IC102-A — Working in Tiny Tapeout Repos (Analog)

Make an analog Tiny Tapeout project from the template on your own GitHub account and get its layout through precheck.

UofT ASIC Team

Contents

1 IC102-A — Working in Tiny Tapeout Repos (Analog) 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. Runs that fail

6 What a cloud runner is 39

7 Reading a workflow file 43

8 The analog workflows 57

9 Lab 02: Read the red X 63

III 3. A layout that passes

10 The analog layout kit 73

11 What precheck checks 77

12 Your tools and the runner's tools 81

13 Lab 03: A layout that passes precheck 85

IV 4. Review

14 Pull requests and review 95

15 Branch rules 101

16 Layout files in git 105

17 How open-source projects work 109

18 Lab 04: A change through review 113

V 5. History

19 History as evidence 131

20 Lab 05: Find the commit that broke it 137

1. IC102-A — Working in Tiny Tapeout Repos (Analog)

In this course you make an analog Tiny Tapeout project on your own GitHub account from Tiny Tapeout's analog template, and push to it until every workflow passes. In Lab 03 you place a CMOS inverter in your tile, draw its output wire to an analog pin, and get DRC, LVS and precheck to pass. Later labs have you review a partner's pull request and use `git bisect` to find which of a small project's 33 commits broke Magic DRC.

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

Who this course is for

This course is for anyone who will push to an analog Tiny Tapeout repository: an ASIC Zoo designer, someone joining a team project with an analog block, or someone starting one of their own. Take it after IC101 and before your first push. It assumes nothing beyond IC101: no Verilog and no layout. Each lab ships the layout it needs, from Tiny Tapeout's empty frame to the CMOS inverter from AD104.

IC102 comes in two versions. Take this one, IC102-A, if you draw your layout yourself in Magic and commit the GDS, as in ASIC Zoo and CrossWalk's analog blocks. Take IC102-D if your project is written in Verilog and Tiny Tapeout's `gds` workflow builds its layout. Eight of the guide pages are the same in both.

Prerequisites	IC101, and a GitHub account
Tools	The workbench image `hpretl/iic-osic-tools:2026.08`, plus the versions of Tiny Tapeout's precheck tools that its runners use (Magic 8.3.568, KLayout 0.30.8, the sky130A PDK at `0536d02d`), which Lab 01 installs next to the workbench's
Machine	The workbench you already have, about 2 GB more disk for those tools, and an internet connection. A local precheck takes about 45 s, and a run on GitHub a few minutes
Total time	About 9 h 30 min, self-paced, including the waits for GitHub's runners. The Course path rows below add up to 9 h 30 min.

The course stops at "ready to submit": every workflow green on a commit you made. It doesn't cover submitting a project to a shuttle, and nothing in it depends on a shuttle being open.

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" section 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 analog template, fill it in, commit Tiny Tapeout's empty frame as a placeholder layout, and turn on Pages and start a new run, so that every job of every workflow ends green.	1 h 15 min
3	Read	What a cloud runner is · Reading a workflow file · The analog workflows	45 min
4	Do	Lab 02: Read the red X. Push five broken versions of your repository, based on mistakes the team has made and on precheck's rules, and for each one write down the job, the step and the first error line before you fix it.	1 h 10 min
5	Read	The analog layout kit · What precheck checks · Your tools and the runner's tools	30 min
6	Do	Lab 03: A layout that passes precheck. Place the AD104 inverter in your tile, move the power stripes, draw its output wire to `ua[1]`, and get DRC, LVS and precheck to pass before you merge it through a pull request.	1 h 45 min
7	Read	Pull requests and review · Branch rules · Layout files in git · How open-source projects work	45 min
8	Do	Lab 04: A change through review. Claim a slot in your partner's	1 h 30 min

#		What	Time
		review repository with a pull request from a fork, review the one your partner opens in yours using the render, the LVS result and the precheck table, and rebuild the layout after a conflict.	
9	Read	History as evidence	15 min
10	Do	Lab 05: Find the commit that broke it. Write a DRC script that `git bisect run` can trust, find the commit that broke a 33-commit layout history, and revert it through a pull request in a repository of its own.	55 min

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 Two places your work happens 11
2 Your GitHub account 11
2 Pushing from the workbench 12
2 Using AI tools 12
2 Where the course stops 13
2 Getting help 13
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 GitHub settings 26
5 Part 3: Make the repository 28
5 Part 4: Fill in info.yaml and docs/info.md 30
5 Part 5: A placeholder layout 32
5 Part 6: Turn on Pages 33
5 Finishing 35
5 Answers 36

2. Getting started

IC102-A is about the repository around an analog Tiny Tapeout design rather than the design itself. Every analog project on a Tiny Tapeout shuttle lives in a GitHub repository made from Tiny Tapeout's analog template, and the workflows in that repository check your layout on GitHub's machines each time you push. In this course you make one of those repositories, learn to read what its checks report, put a small layout in it that passes them, take a change through review with a partner, and use the history to find a commit that broke something.

You don't need to know how to draw layout. Each lab ships the layout it needs: Tiny Tapeout's empty frame in Lab 01, the CMOS inverter from AD104 Lab 03 in Lab 03, and a prepared project in Lab 05. You'll edit a few lines of the scripts that place and wire them, and the course's checks tell you whether the result is right. If you've done AD104, the cell will be familiar; if you haven't, nothing here depends on it.

How the course is laid out

The course path on the home page lists every page in the order to work through them. A Read row points to guide pages like this one, which explain things and have no commands for you to run. A Do row points to a lab writeup, and every command in the course is in a lab. Each lab is split into parts with numbered steps. Most parts start with the question they answer, and every part ends with a "You should see" or "Check" line that tells you whether it worked. When a step asks a question, its answer is in a collapsed block under it; try it first, then open the block. Some steps ask you to predict a result before you see it, which is worth doing honestly, because a wrong prediction is the quickest way to find out what you misunderstood.

The files the labs run are in the course's repository, under `labs/`. You'll get them into the workbench with `mod add ic102-a` in Lab 01.

Two places your work happens

Your work in this course happens in two places, the workbench and GitHub, and it's worth keeping them straight from the start.

The workbench is the IIC-OSIC-TOOLS desktop you set up in IC101. That's where you build layouts with Magic, run checks, and use git: every command in the labs is typed into a terminal on that desktop. Your repositories go in a `projects` folder of your workspace, `/foss/designs/projects`, each in a folder with the repository's name. `/foss/designs` is the `workspace` folder on your own disk, so they survive when the workbench restarts, and `projects/` keeps them apart from `modules/`, where `mod` puts the courses. Anything outside `/foss/designs` doesn't survive a restart, because the container's home folder is thrown away with the container. That's why, in Lab 01, you'll store git's settings in each repository rather than in your home folder.

GitHub is where your repository lives, and where its workflows run each time you push, on GitHub's machines (the *runners*). What a cloud runner is explains what those are. Some things can only be done on GitHub: making the repository, turning on Pages, opening and reviewing pull requests, and reading a run's results.

Tiny Tapeout's precheck, the most important of those results, doesn't use the workbench's tools. It installs its own versions of Magic, KLayout and the PDK on the runner, and a layout that passes with one set can fail with the other. So in Lab 01 you'll install precheck's versions next to the workbench's, and `make precheck` then tells you, on your own machine and before you push, what precheck on the runner will say. Your tools and the runner's tools has the details. IC102-A follows Tiny Tapeout's versions on purpose; the other courses use the workbench's.

Your GitHub account

You'll work in your own GitHub account throughout, so all you need is a free account: every repository you make in this course belongs to your account, and none goes in the team's organization. Every one

of them is public, because GitHub Pages, which the template's `viewer` job needs, works only in public repositories on a free account, and because GitHub's runners are free and unlimited for public repositories. Remember that everything in a public repository is public: your commits, the `author` field in `info.yaml`, and the email address in every commit. Lab 01 has you use GitHub's no-reply address for that reason.

These are the repositories you'll make, and each one's clone goes in `/foss/designs/projects/` under the same name:

Repository	Lab	Made how
`ttsky-ic102-a`	01 to 03	**Use this template** on Tiny Tapeout's analog template
`ttsky-ic102-a-review`	04	an empty repository, into which you push a copy of the course's sandbox
`ttsky-ic102-a-review-fork`	04	**Fork** on your partner's `ttsky-ic102-a-review`, renamed, because you already have a repository with that name
`ttsky-ic102-a-bisect`	05	an empty repository, into which you push a prepared history

Lab 04 is the one lab you do with a partner from the course. Each of you sends a pull request to the other's `ttsky-ic102-a-review`, and reviews the one that arrives in your own.

Pushing from the workbench

GitHub doesn't accept your account's password for `git push`. From the workbench you push over HTTPS with a **personal access token**, a long random string that stands in for a password and that you can limit to particular repositories and permissions. Lab 01 has you make a fine-grained token whose resource owner is your own account, for "All repositories", with 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).

"All repositories" is there because the Lab 04 and Lab 05 repositories don't exist yet when you make the token in Lab 01. Every push in the course goes to a repository you own: the four in the table above, including your fork of your partner's, from which your pull request goes to them. A fine-grained token for your own account covers those, and any other repository in your account, and nothing outside it. The other kind, a classic token with the `public_repo` scope, could push to every public repository your account can write to, including those of any organization you belong to (in `uoftasic`, members get write access to every repository by default), which is far more than the course needs. A fine-grained token can't push to a repository someone else owns, even if they add you as a collaborator, so in Lab 04 you send your change to your partner from a fork, which you own.

When `git push` asks for a username and password, you give your GitHub username and paste the token. A line you run in each clone tells git to keep it in memory for 8 hours, so you're asked about once a session, and again after the workbench restarts. Keep the token in a password manager, not in a file under `/foss/designs`: that folder is your workspace, itself a git repository, and a token saved there is one careless `git add` away from being published. Delete the token when you've finished the course.

Using AI tools

You may use AI tools such as Claude, ChatGPT or Copilot to help with the work in this course, for example to explain an error message, write a script or draft a pull request description. The rules are the ones in Pull requests and review: you answer for every line you submit, however it was written, and you say in a pull request which parts a tool wrote and what you checked yourself. The labs are built so that a check

tells you when your work is right, whoever or whatever wrote it: LVS compares your layout with the reference either way.

Where the course stops

IC102-A takes a project as far as "ready to submit": every workflow green, including precheck, on a commit you made and can name. It doesn't cover submitting to a shuttle. You won't create a project on `app.tinytapeout.com`, pay, or use a coupon, and nothing in the course depends on a shuttle being open. Every project you make here is practice, and Lab 04′s review repositories are never submitted.

Getting help

Ask in the team Discord. A question that can be answered quickly has three things in it: the lab, part and step you're on; the command you ran; and the output, pasted as text rather than a photo, from the first line that looks wrong. If it's about a run on GitHub, include the link to the run. Logs on GitHub are deleted after 90 days, so if the question might matter later, paste the error line itself as well.

Check yourself

1. You restart the workbench, open a terminal, and `git commit` stops with `Author identity unknown`. You set your name and email in Lab 01. What happened?

   Answer: A1, in Answers at the end.

2. Why does the course install a second Magic when the workbench already has one?

   Answer: A2, in Answers at the end.

Answers

A1. Check yourself

You set them with `--global`, which writes them to the container's home folder, and the restart threw that folder away. Set them again without `--global`, inside the repository, where they're stored in `.git/config` on your own disk.

A2. Check yourself

Because precheck, which decides whether your layout passes, uses Magic 8.3.568 with a particular commit of the PDK, and the workbench has Magic 8.3.681 with a different one. Running precheck's own versions locally means a pass on your machine predicts a pass on the runner.

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:

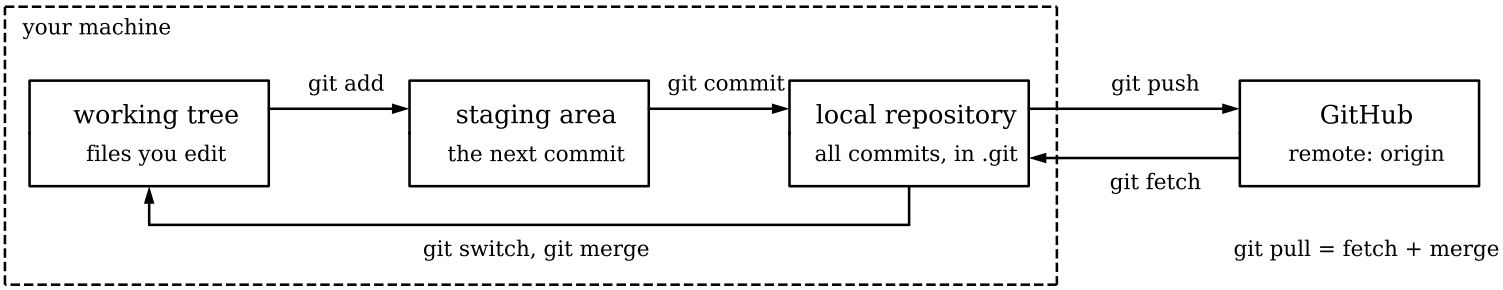


Figure 3.1: The four places a change lives, and the commands that move it between them.

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:

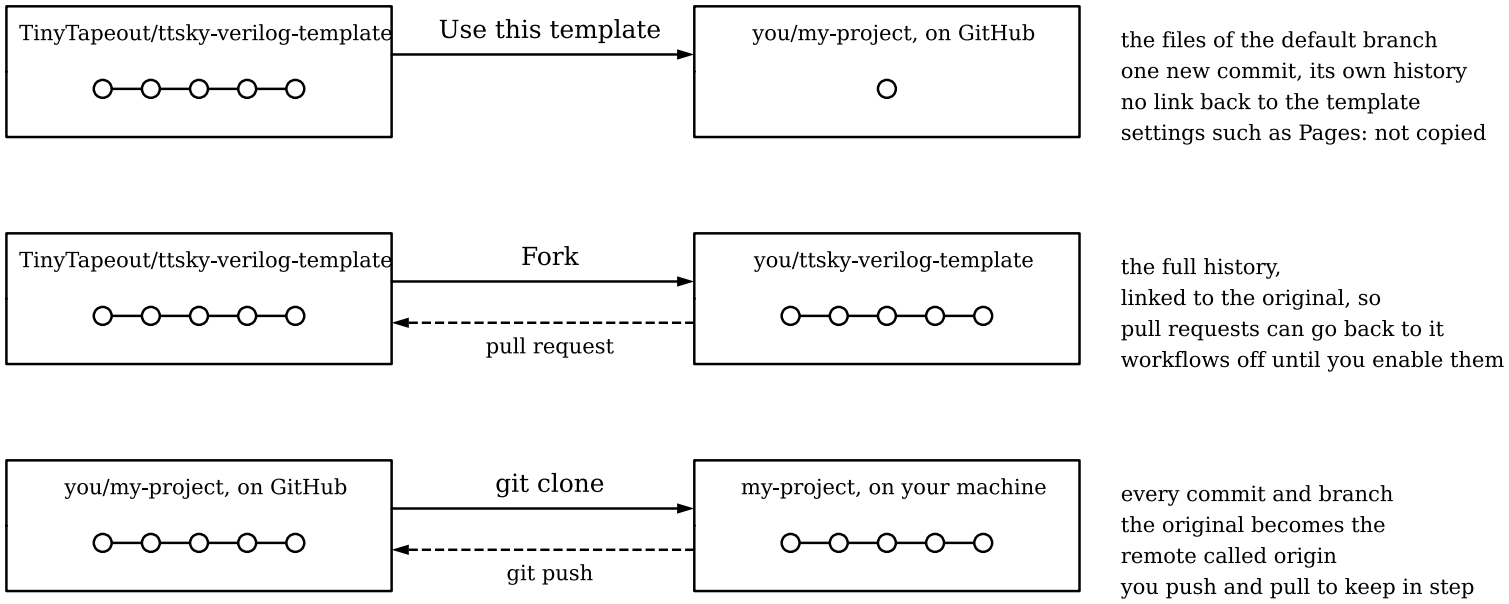


Figure 4.1: What a template, a fork and a clone each take from the repository they start from.

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.

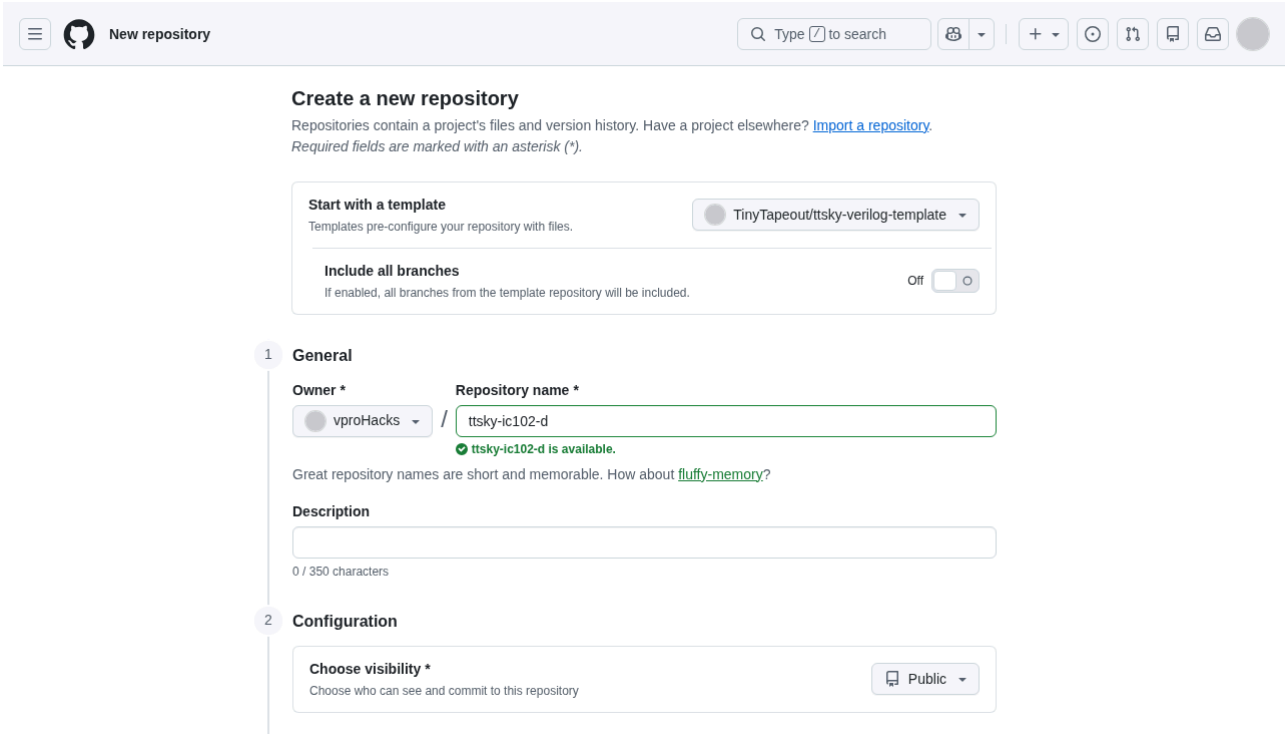


Figure 4.2: Creating a repository from the template. "Include all branches" is off by default, and the course uses a public repository.

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

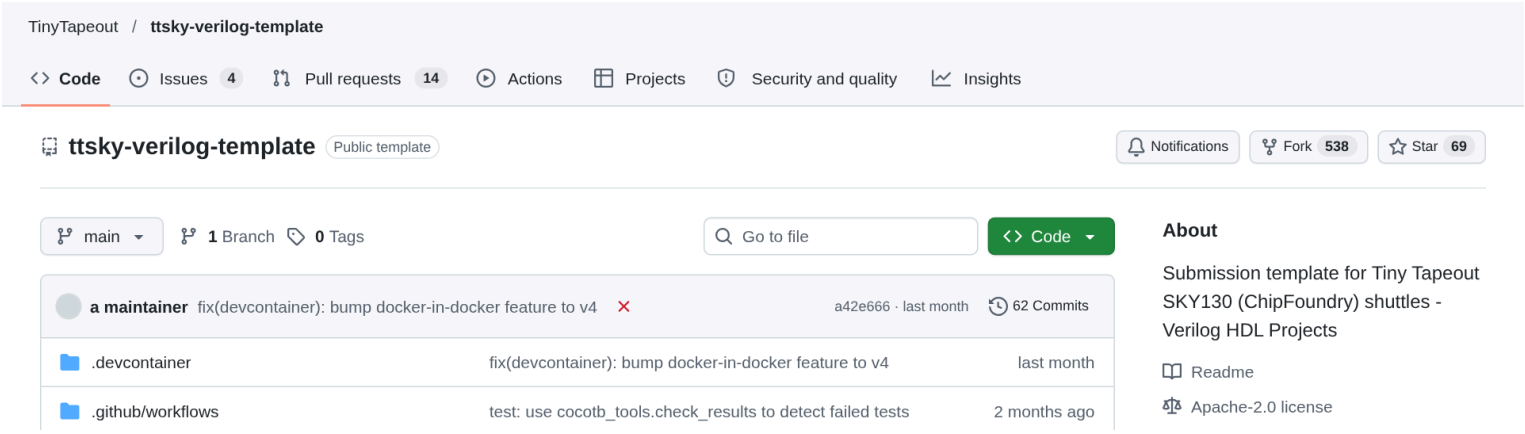


Figure 4.3: Tiny Tapeout's digital template on Nov 15, 2026. The maintainer's name is replaced with a role.

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 your own public repository from Tiny Tapeout's analog template and fill in the two files its checks read. Then you'll commit a placeholder layout, the empty 1x2 frame that Tiny Tapeout's own script draws, and keep going until both of the template's workflows pass on a commit you made, the `viewer` job included. You'll come back to this repository in later labs.

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 IC101 done (the workbench desktop opens and you can get a terminal), a GitHub account, and about 2 GB of free disk for the course tools.

This lab takes about 1 h 15 min. Half of that is waiting for GitHub's runners, which is a good time to read ahead.

In the outputs below, the top module is `tt_um_yourname_ic102`, and yours will have your own username in it. The repository is `ttsky-ic102-a`, the name the rest of the course uses for it.

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. Every command in this course runs in that terminal.
2. Load the environment and get the course files:

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

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

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

   `mod add` leaves you in the course folder, and `mod ic102-a` prints the same `cwd:` line again, as the last line.
3. Install the course's copy of precheck's tools. They're installed alongside the workbench's own tools, in `labs/common/tt-tools/.local`. That folder is on your own disk, so you only do this once:

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

`make install` took 85 s on one CPU core with a fast connection, and the tools take up 1.8 GB. Your tools and the runner's tools explains why the course installs these at all. `make check-tools` should end with this table:

```
Tool             Runner (pinned)     Here              Workbench          Status
----             ---------------     ----              ---------          ------
Python           3.11.18           3.11.16           3.12.3           ok (3.11)
Magic              8.3.568             8.3.568             8.3.681            ok
KLayout            0.30.8              0.30.8              0.30.11            ok
klayout (pip)      0.30.13             0.30.13             0.30.11            ok
gdstk (pip)        0.9.52              0.9.52              1.0.1              ok
yowasp-yosys       0.63.0.0.post1107   0.63.0.0.post1107   -                  ok
ciel               3.2.0               3.2.0               2.6.1              ok
sky130A PDK        0536d02d875c        0536d02d875c        026824c7969c       ok
tt-support-tools   9d13ff3e3e62        9d13ff3e3e62        -                  ok

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

If you see	Do this
`mod: no such module: ic102-a`	You ran `mod ic102-a` before `mod add ic102-a`.
`The read operation timed out` during `make install`	The PDK's download server didn't answer in time. Run `make install` again; it starts from the part that failed.
Anything marked `MISSING` in `make check-tools`	Run `make install` again and read the last lines it prints.

For anything else, see IC101's Troubleshooting page. The tools' README has more rows.

Check: `make check-tools` ends with `All pinned tools are installed at the runner's versions.`

Part 2: Your GitHub settings

This part sets up two things the rest of the course needs: an email address for your commits, and a way for the workbench to push to GitHub.

1. On GitHub, open Settings › Emails, turn on the "Keep my email addresses private" switch, and copy the no-reply address that GitHub then shows, which has the form `ID+USERNAME@users.noreply.github.com`. Git in one page says why.
2. Make a token for the workbench to push with: Settings › Developer settings › Personal access tokens › **Fine-grained tokens** › **Generate new token**. Give it:
   - Token name `IC102 workbench`, and Resource owner your own account, which is what the form starts with.
   - Expiration 60 days. The form starts at 30 days, so pick 60 days from its menu.
   - Repository access: **All repositories**. The form starts at "Public repositories", which gives read access only.
   - Permissions: with the **Repositories** tab selected, click **+ Add permissions** and tick **Contents** and **Workflows** in the list that opens. Workflows comes in at "Read and write", the only choice it has, but Contents comes in at "Read-only", so change its Access menu to "Read and write". GitHub adds Metadata by itself, marked "Required" and "Read-only". Leave everything else as it is.

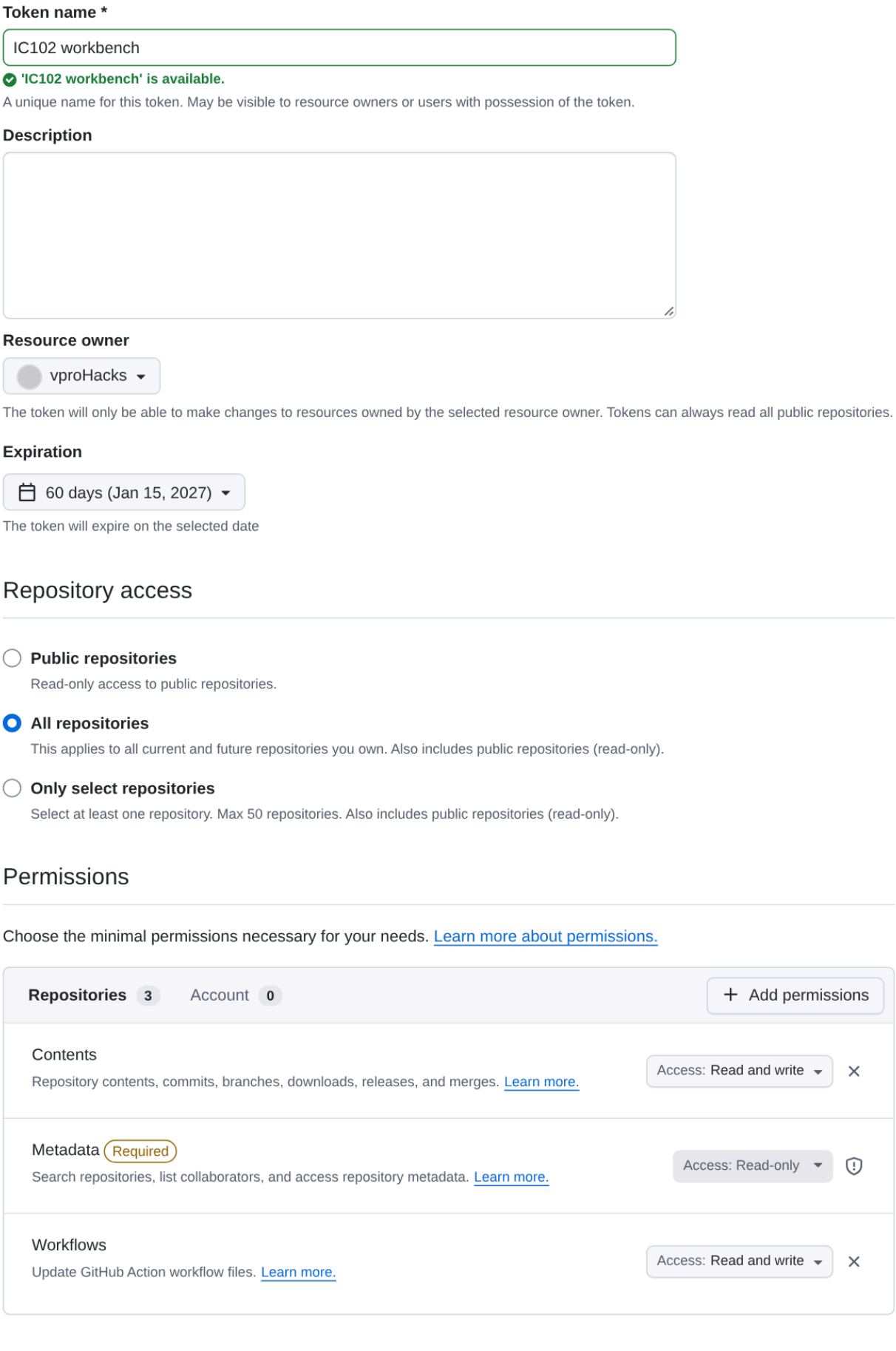


Figure 5.1: The new-token form filled in as above, before Generate token is pressed. The resource owner is the account we tested with, and its avatar is replaced with a grey disc.

Press **Generate token**, and copy the token into your password manager now, because GitHub shows it only once.

Getting started explains why it's this kind of token, why it needs Workflows as well as Contents, and why it lives in your password manager rather than in a file.

Check: you have the no-reply address and the token saved somewhere you can paste from.

Part 3: Make the repository

What does a repository made from the template contain, and which of its workflows pass before you've changed anything?

1. Open https://github.com/TinyTapeout/ttsky-analog-template, click **Use this template** › **Create a new repository**, choose your own account as the owner, name it `ttsky-ic102-a`, make it **Public**, and click **Create repository**. It has to be public, because GitHub Pages, which the `viewer` job needs, works only on public repositories on a free account.

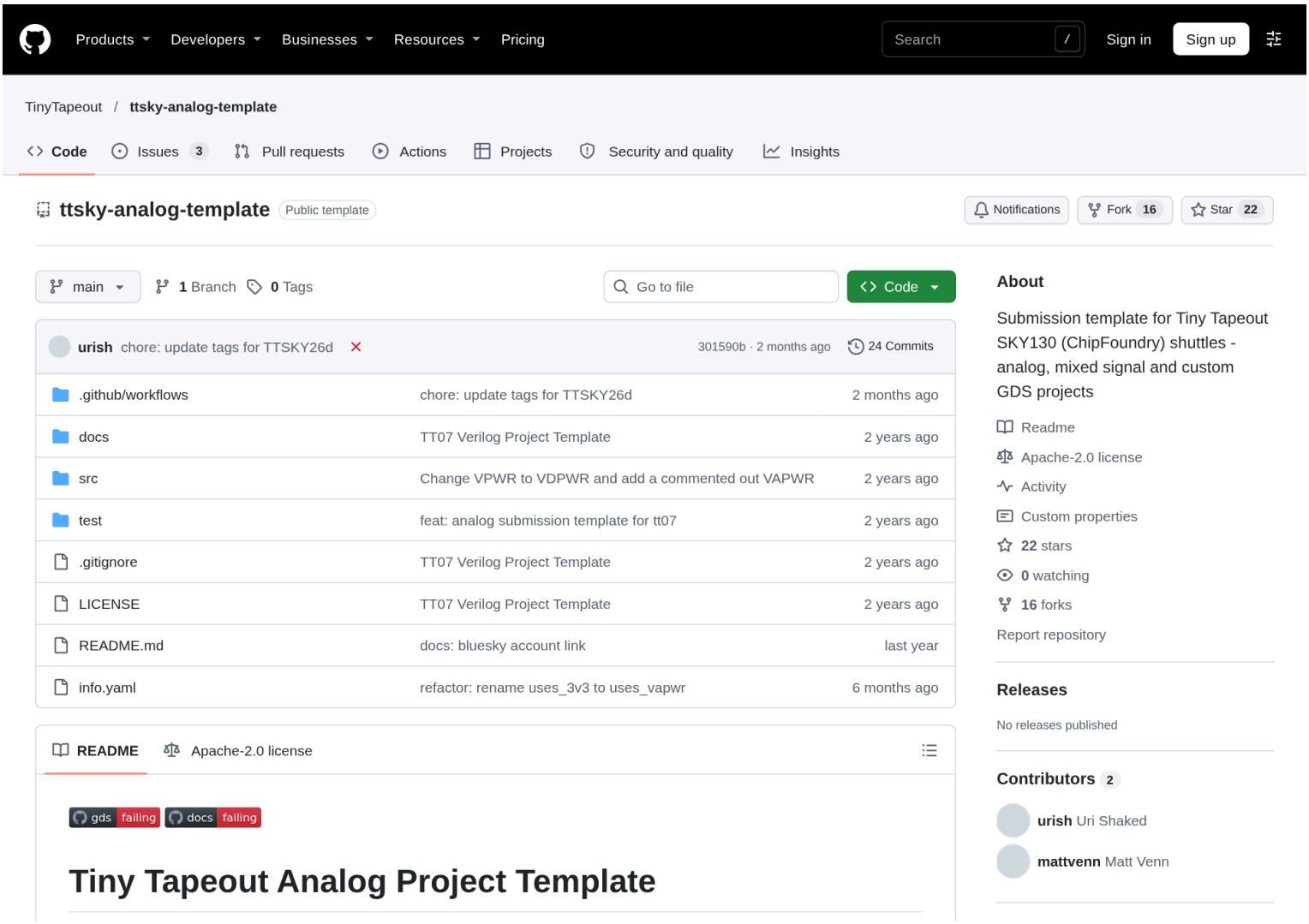


Figure 5.2: The analog template on GitHub, with its "Use this template" button.

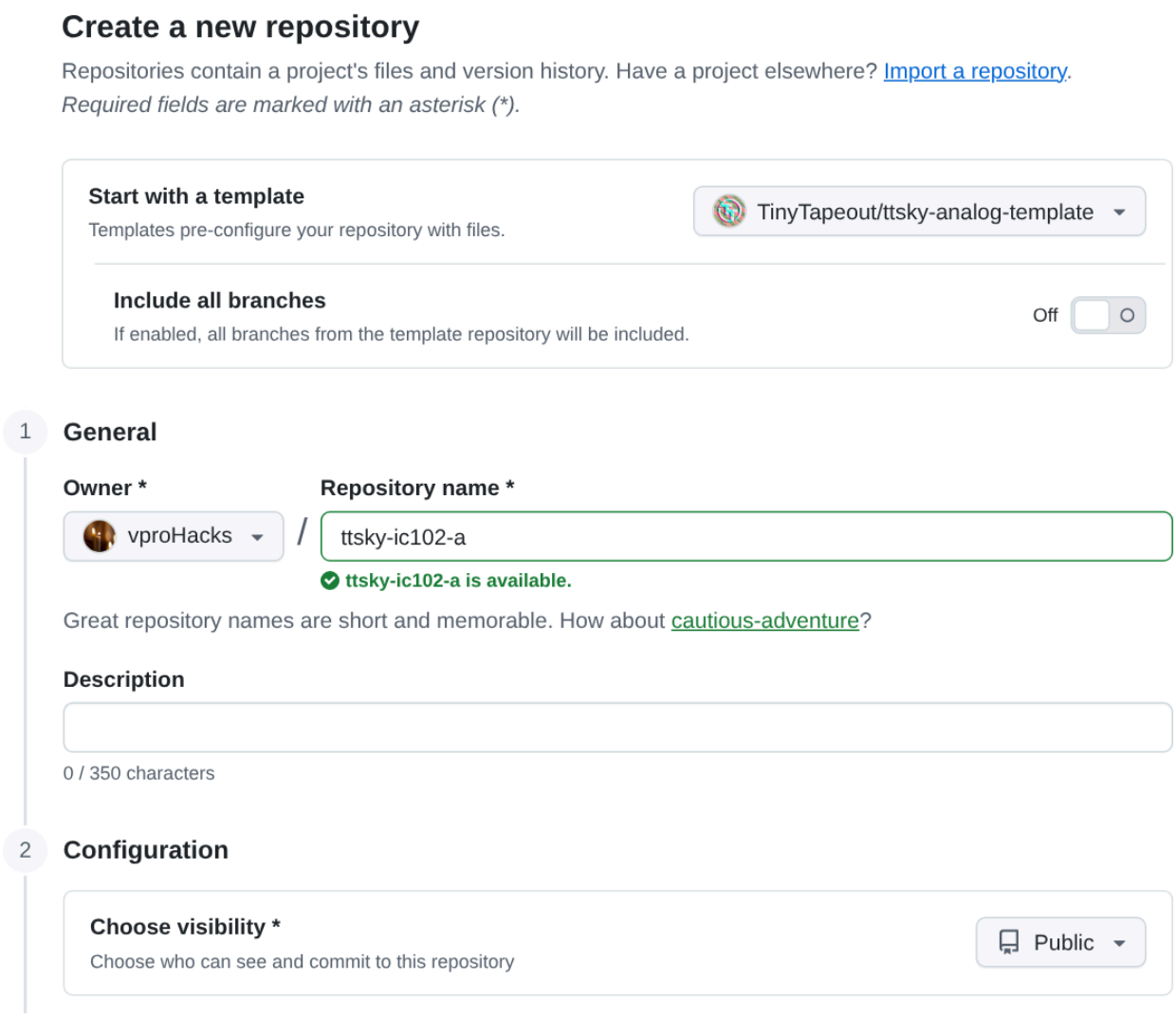


Figure 5.3: Creating a repository from the analog template: your own account as the owner, "Include all branches" off, and Public.

2. Before you look at the Actions tab, write down how many workflow runs you expect to find there already, and whether each will pass. The template's `.github/workflows/` has two files.
3. Open your repository's **Actions** tab. You should see two runs, `docs` and `gds`, both on the commit "Initial commit", and both red, as they were on the team's `asic-zoo-template` on Nov 11,

   2026.

   Answer: A1, in Answers at the end.
4. Clone it into the workbench, into a `projects` folder of your workspace:

   ```
   mkdir -p /foss/designs/projects && cd /foss/designs/projects
   git clone https://github.com/YOUR-USERNAME/ttsky-ic102-a.git
   cd ttsky-ic102-a
   git log --oneline
   ```

   You should see `Cloning into 'ttsky-ic102-a'...` and a log of exactly one commit, "Initial commit", whose hash is your own. The template's own history has been left behind, as Templates, forks and clones says it would.
5. Tell git who you are, and to remember your token, in this repository:

```
git config user.name "Your Name"
git config user.email "ID+USERNAME@users.noreply.github.com"
git config credential.helper 'cache --timeout=28800'
```

The third line keeps your token in memory for 8 hours after you first type it, so you're asked for it about once a session. Leave out `--global`. A global setting goes into the container's home folder, which is thrown away whenever the workbench restarts, and your next commit would then stop with this:

```
Author identity unknown

*** Please tell me who you are.
...
fatal: unable to auto-detect email address (got 'designer@41bf5069a52d.(none)')
```

A setting without `--global` goes into `.git/config` in the repository, which is on your own disk, so it's still there after a restart. You'll repeat these three lines in each repository you clone or create in this course.

6. See which version of Tiny Tapeout's actions your workflows run:

```
grep -n "uses:" .github/workflows/*.yaml
```

You should see four lines that end in `@ttsky26d` (the docs action in `docs.yaml`, and `custom_gds`, `precheck` and `viewer` in `gds.yaml`), plus two that use `actions/checkout@v4`. Your repository's own workflow files now name that tag, and they keep naming it whatever Tiny Tapeout later changes in its template. The analog workflows goes through what each of the three `gds.yaml` jobs does.

Check: `git log --oneline` shows one commit, and `git config user.email` prints your no-reply address.

Part 4: Fill in info.yaml and docs/info.md

Which of the `docs` workflow's complaints can you fix before you push?

1. Add the course's layout kit to your repository. The kit includes a `Makefile` whose `make docs` runs the same check as the `docs` job:

```
/foss/designs/modules/ic102-a/labs/lab-01-your-repo-from-the-template/start.sh .
make docs
```

`start.sh` lists what it added (`Makefile`, `layout/`, and a line in `.gitignore`). From `make docs` you should then see:

```
## 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

make: *** [Makefile:100: docs] Error 1
```

That's the list that the `docs` run on the team's `hart-1` printed on GitHub, word for word, because `make docs` runs the same function as the `docs` job, from the same commit of Tiny Tapeout's support tools.

2. Edit `info.yaml`. Fill in `title`, `author` and `description`; the author is public, so write whatever you're happy to see in a datasheet. Set `top_module` to `tt_um_` followed by your GitHub username and `_ic102`, in lower case with any `-` changed to `_`, because the name becomes a Verilog module name. In the `pinout`

section, give `ua[0]` and `ua[1]`, the first two analog pins, a description each, such as `"inverter input (not connected yet)"` and `"inverter output (not connected yet)"`. Leave `analog_pins: 2` and `tiles: "1x2"` as they are.

3. Replace the two placeholder sentences in `docs/info.md`, "Explain how your project works" and "Explain how to use your project", with a sentence or two of your own.
4. In `src/project.v`, change `module tt_um_example` to your top module's name.
5. Run `make docs` again. You should see `Documentation check passed successfully.`
6. Commit and push:

```
git add -A
git commit -m "Fill in info.yaml and docs/info.md, and add the IC102-A layout kit"
git push
```

   git asks for a username and a password: give your GitHub username, and paste the token as the password. The cache you set up in Part 3 keeps it for 8 hours, and forgets it when the workbench restarts.

7. Before the runs finish, predict which of the two workflows will be green this time. Then open the `gds` run, then its `gds` job, then the failed step, and write down its first error line.

   You should see `docs` green and `gds` red, with the message that every push to the team's `asic-zoo-template` produced before it had a layout. That repository still had the template's top module, so its message names `tt_um_example` where yours names your own:

```
cp: cannot stat 'gds/tt_um_example.gds': No such file or directory
```

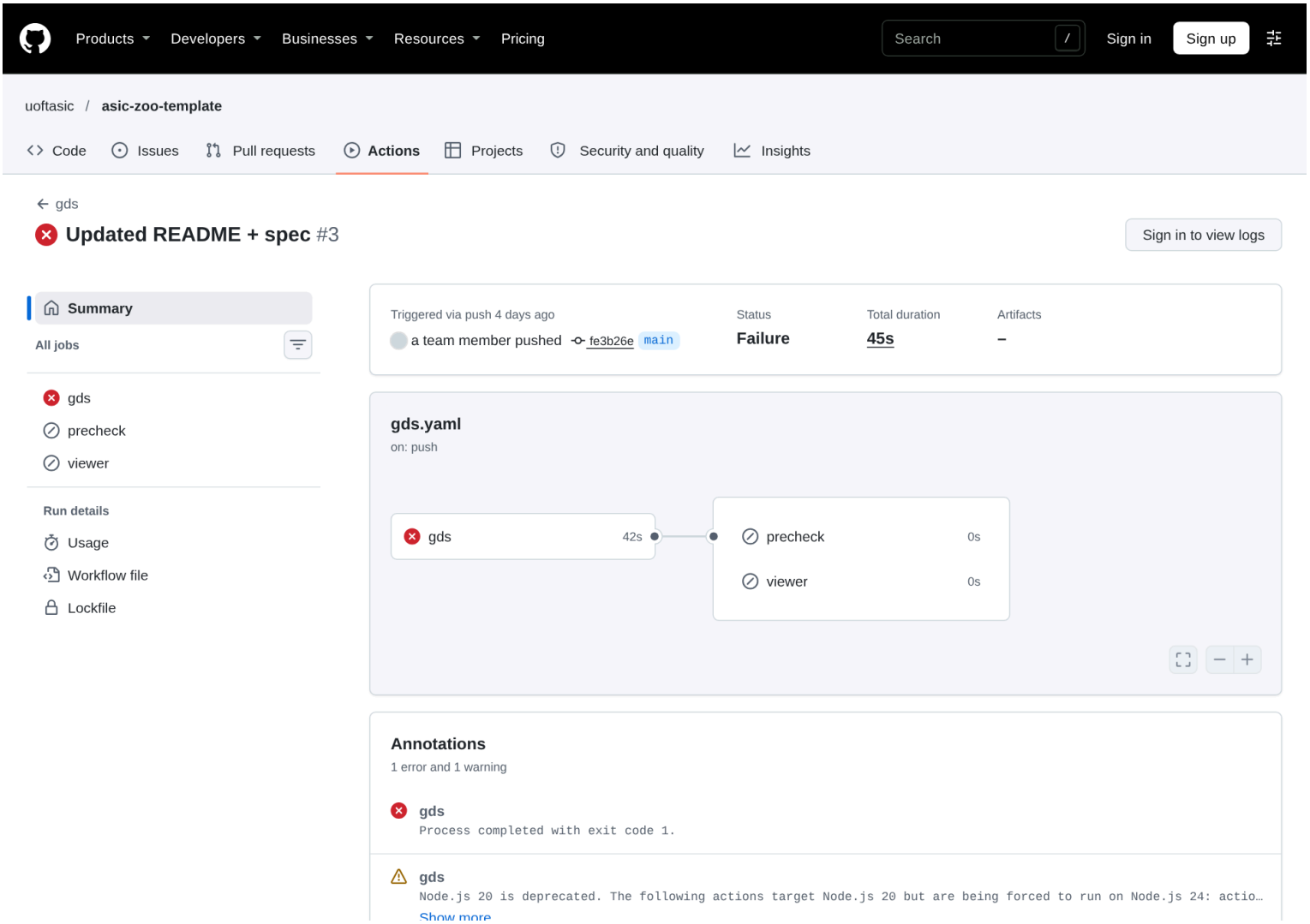


Figure 5.4: The same failure in uoftasic/asic-zoo-template: gds red after 42 s, precheck and viewer skipped. The person who pushed is shown as a team member.

Why are precheck and viewer grey?: A2, in Answers at the end.

Check: the `docs` run on your commit is green.

Part 5: A placeholder layout

What's the least layout that passes precheck?

1. Run `make`. You should see:

```
layout/init.tcl draws a cell called 'tt_um_analog_example', but top_module in
info.yaml is 'tt_um_yourname_ic102'.
Set TOP_LEVEL_CELL in the settings block of layout/init.tcl to tt_um_yourname_ic102.
make: *** [Makefile:71: check-top] Error 1
```

   `layout/init.tcl` is Tiny Tapeout's own `magic_init_project.tcl`, copied unchanged, and its comments ask you to change the settings block at the top. Change `TOP_LEVEL_CELL` to your top module. A GDS whose top cell has a different name from `top_module` fails precheck, as you'll see in Lab 02.

2. Run `make` again. You should see:

```
magic: the kit's frame and power stripes (layout/init.tcl)
magic: your additions (layout/wires.tcl), then GDS and LEF
wrote gds/tt_um_yourname_ic102.gds and lef/tt_um_yourname_ic102.lef
```

The frame has every pin of the 1x2 tile, a VDPWR and a VGND stripe on metal 4, and nothing else. `layout/wires.tcl` adds a 9 µm stub of metal 4 to `ua[0]` and to `ua[1]`. Read its comment: why does an empty frame need those two stubs?

Answer: A3, in Answers at the end.

3. Run precheck on it before you push, with the same tool versions the runner uses:

```
make precheck
```

It takes about 45 s. You should see `INFO: Precheck passed for .../tt_um_yourname_ic102.gds!` 🎉 and a table of 15 checks, every one ✅, from `Magic DRC` to `Verilog syntax check`.

4. Before you commit, check what git will take:

```
git status
```

It should list your change to `layout/init.tcl` and the new `gds/` and `lef/` folders, but not `tt_submission/` or `layout/build/`. The template's `.gitignore` and the kit's line in it keep precheck's and Magic's scratch files out of git, so `git add -A` leaves them behind. Then commit `layout/`, `gds/` and `lef/`, and push:

```
git add -A
git commit -m "Add the 1x2 frame with stubs on ua[0] and ua[1]"
git push
```

5. Predict which jobs of the new `gds` run will pass, then open it. You should see `gds` and `precheck` green and `viewer` red. We made `vproHacks/ic102-test-a` from the template to test this lab, and its `gds` run on Nov 16, 2026 took 3 min 52 s. Its `viewer` job's log had this line, and yours will have one like it:

```
##[error]Error: Failed to create deployment (status: 404) with
build version e9a6f474deb92b2898e062ddb8f54bcf052b1d73. Request ID
5868:1C0C8:4B853B6:F3D6856:6AFAB74A Ensure GitHub Pages has been enabled: https://
github.com/vproHacks/ic102-test-a/settings/pages
```

Yours has its own build version and request ID, and your repository's address at the end.

Check: `precheck` is green on GitHub, and its table on the run's summary page matches the one `make precheck` printed.

Part 6: Turn on Pages

How do you get the `viewer` job green without changing a file?

1. In your repository, open Settings › Pages, and under Build and deployment set Source to **GitHub Actions**.

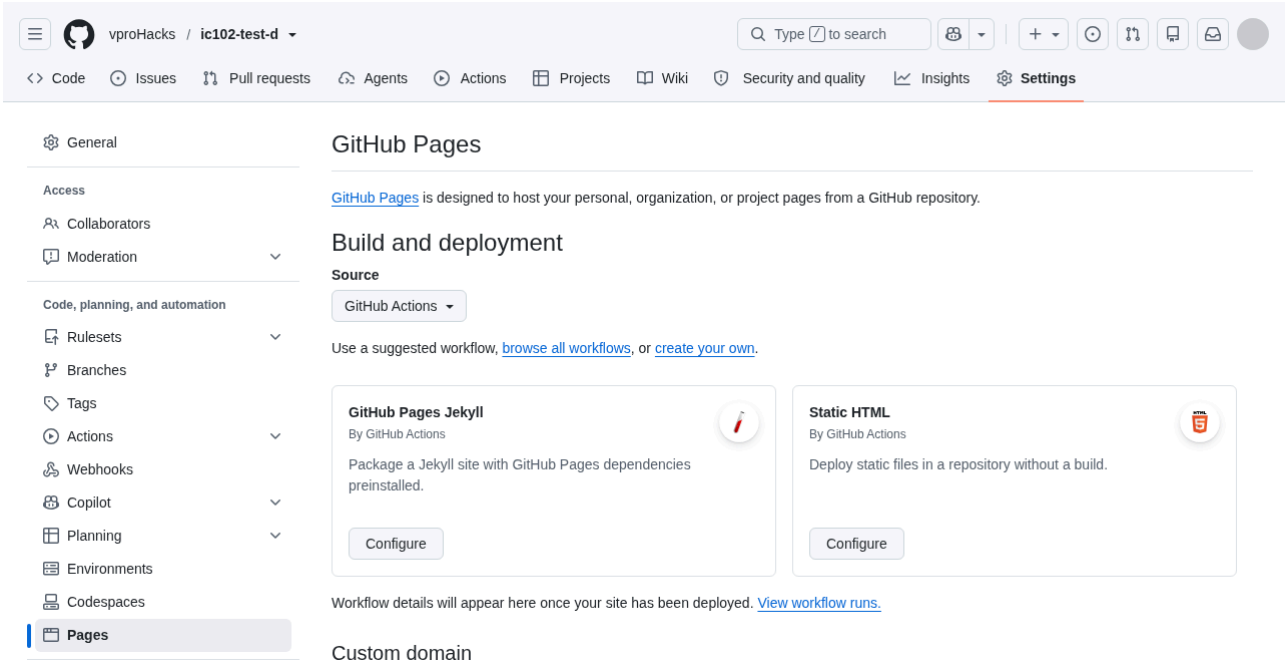


Figure 5.5: Settings › Pages, with the source set to GitHub Actions.

2. Don't press "Re-run failed jobs" on the red run. The `viewer` action at `ttsky26d` uploads an artifact called `github-pages` on every attempt, and a re-run keeps the first attempt's, so the re-run fails with `Multiple artifacts named "github-pages"`. Reading a workflow file explains why. Start a new run instead: Actions › gds › **Run workflow** › branch `main` › **Run workflow**. (A new push would start one too, but you have nothing to push.)

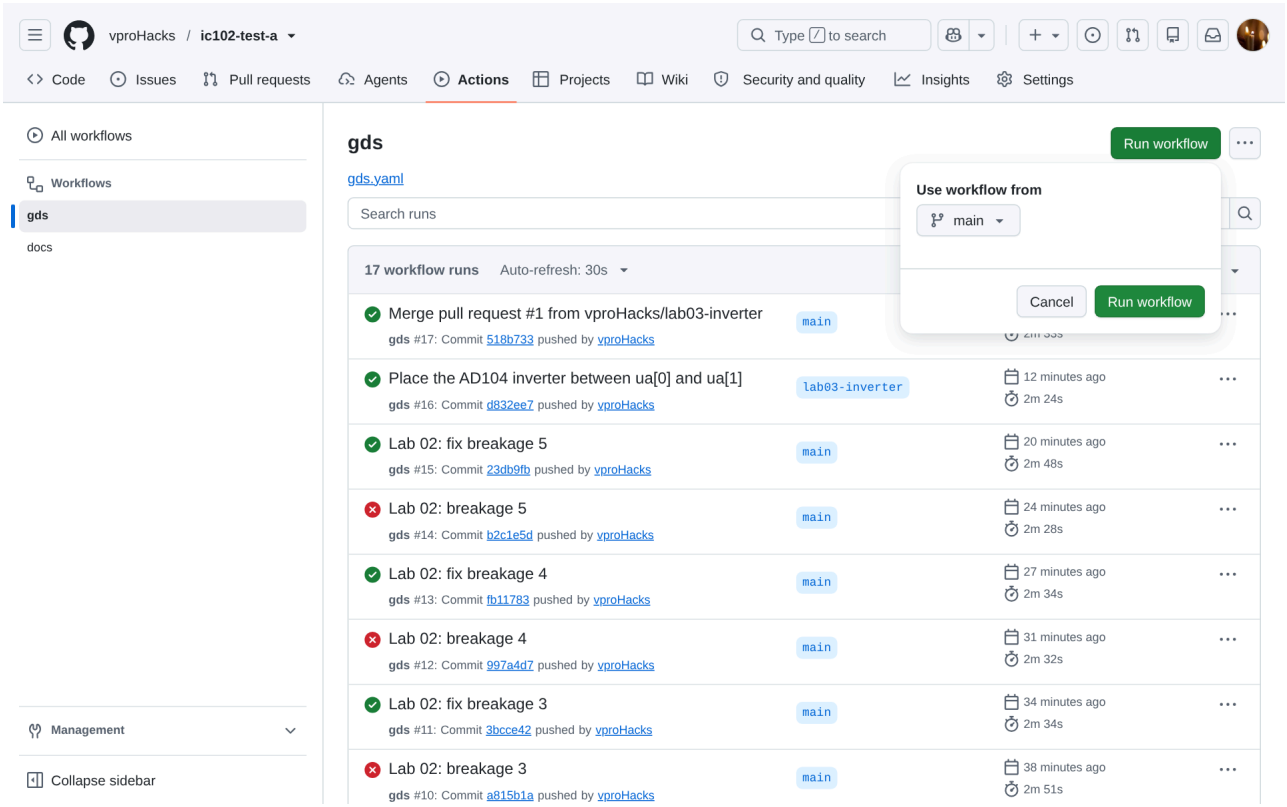


Figure 5.6: Starting a new gds run by hand: Actions, the gds workflow, and the Run workflow dropdown with branch main.

3. You should see all three jobs green. The `viewer` job's summary, on the run's summary page, has an "Open 3D viewer" link, and `https://YOUR-USERNAME.github.io/ttsky-ic102-a/` now redirects to Tiny Tapeout's viewer, with your frame and its two stubs in it.

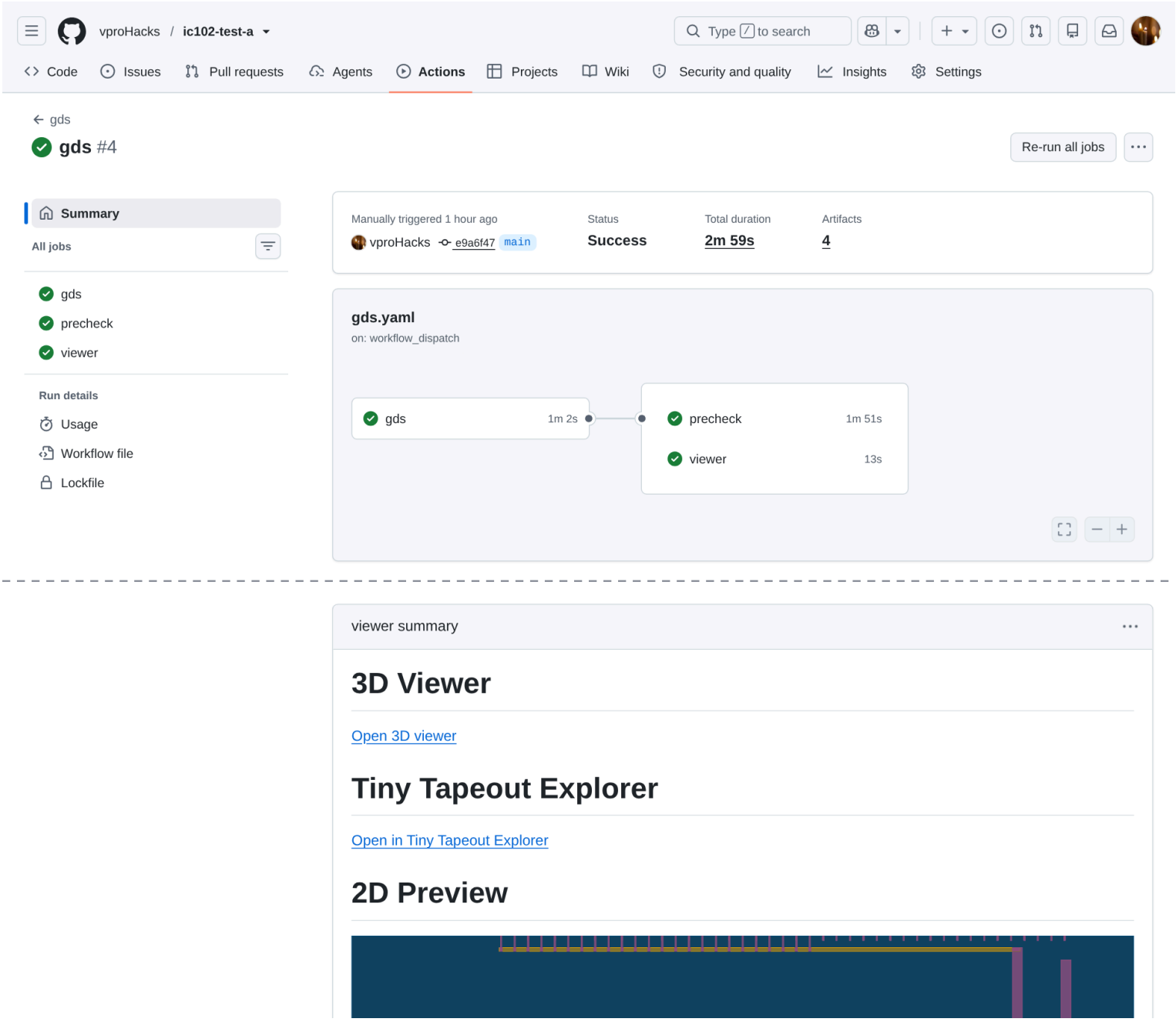


Figure 5.7: A gds run with every job green, started with Run workflow in our test repository, and further down its summary page, the viewer job's links. The 2D preview under them is loaded from the Pages site, so it always shows the layout deployed last.

Check: the run is green from end to end.

Finishing

You're done when the latest commit on your repository's `main`, a commit you made, has a green `docs` run and a green `gds` run with all three jobs (`gds`, `precheck` and `viewer`) passed.

Questions:

1. Your Part 6 run was started by hand, not by a push. Which commit did it check?

Answer: A4, in Answers at the end.

2. The `gds` job in Part 4 failed in well under a minute. When it passed, in Parts 5 and 6, it still took only about a minute: 61 s on the Part 5 push and 62 s in the Part 6 run in our test of this lab, and each run's summary page gives your own times. A digital project's `gds` job took between 207 s and 1,056 s in the team's repositories. Why is the analog one so much faster even when it passes?

   Answer: A5, in Answers at the end.

Submit: nothing for this lab. Keep the repository; Labs 02, 03 and 05 use it.

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

Answers

A1. Part 3: Make the repository

Both workflows run on every push (`on: push`), and creating the repository pushed its first commit. `docs` fails because `info.yaml` and `docs/info.md` still hold the template's empty values and placeholder text. `gds` fails because the template has no layout in it: the `gds` job looks for `gds/tt_um_example.gds` and the file doesn't exist. You'll see both messages in Part 4.

A2. Part 4: Fill in info.yaml and docs/info.md (Why are precheck and viewer grey?)

Both are declared with `needs: gds`, so they run only if the `gds` job succeeds. It failed at its copy step, so they were skipped.

A3. Part 5: A placeholder layout

`info.yaml` says `analog_pins: 2`, and precheck's analog pin check fails unless each of the first two analog pins touches some metal (and has a description in the pinout, which you gave it in Part 4). An empty frame has nothing on `ua[0]` or `ua[1]`. The stubs aren't connected to anything; in Lab 03 you replace them with wires to an inverter.

A4. Finishing

The latest commit on the branch you chose, `main`, which was your Part 5 commit. "Run workflow" runs the workflow file and the files as they are on that branch's latest commit.

A5. Finishing

Because it doesn't build anything. A digital project's `gds` job runs LibreLane to turn Verilog into a layout, whereas the analog template's `custom_gds` step only copies the GDS and LEF you committed, converts the GDS to OASIS and renders a picture of it. All the layout work happened on your machine, when you ran `make`.

II 2. Runs that fail

6 **What a cloud runner is 39**
6 A new machine for every job 39
6 How big it is and how long it takes 40
6 What it costs 40
6 Your tools and the runner's tools 40
6 Check yourself 41
6 Answers 41

7 **Reading a workflow file 43**
7 What a push starts 43
7 The parts of a workflow file 43
7 Composite actions 45
7 Reading a failed run 46
7 Job summaries and artifacts 48
7 Re-running a run 50
7 GitHub Pages and the viewer 52
7 Check yourself 54
7 Answers 54

8 **The analog workflows 57**
8 docs 57
8 gds 57
8 Which push runs what 60
8 Pull requests in a shared repository 60
8 Check yourself 61
8 Answers 61

9 **Lab 02: Read the red X 63**
9 Before you start 63
9 Part 1: The routine 63
9 Part 2: Breakage 1 66
9 Part 3: Breakage 2 66
9 Part 4: Breakage 3 66
9 Part 5: Breakage 4 67
9 Part 6: Breakage 5 67
9 Part 7: A red X that isn't yours 68
9 Finishing 68
9 Answers 68

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**.

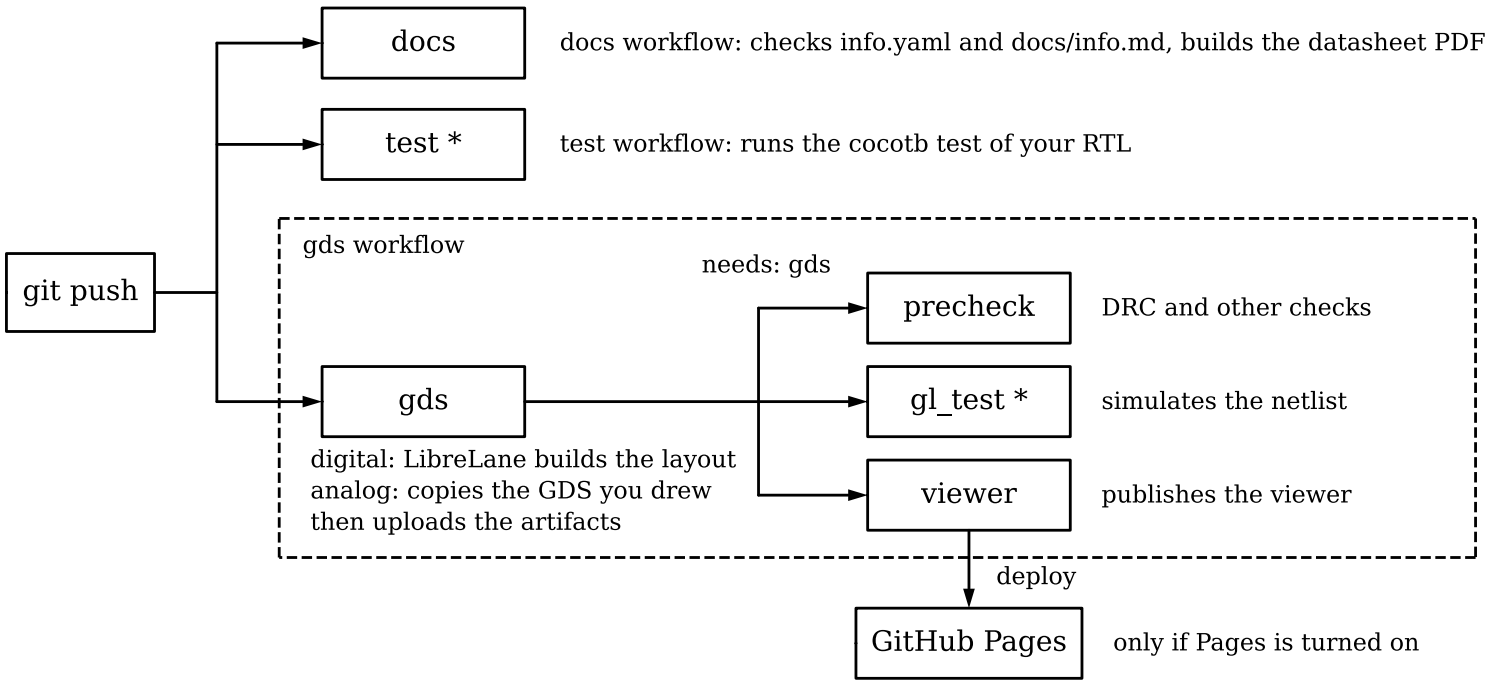


Figure 7.1: What one push starts in a Tiny Tapeout repository.

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):

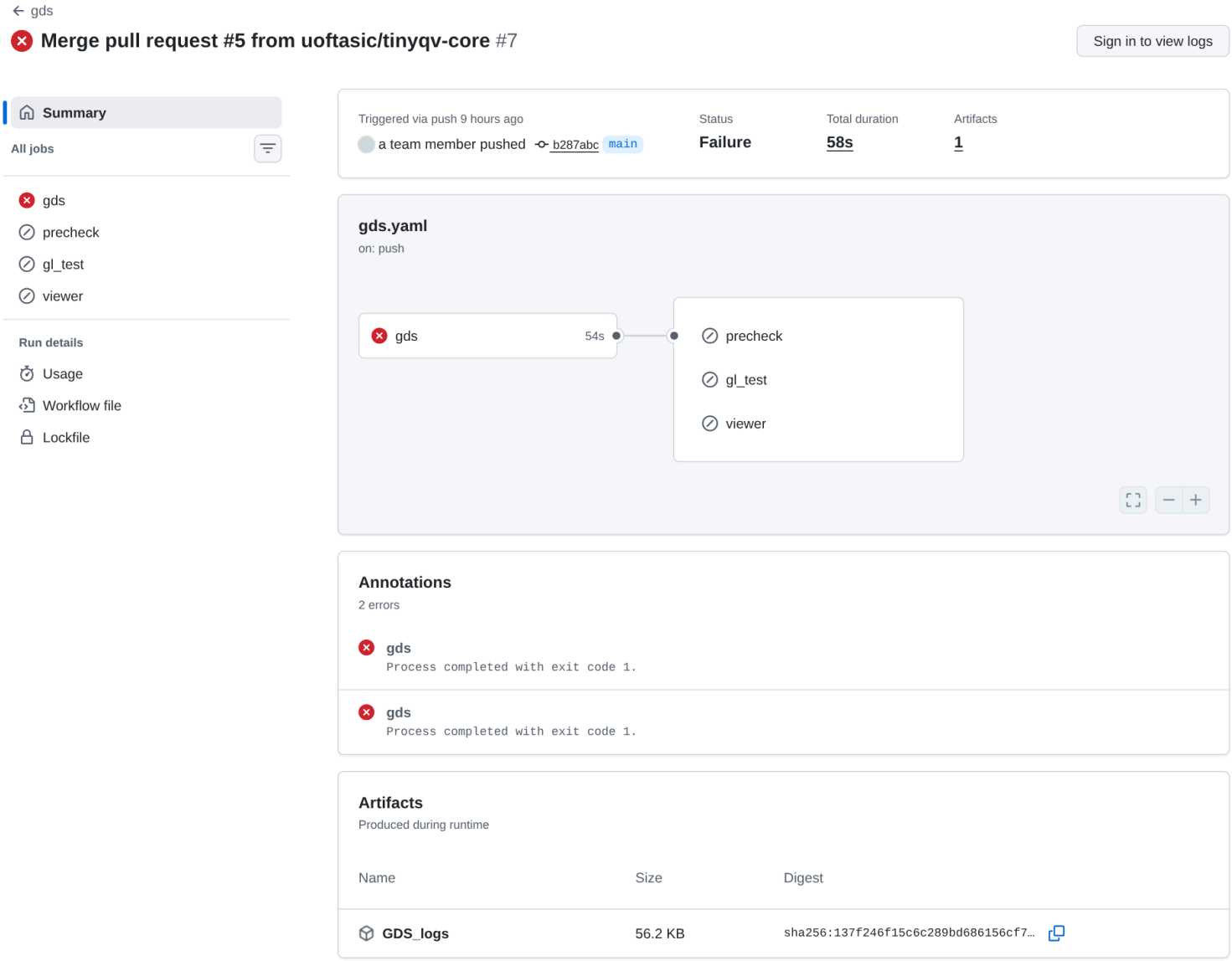


Figure 7.2: A hart-1 gds run that stopped in its first job. The person who pushed is shown as a team member.

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.

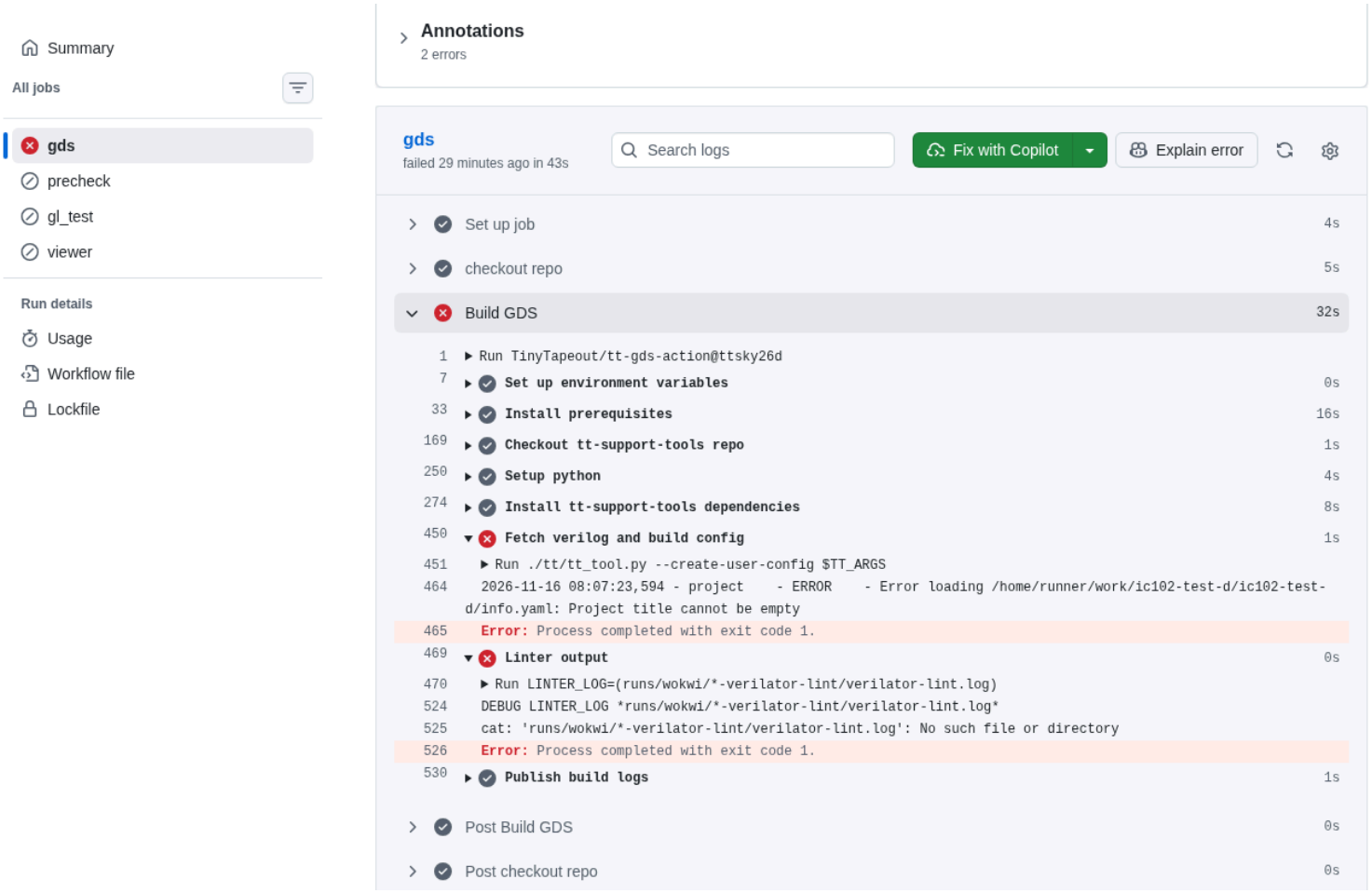


Figure 7.3: The same failure in the course's practice repository, where only the title was empty: the failed "Build GDS" step opens by itself, with its two failed sections, "Fetch verilog and build config" and "Linter output", expanded.

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.

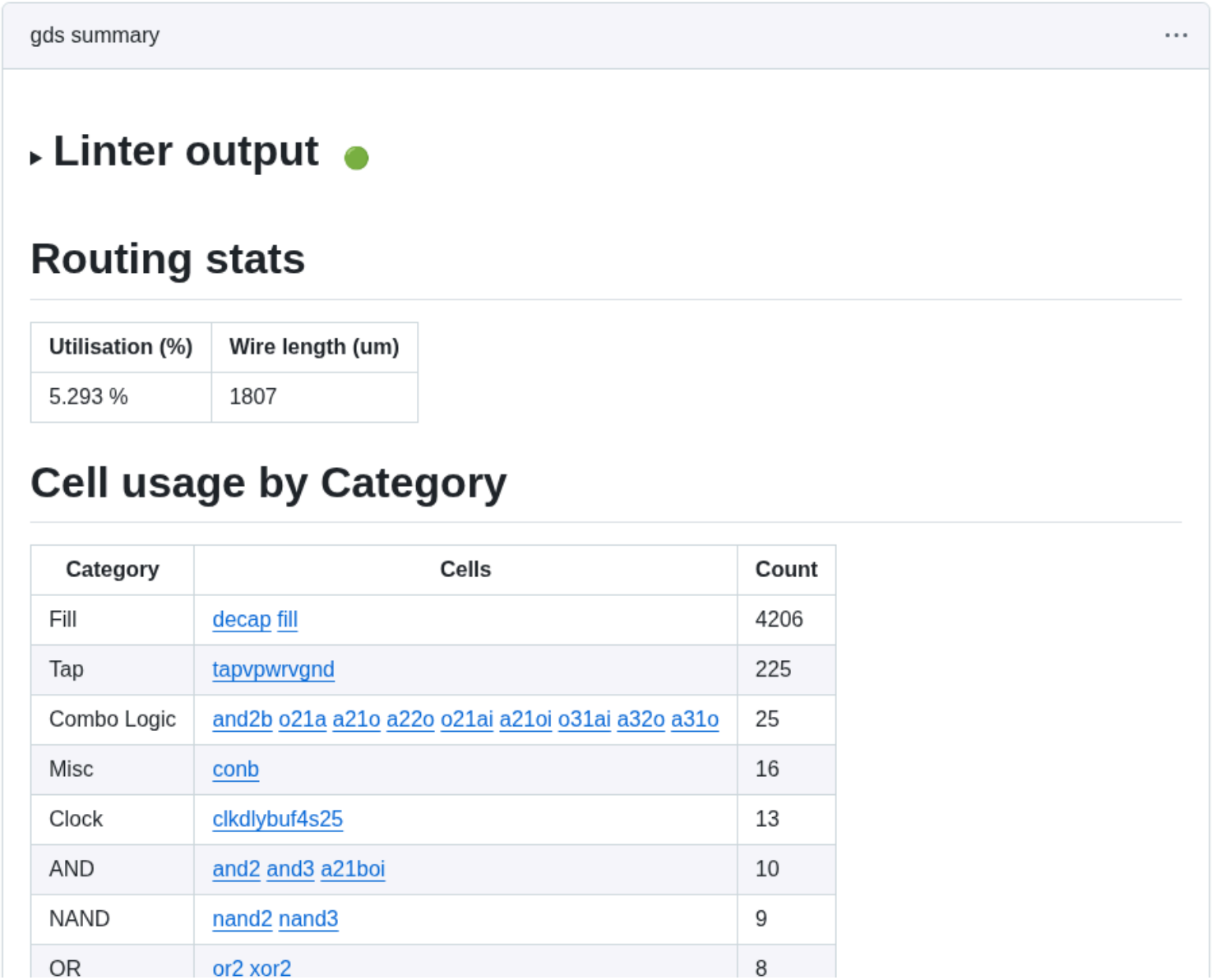


Figure 7.4: The gds job's summary on a run's Summary page, from the course's practice repository: the linter's result, the routing stats and the cell usage.

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:

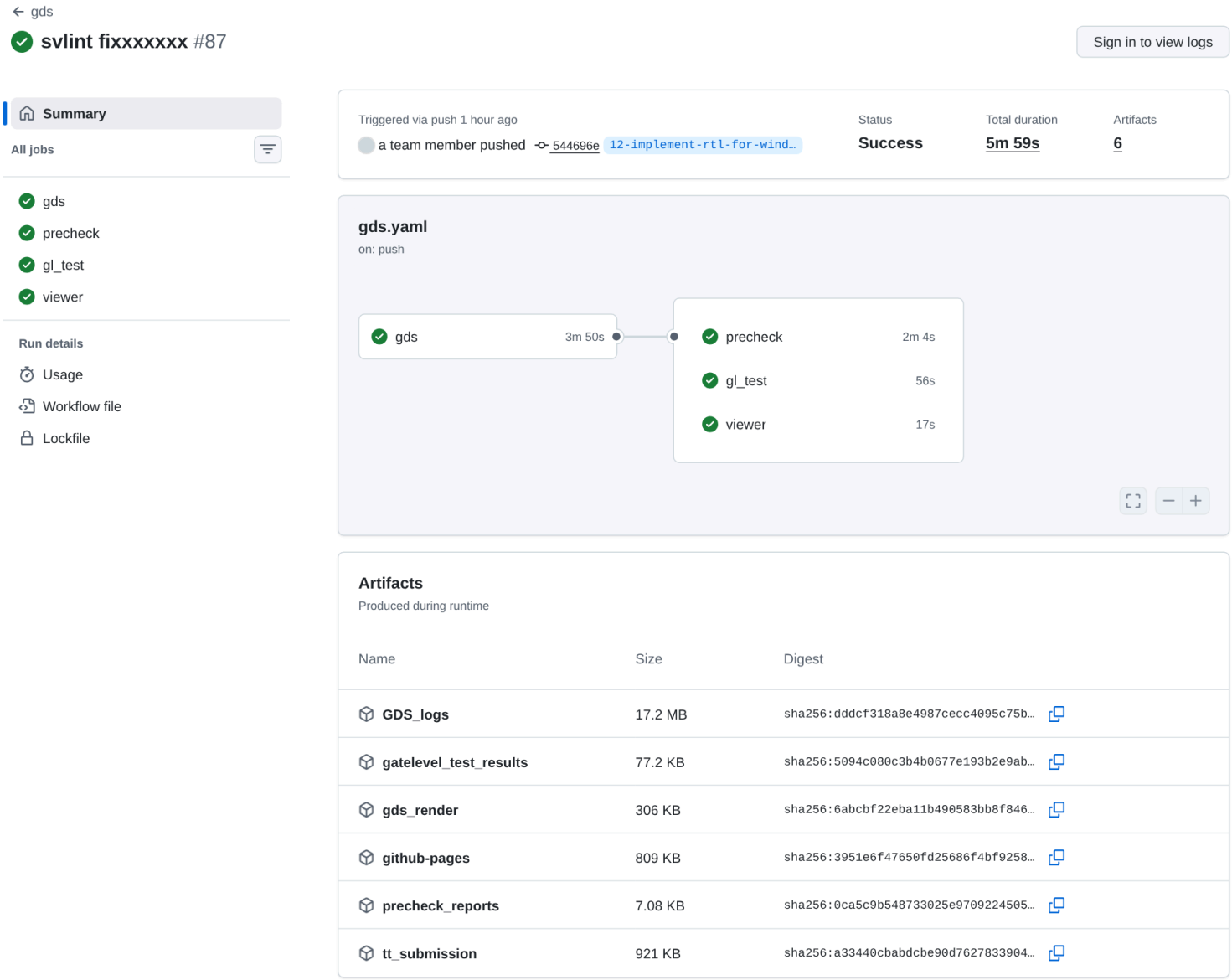


Figure 7.5: A fully green sentry gds run and its six artifacts. The person who pushed is shown as a team member.

`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.

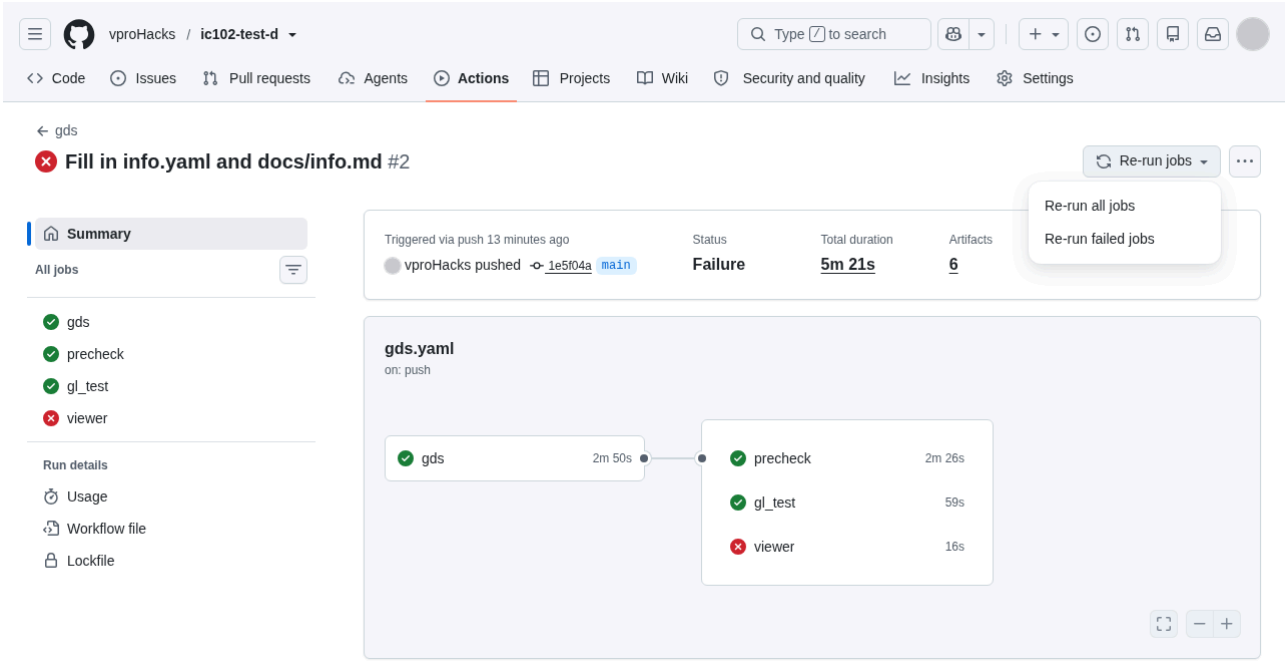


Figure 7.6: The Re-run jobs menu on a failed run.

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.
```

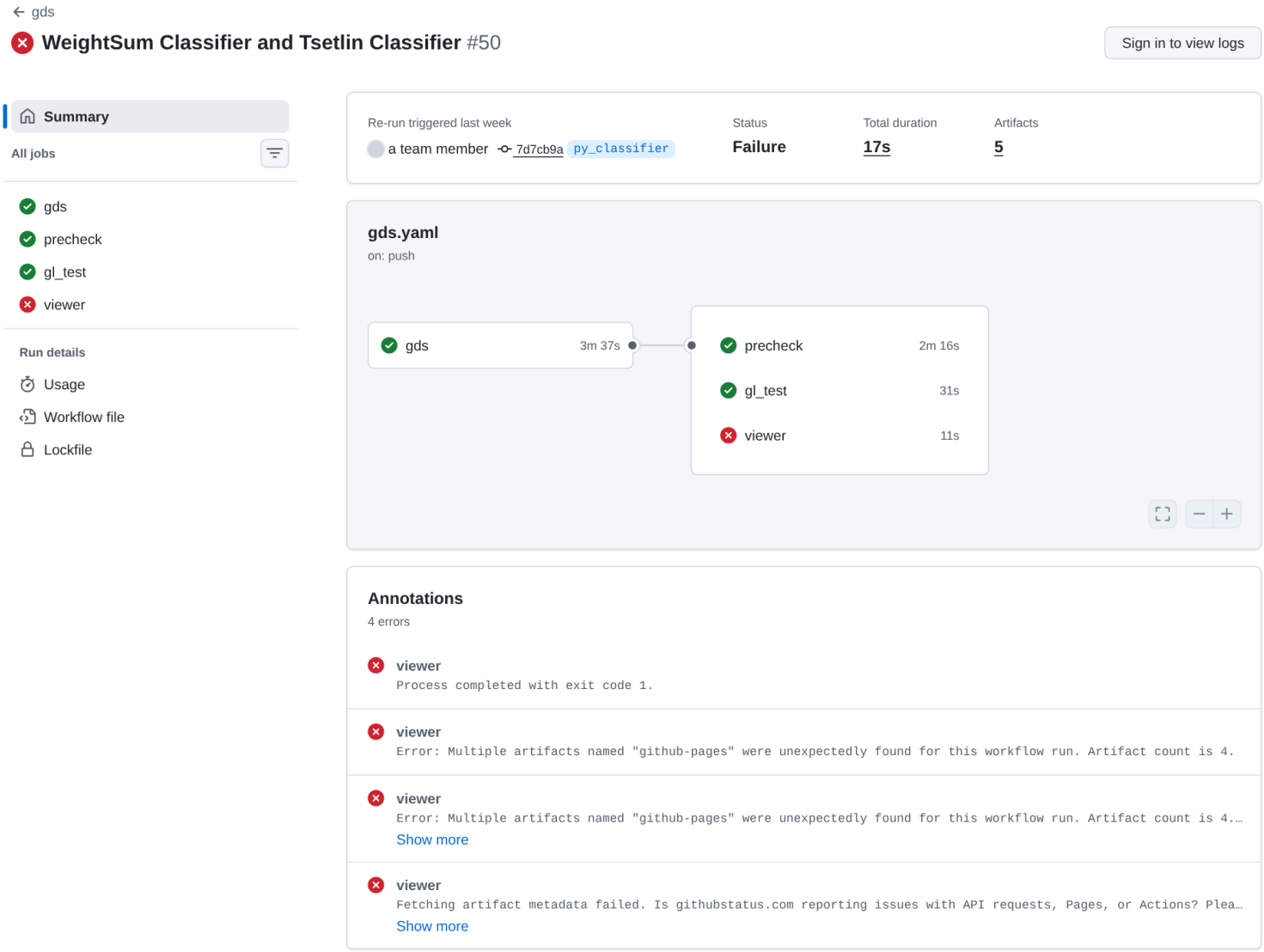


Figure 7.7: The sentry run after its fourth attempt at the viewer job. The person who re-ran it is shown as a team member.

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):

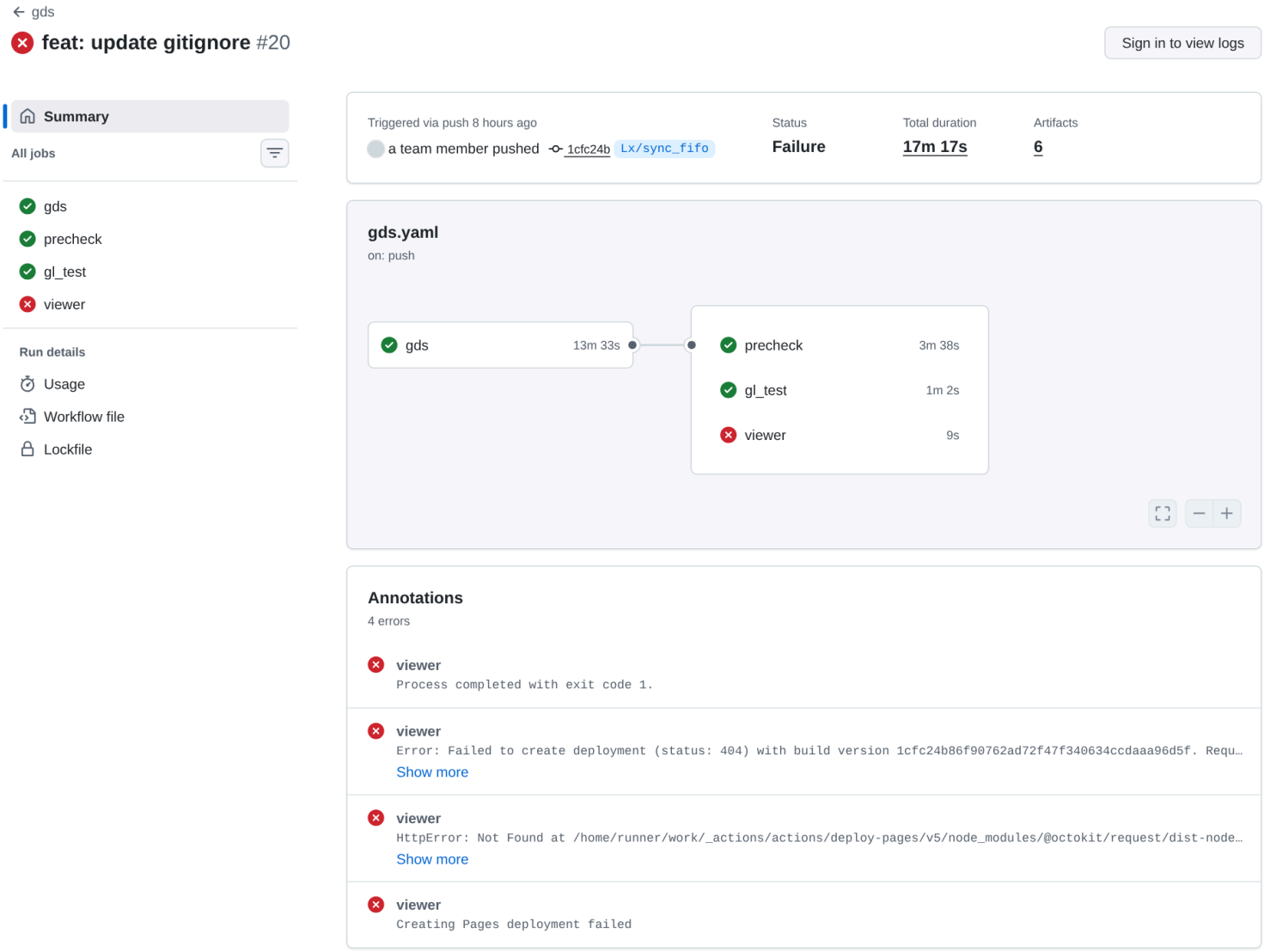


Figure 7.8: A protocol-emulator gds run where only the viewer job failed. The person who pushed is shown as a team member.

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”.

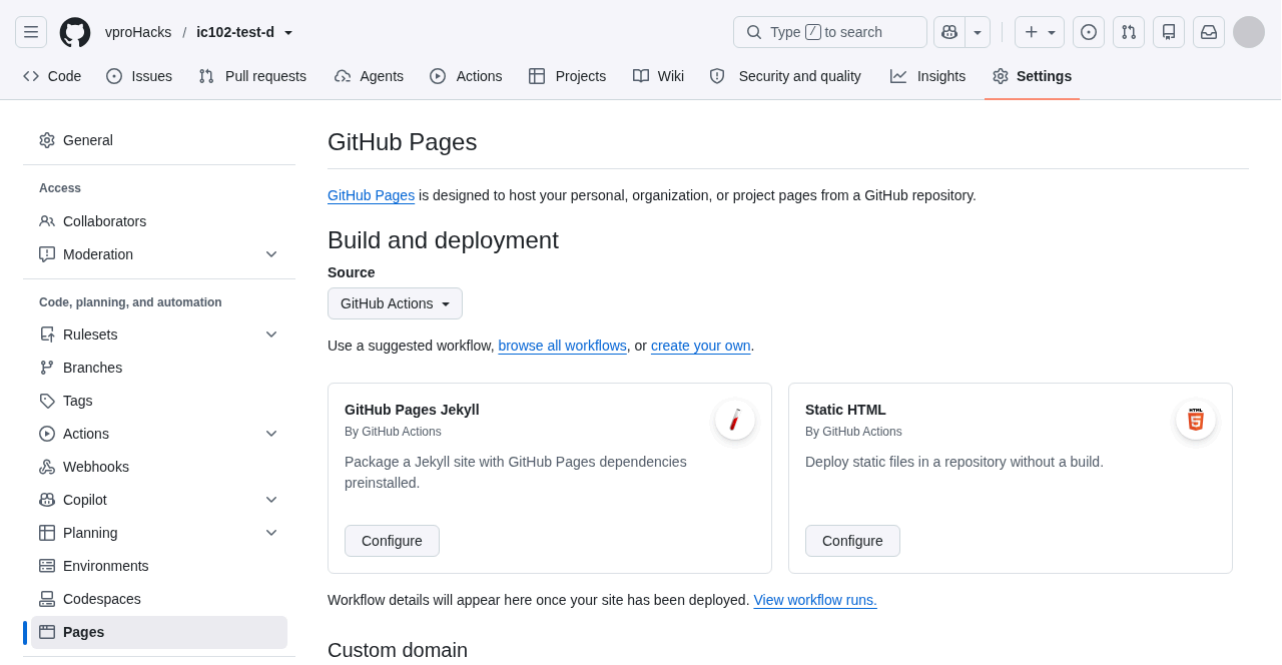


Figure 7.9: Settings › Pages, with the source set to GitHub Actions.

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 analog workflows

This page answers: *what does each job in the analog template's two workflows do, and what does a red cross on each one usually mean?*

Reading a workflow file explains how to read any workflow. This page goes through the two the analog template gives you, `docs.yaml` and `gds.yaml`, as they are at the template's commit `301590b` (Sep 15, 2026), with every action at the tag `ttsky26d`. The steps quoted are from Tiny Tapeout's actions at that tag. You'll see each of the failures described here in Lab 01 and Lab 02.

The biggest difference from the digital template is that nothing on the runner builds your layout. A digital project's `gds` job runs LibreLane for several minutes to turn Verilog into a layout. In an analog project you draw the layout yourself, on your own machine, and commit the result, and the runner only copies it, checks it and publishes it. The analog template also has no `test` workflow and no `gl_test` job, because there's no Verilog to simulate: `src/project.v` is an empty module that only declares the pins.

docs

`docs.yaml` has one job, `docs`, with two steps: it checks out your repository, then runs Tiny Tapeout's docs action. Inside that action, the checking is done by one step, "Check info.yaml and docs/info.md", which runs `tt_tool.py --check-docs` from Tiny Tapeout's support tools. That program reads `info.yaml` with the same validator the rest of Tiny Tapeout uses, looks through `docs/info.md` for the template's placeholder sentences, and prints every problem it finds before it fails:

```
## Errors in info.yaml
- 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
```

That's from the team's `crosswalk` on Nov 8, 2026. If the check passes, the action goes on to build your page of the datasheet as a PDF, with Pandoc and Typst, and uploads it as an artifact called `PDF`.

The `docs` job runs in parallel with `gds` and finishes first, in about half a minute, so it's usually the first red cross you see. It doesn't look at your layout at all. The course's `make docs` runs the same check function, from the same commit of the support tools, on your own machine.

gds

`gds.yaml` has three jobs. The first, `gds`, packages your layout; the other two, `precheck` and `viewer`, each start with `needs: gds`, so they run only if it passes, and then they run side by side.

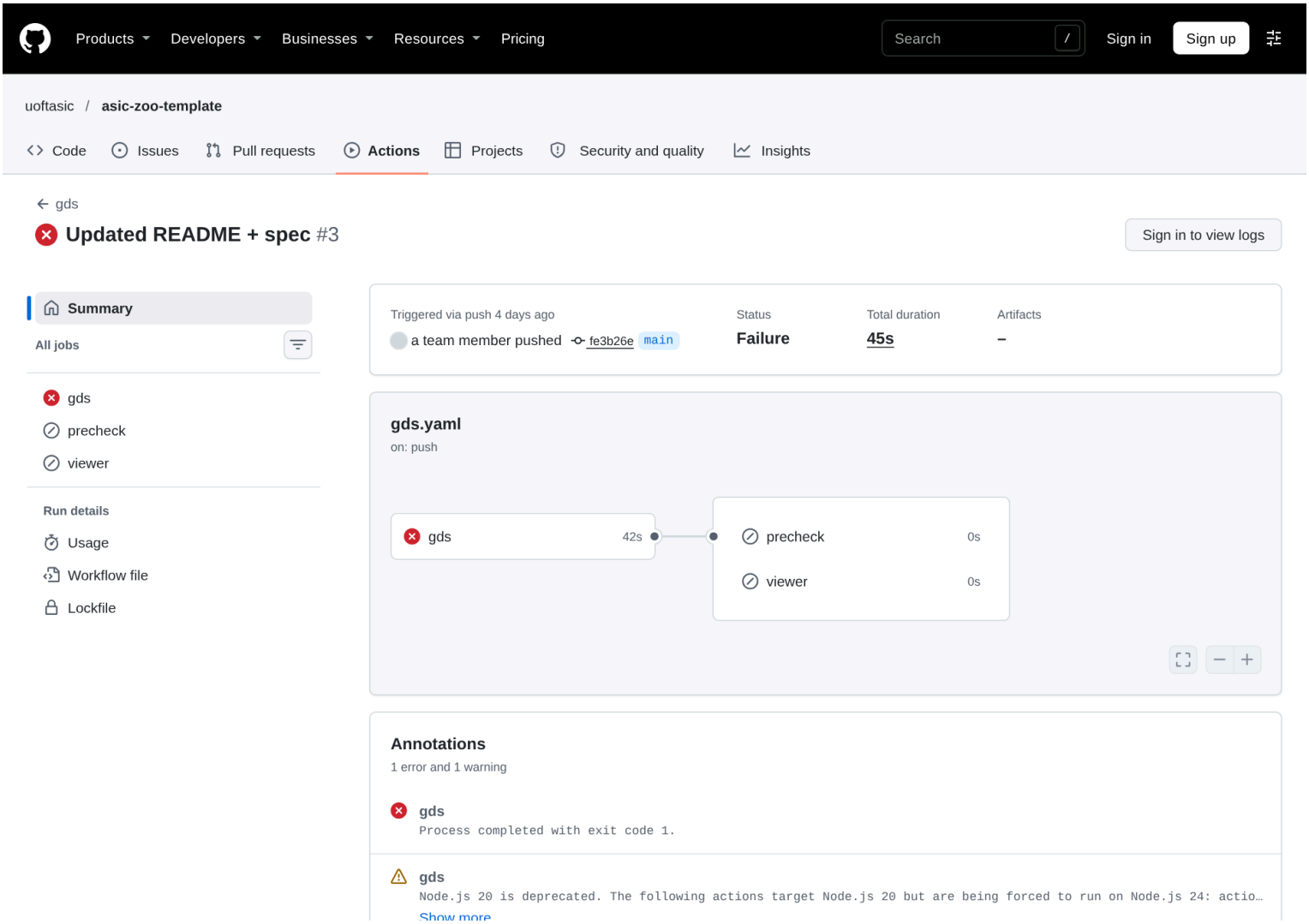


Figure 8.1: The gds workflow in uoftasic/asic-zoo-template on a push with no layout: gds failed after 42 s, and precheck and viewer were skipped. The person who pushed is shown as a team member.

The gds job

Two steps of the job are in your repository's `gds.yaml`. "Read top module name" reads `top_module` from `info.yaml` with `yq`, and "Create and publish the GDS artifact" passes it to Tiny Tapeout's `custom_gds` action, along with three paths built from it: `gds/<top_module>.gds`, `lef/<top_module>.lef` and `src/project.v`. Inside `custom_gds`:

1. It writes two small files, `commit_id.json` (which repository, commit and run this is) and `pdk.json` (`sky130A`), into a folder called `tt_submission`.
2. It installs KLayout 0.29.12.
3. "Copy GDS / LEF files to tt_submission" copies your GDS and LEF into `tt_submission`, under your top module's name, writes an OASIS copy of the GDS with KLayout's `strm2oas`, and copies `src/project.v` in as `<top_module>.v`.
4. It uploads `tt_submission`, together with `info.yaml`, `LICENSE` and `docs/`, as the `tt_submission` artifact. The other two jobs start from that artifact, not from your repository.
5. It renders a PNG picture of the layout and uploads it as the `gds_render` artifact. If the render fails, the step prints a warning and the job carries on.

Step 3 is where a new analog repository first fails, with the GDS file missing:

```
cp: cannot stat 'gds/tt_um_example.gds': No such file or directory
##[error]Process completed with exit code 1.
```

That's from the team's `asic-zoo-template`, which had no layout yet. You get the same line when `top_module` in `info.yaml` doesn't match the file's name, because the path is built from `top_module`. Nothing in this job checks your GDS (it only copies, converts and renders it), so a layout that's wrong in every way passes it as long as the two files exist.

The precheck job

`precheck` downloads the `tt_submission` artifact and checks it. Its own steps install what it needs, and that's most of its run time:

1. It checks out Tiny Tapeout's support tools from their `main` branch, which holds `precheck.py`.
2. It installs the `sky130A` PDK at commit `0536d02d…`, fixed in the action, with Tiny Tapeout's `ciel` action.
3. It builds Magic and KLayout through Nix, at versions fixed by two Nix package commits: Magic 8.3.568 and KLayout 0.30.8.
4. "Run precheck" runs `precheck.py` on your GDS, which runs all 15 checks for `sky130A` and carries on after one of them fails. What precheck checks goes through them.
5. Three more steps run whether or not the checks passed (`if: always()`). "Test Summary" and "Capture test results" put precheck's table of results on the run's summary page, and "Upload reports" uploads the reports folder as the `precheck_reports` artifact.

When `precheck` is red, start with the table on the run's summary page; it names every failed check with its message. Then read the log of "Run precheck" for the details. Once in a while the job fails before it checks anything, in a step that downloads something. On Nov 9, 2026 a public project's `precheck` failed in "Install sky130A PDK" with `The read operation timed out`, which says nothing about the layout; re-running the failed job is the right response to that.

In the team's repositories, `precheck` took between 91 and 315 seconds. In one public analog project's run on Nov 14, 2026 it took 2 min 21 s, after a `gds` job of 1 min 12 s.

The viewer job

`viewer` downloads `tt_submission` and `gds_render`, writes a one-page website that sends its visitor to Tiny Tapeout's 3D viewer with your layout loaded, and deploys that page with GitHub Pages. It needs Pages turned on with GitHub Actions as the source, and a new repository has Pages off, so `viewer` fails on the first run that gets that far:

```
##[error]Error: Failed to create deployment (status: 404) with build version ...
Ensure GitHub Pages has been enabled: https://github.com/uoftasic/protocol-emulator/
settings/pages
```

The fix is to turn Pages on (Settings › Pages › Build and deployment › Source: **GitHub Actions**) and then start a new run, with **Run workflow** or a new push, rather than re-running the failed one: at the `ttsky26d` tag, a re-run of `viewer` within a day of the first attempt fails with `Multiple artifacts named "github-pages"`. Reading a workflow file has the rest of that log, and its section Re-running a run explains why a re-run fails that way. When it works, the job's summary has links to the 3D viewer and to Tiny Tapeout's GDS explorer, and your Pages site redirects to the viewer.

`viewer` doesn't depend on `precheck`, so it passes even when precheck fails, and a run can end with `precheck` red and `viewer` green:

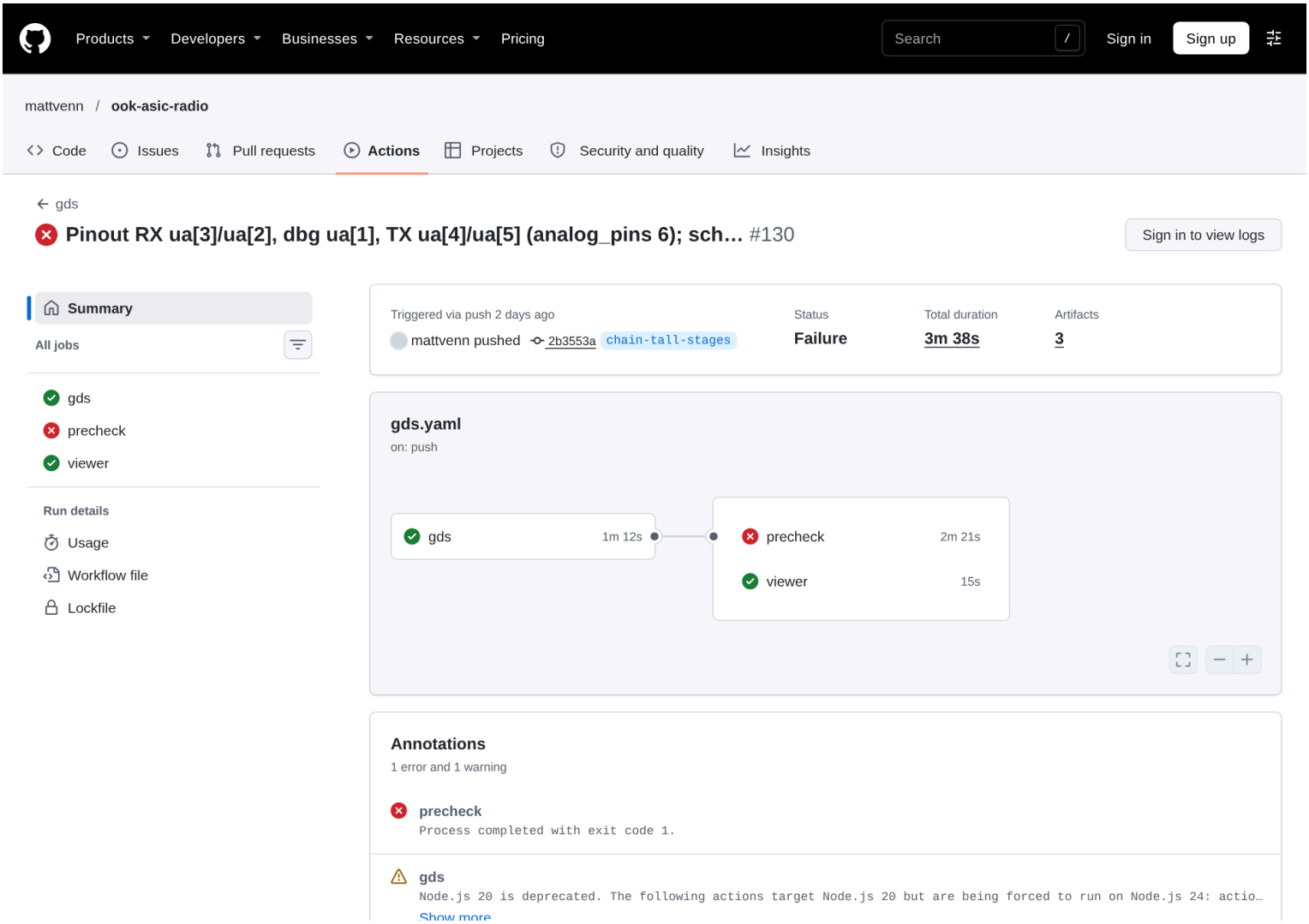


Figure 8.2: A public analog project's gds run on Nov 14, 2026: gds and viewer green, precheck red after 2 min 21 s.

In this course a workflow counts as passing only when all three jobs are green.

Which push runs what

The template's two workflows start on every push, to any branch, and on `workflow_dispatch`, the **Run workflow** button. There is no `pull_request` trigger. A pull request between two branches of your own repository still shows checks, though: every push to the pull request's branch started runs of both workflows, and GitHub shows the runs for the branch's latest commit.

Every push runs `viewer` too, whichever branch it's on, so your Pages site shows the layout from the last run whose `viewer` job deployed, not necessarily `main`'s. The team's `sentry` deploys from feature branches routinely; its `gds` run of Nov 16, 2026 on the branch `12-implement-rtl-for-window-controller-module` (run 43796948508) passed with `viewer` green.

Pull requests in a shared repository

In Lab 04 your partner's change reaches your review repository as a pull request from a fork, and that needs two changes to the template's `gds.yaml` and `docs.yaml`, which the course's sandbox makes:

- **A `pull_request` trigger.** GitHub's documentation says "Workflows don't run in forked repositories by default", so a push to your fork starts nothing. A pull request from the fork into the repository it came

from does start that repository's `pull_request` workflows, which check the commit GitHub would make if the pull request were merged ("`GITHUB_SHA` for this event is the last merge commit of the pull request merge branch"). The sandbox also limits its `push` trigger to `main`.
- **`viewer` skipped on pull requests**, with `if: github.event_name != 'pull_request'`. In a pull request from a fork, "The `GITHUB_TOKEN` has read-only permissions", so a deploy would fail, and a pull request shouldn't replace `main`'s page anyway. A skipped job doesn't fail the run.

GitHub's documentation adds one more rule that you'll meet in Lab 04: "Workflows will not run on `pull_request` activity if the pull request has a merge conflict." Once someone else's change to the same lines is merged, your pull request's checks stop until you resolve the conflict.

Check yourself

1. A `gds` run shows the `gds` job red after 40 seconds, and `precheck` and `viewer` grey. Which step do you open first, and what are the two likely causes?

   Answer: A1, in Answers at the end.

2. Your `precheck` is red with a ❌ on "Magic DRC", and your `viewer` is green. Does the 3D viewer show your layout?

   Answer: A2, in Answers at the end.

3. Why can't you run the template's `gds` workflow on a pull request from a fork without changing it?

   Answer: A3, in Answers at the end.

Answers

A1. Check yourself

"Copy GDS / LEF files to tt_submission", inside "Create and publish the GDS artifact". It's the step that copies your GDS and LEF, and a short failure there means one of them isn't where the workflow looks: no layout committed yet, or a `top_module` in `info.yaml` that doesn't match the files' names. The other two jobs are grey because they `need` this one.

A2. Check yourself

Yes. `viewer` needs only `gds`, so it published the layout that `gds` packaged, DRC errors and all. A green `viewer` says nothing about whether the layout is right.

A3. Check yourself

It has no `pull_request` trigger, and the fork's own workflows don't run by default, so nothing would run at all.

9. Lab 02: Read the red X

You'll put five mistakes into your repository from Lab 01, one at a time. Each is one that a real analog project has made, or one that a precheck rule exists to catch. For each one, you predict which job will fail and push the commit. Then you find the failed job, the step inside it and the first error line on GitHub, and write them down in `DIAGNOSIS.md` before you fix the mistake.

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

Before you start

- Read What a cloud runner is, Reading a workflow file and The analog workflows.
- Finish Lab 01: every workflow green on your latest commit.

This lab takes about 1 h 10 min, about a quarter of an hour for each breakage, most of it spent waiting for runs. When we ran this lab on a test repository on Nov 16, 2026, each `gds` run took between 2 min 28 s and 3 min 11 s from the push until precheck had finished, so expect about three minutes.

Part 1: The routine

Set up the file you'll fill in, and learn the routine you'll repeat five times.

1. Copy the template into your repository and commit it:

```
cd /foss/designs/projects/ttsky-ic102-a
cp /foss/designs/modules/ic102-a/labs/lab-02-read-the-red-x/DIAGNOSIS.md .
git add DIAGNOSIS.md && git commit -m "Lab 02: add DIAGNOSIS.md" && git push
```

2. For each breakage N, from 1 to 5, do this:
   a. Apply it, and read what it changed:

```
/foss/designs/modules/ic102-a/labs/lab-02-read-the-red-x/break.sh N .
git diff --stat HEAD
git diff HEAD -- info.yaml layout/wires.tcl src/project.v
```

   b. In `DIAGNOSIS.md`, under "Predicted", write which job you expect to fail first and why. You know enough from The analog workflows to make a real guess.

   c. Commit and push it: `git commit -am "Lab 02: breakage N"` and `git push`.

   d. On GitHub, find the failed run, the job, the step inside it, and the first error line, and write them down. Then write the cause in one sentence.

   e. Fix it, fill in the rest of the entry, commit (`DIAGNOSIS.md` included), push, and wait until the run is green before the next breakage. Most fixes are an undo of the breakage's commit. That commit holds your prediction in `DIAGNOSIS.md` as well as the breakage, so undo it like this:

```
git revert --no-commit HEAD
git checkout HEAD -- DIAGNOSIS.md
```

   The first line puts every file back as it was before the breakage, and the second puts your notes back. A plain `git revert HEAD` would commit the undo straight away, with your prediction gone from `DIAGNOSIS.md`. Then finish the entry and commit, with `git commit -am "Lab 02: fix breakage N"`. `break.sh` won't run while you have uncommitted changes, so each breakage gets a commit of its own.

You can check a fix before you push with `make docs` and `make precheck`, which run the same checks as the `docs` and `precheck` jobs. Don't use them to skip the prediction, though, because this lab is for getting faster at reading the runner's answer.

3. When a check fails, precheck doesn't stop. It runs all 15 and prints the table at the end, so on GitHub the quickest way in is the table on the run's summary page. When something failed, the summary starts with a box headed "Some tests failed", which counts the checks that passed and failed and shows the failure as a Python traceback; the table below it has the same results with plainer messages. Find the failed check in the table first, then look for the first error line in the log of the step that failed.

   Every precheck log, green or red, has a block near the start of "Run Tiny Tapeout Precheck" that begins with `error:` and ends with these two lines:

```
        error: file 'nixpkgs' was not found in the Nix search path (add it using
$NIX_PATH or -I)
will use bash from your environment
```

   Precheck's setup prints it and carries on, so skip past it when you look for the first error.

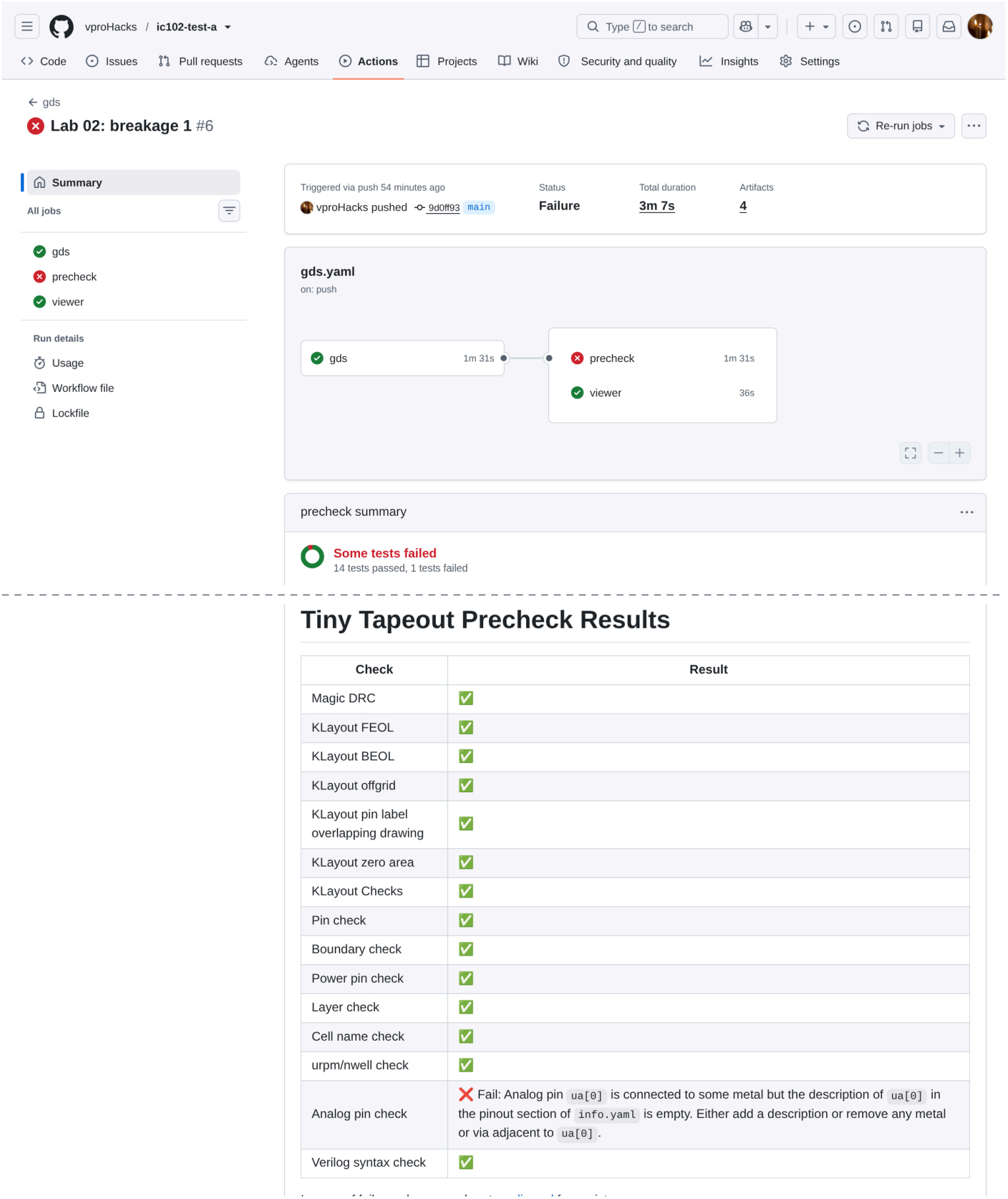


Figure 9.1: Precheck's table on a run's summary page, from breakage 1 in our test repository: the "Some tests failed" box at the top of the precheck summary, and further down, the table with its one ❌ row.

Check: `DIAGNOSIS.md` is on GitHub and both workflows are green.

Part 2: Breakage 1

What does an empty pinout break?

1. Apply breakage 1, read the diff, predict, commit and push.
2. Find the failures. You should see two red runs this time, and `docs` red first, because it's a single short job (27 s in our test), while the `gds` workflow runs `gds` and then `precheck`. The `docs` job's step "Build docs" fails in its inner step "Check info.yaml and docs/info.md" with:

   ```
   ## Errors in info.yaml
   - Please fill in the 'pinout' section
   ```

   That's the line the team's `crosswalk` got on every push while its pinout was empty. About three minutes later `precheck` fails too, with one ❌ row in its table:

   ```
   | Analog pin check | ❌ Fail: Analog pin `ua[0]` is connected to some metal but the
   description of `ua[0]` in the pinout section of `info.yaml` is empty. Either add a
   description or remove any metal or via adjacent to `ua[0]`. |
   ```

3. Why does precheck care about the pinout, which is documentation?

   Answer: A1, in Answers at the end.

4. Fix it by putting your two descriptions back. The undo from Part 1, step 2.5 does that, and so does editing `info.yaml`; either is fine here.

Part 3: Breakage 2

What happens when the layout touches an analog pin that `info.yaml` doesn't claim?

1. Apply breakage 2. This one changes `layout/wires.tcl` and rebuilds `gds/` and `lef/`, so `git diff --stat HEAD` lists the GDS as `Bin`, and the diff of `wires.tcl` is the only place you can read what changed. Predict, commit and push.
2. You should see `docs` green and `precheck` red, with:

   ```
   | Analog pin check | ❌ Fail: Analog pin `ua[2]` is connected to some metal but
   `analog_pins` is set to 2 in `info.yaml`. Either increase `analog_pins` to at least
   3, or remove any metal or via adjacent to `ua[2]`. |
   ```

3. The message offers two fixes. Which is right, and how would you know?

   Answer: A2, in Answers at the end.

Part 4: Breakage 3

What does precheck say when the name on the outside of the file doesn't match the cell inside?

1. Apply breakage 3. It renames your project the way someone might by hand: a new `top_module` in `info.yaml`, the module in `src/project.v`, and the two files in `gds/` and `lef/`, with the layout itself left as it was. Predict, commit and push.
2. You should see three ❌ rows:

   ```
   | KLayout Checks | ❌ Fail: Top macro name mismatch: expected tt_um_yourname_ic102_v2,
   got tt_um_yourname_ic102 |
   | Pin check | ❌ Fail: Wrong cell at GDS top-level |
   | urpm/nwell check | ❌ Fail: Klayout nwell_urpm failed |
   ```

If you then go to the log for the first error line, you'll find 53 lines like the one below before any other error from the checks (on GitHub, they come after the Nix block from Part 1, step 3). This one is from `make precheck`; on GitHub the path starts with the runner's work folder instead of `/foss/designs/projects`:

```
ERROR: Pin clk not found in /foss/designs/projects/ttsky-ic102-a/tt_submission/
tt_um_yourname_ic102_v2.lef
```

3. Which of these is the cause, and which are consequences of it?

   Answer: A3, in Answers at the end.

4. Magic DRC is ✅ in that table. Did it check your layout?

   Answer: A4, in Answers at the end.

5. Fix it with the undo from Part 1, which reverses the rename. If you'd rather keep the new name, set `TOP_LEVEL_CELL` in `layout/init.tcl` to it as well and run `make`, so that the cell inside the GDS changes too.

Part 5: Breakage 4

Can a layout pass DRC and still fail precheck?

1. Apply breakage 4, which adds a 10 µm square on metal 5 in an empty corner. Predict, commit and push.
2. You should see one ❌ row, with Magic DRC and every KLayout DRC deck green:

   ```
   | KLayout Checks | ❌ Fail: Forbidden layer met5.drawing found in /foss/designs/
   projects/ttsky-ic102-a/tt_submission/tt_um_yourname_ic102.gds |
   ```

   (That's `make precheck`'s row; on GitHub the path is the runner's.)
3. Why does a square that passes every DRC deck fail precheck?

   Answer: A5, in Answers at the end.

4. Fix it with the undo from Part 1.

Part 6: Breakage 5

What does one word in `info.yaml` change?

1. Apply breakage 5. It changes one line, `uses_vapwr: false` to `true`. Predict, commit and push. Which rows of precheck's table do you expect to go red?
2. You should see four:

   ```
   | Pin check | ❌ Fail: Inconsistent die area between LEF and template DEF: (161000,
   225760) != (145360, 225760) |
   | Boundary check | ❌ Fail: Shapes outside project area |
   | Power pin check | ❌ Fail: Verilog doesn't contain VAPWR |
   | Analog pin check | ❌ Fail: Analog pin `ua[0]` is not connected to any adjacent
   metal but `analog_pins` is set to 2 in `info.yaml`. Either wire up `ua[0]` to your
   design or decrease `analog_pins` to 0. |
   ```

3. Your layout didn't change. Why do the pin check and the boundary check both say it's the wrong size?

   Answer: A6, in Answers at the end.

4. Fix it with the undo from Part 1, or by setting `uses_vapwr: false` again.

Check: `DIAGNOSIS.md` has five entries, and the run after each fix is green.

Part 7: A red X that isn't yours

Not every failure comes from your files. On Nov 9, 2026, the `precheck` job of a public project, `erik1567/ttsky26d-VCO`, failed in the step "Install sky130A PDK" before any check had run (run 42570954316):

```
Version 0536d02d875c8f67dd7cca3902ac457e62f20005 not found locally, attempting
to download…
The read operation timed out
##[error]Process completed with exit code 255.
```

1. What would you do if this happened on your repository?

   Answer: A7, in Answers at the end.

2. In `DIAGNOSIS.md`, under "A mistake no workflow catches", write down one value in your repository that could be wrong while every workflow still passes. Then check your answer against the one below.

   Answer: A8, in Answers at the end.

Finishing

You're done when `DIAGNOSIS.md` in your repository has five filled-in entries, each naming a green commit that fixed its breakage, and the latest commit on `main` has both workflows green.

Submit: nothing for this lab.

Next on the Course path: read The analog layout kit, What precheck checks and Your tools and the runner's tools, then do Lab 03: A layout that passes precheck.

Answers

A1. Part 2: Breakage 1

Precheck reads `info.yaml` to learn which analog pins you use, because a `ua` pin is connected to the outside world only if `analog_pins` says so (the template's comment above the analog pins: "make sure to also set"analog_pins" above, else the pins won't be connected"). Its analog pin check requires the metal at each `ua` pin, `analog_pins` and the pinout to agree, and a pin with metal on it and no description is one of the four ways they can disagree.

A2. Part 3: Breakage 2

It depends on what you meant, which precheck can't know. If the metal on `ua[2]` is a mistake, remove it. If you meant to use a third pin, set `analog_pins: 3` and describe `ua[2]` in the pinout. Here the stub is a mistake, so remove it, with the undo from Part 1, or by deleting the line from `layout/wires.tcl` and running `make`.

A3. Part 4: Breakage 3

The cause is the mismatch: the GDS's top cell, and the LEF's macro, are still called `tt_um_yourname_ic102`, and precheck looks for `tt_um_yourname_ic102_v2`. The pin check looks for the pins of a macro by that name and finds none, which is why it prints a "not found" line for every one of the 53 pins, VGND and VDPWR included. The urpm/nwell check fails because the KLayout script it runs is told to look at a top cell that isn't there. So in this case the first error line in the log is a consequence, and the table is the better guide.

A4. Part 4: Breakage 3

No. `make precheck` shows what Magic printed:

```
Cell tt_um_yourname_ic102_v2 couldn't be read
No such file or directory
Creating new cell
...
No errors found.
```

Magic was asked for a cell that the GDS doesn't contain, so it made an empty one and checked that. A check that ran on an empty cell still reports ✅, so when the rows around a ✅ are red, read the log behind it too. You'll meet this again in Lab 05.

A5. Part 5: Breakage 4

Metal 5 is a legal layer in SKY130, so no DRC deck objects to a clean square of it. Tiny Tapeout's analog specs forbid it in your tile, though: "You are not allowed to use the metal 5 layer in your design." That's a rule of Tiny Tapeout's, not of the process, and precheck enforces it with its forbidden layer check.

A6. Part 6: Breakage 5

`uses_vapwr: true` tells precheck to compare your layout with a different template, the `_3v3` one, which has a VAPWR pin and is 145.36 µm wide instead of 161.00. Against that template your 161 µm tile is too wide, its pins are in the wrong places, and `ua[0]` is at a different x, so the stub you drew is no longer on it. The power pin check reads `src/project.v`, where the template left `VAPWR` commented out. All four rows come from the one setting.

A7. Part 7: A red X that isn't yours

Re-run it. The step failed while downloading the PDK from a server that didn't answer in time, which has nothing to do with your commit, and nothing you push would fix it. Use "Re-run failed jobs": that re-runs only `precheck`, which downloads the `tt_submission` artifact from the same run and tries again. (The problem with re-running that you met in Lab 01 is in the `viewer` job, and `viewer` isn't re-run here, because it passed.) If it fails the same way again, wait, or ask on Discord whether others see it.

A8. Part 7: A red X that isn't yours

There are many; here are three. Swap the two descriptions in the pinout, so that `ua[0]` claims to be the output: precheck checks only that a used pin has a description, not what it says. Or set `clock_hz` to a wrong number, as `protocol-emulator` did with `clock_hz: 50`: it's an integer, so it passes. Or write a "How to test" in `docs/info.md` that describes a different design. The workflows check only that values are present and well formed; whether they're right is for a reviewer to catch, and Lab 04 is about reviewing.

III 3. A layout that passes

10 The analog layout kit 73
10 The DEF templates 73
10 magic_init_project.tcl 73
10 Tiny Tapeout's rules for an analog layout 74
10 What the course adds 74
10 Check yourself 75
10 Answers 75

11 What precheck checks 77
11 How it runs 77
11 The fifteen checks 77
11 Reading the table and the log 79
11 What precheck leaves to you 79
11 Check yourself 80
11 Answers 80

12 Your tools and the runner's tools . . . 81
12 Three sets of tools 81
12 Why the versions matter 81
12 What the course installs 82
12 When Tiny Tapeout moves the tag 82
12 Check yourself 82
12 Answers 83

13 Lab 03: A layout that passes precheck 85
13 Before you start 85
13 Part 1: A branch and a cell 85
13 Part 2: The power stripes 85
13 Part 3: The output wire 86
13 Part 4: What precheck doesn't see 87
13 Part 5: Through a pull request 88
13 Finishing 91
13 Answers 91

10. The analog layout kit

This page answers: *what does Tiny Tapeout give you to start an analog layout, and what has a layout to contain before precheck will accept it?*

A digital Tiny Tapeout project gets its layout from LibreLane, which places the pins where Tiny Tapeout wants them because Tiny Tapeout configured it to. An analog project's layout is yours, pins included, so Tiny Tapeout publishes the pieces you need to start one in the right shape, in its support tools under `tech/sky130A/def/analog/`. In this course we call those pieces the kit. You'll use them in Lab 01, Part 5, and change their settings in Lab 03.

The DEF templates

A DEF file describes a layout's outline and its pins. The kit has one per tile size, `tt_analog_1x2.def`, `tt_analog_2x2.def` and so on up to `6x2`, each with a `_3v3` variant for projects that use the second analog supply. The analog template's `info.yaml` says "Valid values for analog projects: 1x2, 2x2", and so do Tiny Tapeout's analog specs; the 3x2 to 6x2 templates were added to the kit later, on Apr 18, 2026.

The 1x2 template, the one this course uses, describes a tile 161.00 µm wide and 225.76 µm tall, with 51 signal pins, all on metal 4 and all on the tile's edge:

- **On the top edge**, the digital pins, each 0.30 µm wide and 1 µm tall, 2.76 µm apart, from right to left: `ena`, `clk` and `rst_n`, then `ui_in[0..7]`, `uio_in[0..7]`, `uo_out[0..7]`, `uio_out[0..7]` and `uio_oe[0..7]`. Their centres run from 146.74 µm (`ena`) down to 30.82 µm (`uio_oe[7]`).
- **On the bottom edge**, the eight analog pins `ua[0..7]`, each 0.90 µm wide and 1 µm tall, 19.32 µm apart, `ua[0]` furthest right with its centre at 152.26 µm. Tiny Tapeout's specs say "Even though there are 8 pins in the templates, you can only use the first 6", and "You must use the pins in order, starting from 0."

The DEF also names the two power nets, `VDPWR` (1.8 V) and `VGND`, but doesn't place them. Where they go is up to you, within the rules below.

The `_3v3` templates add a third power net, `VAPWR`, and are narrower: the 1x2 one is 145.36 µm wide, and its analog pins sit at different x positions. So setting `uses_vapwr: true` in `info.yaml` changes a lot at once, because precheck then compares your layout with the `_3v3` template; you'll see this in Lab 02′s breakage 5.

magic_init_project.tcl

The kit's other file is a Magic script, `magic_init_project.tcl`, by Uri Shaked of Tiny Tapeout. It makes the starting point of a layout from a DEF template, and its comments ask you to edit only its settings block:

```
set TOP_LEVEL_CELL     tt_um_analog_example
set TEMPLATE_FILE      tt_analog_1x2.def
set POWER_STRIPE_WIDTH 2um                ;# The minimum width is 1.2um

# Power stripes: NET name, x position. You can add additional power stripes for each
net, as needed.
set POWER_STRIPES {
    VDPWR 1um
    VGND  4um
}
```

Then it does four things:

1. `def read` reads the template, which creates a cell with the tile's outline and every pin, and `cellname rename` gives the cell your top module's name.
2. For each entry in `POWER_STRIPES`, it draws a vertical stripe of metal 4 from y = 5 µm to y = 220.76 µm, `POWER_STRIPE_WIDTH` wide, labels it with the net's name, and makes the label a port with `port use power` (or `ground` for VGND).

3. It saves the cell as `<TOP_LEVEL_CELL>.mag`, Magic's own format.
4. It writes `gds/<TOP_LEVEL_CELL>.gds`, and `lef/<TOP_LEVEL_CELL>.lef` with `lef write -hide -pinonly`, which describes only the outline and the pins and hides the rest.

Those two paths are exactly the ones the template's `gds` workflow reads, so the script's output could go straight into a repository. On its own, though, it isn't a passing layout: the template's `info.yaml` has `analog_pins: 2`, and precheck's analog pin check fails unless `ua[0]` and `ua[1]` each touch some metal.

`TOP_LEVEL_CELL` has to be the same as `top_module` in `info.yaml`. The workflow finds the files by `top_module`, and precheck then checks that the cell inside the GDS has that name; a mismatch is Lab 02's breakage 3.

Tiny Tapeout's rules for an analog layout

Tiny Tapeout's analog specs set the rules a layout has to follow, beyond the process's own design rules. These are the ones this course's layouts touch, in the specs' words where possible:

Rule	What it means for your layout	Checked by
Power pins are "vertical stripes on met4" that "Start within the bottom 10 µm of the module and extend at least to the top 10 µm", with a "Minimum width of 1.2 um"	The kit's stripes run from 5 to 220.76 µm in a 225.76 µm tile, 2 µm wide, so they pass wherever you put them across the tile	precheck's pin check, which measures each power port's layer, width and distance from the top and bottom edges
"You are not allowed to use the metal 5 layer in your design."	Nothing on metal 5, even where it's legal SKY130	precheck's forbidden layer check
"You must use the pins in order, starting from 0."	With `analog_pins: 2`, you use `ua[0]` and `ua[1]`, and nothing touches `ua[2]` or above	precheck's analog pin check
"Connect any unused uo_out, uio_out and uio_oe pins to GND"	24 short wires to VGND	nothing automatic; LVS against a reference that includes them
"If you're using Magic - do not flatten the .mag file"	Keep cells as cells; the course's builds place them as instances	—
"Make sure the version of Magic matches the PDK."	Use precheck's Magic with precheck's PDK; see Your tools and the runner's tools	—

Each analog pin's path to the outside of the chip also has limits ("Resistance: < 500 ohm", "Capacitance: < 5 pf", "Max current: 4 mA"), which matter for a real analog circuit and not for a practice inverter.

The template's `src/project.v` belongs with the layout too. It declares a module with your top module's name and every pin, `VGND` and `VDPWR` included, and nothing inside. The `gds` job copies it into `tt_submission` beside the GDS as `<top_module>.v`, and precheck checks that it parses and that it declares the same power pins as the LEF.

What the course adds

The course's layout kit, which Lab 01 copies into your repository, wraps Tiny Tapeout's two files without changing them:

- `layout/init.tcl` is `magic_init_project.tcl`, and `layout/tt_analog_1x2.def` is the 1x2 template, copied with a note of where they came from in `layout/UPSTREAM.md`. You edit only the settings block of `init.tcl`, as its comments ask.
- `layout/wires.tcl` holds everything your layout adds to the frame: two short stubs on `ua[0]` and `ua[1]` in Lab 01, and in Lab 03 an inverter and its wires. Its commands are Magic's own, with two shorthands, `p` to paint a rectangle and `place` to put a cell down, and a third, `tieoff`, that draws the 24 wires from the digital outputs to a VGND stripe.

- `make` runs Tiny Tapeout's script to make the frame, then `layout/build.tcl`, which adds `wires.tcl` to it and writes `gds/` and `lef/`. Both runs use precheck's Magic 8.3.568 and PDK, from the course tools. Running Tiny Tapeout's script unchanged and adding to its result means you can always see which part of the layout came from Tiny Tapeout and which from you.

`make precheck` runs precheck on the result, `make lvs` compares it with the circuit it should contain, and `make view` opens it in Magic on the desktop. Here's the frame that Lab 01 commits, rendered the way the `gds_render` step renders it: the tile, the pins along both edges, the two power stripes at the left edge, and the two stubs at the bottom right.

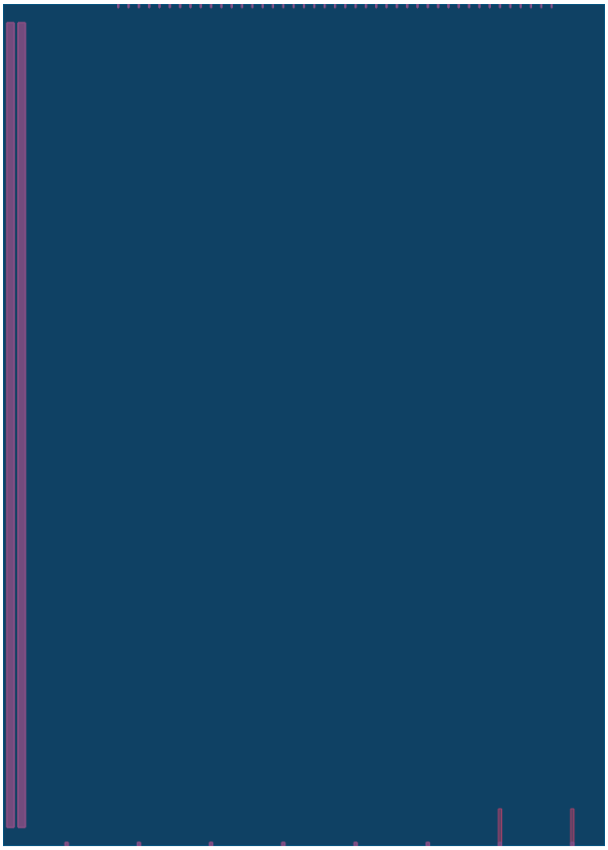


Figure 10.1: The Lab 01 frame: a 161 × 225.76 µm tile with pins on the top and bottom edges, two power stripes near the left edge, and short stubs on ua[0] and ua[1].

Check yourself

1. You set `TOP_LEVEL_CELL` to `tt_um_alice_ic102` and `top_module` to `tt_um_alice_ic102a`. Which job fails, and with what?

   Answer: A1, in Answers at the end.

2. Could you put both power stripes at x = 150 µm and 154 µm?

   Answer: A2, in Answers at the end.

Answers

A1. Check yourself

The `gds` job, at the copy step, because it looks for `gds/tt_um_alice_ic102a.gds` and Magic wrote `gds/tt_um_alice_ic102.gds`: `cp: cannot stat 'gds/tt_um_alice_ic102a.gds': No such file or directory`. (The course's `make` stops before that, with a message that the two names differ.)

A2. Check yourself

Not without trouble, because of the one at 150 µm. Each stripe is 2 µm wide and runs from y = 5 µm up, and `ua[0]` is at x = 151.81 to 152.71 µm with its wire running up from the bottom edge, so a stripe from 150 to 152 µm would sit on top of that wire, both on metal 4. The rules allow a stripe anywhere, so where you can put one depends on your own wiring.

11. What precheck checks

This page answers: *what does each row of precheck's table check, and what does a green table still not tell you?*

Precheck is a Python program, `precheck/precheck.py` in Tiny Tapeout's support tools, and the `precheck` job runs it on the GDS from the `tt_submission` artifact. Tiny Tapeout's own guide says of it that "All the checks need to be green to submit to the chip". The details on this page come from `precheck.py` at commit `9d13ff3`, which is the commit the runner checked out in November 2026, and from runs of it on this course's layouts and on public projects' layouts. You'll see most of these rows fail in Lab 02, and find two mistakes in Lab 03 that none of them catch.

How it runs

Precheck is given one file, `tt_submission/<top_module>.gds`. It finds `info.yaml` by looking in the folders above that file, reads `top_module`, `tiles`, `analog_pins`, `uses_vapwr` and the pinout from it, and picks the DEF template to compare your layout with: `tt_analog_1x2.def` for a 1x2 analog project, or `tt_analog_1x2_3v3.def` if `uses_vapwr` is true. (With `analog_pins: 0` it would pick a digital template instead, so an analog layout has to use at least one analog pin.) The LEF and the Verilog are found next to the GDS, under the same name.

Then it runs every check for `sky130A`, in a fixed order, and doesn't stop at the first failure. At the end it writes the table you see on the run's summary page, with a ✅ or a ❌ and a message for each row, and the job fails if any row failed. The reports go into a folder that the job uploads as the `precheck_reports` artifact.

The fifteen checks

#	Row	What it checks	Fails like this
1	Magic DRC	Magic's full SKY130 design-rule deck, `drc(full)`, on the top cell. Writes `magic_drc.txt`, each broken rule with its boxes	`Magic DRC failed`; the report names the rule, such as `Metal2 spacing < 0.14um (met2.2)`
2	KLayout FEOL	KLayout's SKY130 rule deck for the layers below metal: wells, diffusion, poly, contacts	`Klayout feol failed with 1740 DRC violations` (a public project, Nov 2026)
3	KLayout BEOL	The same deck for the metal and via layers	`Klayout beol failed with 1 DRC violations`
4	KLayout offgrid	Shapes whose corners aren't on the manufacturing grid	
5	KLayout pin label overlapping drawing	Every pin label sits on drawn metal of its own layer	`Klayout pin_label_purposes_overlapping_ failed`
6	KLayout zero area	Shapes of zero area	

#	Row	What it checks	Fails like this
7	KLayout Checks	Three things: the GDS's top cell is named `top_module`; no forbidden layer (metal 5) is used; the boundary layer exists	`Top macro name mismatch: expected …, got …`, or `Forbidden layer met5.drawing found in …`
8	Pin check	Every pin of the DEF template is in the LEF and the GDS, on the right layer, with the right size and place; the die is the template's size; power ports are on metal 4, at least 1.2 µm wide, and reach within 10 µm of the top and bottom edges	`Some ports are missing or have wrong dimensions, see 53 LEF errors above`, `Wrong cell at GDS top-level`, or `Inconsistent die area between LEF and template DEF`
9	Boundary check	Nothing outside the tile, and a boundary rectangle that covers exactly the tile	`Shapes outside project area`, `Boundary layer doesn't cover project area`
10	Power pin check	`VGND` and `VDPWR` are in the Verilog and the LEF, with `USE GROUND` and `USE POWER`; `VAPWR` is there if and only if `uses_vapwr` is true	`Verilog doesn't contain VAPWR`, `VGND does not have a corresponding 'USE GROUND ;'`
11	Layer check	Every layer in the GDS is one Tiny Tapeout allows	`Invalid layers in GDS: …`
12	Cell name check	No cell name contains `#` or `/`	
13	urpm/nwell check	A KLayout rule about the spacing of a resistor implant from n-well	`Klayout nwell_urpm failed`
14	Analog pin check	Each `ua` pin touches metal if and only if it's below `analog_pins`, and has a pinout description if and only if it touches metal	four messages, below
15	Verilog syntax check	`<top_module>.v`, your `src/project.v`, parses in Yosys	`Verilog syntax check failed`

Rows 2 to 6 and 13 are KLayout scripts; row 1 is Magic. Precheck builds its own Magic and KLayout for this, at versions fixed in its Nix file, and installs its own copy of the PDK, which is why the course installs the same versions for you (Your tools and the runner's tools).

The analog pin check has four messages, one for each way the metal, `analog_pins` and the pinout can disagree. Each says which pin and suggests two fixes:

- `Analog pin`ua[2]`is connected to some metal but`analog_pins`is set to 2 in`info.yaml.

- `Analog pin ua[0] is connected to some metal but the description of ua[0] in the pinout section of info.yaml is empty.`
- `Analog pin ua[1] is not connected to any adjacent metal but analog_pins is set to 2 in info.yaml.`
- `… is not connected to any adjacent metal but the description of ua[n] in the pinout section of info.yaml is non-empty.`

"Connected" here means metal 4 reaching into a band from 0.1 to 0.5 µm around the pin's rectangle, or a via 3 on the pin itself. Precheck doesn't follow the metal any further, so a stub that goes nowhere counts, which is why Lab 01's placeholder passes.

Reading the table and the log

Because precheck runs every check, one mistake can turn several rows red. In Lab 02's breakage 5, one line in `info.yaml`, `uses_vapwr: true`, makes four rows fail, because it changes the template every later check compares against. The table on the run's summary page is the quickest place to start: read every ❌ row, and look for the one that explains the others.

The log isn't always ordered the way the table is. Some checks print their errors as they go and some only fail at the end, so the first `ERROR:` line in the log can be a consequence of another check's failure. In Lab 02's breakage 3, a renamed `top_module` with an unchanged GDS, the log's first 53 error lines are the pin check's, one per pin:

```
ERROR: Pin clk not found in /foss/designs/projects/ttsky-ic102-a/tt_submission/
tt_um_yourname_ic102_v2.lef
```

while the cause is in row 7 of the table, `Top macro name mismatch: expected tt_um_yourname_ic102_v2, got tt_um_yourname_ic102`.

A ✅ needs reading as well. In the same breakage, Magic DRC passed. It had been asked to check a cell called `tt_um_yourname_ic102_v2`, which the GDS doesn't contain, and its log says what it did instead:

```
Cell tt_um_yourname_ic102_v2 couldn't be read
No such file or directory
Creating new cell
...
No errors found.
```

It made an empty cell, checked that, and found nothing wrong, so the table shows a ✅ for Magic DRC, and only the log tells you that the check ran on an empty cell. You'll write a script in Lab 05 that has to tell a pass like this one from a real pass.

What precheck leaves to you

Precheck checks that a layout is legal and fits the tile. It doesn't check that it's the circuit you meant, and several things a reviewer cares about are outside every one of its rows:

- **Whether the layout is your circuit.** There's no LVS (layout versus schematic) in precheck, so nothing compares your layout with a netlist. In Lab 03 you'll make a short between an inverter's input and output: Magic DRC finds 0 errors, all 15 rows are ✅, and only `make lvs` objects. The course runs LVS with Magic's extractor and netgen, on your machine, against a reference netlist kept in your repository.
- **The tie-offs.** Tiny Tapeout's specs say "Connect any unused uo_out, uio_out and uio_oe pins to GND". Nothing in precheck looks; in Lab 03, with the 24 tie-offs removed, precheck still passes and LVS lists the 24 pins.
- **What the pinout says.** Precheck checks only that a used analog pin has a description, never what it says, and doesn't read the digital pins' descriptions at all. A pinout that calls the input the output passes.
- **Whether the circuit works.** Precheck simulates nothing. For an analog design, simulation of the extracted layout, with its parasitics, is how you find out, and it happens in the workbench.
- **Whether the GDS is the one your sources build.** Precheck checks the file you committed. If you changed `wires.tcl` and forgot to run `make`, it checks the old layout, and passes it.

Between them, the `docs` job and precheck check that your files are complete and legal. Checking that they describe the circuit you meant is left to you and your reviewer, and Pull requests and review is about how you do that.

Check yourself

1. A layout passes all 15 checks. Name two things that could still be wrong with it.

   Answer: A1, in Answers at the end.

2. Your table has ❌ on “KLayout Checks” and “Pin check”, and the first error in the log is `ERROR: Pin ua[0] not found in …lef`. Where do you start?

   Answer: A2, in Answers at the end.

Answers

A1. Check yourself

Any two of: a short or an open that LVS would find; missing tie-offs; a pinout that describes the pins wrongly; a circuit that doesn’t do what it should; a GDS that doesn’t match the committed sources.

A2. Check yourself

With row 7, “KLayout Checks”, because it comes first in the table and its message names the cause. If it says `Top macro name mismatch`, the pin errors follow from it: the pin check looks for pins on a macro with the expected name and can’t find any.

12. Your tools and the runner's tools

This page answers: *which versions of Magic, KLayout and the PDK check your layout on GitHub, how do they differ from the workbench's, and why does the course install a second set?*

What a cloud runner is explains that a runner installs its tools fresh for every job, at whatever versions the workflow asks for. In an analog project, the tools that decide whether your layout passes are precheck's, and their versions aren't the workbench's.

Three sets of tools

Your GDS goes through three sets of layout tools on its way from your keyboard to a green run:

Where	Magic	KLayout	sky130A PDK	Used for
The workbench (`hpretl/iic-osic-tools:2026.08`)	8.3.681	0.30.11	`026824c7`	Whatever you run in it yourself, unless you choose otherwise
The `gds` job's `custom_gds` step	–	0.29.12	–	The OASIS copy and the PNG render; it never checks anything
The `precheck` job	8.3.568	0.30.8	`0536d02d`	All 15 checks

The precheck versions are fixed in two places. The PDK commit is written into the `precheck` action at the tag your workflows use, `ttsky26d`. Magic and KLayout come from a Nix file in Tiny Tapeout's support tools, which pins each to a commit of the Nix package collection, and a precheck log shows what that gave: the job prints `Magic 8.3 revision 568`, and the KLayout it runs is from a folder called `klayout-0.30.8`. The support tools themselves aren't pinned at all: the action checks out their `main` branch on every run, so precheck's own code can change between two runs of the same commit. In November 2026, `main` was at `9d13ff3`, and precheck's code hadn't changed since Oct 29.

Why the versions matter

Tiny Tapeout's analog specs say "Make sure the version of Magic matches the PDK", and the reason is that the design rules live in the PDK's technology files and Magic reads them. A different Magic, or the same Magic with a different PDK, is in principle a different check.

In practice the differences are smaller than that sounds. When the course's tools were set up, we ran precheck on 11 commits from 6 public analog projects whose runs on GitHub were still readable: six commits that had failed precheck and five that had passed. We ran each one twice, once with the runner's versions and once with the workbench's. With the runner's versions, the table matched the runner's every time, row for row and message for message; for two of the failing runs, the Magic report was the same file as the one the runner uploaded, byte for byte. The workbench's versions gave the same pass or fail on all 11, though some counts along the way differed: on one layout, KLayout 0.30.11 counted 7,089 edge pairs at one rule of the deck where 0.30.8 and the runner counted 7,053, and 1,303 where they counted 1,258, and both still ended at the same 1,740 violations. Only one of the 11 layouts had a Magic DRC error, which is too few to say that Magic 8.3.681 always agrees with 8.3.568. The technology files of the two PDK commits differ by one line in their `wiring` section and in the version string; the DRC section is identical.

So the course treats precheck with the runner's versions as the check, and the workbench's tools as a second opinion.

What the course installs

In Lab 01, Part 1, `make install` puts precheck's versions next to the workbench's, in `labs/common/tt-tools/.local` of the course folder, without changing the image: Python 3.11 with precheck's packages at the runner's versions, the PDK at `0536d02d` through the same `ciel` tool the runner uses, KLayout 0.30.8, Magic 8.3.568 built from its source, and the support tools at `9d13ff3`. It takes about a minute and a half and 1.8 GB. `make check-tools` prints each version next to the runner's and the workbench's.

A few things are different from the runner, none of which the comparison above found to matter:

- Python 3.11.16 here, 3.11.18 on the runner, because the workbench's installer offers nothing newer in 3.11. The packages are the same versions, built for 3.11.
- Magic is compiled here without its 3D display, because the image has no headers for it.
- KLayout is klayout.org's own package of 0.30.8, not Nix's.
- The pip `klayout` module that precheck imports is pinned at 0.30.13. The runner takes whatever 0.30.x is newest on the day, which was 0.30.12 on Nov 2 and 0.30.13 on Nov 15.
- The support tools are pinned at `9d13ff3`, where the runner follows `main`.

Which tools each of your `make` targets uses:

Target	Magic	KLayout	PDK
`make` (build the layout)	8.3.568	–	`0536d02d`
`make precheck`	8.3.568	0.30.8	`0536d02d`
`make lvs`	8.3.568 to extract, the workbench's netgen to compare	–	`0536d02d`
`make drc`	the workbench's 8.3.681	–	the workbench's `026824c7`
`make view`	the workbench's 8.3.681, with a display	–	the workbench's

`make drc` is there so you can compare the workbench's Magic and PDK with precheck's. On the layouts we tried it on while writing the course, it agreed with precheck's Magic DRC row about whether there were errors: 0 on Lab 01's frame, on Lab 03's finished inverter and on the empty sandbox, and 8 on a Lab 03 wire with its metal 2 moved off its via, where precheck's row was ❌.

The course builds your layout with precheck's Magic, rather than the workbench's, for two reasons. It's the version precheck checks with, which is what the specs' "matches the PDK" asks for. And the workbench's Magic 8.3.681 sometimes crashes when it runs a script in batch mode. When we set up the course's tools, we saw it crash on exit after it had written its files, and while this course was being written it crashed once halfway through writing a LEF and left the file 222 lines short. So every `make` target judges a Magic run by the files it wrote and never by its exit status, and you'll need the same habit in Lab 05.

When Tiny Tapeout moves the tag

All of this was read from the tag `ttsky26d`, and it can change: each new shuttle brings a new template with a new tag, and Tiny Tapeout sometimes moves an existing tag to a new commit, as Reading a workflow file describes. Your repository keeps the tag it was made with, so its runs keep the same PDK commit, though the support tools still follow `main`. The course's pins live in one file, `labs/common/tt-tools/pins.env`, with the steps for reading them again at its top, and whoever maintains the course reads them again for each new tag.

Check yourself

1. Your layout passes `make drc` with 0 errors and fails precheck's Magic DRC row on GitHub. Which do you believe, and what would you run next?

   Answer: A1, in Answers at the end.

2. Two runs of the same commit, a month apart, give different precheck results. Your files didn't change. Name two things that could have.

Answer: A2, in Answers at the end.

Answers

A1. Check yourself

Precheck's: it's the check that counts, with its own Magic and PDK. Run `make precheck`, which uses the same versions, and read `magic_drc.txt` in `tt_submission/precheck_reports/` for the rule and the place. If `make precheck` passes too, look at what differs between your machine and the run: most often, the GDS you committed isn't the one you checked.

A2. Check yourself

The support tools' `main` branch, which the action checks out on every run, and the tag itself, if Tiny Tapeout moved it. The log records both: the commit the tag pointed to, in the line that downloads the action, and the support tools' commit in the checkout step.

13. Lab 03: A layout that passes precheck

You'll replace Lab 01's two stubs with a real circuit: the CMOS inverter from AD104 Lab 03, placed in your 1x2 tile, with its input on `ua[0]` and its output on `ua[1]`. The course gives you the cell and most of its wiring. You move the power stripes and draw the output wire yourself, then check the result three ways: with Magic DRC, with precheck, and with LVS, which precheck doesn't do. The change goes into `main` through a pull request in your own repository.

Package: `labs/lab-03-a-layout-that-passes-precheck/`

Before you start

- Read The analog layout kit, What precheck checks and Your tools and the runner's tools.
- Finish Lab 02, with both workflows green on `main`.
- You don't need to have taken AD104. The cell is DRC and LVS clean as it comes, and you won't change it.

This lab takes about 1 h 45 min.

Part 1: A branch and a cell

Start the change on a branch of its own, so that it reaches `main` only through a pull request.

1. Make the branch and add the lab's files:

   ```
   cd /foss/designs/projects/ttsky-ic102-a
   git switch -c lab03-inverter
   /foss/designs/modules/ic102-a/labs/lab-03-a-layout-that-passes-precheck/start.sh .
   ```

   You should see:

   ```
   Added the inverter and its wiring to /foss/designs/projects/ttsky-ic102-a/layout:
    M layout/wires.tcl
   ?? layout/cells/
   ?? layout/lvs/
   ```

2. Read the new `layout/wires.tcl` from the top. It places the cell, `layout/cells/ic102_inv.mag`, at (140, 10) µm; runs each of its supply rails sideways on metal 1 and up a stack of vias to a power stripe on metal 4; and carries the input `A` east on metal 2 to `ua[0]`, up to metal 4 and down to the pin. Near the end there's a TODO where the output wire belongs, and the last line, `tieoff 136`, ties the 24 digital outputs to ground.
3. Look at `layout/lvs/reference.spice`, the circuit the finished tile should contain. Find the one line that says what the inverter connects to, and the 24 lines that tie outputs to `VGND`.

Check: `git status` shows the three changes above, and you're on `lab03-inverter`.

Part 2: The power stripes

Where do the stripes have to be, and are they legal there?

1. `wires.tcl` expects a VGND stripe from x = 136 to 138 µm and a VDPWR stripe from 146 to 148 µm, on either side of the cell. The kit's script draws its stripes at 1 and 4 µm. Open `layout/init.tcl` and change its settings block to:

   ```
   set POWER_STRIPES {
       VDPWR 146um
   ```

```
    VGND  136um
}
```

2. Before you build, check the stripes against Tiny Tapeout's rules for analog power pins: "vertical stripes on met4" that "Start within the bottom 10 µm of the module and extend at least to the top 10 µm", with a "Minimum width of 1.2 um". Read `draw_power_stripe` in `init.tcl` and write down where each stripe starts and ends, how wide it is, and whether it passes. The tile is 225.76 µm tall.

   Answer: A1, in Answers at the end.

3. Run `make`, then `make view` to look at the result in Magic on the desktop. You should see the tile, its two new stripes in the right-hand third, and the cell between them, near the bottom edge.

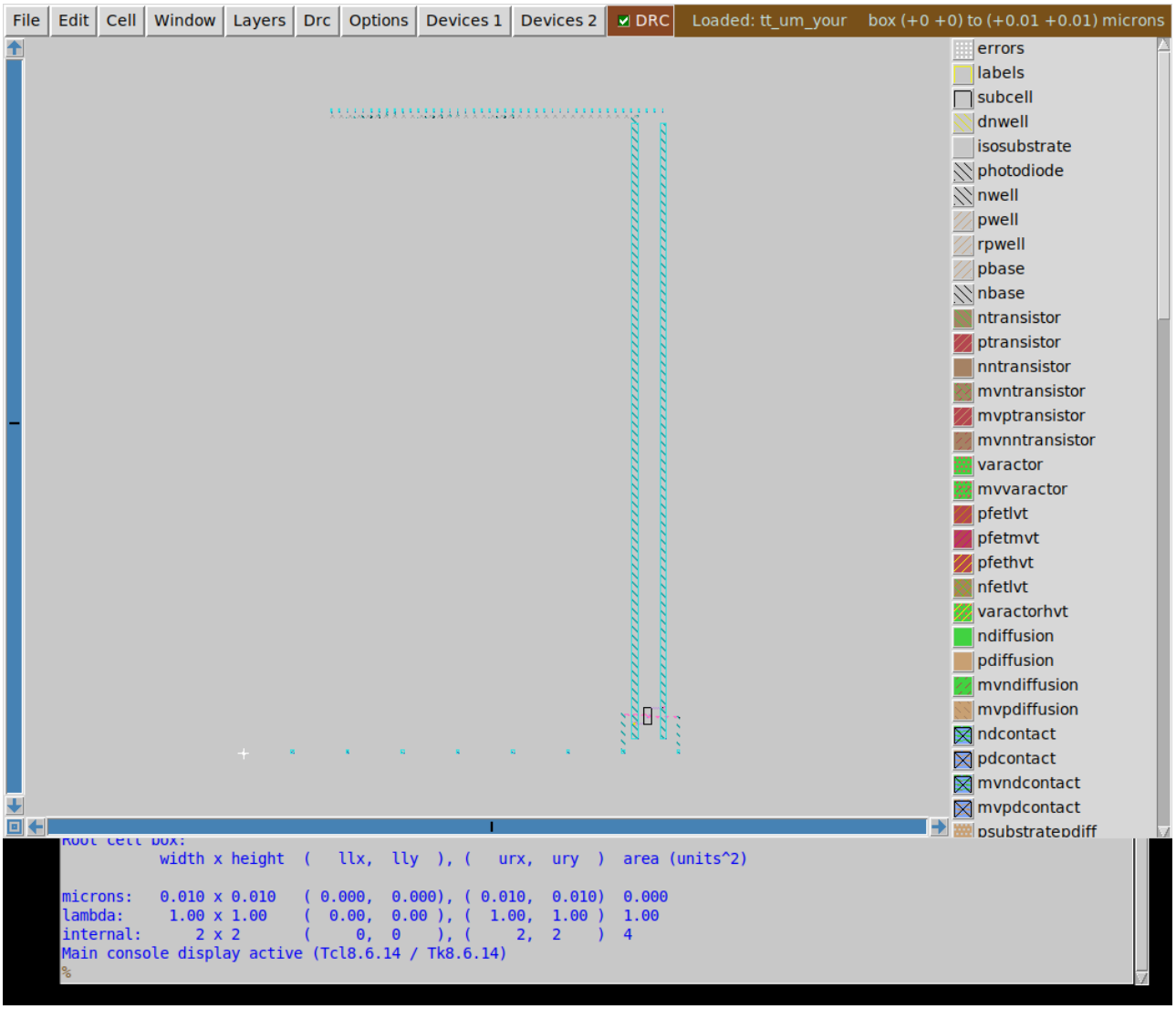


Figure 13.1: `make view`: the whole tile in Magic, with the two stripes and the cell between them near the bottom edge.

Check: `make` ends with `wrote gds/tt_um_yourname_ic102.gds and lef/tt_um_yourname_ic102.lef`.

Part 3: The output wire

Can you draw a wire that DRC, LVS and precheck all accept?

1. First see what the checks say about the layout as it is, with the TODO still open. Run `make lvs`. You should see:

```
magic: extract a netlist from gds/tt_um_yourname_ic102.gds
netgen: the extracted netlist against layout/lvs/reference.spice
  Final result: Top level cell failed pin matching.
  (no pin, node is inv0/Y)                |ua[1]
  ua[1]                                  |(no matching pin)
full report: layout/build/lvs.report
```

The left column is your layout and the right column is the reference. The reference has a pin `ua[1]` on the inverter's output, while in your layout the inverter's output, `inv0/Y`, isn't connected to any pin at all. Precheck sees the same gap in its own way: `make precheck` fails one check:

```
| Analog pin check | ❌ Fail: Analog pin `ua[1]` is not connected to any adjacent metal but `analog_pins` is set to 2 in `info.yaml`. Either wire up `ua[1]` to your design or decrease `analog_pins` to 1. |
```

2. Work out the wire on paper before you type anything, using `A`'s wire in `wires.tcl` as the pattern. You need four things: a point on `Y`'s locali strip for the contact (the strip runs from x = 142.18 to 142.55 µm and y = 11.45 to 14.17 µm once the cell is placed); a y for the metal 2 that keeps it at least 0.14 µm from `A`'s metal 2 at y = 12.3 to 12.75; `ua[1]`'s x, 132.49 to 133.39 µm; and a metal 4 wire from the via stack down to y = 0. Write the coordinates down.
3. Type the wire into the TODO in `layout/wires.tcl`, one `p` command per rectangle, in the same order as `A`'s: the locali pad, `mcon`, the metal 1 pad and `via1` at the contact; the metal 2 wire; the metal 2 and metal 3 pads, `via2` and `via3` at `ua[1]`; the metal 4 wire. Keep the via rectangles at least as big as `A`'s, because in Magic a via rectangle includes the metal around its cut.
4. Run `make` and `make drc`. You should see `Magic DRC on gds/tt_um_yourname_ic102.gds: 0 errors`. If you get errors, the report names the rule, such as `Metal2 overlap of Via1 < 0.03um in one direction (met2.5 - met2.4)`; fix the rectangle it's about and run both again.
5. Run `make lvs`. You should see:

   ```
   Final result: Circuits match uniquely.
   ```

6. Run `make precheck`. You should see `Precheck passed` and all 15 checks ✅.

   If you're stuck, `solutions/wires.tcl` in the lab's package has one answer. Yours can differ from it in every number and still be right, as long as it passes the three checks.

Check: 0 DRC errors, `Circuits match uniquely.`, and precheck passed, all on the same build.

Part 4: What precheck doesn't see

Which of the three checks catches a short between the input and the output?

1. Make a mistake on purpose. In your wire, change the bottom edge of `Y`'s long metal 2 rectangle so that it reaches down into `A`'s, which runs at y = 12.3 to 12.75. In the reference solution that's changing `p metal2 132.64 13.26 142.6 13.71` to `p metal2 132.64 12.6 142.6 13.71`. The two wires are now one piece of metal.
2. Before you run anything, write down which of `make drc`, `make lvs` and `make precheck` will complain.
3. Run `make`, then the three checks. In the reference solution with that change, you should see `Magic DRC on gds/tt_um_yourname_ic102.gds: 0 errors`, precheck passing all 15 checks, and LVS failing:

   ```
   Final result: Top level cell failed pin matching.
   (no matching net)                      |Net: ua[1]
   (no pin, node is ua[0])                |ua[1]
  Number of nets: 3 **Mismatch**          |Number of nets: 4 **Mismatch**
   ```

Why do DRC and precheck both pass?: A2, in Answers at the end.

4. Put the number back, and run `make`, `make lvs` and `make precheck` until you're back to `Circuits match uniquely.` and a pass.
5. Next, try a quicker mistake: comment out the `tieoff 136` line, run `make`, `make lvs` and `make precheck`, and compare. LVS lists the 24 output pins as unmatched, and precheck passes again. Tiny Tapeout's analog specs say "Connect any unused uo_out, uio_out and uio_oe pins to GND", but no automatic check holds you to it. Put the line back and rebuild.

Check: you're back to a layout that passes all three, and you can say which check caught each mistake.

Part 5: Through a pull request

How does a layout change get into `main` in a way someone else can check?

1. Update the pinout to match the new circuit: in `info.yaml`, describe `ua[0]` and `ua[1]` as the inverter's input and output, and update `docs/info.md`. Run `make docs`.
2. Commit everything on the branch. `git status` should show `info.yaml`, `docs/info.md`, `layout/init.tcl`, `layout/wires.tcl`, the two new folders under `layout/`, and the GDS and LEF. Then push the branch:

```
git add -A
git commit -m "Place the AD104 inverter between ua[0] and ua[1]"
git push -u origin lab03-inverter
```

3. On GitHub, open a pull request from `lab03-inverter` into `main`. Your workflows run on every push, so the branch already has a `docs` and a `gds` run, and the pull request shows their results. In the description, say what changed, and paste the `Final result` line of `make lvs` and the `Precheck passed` line of `make precheck`. (Paste that line rather than precheck's last line, which is only the path of its reports.) LVS doesn't run on GitHub, so the reviewer has no other way to know that you ran it.

Place the AD104 inverter between ua[0] and ua[1] #1

Open vproHacks wants to merge 1 commit into `main` from `lab03-inverter`

Conversation 0 Commits 1 Checks 4 Files changed 8

vproHacks commented 4 minutes ago Owner

Replaces the two metal stubs with the AD104 Lab 03 inverter: its input `A` wired to `ua[0]`, its output `Y` to `ua[1]`, and its supplies to new VDPWR and VGND stripes at x = 146 and 136 µm. The 24 digital outputs are tied to VGND. The pinout and `docs/info.md` describe the inverter.

Checked on this commit, in the course image:

- `make drc`: `Magic DRC on gds/tt_um_vprohacks_ic102.gds: 0 errors`
- `make lvs`: `Final result: Circuits match uniquely.`
- `make precheck`: `INFO: Precheck passed for /foss/designs/projects/ttsky-ic102-a/tt_submission/tt_um_vprohacks_ic102.gds!` 🎉

LVS doesn't run on GitHub, so those lines are the only evidence of it here.

This pull request is part of a live test of IC102-A's Lab 03. Claude Code (an AI tool) made the change, using the wire from th lab's reference solution, and wrote this description; the three lines above were copied from its runs.

🤖 Generated with Claude Code

https://claude.ai/code/session_014vNQo86iMp4KCodfkrP84R

All checks have passed
4 successful checks

- docs / docs (push) Successful in 1m
- gds / gds (push) Successful in 48s
- gds / precheck (push) Successful in 1m
- gds / viewer (push) Successful in 24s

No conflicts with base branch
Merging can be performed automatically.

Figure 13.2: A pull request in your own repository, with the LVS result in its description, and further down, the four checks the branch's push started: docs, and gds, precheck and viewer.

4. When the branch's `gds` run is green in all three jobs, merge the pull request. The merge is a push to `main`, so both workflows run again; wait for them.
5. Download the `gds_render` artifact from the merge's `gds` run and open the PNG. You should see your tile as Tiny Tapeout renders it. The inverter is 3 µm wide in a 161 µm tile, so it's a speck near the bottom right, with its short wires down to `ua[0]` and `ua[1]`. At this scale the two stripes stand out, and so does the strap along the top edge that ties the 24 outputs to VGND.

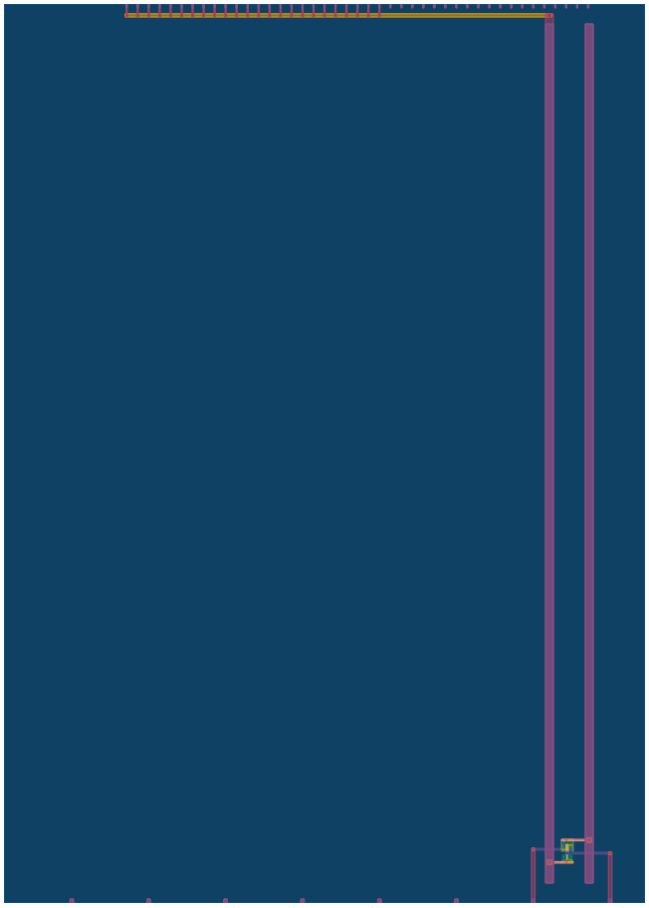


Figure 13.3: The tile with the inverter from the reference solution, rendered locally with the same gdstk call the gds_render step uses.

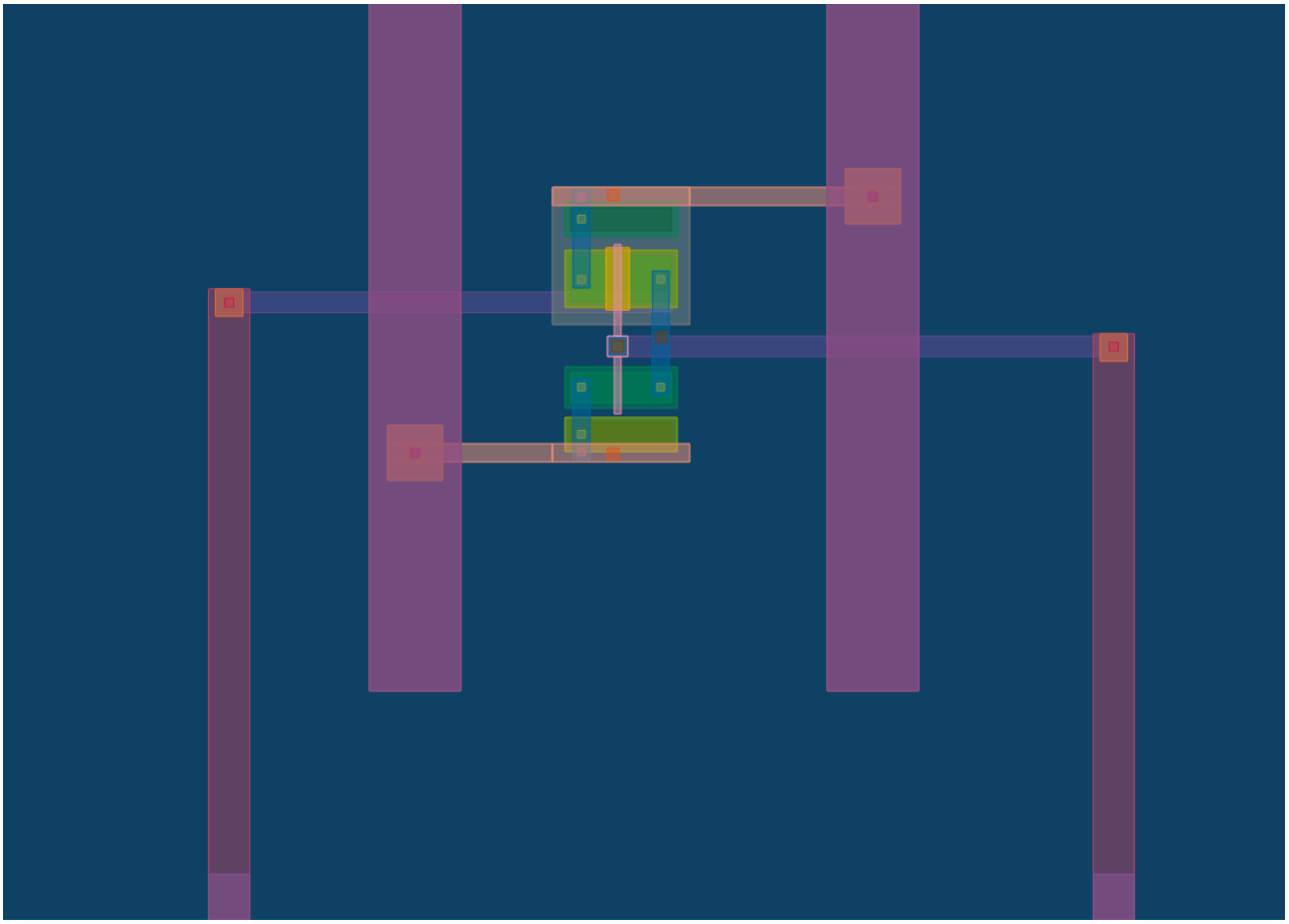


Figure 13.4: The same render, close up: the cell between its two stripes, the two supply stacks, and the input and output wires running out to ua[0] (right) and ua[1] (left).

Check: the pull request is merged, and the `gds` run on the merge commit is green.

Finishing

You're done when the merge commit on your `main` has a green `gds` run, with `gds`, `precheck` and `viewer` all passed, and `make lvs` on that commit prints `Final result: Circuits match uniquely.`

Questions:

1. Suppose you'd left the stripes where the kit puts them, VDPWR at 1 µm and VGND at 4 µm, and built the reference solution's `wires.tcl` as it is. Which checks would fail?

   Answer: A3, in Answers at the end.

2. Your pull request's diff of `gds/tt_um_yourname_ic102.gds` says only that a binary file changed. What in the pull request lets a reviewer judge the layout?

   Answer: A4, in Answers at the end.

Submit: nothing for this lab.

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

Answers

A1. Part 2: The power stripes

`draw_power_stripe` draws a box from y = 5 µm to 220.76 µm, `POWER_STRIPE_WIDTH` (2 µm) wide, starting at the x you give it. 5 µm is within the bottom 10 µm, 220.76 µm is above 225.76 − 10 = 215.76 µm, and 2

µm is more than 1.2 µm, so both stripes pass. Precheck checks the stripes too, in its pin check, against the template.

A2. Part 4: What precheck doesn't see (Why do DRC and precheck both pass?)

The metal itself is legal: it's wide, well clear of everything else, and joined properly. It's just connected to the wrong thing, and neither DRC nor any of precheck's 15 checks compares the layout with what it's meant to be. LVS does: the extracted circuit has the inverter's `A` and `Y` on one net, called `ua[0]`, where the reference has them on two. So run LVS yourself, every time, before you push a layout.

A3. Finishing

We tried it: precheck passes all 15 checks, because the stripes are just as legal at 1 and 4 µm, and LVS fails. The cell's two via stacks now land on empty space at x = 136 and 146 µm, so the inverter's supplies connect to nothing, and neither do the 24 tied outputs, whose strap ends at the same empty spot. `make lvs` starts its list with `(no pin, node is inv0/VPWR)  |VDPWR` and goes on to the 24 outputs.

A4. Finishing

The text files that built it (`wires.tcl`, `init.tcl`'s settings, the reference netlist), the runs' results, precheck's table, the `gds_render` picture, and the LVS line you pasted. In Lab 04 you review a layout change from those.

IV 4. Review

14 Pull requests and review 95
14 Issues, branches and pull requests 95
14 What review is for 95
14 Writing a pull request someone can review . . . 96
14 What a reviewer checks that CI can't 97
14 Pull requests written with AI tools 98
14 Check yourself 98
14 Answers 98

15 Branch rules 101
15 What a ruleset is 101
15 Three rulesets in the team's repositories 101
15 What write access allows 103
15 Rules that do what they say 103
15 Check yourself 104
15 Answers 104

16 Layout files in git 105
16 Sources and outputs 105
16 What a diff of a GDS shows 105
16 Conflicts in binary files 106
16 Check yourself 106
16 Answers 107

17 How open-source projects work . . . 109
17 Licences 109
17 Copying someone else's code 109
17 Working with an upstream project 110
17 Working in public 110
17 Check yourself 110
17 Answers 111

18 Lab 04: A change through review . . 113
18 Before you start 113
18 Part 1: Your review repository 113
18 Part 2: Claim a slot in your partner's repository . 116
18 Part 3: Main moves 118
18 Part 4: Review your partner's pull request . . . 121
18 Part 5: Answer the review and resolve the conflict 123
18 Part 6: Approve and merge 124
18 Part 7: A pull request written by an AI tool (optional) 125
18 Finishing 125
18 Answers 126

14. 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:

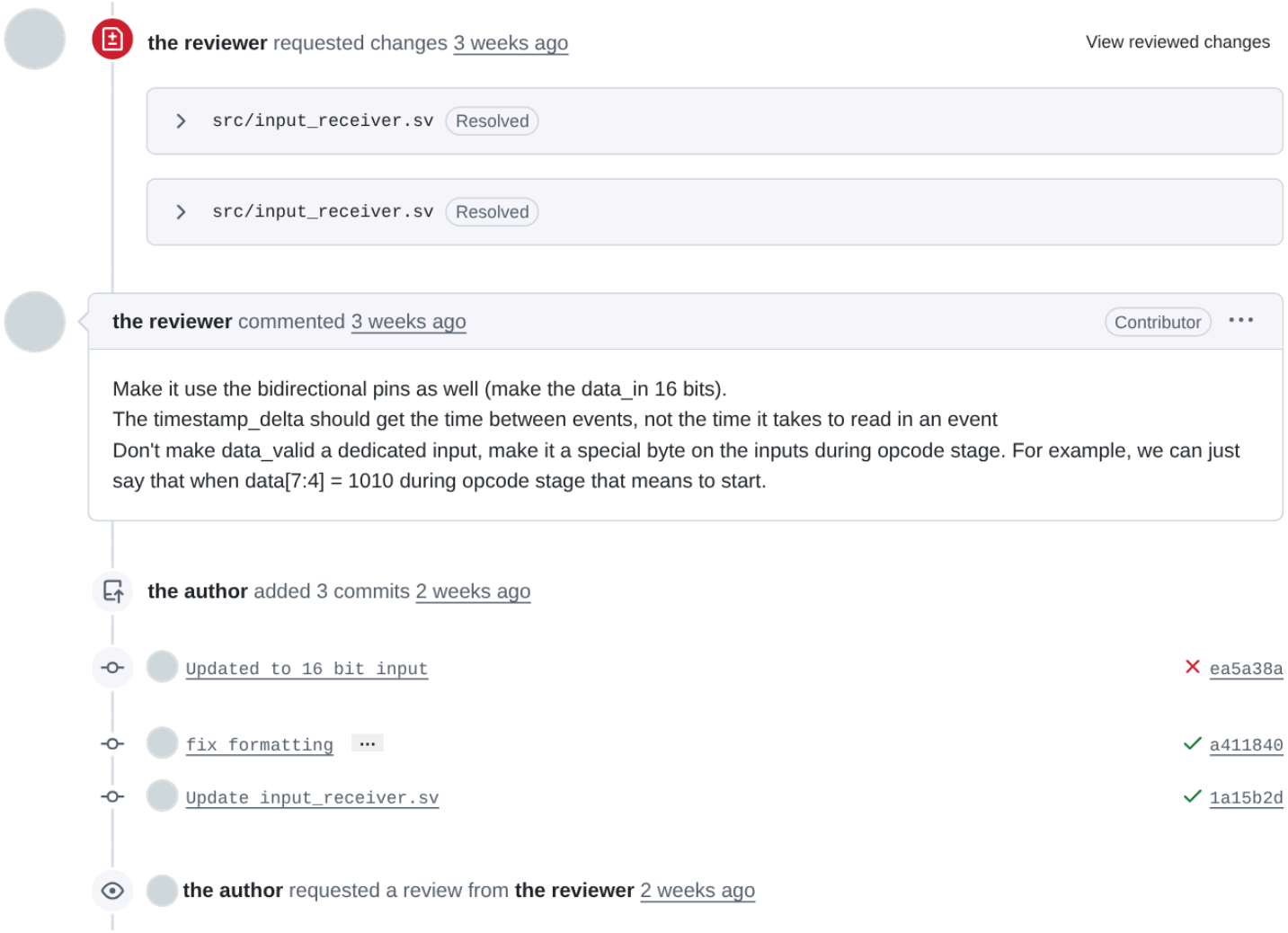


Figure 14.1: One round of review on sentry's pull request #33. Names are replaced with roles, and avatars with grey discs.

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.

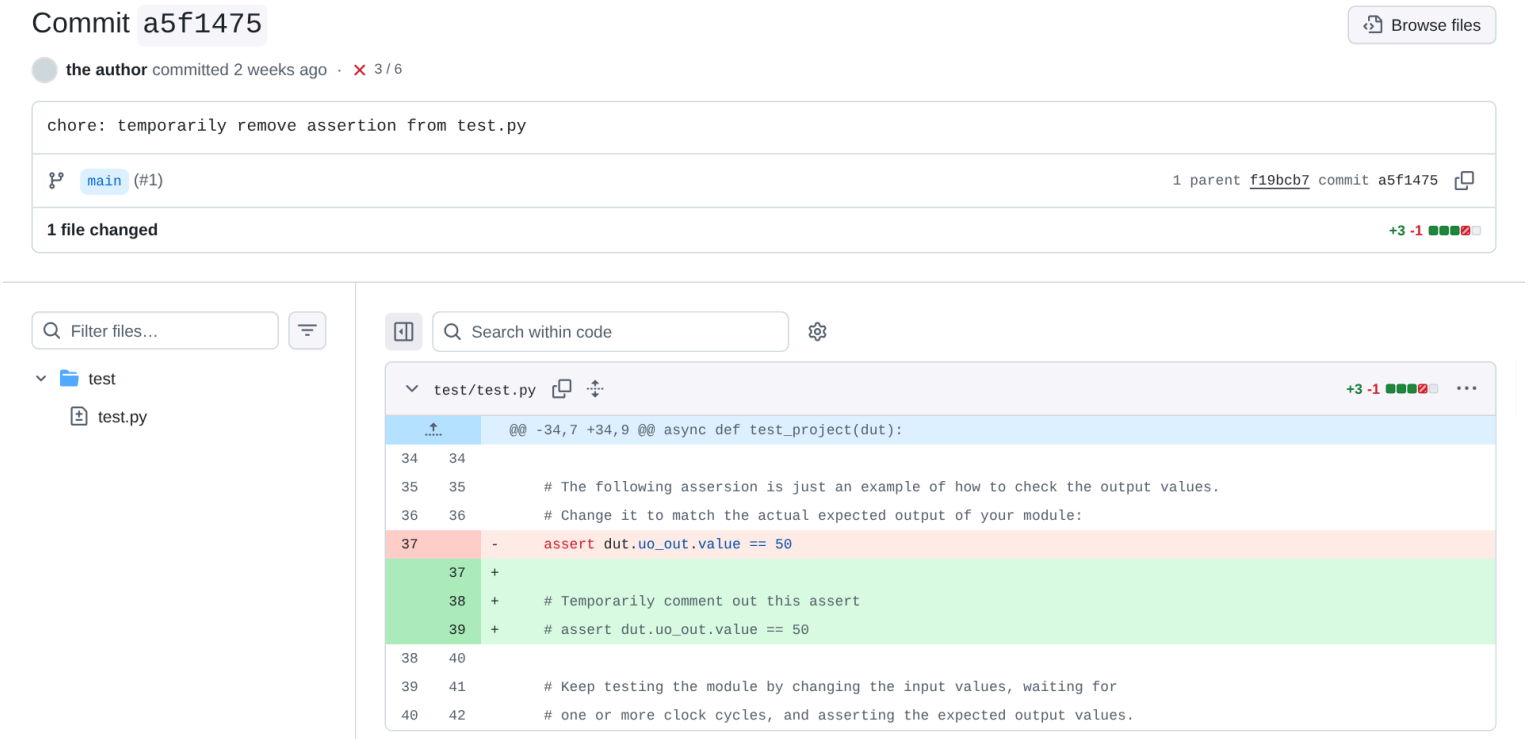


Figure 14.2: The commit that commented out protocol-emulator's only assertion. The author's name is replaced with a role.

`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.

15. 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":

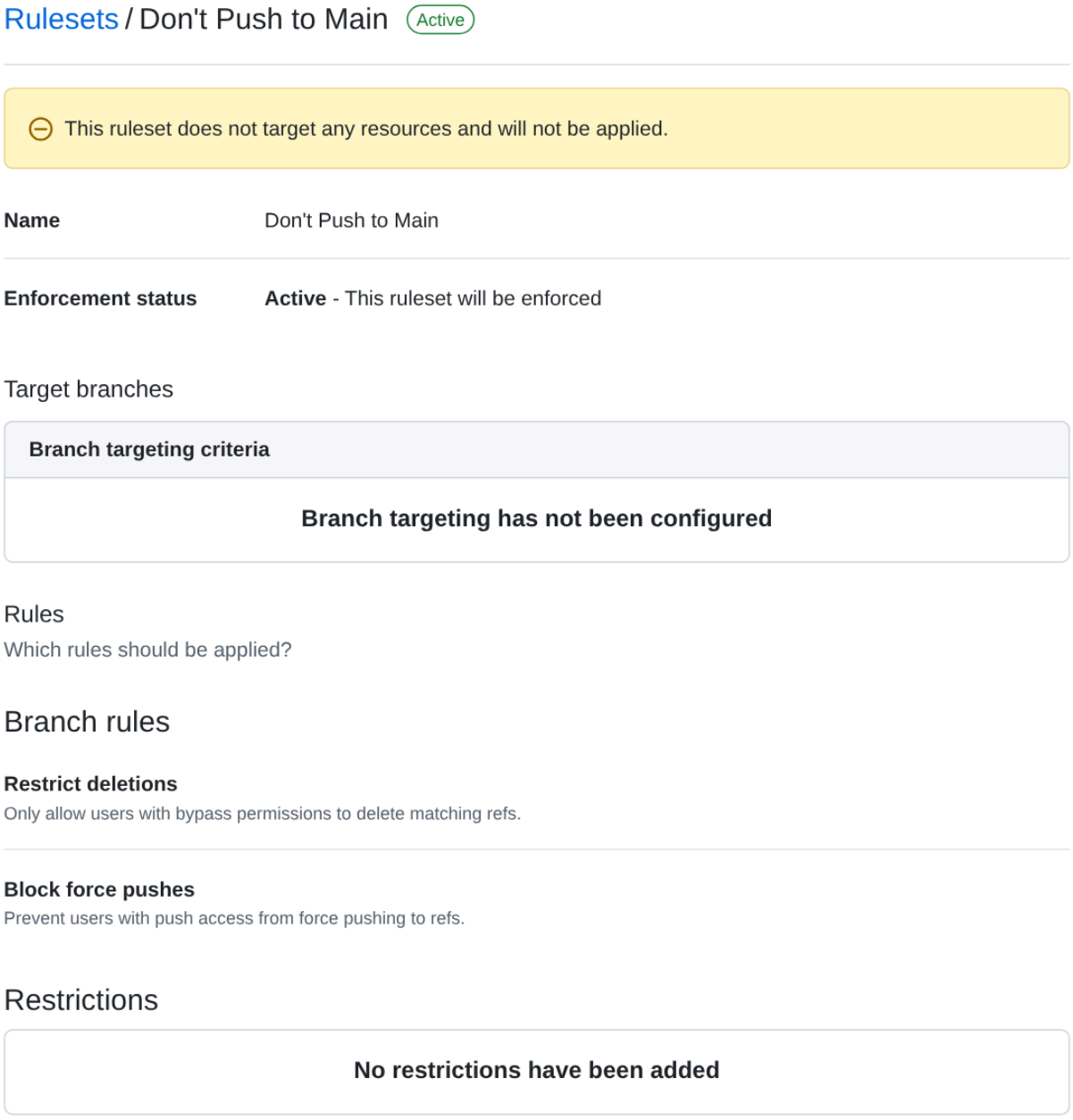


Figure 15.1: hart-1's ruleset "Don't Push to Main", seen without signing in.

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.

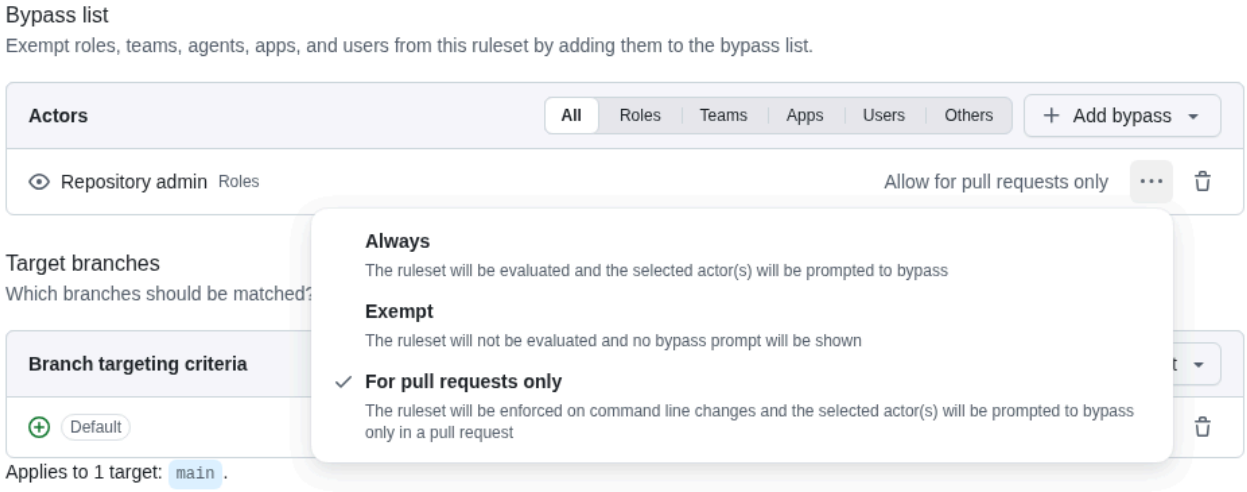


Figure 15.2: A ruleset's bypass list with its entry's menu open, offering "Always", "Exempt" and "For pull requests only". This entry is set to "For pull requests only", which the list shows as "Allow for pull requests only".

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.

16. Layout files in git

This page answers: *which of an analog project's files can git compare and merge, which can't, and what does that change about committing, reviewing and resolving conflicts?*

Git in one page describes git working on text, line by line. An analog repository isn't all text: its main product, the GDS, is a binary file, and the files around it are a mix of sources you edit and outputs a tool writes. Which file is which tells you what goes into a commit, what a reviewer can read, and what to do when two people's changes collide. You'll resolve a conflict like that in Lab 04, and another in Lab 05.

Sources and outputs

File	Text or binary	Made by	Commit it?
`info.yaml`, `docs/info.md`, `src/project.v`	text	you	yes
`layout/init.tcl`, `layout/wires.tcl`	text	you	yes: they're the layout's source
`layout/tt_analog_1x2.def`	text	Tiny Tapeout	yes, with `UPSTREAM.md` saying where it's from
`layout/cells/*.mag`	text, in Magic's own format	Magic, from someone's drawing	yes: a hand-drawn cell has no other source
`gds/<top_module>.gds`	binary	`make` (Magic)	yes, because the `gds` workflow reads it from the repository
`lef/<top_module>.lef`	text	`make` (Magic)	yes, for the same reason
`layout/build/`, `tt_submission/`, Magic's `.ext` files, core dumps	mixed	`make`, `make precheck`	no: `.gitignore` keeps them out

The usual rule in software is to commit sources and leave out anything a build makes. The analog template breaks it on purpose: its `gds` workflow doesn't build anything, so the GDS and LEF have to be in the repository for it to check. The rule that's left is to commit them together with the change to the sources that made them, in the same commit, so that every commit's GDS is the layout of that commit's sources. Lab 05's history has a commit called "Keep build output out of git" that removed `gds/` from the repository, and the next one had to put it back, because the workflow reads it from there.

`.mag` files sit in between. Magic writes them, but for a cell drawn by hand in Magic's editor they are the only source there is, and they're text, so git can compare them. A reviewer still can't do much with a diff of one: `layout/cells/inv_bob.mag` is 116 lines of rectangles.

What a diff of a GDS shows

git treats a file as binary when it contains bytes that aren't text, and then shows only that it changed. This is from a rehearsal of Lab 04, a student claiming slot 4 of the sandbox:

```
gds/tt_um_ic102a_sandbox.gds | Bin 19716 -> 24770 bytes
info.yaml                    |   4 +-
layout/cells/inv_bob.mag     | 116 ++++++++++++++++++++++++++++++++++++++++++++
```

```
 layout/slots.tcl             |   1 +
 lef/tt_um_ic102a_sandbox.lef |  57 ++++++++++++-----------
```

Five lines of that are readable, in `slots.tcl` and `info.yaml`. The GDS shows up only as a change in size. The LEF is text, but Magic generates it, and its 57 changed lines are Magic's bookkeeping. So a reviewer has to judge a layout change from other evidence:

- the source files that built it, which are text;
- the `gds_render` artifact, a picture of the layout;
- precheck's table and its `precheck_reports` artifact;
- the author's LVS result, which no workflow produces, so it has to be in the description.

Pull requests and review makes the same point for reviewers in general, and the sandbox's `REVIEWING.md` turns it into a checklist.

Two more things follow from the GDS being binary. First, every commit that changes it adds another whole version of it to the history, and every clone downloads all of them. Ours are about 20 KB, which is nothing, but a large layout is worth committing only when it has really changed. Git LFS, which keeps large files outside the history, isn't an option for a Tiny Tapeout project, because GitHub's documentation says a template "cannot include files stored using Git LFS", and a repository made from one starts without it. Second, Magic writes the time into every GDS it saves. Building the same sources twice gives two files that differ in 10 bytes, the dates in the file's headers, so `git status` shows your GDS as changed after any `make`, even when the layout didn't change. If you didn't mean to change the layout, `git checkout gds/ lef/` puts the committed files back.

Conflicts in binary files

When two branches change the same lines of a text file, a merge stops and shows you both versions between `<<<<<<<` and `>>>>>>>` markers. When they both change a binary file, there are no lines to show, and git says so:

```
warning: Cannot merge binary files: gds/tt_um_ic102a_sandbox.gds (HEAD vs. upstream/
main)
Auto-merging gds/tt_um_ic102a_sandbox.gds
CONFLICT (content): Merge conflict in gds/tt_um_ic102a_sandbox.gds
```

git can offer you one side's file or the other's (`git checkout --ours` or `--theirs`), and both are wrong after a merge, because neither is the layout of the merged sources. In Lab 04′s rehearsal, one side's GDS had `bob`'s inverter and not `alice`'s, and the other had `alice`'s and not `bob`'s; either would have passed precheck, so nothing on GitHub would have caught the loss. The right answer is to merge the sources, which are text, and build the GDS again from the result. The LEF needs the same treatment even though it's text, because git merges it line by line into a file that describes no real layout: in the rehearsal it came out with three sets of conflict markers.

A revert is a merge too, run backwards: undoing a commit means applying its change in reverse to today's files. In Lab 05, reverting one commit stops on the GDS, which three later commits changed, while the one-line change to `wires.tcl` reverts cleanly. The fix is the same: run `make`, add the new GDS and LEF, and continue.

So, in a layout repository:

1. Resolve conflicts in the text sources by hand.
2. Don't resolve conflicts in `gds/` or `lef/`; rebuild them with `make`.
3. Check the result with `make lvs` and `make precheck` before you commit it, because a merge can produce a layout that neither side tested.

Check yourself

1. You change one coordinate in `wires.tcl`, run `make`, and commit only `wires.tcl`. What does the next `gds` run check?

   Answer: A1, in Answers at the end.

2. After a merge, `make` fails and `info.yaml` won't parse. What's the likely cause?

   Answer: A2, in Answers at the end.

Answers

A1. Check yourself

The GDS from the previous commit, because that's the one in the repository. The run tests a layout that doesn't match the sources in the same commit. Commit `gds/` and `lef/` with the change.

A2. Check yourself

Conflict markers left in `info.yaml`. The course's `make` says so when it can't read `top_module`, and lists the marker lines.

17. 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.

18. Lab 04: A change through review

You and a partner each make a review repository from the course's sandbox, a small analog project with eight empty slots, each for one inverter. You'll claim a slot in your partner's repository through a pull request from a fork, and review the one your partner opens in yours. Because you also claim the slot next to theirs in your own repository, their pull request will conflict with `main`, and they'll resolve the conflict the way a layout repository needs it resolved, by building the GDS and LEF again from the merged sources.

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

Before you start

- Read Pull requests and review, Branch rules, Layout files in git and How open-source projects work.
- Finish Lab 03. You'll copy your inverter cell from your `ttsky-ic102-a` repository.
- Find a partner who is taking IC102-A, and swap GitHub usernames. You both do every part: Parts 1, 3, 4 and 6 in your own repository, as its owner, and Parts 2 and 5 in your partner's, as a contributor.
- Your token from Lab 01, Part 2 already covers the two repositories you'll make here, because it was made for "All repositories".

This lab takes about 1 h 30 min of your own time, spread over a day or two, because each of you waits for the other: your partner's pull request has to be open before Part 3, and yours can't be merged until your partner approves it.

Part 1: Your review repository

Set up the repository your partner will send a pull request to, with rules on its `main`.

1. On GitHub, make a new repository (the **+** menu › **New repository**) with your own account as the owner, the name `ttsky-ic102-a-review`, **Public**, and no README, `.gitignore` or licence, so that it starts empty.

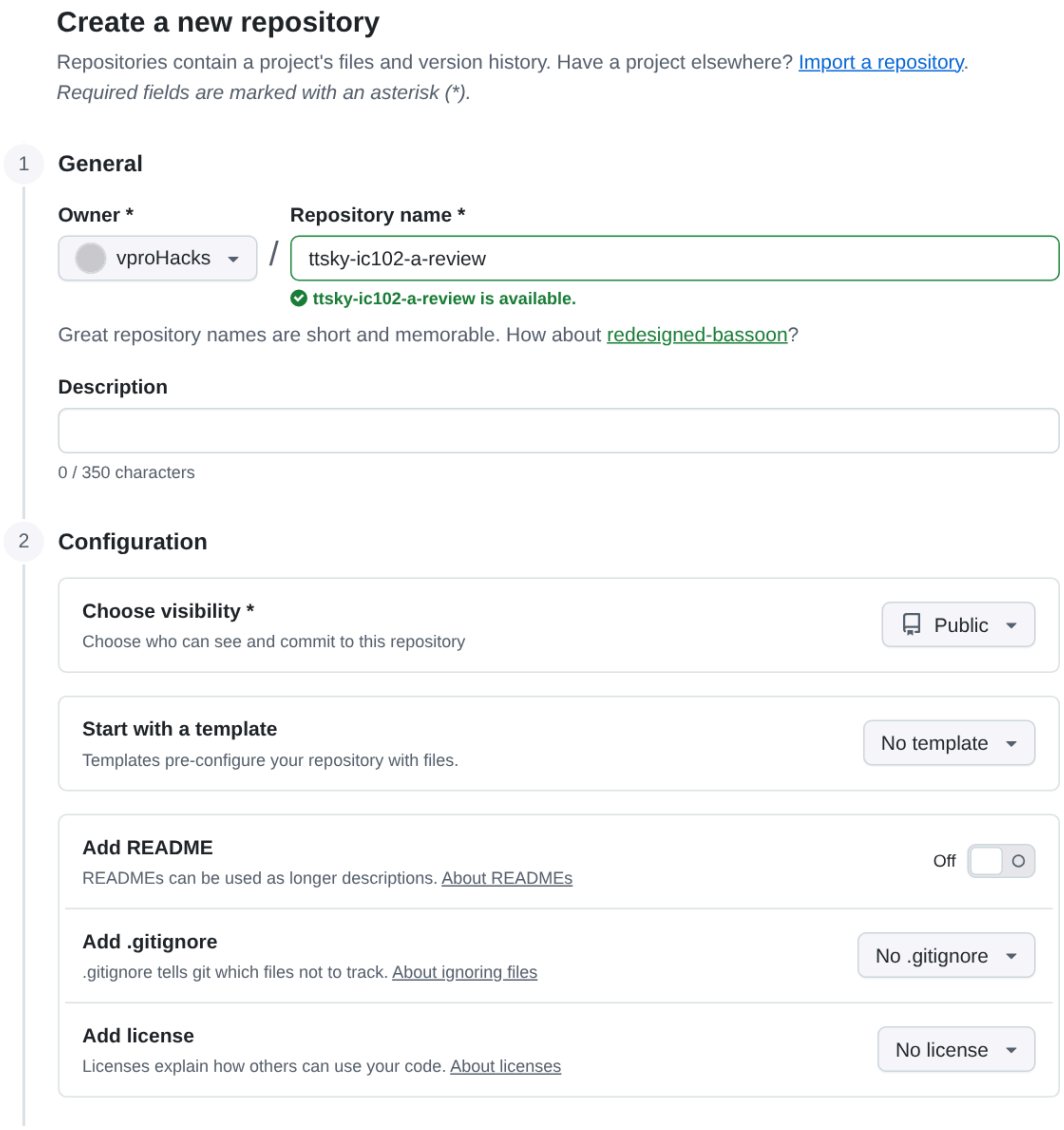


Figure 18.1: The new-repository form filled in for the review repository: your own account as the owner, Public, no template, Add README off, and no .gitignore or licence.

2. Copy the sandbox into your projects folder, make it a git repository, and push it:

```
mkdir -p /foss/designs/projects && cd /foss/designs/projects
cp   -r   /foss/designs/modules/ic102-a/labs/lab-04-a-change-through-review/sandbox
ttsky-ic102-a-review
cd ttsky-ic102-a-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-A sandbox"
git remote add origin https://github.com/YOUR-USERNAME/ttsky-ic102-a-review.git
git push -u origin main
```

You should see `[main (root-commit) ...] Start from the IC102-A sandbox` and `25 files changed`. This push adds files under `.github/workflows/`, which is why the token needs the Workflows permission as well as Contents.

3. The push starts the `docs` and `gds` workflows, and `viewer` fails with the 404 from Lab 01, because Pages is off in a new repository. Turn it on (Settings › Pages › Source: **GitHub Actions**), then start a new run: Actions › `gds` › **Run workflow** on `main`. As in Lab 01, don't use "Re-run failed jobs" for this. You should see `docs` green, and `gds`, `precheck` and `viewer` green in the new run. In our test of this lab on Nov 16, 2026, the push's `docs` run passed in 42 s; its `gds` run had `gds` and `precheck` green and `viewer` red after 8 s, with `Failed to create deployment (status: 404)`; and the Run workflow run after Pages was on passed all three jobs in 2 min 42 s.
4. Add the rules. Open Settings › Rules › Rulesets › **New ruleset** › **Import a ruleset**, and choose `ruleset.json` from your clone; on your own computer it's in your workspace folder, under `projects/ttsky-ic102-a-review/`. Check what the page shows against the table in `RULESET.md`, then press **Create**.

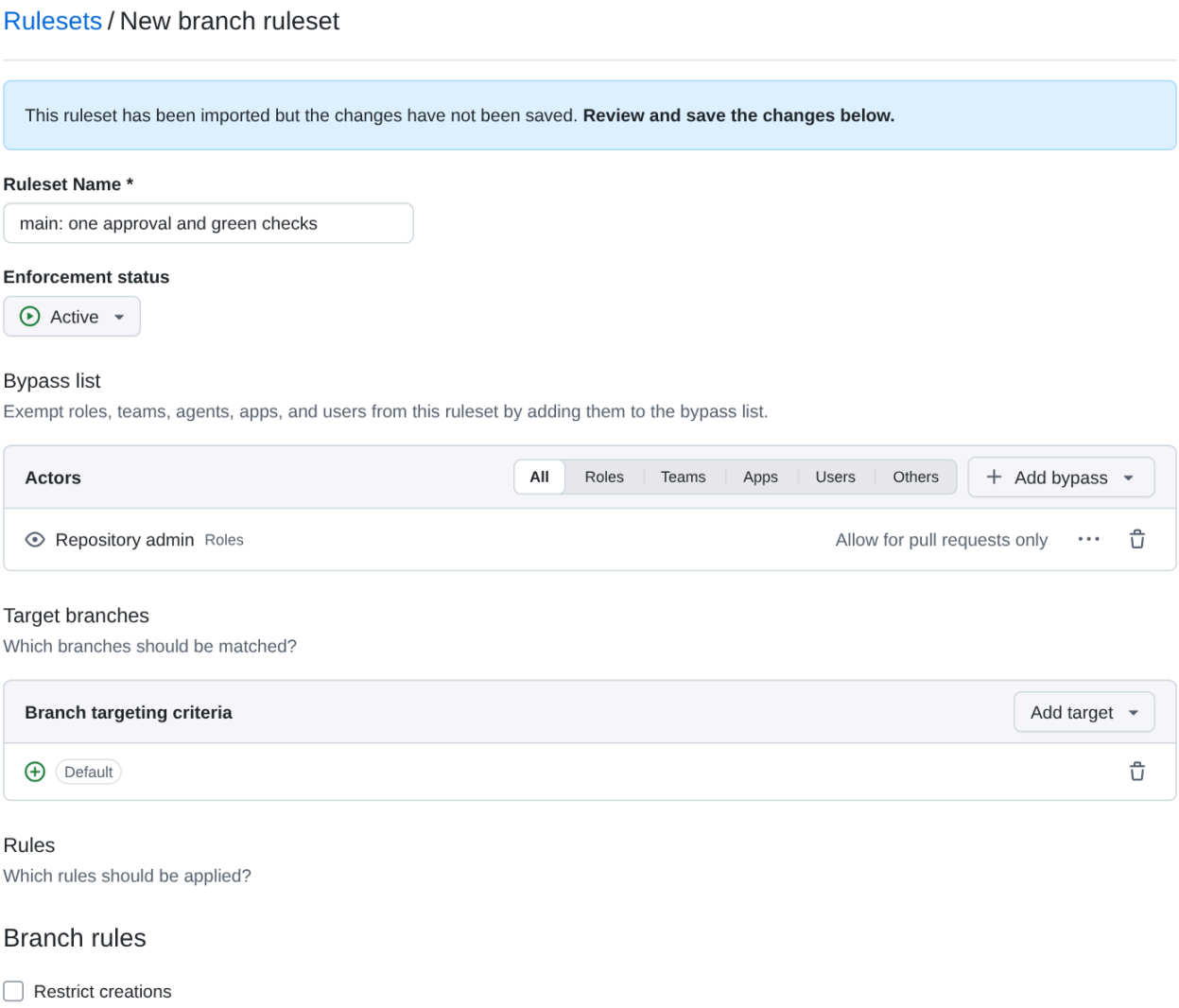


Figure 18.2: The imported ruleset before Create, with the repository admin on the bypass list, "Allow for pull requests only".

GitHub takes the file as written: the bypass entry shows as "Repository admin" with "Allow for pull requests only", and the rules below it match `RULESET.md`. One box is ticked that the file doesn't mention, "Require an additional approval for unattributed Copilot pull requests". That's GitHub's default, and it matters only for a pull request that Copilot opens by itself.

5. You're the repository's admin, and the admin role is on the bypass list. Before you try it, write down whether a commit pushed straight to `main` will be accepted. Then try one that changes nothing, and throw it away afterwards:

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

Answer: A1, in Answers at the end.

6. Send your partner the address of your repository.

Check: a green `gds` run on `main` with all three jobs, an active ruleset, and a refused push.

Part 2: Claim a slot in your partner's repository

What does a layout change look like when you can't read it in a diff?

1. Open `https://github.com/PARTNER-USERNAME/ttsky-ic102-a-review` and click **Fork**, in the top-right corner of the page. On the form that opens, keep your own account as the Owner and keep "Copy the main branch only" ticked; both were already set that way in our test. Without that box, every branch in your partner's repository is copied into your fork as well. Change the Repository name to `ttsky-ic102-a-review-fork`. A fork gets the name of the repository it was made from unless you change it, and you already have a repository called `ttsky-ic102-a-review`. GitHub says the new name "is available" once you've typed it. Then click **Create fork**. GitHub describes the form in forking a repository.

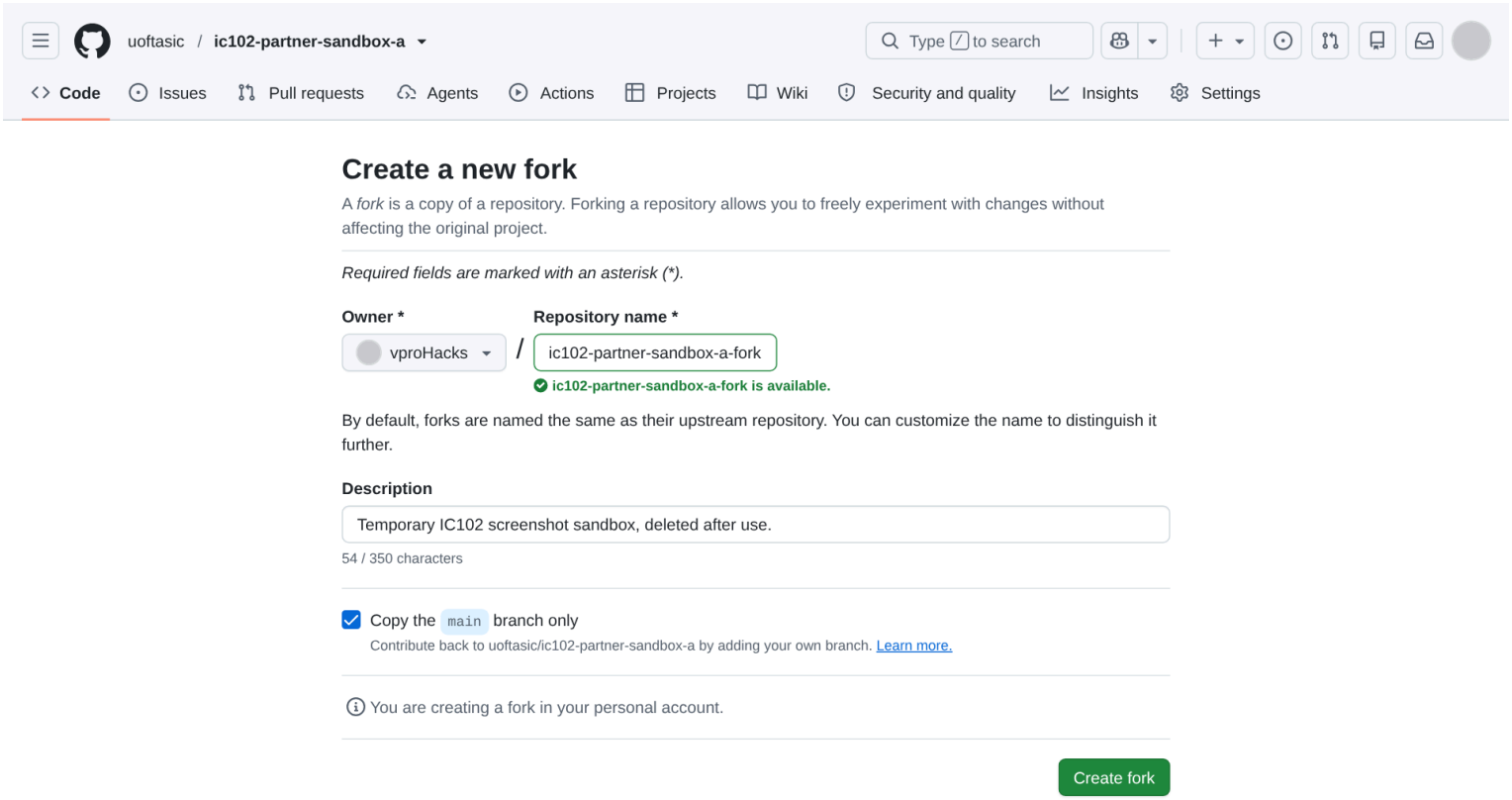


Figure 18.3: The fork form for your partner's repository. In our test the uoftasic organisation stood in for the partner, so the form forks uoftasic/ic102-partner-sandbox-a, and the new name is that repository's name with -fork on the end.

2. Clone your fork, set it up as you did your own, 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-a-review-fork.git
```

```
cd ttsky-ic102-a-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-a-review.
git
git remote -v
```

`git remote -v` should list `origin` (your fork) and `upstream` (your partner's repository), each twice.

3. Pick a slot, K, from 0 to 7, and tell your partner which one. Make a branch for it:

```
git switch -c slot-K-YOUR-USERNAME
```

4. Make the change, with `you` standing for your GitHub username with any `-` changed to _:
   - Copy your cell from Lab 03: `cp ../ttsky-ic102-a/layout/cells/ic102_inv.mag layout/cells/inv_you.mag`.
   - Add one line to the end of `layout/slots.tcl`: `slot K inv_you`.
   - In `info.yaml`'s pinout, describe `ui[K]` and `uo[K]`, for example `"you's inverter, input"` and `"you's inverter, output"`.
5. Run `make`, `make lvs` and `make precheck`. You should see `wrote gds/tt_um_ic102a_sandbox.gds and lef/tt_um_ic102a_sandbox.lef`, then `Final result: Circuits match uniquely.`, then `Precheck passed` with all 15 checks ✅. `make lvs` checks the layout against a reference that `layout/lvs/reference.py` writes from `slots.tcl`, so your slot is in it.
6. Commit, and look at what you're about to ask someone to review:

```
git add -A
git commit -m "Claim slot K for YOUR-USERNAME's inverter"
git show --stat HEAD
```

Here's the same change for a student called `bob` in slot 4, from a rehearsal of this lab:

```
 gds/tt_um_ic102a_sandbox.gds | Bin 19716 -> 24770 bytes
 info.yaml                    |   4 +-
 layout/cells/inv_bob.mag     | 116 ++++++++++++++++++++++++++++++++++++++++++++
 layout/slots.tcl             |   1 +
 lef/tt_um_ic102a_sandbox.lef |  57 ++++++++++++-----------
 5 files changed, 148 insertions(+), 30 deletions(-)
```

Which of these lines can a reviewer actually read?: A2, in Answers at the end.

7. Push the branch to your fork with `git push -u origin slot-K-YOUR-USERNAME`. git prints a link for opening a pull request; open it, and check that the base is your partner's repository and its `main`. Fill in the description from the template: which slot, the cell's name, the `Final result` line of `make lvs` and the `Precheck passed` line of `make precheck`. If an AI tool wrote part of the change or the description, say which part, and what you checked yourself. Create the pull request and tell your partner its number.

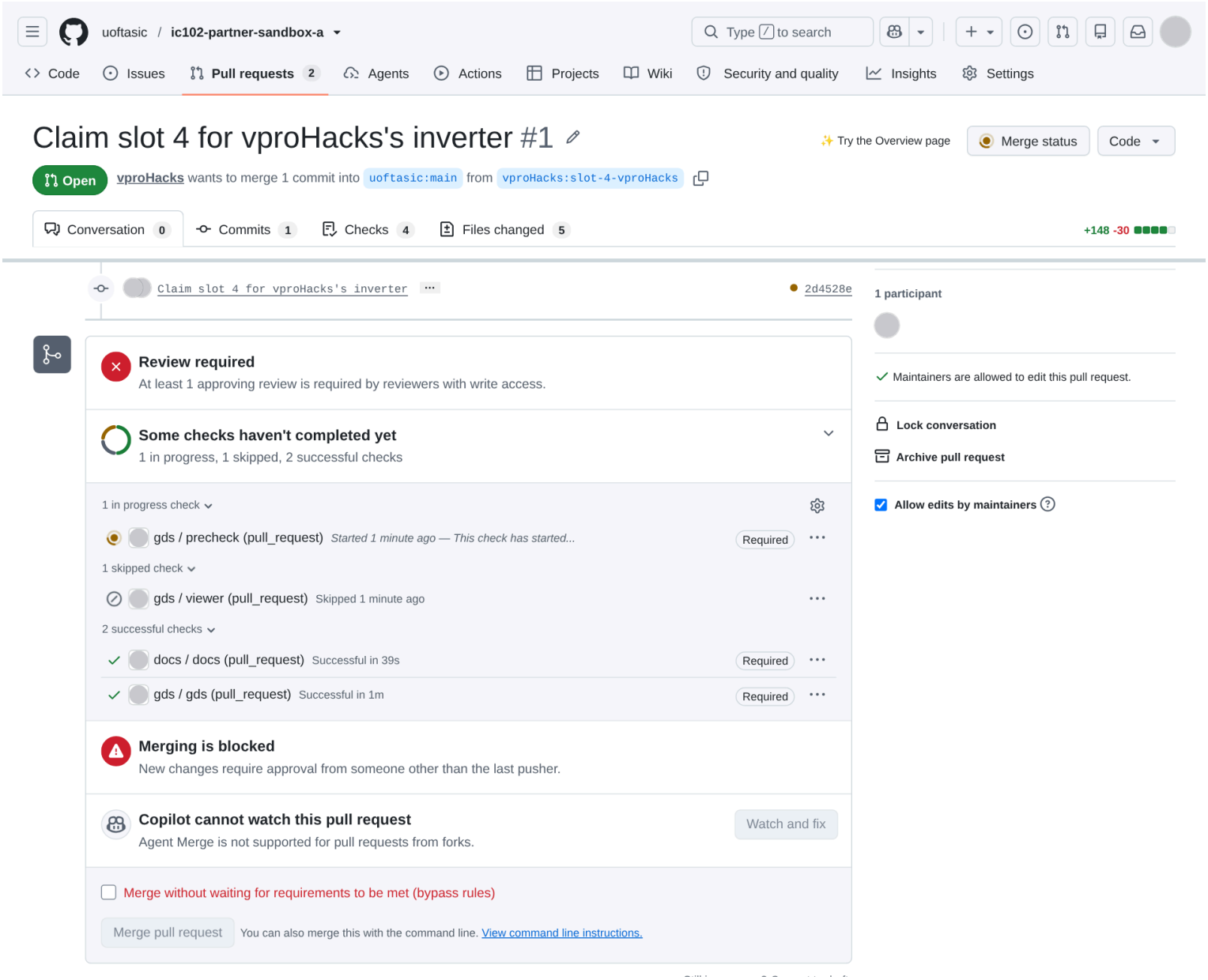


Figure 18.4: Your pull request on your partner's repository, in our test, where uoftasic/ic102-partner-sandbox-a stood in for the partner's. The line under the title names the base, uoftasic:main, and the branch in the fork. Docs and gds have passed, and precheck is still running. The bypass box near the bottom is there because the account that opened the pull request also owns the organisation; the ruleset gives the bypass only to the repository's admin, so you won't have it on your partner's repository.

Check: your pull request is open on your partner's repository, with one commit that changes those five files.

Part 3: Main moves

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

1. Open your partner's pull request in your repository. Its checks haven't started, because by default GitHub holds the runs of anyone who has never had a commit or pull request merged into a repository, until someone with write access approves them (repository Actions settings). A `pull_request` run uses the workflow files from the pull request itself, so read **Files changed** first and check that nothing under `.github/` changed. Then click **Awaiting approval**, in the upper right corner of the page, to open the **Merge status** panel, and click **Approve workflows to run** in it (approving workflow runs

from forks). Do this before you merge anything of your own in step 4, because once the pull request has a conflict it gets no runs at all.

Those two labels are from GitHub's documentation. In both of our tests of this lab, the pull request came from an account with write access to the repository, so GitHub never held its runs and we never saw the panel. With nothing held, the button in the upper right corner read **Merge status**, as in the picture in Part 2, step 7.

You should see `docs`, `gds` and `precheck` run and pass, and `viewer` skipped. The merge box lists `docs / docs (pull_request)`, `gds / gds (pull_request)` and `gds / precheck (pull_request)`, each marked Required, and `gds / viewer (pull_request)` as skipped, and it says "Review required" and "Merging is blocked".

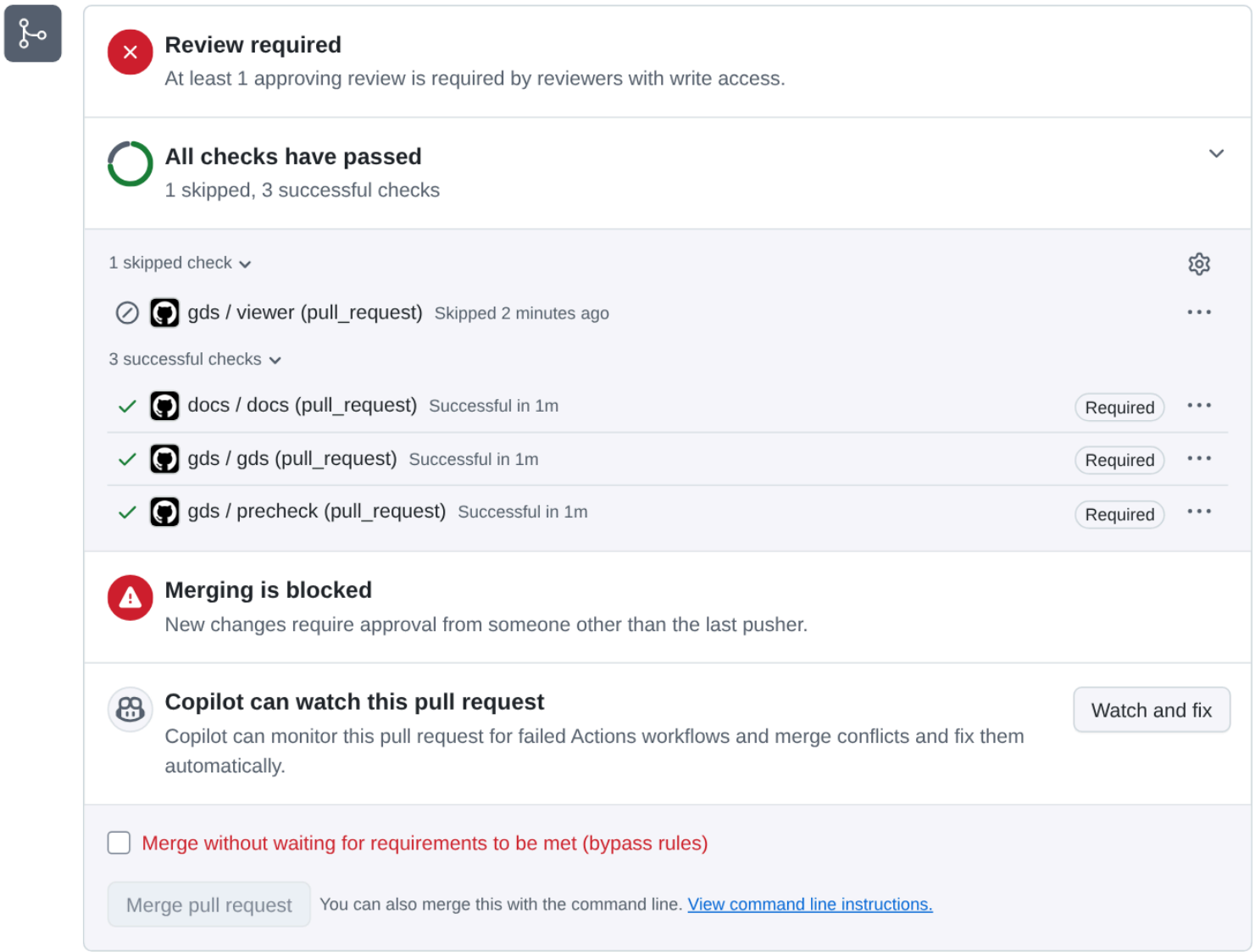


Figure 18.5: A pull request waiting for its approval, with its three required checks passed and viewer skipped.

2. Now claim the slot next to your partner's in your own repository: if they took slot K, take K + 1, or K − 1 if they took 7. In your own clone (`cd /foss/designs/projects/ttsky-ic102-a-review`), make a branch with `git switch -c slot-J-YOUR-USERNAME`, with J your slot, and repeat Part 2's steps 4 to 6 there.
3. Push the branch with `git push -u origin slot-J-YOUR-USERNAME` and open a pull request into your own `main`. Wait for its three checks to go green.

4. Your partner can't approve it: only an approval from someone with write access counts, and on a repository in your account that's only you. Merge it yourself with the bypass: tick **Merge without waiting for requirements to be met (bypass rules)**, and the button below it turns into **Bypass rules and merge**. The ruleset allows only merge commits, so there's no merge method to choose.

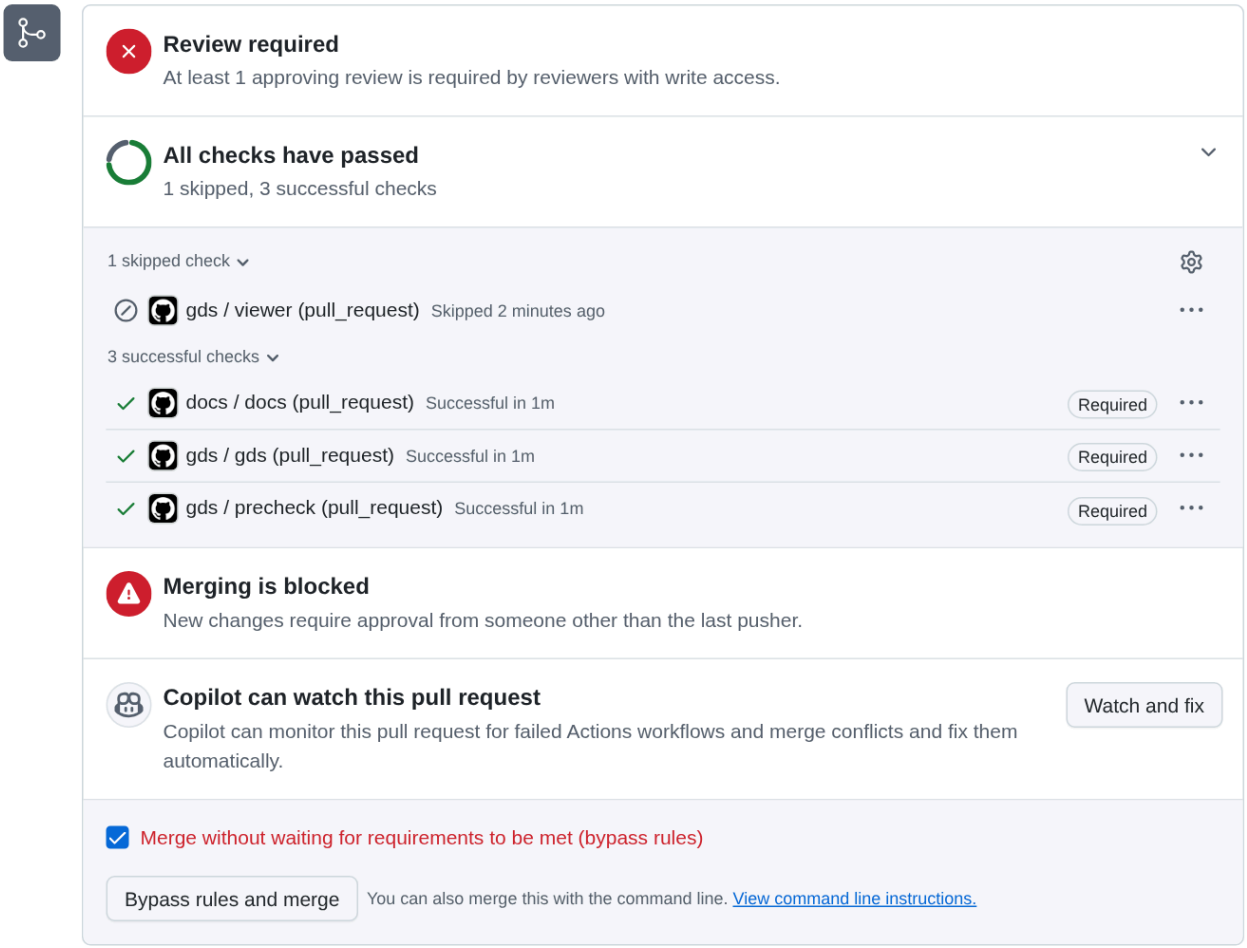


Figure 18.6: Merging your own pull request with the bypass.

5. Before you look at your partner's pull request again, predict which of its five files will conflict with your `main` now. Then open it. You should see "This branch has conflicts that must be resolved", and no new runs, because GitHub's documentation says "Workflows will not run on `pull_request` activity if the pull request has a merge conflict." The checks box keeps the results of the runs that the pull request already had, under the heading "Checks awaiting conflict resolution". While the conflict lasts, the merge box also drops "Review required" and the bypass box.

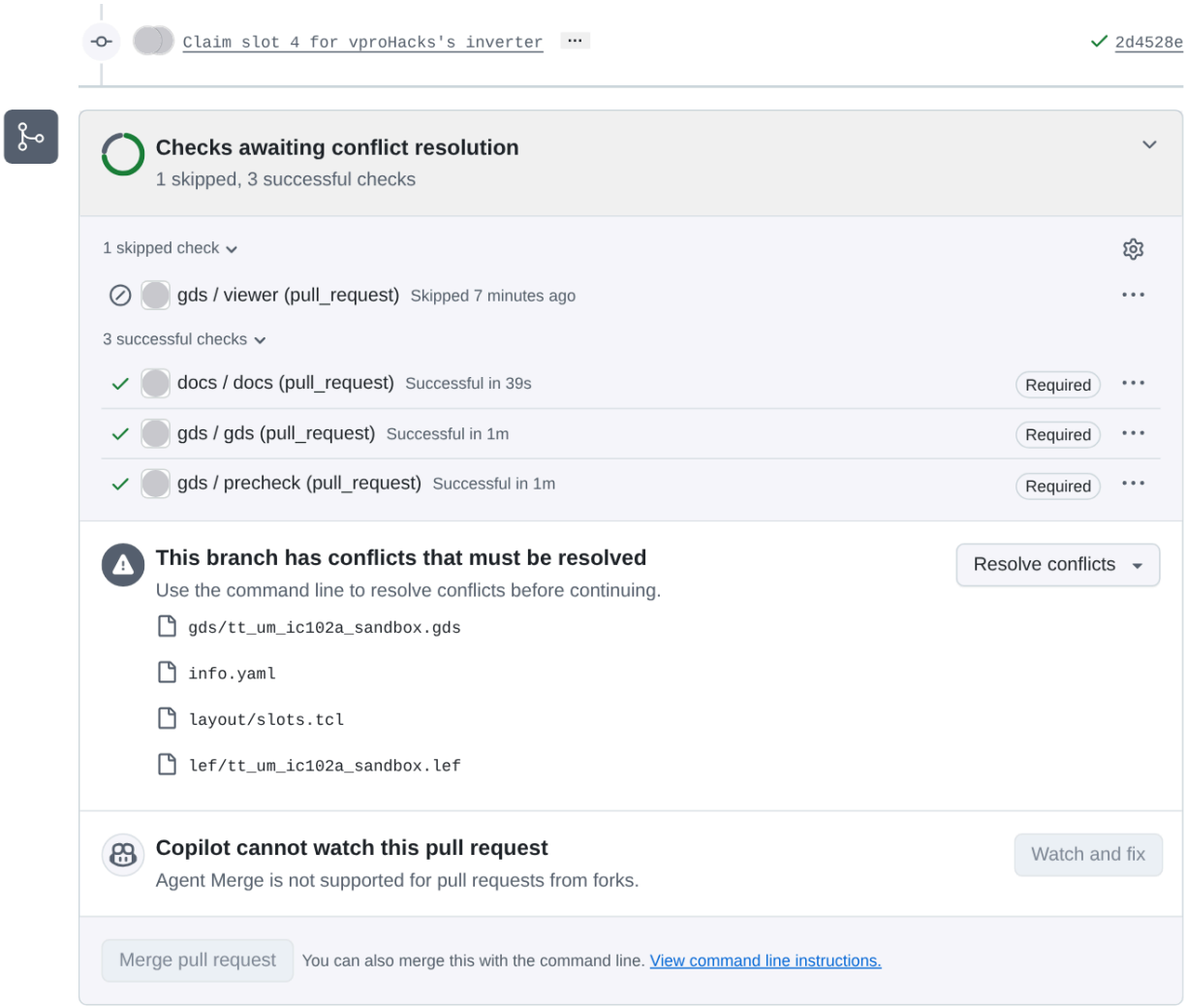


Figure 18.7: Your partner's pull request once your claim is merged: the checks it had, and the four files that conflict. In our test the roles were the other way round. This is our pull request from a fork to uoftasic/ic102-partner-sandbox-a, which stood in for a partner's repository, after the organisation merged its own claim.

Check: your claim is merged into your `main`, and your partner's pull request shows a conflict.

Part 4: Review your partner's pull request

Can you tell, without reading the GDS, whether a layout change is what its description says?

1. Go through `REVIEWING.md` from the top, and note what you checked, with the evidence for each line.
2. Download the `gds_render` artifact from the pull request's `gds` run and open the PNG. Slot K sits at y = 40 + 20K µm, between the two stripes. You should see one more inverter than the empty sandbox has, with a wire up to the top edge on each side. Here's a sandbox with slots 3 and 4 claimed, rendered the same way:

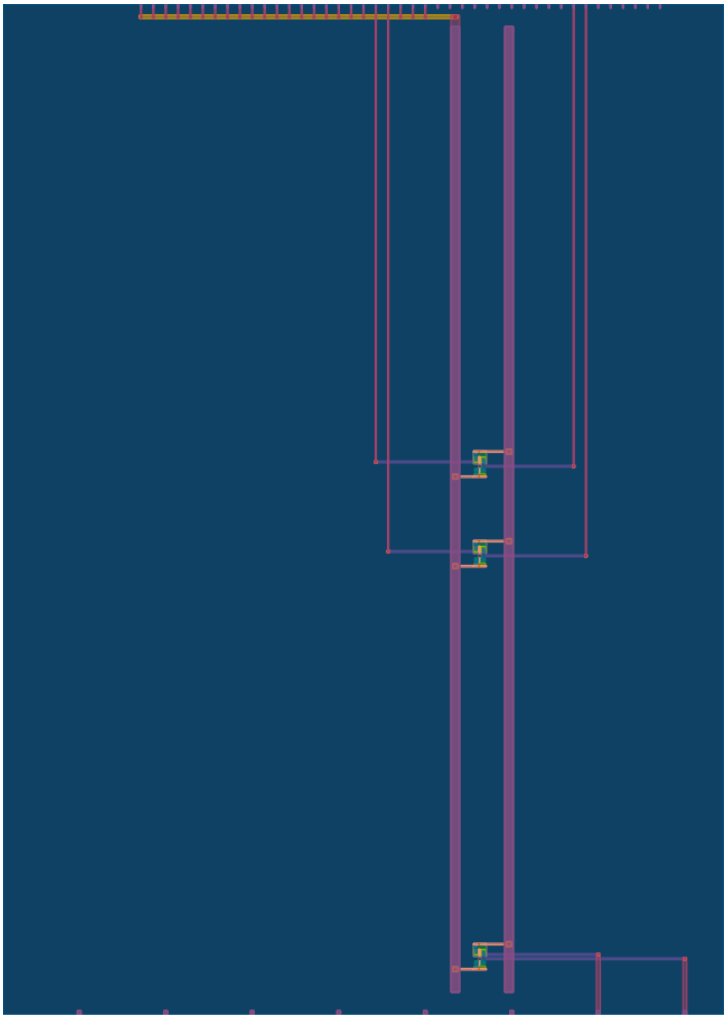


Figure 18.8: The sandbox with two slots claimed: the anchor at the bottom, and two inverters between the stripes with their wires running up to ui_in and uo_out on the top edge.

3. Run LVS on the pull request yourself. GitHub keeps every pull request's commits under a name in your repository, `pull/N/head`, so you can fetch them without adding your partner's fork:

```
cd /foss/designs/projects/ttsky-ic102-a-review
git fetch origin pull/N/head:pr-N
git switch pr-N
make lvs
git switch main
```

   You should see `Final result: Circuits match uniquely.` if the slot is right.

4. Compare the slot in the render with the line in `slots.tcl` and the two pinout lines. A description that says slot 5 for a pinout that describes `ui[4]` is a mistake that precheck can't catch, because it doesn't read the digital pins' descriptions at all.

5. Write the review. On **Files changed**, click **Submit review** above the changed code. It opens the **Finish your review** dialog: type a comment that says what should change and why, select **Request changes**, and click **Submit review** in the dialog (submitting your review). There usually is something, even if it's only a description that's missing the LVS line. Make each request specific enough to act on: your partner can act on "Paste the `Final result` line of `make lvs` from the latest commit", but not on "looks off".

Check: you've submitted a review on your partner's pull request that requests a specific change.

Part 5: Answer the review and resolve the conflict

What happens to an approval when you push again?

1. Back in your fork's clone, make the change your partner asked for, commit it, and push. That push starts no runs, because the pull request still has its conflict; the checks can run again after step 4. Reply to each of their comments and resolve it, because the ruleset blocks the merge while a conversation is still open. Ask your partner to look again; they can approve a pull request that has a conflict.
2. Once they've approved, bring their `main` into your branch. Predict which files will conflict first, then:

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

   In the rehearsal, `bob` had claimed slot 4 and `alice`, the owner, then merged slot 5. `bob` saw:

   ```
   warning: Cannot merge binary files: gds/tt_um_ic102a_sandbox.gds (HEAD vs. upstream/
   main)
   Auto-merging gds/tt_um_ic102a_sandbox.gds
   CONFLICT (content): Merge conflict in gds/tt_um_ic102a_sandbox.gds
   Auto-merging info.yaml
   CONFLICT (content): Merge conflict in info.yaml
   Auto-merging layout/slots.tcl
   CONFLICT (content): Merge conflict in layout/slots.tcl
   Auto-merging lef/tt_um_ic102a_sandbox.lef
   CONFLICT (content): Merge conflict in lef/tt_um_ic102a_sandbox.lef
   Automatic merge failed; fix conflicts and then commit the result.
   ```

   Why does info.yaml conflict when the two of them described different pins?: A3, in Answers at the end.
3. Resolve the two text files by hand, keeping both sides' claims: in `info.yaml`, each pin line keeps whichever side has a description; in `layout/slots.tcl`, keep both `slot` lines, in slot order. Delete every `<<<<<<<`, `=======` and `>>>>>>>` line.
4. Don't resolve `gds/` or `lef/` by taking either side's version. Instead run `make`, which builds both again from the merged sources, check the result, and then finish the merge and push:

   ```
   make && make lvs && make precheck
   git add info.yaml layout/slots.tcl gds lef
   git commit --no-edit
   git push
   ```

   The merge commit's message is `Merge remote-tracking branch 'upstream/main' into slot-K-YOUR-USERNAME`. If `make` stops with `make: can't read project.top_module from info.yaml`, there's a conflict marker left in `info.yaml`, and `make` lists the lines. If it stops with `Error parsing "../build.tcl": slot 3 is claimed twice in slots.tcl` (with your slot number), you and your partner claimed the same slot, and you need to pick another one and change it in both `slots.tcl` and `info.yaml`.
5. The conflict is gone, so the new commit can have runs again, but they may have to wait for your partner to approve them, the way you approved theirs in Part 3. To decide, GitHub checks both the pull request's author and whoever triggered the run. Here both are you, and you still have nothing merged into your partner's repository (repository Actions settings). Before you look, predict whether your pull request is still approved.

   Answer: A4, in Answers at the end.

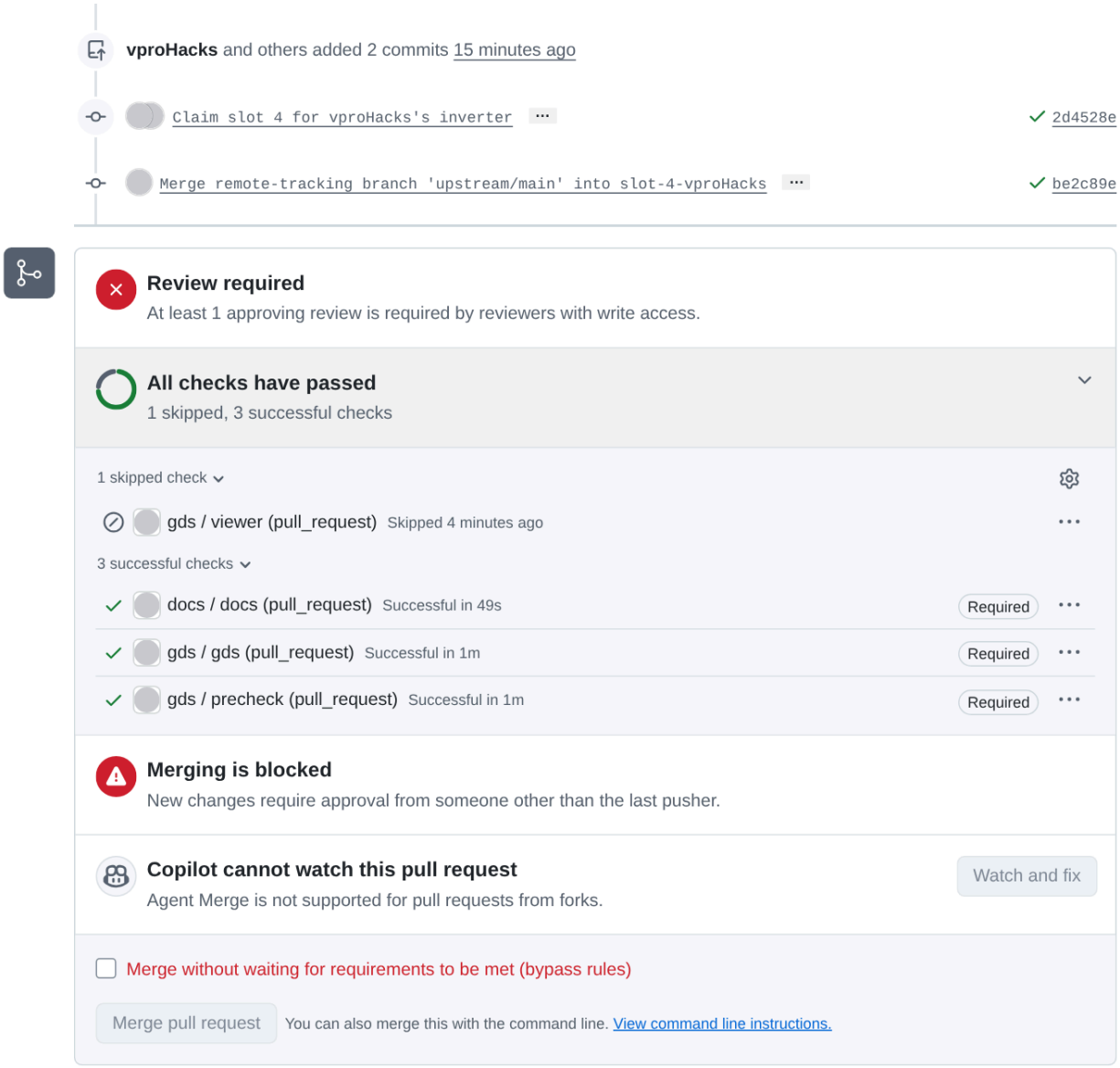


Figure 18.9: Your pull request on your partner's repository once the merge is pushed, in our test, where uoftasic/ic102-partner-sandbox-a stood in for the partner's. The conflict is gone, the three required checks are green again and viewer is skipped, and it still says "Review required" and "Merging is blocked". Nobody had approved it, so the picture can't show an approval being dismissed. Its runs weren't held either, because the account that pushed owns the organisation.

Check: your branch has a merge commit, `make lvs` matches and `make precheck` passes on it, and the pull request shows no conflict.

Part 6: Approve and merge

1. When your partner has pushed their merge, look at their pull request again. If its runs are waiting for approval again, approve them as you did in Part 3, step 1, reading **Files changed** first. Then look at the checks, the new render, and the `slots.tcl` and pinout lines, which now hold both claims. When it's right, approve it: click **Submit review** on **Files changed**, select **Approve** in the **Finish your review** dialog, and click **Submit review** there. Then click **Merge pull request** and **Confirm merge**. GitHub's merging a pull request gives those two clicks for a merge commit, the only kind the ruleset allows. You have to do this yourself, because your partner has no write access to your repository. The merge is a push to `main`, so the workflows run there, `viewer` included, and your Pages site shows the layout with both slots.

2. Look at your repository's history:

```
cd /foss/designs/projects/ttsky-ic102-a-review
git switch main
git pull
git log --oneline --graph
```

In our test of this lab it looked like this, with the oldest commit at the bottom. One account played both parts, so both claims are `vproHacks`'s, and slot 2's branch stood in for a partner's fork:

```
*   195a352 Merge pull request #1 from vproHacks/slot-2-vproHacks
|\
| *   adea902 Merge remote-tracking branch 'upstream/main' into slot-2-vproHacks
| |\
| |/
|/|
* |   fbb2e43 Merge pull request #2 from vproHacks/slot-3-vproHacks
|\ \
| * | 40dd01d Claim slot 3 for vproHacks's inverter
|/ /
| * 6ee85be Name the slot in the pinout descriptions
| * f8867cd Claim slot 2 for vproHacks's inverter
|/
* 9672e13 Start from the IC102-A sandbox
```

`6ee85be` is the change the review asked for, pushed while the pull request still had its conflict.

Check: your partner's pull request is merged, and the `gds` run on your `main` is green with all three jobs.

Part 7: A pull request written by an AI tool (optional)

This part takes about 20 minutes. The package has a pull request that an AI coding tool wrote. We left problems in it on purpose, and every check passes on it. Each of you opens it on the other's repository and reviews the copy opened on your own.

1. In your fork's clone, with your own pull request merged, run:

```
cd /foss/designs/projects/ttsky-ic102-a-review-fork
git switch main
/foss/designs/modules/ic102-a/labs/lab-04-a-change-through-review/ai-pr/ai-pr.sh
```

It branches from your partner's `main`, commits the change to a free slot, pushes the branch to your fork, and prints a title and a description. Open a pull request from that branch into your partner's `main`, using that title and description, without reading the change yourself.

2. When your partner's copy arrives in your repository, approve its runs if GitHub holds them (it shouldn't, now that your partner has had a pull request merged). Review it as you reviewed theirs, but treat every statement in its description as a claim to check, as Pull requests and review suggests. Request changes, and name each problem you find with its file and line. Don't approve it or merge it; close it when you're done.

Answer: A5, in Answers at the end.

Finishing

You're done when your pull request on your partner's repository has been merged after your partner approved it, and your review on your own repository requested a change that your partner then pushed.

Questions:

1. In Part 5 you rebuilt `gds/` with `make` instead of picking one side's file. What would have been wrong with `git checkout --theirs gds/`?

Answer: A6, in Answers at the end.

2. The ruleset requires `docs`, `gds` and `precheck` but not `viewer`. Why?

Answer: A7, in Answers at the end.

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

Next on the Course path: read History as evidence, then do Lab 05: Find the commit that broke it.

Answers

A1. Part 1: Your review repository

It's refused, and GitHub's message names the rule that requires a pull request. In our test repository, `git 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-a/rules?ref=refs%2Fheads%2Fmain
remote:
remote: - Changes must be made through a pull request.
remote:
remote: - 3 of 3 required status checks are expected.
remote:
To https://github.com/vproHacks/ic102-review-sandbox-a.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-
a.git'
```

The bypass is "For pull requests only": it lets you merge a pull request of your own without an approval (you'll use that in Part 3), but it doesn't let you push to `main` directly.

A2. Part 2: Claim a slot in your partner's repository (Which of these lines can a reviewer actually read?)

`slots.tcl` and `info.yaml`, five changed lines between them. The cell is 116 lines of Magic's format, which reviewers don't read line by line, and it's a copy of a cell that already passed LVS. The GDS is binary, so the diff shows only a change of size. The LEF is text, but it's generated, so its 57 lines say nothing a reviewer can act on. That's why your description has to carry the evidence, and why the reviewer looks at the render and the precheck table, as Layout files in git says.

A3. Part 5: Answer the review and resolve the conflict (Why does info.yaml conflict when the two of them described different pins?)

git compares changes line by line, and treats two changes to neighbouring lines as one conflict, because it can't know whether they depend on each other. `ui[4]` and `ui[5]` are next to each other in the pinout:

```
<<<<<<< HEAD
  ui[4]: "bob's inverter, input"
  ui[5]: ""
=======
  ui[4]: ""
  ui[5]: "alice's inverter, input"
>>>>>>> upstream/main
```

`slots.tcl` conflicts because both added a line at its end, and the GDS and LEF because both sides rebuilt them.

A4. Part 5: Answer the review and resolve the conflict

No. The ruleset dismisses an approval when new commits arrive, because your partner approved the commits they saw, and the merge you pushed is a new layout that neither of you has reviewed. Ask them to look again. GitHub's pull request rule calls the setting "Dismiss stale pull request approvals when new commits are pushed".

A5. Part 7: A pull request written by an AI tool (optional)

There are three, and none of them turns a check red.

- `gds/` and `lef/` weren't rebuilt, so every check ran on `main`'s old layout. The file list has no GDS, the render has no new inverter, and `make lvs` on the branch warns that `layout/slots.tcl layout/cells/inv_ai.mag changed after gds/tt_um_ic102a_sandbox.gds was written` and reports `Final result:`

`Top level cell failed pin matching.`, so the description's LVS line is false. Its precheck line is true, for the old GDS.
- The pinout describes `ui[K+1]` and `uo[K+1]` for slot K. Neither precheck nor LVS reads it.
- `.gitignore` now lists `gds/` and `lef/` as build output, which the claim didn't need and which is wrong for a template whose `gds` workflow reads both from the repository.

A6. Finishing

"Theirs" is `main`'s GDS, which has your partner's slot and not yours; "ours" has yours and not theirs. Neither is the layout of the merged sources. The only correct GDS is one built from the merged `slots.tcl`, which is what `make` does. A GDS committed from the wrong side would still pass precheck, because both sides are legal layouts, so nothing on GitHub would tell you.

A7. Finishing

`viewer` is skipped on pull requests, so it never reports a pass or a fail on one. It runs when the merge is pushed to `main`.



5. History

19 History as evidence 131
19 What the history keeps 131
19 Reading a history 131
19 Finding the commit that broke it 132
19 Marking a commit 134
19 Check yourself 134
19 Answers 134

20 Lab 05: Find the commit that broke it 137
20 Before you start 137
20 Part 1: The history 137
20 Part 2: A first try 138
20 Part 3: A script git can trust 138
20 Part 4: Bisect 139
20 Part 5: Undo it through a pull request 140
20 Finishing 143
20 Answers 143

19. 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:

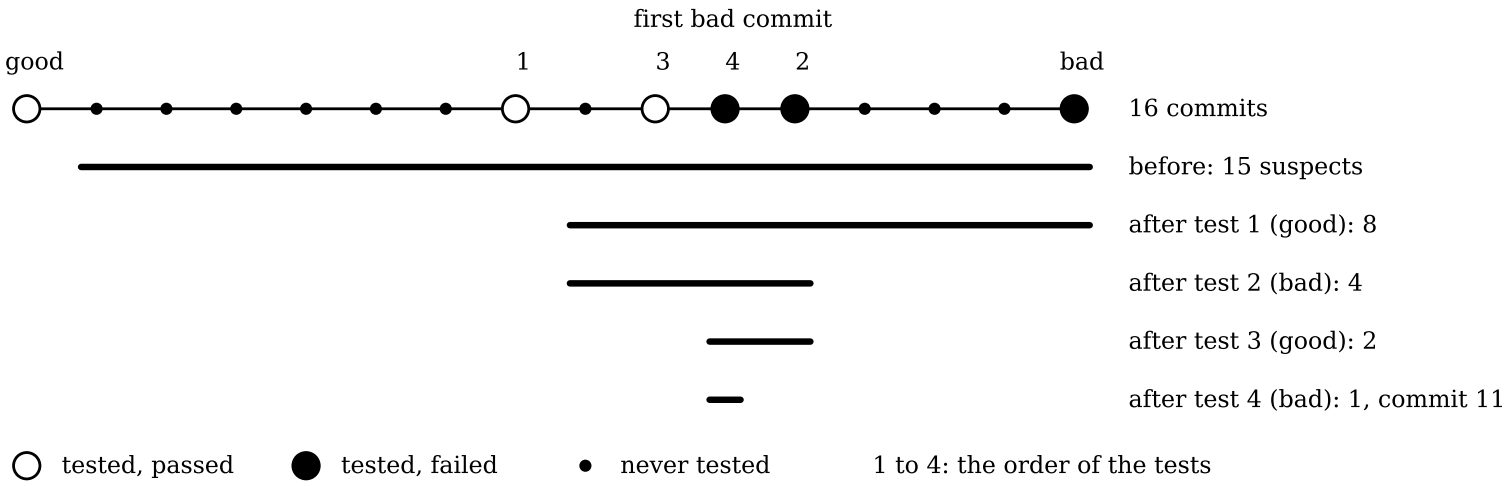


Figure 19.1: git bisect on 16 commits, where commit 1 is known to be good and commit 16 bad. Test 1 checks commit 8, which passes; test 2 checks 12, which fails; test 3 checks 10, which passes; test 4 checks 11, which fails. The suspects shrink from 15 to 8, 4, 2 and finally 1, commit 11, the first bad 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.

20. Lab 05: Find the commit that broke it

You'll get the history of a small analog project, 33 commits long. Its `gds` workflow went red some time after its last good release, and that release is now too long ago for anyone to remember what broke it. You'll write a script that `git bisect run` can trust, use it to find the commit that broke Magic DRC, and undo that commit through a pull request.

Package: `labs/lab-05-find-the-commit-that-broke-it/`

Before you start

- Read History as evidence.
- Finish Lab 03. This lab uses the course tools from Lab 01 and your token, but not your `ttsky-ic102-a`: the history gets a repository of its own in Part 5.

This lab takes about 55 minutes.

Part 1: The history

What does the history tell you before you test anything?

1. Clone the history from the bundle file that the course ships, into a folder named after the repository you'll put it in on GitHub in Part 5, and set your identity in it:

   ```
   mkdir -p /foss/designs/projects && cd /foss/designs/projects
   git clone /foss/designs/modules/ic102-a/labs/lab-05-find-the-commit-that-broke-it/
   history.bundle ttsky-ic102-a-bisect
   cd ttsky-ic102-a-bisect
   git config user.name "Your Name"
   git config user.email "ID+USERNAME@users.noreply.github.com"
   git config credential.helper 'cache --timeout=28800'
   ```

   A bundle is a whole repository in one file. The commits in it have fixed authors and dates, so your hashes are the same as the ones below.

2. Look around:

   ```
   git log --oneline -3
   git tag -n
   git log --oneline -1 v0.1
   git log --oneline v0.1..main | wc -l
   ```

   You should see `main` at `fe045fa README: note that the gds workflow is red`; one tag, `v0.1`, described as `First layout that passes precheck and LVS`, on `4227d07 Place the inverter and wire it to ua[0] and ua[1]`; and 26 commits after it.

3. Confirm that `main` is broken the way the README says, with the runner's tools:

   ```
   make precheck
   ```

   You should see two ❌ rows, `Magic DRC | ❌ Fail: Magic DRC failed` and `KLayout BEOL | ❌ Fail: Klayout beol failed with 1 DRC violations`.

4. `git bisect` halves the suspects with every test. With 26 commits after the good one, how many tests should it need? Write the number down.

   Answer: A1, in Answers at the end.

Check: you know the good commit (`v0.1`), the bad one (`main`), and how many suspects lie between.

Part 2: A first try

Can you hand git Magic's exit status and be done?

1. The package has a script that runs precheck's first check, Magic DRC, on one GDS file, with precheck's own Magic and PDK: `magic-drc.sh GDS TOP_CELL REPORT_DIR`. Run it on `main`:

   ```
   LAB=/foss/designs/modules/ic102-a/labs/lab-05-find-the-commit-that-broke-it
   $LAB/magic-drc.sh gds/tt_um_ic102_lab05.gds tt_um_ic102_lab05 /tmp/lab05-drc; echo
   "exit=$?"
   head -6 /tmp/lab05-drc/magic_drc.txt
   ```

   You should see `exit=255`, and a report that starts with the rule that's broken:

   ```
   tt_um_ic102_lab05
   ----------------------------------------
   Metal2 spacing < 0.14um (met2.2)
   ----------------------------------------
    141.200 12.710 142.740 12.750
    142.600 12.710 142.740 12.750
   ```

   and, two lines from its end, `[INFO]: COUNT: 4`.

2. Read `$LAB/first-try.sh`. It runs that script at whatever commit git has checked out and exits with its status. Before you run it, read the rule from the git manual that History as evidence quotes, and write down what git will do with an exit status of 255.

3. Run it:

   ```
   git bisect start main v0.1
   git bisect run $LAB/first-try.sh
   git bisect reset
   ```

   You should see the first test pass and the second stop the whole bisect:

   ```
   Bisecting: 12 revisions left to test after this (roughly 4 steps)
   [b08fd6d6b800b1136b1b8972582ebc554781965d] Keep build output out of git
   running  '/foss/designs/modules/ic102-a/labs/lab-05-find-the-commit-that-broke-it/
   first-try.sh'
   Bisecting: 6 revisions left to test after this (roughly 3 steps)
   [02d92af0591d429d331499929756e07d71531c48] README: rename the build section
   running  '/foss/designs/modules/ic102-a/labs/lab-05-find-the-commit-that-broke-it/
   first-try.sh'
   error: bisect run failed: exit code 255 from '/foss/designs/modules/ic102-a/labs/
   lab-05-find-the-commit-that-broke-it/first-try.sh' is < 0 or >= 128
   ```

   Answer: A2, in Answers at the end.

Check: you've seen `bisect run failed: exit code 255`, and run `git bisect reset`.

Part 3: A script git can trust

What can go wrong at a commit, besides the layout being bad?

1. The first test above was `b08fd6d Keep build output out of git`, and `first-try.sh` called it good. Check out that commit and look at what Magic did:

   ```
   git checkout b08fd6d
   ls gds/
   ```

```
$LAB/magic-drc.sh gds/tt_um_ic102_lab05.gds tt_um_ic102_lab05 /tmp/lab05-drc; echo
"exit=$?"
grep -A2 "couldn't be read" /tmp/lab05-drc/magic.log
```

You should see `ls: cannot access 'gds/': No such file or directory`, `exit=0`, and:

```
Cell tt_um_ic102_lab05 couldn't be read
No such file or directory
Creating new cell
```

There's no layout at this commit at all. Magic couldn't find the cell, made an empty one, checked it, and found nothing wrong. Its report says `COUNT: 0` too, so a script that reads only the count is fooled just as `first-try.sh` was.

2. List the commits after `v0.1` that changed the GDS, with the size of each version:

```
git log --oneline --stat v0.1..main -- gds/
```

Find the second commit that can't be tested: one where the GDS shrank to 6000 bytes and the next commit put it back. Check it out and run `magic-drc.sh` on it. You should see `exit=255`, `COUNT: 58`, and in `magic.log`, twice:

```
Error while reading cell "tt_um_ic102_lab05" (byte position 6000): Unexpected EOF.
```

That's `5e0f367 Add a second inverter (WIP)`, a GDS cut short when Magic crashed while writing it. Magic read what was there and reported 58 errors in the fragment, so a script would call this commit bad when in fact it can't be judged at all.

3. Go back with `git checkout main`, and write your own script, `/foss/designs/projects/bisect.sh`, outside the repository so that bisect's checkouts can't change it. It has to exit:

- **125** when there's no `gds/<top_module>.gds`, or Magic's log shows it couldn't read the cell, or the report has no `COUNT` line;
- **1** when the count is above 0;
- **0** when the count is 0.

Read `top_module` from `info.yaml` with `python3 -c 'import yaml; print(yaml.safe_load(open("info.yaml"))["project"]["top_module"])'`, run `magic-drc.sh` without using its exit status, and decide from `magic.log` and `magic_drc.txt`. Then `chmod +x` it.

4. Test it by hand on the four kinds of commit before you give it to git:

```
for c in v0.1 b08fd6d 5e0f367 main; do git checkout -q $c; /foss/designs/projects/
bisect.sh > /dev/null 2>&1; echo "$c $?"; done
git checkout -q main
```

You should see `v0.1 0`, `b08fd6d 125`, `5e0f367 125` and `main 1`. If not, fix the script before Part 4, because git decides from your script's exit status alone, and one wrong answer sends the bisect into the wrong part of the history.

Check: your script gives those four answers.

Part 4: Bisect

1. Run it:

```
git bisect start main v0.1
git bisect run /foss/designs/projects/bisect.sh
```

It takes a few seconds. You should see seven tests, the first one skipped, and then:

```
380836cbb3704d4afa0915f9b43a4bb4d3bf0111 is the first bad commit
commit 380836cbb3704d4afa0915f9b43a4bb4d3bf0111
Author: Sam Example <sam@example.com>
Date:   Sun Oct 11 22:40:00 2026 -0400

    Widen Y's wire to ua[1]

     The output wire carries more current than the input one, so give it twice
the width.

 gds/tt_um_ic102_lab05.gds | Bin 19636 -> 19572 bytes
 layout/wires.tcl          |   2 +-
 2 files changed, 1 insertion(+), 1 deletion(-)
bisect found first bad commit
```

That's seven tests against the five you predicted, because git skipped `b08fd6d` and had to choose another commit near the middle, and the skipped test still counts as one of the seven.

2. Run `git bisect reset`, then read the commit: `git show 380836c -- layout/wires.tcl`. The change is one number: the bottom edge of `Y`'s metal 2 moved from 13.26 to 12.85 µm. `A`'s metal 2 runs from 12.3 to 12.75 µm. How far apart are they now, and what's the minimum?

   Answer: A3, in Answers at the end.

3. A classmate's script reads only the `COUNT` line and exits 0 when it's 0. Which commit would git have named with it?

   Answer: A4, in Answers at the end.

Check: your bisect named `380836c`.

Part 5: Undo it through a pull request

1. Put the history on GitHub, so that its workflows run. Make a new repository (the + menu › **New repository**) with your own account as the owner, the name `ttsky-ic102-a-bisect`, **Public**, and no README, `.gitignore` or licence. Then point your clone at it, instead of at the bundle, and push `main`:

```
git remote set-url origin https://github.com/YOUR-USERNAME/ttsky-ic102-a-bisect.git
git push -u origin main
```

   The history includes `.github/workflows/`, so this is one of the pushes that needs your token's Workflows permission.

2. Turn on Pages (Settings › Pages › Source: **GitHub Actions**), then start a new run: Actions › `gds` › **Run workflow** on `main`, not "Re-run failed jobs". You should see `docs` green, and in the `gds` workflow, the `gds` and `viewer` jobs green and `precheck` red. That's the state the README describes.

   When we tested this on Nov 16, 2026 (with the history pushed to a branch of a test repository rather than to a repository of its own), the push's `docs` run passed, and its `gds` run had `gds` green, `viewer` red with `Failed to create deployment (status: 404)`, and `precheck` red with the two rows from Part 1:

```
| Magic DRC | ❌ Fail: Magic DRC failed |
| KLayout BEOL | ❌ Fail: Klayout beol failed with 1 DRC violations |
```

   The run started with Run workflow after Pages was on took 2 min 51 s, with `gds` and `viewer` green and `precheck` red with the same two rows.

3. Make a branch from `main` and revert the commit:

```
git switch -c lab05-revert
git revert --no-edit 380836c
```

You should see:

```
warning: Cannot merge binary files: gds/tt_um_ic102_lab05.gds (HEAD vs. parent of
380836c (Widen Y's wire to ua[1]))
Auto-merging gds/tt_um_ic102_lab05.gds
CONFLICT (content): Merge conflict in gds/tt_um_ic102_lab05.gds
Auto-merging layout/wires.tcl
error: could not revert 380836c... Widen Y's wire to ua[1]
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git revert --continue".
hint: You can instead skip this commit with "git revert --skip".
hint: To abort and get back to the state before "git revert",
hint: run "git revert --abort".
```

Why does the GDS conflict when the text file doesn't?: A5, in Answers at the end.

4. Resolve it the way you did in Lab 04: rebuild the GDS and LEF from the reverted sources, then check and finish the revert:

```
make
git add gds lef
git revert --continue --no-edit
make lvs
make precheck
```

`--no-edit` keeps git's message, `Revert "Widen Y's wire to ua[1]"`, instead of opening an editor. You should see `Final result: Circuits match uniquely.` and `Precheck passed`.

5. Push the branch and open a pull request from `lab05-revert` into `main`:

```
git push -u origin lab05-revert
```

In the description, name the commit you're reverting, say how you found it, and paste the `make lvs` and `make precheck` lines. The template's workflows run on every push, so the branch gets its own `docs` and `gds` runs, and the pull request shows them.

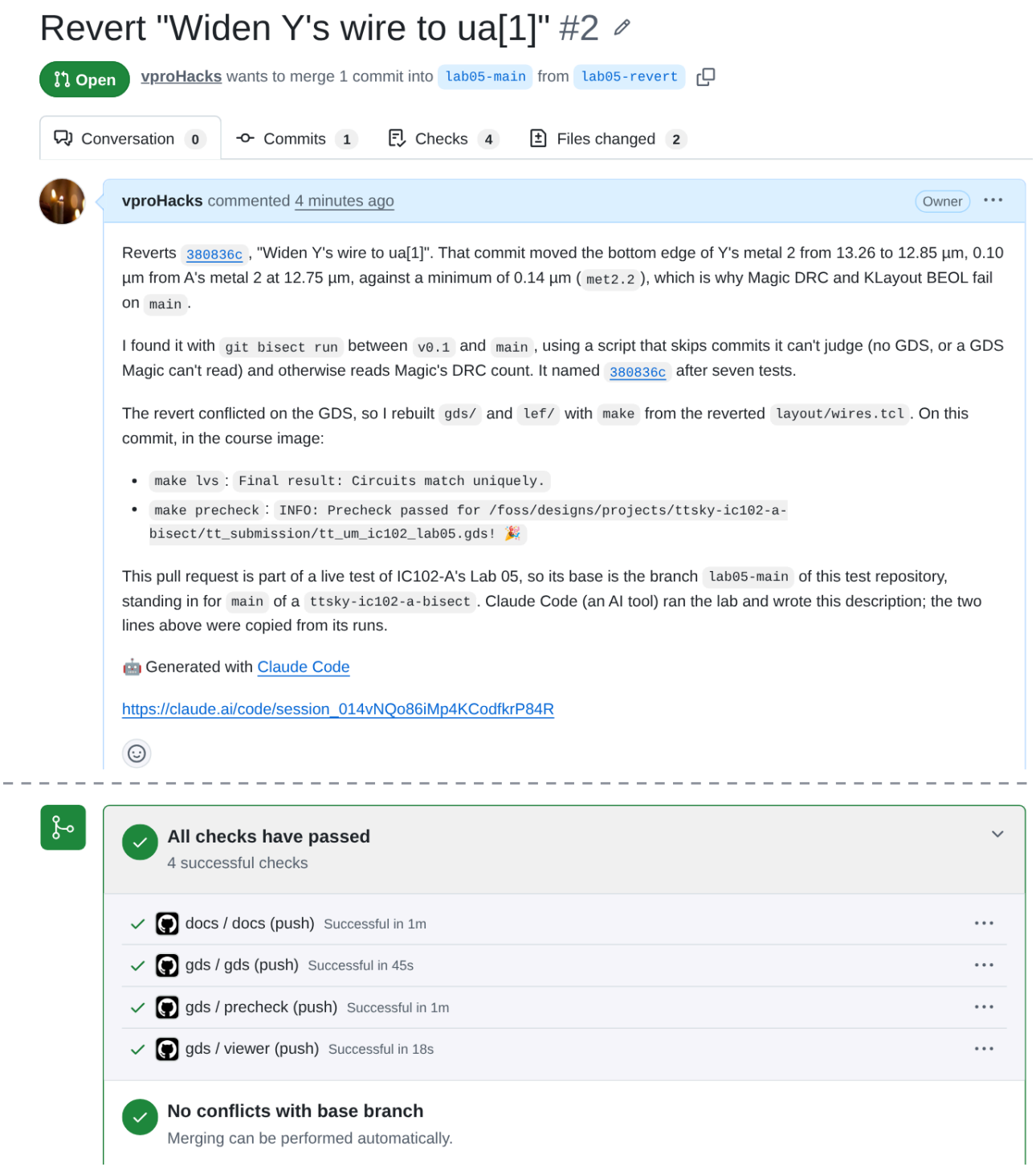


Figure 20.1: The revert's pull request, with its description and its four checks green.

6. When the branch's `gds` run is green in all three jobs, merge the pull request with **Create a merge commit**. The merge is a push to `main`, so it gets a run of its own. In our test, the branch's `gds` run passed all three jobs in 2 min 44 s (`gds` 45 s, `precheck` 1 min 54 s, `viewer` 18 s), and the run on the merge passed all three in 3 min 8 s.

Check: the `gds` run on `main` after the merge is green in all three jobs.

Finishing

You're done when your bisect has named `380836c`, and `main` of your `ttsky-ic102-a-bisect`, with the revert merged, has a green `docs` run and a green `gds` run with all three jobs.

Questions:

1. Your script had to treat a missing GDS as "can't test". In the history, is there anything else that would make a commit untestable for this check, even with a GDS that Magic reads?

   Answer: A6, in Answers at the end.

2. The good commit was easy to find because someone tagged it. What would you have used without the tag?

   Answer: A7, in Answers at the end.

Submit: nothing for this lab.

This is the last row of the Course path.

Answers

A1. Part 1: The history

Five: $2^5 = 32 \geq 26$, and four halvings would leave two suspects. git prints its own guess as it starts, "roughly 4 steps" after the first test.

A2. Part 2: A first try

git accepts 0 (good), 1 to 127 except 125 (bad) and 125 (skip). Anything else, and 255 in particular, it treats as the script itself having failed, and it stops. Magic's DRC script exits with `exit -1`, which the shell sees as 255, so the first commit with an error stops the bisect. Mapping 255 to 1 would fix that. It wouldn't fix what went wrong in the first test, though, and that's what Part 3 is about.

A3. Part 4: Bisect

12.85 − 12.75 = 0.10 µm, against a minimum of 0.14 µm for metal 2 (`met2.2` in Magic's report). The commit's message gives a good reason for a wider wire, and the wire is wider; it just grew towards its neighbour. git bisect never reads commit messages, only your script's exit status.

A4. Part 4: Bisect

`a4605db Put gds/ and lef/ back: the gds workflow reads them from the repository`. The script calls `b08fd6d` good even though it has no GDS, so git looks only at the commits after it, and the next commit with a GDS is the one that put it back. That commit's message would make a very convincing culprit. We ran that script on this history to check, and that's the commit it named.

A5. Part 5: Undo it through a pull request (Why does the GDS conflict when the text file doesn't?)

Undoing a commit means applying its change backwards to today's files. For `wires.tcl` that's easy, because no later commit changed that line. Three later commits changed the GDS, and git can't apply a binary change to a different binary, so it stops and asks.

A6. Finishing

A `top_module` in `info.yaml` that doesn't match the cell inside the GDS, as in Lab 02's breakage 3: Magic would look for the wrong cell, find none, and check an empty one. Your script catches that through the "couldn't be read" line, and that's one reason to judge from Magic's log rather than from guesses about which files exist.

A7. Finishing

A commit with a green `gds` run on GitHub, if its runs were still there (they're kept 90 days by default), or the oldest commit with a GDS, tested by hand with the same script. Tagging the commit you'd stand behind, as History as evidence suggests, saves this search.