



FSDL 2022

Troubleshooting & Testing

Charles Frye

AUGUST 22, 2022

Agenda

00

TESTING SOFTWARE

Standard tools & practices can
de-risk shipping at speed

01

TESTING ML SYSTEMS

Use specialized techniques
to test ML systems

02

TROUBLESHOOTING MODELS

Run fast first,
run right later

00

Testing software



Key points in this section

- Tests help us ship faster, they don't catch all bugs.
- Use testing tools, but don't chase coverage.
- Use linting tools, but leave escape valves.
- Always be automating.

FSDL Pancakes of Approval

 **Automation:** pre-commit and GitHub Actions

 **Hygiene:** black, flake8, and flake8

 **Testing:** pytest and doctests

Tests help us ship faster,
they don't catch all bugs.

What are tests?

Tests are code we write that's
designed to fail,
in an intelligible way,
when our other code has bugs.

```
def _test_paragraph_text_recognizer(image_filename, expected_text, text_recognizer):  
    """Test ParagraphTextRecognizer on 1 image."""  
  
    predicted_text = text_recognizer.predict(image_filename)  
  
    error_msg = f"predicted text does not match expected for {image_filename.name}"  
    assert predicted_text == expected_text, error_msg
```

Tests alert us to *some* bugs
before they're shipped.

Test suites are not certificates of correctness.

In formal systems, tests are witnesses or proofs of code correctness.

But we aren't writing in Agda or Idris 2.

We are writing in Python. All bets are off.

Test suites are more like classifiers.

The classification problem: does this commit have a 🐛 or is it ?





The classifier outputs: tests pass  or tests fail .

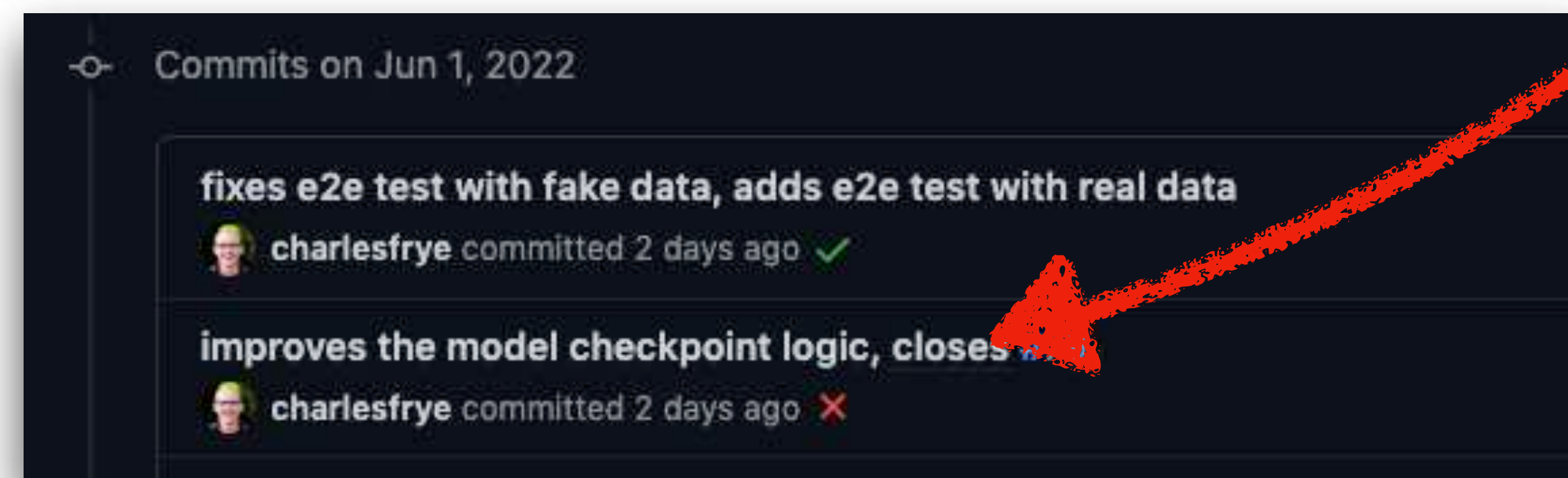
Treat that as a "prediction" of whether there is a bug.

Classifiers trade off detection and false alarms.

As with any classifier, we want:

- Few "missed alarms"
- Few "false alarms"

		
	Missed Alarm	Smooth Sailing
	Caught Bug	False Alarm



Before adding a test, ask yourself two questions.

- Which real bugs will this test catch?
- Which false alarms will this test raise?



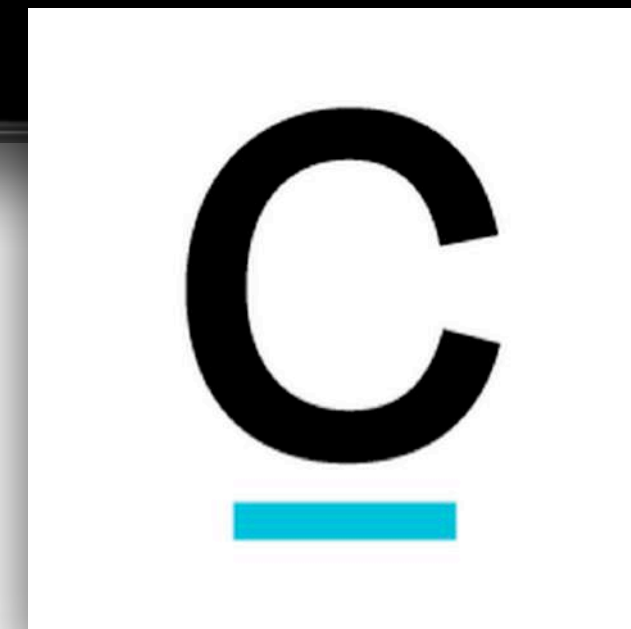
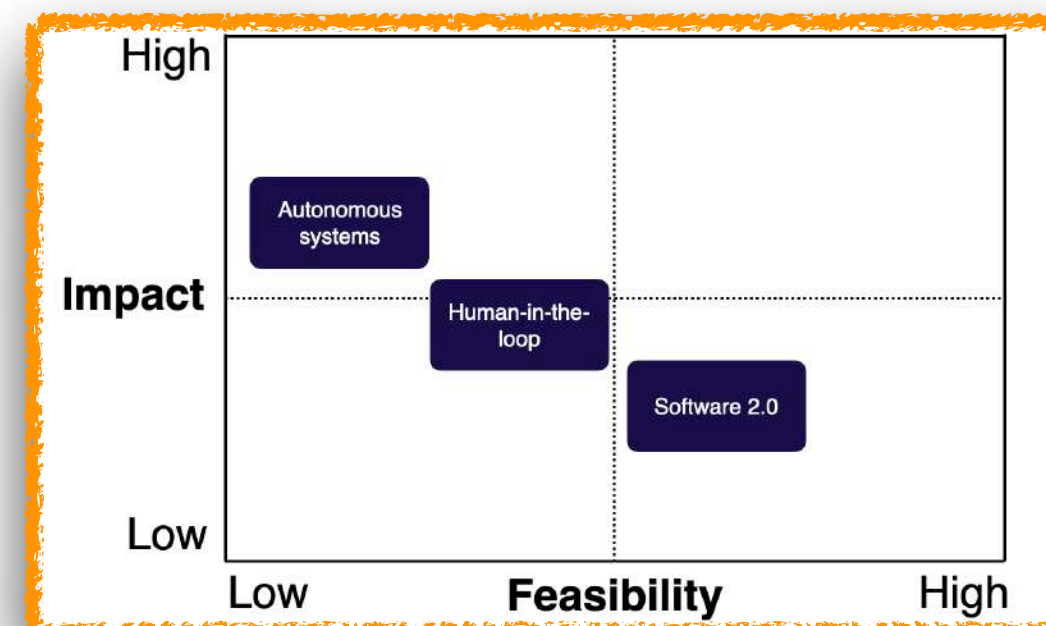
Caveat: in some settings, correctness is important.

Cases

- Medical dx & intervention
- Self-driving vehicles
- Banking, finance

Patterns

- Autonomous systems
- High stakes



Use testing tools,
but don't chase coverage.

Use `pytest` to test Python code.

The standard tool, Pythonic implementation.

Powerful features:

- Separate suites
- Shared resources across tests
- Run parametrized variations of a test



pytest

```
===== test session starts =====
platform linux -- Python 3.7.13, pytest-7.1.1, pluggy-1.0.0
rootdir: /home/charles/fsdl/fsdl-text-recognizer-2022, configfile: pyproject.toml
plugins: cov-3.0.0, anyio-3.6.1, typeguard-2.13.3
collected 8 items / 2 deselected / 6 selected

app_gradio/tests/test_app.py . [ 16%]
text_recognizer/lit_models/util.py .. [ 50%]
text_recognizer/tests/test_iam.py .. [ 83%]
text_recognizer/tests/test_paragraph_text_recognizer.py . [100%]
```

Build trust in docs with `doctests`.

Pure text docs can't be checked for correctness automatically, so they are hard to maintain or trust.

Python has a nice module, `doctests`, for checking code in documentation, preventing rot.

Examples

```
>>> np.dot(3, 4)
12
```

Neither argument is complex-conjugated:

```
>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

For 2-D arrays it is the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

```
804 Examples
805 -----
806 >>> np.dot(3, 4)
807 12
808
809 Neither argument is complex-conjugated:
810
811 >>> np.dot([2j, 3j], [2j, 3j])
812 (-13+0j)
813
814 For 2-D arrays it is the matrix product:
815
816 >>> a = [[1, 0], [0, 1]]
817 >>> b = [[4, 1], [2, 2]]
818 >>> np.dot(a, b)
819 array([[4, 1],
820        [2, 2]])
821
822 >>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
823 >>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
824 >>> np.dot(a, b)[2,3,2,1,2,2]
825 499128
826 >>> sum(a[2,3,2,:] * b[1,2,:,2])
827 499128
828
829 """
```



Enrich tests and documentation with tested notebooks.

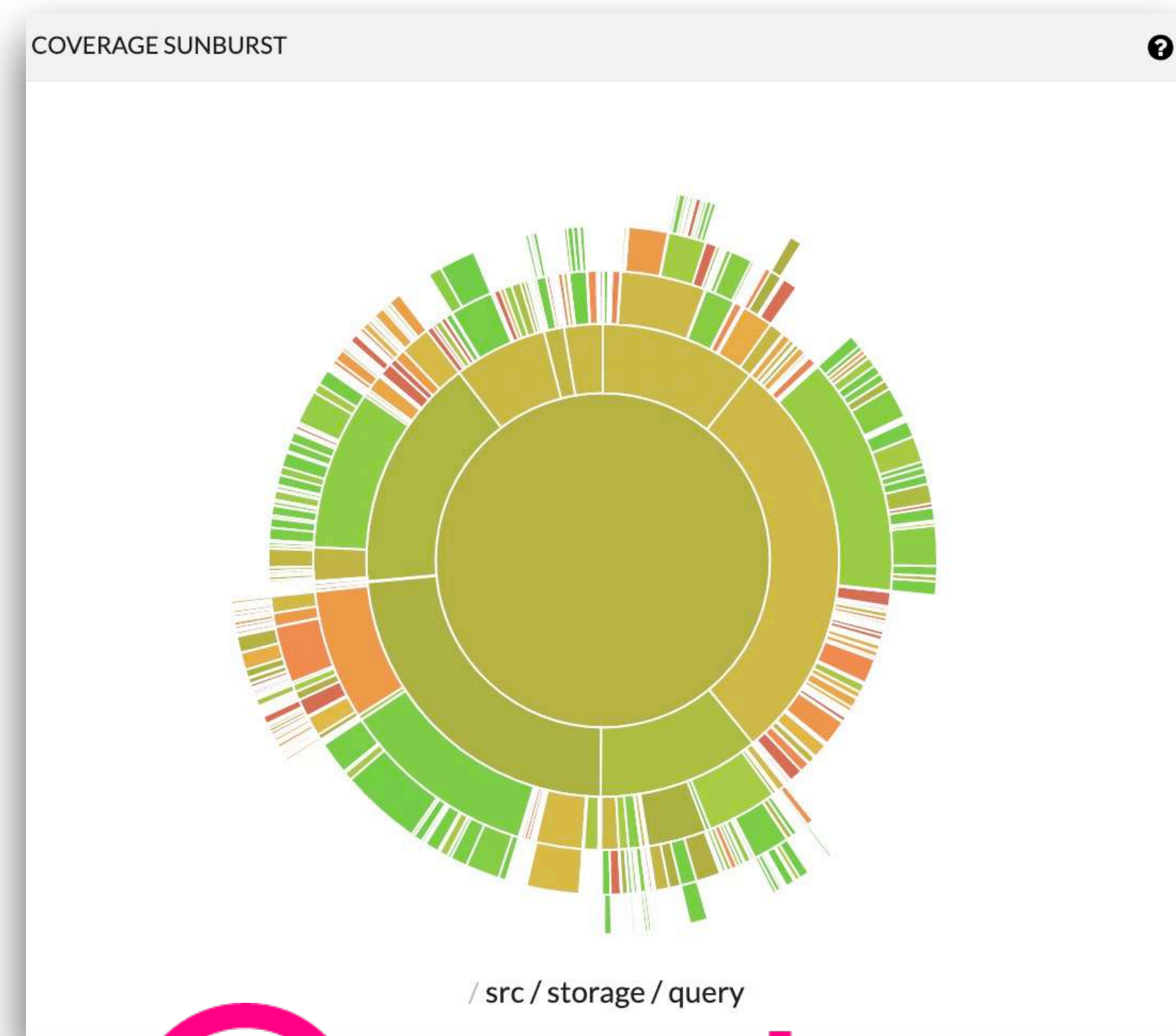
Notebooks help connect rich media, like charts, images, and web pages, with code execution.

Cheap and dirty solution: add some asserts and use `nbformat` to run the notebooks.



We recommend Codecov for checking test coverage.

- Coverage tools record which code is checked or "covered" by tests.
- Typically in terms of lines of code, but sometimes finer-grained.
- Helps you understand your tests.
- Can be incorporated into testing: reject commits that reduce



Friends don't let friends target test coverage.



Charity Majors ✓
@mipsytipsy

You will get 80% of the value with 20% of the effort you spend on your test suite.

[@caitie](#) presented a paper on this at [@paperswelovenyc](#) a few years ago. So it's not even just intuitively thus, there's actual research behind it. 🙄

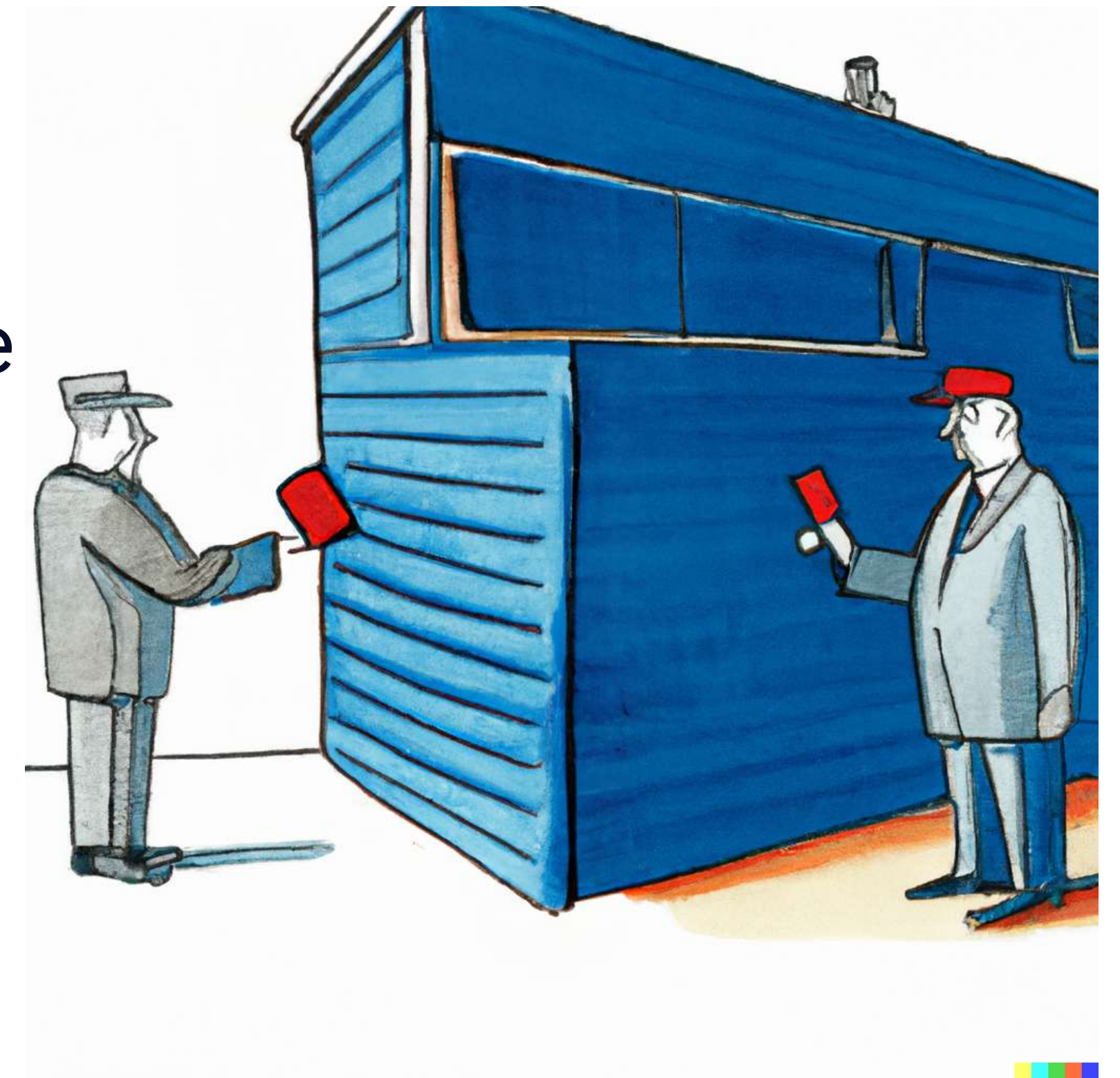
1:50 PM · May 23, 2022 · Twitter for iPhone

Use linting tools,
but leave escape valves.

Clean code is of uniform and standard style.

Why?

- Avoid "bike-shedding" arguments over style
- Cut down on noise in & size of diffs
- More welcoming to readers



"Let's repaint the bike shed red this sprint."

We recommend `black` for Python auto-formatting.

- Standard tool
- Highly opinionated with a narrow scope
- Fully automated

```
reformatted ../fsdl-text-recognizer-2022-labs/lab01/text_recognizer/models/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab02/text_recognizer/lit_models/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab02/text_recognizer/data/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab02/text_recognizer/models/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab03/text_recognizer/data/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab01/text_recognizer/util.py
reformatted ../fsdl-text-recognizer-2022-labs/lab03/text_recognizer/lit_models/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab02/text_recognizer/util.py
reformatted ../fsdl-text-recognizer-2022-labs/lab03/text_recognizer/models/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab02/text_recognizer/lit_models/base.py
reformatted ../fsdl-text-recognizer-2022-labs/lab02/training/run_experiment.py
reformatted ../fsdl-text-recognizer-2022-labs/lab04/text_recognizer/data/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab04/text_recognizer/lit_models/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab03/text_recognizer/lit_models/transformer.py
reformatted ../fsdl-text-recognizer-2022-labs/lab03/text_recognizer/util.py
reformatted ../fsdl-text-recognizer-2022-labs/lab04/text_recognizer/models/__init__.py
reformatted ../fsdl-text-recognizer-2022-labs/lab03/text_recognizer/lit_models/base.py
reformatted ../fsdl-text-recognizer-2022-labs/lab04/text_recognizer/util.py
reformatted ../fsdl-text-recognizer-2022-labs/lab04/text_recognizer/lit_models/base.py

All done! ✨ 🍰 ✨
19 files reformatted, 87 files left unchanged.
[~/fsdl/fsdl-text-recognizer-2022]$
[fsdl] 1:..cognizer-2022* 2:~/fsdl- 3:jupyter
```

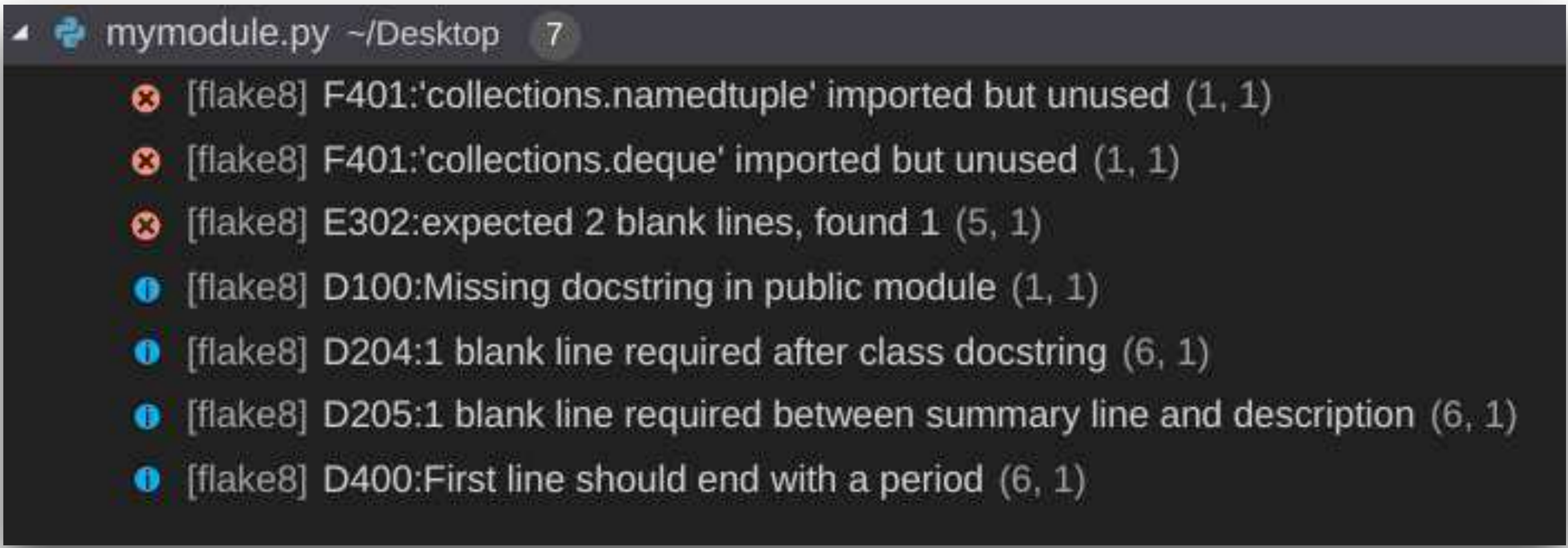


We recommend `flake8` for Python style.

- The standard tool for non-automatable style.

- Tons of plugins:

- Docstring style (`flake8-docstrings`)
- Docstring completeness (`darglint`)
- Type hinting (`flake8-annotations`)
- Security (`flake8-bandit`)
- Common bugs (`flake8-bugbear`)



```
mymodule.py ~/Desktop 7
[flake8] F401:'collections.namedtuple' imported but unused (1, 1)
[flake8] F401:'collections.deque' imported but unused (1, 1)
[flake8] E302:expected 2 blank lines, found 1 (5, 1)
[flake8] D100:Missing docstring in public module (1, 1)
[flake8] D204:1 blank line required after class docstring (6, 1)
[flake8] D205:1 blank line required between summary line and description (6, 1)
[flake8] D400:First line should end with a period (6, 1)
```


We recommend shellcheck to test shell scripts.

```
1
>>24 echo "Testing these notebooks: ${NOTEBOOKS}"
1 FAILURE=false
>> 2 for NOTEBOOK in ${NOTEBOOKS[@]}
3 do # loop over notebooks, executing each one
4     echo "Testing $NOTEBOOK"
S> 5     jupyter nbconvert --to notebook --ExecutePreprocessor.timeout=$MAX_RUNTIME --execute "$NOTEBOOK" | _FAILURE=true
6 done
7
S> 8 if [ "$_FAILURE" = true ]; then
9     echo "Notebook tests failed"
10     exit 1
11 fi
12 echo "Notebook tests passed"
[Syntax: line:12 (5)]
1 ./tasks/notebook_test.sh|12 col 22 warning| SC2046: Quote this to prevent word splitting.
1 ./tasks/notebook_test.sh|24 col 32 warning| SC2128: Expanding an array without an index only gives the first element.
2 ./tasks/notebook_test.sh|26 col 17 error| SC2068: Double quote array expansions to avoid re-splitting elements.
3 ./tasks/notebook_test.sh|29 col 104 info| SC2030: Modification of FAILURE is local (to subshell caused by pipeline).
4 ./tasks/notebook_test.sh|32 col 7 info| SC2031: FAILURE was modified in a subshell. That change might be lost.
```

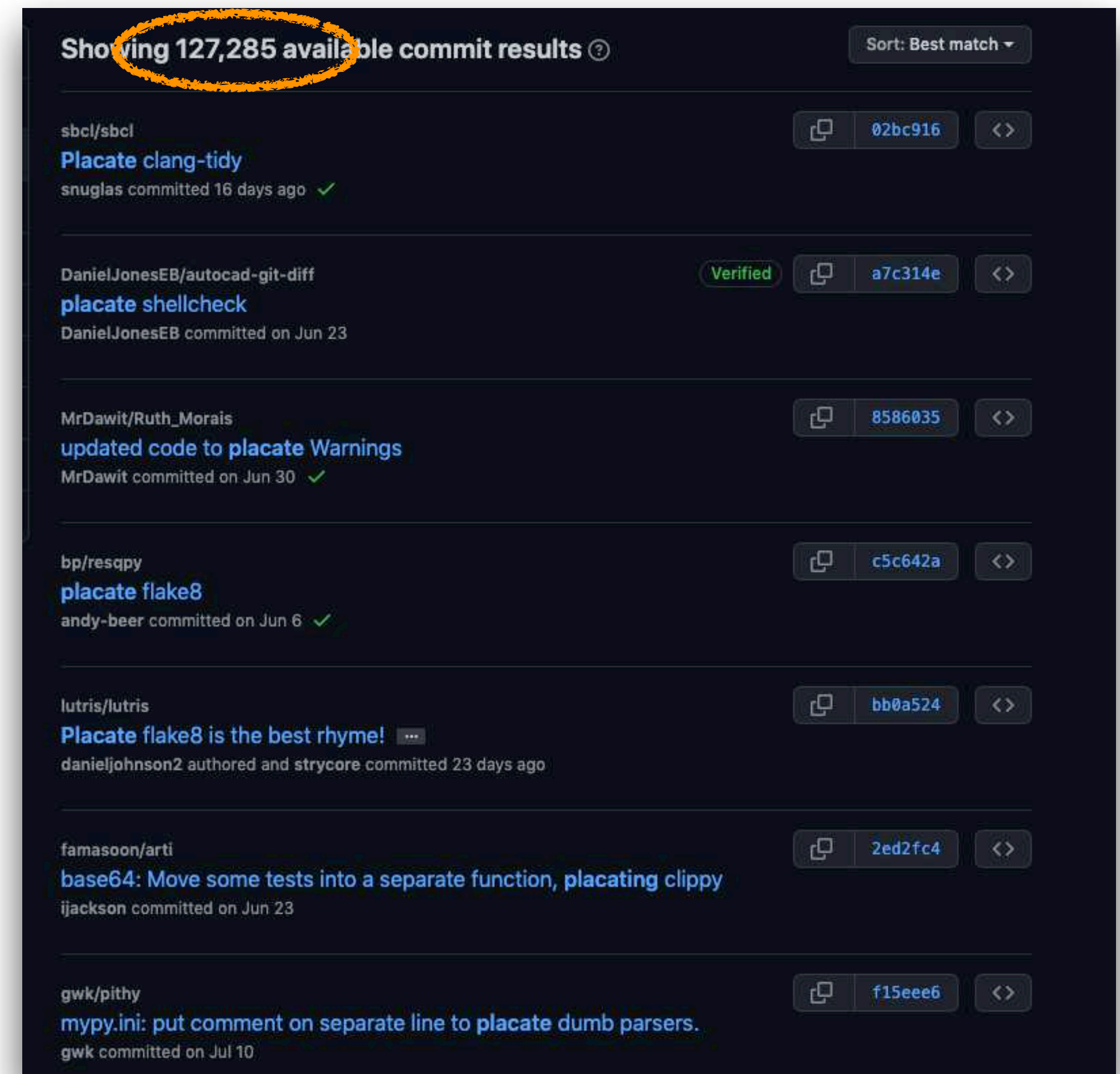
- Knows all the weird behaviors of bash, explains them
- Very fast, so tight in-editor loop



Caveat: pedantic enforcement of style is obnoxious.

We recommend

- filtering rules down to the minimal style that achieves goals
- "opt-in" application of rules, growing coverage over time, especially for existing codebases



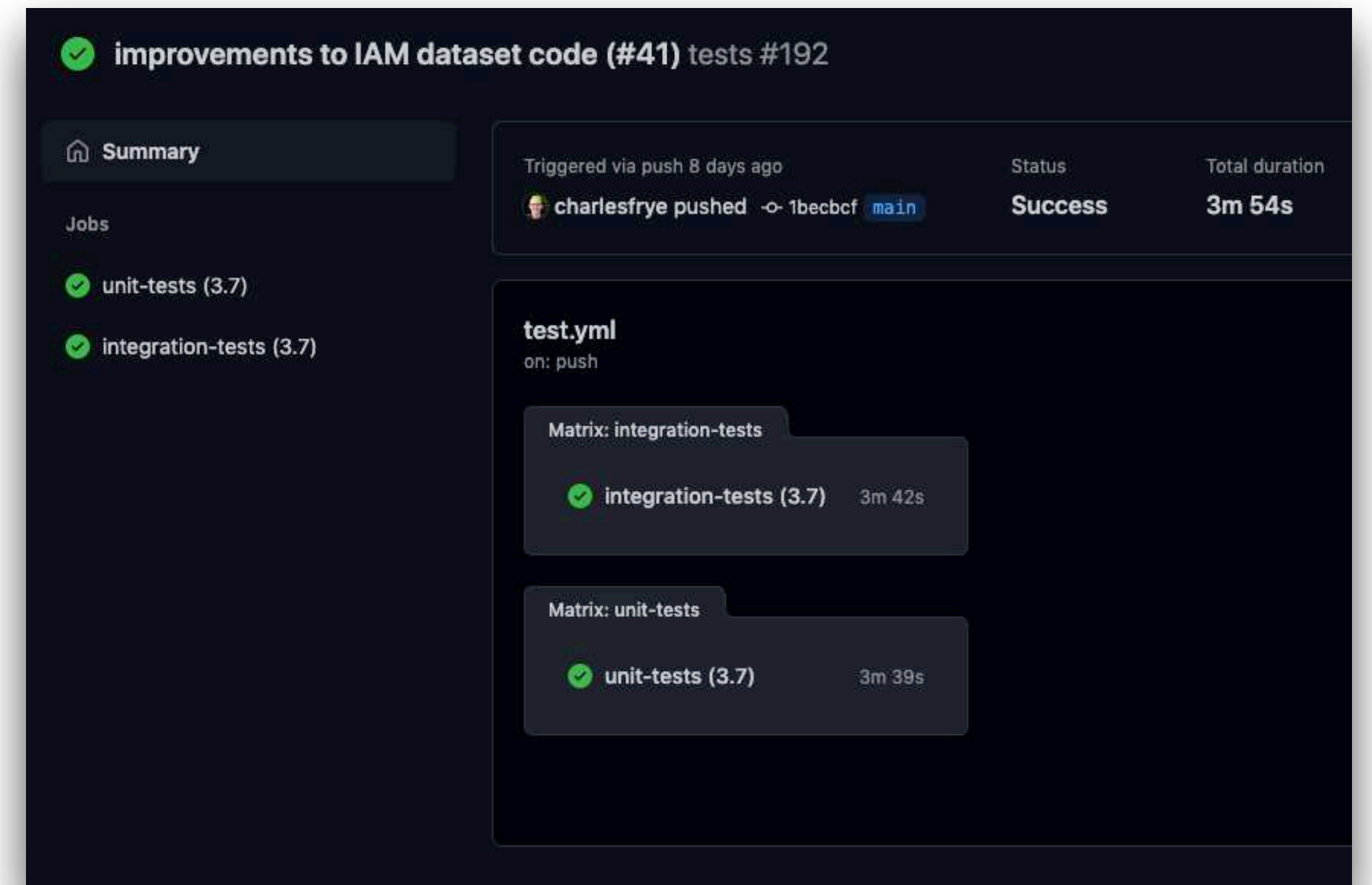
Search results for "placate" on GitHub, 2022-08-18

Always be automating.

Automate these tasks and connect to your cloud VCS.

Some benefits:

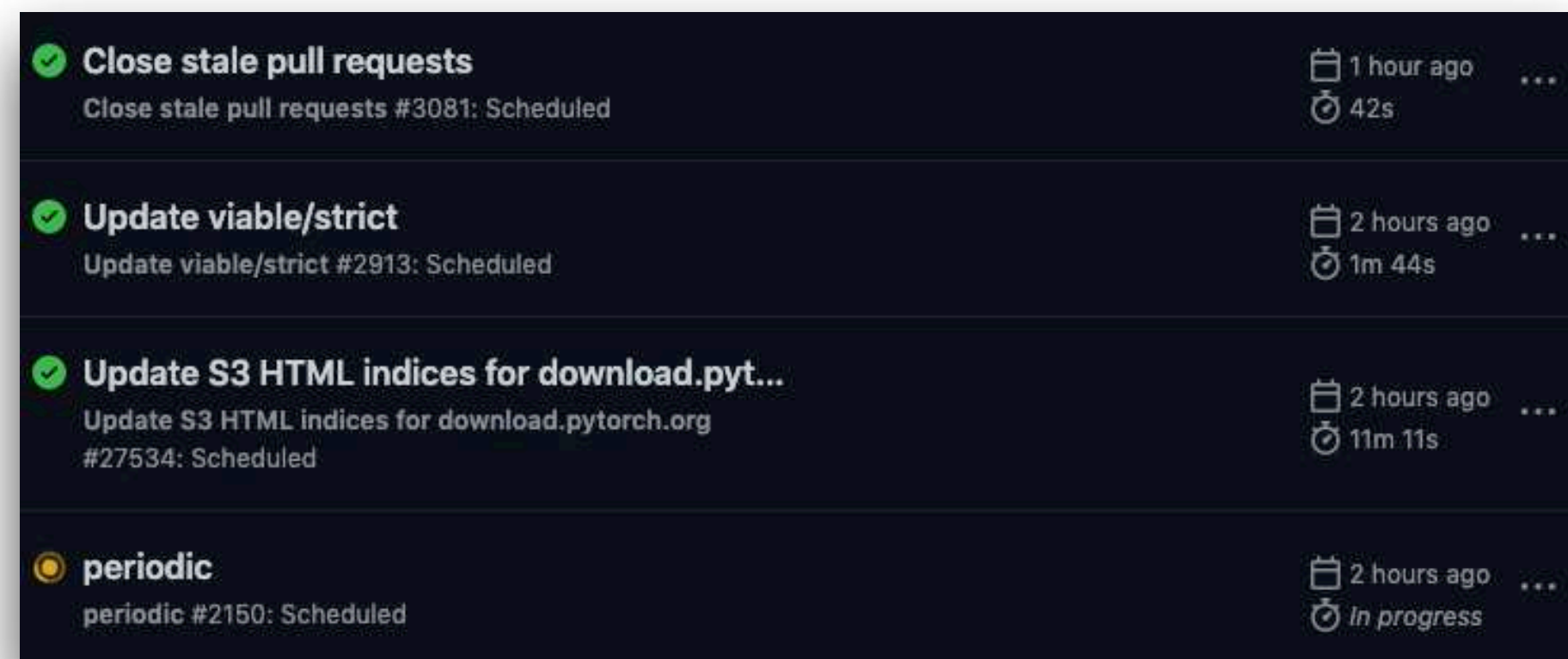
- connection to VCS state reduces friction when reproducing errors
- run tests automatically, in parallel to other dev work



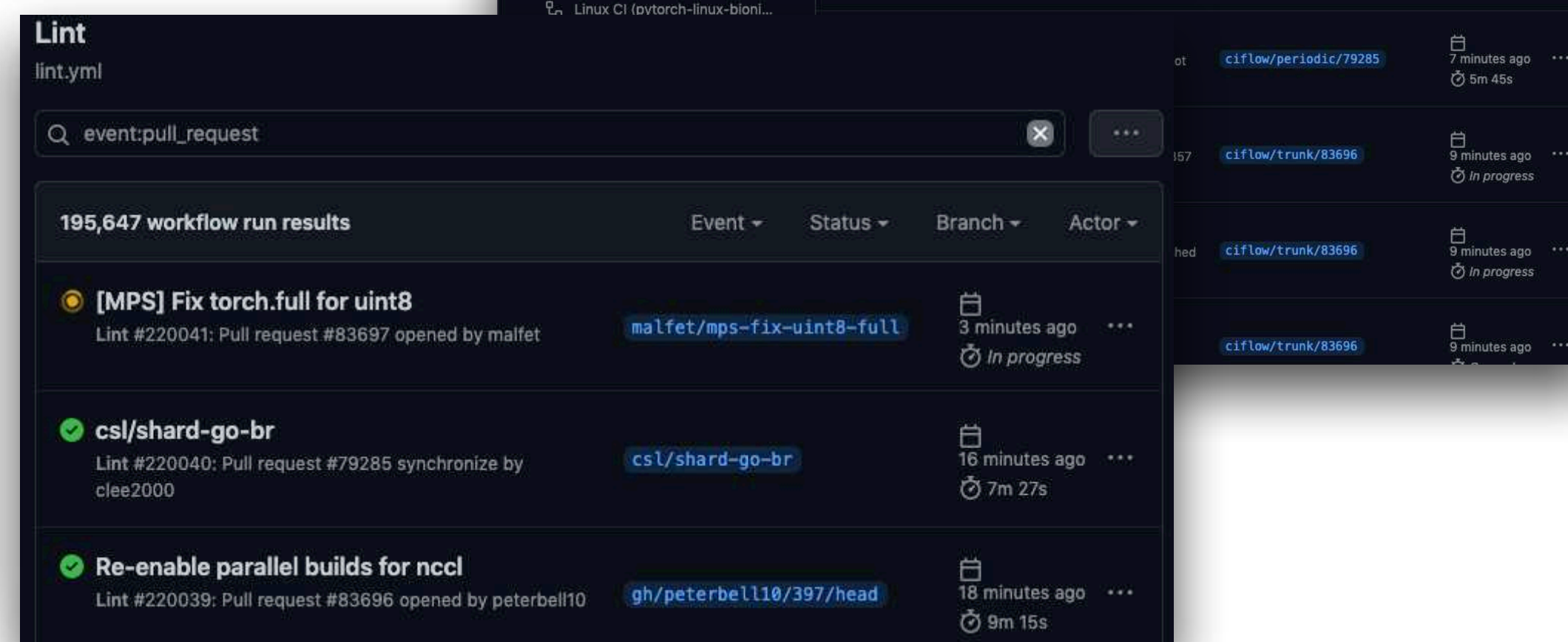
Popular OSS repos have best practices in place.

Code-related tasks
up to 10 min on push/pull.

Other tasks and any tasks
>10 min on a schedule.



✓ Close stale pull requests Close stale pull requests #3081: Scheduled	1 hour ago 42s	...
✓ Update viable/strict Update viable/strict #2913: Scheduled	2 hours ago 1m 44s	...
✓ Update S3 HTML indices for download.pyt... Update S3 HTML indices for download.pytorch.org #27534: Scheduled	2 hours ago 11m 11s	...
🕒 periodic periodic #2150: Scheduled	2 hours ago In progress	...



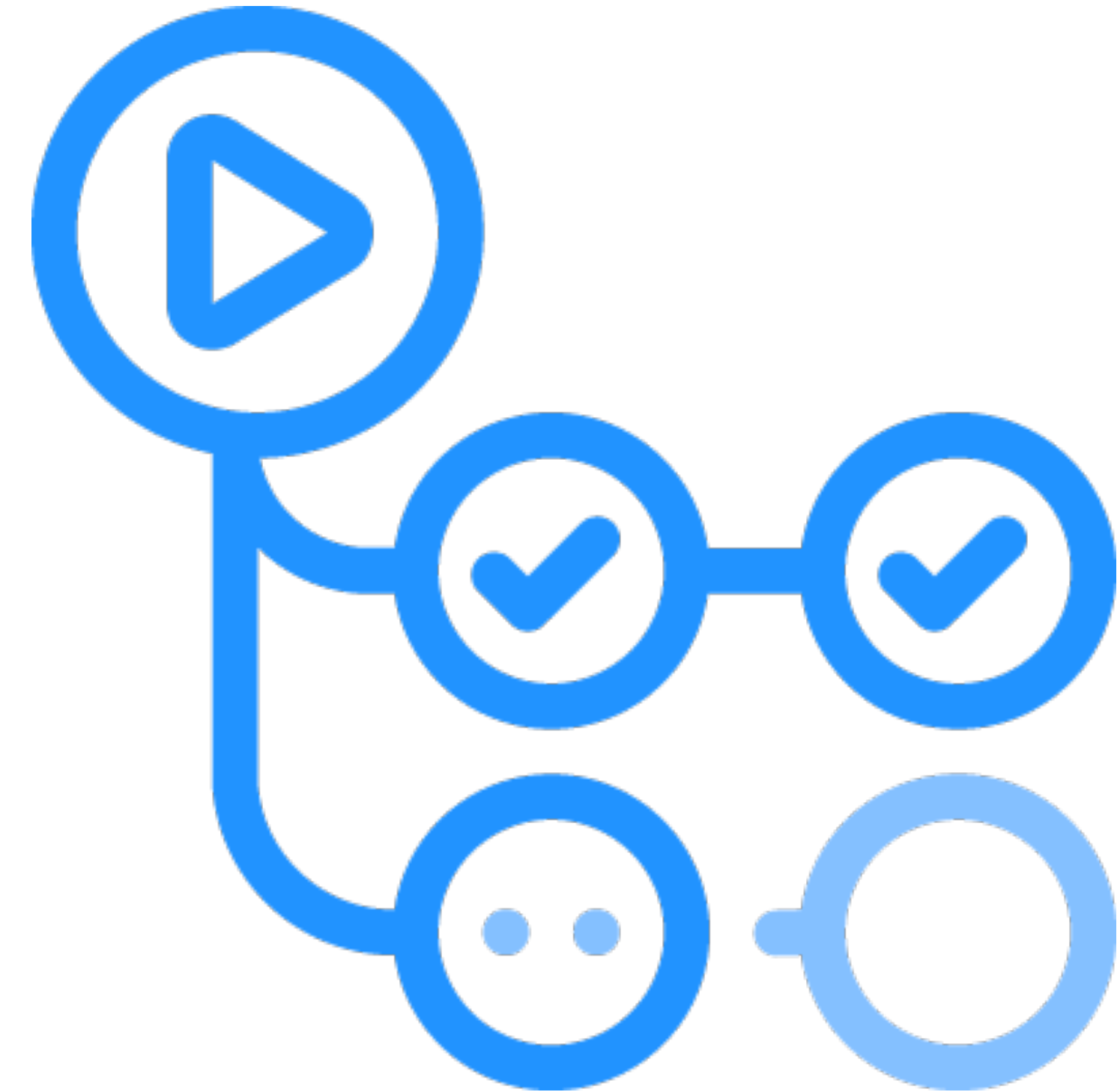
Lint lint.yml			
event:pull_request			
195,647 workflow run results			
🕒 [MPS] Fix torch.full for uint8 Lint #220041: Pull request #83697 opened by malfet	malfet/mps-fix-uint8-full	3 minutes ago	...
✓ cs/shard-go-br Lint #220040: Pull request #79285 synchronize by cle2000	cs/shard-go-br	16 minutes ago	...
✓ Re-enable parallel builds for nccl Lint #220039: Pull request #83696 opened by peterbell10	gh/peterbell10/397/head	18 minutes ago	...

We recommend GitHub Actions for general automation.

- Ties automation directly to VCS
- Powerful, flexible, performant
- Easy to use
 - Well-documented
 - "Just" a YAML file
 - Large open source community

Other options:

- pre-commit.ci, CircleCI, Jenkins



"This could've been a GitHub Action"
is the new "this could've been an email".

We recommend `pre-commit` to enforce hygiene checks.

- Use it to run formatting, linting, etc. on every commit
- Keep total runtime to a few seconds or you'll discourage commits
- Easy to run locally
- Easy to automate with GitHub Actions
- Separates linting tool env from development env

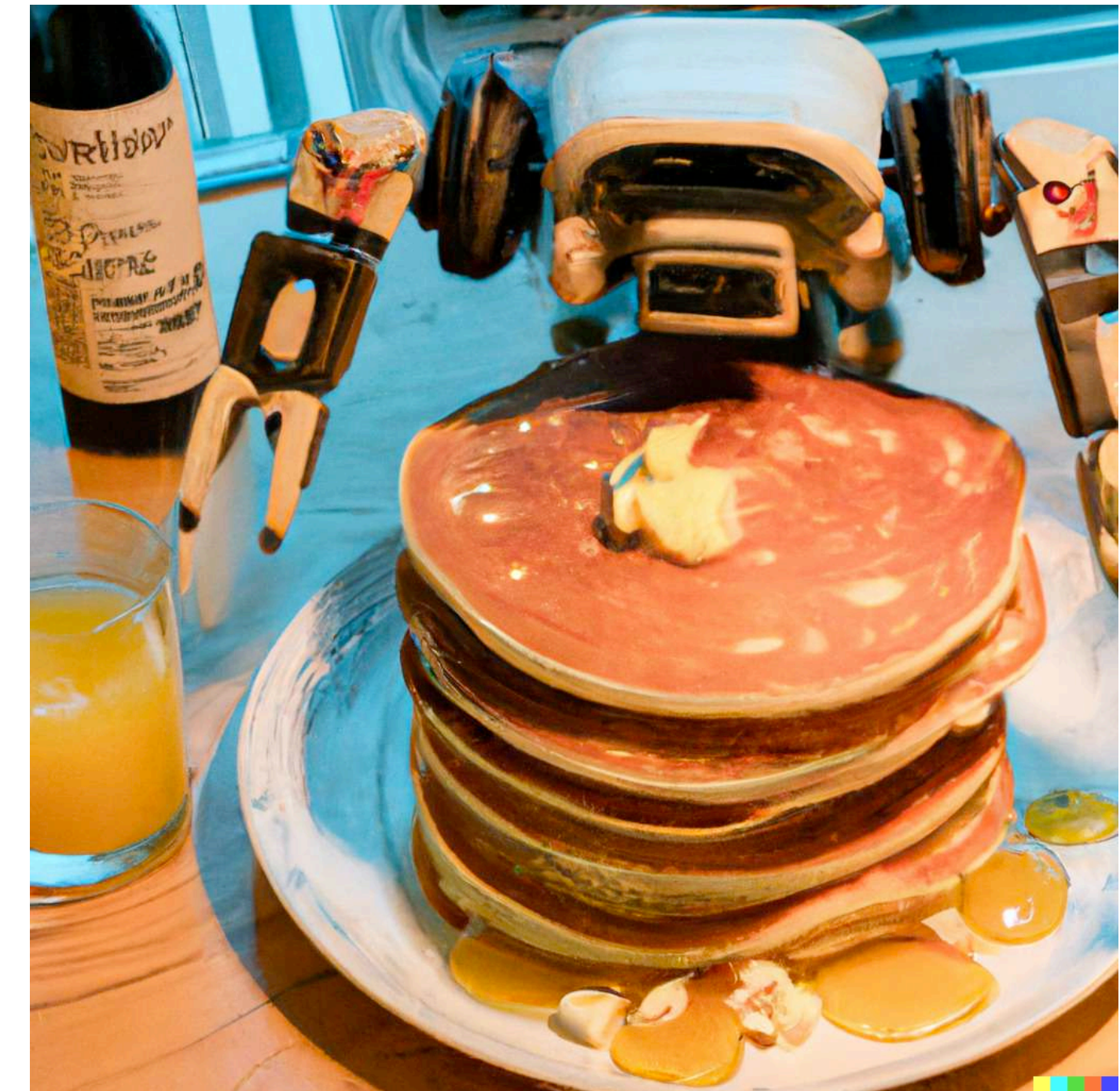
```
[~/fsdl/fsdl-text-recognizer-2022]$ pre-commit run --all
trim trailing whitespace.....Passed
check toml.....Passed
check yaml.....Passed
check json.....Passed
check for merge conflicts.....Passed
check for added large files.....Passed
debug statements (python).....Passed
detect private key.....Passed
black.....Passed
flake8.....Passed
shellcheck.....Passed
mypy (training).....Passed
mypy (text_recognizer).....Passed
```



Automation is a productivity enhancer.

Automation:

- avoids context switching.
- surfaces issues early.
- is a force multiplier for small teams.
- is better documented by default.



*"What is my purpose?"
"You butter the pancakes."*

This is broader than just "CI/CD".



Caveat: automation requires *really* knowing your tools.

Knowing Docker well enough to use it is not the same
as knowing Docker well enough to automate it.

Bad automation, like bad tests, takes more time than it saves.

Organizationally, they're a good task for senior engineers.

Summary

- Automate tasks with GitHub Actions to reduce friction.
- Use the standard Python toolkit for testing and cleaning your projects.
- Choose testing & linting practices with the 80/20 principle, velocity, and usability in mind.

01

Testing ML systems



Key points in this section

- Testing ML is hard, but not impossible.
- Stick with the low-hanging fruit to start:
 - expectation tests for data,
 - memorization tests for training,
 - regression tests for models.
- Test in production, but don't YOLO.

 FSDL Pancakes of Approval 

 The ML Test Score

 Great Expectations

Testing ML is hard,
but not impossible.

In **SWE**, we compile source code into programs.



In **ML**, training "compiles" data into model.



All these components are harder to test.

ML is the *Dark Souls* of software testing.



YOU DIED

But *Souls* games can be beaten.



“A player in Elden Ring named ‘Let me solo her’ has been soloing Malenia, one of the game's hardest bosses for players with nothing but a jar on his head and 2 katanas, and often without taking any damage. Recently he's been worshiped and has had art and figures made in his honor.”

We'll focus mostly on "smoke" tests.



IT'S NOT FINE, AND YOU HAVE
A RESPONSIBILITY TO ACT.

 @arisupaints

These are easy to implement and still effective.

Use expectation testing on data.

To start: test data by checking basic properties.

- Check that simple properties hold: express "expectations".
- Start small and grow slowly.
- Only test things that are worth raising alarms over:
 - "Heights are between 4 and 8 feet"
 - ➡ "between 2 and 10 feet" or "not negative and less than 30 feet"
 - "Exactly these columns are present"
 - ➡ "At least" or "At most"

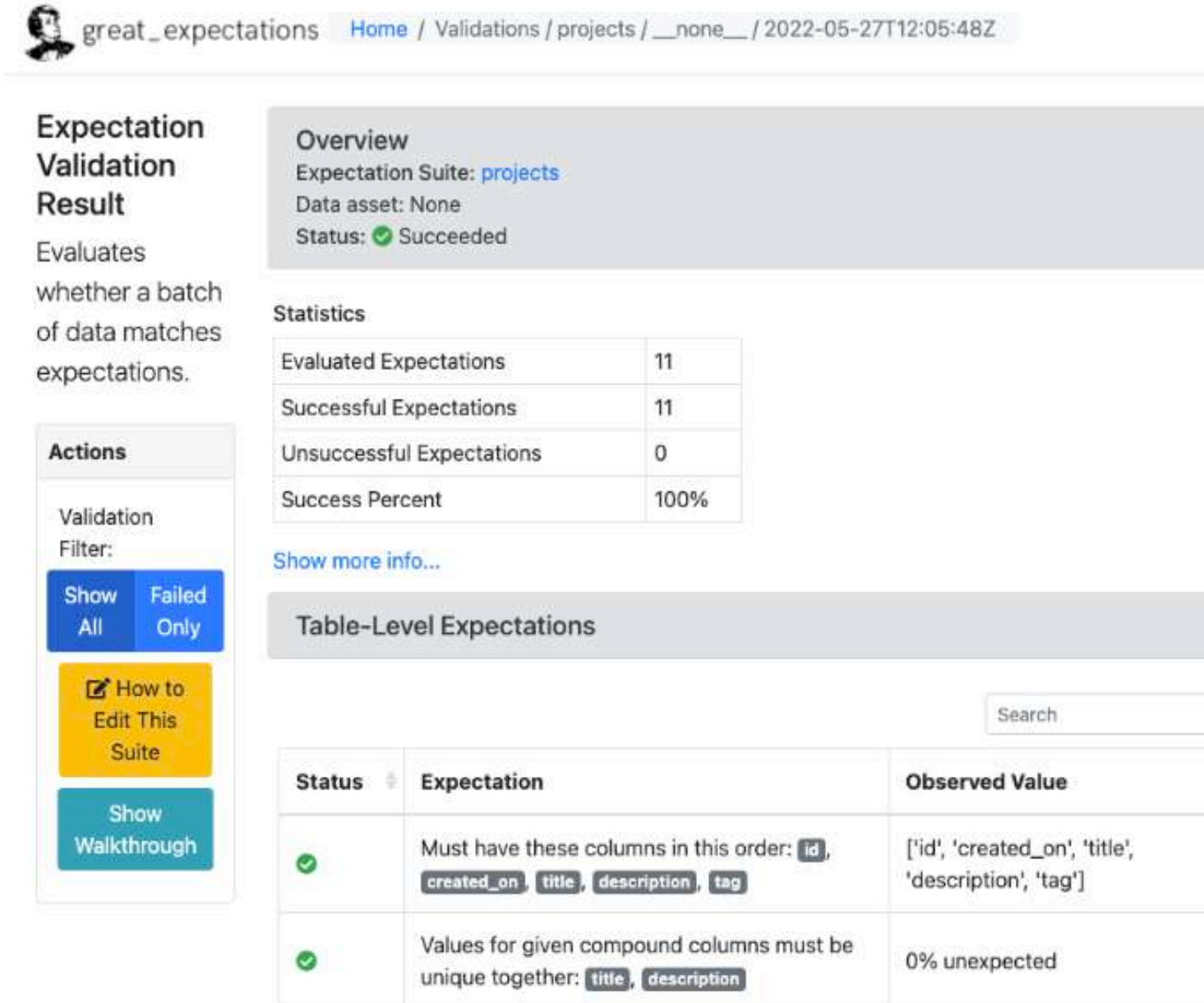
We recommend `great_expectations` for data testing.



We recommend the MadeWithML tutorial.

Documentation

When we create expectations using the CLI application, Great Expectations automatically generates documentation for our tests. It also stores information about validation runs and their results. We can launch the generate data documentation with the following command: `great_expectations docs build`



The screenshot shows the Great Expectations documentation interface. At the top, it says "great_expectations Home / Validations / projects / __none__ / 2022-05-27T12:05:48Z". The main section is titled "Expectation Validation Result" and states "Evaluates whether a batch of data matches expectations." Below this, there are "Actions" like "Validation Filter:", "Show All", "Failed Only", "How to Edit This Suite", and "Show Walkthrough". The "Overview" section shows "Expectation Suite: projects", "Data asset: None", and "Status: Succeeded". A "Statistics" table shows 11 evaluated expectations, all successful, with a 100% success rate. Below this is a "Table-Level Expectations" section with a search bar and a table of expectations.

Status	Expectation	Observed Value
✓	Must have these columns in this order: <code>id</code> , <code>created_on</code> , <code>title</code> , <code>description</code> , <code>tag</code>	['id', 'created_on', 'title', 'description', 'tag']
✓	Values for given compound columns must be unique together: <code>title</code> , <code>description</code>	0% unexpected

Expectations

When it comes to creating expectations as to what our data should look like, we want to think about our entire dataset and all the features (columns) within it.

```
# Presence of specific features
df.expect_table_columns_to_match_ordered_list(
    column_list=["id", "created_on", "title", "description", "tag"]
)

# Unique combinations of features (detect data leaks!)
df.expect_compound_columns_to_be_unique(column_list=["title", "description"])

# Missing values
df.expect_column_values_to_not_be_null(column="tag")

# Unique values
df.expect_column_values_to_be_unique(column="id")

# Type adherence
df.expect_column_values_to_be_of_type(column="title", type_="str")

# List (categorical) / range (continuous) of allowed values
tags = ["computer-vision", "graph-learning", "reinforcement-learning",
        "natural-language-processing", "mlops", "time-series"]
df.expect_column_values_to_be_in_set(column="tag", value_set=tags)
```


Moving forward: stay close to your data.



Benchmark dataset or external annotation team only.

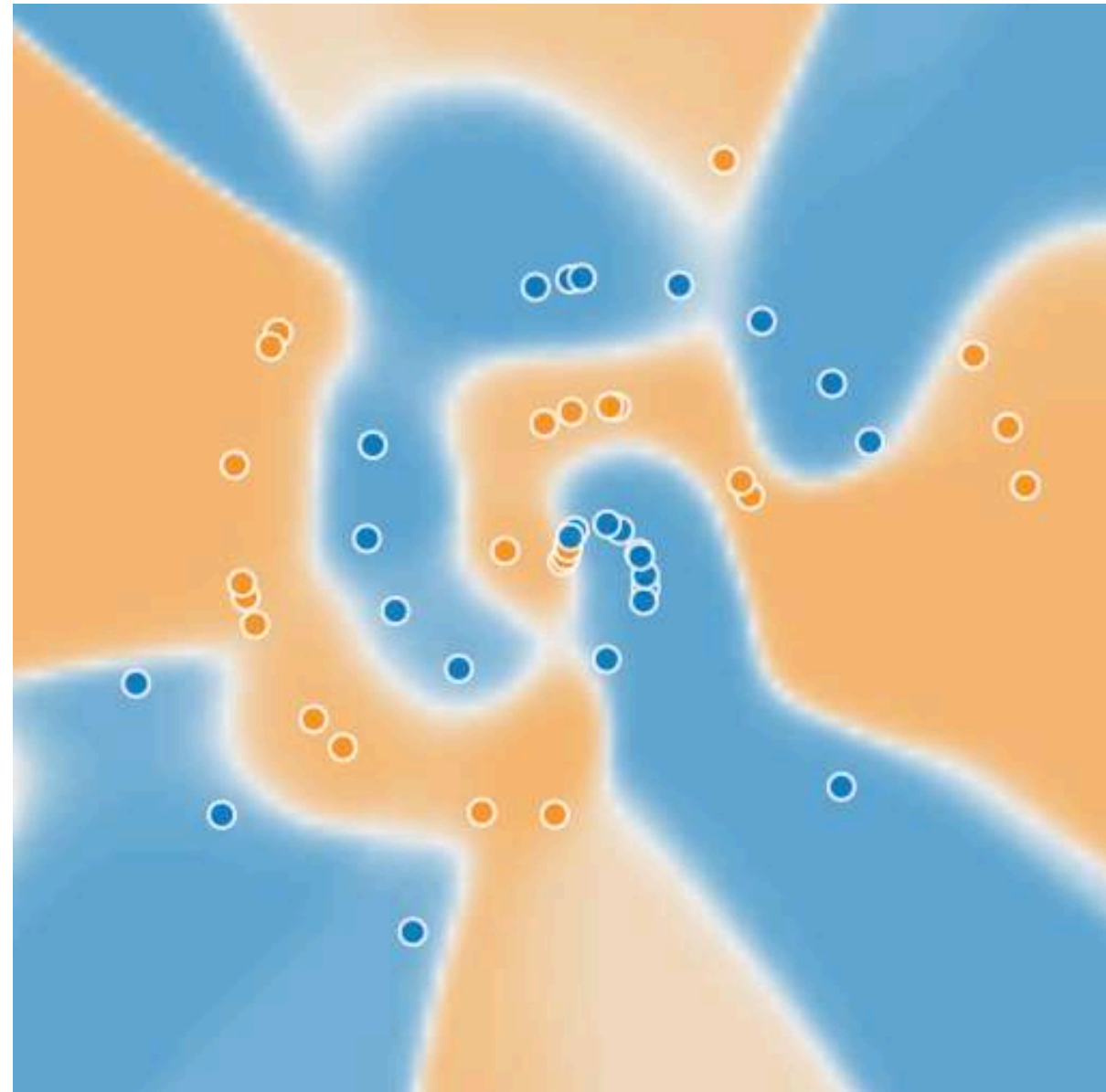
Model devs annotate data ad hoc at the start of the project.

Internal annotation team. Regular information flow with model devs.

Regular on-call rotation where model devs annotate data.

Use memorization testing on training.

As a smoke test of training, confirm memorization.



Train the model to "memorize" a very **small fraction** of the **full dataset**.

Memorization is the simplest form of learning.

If the model can't memorize, something is very wrong.

- Only gross issues with training will show up.
 - Shuffled labels, busted gradients, (some) numerical issues
- Incorporate these tests into end-to-end model deployment testing.
- Include runtime in test, because regressions there can mean bugs

 `--overfit_batches`

Tune memorization tests to run quickly.

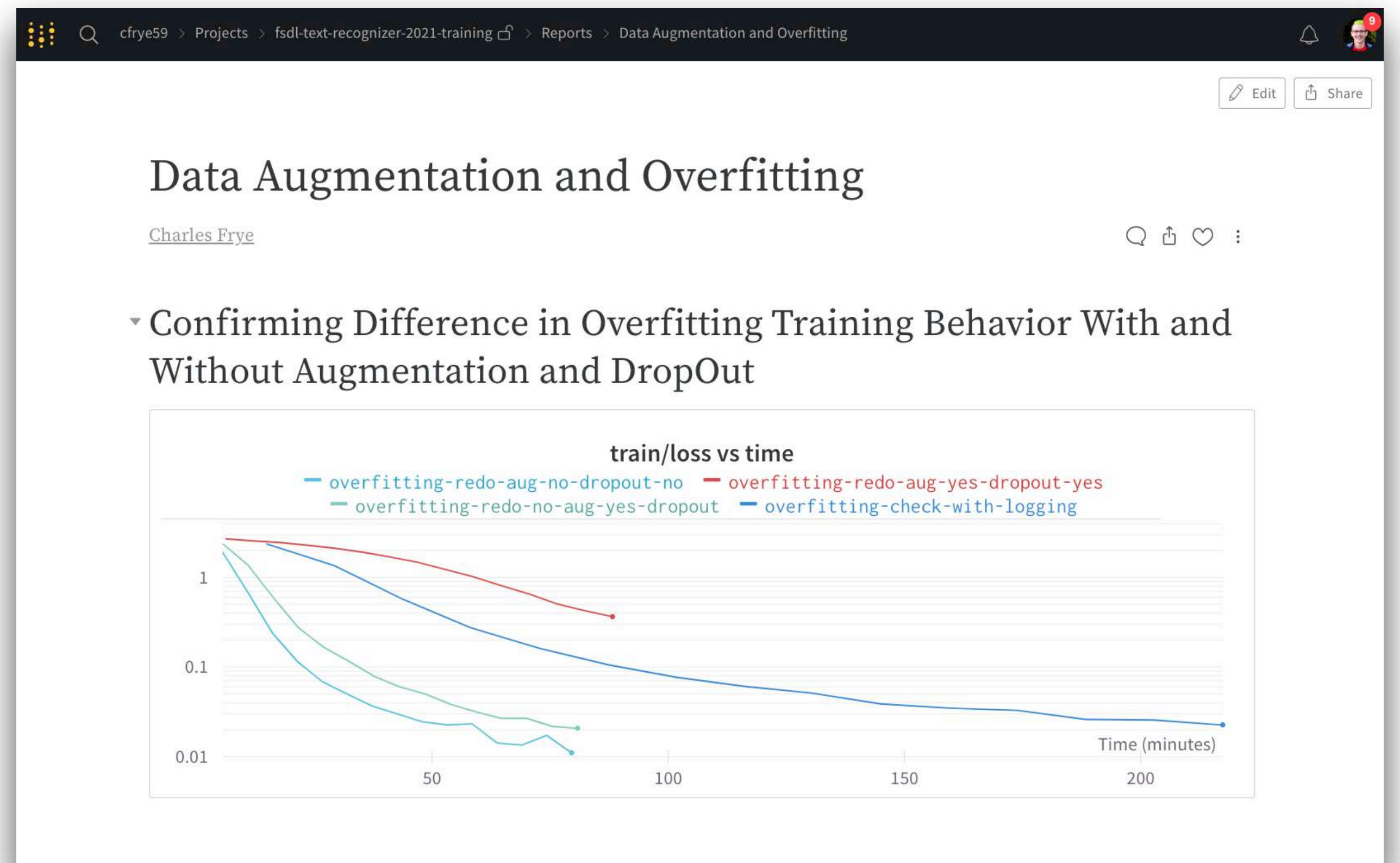
Buy faster machines

Reduce dataset size

Turn off regularization

Reduce model size

Remove expensive components



Moving forward: rerun old training jobs with new code?

- Run nightly or weekly, not on push!
- Puts you on the hook for backwards compatibility
- Expensive, no matter how you do it:
 - CircleCI has GPU runners, but only in the enterprise-level plan, which is **>\$24k a year**.
 - GitHub Actions for now requires you to self-host, meaning you're about doubling training spend.

CLOUD

Scale

Enterprise-level confidence and support for teams who deliver.

Starting at

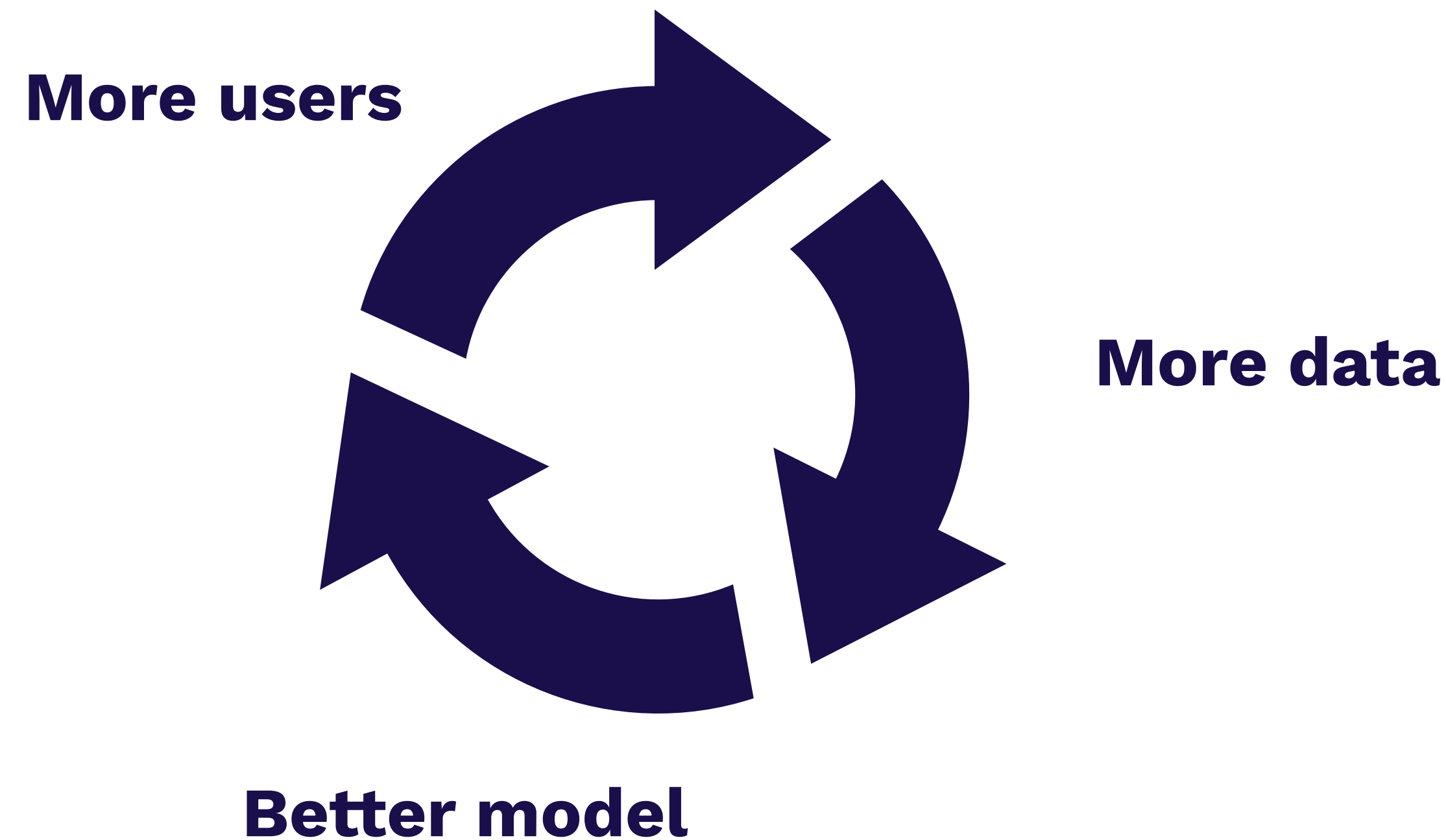
\$2,000 per month

[Contact Us](#)

✓ **Power:** Our largest resource classes for complex processes and speed

✓ **Flexibility:** All the environments in Performance, plus access to GPU resource classes

Better: regularly train with new data.



- Still expensive, but now you're spending on model dev, not testing.
- It also requires **data flywheel** and prod monitoring tools.
- We'll return to this in monitoring and continual learning lecture.

Adapt regression testing for models.

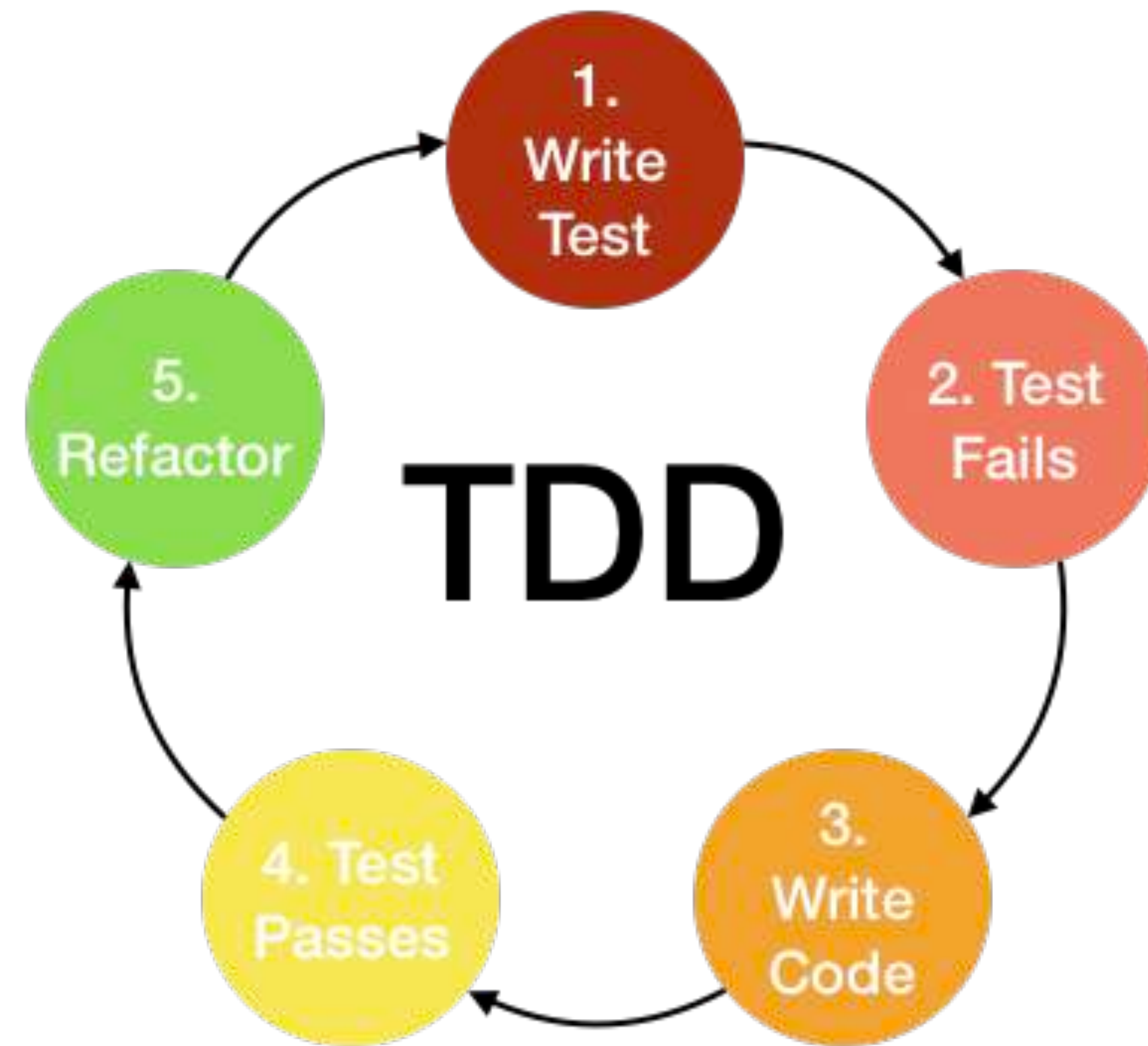
Models are functions, so test them like functions.

```
def _test_paragraph_text_recognizer(image_filename, expected_text, text_recognizer):  
    """Test ParagraphTextRecognizer on 1 image."""  
  
    predicted_text = text_recognizer.predict(image_filename)  
  
    error_msg = f"predicted text does not match expected for {image_filename.name}"  
    assert predicted_text == expected_text, error_msg
```

- Easiest for classification and other tasks with simple output
- Even for structured outputs, you can use these to check for differences between model-in-training and model-in-prod.

Use the loss to build documented
regression test suites out of your data.

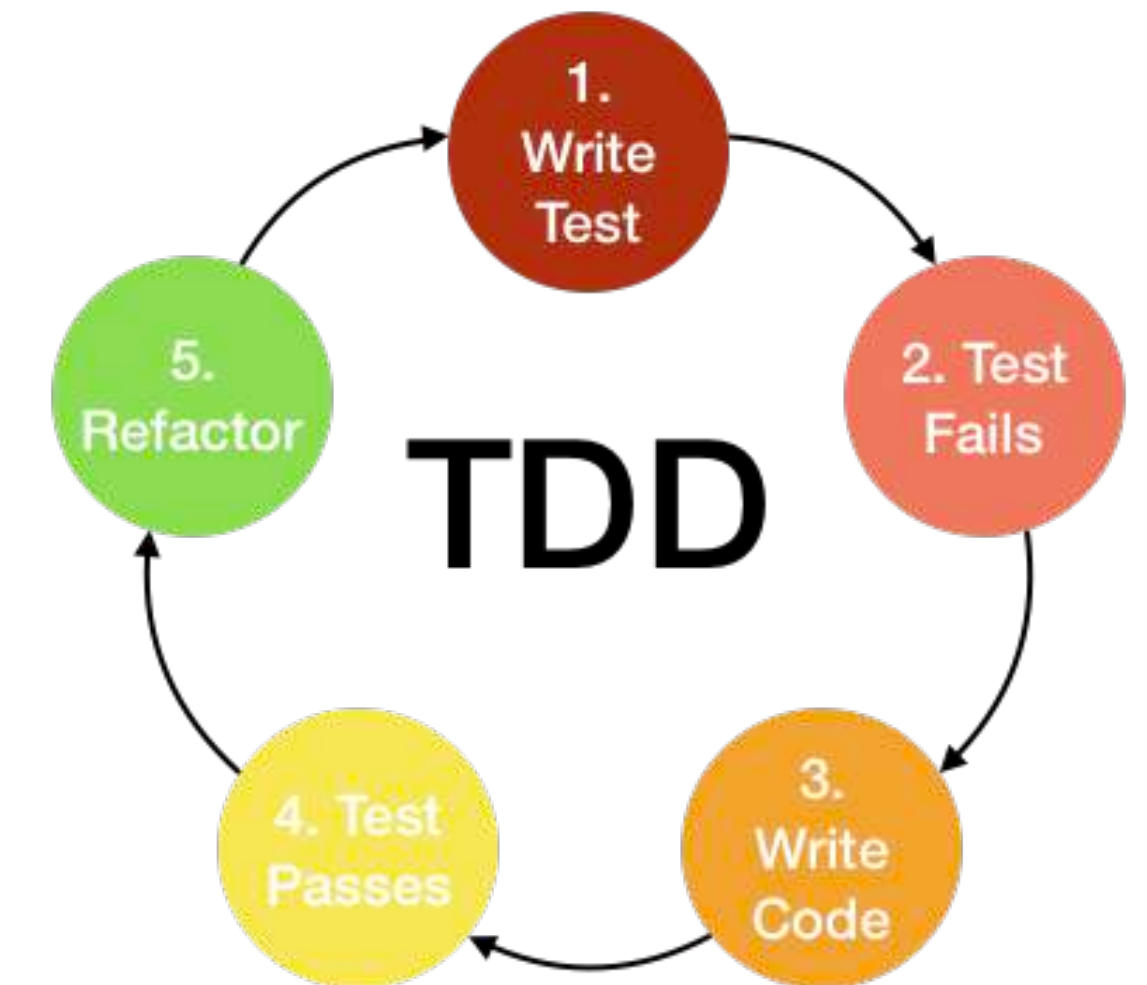
Test-driven development is a paradigm for testing.



We can incorporate it into testing models.

Think of the loss as a "fuzzy" test signal.

- Treat the loss as "how **badly this test was failed**".
- During training, **our model changes** to do **better on the test**.
- Model stops changing once it passes the tests well enough.



Conclusion: gradient descent is test-driven development.

Collect up high-loss examples.

- Find the data points with highest loss and put them in a suite labeled "hard".
- But note that problem could be in the model or it could be in the data!



Fig. 1. Mislabeled instances in the CIFAR-100 training set, with corresponding label and index of the image in the data set.

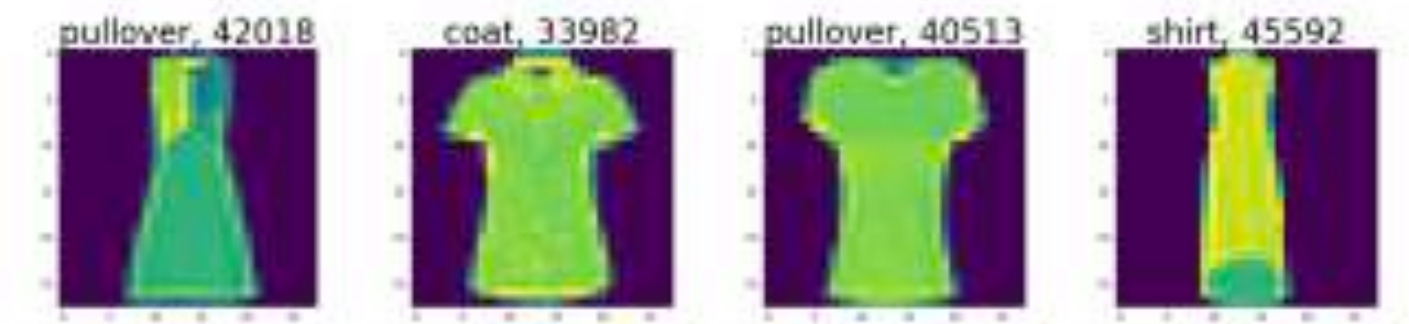


Fig. 2. Mislabeled instances in the Fashion-MNIST training set.

<https://arxiv.org/abs/1912.05283>

Aggregate individual failures into named suites.

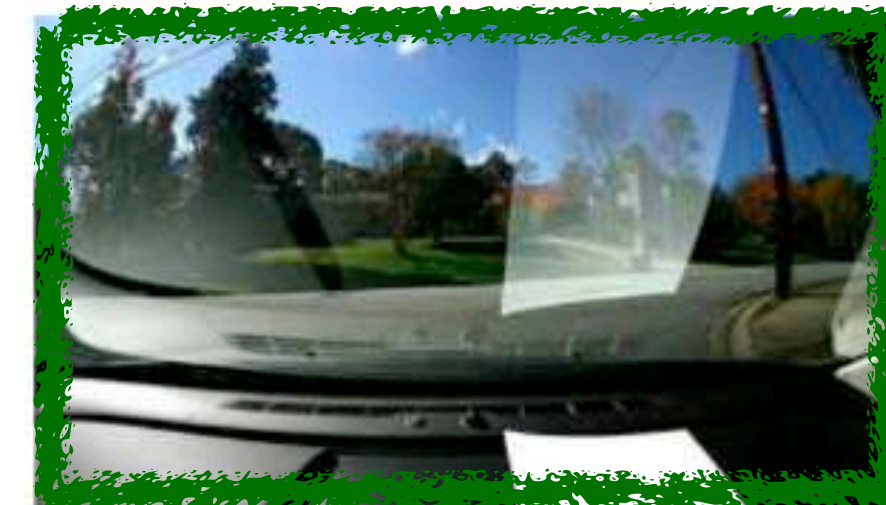
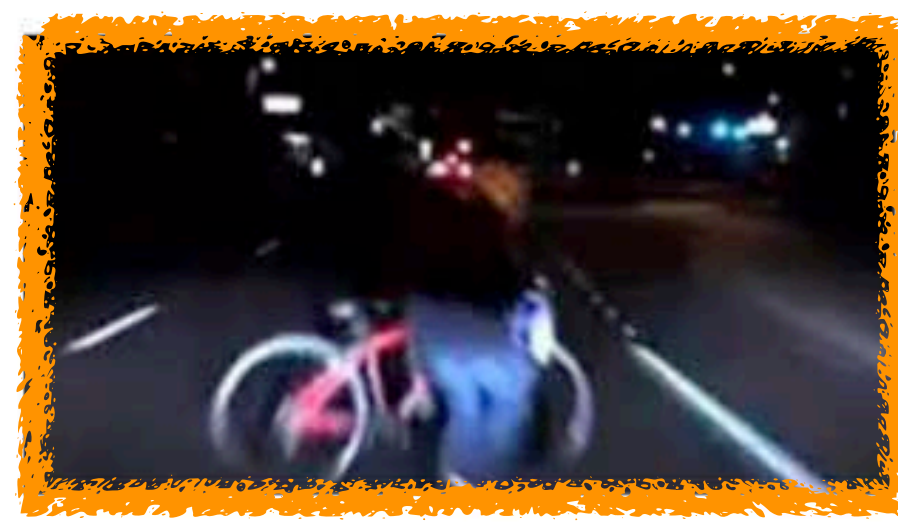
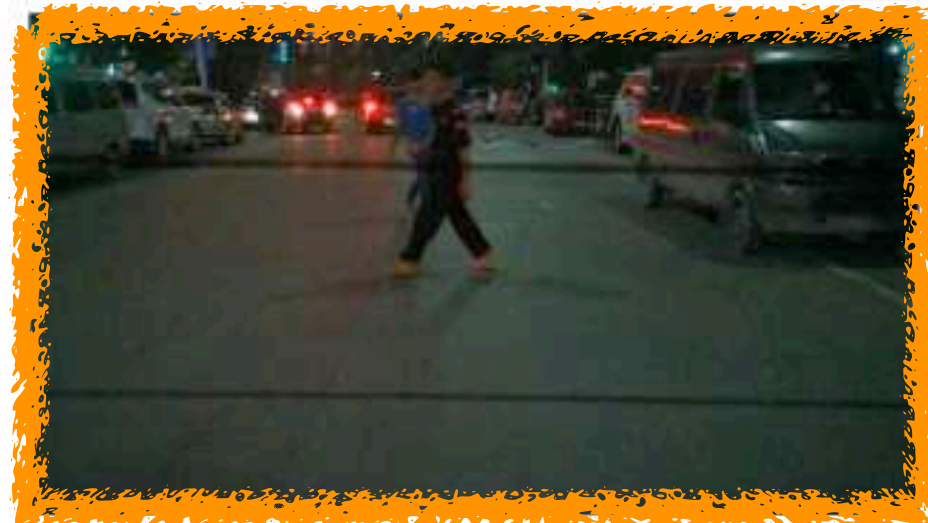
Test-val set errors (no pedestrian detected)



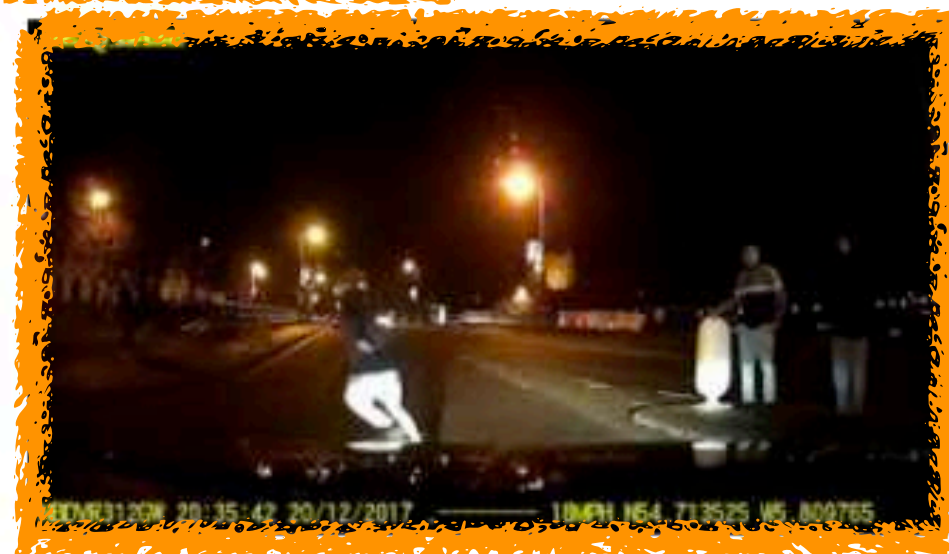
Train-val set errors (no pedestrian detected)



Shadows



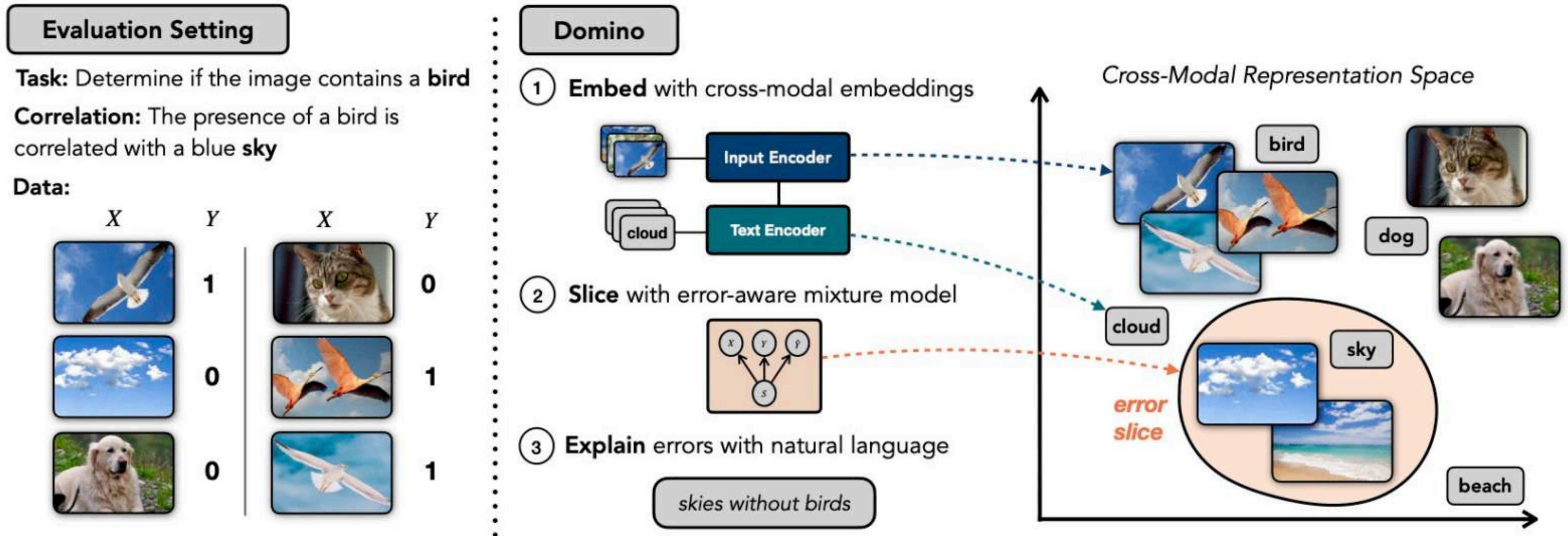
Reflections



Nighttime

This is the ML version of "regression tests".

Automate discovery with Domino.



<https://arxiv.org/abs/2203.14960>

For more ways to test models, see CheckList.

Capability	Min Func Test	INVariance	DIRectional
Vocabulary	Fail. rate=15.0%	16.2%	C 34.6%
NER	0.0%	B 20.8%	N/A
Negation	A 76.4%	N/A	N/A
...			

Test case	Expected	Predicted	Pass?
A Testing Negation with MFT Labels: negative, positive, neutral Template: I {NEGATION} {POS_VERB} the {THING}.			
I can't say I recommend the food.	neg	pos	x
I didn't love the flight.	neg	neutral	x
...			
Failure rate = 76.4%			
B Testing NER with INV Same pred. (inv) after removals / additions			
@AmericanAir thank you we got on a different flight to [Chicago → Dallas].	inv	pos neutral	x
@VirginAmerica I can't lose my luggage, moving to [Brazil → Turkey] soon, ugh.	inv	neutral neg	x
...			
Failure rate = 20.8%			
C Testing Vocabulary with DIR Sentiment monotonic decreasing (↓)			
@AmericanAir service wasn't great. You are lame.	↓	neg neutral	x
@JetBlue why won't YOU help them?! Ugh. I dread you.	↓	neg neutral	x
...			
Failure rate = 34.6%			

Test in production,
but don't YOLO.

Especially for ML models, production environments differ from development.



Charity Majors ✓
@mipsytipsy

There is no, has never been, and will never be any environment that looks just like prod.

🌸 Only production is production. 🌸
Test in prod or live a lie.



Steven Bower @raggylugthumps · Sep 18

@mipsytipsy Hard disagree.. the question should be, “how do o create environments that look just like prod so we can test there”.. and try to tell this to any regulator..

10:37 PM · Sep 22, 2020 · Twitter Web App

The solution is to run tests in production.

Test-in-prod means monitoring for errors in prod and fixing them quickly.



Chip Huyen
@chipro

Deploying ML systems isn't just about getting ML systems to the end-users.

It's about building an infrastructure so the team can be quickly alerted when something goes wrong, figure out what went wrong, test in production, roll-out/rollback updates.

It's fun!

(6/6)

7:40 AM · Sep 29, 2020 · Twitter Web App

It reduces pressure on other kinds of testing,
but does not replace them.



Safely testing in prod requires tooling to monitor prod, & much of that tooling is new.



We'll cover it along with monitoring & continual learning.

Summary

- Focus on low-hanging fruit and "smoke" tests for ML.
 - Expectation tests for data
 - Memorization tests for training
 - Regression tests for models

Eventually, mature into
(something like) the ML Test Score.

The ML Test Score is a strict rubric for ML test quality.

1	Feature expectations are captured in a schema.
2	All features are beneficial.
3	No feature's cost is too much.
4	Features adhere to meta-level requirements.
5	The data pipeline has appropriate privacy controls.
6	New features can be added quickly.
7	All input feature code is tested.

Table I
BRIEF LISTING OF THE SEVEN DATA TESTS.

1	Model specs are reviewed and submitted.
2	Offline and online metrics correlate.
3	All hyperparameters have been tuned.
4	The impact of model staleness is known.
5	A simpler model is not better.
6	Model quality is sufficient on important data slices.
7	The model is tested for considerations of inclusion.

Table II
BRIEF LISTING OF THE SEVEN MODEL TESTS

1	Training is reproducible.
2	Model specs are unit tested.
3	The ML pipeline is Integration tested.
4	Model quality is validated before serving.
5	The model is debuggable.
6	Models are canaried before serving.
7	Serving models can be rolled back.

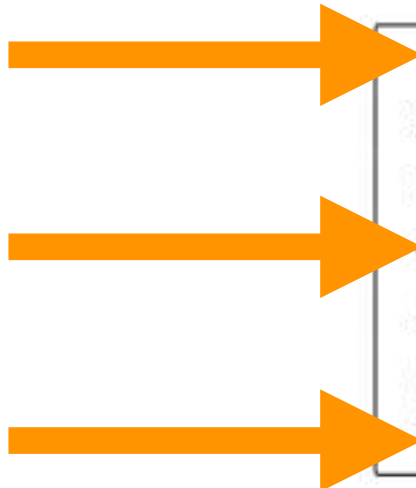
Table III
BRIEF LISTING OF THE ML INFRASTRUCTURE TESTS

1	Dependency changes result in notification.
2	Data invariants hold for inputs.
3	Training and serving are not skewed.
4	Models are not too stale.
5	Models are numerically stable.
6	Computing performance has not regressed.
7	Prediction quality has not regressed.

Table IV
BRIEF LISTING OF THE SEVEN MONITORING TESTS

<https://research.google/pubs/pub46555/>

Our recommendations overlap with it.



1	Feature expectations are captured in a schema.
2	All features are beneficial.
3	No feature's cost is too much.
4	Features adhere to meta-level requirements.
5	The data pipeline has appropriate privacy controls.
6	New features can be added quickly.
7	All input feature code is tested.

Table I
BRIEF LISTING OF THE SEVEN DATA TESTS.

1	Model specs are reviewed and submitted.
2	Offline and online metrics correlate.
3	All hyperparameters have been tuned.
4	The impact of model staleness is known.
5	A simpler model is not better.
6	Model quality is sufficient on important data slices.
7	The model is tested for considerations of inclusion.


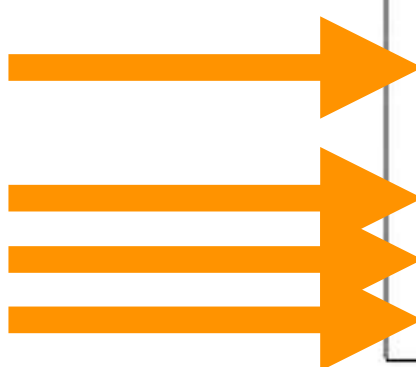


Table II
BRIEF LISTING OF THE SEVEN MODEL TESTS



1	Training is reproducible.
2	Model specs are unit tested.
3	The ML pipeline is Integration tested.
4	Model quality is validated before serving.
5	The model is debuggable.
6	Models are canaried before serving.
7	Serving models can be rolled back.

Table III
BRIEF LISTING OF THE ML INFRASTRUCTURE TESTS

1	Dependency changes result in notification.
2	Data invariants hold for inputs.
3	Training and serving are not skewed.
4	Models are not too stale.
5	Models are numerically stable.
6	Computing performance has not regressed.
7	Prediction quality has not regressed.

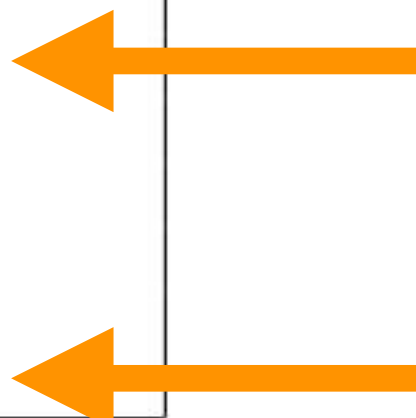





Table IV
BRIEF LISTING OF THE SEVEN MONITORING TESTS







<https://research.google/pubs/pub46555/>

How does our Text Recognizer do?

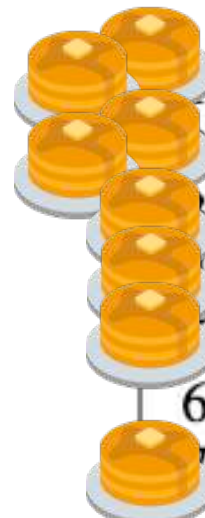
1	Feature expectations are captured in a schema.
2	All features are beneficial.
3	No feature's cost is too much.
4	Features adhere to meta-level requirements.
5	The data pipeline has appropriate privacy controls.
6	New features can be added quickly.
7	All input feature code is tested.

Table I
BRIEF LISTING OF THE SEVEN DATA TESTS.

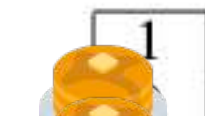



1	Model specs are reviewed and submitted.
2	Offline and online metrics correlate.
3	All hyperparameters have been tuned.
4	The impact of model staleness is known.
5	A simpler model is not better.
6	Model quality is sufficient on important data slices.
7	The model is tested for considerations of inclusion.

Table II
BRIEF LISTING OF THE SEVEN MODEL TESTS



1	Training is reproducible.
2	Model specs are unit tested.
3	The ML pipeline is Integration tested.
4	Model quality is validated before serving.
5	The model is debuggable.
6	Models are canaried before serving.
7	Serving models can be rolled back.

Table III
BRIEF LISTING OF THE ML INFRASTRUCTURE TESTS

1	Dependency changes result in notification.
2	Data invariants hold for inputs.
3	Training and serving are not skewed.
4	Models are not too stale.
5	Models are numerically stable.
6	Computing performance has not regressed.
7	Prediction quality has not regressed.

Table IV
BRIEF LISTING OF THE SEVEN MONITORING TESTS

 Yes, manual & discoverable
 Yes, automated

 Top priorities

How does our Text Recognizer do?

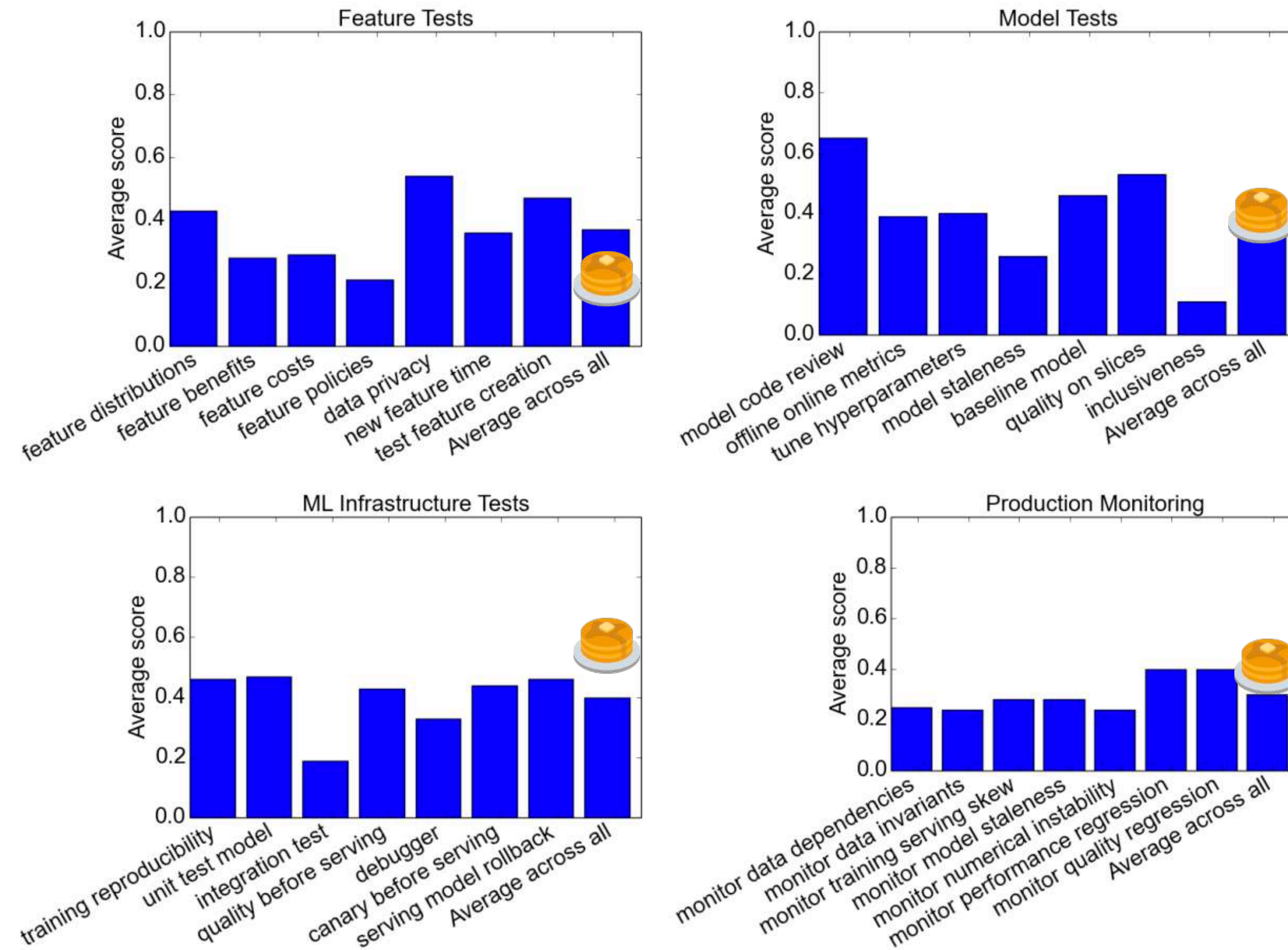


Figure 4. **Average scores for interviewed teams.** These graphs display the average score for each test across the 36 systems we examined.

02

Troubleshooting models



Key points in this section

- "Make it run" by avoiding common errors: shapes, OOM, numerics.
- "Make it fast" by profiling and removing bottlenecks.
- "Make it right" by scaling model/data and sticking with proven architectures.

Make it run.

Only a small portion of ML bugs cause loud failure.

- Shape errors: Tensors don't match.
- Out-of-memory errors: Tensors don't fit on GPU.
- Numerical errors: Tensors have NaNs or $\pm\text{infs}$.

Shape errors when tensors don't match.

- Step through in a debugger (easier in PyTorch)
- Annotate shapes in code

```
_B, C, _H, _W = x.shape
if C == 1:
    x = x.repeat(1, 3, 1, 1)
x = self.resnet(x) # (B, RESNET_DIM, _H // 32, _W // 32),
x = self.encoder_projection(x) # (B, E, _H // 32, _W // 32),
```

OOM errors when tensors don't fit on GPUs.

- ⚡ `--precision=16`
- ⚡ `--auto_scale_batch_size`
- ⚡ `--accumulate_grad_batches`
- Tensor parallelism
- Gradient checkpointing



This is currently the bane of my existence:
"RuntimeError: CUDA out of memory. Tried to allocate 220.00 MiB (GPU 0; 15.78 GiB total capacity; 13.16 GiB already allocated; 201.44 MiB free; 14.44 GiB reserved in total by PyTorch)"

This must be an AI rite of passage.

5:41 PM · Jul 19, 2022 · Twitter Web App

Numerical errors when tensors go bad.

⚡ `--track_grad_norm`

⚡ `--precision=64`

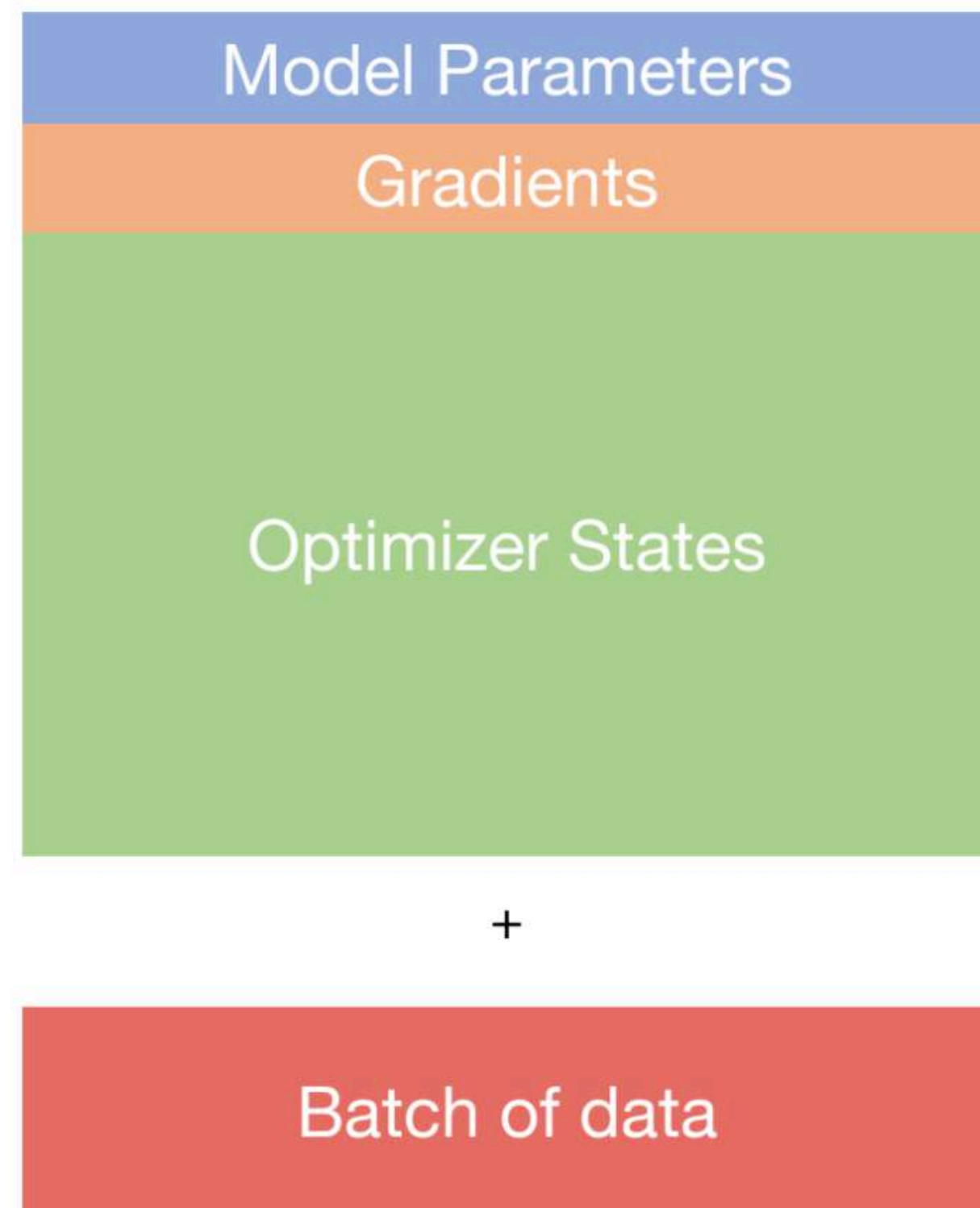
- Look into normalization tricks for your architecture



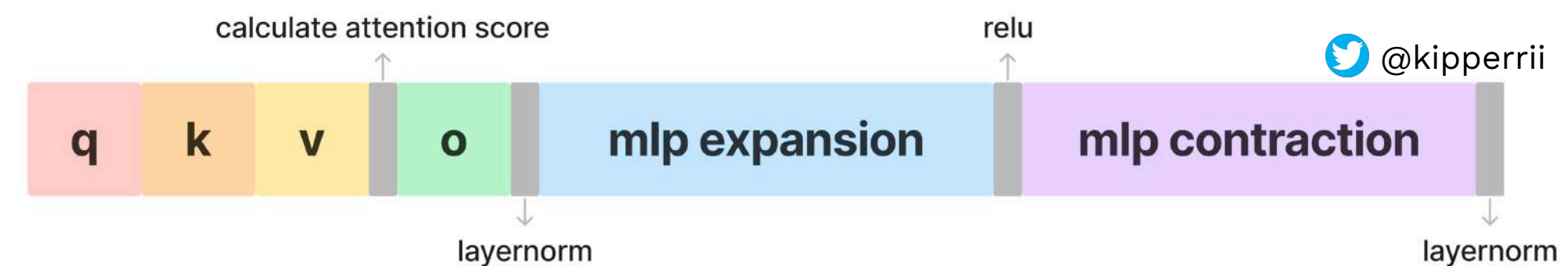
Gradient norms from DALL-E mini experiments.

Make it fast.

DNN performance is very counter-intuitive.



Popular optimizers' states take up the majority of the GPU memory budget.



Typical Transformer layers spend more time in the MLP than they do in the attention.

num_workers=0



num_workers=1



Without parallelization, data loading can easily dwarf model forward, backward, and parameter update.

So roll up your sleeves and dig in.

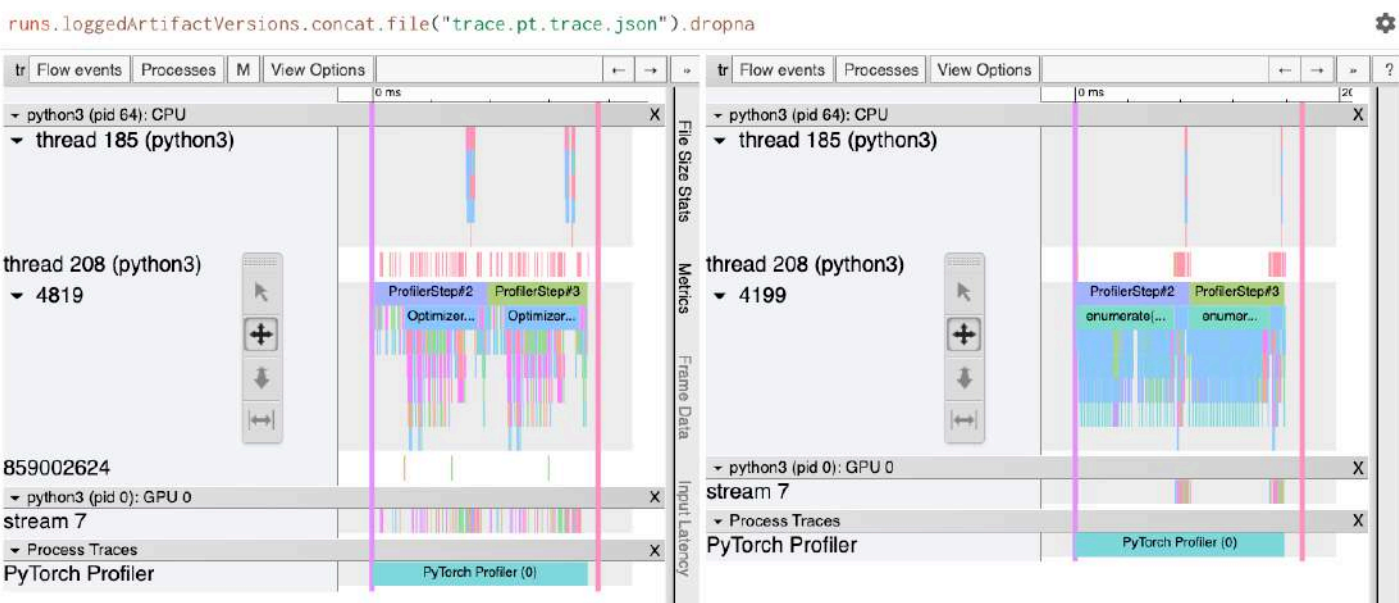
A Public Dissection of a PyTorch Training Step

What really happens when you call `.forward`, `.backward`, and `.step`?

[Charles Frye](#)

🗨️ 📌 ❤️ ⋮

Last Updated: Jan 4, 2022

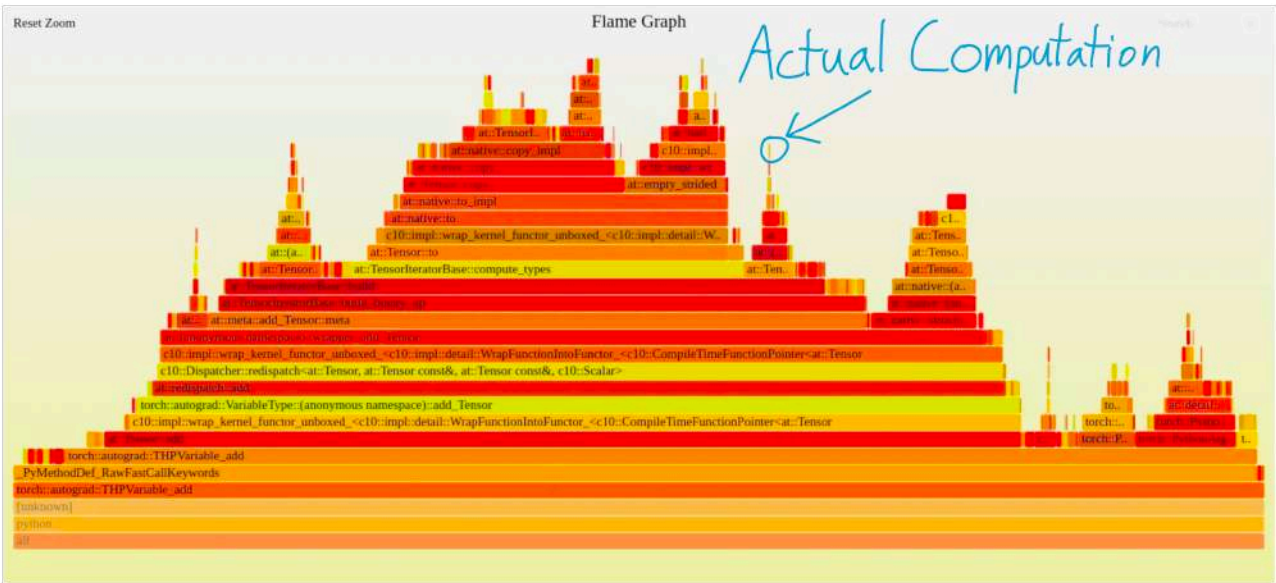
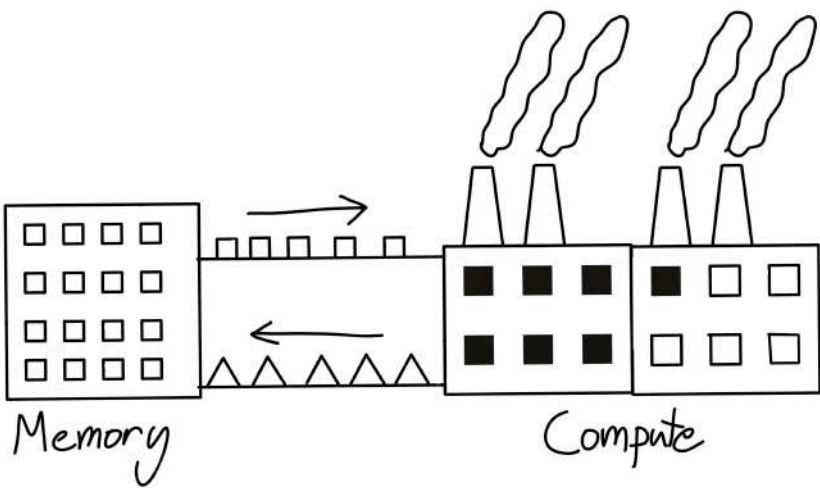


<https://wandb.me/trace-report>

Making Deep Learning Go Brrrr From First Principles

So, you want to improve the performance of your deep learning model. How might you approach such a task? Often, folk fall back to a grab-bag of tricks that might've worked before or saw on a tweet. "Use in-place operations! Set gradients to None! Install PyTorch 1.10.0 but not 1.10.1!"

It's understandable why users often take such an ad-hoc approach performance on modern systems (particularly deep learning) often feels as much like alchemy as it does science. That being said, reasoning from first principles can still eliminate broad swathes of approaches, thus making the problem much more approachable.



https://horace.io/brrr_intro.html



`--profiler`

More on profiling PyTorch code in the lab.

Make it right.

Machine learning models are always wrong.

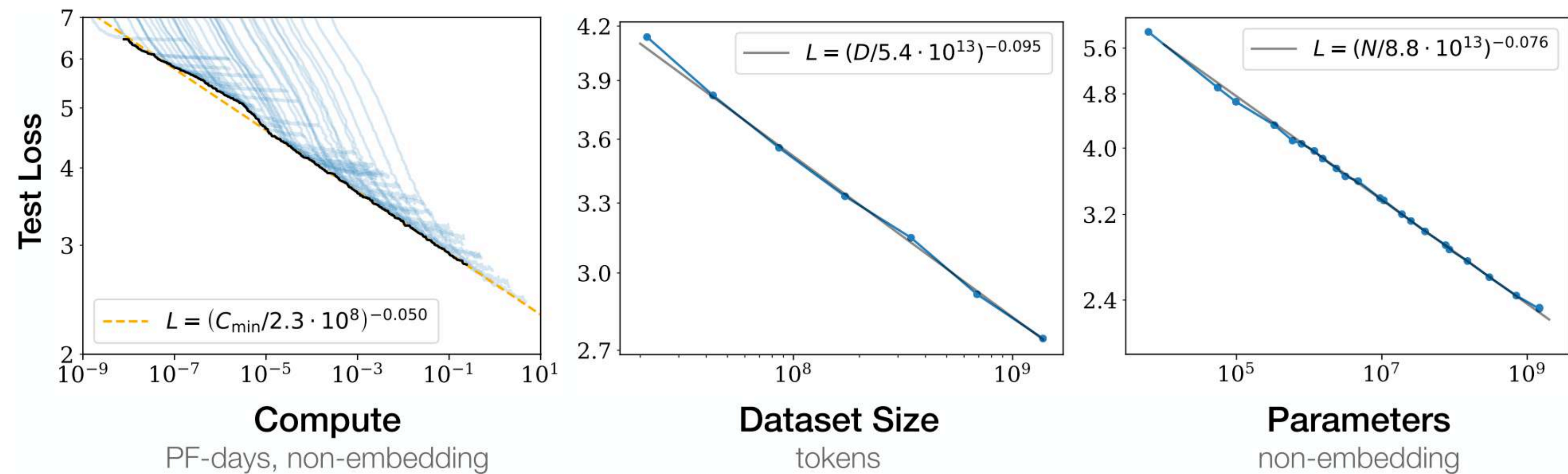
- Production performance is never perfect.
- If non-zero loss is a partial test failure, our tests are always partially failing.



So it's never possible to "make it right".

Solve all your problems with scale.

- Overfitting?
Scale up data.
- Underfitting?
Scale up model.
- Distribution shift?
Scale up both.
- Can't afford scale?
Finetune a model trained at scale.



<https://arxiv.org/abs/2001.08361>

All other advice will be model and task specific.

- Use BatchNorm
- Use LayerNorm
- Use XavierInit
- Use OrthonormalInit
- Add DropOut
- Get rid of DropOut
- Clip gradients
- Tune Adam epsilon parameter
- Use rotary embeddings
- Don't use rotary embeddings
- Use Shampoo optimizer
- LayerNorm before MLP



[huggingface_hub](#)



[Papers With Code](#)

Stick close to working architectures and hyperparameters.

Agenda

00

TESTING SOFTWARE

Standard tools & practices can
de-risk shipping at speed

01

TESTING ML SYSTEMS

Use specialized techniques
to test ML systems

02

TROUBLESHOOTING MODELS

Run fast first,
run right later

XX

Resources



Tweeters

General T&T Experts

 @b0rk

 @mipsytipsy

 @nelhage

Deep Learning T&T Experts

 @kipperrii

 @jeremyphoward

 @cHHillee

 @wightmanr

 @karpathy

 @chipro

 @handle

Templates

- <https://github.com/ashleve/lightning-hydra-template>
- <https://github.com/grok-ai/nn-template>
- <https://github.com/sudomaze/deep-learning-project-template>

Texts

- [MadeWithML course](#) on MLOps, by Goku Mohandas
 - Especially section on [testing](#)
- [Full Of Bugs blog](#), by Nelson Elhage
 - "[Computers can be understood](#)"
 - "[Systems that defy detailed understanding](#)"
 - "[Attack of the Cosmic Rays!](#)"
- *Designing ML Systems* book, by Chip Huyen
- "[ML Test Score](#)" [paper](#), by Breck et al.
- [Reliable ML Systems talk](#), by Underwood and Papasian

 @GokuMohandas

 @chipro  @nelhage



Thank you!