Testing, CI/CD and DevOps for Health Software

Keeping automation correct as it changes: testing, version control, integration, and monitoring.

This guide accompanies Module 4, which covers the practices that keep software correct as it evolves. When code touches health data, a quiet error can affect an analysis or a decision, so correctness cannot rest on manual checking alone. The module treats automated testing, version control, continuous integration and delivery, and deployment with monitoring as a connected set of practices, and this guide grounds them in the software-engineering and reproducibility literature, with worked examples and short code.

The through-line is that discipline must be automated to be reliable. As a tutorial on practical software engineering for scientists argues, version control, testing, documentation, continuous integration, and packaging are the techniques that address the complexity, maintainability, and reproducibility problems that research software encounters as it grows [3]. Each practice in this module removes a way for evolving software to go silently wrong.

Learning outcomes

After working through this guide you will be able to:

- Explain what an automated test is and write a simple unit test.
- Use the core ideas of version control and explain why an auditable history matters in healthcare.
- Describe continuous integration and delivery and why an automatically failing build is protective.
- Deploy software in small, reversible steps and monitor it once it is live.
- Close the loop by turning each failure into a new test.
- Relate these practices to published guidance on reproducible scientific software.

1. Automated testing

1.1 What a test is

An automated test is a small piece of code that runs your code with known inputs and checks that the output matches what you expect. A unit test targets a single function, such as a unit conversion, and verifies it against cases whose answers are known. The value of such a test is that it runs identically on every change, so a mistake introduced later cannot hide.

1.2 Why testing matters more in healthcare

Testing matters more here than in many domains because automation applies any error to every record; a test is how you ensure it is correct behaviour, not incorrect behaviour, that is being multiplied. Manual checking cannot realistically cover every case as code grows, and people tire and skip things, whereas tests run every time without tiring.

1.3 A unit test in practice

A small, concrete example makes the idea tangible. A function converts a glucose value between units, and a test checks it against known answers:

```
def to_mmol(mg_dl):
    return round(mg_dl / 18.0, 1)

def test_to_mmol():
    assert to_mmol(90) == 5.0
    assert to_mmol(0) == 0.0

# run with: pytest
```

A passing unit test. If a later change breaks the conversion, this test fails immediately and names the problem, long before a wrong value could reach a dataset.

The word assert states what must be true; if it is not, the test fails, and a tool such as pytest reports pass or fail for every test. Two lines of test thus protect every future change to that function.

1.4 A failing test is good news

It may seem counter-intuitive, but a failing test should be reframed as good news: it points to the exact function that broke, appears the moment you break something, costs seconds rather than a corrupted dataset, and lets you fix the problem before it reaches real data. The alternative, finding the same mistake in a published result, is far more expensive.

1.5 What to test

You cannot test everything, so focus on where errors are both likely and costly: calculations and conversions with known answers, the validation rules that guard data quality, and edge cases such as empty inputs, zeros, and unusual but valid values. Continuous-integration research in the scientific context makes the complementary point that tests need not reproduce a full analysis to be useful; reduced tests that run a smaller version of a computation can verify that code and data still work while remaining fast enough to run often [1].

> **Key idea.** Automation multiplies behaviour. Tests are how you ensure the behaviour being multiplied is correct. A failing test is not a setback but the cheapest place to find a mistake.

2. Version control

2.1 The core ideas

Version control, provided by Git, records every change to code as a commit with a short note, builds an ordered history that can be revisited, and supports branches that let people work in parallel without overwriting one another. A merge brings a branch's changes back into the main line once they are ready. Version control and code sharing are identified in the reproducibility literature as one of the essential pillars of reproducible computational research [5].

2.2 The daily cycle

In everyday use, a short cycle repeats: edit code, commit it with an honest note, push it to the shared copy, and have a colleague review it before it merges. The habit that makes this work is small, frequent commits

with clear notes; a commit message such as fixed unit conversion in glucose function is far more useful months later than one that says changes.

2.3 Branches and parallel work

Branches are what let people work together without chaos. Each person works on their own branch, the main version always stays working because unfinished work lives on branches, experiments cannot break others' work, and finished branches merge back after review. Without branches, everyone editing the same code would constantly overwrite one another; with them, a team of any size can move in parallel and come together cleanly.

2.4 Review and the audit trail

Code review, in which a colleague checks a change before it is merged, both catches mistakes the author cannot see and produces a record of who approved what. This auditability is particularly valuable in regulated healthcare work, where being able to show what changed, when, and by whom is often required. The complete, attributed history is created as a by-product of working normally, which makes it a rare case of accountability obtained almost for free. A standing safety rule, revisited in Module 6, is never to commit passwords or patient data, because a secret placed in a repository's history is difficult to remove.

2.5 Habits that keep the history useful

A version history is only as valuable as the discipline behind it. Five habits keep it useful: commit small, related changes together rather than mixing many unrelated edits into one confusing snapshot; write a clear note for every commit; use a branch for each new piece of work; have someone review a change before it merges; and never commit passwords or patient data. That last habit connects directly to Module 6, where we see why a secret placed in a repository's history is so difficult to remove. Together these habits keep the history readable, the collaboration smooth, and sensitive information out of a place it should never be. They cost little in the moment and repay the effort every time someone, including your future self, needs to understand how the code reached its current state.

> **Key idea.** Version control keeps a complete, attributed history as a by-product of normal work. In regulated settings, that history is not merely convenient; it is often exactly the accountability record that is required.

3. Continuous integration and delivery

3.1 Definitions

Continuous integration (CI) runs checks, especially tests, automatically on every change to the code; continuous delivery (CD) extends this by keeping the software always in a state that can be released safely. The word continuous is the key: rather than testing occasionally, you test constantly, so problems are caught while small and fresh.

3.2 What the pipeline does

On every change, a CI pipeline builds the code in a clean environment, runs all the tests, and reports the result, green for pass and red for fail, within minutes and with no one remembering to run anything. In the

scientific-software context, CI serves two roles: it provides an established platform for testing the reproducibility of results, and it demonstrates to others how code and data generate the published results [1].

3.3 A small configuration

A CI configuration is often a short, readable file. The following runs the tests on every push:

```
name: tests
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pip install pytest
      - run: pytest
```

A minimal continuous-integration configuration. On every push it checks out the code, installs the test tool, and runs the tests; green means all passed.

3.4 Why a red build is protective

When a check fails and the build turns red, that is protective rather than obstructive: it stops a broken change from spreading, names the failure within minutes, and blocks the change from merging until it is fixed. In operational settings, faithful application of CI/CD promotes quality by requiring software to pass predetermined tests before it is promoted from one stage to the next, with governance gateways for the checks that cannot be automated [2].

3.5 Delivery in small, reversible steps

Delivery is safest when releases are small, frequent, and reversible, because a small change that goes wrong is easy to identify and undo. A healthy setup runs on every change, blocks merges on failure, finishes quickly, reports failures clearly, and keeps releases reversible.

> **Key idea.** CI lets a machine enforce the discipline a person would eventually skip. A red build is the safety net doing its job: it blocks broken code before it spreads.

4. Deployment and monitoring

4.1 What deployment is

Deployment moves software from a development machine into the environment where it will run and be used. The differences between these environments are a common source of surprises, which is why deployment deserves care rather than being treated as a final formality.

4.2 Deploying safely

Safe deployment proceeds in small, reversible steps, with a clear way to roll back; if a change cannot be undone quickly, it is not ready to release. Deploy at a low-risk time when you can, and confirm the software works after deploying before trusting it. The golden rule is simple: always know how to undo a deployment.

4.3 Monitoring the essentials

Going live is a beginning rather than an end, because software can fail in use for reasons no test anticipated, when data, load, or environment differ from expectations. Monitoring is the continuous observation of running software, with prompt alerting when something goes wrong. For an automated pipeline, four questions capture most of what matters: did it run, did it succeed, how much did it process, and how long did it take. A night that processes ten records instead of the usual thousand signals a problem even if nothing technically failed.

4.4 Alerts that reach a person

Monitoring only helps if a problem reaches a person, so an alert should fire the moment a run fails, say what failed and when, and go where someone will see it; one clear alert is better than many noisy ones. The goal is that the operator, not the user, is the first to know of a failure.

4.5 Closing the loop

The greatest value of monitoring appears when the loop is closed: a detected failure is fixed, a test is added so that it cannot recur unnoticed, and the fix is redeployed. In this way each incident becomes a new test, and the system grows more reliable over time, an outcome the reproducibility and software-quality literature frames as the goal of sustainable, maintainable software [2], [3].

> **Key idea.** Deploy small and reversible; monitor whether the software ran, succeeded, and behaved as expected; and turn every failure into a new test so the same problem cannot recur silently.

5. Case studies and worked discussion

The practices of this module are widely discussed in the scientific-software literature, and a few sources repay closer attention because they connect the practices to reproducibility, which is the deeper goal.

5.1 Continuous integration for reproducible results

Krafczyk and colleagues examine what it takes to reproduce published computational results and argue that continuous integration can serve two distinct scientific functions: providing an established platform for testing the reproducibility of results, and demonstrating to other scientists how code and data generate those results [1]. A practical insight from their work is that full reproduction can be too slow or resource-intensive to run routinely, so they recommend including reduced versions of an analysis, run at lower resolution or on smaller data, as fast tests that still verify the code and data are working. This idea, the reduced test, is directly useful for health-data pipelines, where a full run may be expensive but a small synthetic sample can exercise the same logic on every change.

5.2 CI/CD in an operational setting

Schubert and colleagues describe the transition of scientific modelling software to continuous integration and delivery in a production environment, where everyday challenges such as missing data, unintended inputs, and values outside previously tested bounds expose gaps in software quality [2]. Their account is candid about the difficulty of the transition and stresses the role of governance gateways for the checks that cannot be automated. The lesson for healthcare automation is that CI/CD is not only a technical

convenience but a quality-assurance discipline: software is promoted from one stage to the next only after passing predetermined checks, which is exactly the assurance that regulated environments demand.

5.3 Engineering habits for scientists

Peláez offers a tutorial aimed at scientists who write software without formal engineering training, gathering version control, testing, documentation, continuous integration, and packaging into a coherent set of habits and reflecting on how these address complexity, maintainability, and reproducibility as projects grow [3]. The value of such a synthesis is that it presents the practices of this module not as isolated techniques but as a mutually reinforcing system: tests make change safe, version control records it, continuous integration enforces it, and documentation makes it usable by others.

> **Key idea.** Testing, version control, and continuous integration are not separate chores but a single system whose purpose is reproducible, trustworthy software, the same goal that runs through Module 6.

6. Further reading and practice

The practices in this module reinforce one another and are best learned by applying them to a small project rather than in isolation. The exercises below are designed to take an afternoon and to produce material you can reuse in the module assignment, which asks you to write tests for a small program and set up a continuous-integration pipeline that runs them.

For deeper reading, the scientific continuous-integration study [1] shows how reduced tests can verify reproducibility affordably, the software-quality account [2] describes CI/CD in an operational setting with governance gateways, and the practical software-engineering tutorial [3] gathers the whole set of habits for research software. The reproducibility references [4], [5] connect these engineering practices to the wider goal of trustworthy computational work.

- Take a small function of your own and write two unit tests for it, including one edge case.
- Put the code under version control and make three small, well-described commits.
- Add a minimal continuous-integration configuration that runs your tests on every change.
- Deliberately break the function and confirm that the build turns red and names the failure.
- Write a one-paragraph plan describing how you would deploy the code reversibly and what you would monitor.

Completing these exercises will leave you with a tested, version-controlled, continuously integrated project and a deployment plan, which together demonstrate the whole discipline of the module.

Key terms

Automated test — A small piece of code that runs your code with known inputs and checks the output is as expected.

Unit test — A test that targets a single function against cases whose answers are known.

Version control — A system, such as Git, that records every change as a commit and keeps an ordered, attributable history.

Branch — A separate line of work off the main version, allowing parallel work without collisions.

Continuous integration (CI) — Automatically running checks, especially tests, on every change to the code.

Continuous delivery (CD) — Keeping software always in a releasable state, favouring small, reversible releases.

Rollback — Undoing a deployment to return to a previous working version.

Monitoring — Continuous observation of running software with prompt alerting when it fails.

Self-check questions

1. Write, in pseudocode, a unit test for a function that converts a glucose value between two units. What cases would you include?
2. Explain why an error in automated code is more dangerous than the same error made once by hand.
3. Describe the daily Git cycle and explain why small, well-described commits matter.
4. Explain why the Git history is described as an 'asset' in regulated healthcare work, and give one rule about what must never be committed.
5. Why is an automatically failing build described as protective? What does it prevent?
6. State the golden rule of safe deployment and explain why small, reversible releases are safer than large ones.
7. List the four monitoring questions for a pipeline and give an example where 'how much' reveals a hidden problem.
8. Describe how monitoring 'closes the loop' and why turning failures into tests makes a system more reliable over time.

References

Peer-reviewed sources located via the Consensus academic search tool; standards cited from their issuing bodies. Formatted in IEEE style.

[1] M. S. Krafczyk, A. Shi, A. Bhaskar, D. Marinov, and V. Stodden, "Scientific tests and continuous integration strategies to enhance reproducibility in the scientific software context," in Proc. 2nd Int. Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS), 2019. [Online]. Available: https://consensus.app/papers/details/d98713670d375337a7608154160a0c44/

[2] A. Schubert et al., "Promoting scientific software quality through transition to continuous integration and continuous delivery," Socio-Environmental Systems Modelling, 2024. [Online]. Available: https://consensus.app/papers/details/e00e2a5cea3e53cfa7f17b3239c26686/

[3] R. P. Peláez, "Practical software engineering for software-writing scientists," The Journal of Chemical Physics, 2025. [Online]. Available: https://consensus.app/papers/details/f97a0be33de452c9819b4320969fde4a/

[4] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," PLoS Computational Biology, vol. 9, no. 10, 2013. [Online]. Available: https://consensus.app/papers/details/844402aaa5105a5280cdc99952699243/

[5] M. Ziemann, P. Poulain, and A. Bora, "The five pillars of computational reproducibility: bioinformatics and beyond," Briefings in Bioinformatics, 2023. [Online]. Available: https://consensus.app/papers/details/09aac82bfb2550a7a72f0fd387a4fb91/