The Tiny Tapeout workflows

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

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

One push, six jobs

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

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

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

docs

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

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

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

test

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

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

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

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

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

gds

The gds job

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

- **"Fetch verilog and build config"** runs `tt_tool.py --create-user-config`. It reads `info.yaml`, checks that every file in `source_files` exists in `src/`, and turns your tile size and top module into LibreLane's configuration. An `info.yaml` problem stops the job here, within the first minute, with a line that starts `Error loading .../info.yaml:`.
- **"Make GDS with LibreLane"** runs the whole flow: lint, synthesis with Yosys, floorplan, placement, clock tree, routing, and LibreLane's own DRC, LVS and antenna checks. At the `ttsky26d` tag the action installs LibreLane 3.0.14 and runs it in LibreLane's Docker image. A design that doesn't fit its tiles, or that has an error Yosys or the linter catches, stops here.
- **"Linter output"**, **"Yosys warnings"**, **"Routing summary"** and **"Cell usage summary"** write tables to the run's summary page, and **"Publish build logs"** uploads the whole run folder as `GDS_logs`. "Linter output" and

"Publish build logs" run even after an earlier step has failed, and when the build stopped early, "Linter output" prints an error of its own about a log file that doesn't exist.
- **"Prepare tt_submission artifact"** collects the GDS, the LEF, the gate-level netlist and a `pdk.json` naming the PDK commit, which is what the next three jobs download.

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

precheck

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

gl_test

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

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

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

viewer

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

What green means in this course

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

Tags and support tools that move

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

Check yourself

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

Answer: A1, in Answers at the end.

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

Answer: A2, in Answers at the end.

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

Answer: A3, in Answers at the end.

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

Answers

A1. Check yourself

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

A2. Check yourself

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

A3. Check yourself

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