows run again, now with the revert in them and Pages on. In the course's practice repository all six were green 5 min 45 s after the merge, `viewer` included.

You should see: all six jobs green on the pull request, `VERDICT: PASS`, and all six green on `main` after the merge.

## Finishing

You're done when `check.py`, run on your `revert-pwm-compare` branch, prints `VERDICT: PASS` (your script gives the right answer at all three commits, your branch reverts the commit that broke the test, and all six jobs passed on the branch's latest commit), and every job is green on `main` after you've merged the pull request.

Questions:

1. Your script and the `protocol-emulator` script in History as evidence give opposite answers for a commit that doesn't compile. Can both be right?

   *Answer: A7, in Answers at the end.*

2. The `v1.0` tag saved you a step. What would you have had to do without it?

   *Answer: A8, in Answers at the end.*

3. The bad commit was made on Oct 25, and seven more commits followed before anyone noticed. In the team's real repositories the test workflow runs on every push. What does that suggest the made-up team was doing wrong?

   *Answer: A9, in Answers at the end.*

Submit: the link to your pull request.

## Going further (optional)

When you already know which line is wrong, git can find the commit that touched it without running anything. `git log -G` lists the commits whose diffs add or remove a line matching a pattern:

```
git log --oneline -G "phase <" -- src/pwm.v
```

On the `revert-pwm-compare` branch of the course's practice repository it printed:

```
840c251 Revert "pwm: tidy up the comparator and its comments"
d18e316 pwm: tidy up the comparator and its comments
c6218cc Add an 8-bit PWM
```

From the bottom up, those are the commit that wrote the comparison, the commit that changed it, and the revert that changed it back, which has a hash of its own in your clone. `git bisect` is for the more common case, where you know only that the test fails.

This is the last row of the Course path.

## Answers

**A1. Part 1: Clone the history**

About five. Each test halves the number of suspects, and $2^5 = 32 \geq 27$. One commit that can't be tested may add a test or two, as you'll see in Part 2.

**A2. Part 2: The bisect script**

No. The log has a line ending in `src/seg7.v:15: error: Unknown module type: seg7_table`: the commit added a new file to `info.yaml` but not to `PROJECT_SOURCES` in `test/Makefile`, so the design didn't compile and the test never ran. The very next commit, `ae2c625 test/Makefile: add seg7_table.v to PROJECT_SOURCES`, fixed that, and nothing about the PWM changed. The script reported a commit that couldn't be tested at all as bad, and `git bisect` goes by whatever the script reports.

The same rule was right for `protocol-emulator`, because there the failure being hunted *was* a compile error (a module the testbench could no longer find). Here the failure being hunted is a wrong value from the PWM, and a commit that doesn't compile tells you nothing about it.

**A3. Part 2: The bisect script (Answer: the bisect, from git bisect log)**

```
# bad: [00c3d87894d0c8bce7a127e029cd9f6cbbb36da2] test: name the LED's pin in failure
messages
# good: [4e04d775e153a5c2526f9687f9ef83a185f62b7d] docs: say how the dimmer works
git bisect start 'main' 'v1.0'
# skip: [cbba6a3831a677c6552cf449fd3ae56728bf2565] WIP: preset brightness levels on
uio_in[2:1]
git bisect skip cbba6a3831a677c6552cf449fd3ae56728bf2565
# good: [921b40562b8fbf021b317bc475c61ee5f525c1b7] presets: finish the case statement
git bisect good 921b40562b8fbf021b317bc475c61ee5f525c1b7
# bad: [d18e3169a759beec08f58e0f6327310941a607de] pwm: tidy up the comparator and
its comments
git bisect bad d18e3169a759beec08f58e0f6327310941a607de
# good: [3ad66712b887c83d196dd5e4038357ec66c468b8] test: the three presets
git bisect good 3ad66712b887c83d196dd5e4038357ec66c468b8
# good: [60651505fa672660194fbb3cb412ff63d39e56f9] test: run the testbench clock at 10
MHz, to match clock_hz
git bisect good 60651505fa672660194fbb3cb412ff63d39e56f9
# first bad commit: [d18e3169a759beec08f58e0f6327310941a607de] pwm: tidy up the
comparator and its comments
```

**A4. Part 2: The bisect script**

`assign out = (phase < duty);` became `assign out = (phase <= duty);`. The phase counts 0 to 255, once per PWM period, so with duty 0 the new comparison is true for one step in every period, when the phase is 0. A step lasts four clock cycles (the prescaler divides by 4), and the test watches two periods: $2 \times 4 = 8$ cycles. The same extra step gives 260 instead of 256 cycles at duty 64, and turns duty 255 into always on. The four changed comment lines around it make the diff look like a comment edit, which is how it got past its author.

**A5. Part 3: The revert**

It should. None of the later commits touch `src/pwm.v`, so the lines the revert puts back haven't moved, and git can apply the reverse of the old diff directly.

**A6. Part 4: The pull request**

`docs`, `gds` and `precheck` should pass, because the files they check are fine. `test` fails on the three PWM cases, and so does `gl_test`, which runs the same test on the gate-level netlist. `viewer` fails with the 404 from Lab 01, because a new repository has Pages off.

That's what the course's practice repository got. `test` and `gl_test` both stopped at `AssertionError: with duty 0 the LED (uo_out[7]) was on for 8 of 2048 cycles, expected 0`, the first of the three failures you saw in Part 1, and `viewer` at `Error: Failed to create deployment (status: 404)`.

**A7. Finishing**

Yes, because they hunt different failures. In `protocol-emulator` the test broke *because* the testbench stopped compiling, so a commit that doesn't compile shows the bug. In the dimmer the bug is a wrong value from a design that compiles, so a commit that doesn't compile can't show it either way, and 125 tells git to look at a neighbour instead. So decide what you're hunting before you decide what the script says about a commit that doesn't compile.

**A8. Finishing**

Find a good commit yourself before `git bisect start` could begin, by checking out older commits and running the test until one passed. A tag on a commit that everyone saw green is a known-good starting point for anyone who comes later, which is why History as evidence suggests marking one.

**A9. Finishing**

They weren't looking at their runs. On GitHub, the `test` workflow would have gone red on the push that included `d18e316`, and on every push after it. A red cross only stays red for a week when nobody is reading it, and that's how `hart-1`'s `docs` and `gds` stayed red for four weeks.