Risky Business:
Model Testing and Development

May 21, 2024

Carr Actuarial: Kevin Carr, FSA, MAAA
Martello Re: Matt Heaphy, FSA, MAAA

PRESENTATION DISCLAIMER

Presentations are intended for educational purposes only and do not replace independent professional judgment. Statements of fact and opinions expressed are those of the speakers individually and, unless expressly stated to the contrary, are not the opinion or position of the Society of Actuaries, the American Academy of Actuaries, the Actuarial Club of Hartford & Springfield, or the speakers' respective employers.

SPEAKERS

Kevin Carr, FSA, MAAA

Founder & Principal at Carr Actuarial

Expertise: actuarial modeling including (re)building, testing, and ongoing management, universal life pricing and product development, experience studies, and reinsurance

Education: Master of Science in Mathematics from Northeastern University

Software: AXIS™, MG-ALFA, TAS

Languages: VBA, Python, R, SQL, C++

Matt Heaphy, FSA, MAAA

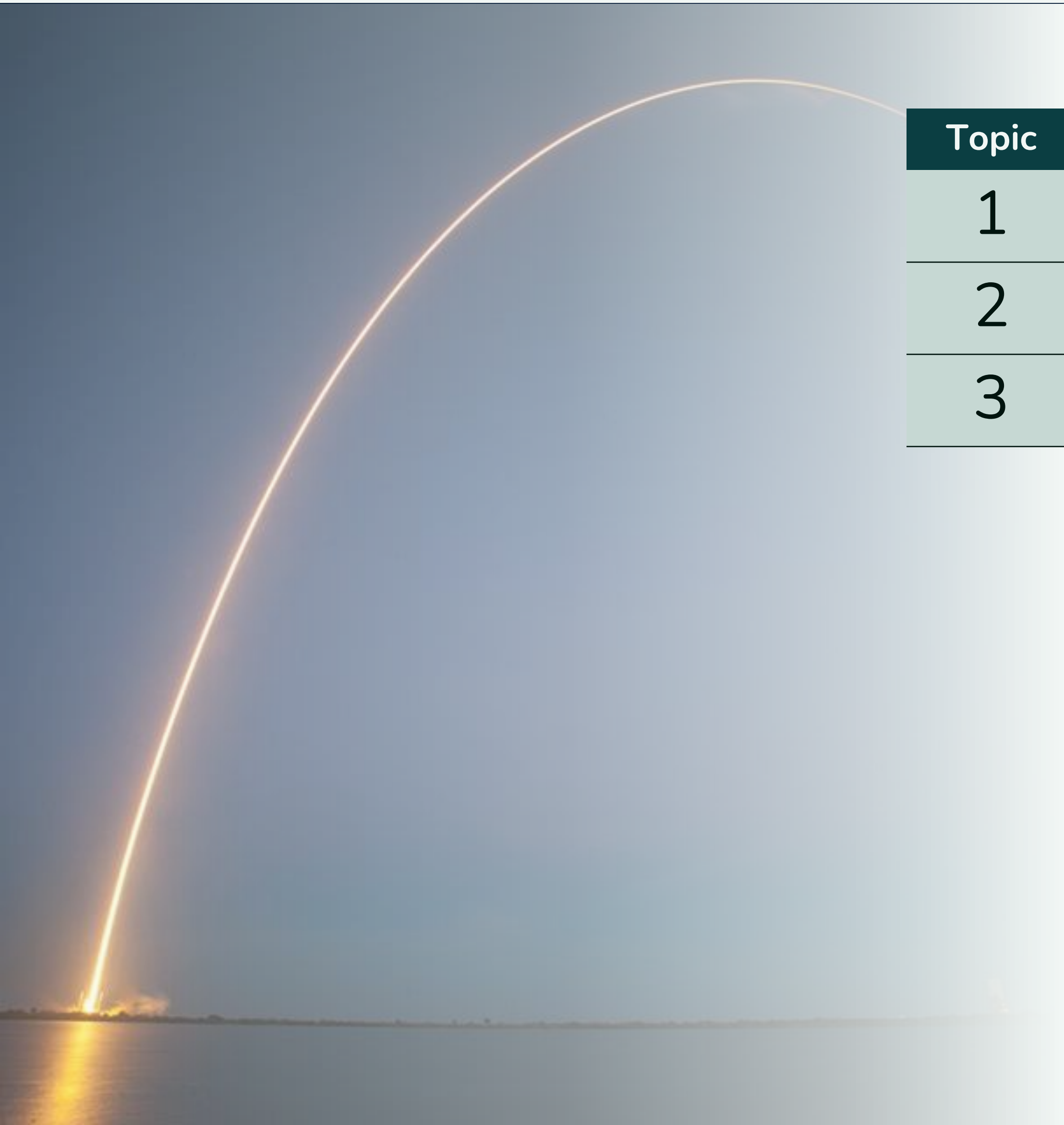
Head of Pricing & Risk Analytics at Martello Re

Expertise: data science, experience studies, annuity pricing and product development, actuarial modeling, hedging, and reinsurance.

Education: Bachelor of Science degree in Actuarial Science from the University of Connecticut.

Open-source contributions: actxps (an actuarial experience study toolkit for R and Python), offsetreg (a predictive modeling R package for offset terms).

AGENDA PART 1



Topic	Title
1	ASOP 56 & MDLC
2	Testing & pitfalls
3	Model testing/review recommendations

1

ASOP 56 & MDLC

ASOP 56: A BRIEF REVIEW

ASOP 56 can be interpreted to apply whenever a model is involved

ASOP 56
Modeling

Purpose [Section 1.1]
"This actuarial standard of practice (ASOP or standard) provides guidance to actuaries when performing actuarial services with respect to designing, developing, selecting, modifying, using, reviewing, or evaluating models."

Scope [Section 1.2]
All practice areas when performing actuarial services to the extent of the services provided

Definitions [Section 2]
Thirteen definitions covering items like assumption, data, input, model, model run, and output

Analysis of Issues and Recommended Practices [Section 3]
Main body covering model meeting intended purpose, understanding the model, reliance on others, mitigation of model risk, and documentation

Communications & Disclosures [Section 4]
Required disclosures, reference to ASOPs 23 & 41, recommended additional disclosures

ASOP 56: DEFINITION OF A MODEL

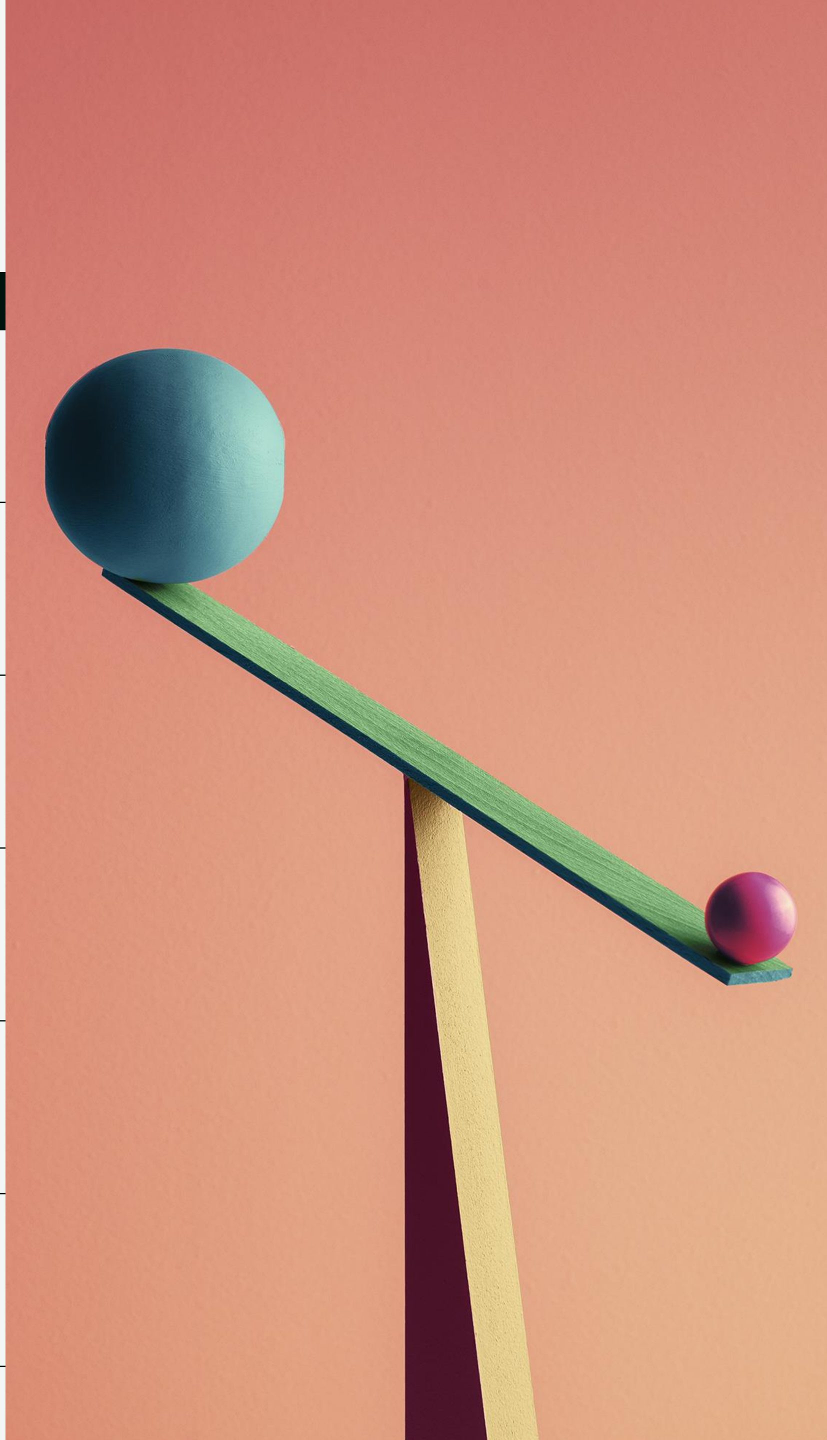
2.8 MODEL

A simplified representation of relationships among real world variables, entities, or events using statistical, financial, economic, mathematical, non-quantitative, or scientific concepts and equations.
A **model** consists of three components: an information **input** component, which delivers **data** and **assumptions** to the **model**; a processing component, which transforms **input** into **output**; and a results component, which translates the **output** into useful business information.

ASOP 56: MITIGATING MODEL RISK

"Actuary should evaluate model risk and … take reasonable steps to mitigate"

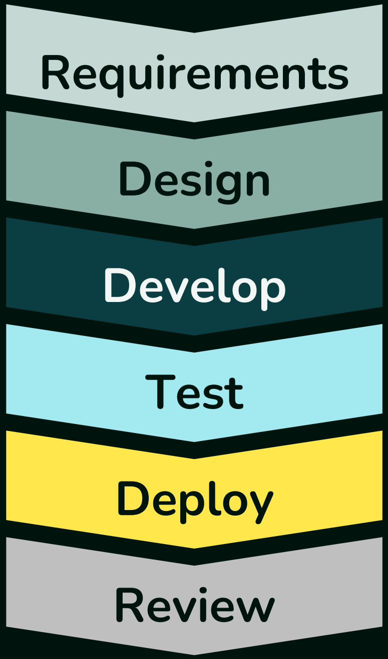
Section	Key points
3.6.1 Model testing	Certain **testing activities of model** may be reasonable including input validation, formula checking, sensitivity testing, and output reconciliation
3.6.2 Model output validation	Activities **focused on output** including A/E analysis, implications of different hold-out periods for predictive models, assumption change implications, and alternative model output comparisons
3.6.3 Review by another professional	Having another **qualified** professional review may be appropriate
3.6.4 Reasonable governance & controls	**Governance and controls** do not need to be directly tied to actuary utilizing model or output
3.6.5 Mitigating misuse and misinterpretation	**ASOP 41** provides additional guidance
3.7 Documentation	Document. Document. Document.



MODEL DEVELOPMENT LIFE CYCLE

Two common approaches in actuarial projects are Waterfall and Agile

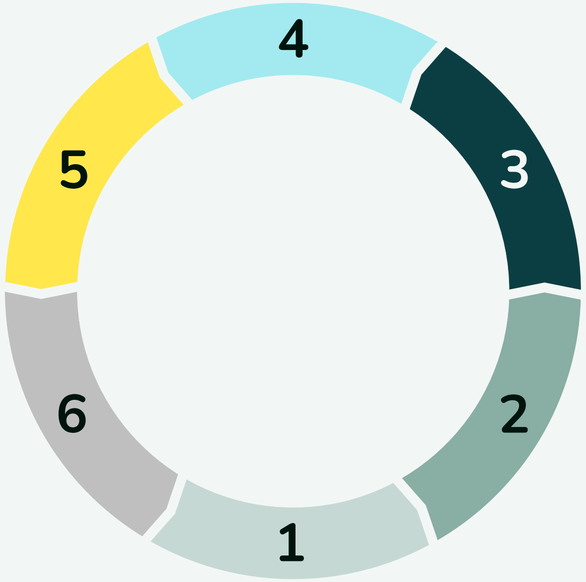
Waterfall



- Big bang approach
- Full requirements up front
- Less flexible than Agile
- **Once** through entire sequence

VS

Agile



- Incremental approach
- Less initial planning
- Less rigid than Waterfall
- **Numerous** cycles over the course of the project

2

TESTING & PITFALLS

COMMON TESTING ACTIVITIES & PITFALLS

Examples of highlighted items later in presentation

Testing Activities

Dynamic validation
Static validation
Input validation
Calculation validation
Single cell/policy testing
Regression testing
Sensitivity testing
Implied rate analysis
Attribution analysis
Comparison to alternative models
Review by another professional

Pitfalls

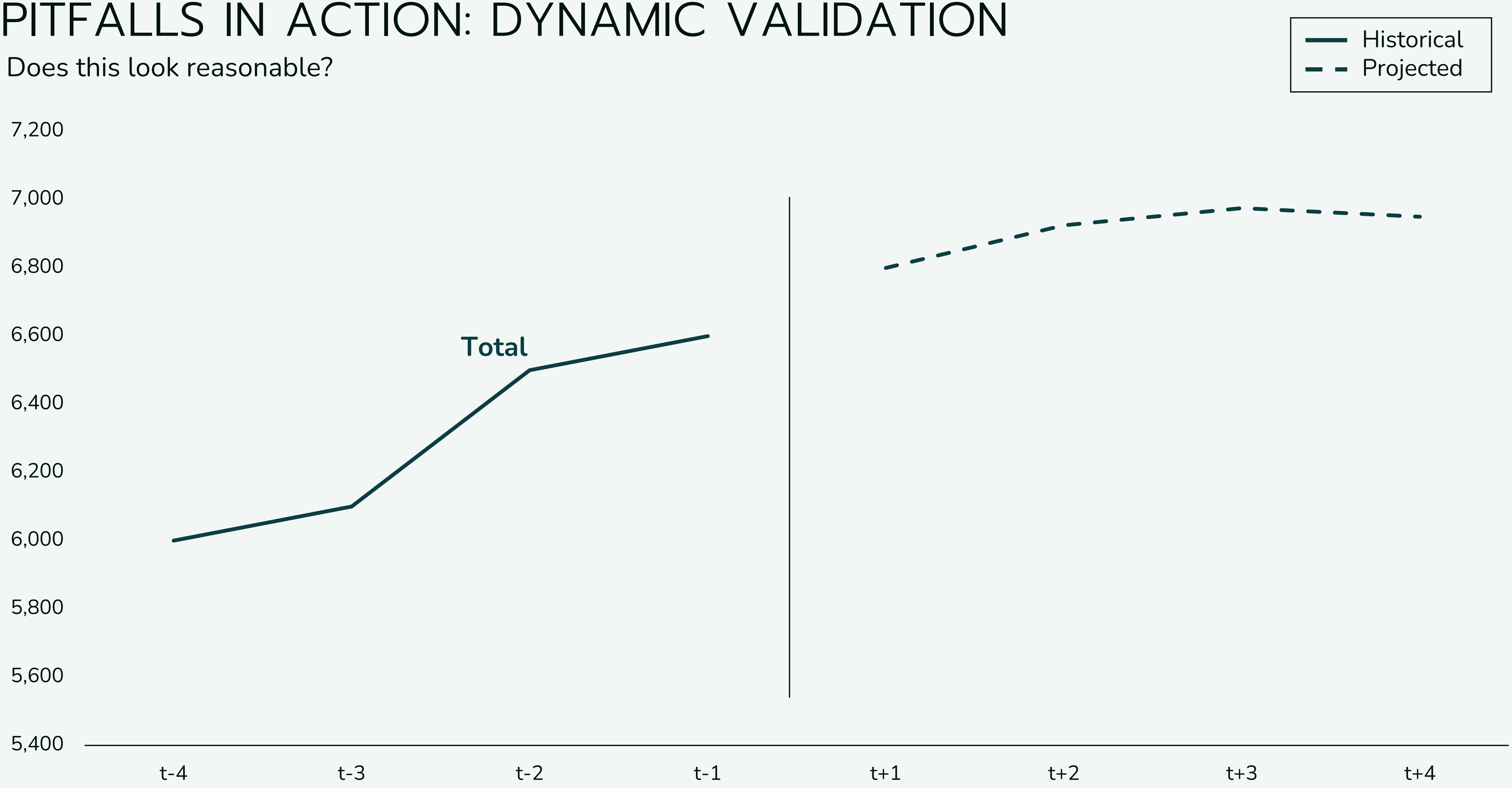
Insufficient granularity
Not comprehensive
Lack of documentation
Invalid source of truth
Inconsistent execution/application
Lack of automation

And the biggest...

Not regularly testing your model

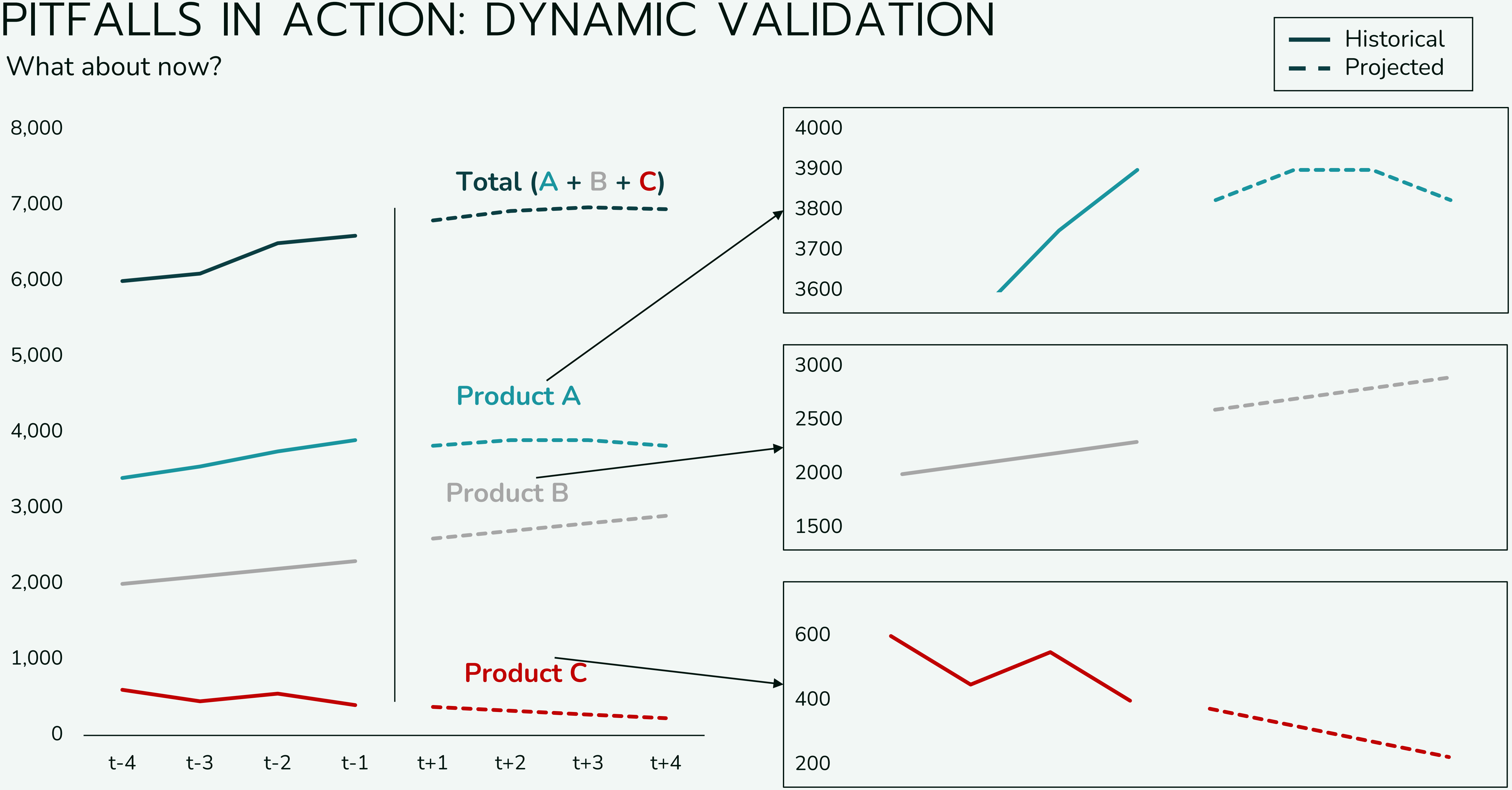
PITFALLS IN ACTION: DYNAMIC VALIDATION

Does this look reasonable?



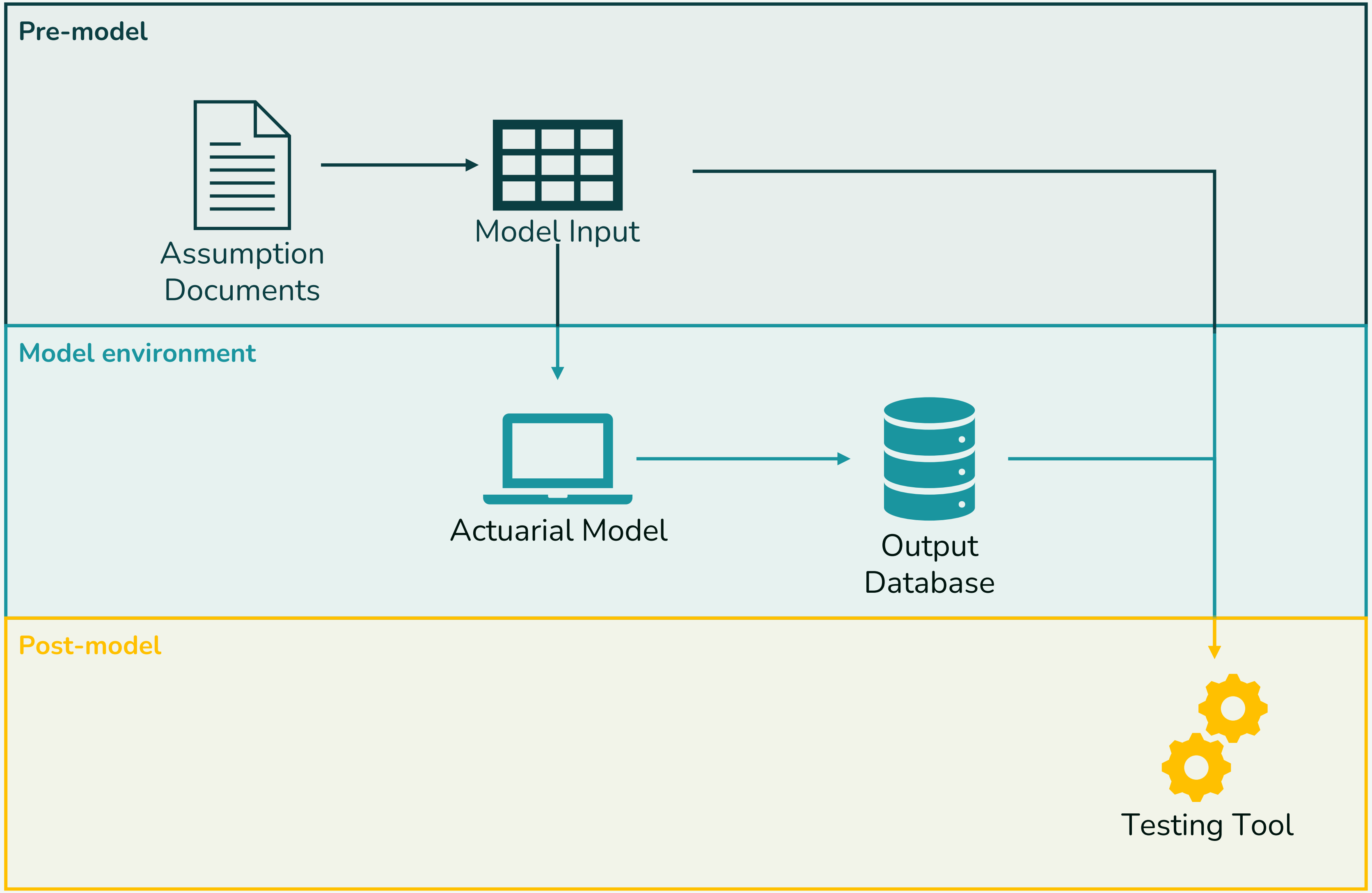
PITFALLS IN ACTION: DYNAMIC VALIDATION

What about now?



PITFALLS IN ACTION: SINGLE CELL TESTING

A common approach is to utilize inputs structured for models in downstream testing tools



What sort of issues can arise in this workflow?

In general?

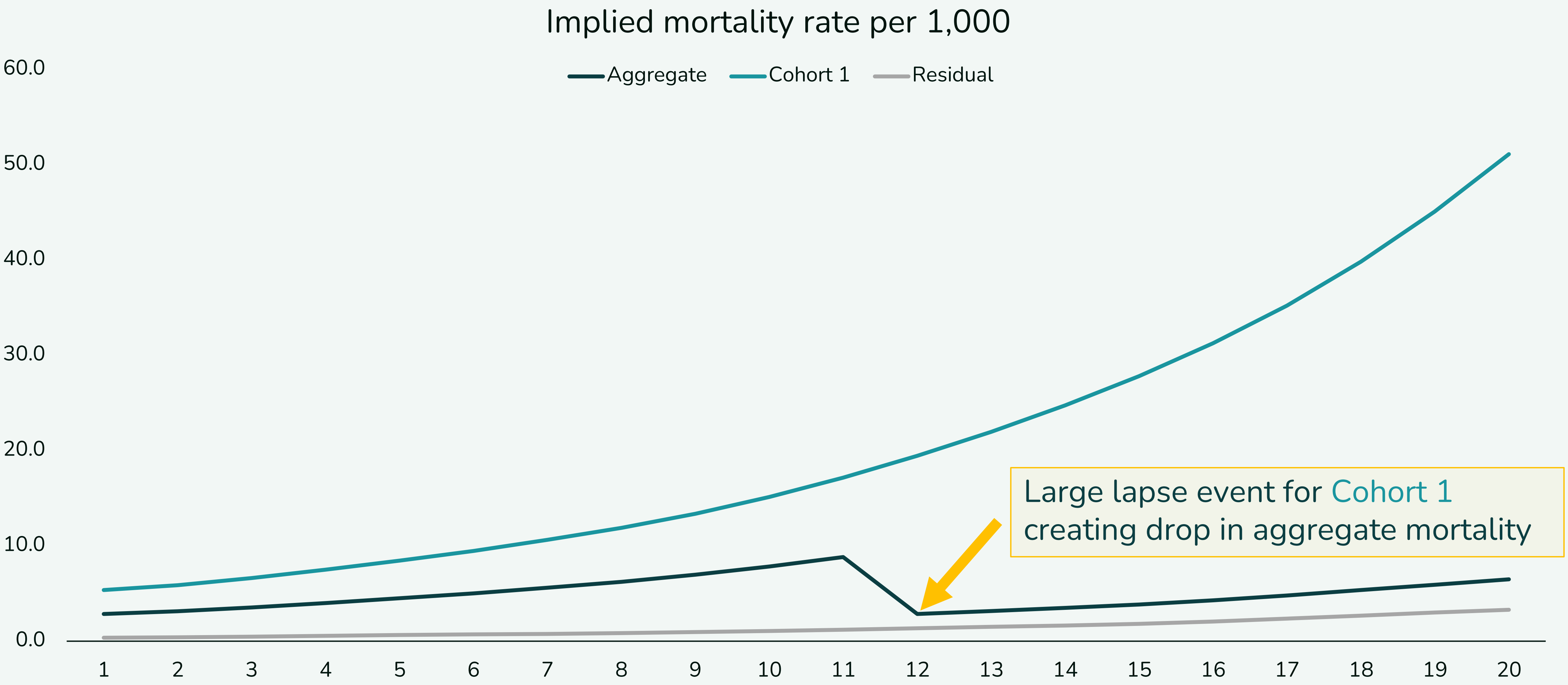
How can we protect against them?

PITFALLS IN ACTION: IMPLIED RATE ANALYSIS (1/2)

What is happening between durations 11 and 12? Is something wrong?

PITFALLS IN ACTION: IMPLIED RATE ANALYSIS (2/2)

A large lapse event for an older age cohort is creating the aggregate distortion – no actual problem!



Large lapse event for Cohort 1 creating drop in aggregate mortality

4

MODEL TESTING/REVIEW RECOMMENDATIONS

MODEL TESTING/REVIEW RECOMMENDATIONS

Review documentation
- Know where the model came from
- Learn the limitations

Waterfall changes
- Estimate impact ahead of time if possible
- Review incremental updates for reasonability

Have a required **testing/review framework**
- Static validation
- Input validation on new inputs
- Calculation validation on new functionality
- Single cell testing
- Model regression testing
- Review of items changed in model

Martello Re

Open Source Model Considerations

In-house actuarial models built with open source software are increasingly common.

What best practices can we learn from software developers?

Agenda

- 00 What do we mean by "open source"?
- 01 Unit testing
- 02 Version control
- 03 Dependency management

What do we mean by “open source”?

True Open Source

Free to anyone

Complete access to source code

Communal

“As-is” / use at your own risk

Open Systems

Proprietary or vendor-provided

Authorized users have complete access to source code

Can be built with open source tools

Requires model governance and controls

Built with open source

Company A's Python Project

Company B's R Project

Company C's Julia Project

Vendor "open" systems

MG-ALFA

FIS Prophet

Open source ≠ open systems built with open source tools

But for the purposes of this talk, we're going to use "open source" as shorthand

Unit Testing

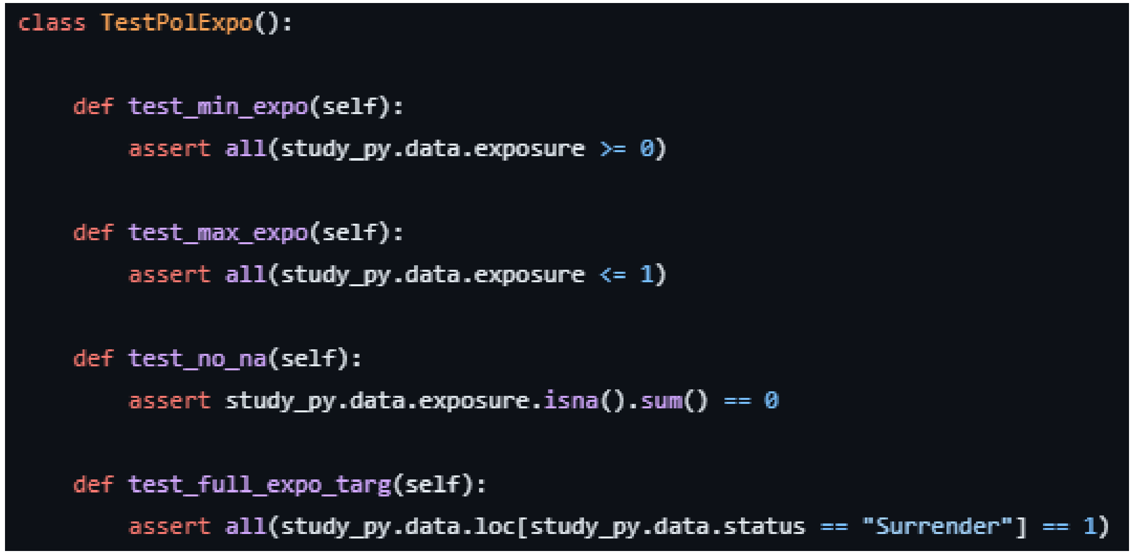
What are Unit Tests?

Automated tests to verify that individual components of a program are working correctly

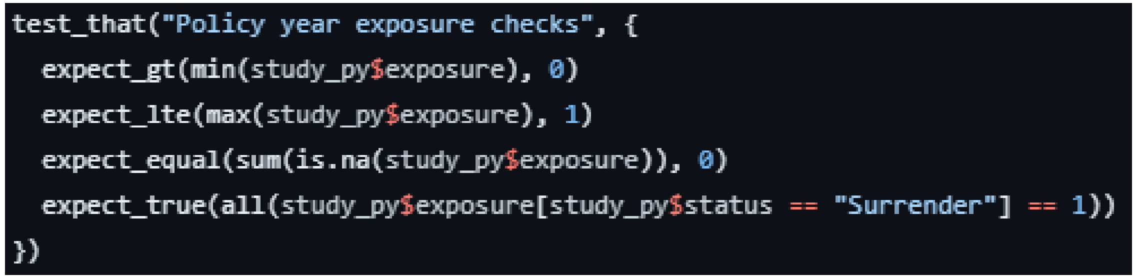
- Calculation tests *Does my PV function return the correct value?*
- Sanity checks *Are mortality rates between 0 and 1?*
- Error handling tests *Is an error returned if I pass text to a numeric input?*
- Regression tests *Does my model return the same results as before?*

Unit tests are the first line of defense for catching bugs and verifying that future changes haven't broken anything

Popular Unit Testing Frameworks



```python
class TestPolExpo():

    def test_min_expo(self):
        assert all(study_py.data.exposure >= 0)

    def test_max_expo(self):
        assert all(study_py.data.exposure <= 1)

    def test_no_na(self):
        assert study_py.data.exposure.isna().sum() == 0

    def test_full_expo_targ(self):
        assert all(study_py.data.loc[study_py.data.status == "Surrender"] == 1)
```



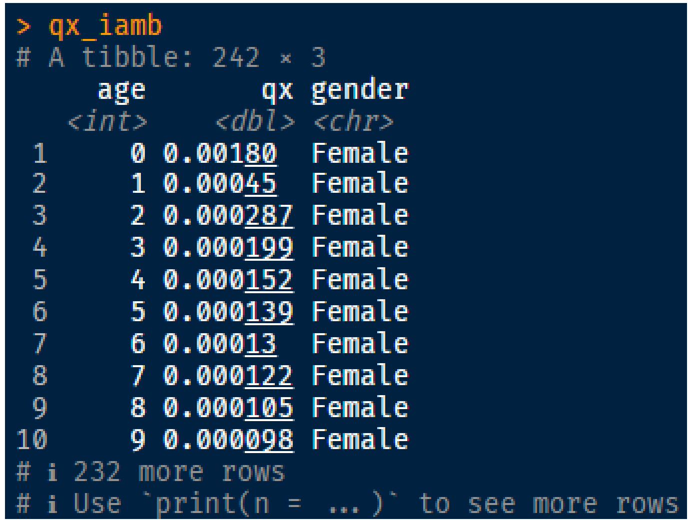
```r
test_that("Policy year exposure checks", {
  expect_gt(min(study_py$exposure), 0)
  expect_lte(max(study_py$exposure), 1)
  expect_equal(sum(is.na(study_py$exposure)), 0)
  expect_true(all(study_py$exposure[study_py$status == "Surrender"] == 1))
})
```

R Case Study

Assume we need a function that calculates life annuity present values ($\ddot{a}_x$)

Function design

- Inputs for age, gender, and a discount rate
- Annual payments at the beginning of each projection year
- Mortality = 2012 IAM Basic



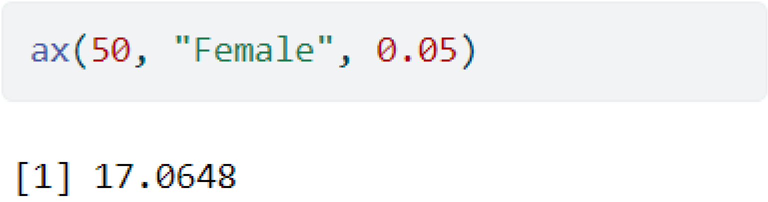
```
> qx_iamb
# A tibble: 242 × 3
     age       qx gender
   <int>    <dbl> <chr> 
 1     0 0.00180  Female
 2     1 0.00045  Female
 3     2 0.000287 Female
 4     3 0.000199 Female
 5     4 0.000152 Female
 6     5 0.000139 Female
 7     6 0.00013  Female
 8     7 0.000122 Female
 9     8 0.000105 Female
10     9 0.000098 Female
# ℹ 232 more rows
# ℹ Use `print(n = ...)` to see more rows
```

annuity_calc.R

```r
library(tidyverse)
library(actxps)

ax <- function(iss_age, i_gender, disc_rate) {

  qx <- qx_iamb |>
    filter(gender == i_gender, age >= iss_age) |>
    pull(qx)
  tpx <- cumprod(1 - qx)
  vt <- (1 + disc_rate) ^ -seq_along(tpx)

  1 + sum(tpx * vt)

}
```

Informal testing

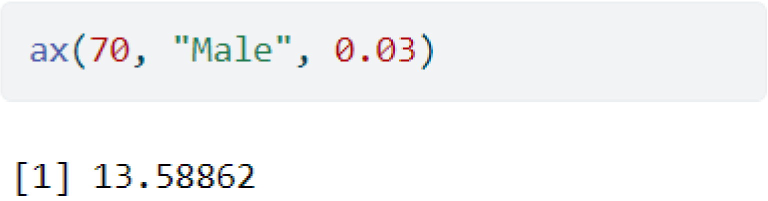
Reasonable results are returned



```
ax(50, "Female", 0.05)
```

```
[1] 17.0648
```

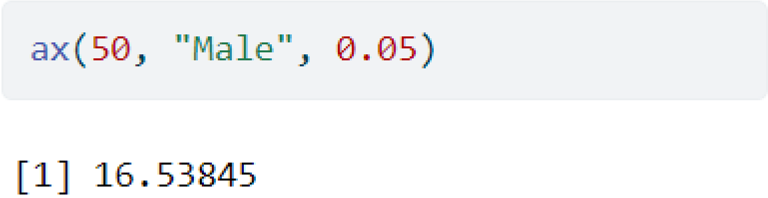
Higher mortality at older ages



```
ax(70, "Male", 0.03)
```

```
[1] 13.58862
```

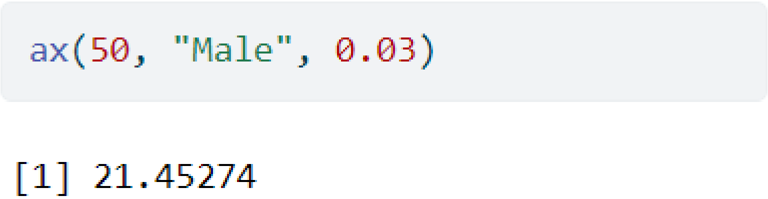
Higher mortality for males



```
ax(50, "Male", 0.05)
```

```
[1] 16.53845
```

Lower discount rates, higher present values



```
ax(50, "Male", 0.03)
```

```
[1] 21.45274
```

Introducing the `testthat` package

`testthat` provides functions for writing and running tests

Testing Workflow

- Create a `tests/` directory
- Save scripts containing unit tests into `tests/`
- Run tests
  - One file at a time: `test_file("tests/test-{name}.R")`
  - An entire directory: `test_dir("tests")`
  - When developing an R package: `devtools::test()`

Unit Test Structure

Start with `test_that()` and a simple description

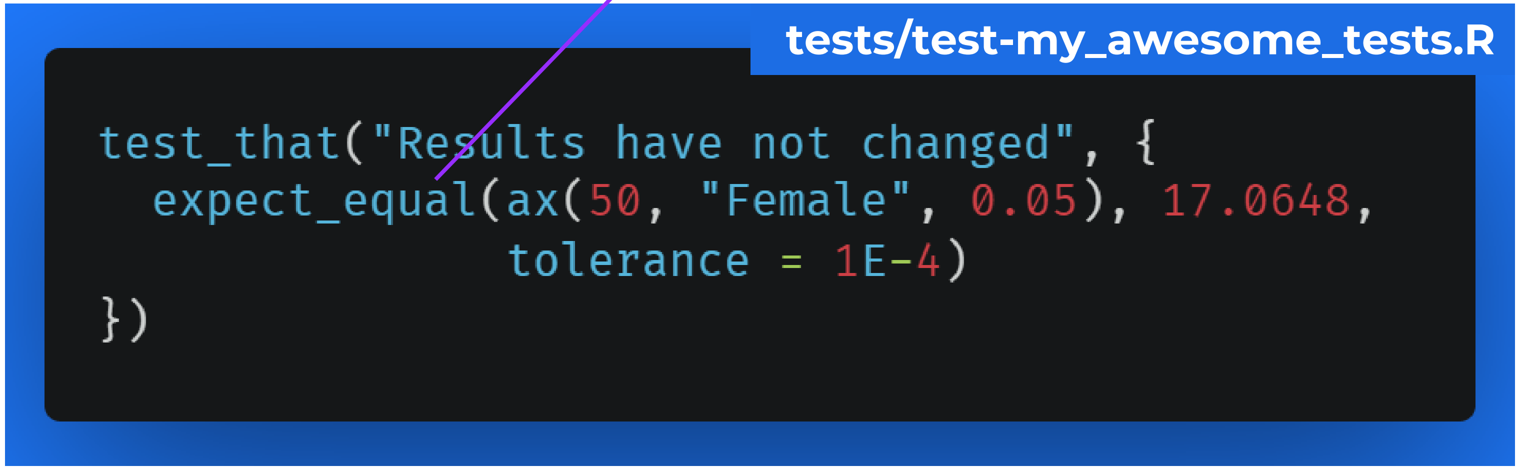
```
test_that("{Description goes here}", {
  expect_*( ... )
  expect_*( ... )
   ...
})
```

One or more calls to functions beginning with `expect_*()`

Annuity Factor Unit Tests 1/2

A regression test

`expect_equal():` the first two arguments must be equal within a specified degree of tolerance



tests/test-my_awesome_tests.R

```r
test_that("Results have not changed", {
  expect_equal(ax(50, "Female", 0.05), 17.0648,
                     tolerance = 1E-4)
})
```

Annuity Factor Unit Tests 2/2

Verify that mortality is higher at older ages

`expect_gt():` The first argument must be greater than the second argument

`expect_lt():` The first argument must be less than the second argument

`expect_true():` The argument must result in `True`.

More robust: test all ages from 30 to 90

tests/test-my_awesome_tests.R

```
test_that("ax's decrease with age", {

  expect_gt(
    ax(50, "Female", 0.05),
    ax(51, "Female", 0.05),
  )

  expect_lt(
    ax(66, "Male", 0.05),
    ax(65, "Male", 0.05),
  )

  ages ← 30:90
  n ← length(ages)
  ax_vec ← map_dbl(ages,
                   \(x) ax(x, "Female", 0.05))
  expect_true(
    all(ax_vec[-n] > ax_vec[-1])
  )

})
```

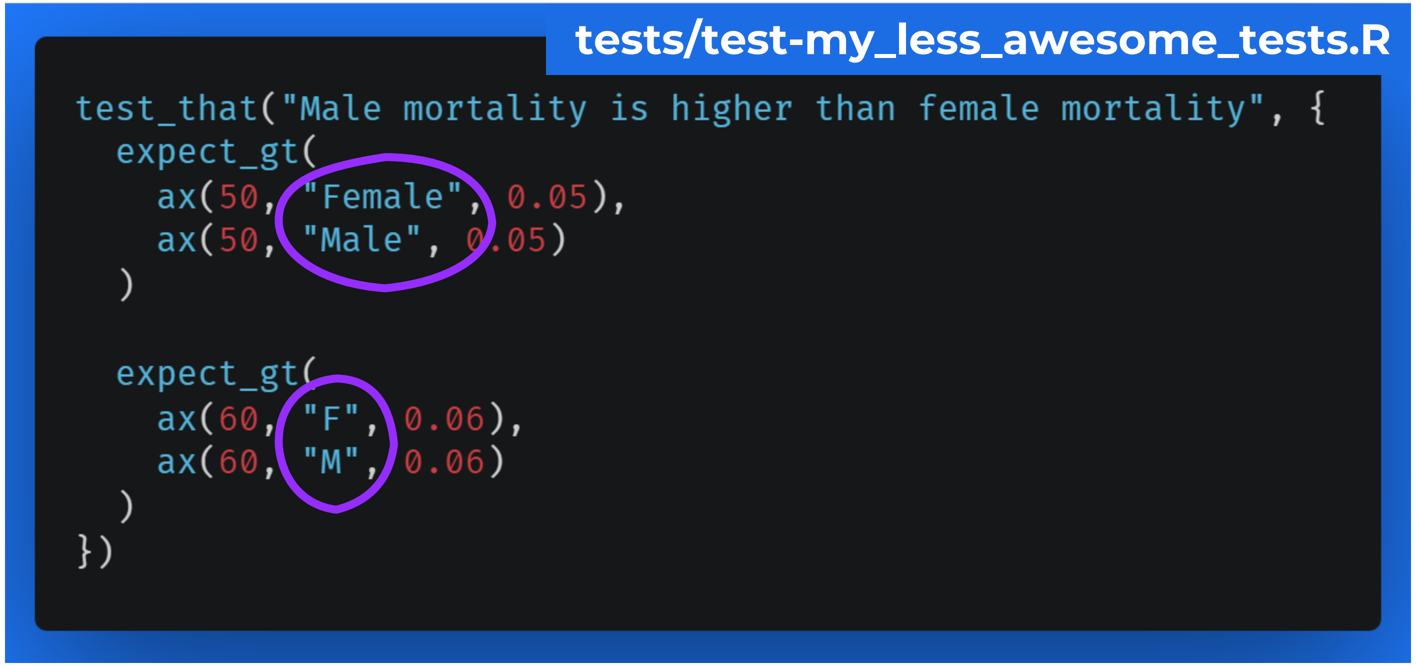
Running tests in a single file

```
library(testthat)
source("annuity_calc.R")
test_file("tests/test-my_awesome_tests.R")
```

```
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 6 ]
```

A test doomed to fail

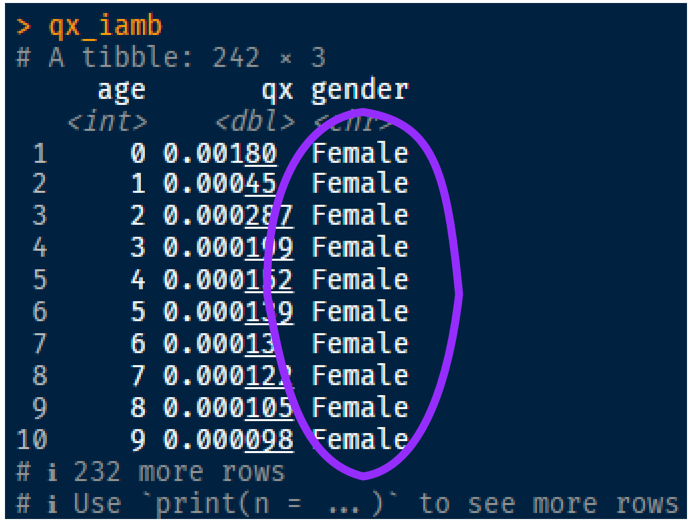
Can you spot the problem?



tests/test-my_less_awesome_tests.R

```r
test_that("Male mortality is higher than female mortality", {
  expect_gt(
    ax(50, "Female", 0.05),
    ax(50, "Male", 0.05)
  )

  expect_gt(
    ax(60, "F", 0.06),
    ax(60, "M", 0.06)
  )
})
```

Hint:



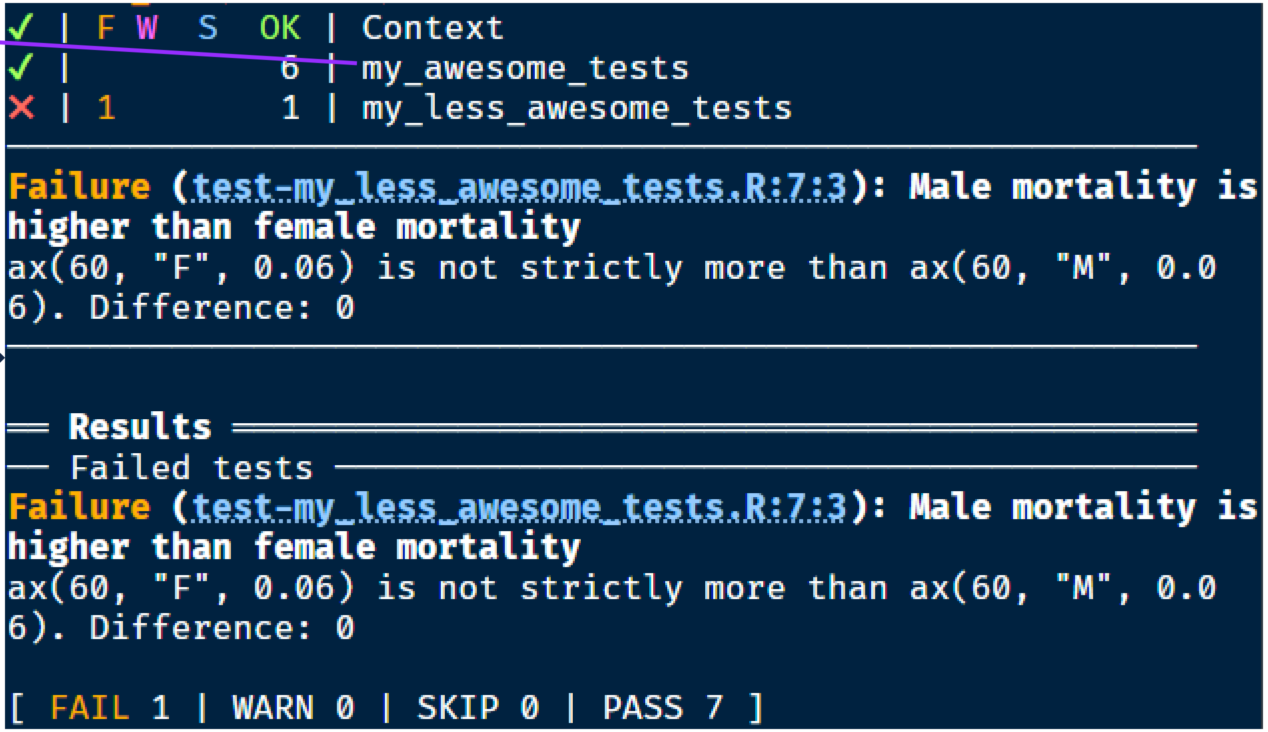
```
> qx_iamb
# A tibble: 242 × 3
      age       qx gender
    <int>    <dbl> <chr>
 1      0 0.00180  Female
 2      1 0.00045  Female
 3      2 0.000287 Female
 4      3 0.000199 Female
 5      4 0.000152 Female
 6      5 0.000139 Female
 7      6 0.00013  Female
 8      7 0.000122 Female
 9      8 0.000105 Female
10      9 0.000098 Female
# ℹ 232 more rows
# ℹ Use `print(n = ...)` to see more rows
```

Testing failures

Testing a directory

`my_awesome_tests` is still passing

```
test_dir("tests")
```



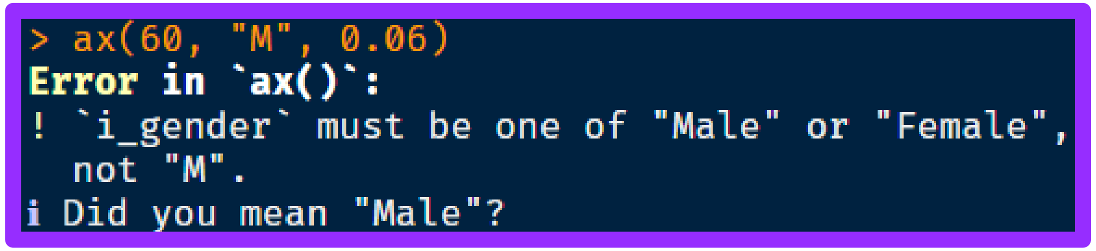
```
✓ | F W  S  OK | Context
✓ |          6 | my_awesome_tests
✗ | 1        1 | my_less_awesome_tests
──────────────────────────────────────────────────────────
Failure (test-my_less_awesome_tests.R:7:3): Male mortality is
higher than female mortality
ax(60, "F", 0.06) is not strictly more than ax(60, "M", 0.0
6). Difference: 0
──────────────────────────────────────────────────────────


══ Results ═══════════════════════════════════════════════
── Failed tests ──────────────────────────────────────────
Failure (test-my_less_awesome_tests.R:7:3): Male mortality is
higher than female mortality
ax(60, "F", 0.06) is not strictly more than ax(60, "M", 0.0
6). Difference: 0

[ FAIL 1 | WARN 0 | SKIP 0 | PASS 7 ]
```

One failure in `my_less_awesome_tests` because “M” and “F” were passed instead of “Male” and “Female”

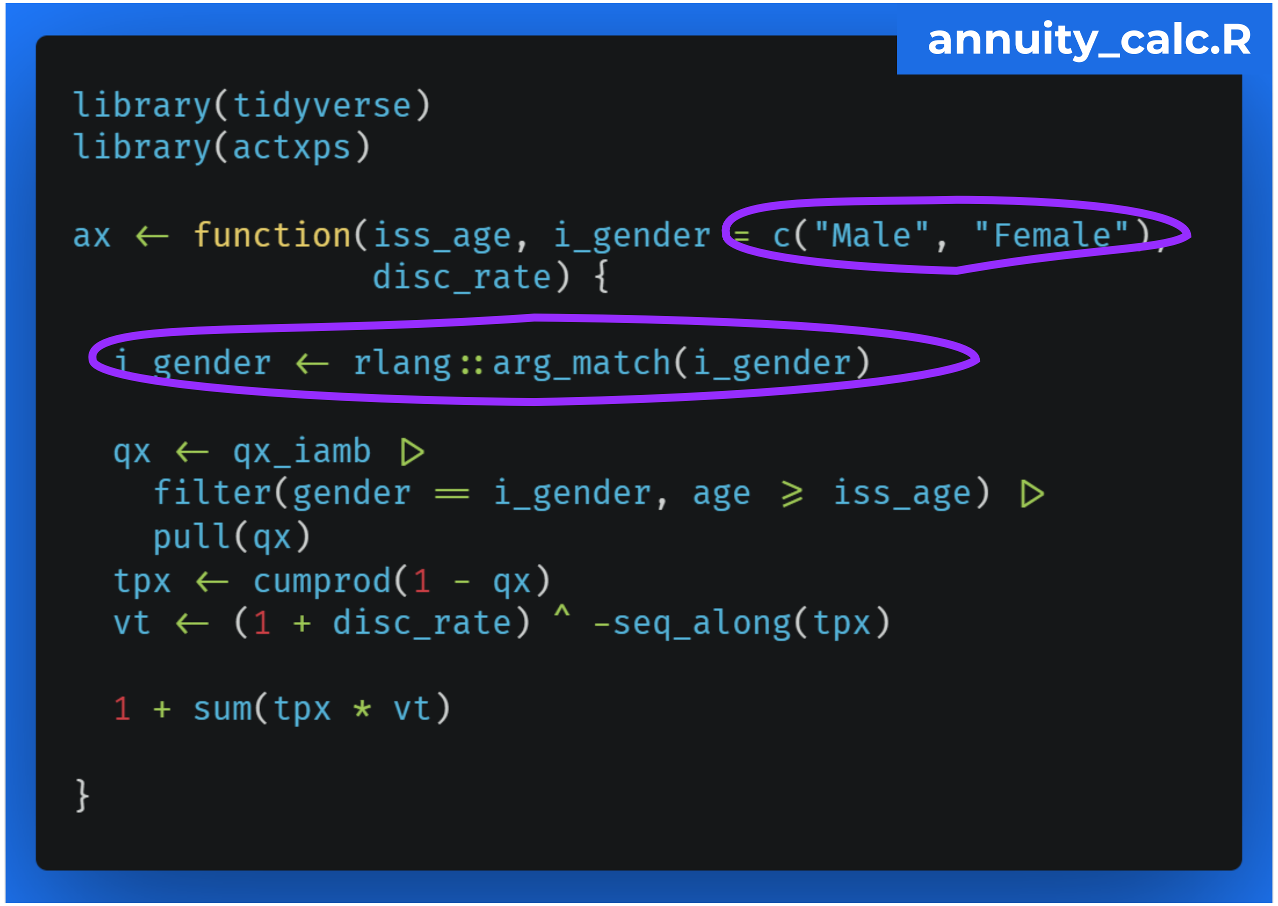
Dealing with failure 1/3

Write more robust functions to automatically catch bad inputs and return informative error messages

This change will send an informative error message if `i_gender` is anything other than “Male” or “Female”



```
> ax(60, "M", 0.06)
Error in `ax()`:
! `i_gender` must be one of "Male" or "Female",
  not "M".
ℹ Did you mean "Male"?
```



annuity_calc.R

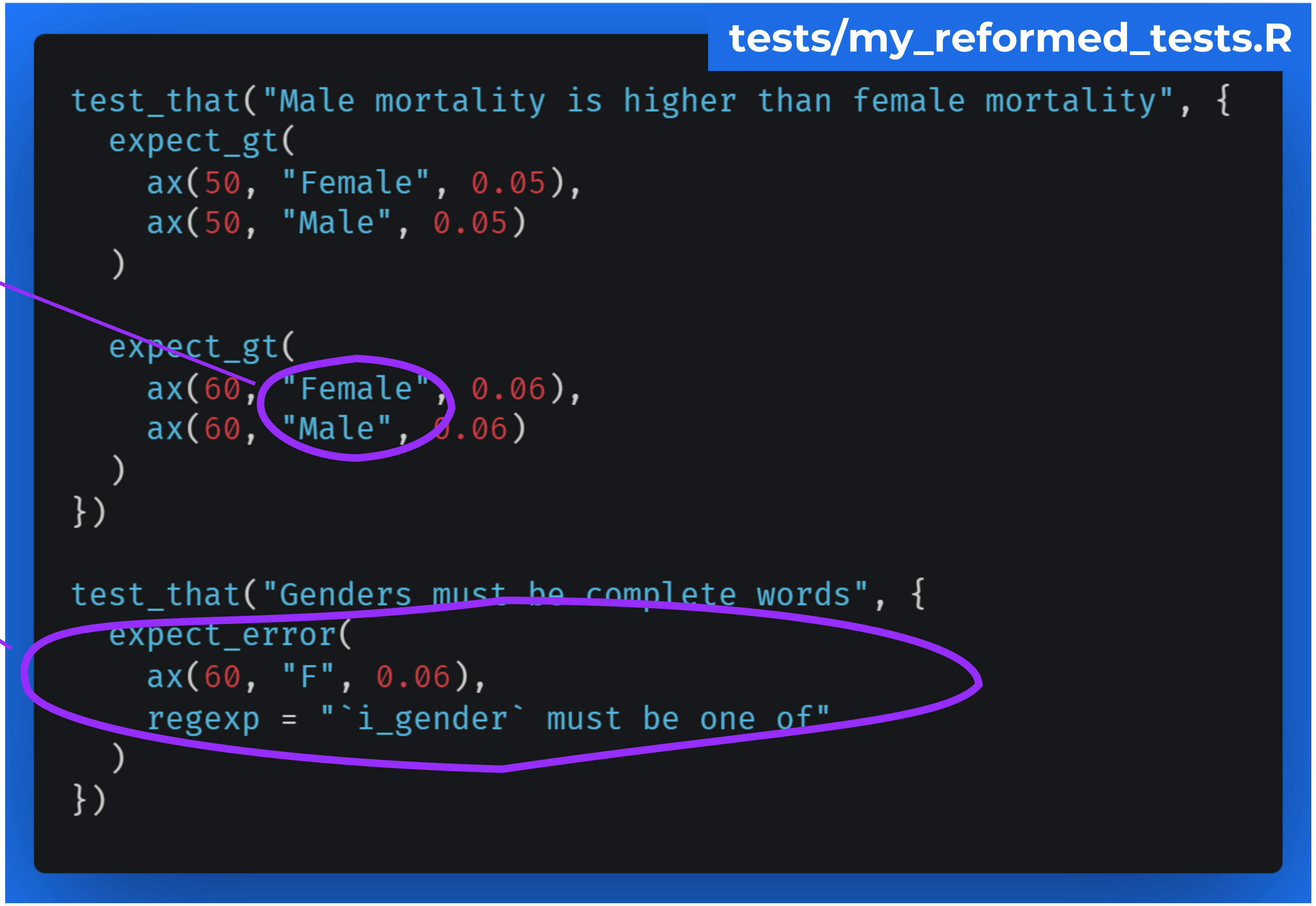
```r
library(tidyverse)
library(actxps)

ax ← function(iss_age, i_gender = c("Male", "Female"),
               disc_rate) {

  i_gender ← rlang::arg_match(i_gender)

  qx ← qx_iamb ▷
    filter(gender == i_gender, age ≥ iss_age) ▷
    pull(qx)
  tpx ← cumprod(1 - qx)
  vt ← (1 + disc_rate) ^ -seq_along(tpx)

  1 + sum(tpx * vt)

}
```

Dealing with failure 2/3

Write a test to capture errors

Corrected

This verifies that an error containing the message "`i_gender` must be one of" is returned



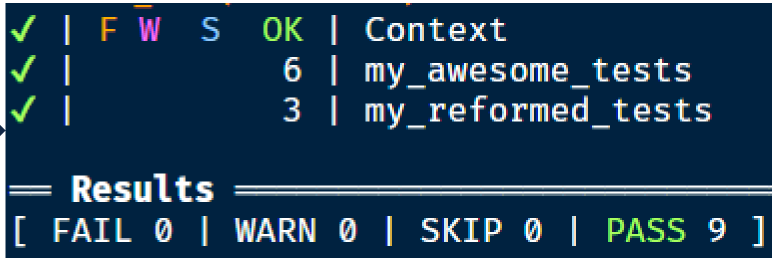
tests/my_reformed_tests.R

```r
test_that("Male mortality is higher than female mortality", {
  expect_gt(
    ax(50, "Female", 0.05),
    ax(50, "Male", 0.05)
  )

  expect_gt(
    ax(60, "Female", 0.06),
    ax(60, "Male", 0.06)
  )
})

test_that("Genders must be complete words", {
  expect_error(
    ax(60, "F", 0.06),
    regexp = "`i_gender` must be one of"
  )
})
```

Dealing with failure 3/3

Run tests again and verify a successful outcome

```
test_dir("tests")
```



```
✓ | F W  S  OK | Context
✓ |          6 | my_awesome_tests
✓ |          3 | my_reformed_tests

══ Results ═════════════════════════════
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 9 ]
```

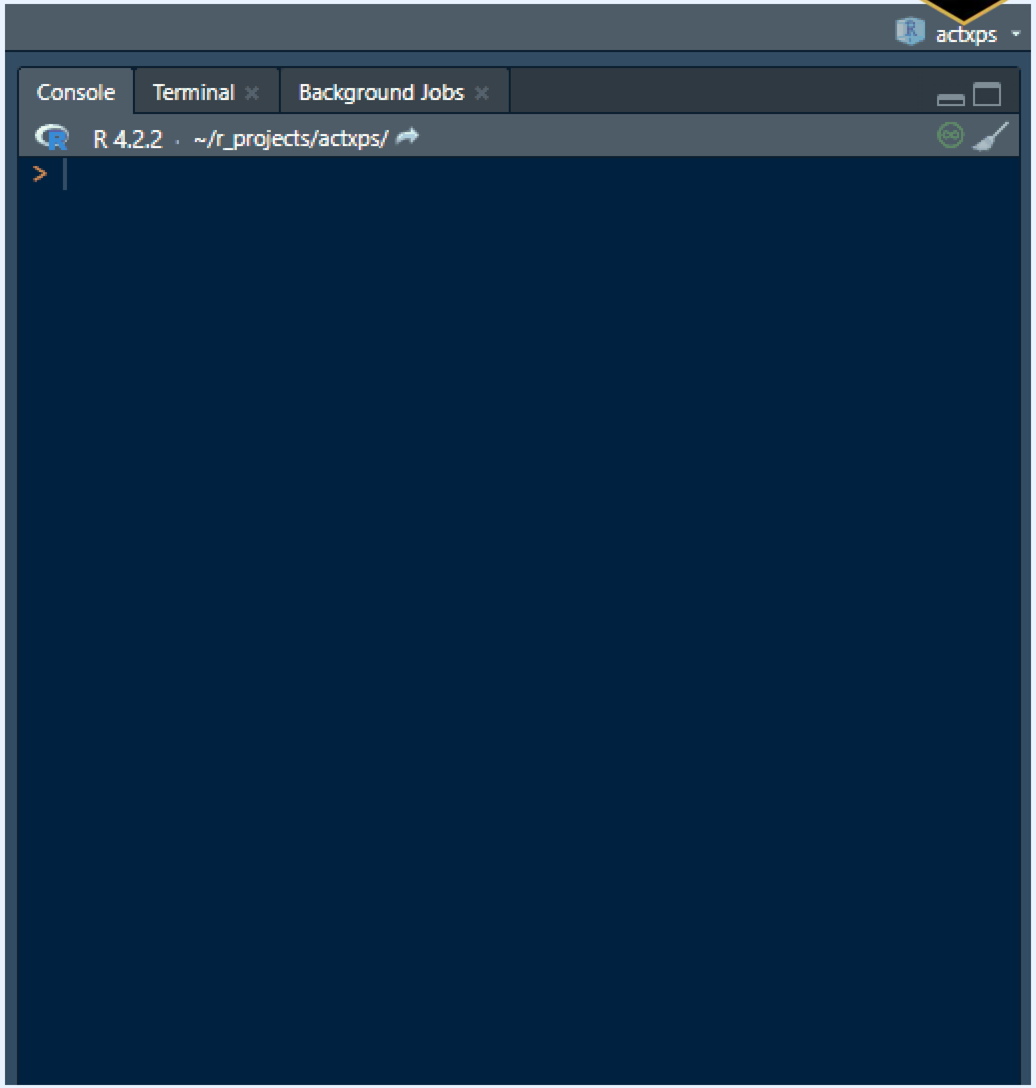
Advanced: Unit Testing & Package Development

Unit testing is an integral component of package development and can be fully automated into routine workflows.

- `devtools::test()`: A single command to run all tests against the current version of the package
- Configure tests to run automatically:
  - Whenever the package is quality checked using R's package checking program
  - Upon creation of a pull request

For more information, see [R Packages (2e) - 13 Testing basics (r-pkgs.org)](https://r-pkgs.org)

Unit testing the actxps R package



```
>
```

Unit Testing Advice

Write tests as you code

Write concise tests

Consider likely input errors

Be comprehensive

Immortalize bugs with new tests

Test, and re-test often

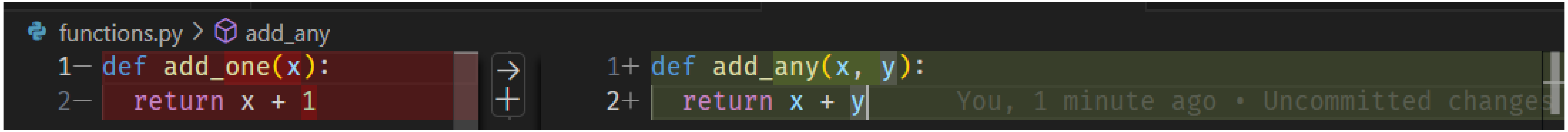
Putting in the upfront work to create robust unit tests can substantially improve the quality and stability of solutions built using open source tools.

Version Control

What is Git?

Distributed Version Control System

Tracks changes to code



```
def add_one(x):
    return x + 1
```

```
def add_any(x, y):
    return x + y
```

You, 1 minute ago • Uncommitted changes

Manages incorporation of changes

Benefits of Git

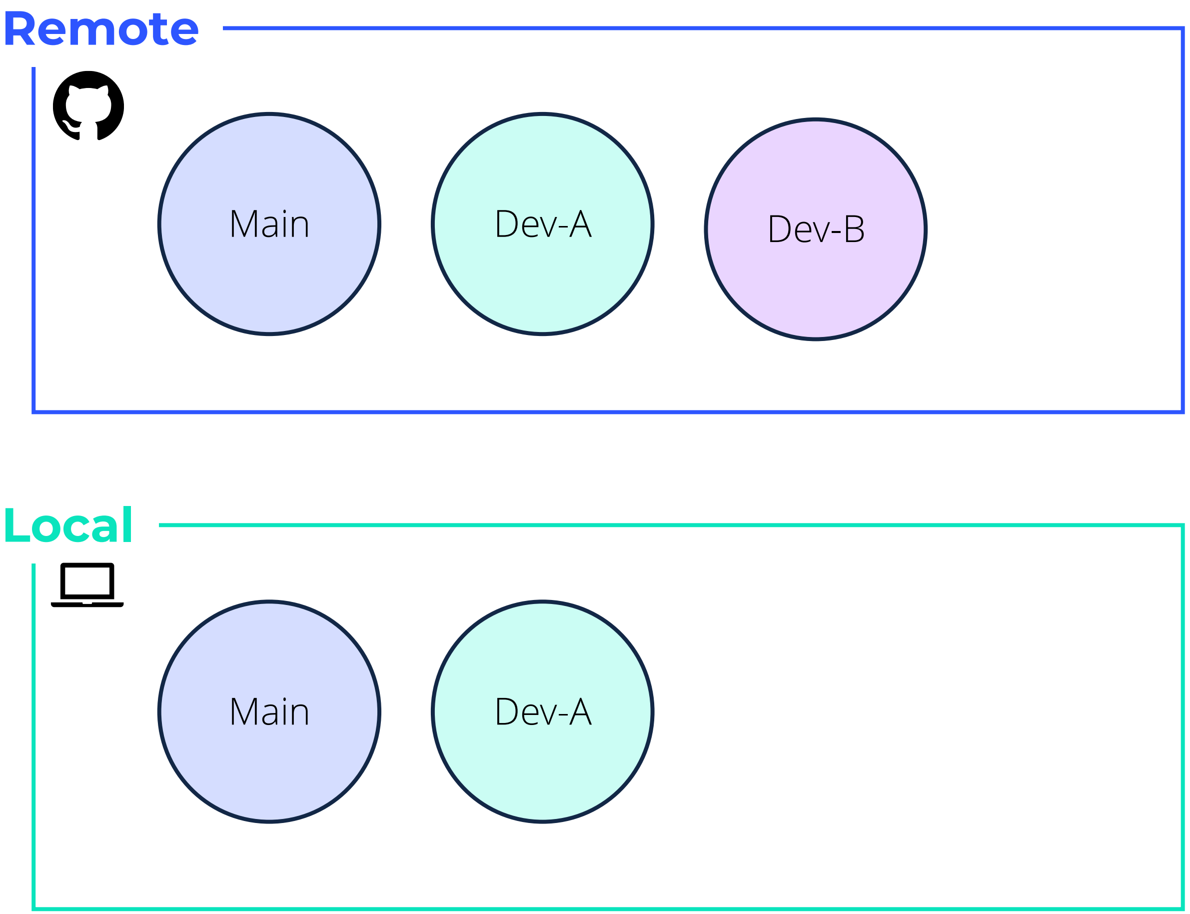
Discipline and Control	Identify changes immediately Require approval and testing
Stability	Reproduce prior results, or revert to prior versions if necessary
Single source of truth	User A and user B are using consistent, tested versions
Encourage exploration	Safely develop new features in a walled-off environment

GitFlow

A code-free primer on working with Git

Definitions

- **Repository (“repo”)**: All the code associated with a particular project
- **Branch**: A copy of the repository that was created for a specific purpose, usually to make a change in a walled-off development environment
- **“Main” Branch**: The tested / approved / locked-down production version of the code
- **Remote**: The centralized official location of the repository, hosted in a service like [GitHub]
- **Local**: A local copy of the repository on your computer

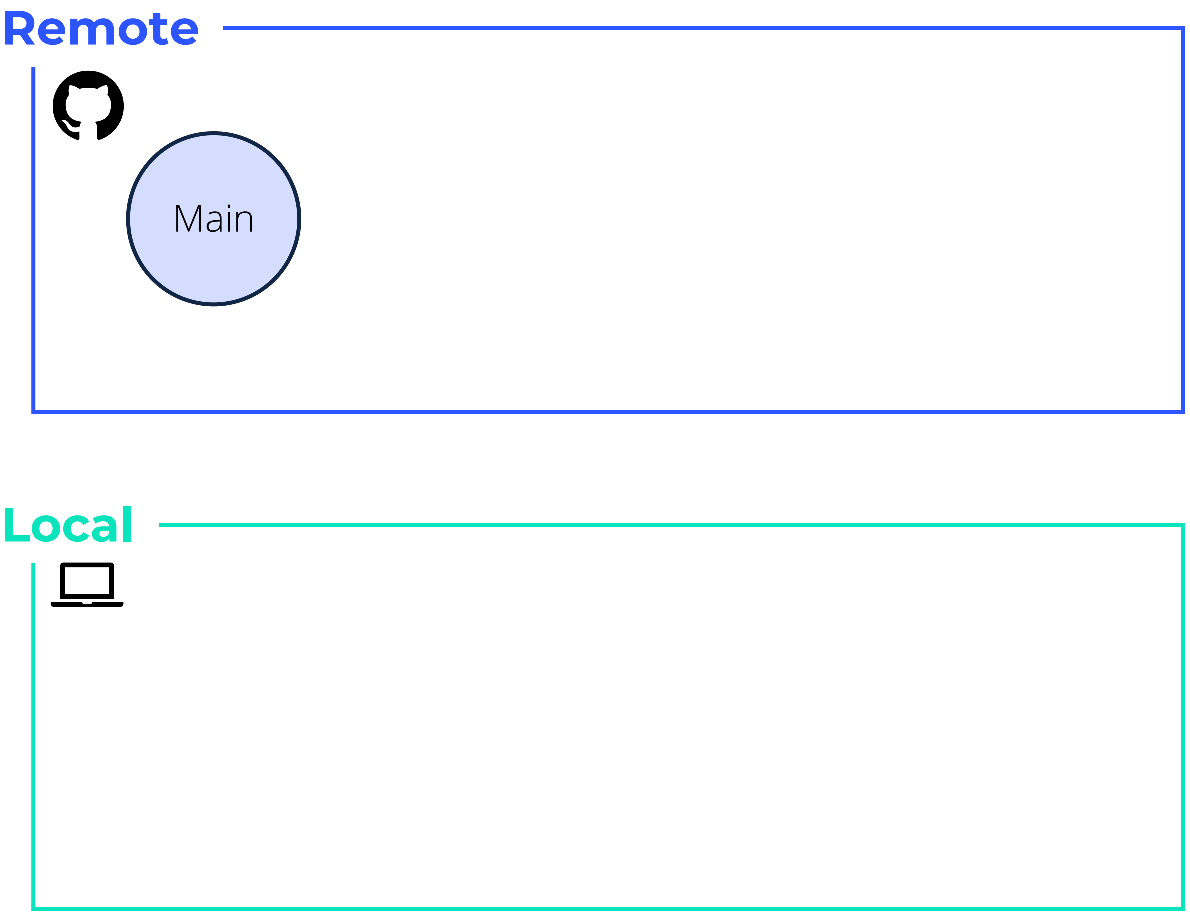


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and save changes**
3. **Commit changes**
4. **Push commits**
5. **Pull request**
6. **Merge**

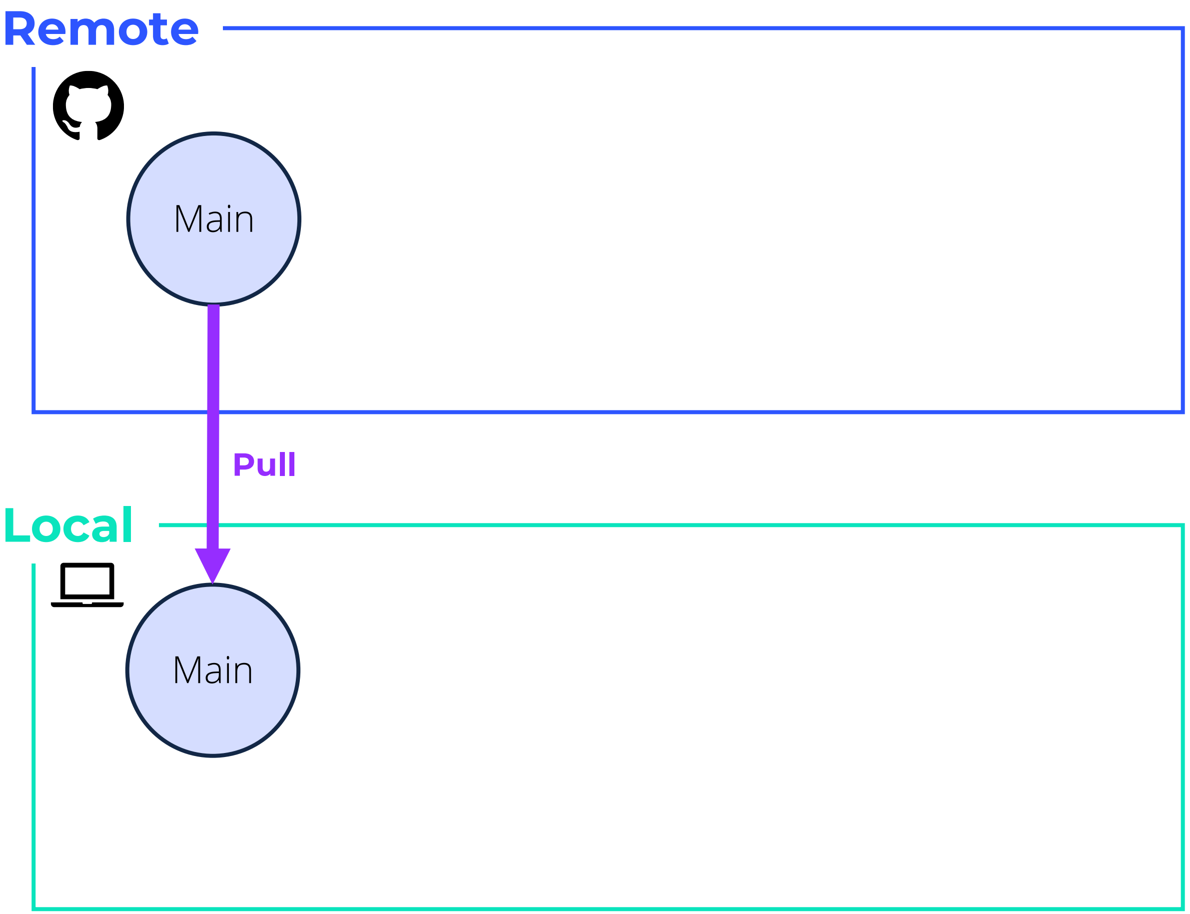


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**: update the local copy of main* to ensure it's up-to-date with the remote.
2. **Make and save changes**
3. **Commit changes**
4. **Push commits**
5. **Pull request**
6. **Merge**



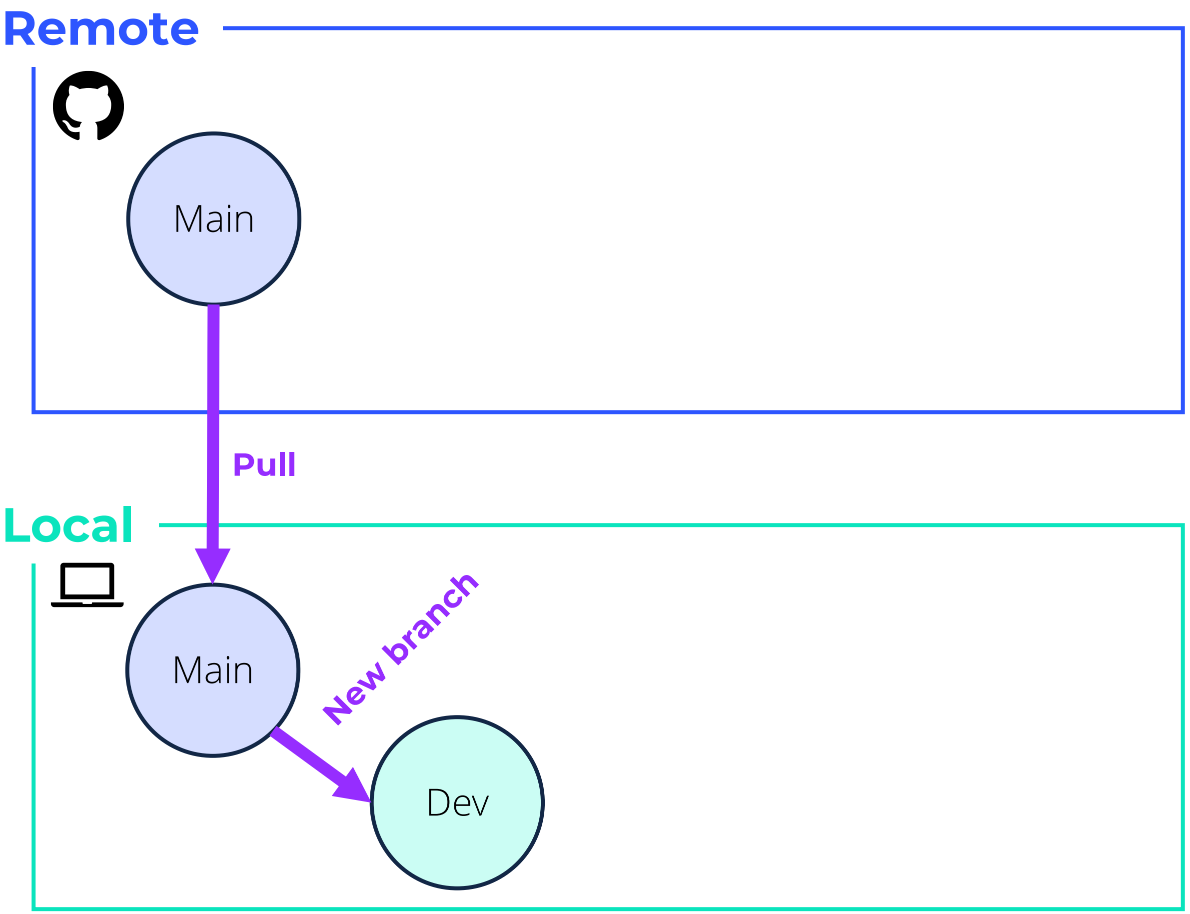
* Or another target to "branch off" from

GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**: update the local copy of main* to ensure it's up-to-date with the remote. Create a development branch.
2. **Make and save changes**
3. **Commit changes**
4. **Push commits**
5. **Pull request**
6. **Merge**



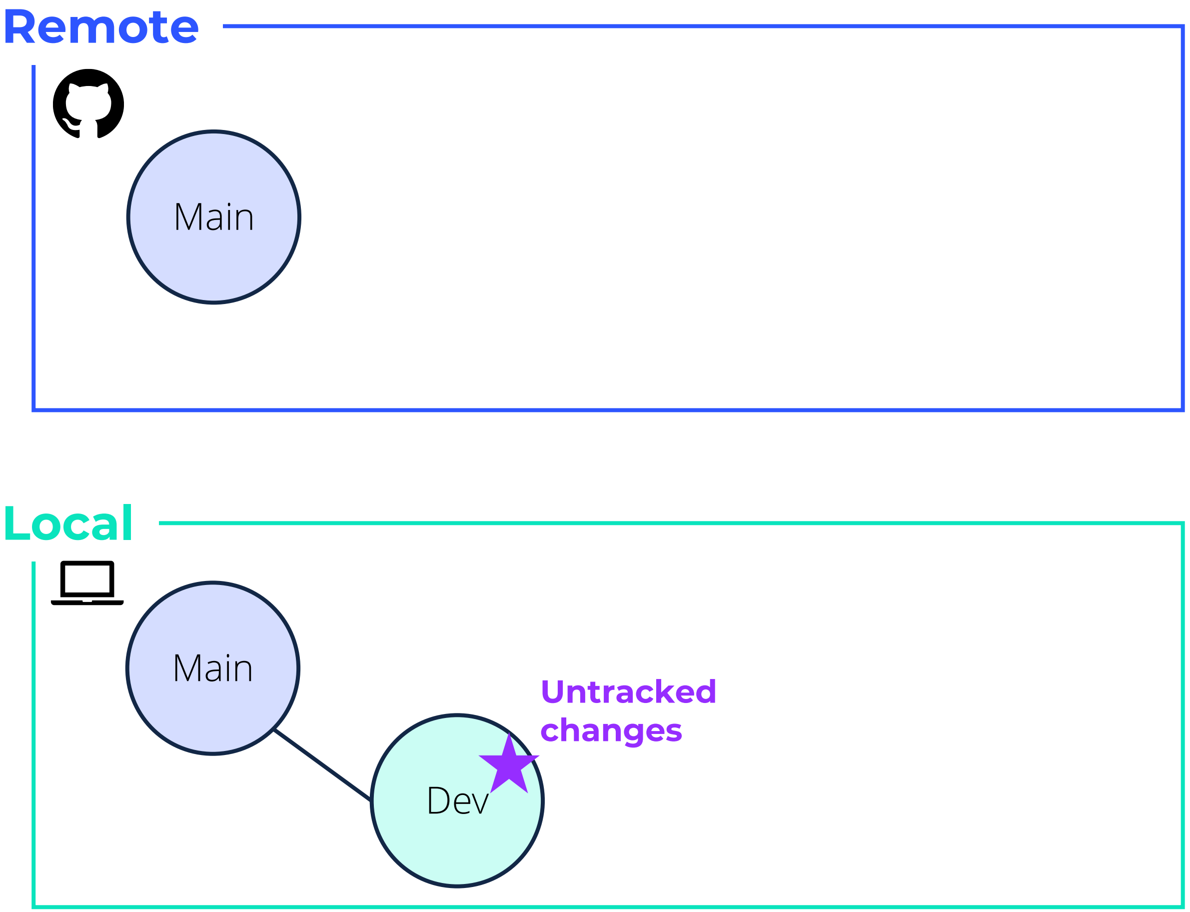
* Or another target to "branch off" from

GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**: Update code as needed.
3. **Commit changes**
4. **Push commits**
5. **Pull request**
6. **Merge**

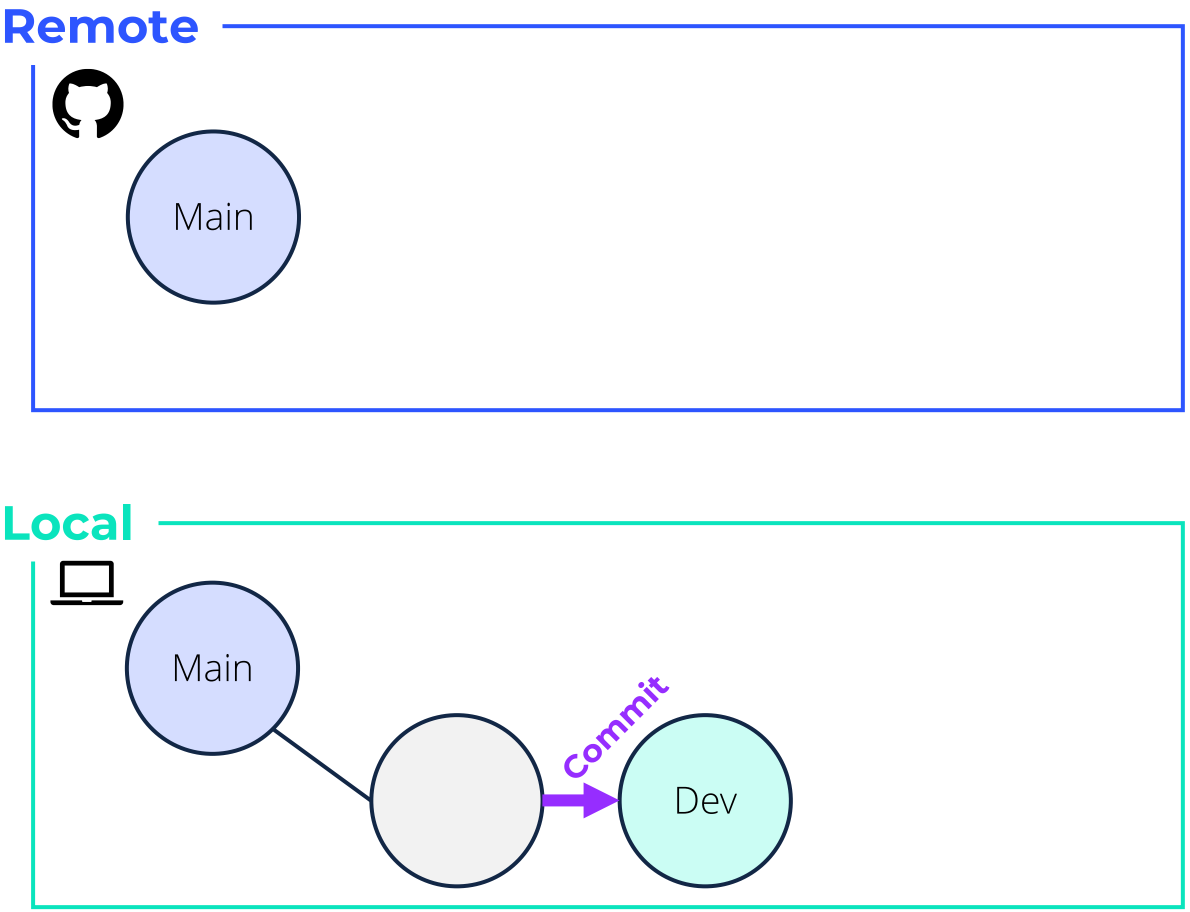


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**
3. **Commit changes**: Formally log changes, telling Git these changes are “good to go”.
4. **Push commits**
5. **Pull request**
6. **Merge**

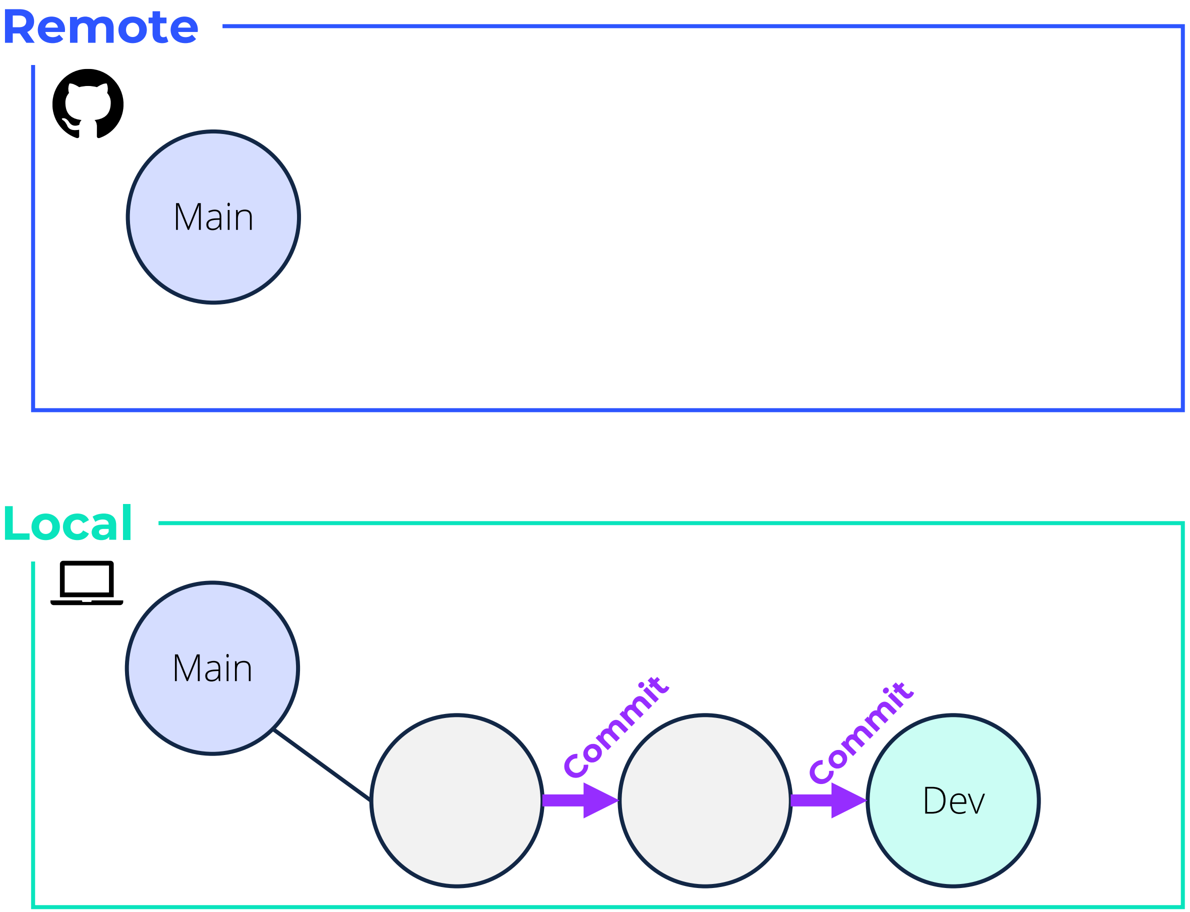


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**
3. **Commit changes**: Formally log changes, telling Git these changes are “good to go”. Repeat as needed.
4. **Push commits**
5. **Pull request**
6. **Merge**

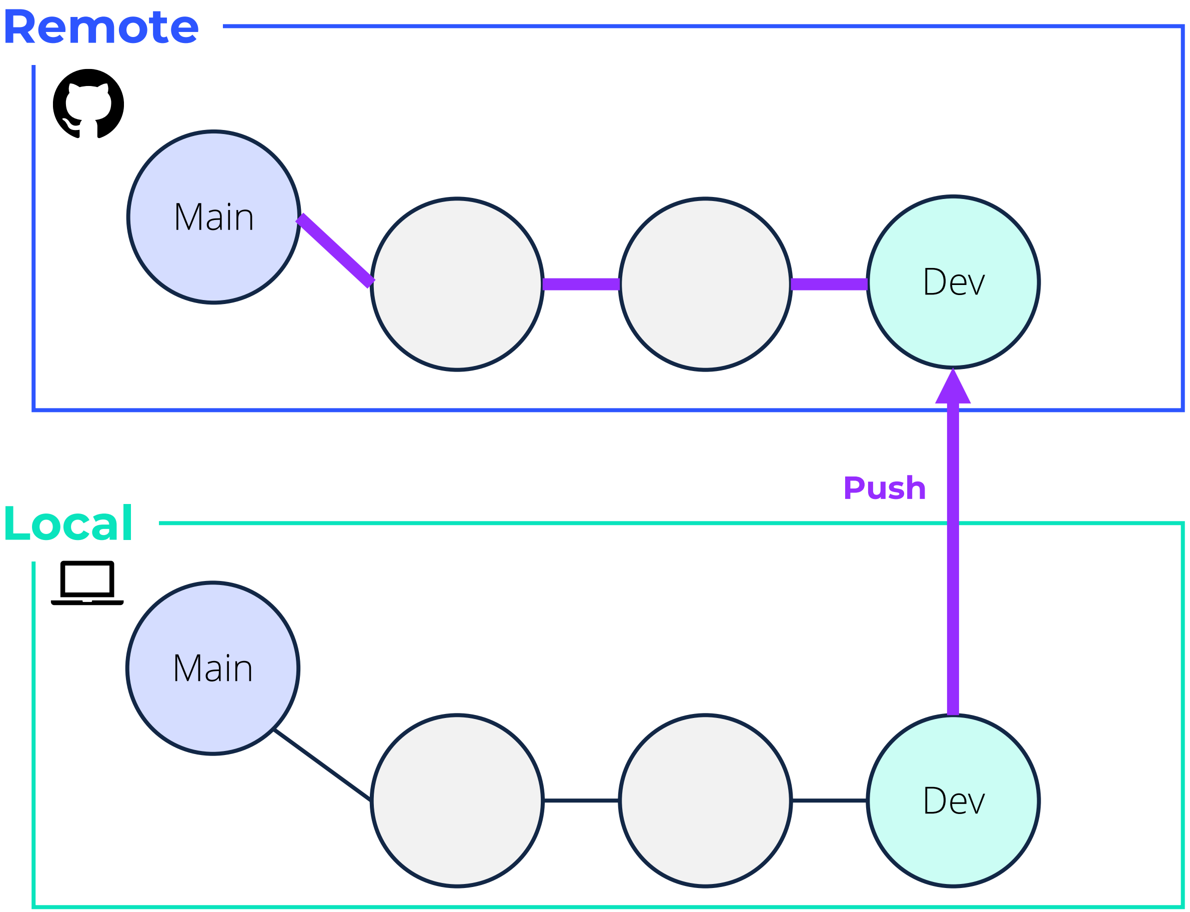


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**
3. **Commit changes**
4. **Push commits**: Publish the development branch to the remote.
5. **Pull request**
6. **Merge**

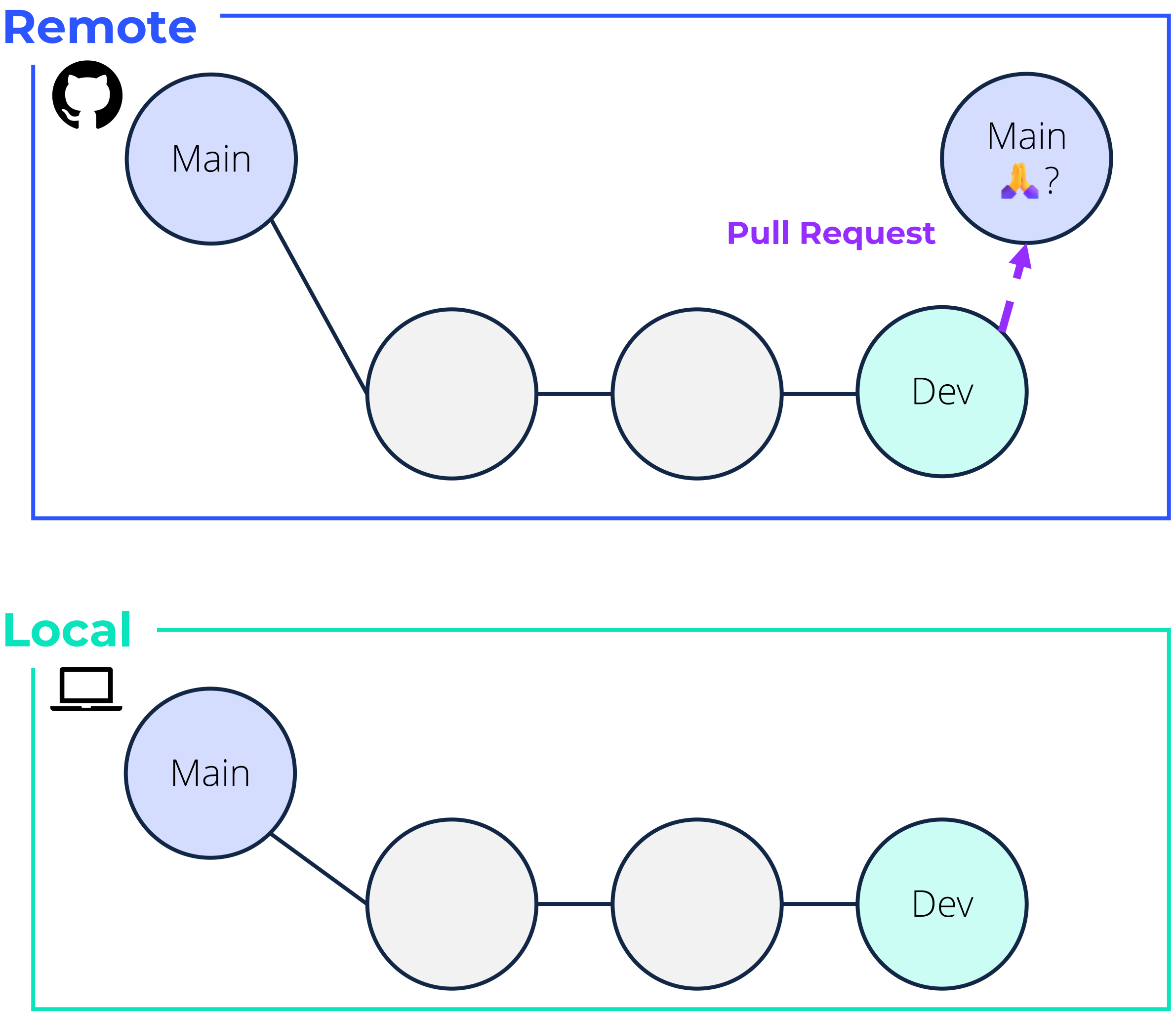


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**
3. **Commit changes**
4. **Push commits**
5. **Pull request**: Submit a request asking for changes in the development branch to be merged into main.
6. **Merge**

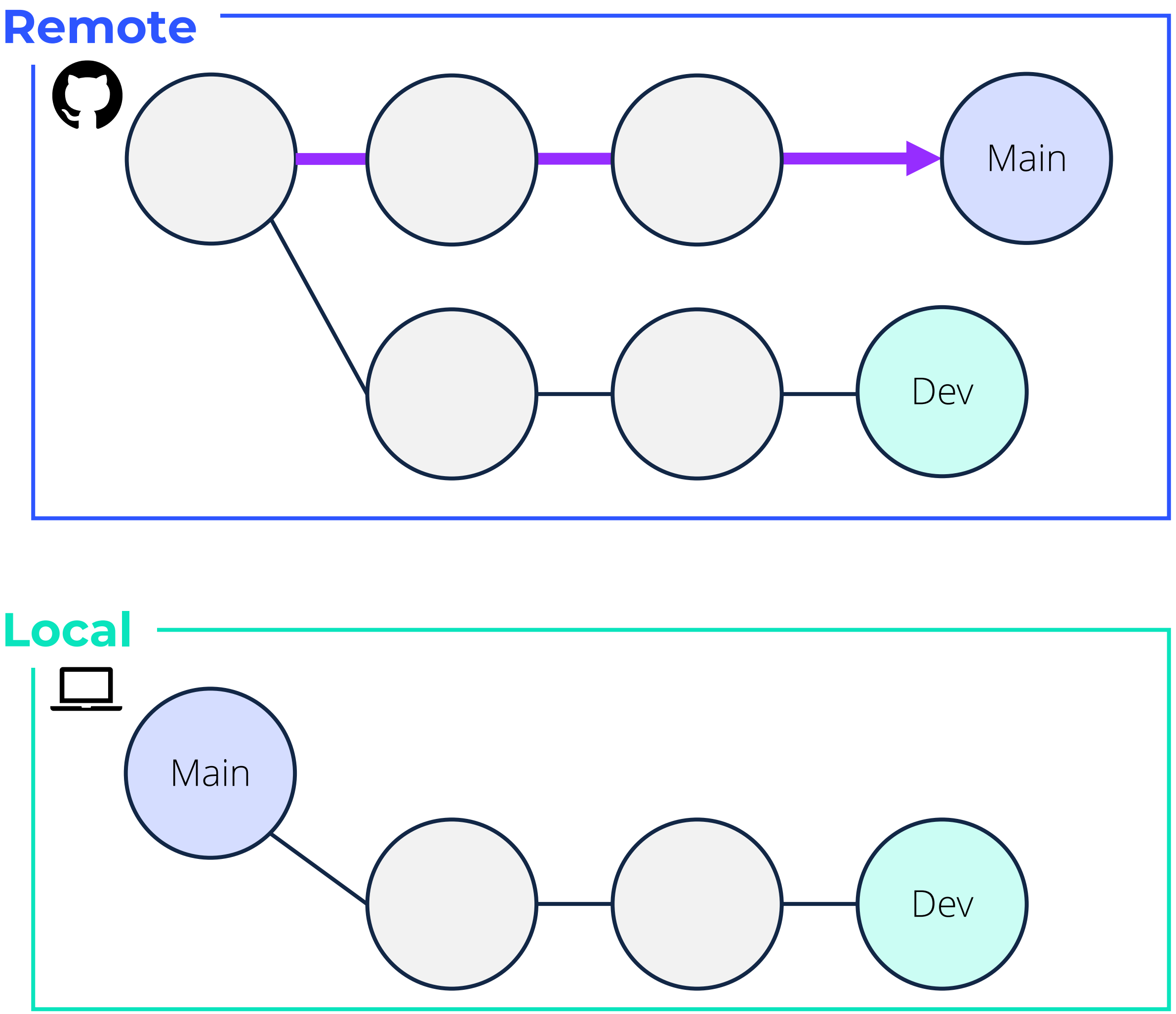


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**
3. **Commit changes**
4. **Push commits**
5. **Pull request**
6. **Merge**: If approved, commits from the development branch are merged into the main branch.

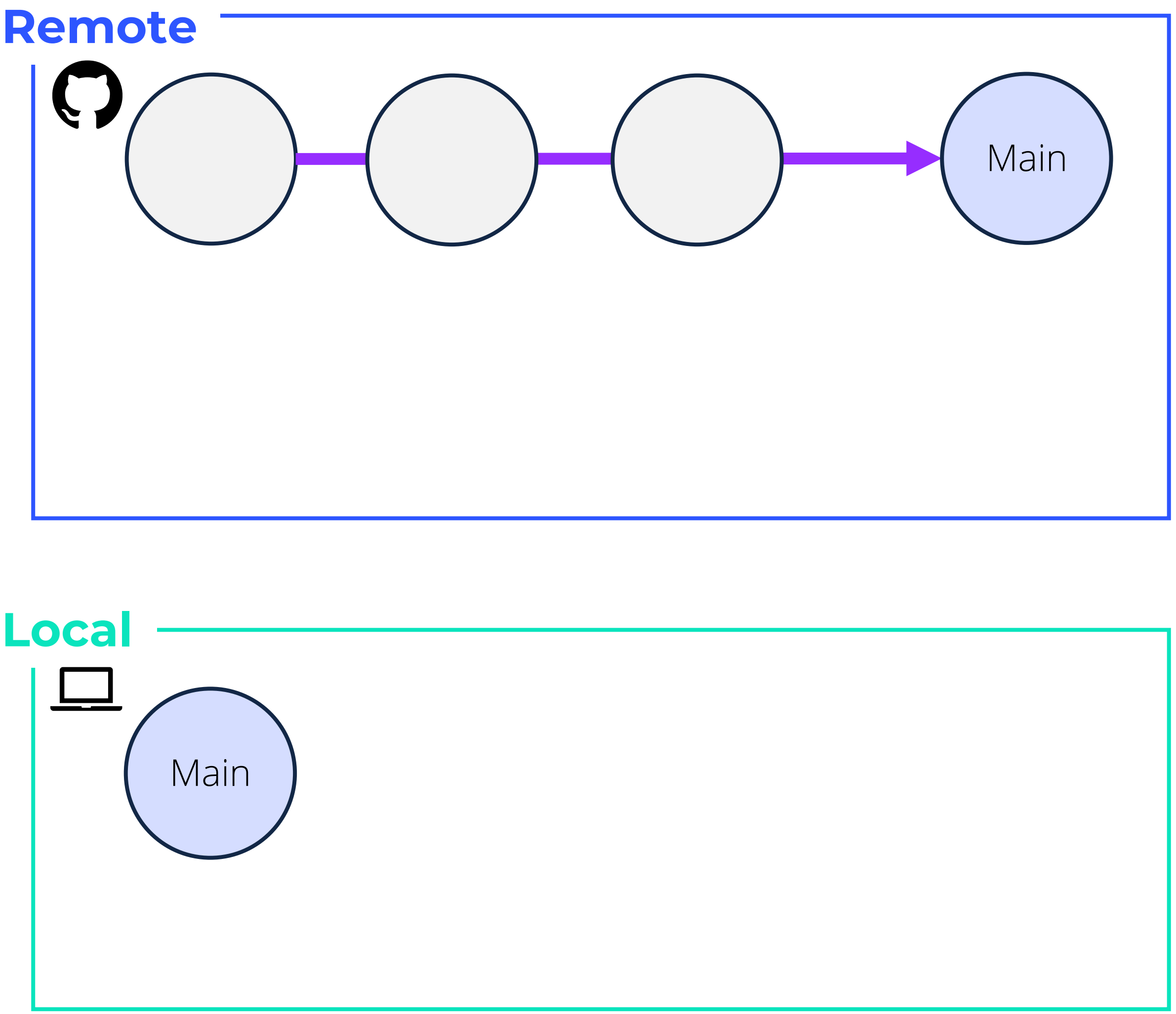


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**
3. **Commit changes**
4. **Push commits**
5. **Pull request**
6. **Merge**: If approved, commits from the development branch are merged into the main branch. Its job complete, the development branch is typically deleted.

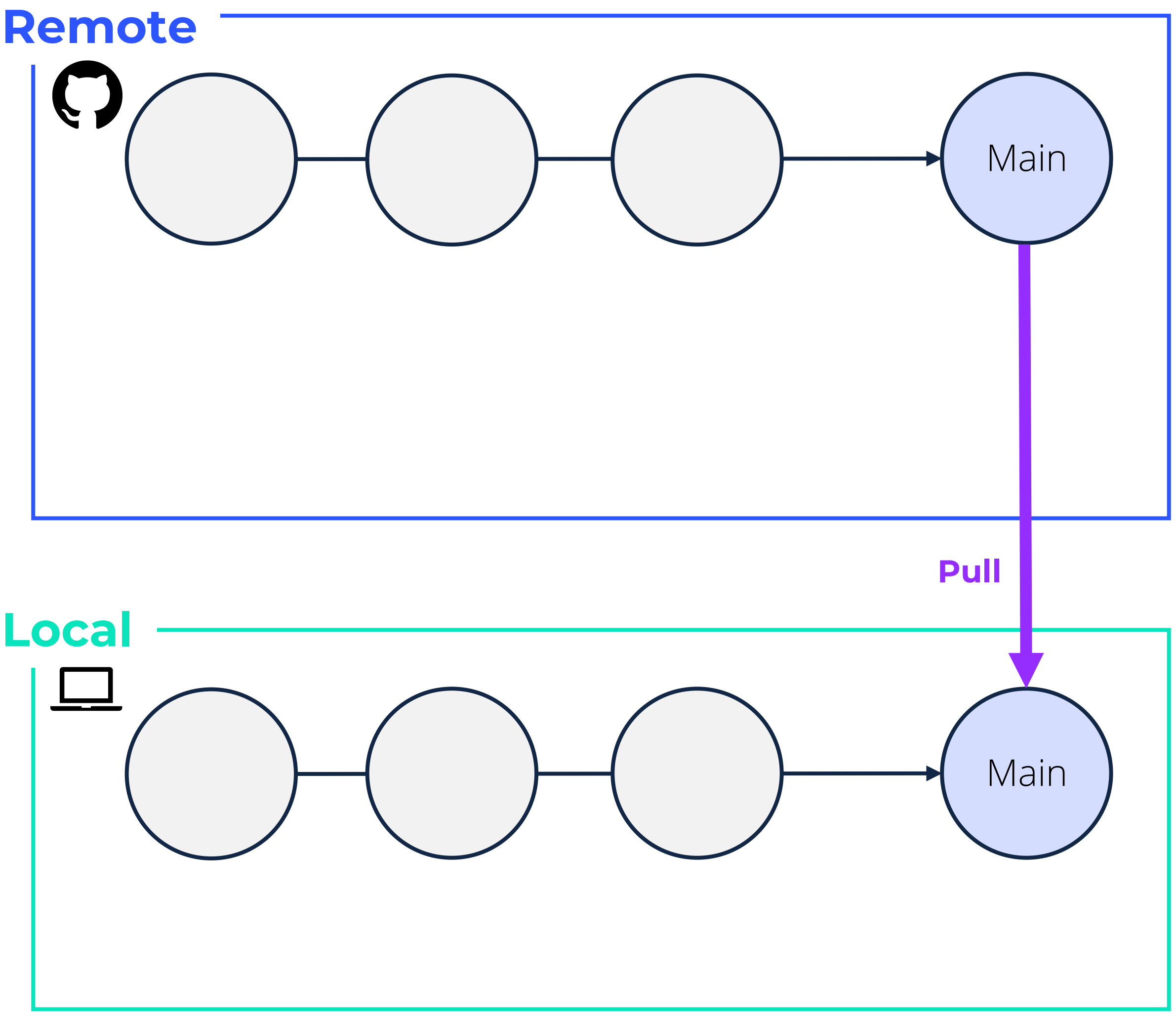


GitFlow

Making changes requires more than Ctrl+S

Change process

1. **Create a branch**
2. **Make and changes**
3. **Commit changes**
4. **Push commits**
5. **Pull request**
6. **Merge**
7. **Wrap-up** (optional): For good measure, synch up the local version of main with the remote



Learning Curve and Best Practices

It takes time to learn Git

- Command line program
- R Studio and VS Code have GUI's – use them

It's a double-edged sword

- Everyday mistakes can be rolled back
- However, after a pull request, Git sins are recorded in the repository's history

Best practices

- Exclude files that shouldn't be tracked
- Never commit large files
- Have robust review and approval process

https://xkcd.com/1597/

Dependency Management

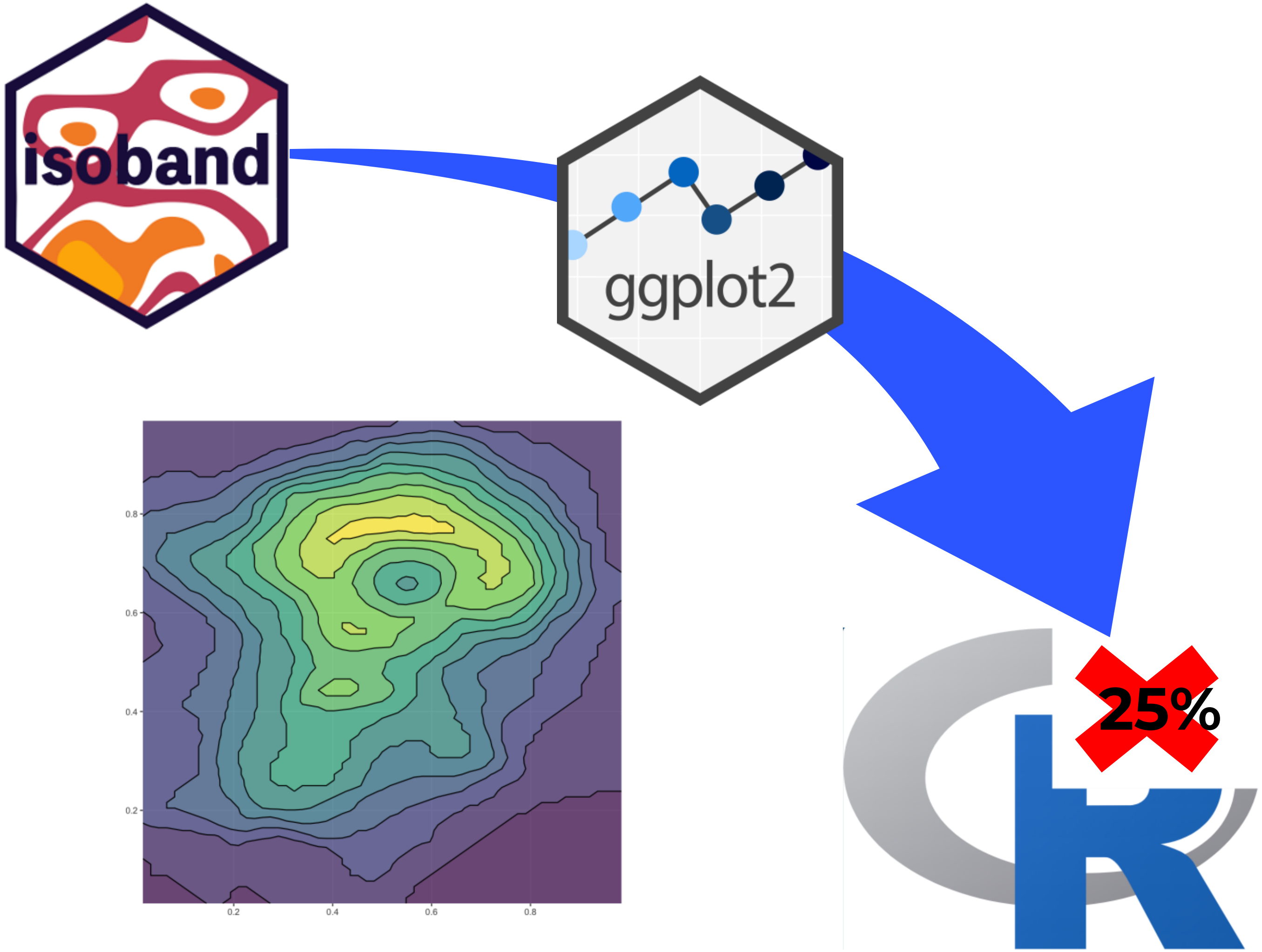
Third Party Dependency Risks

and

Environment Management

The isoband incident

A dramatic near-miss for R users that you haven’t heard about



https://www.r-bloggers.com/2022/10/cran-and-the-isoband-incident-is-your-project-at-risk-and-how-to-fix-it/

October 2022

A small technical issue in a single R package triggered a dependency contagion that momentarily threatened the availability of 25% of all R packages!

Third Party Dependency Advice

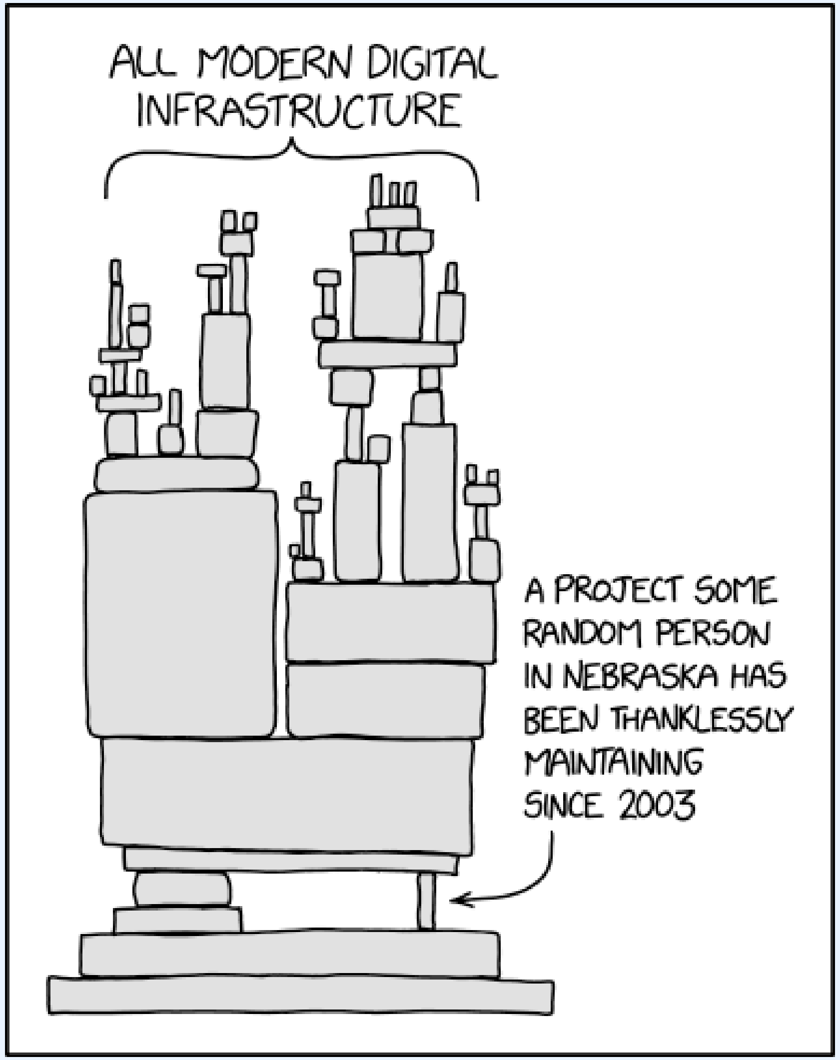
Third party dependency risks are easy to neglect and require a risk assessment

1 Inspect your dependencies

- How mature is the dependent package, and when was the last update?
- Is there a large, existing user base?
- Who is maintaining the package?
- Does the package contain sufficient unit test coverage?
- Have you verified the package works as intended for your purpose?

2 Have a fallback plan

- What happens if a dependency is no longer available from a public repository?
- Is there an alternative public repository?
- Does your company have an internally mirror repository?
- Are there alternative packages?



https://xkcd.com/2347/

Advanced: Environment Management

“But it works for me!” – everyone who uses open source, at some point

1 What is an environment?

A collection of packages, utilities, and functions that your project depends upon.

Examples

- R and Python language versions
- R and Python packages
- C++ header files

2 What’s the risk?

A new feature or a change in a package leads to different results, and your team is using inconsistent package versions.

Example

- R’s native pipe, `|>`, requires v4.1

3 What’s the solution?

Use virtual environment management software to create project-specific environments with locked down version requirements

Available software

- Python: `venv`, `conda`, `virtualenv`
- R: `renv`

For serious production use-cases, environments must be locked down to guarantee everyone is using identical versions of third-party packages.

Wrap-Up

Recap

3 best practices we can glean from software developers

- Unit Testing
- Version Control
- Dependency Management

Martello Re

Thank you