Lab 3 — Testing the Pipeline

AI-Assisted Programming for PhD Researchers

In Lab 2 you built a spec-driven cleaning module and trusted the spec as your contract. A spec says what the code *should* do; a test *proves* it still does. In this lab you turn the agent's work into something you can re-check with a single command, and you do it in three movements. First you **pin the current behavior** with characterization tests, so nobody can quietly break it. Then you **drive a new module test-first**: the test comes before the code, and the code is done when the test is green. Finally you **debug systematically**: you plant a realistic bug and hunt it down with the agent instead of guessing.

Budget about **40 minutes**. Continue in the same repository, on the state you reached at the end of Lab 2.

Task 1 — Characterization Tests

Goal: lock in the behavior your cleaning module *already* has, so any future change that breaks it fails loudly. You write the first test by hand: one test, typed by you, before you let the agent write the rest.

Add pytest as a development dependency:

```
uv add --dev pytest
```

Now create `tests/test_io.py` and type this test in yourself. Do not paste it from the agent. Writing one test by hand is how you learn what a test *is*:

```
# tests/test_io.py
import pandas as pd
from pipeline.io import load_raw, clean

def test_na_humidity_becomes_nan(tmp_path):
    p = tmp_path / "m.csv"
    p.write_text(
"reading_id,station,timestamp,temperature,temp_unit,humidity_pct,wind_ms\n"
        "R0001,Alpha,2025-03-01 06:00,12.0,C,n/a,3.0\n"
    )
    df = clean(load_raw(p))
    assert pd.isna(df.loc[0, "humidity_pct"])
```

The test writes a tiny one-row CSV into pytest's `tmp_path`, runs it through your own `load_raw` and `clean`, and asserts the one thing that matters: the `n/a` humidity became a real missing value. Run the suite:

```
uv run pytest
```

You should see one test pass (green). If it is red, your Lab 2 module and this test disagree about what "missing" means; find out which one is wrong before you go on.

Now let the agent write the rest, one test per acceptance criterion:

> **Prompt**
>
> Read `docs/spec-cleaning.md`. Write pytest tests in `tests/test_io.py` covering every remaining acceptance criterion: °F conversion (with an exact expected value), the `-99` sentinel, decimal-comma parsing, `calm` wind, whitespace in station names, and duplicate-row dropping. One test per criterion. Do not change anything in `src/`.

When it finishes, **read every test line by line** before you run them. A test you did not read is not a test: it is a second, unverified copy of the code. For each one, ask: does the input row actually trigger the case, and is the asserted value the *correct* answer, not just whatever the code happens to return today?

Run the full suite again:

```
uv run pytest
```

> **Checkpoint**
>
> `tests/test_io.py` contains your hand-written test plus one agent-written test per acceptance criterion, and `uv run pytest` is green. You have read each generated test and confirmed it exercises the case it claims to.

> **Explain it**
>
> 1. Find the °F conversion test the agent wrote. Take its raw input temperature (the value with `temp_unit` = `F`) and **compute the Celsius value yourself**: `(value - 32) × 5 / 9`. Does your result equal the number the test asserts? If the test just copied whatever the code produced, a wrong formula would still pass. Show the arithmetic that proves the expected value is genuinely correct.
> 2. Your hand-written test uses `pd.isna(...)` rather than `== None` or `== "n/a"`. Why must a "this value is missing" check use `pd.isna` and not an equality comparison? What does `NaN == NaN` evaluate to?

Task 2 — TDD a Statistics Module

Goal: build a brand-new module the other way around: the test first, the code second. You will write a failing test that *is* the specification, then have the agent make it pass without inventing anything the test did not ask for.

Create `tests/test_stats.py` with this test. It refers to a module that does not exist yet, so it *must* fail when you run it — that is the point:

```
# tests/test_stats.py
import pandas as pd
from pipeline.stats import station_summary

def test_station_summary_means():
    df = pd.DataFrame({
        "station": ["Alpha", "Alpha", "Bravo"],
        "temperature": [10.0, 14.0, 20.0],
        "humidity_pct": [50.0, None, 80.0],
    })
    out = station_summary(df)
    assert out.loc["Alpha", "temp_mean"] == 12.0
    assert out.loc["Alpha", "hum_mean"] == 50.0
    assert out.loc["Bravo", "temp_max"] == 20.0
```

Run it and confirm it is red for the right reason: an import error, because `pipeline.stats` does not exist:

```
uv run pytest
```

Now hand the agent the test as a fixed contract:

Prompt

`tests/test_stats.py` is the spec and is read-only. Do not modify it. Create `src/pipeline/stats.py` with a function `station_summary(df)` that makes the test pass. It returns a per-station summary indexed by station, with columns `temp_mean`, `temp_min`, `temp_max`, and `hum_mean`. The aggregations must be NaN-aware: missing values are skipped, not treated as zero.

Run the suite until it is green:

```
uv run pytest
```

Green means the code satisfies the contract you wrote: no more, no less. The test drove the design; the agent only filled it in.

Checkpoint

`src/pipeline/stats.py` exists, `tests/test_stats.py` is unchanged, and `uv run pytest` passes every test (Task 1's and Task 2's together).

⚠ Explain it

1. Alpha's two humidity readings in the test are `50.0` and `None`. The test asserts `hum_mean == 50.0`. **Work out by hand** why the answer is `50.0` and not `25.0`: what does a NaN-aware mean do with the missing value, divide by two, or by one? Check the pandas docs for `mean()` if you are unsure, and state which denominator it used.
2. The test never says "index the result by station name" in prose, yet it relies on `out.loc["Alpha", ...]`. Point to the exact lines in the test that pin down the shape of the returned object. This is why the test *is* the spec.

Task 3 — A Debugging Hunt

Goal: practice systematic debugging. You will introduce a realistic, plausible-looking bug (the kind an agent might cheerfully suggest as a "cleanup") and then use the failing test and the agent to track it down by evidence, not by guessing.

Open `src/pipeline/stats.py` and replace the body of `station_summary` with this "optimized" version. It looks tidy and it runs without error:

```
def station_summary(df):
    df = df.fillna(0)  # "cleanup" - this is the bug
    g = df.groupby("station")
    return pd.DataFrame({
        "temp_mean": g["temperature"].mean(),
        "temp_min": g["temperature"].min(),
        "temp_max": g["temperature"].max(),
        "hum_mean": g["humidity_pct"].mean(),
    })
```

Run the suite:

```
uv run pytest
```

`test_station_summary_means` now fails. Do **not** immediately ask the agent to "just fix it": that is how you trade one bug for another. Instead, run the debugging protocol *with* the agent:

ℹ Prompt

`test_station_summary_means` is failing. First reproduce the failure and show me the actual vs. expected values. Then give me three ranked hypotheses for the cause, each with the specific evidence that supports it, before you change any code.

Read the hypotheses and **judge them yourself** against the evidence. The correct one is that `fillna(0)` runs *before* the aggregation, so Alpha's missing humidity is turned into a

real `0.0` and dragged into the mean: `(50 + 0) / 2 = 25.0` instead of `50.0`. Only once you agree on the diagnosis, approve the fix: remove the `fillna(0)` and restore the NaN-aware aggregation. Keep the test as it is; it now stands guard as a regression test.

Run the suite one last time to confirm green:

```
uv run pytest
```

Checkpoint

`station_summary` no longer calls `fillna(0)`, `uv run pytest` is green again, and the test that caught the bug is still in place. Commit your work. The reference state for this lab is tagged `lab-03-done`.

If you fell behind, catch up with:

```
git fetch --tags && git checkout tags/lab-03-done -b catchup-03
```

Explain it

1. Imagine this bug had shipped and `station_summary` produced Table 1 of your paper (per-station mean humidity). Every station with any missing reading would report a mean that is **too low**. Explain concretely why: which direction does folding missing values in as zeros push a mean, and would you have *noticed* the numbers were wrong just by looking at the table?
2. `fillna(0)` is a real, common pandas idiom: it is genuinely correct in some places. Why did it look harmless here, and what is the one-sentence rule that tells you *when* filling missing values with zero is safe and when it silently corrupts a statistic?

If You Are Ahead

Your tests so far use tiny hand-made frames. Now aim one at the real data. Add a **sanity test** that loads and cleans the full dataset and asserts that every cleaned temperature is physically plausible — no station on Earth in this record should be below −40 °C or above +50 °C:

1. Write a test that calls `clean(load_raw("data/raw/measurements.csv"))` and asserts `df["temperature"].between(-40, 50).all()`.
2. Run it. If it is green, you have a cheap guard that would catch a future °F row slipping through unconverted (an unconverted 47.5 °F station would read as 47.5 °C, inside the range, so think about whether this bound is tight enough).
3. If it is red, do not relax the bound to make it pass: a failing sanity test on real data means the cleaning missed something. Track down which row and why.

A test like this costs one line to write and runs on every commit forever. That is the whole point of the movement you just learned: turn “I checked it once” into “the machine checks it every time.”

> **i** Note
>
> That’s it for this lab. If anything is unclear, ask now — the next session builds on this state. If you fell behind, use the checkpoint tag from the last checkpoint box to catch up.

Bibliography