Code coverage

Introducing it to **your** pipeline

OVERVIEW

Meet the group

POWERPOINT ENGINEER

@landeholt

SOFTWARE ENGINEER

@personligapersson

OVERVIEW

What gets measured gets done

OVERVIEW

What gets measured gets done

- That f***ing manager

OVERVIEW

Structure

WHAT

OVERVIEW

Structure

WHAT

WHY

OVERVIEW

Structure

WHAT

WHY

THE PROBLEM

OVERVIEW

Structure

WHAT **WHY** **THE PROBLEM** **HOW**

OVERVIEW

Structure

WHAT **WHY** **THE PROBLEM** **HOW** **TAKEAWAYS**

WHAT

What is code coverage?

WHAT

Execution

Through code execution, the tool can **evaluate what lines that was executed**

Analysis

Comparing to source code, the tool can base a **metric of coverage.**

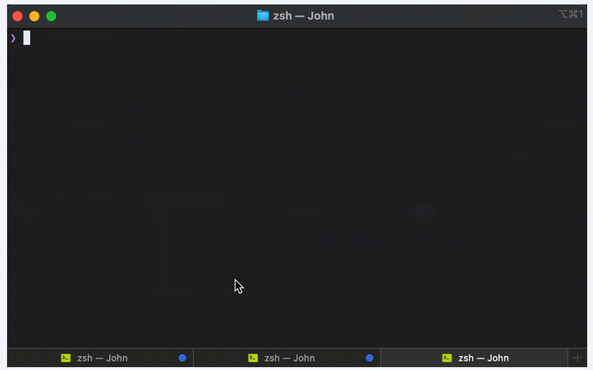
Reporting

By accumulating data, the tool can visualize what branches aren't being executed.

This is usually done in combination with a testing suite

Coverage.py+ Pytest.py = ❤️

WHAT: EXAMPLE TOOLING



Coverage.py

Executes pytest that consequently executes **your** test cases

```
$ pip install coverage
$ python -m coverage
```

Pytest.py

Provides test cases for **your** project

```
$ pip install pytest
$ python -m pytest
```

Benefits

- Cost effective technique to deliver high code quality
- Risk mitigation strategy, no **conditionals** are left behind
- Advertisable, a project with 100% code coverage is less prone to errors than one with 70%
- Accountable, commits that **degrades** code coverage is measurable
- Helps to find forgotten test cases
- Reduced risks of **false positives**
- Key performance indexable (KPI)

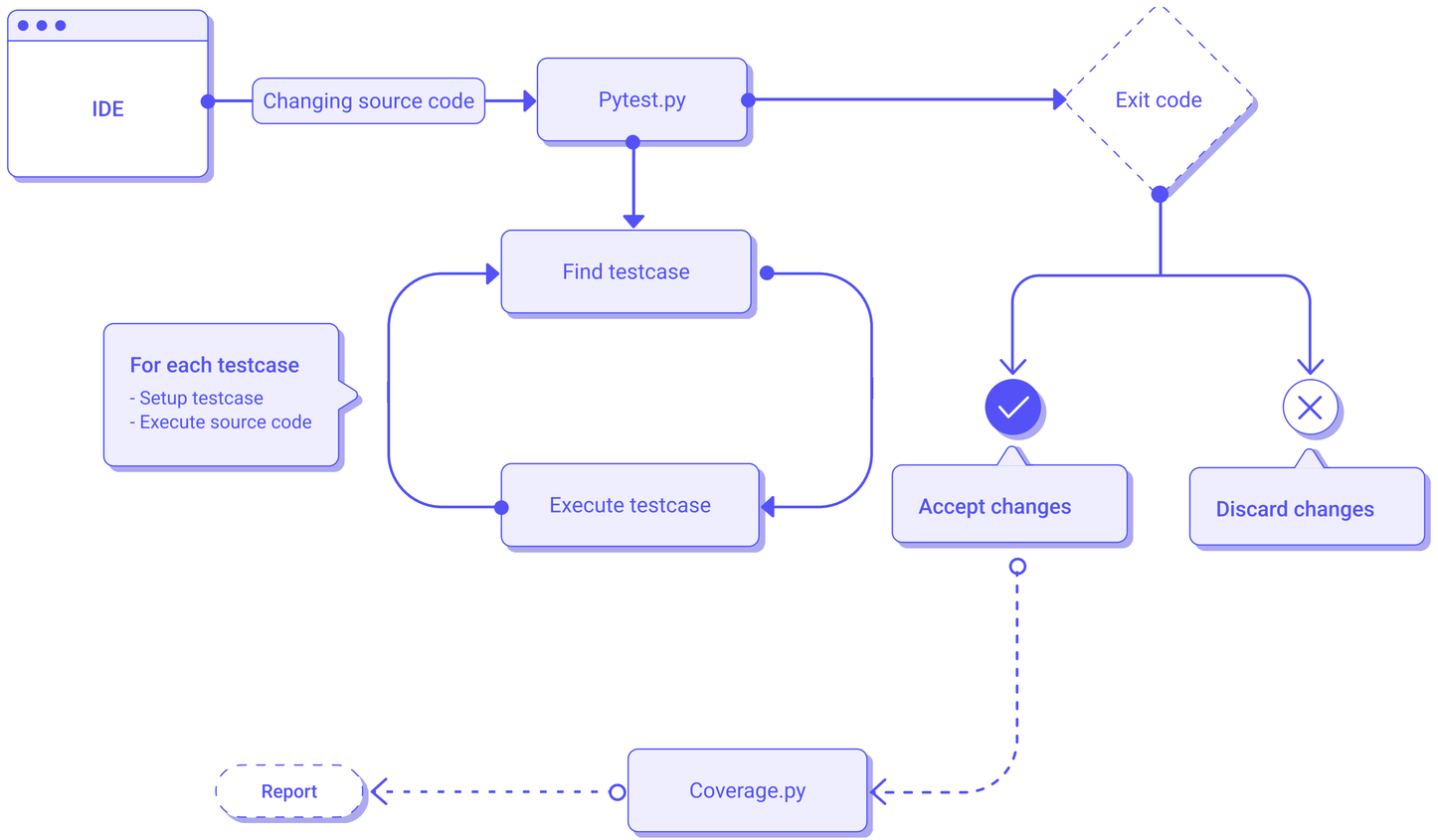
Drawbacks

- Adds **overhead (time)** to the CI/CD pipeline
- Is computational **expensive**
- Is language dependent
- May introduce a **false sense of security**
- Annoying to maintain yet another metric (KPI)

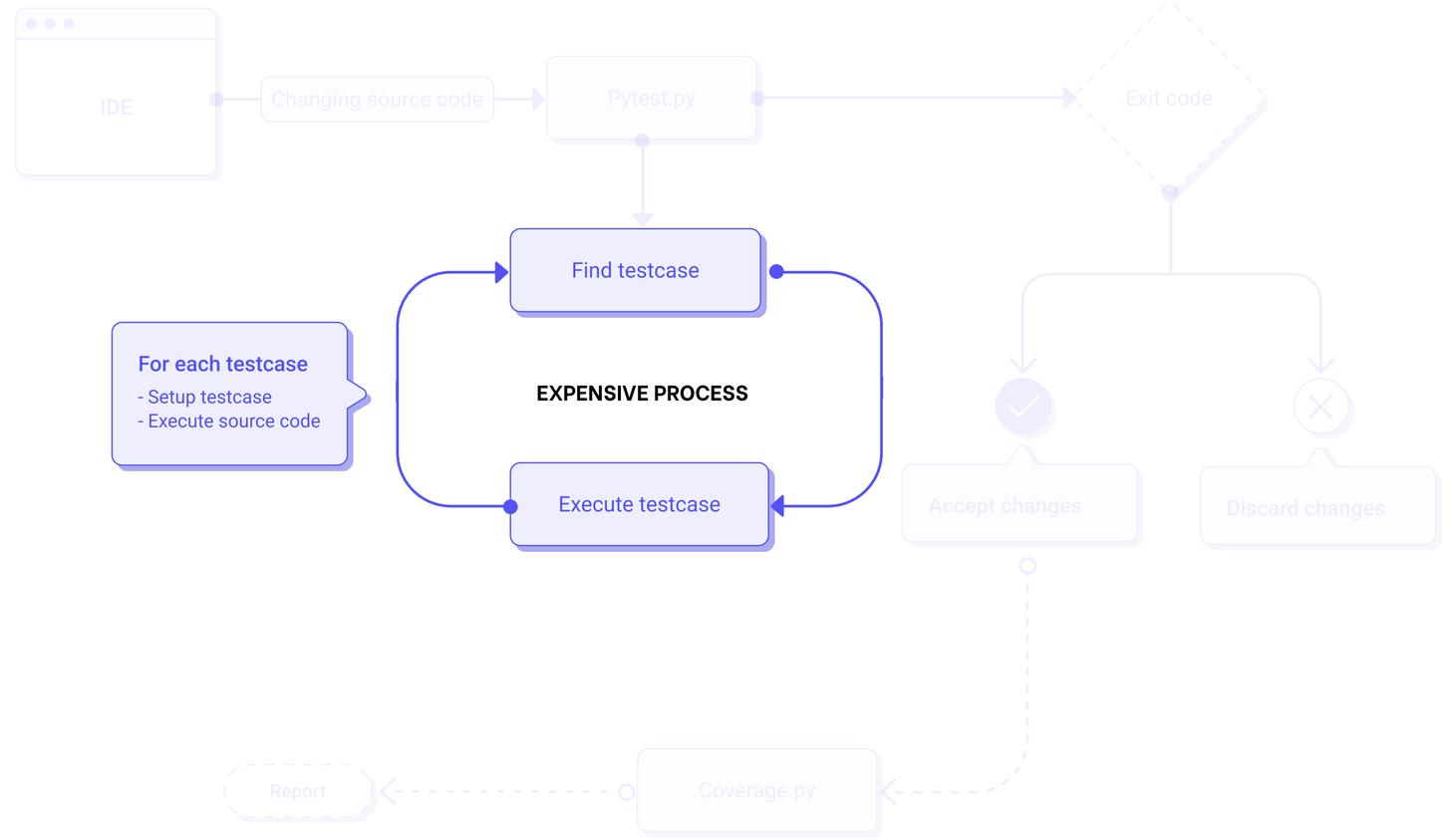
THE PROBLEM

Tests are **computational expensive**

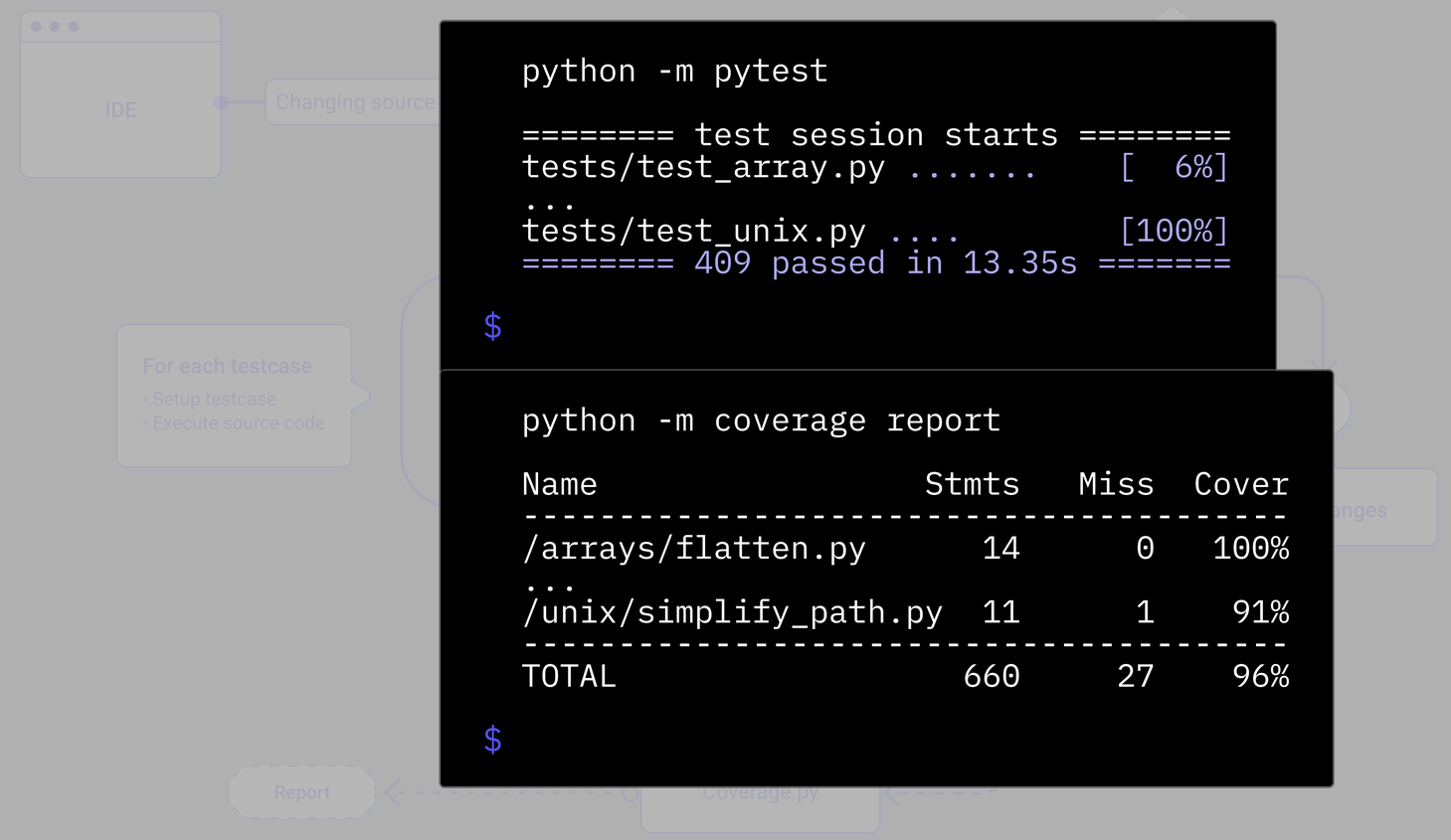
THE PROBLEM


IDE
Changing source code
Pytest.py
Exit code
Find testcase
For each testcase
- Setup testcase
- Execute source code
Execute testcase
Accept changes
Discard changes
Report
Coverage.py

THE PROBLEM


IDE
Changing source code
Pytest.py
Exit code
Find testcase
For each testcase
- Setup testcase
- Execute source code
EXPENSIVE PROCESS
Execute testcase
Accept changes
Discard changes
Report
Coverage.py

THE PROBLEM




```
$ python -m pytest
======== test session starts ========
tests/test_array.py .......    [  6%]
...
tests/test_unix.py ....        [100%]
======== 409 passed in 13.35s =======
$
```

```
$ python -m coverage report
Name                    Stmts   Miss  Cover
-------------------------------------------
/arrays/flatten.py         14      0   100%
...
/unix/simplify_path.py     11      1    91%
-------------------------------------------
TOTAL                     660     27    96%
$
```

HOW

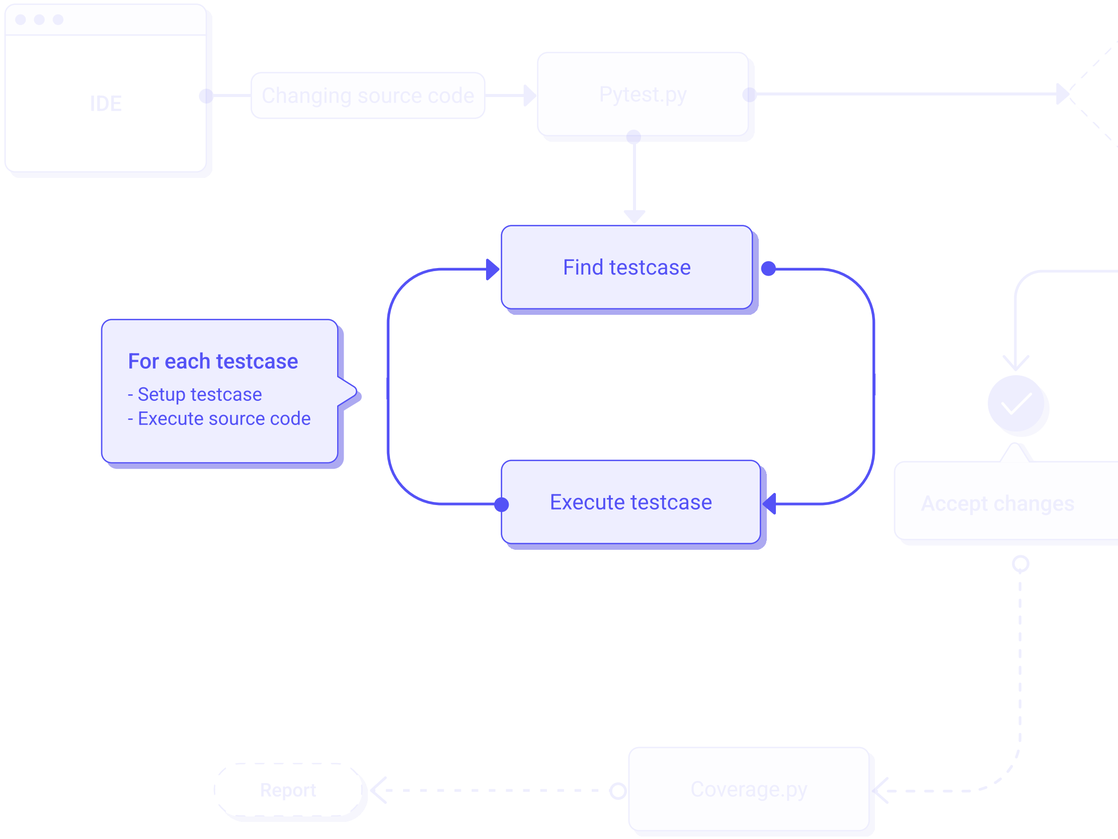
The **naïve** approach

The **naïve** approach

Brute force

The **naïve** approach

Parallellization



HOW: UTILIZING MORE CORES

The **naïve** approach

Parallellization

```python
def execute_process(file):
    run(["coverage", "run", "-m", "pytest", file], shell=True)

if __name__ == '__main__':
    freeze_support()
    start = time.perf_counter()
    l = glob.glob('**/test_*.py')
    processes = []

    for _ in l:
        p = Process(target=execute_process, args=(_,))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    end = time.perf_counter()
    print(f'Elapsed time: {round(end-start,2)} s')
```

```
python -m pytest

======== test session starts ========
tests/test_array.py .......    [  6%]
...
tests/test_unix.py ....        [100%]
======== 409 passed in 13.35s =======

$
```

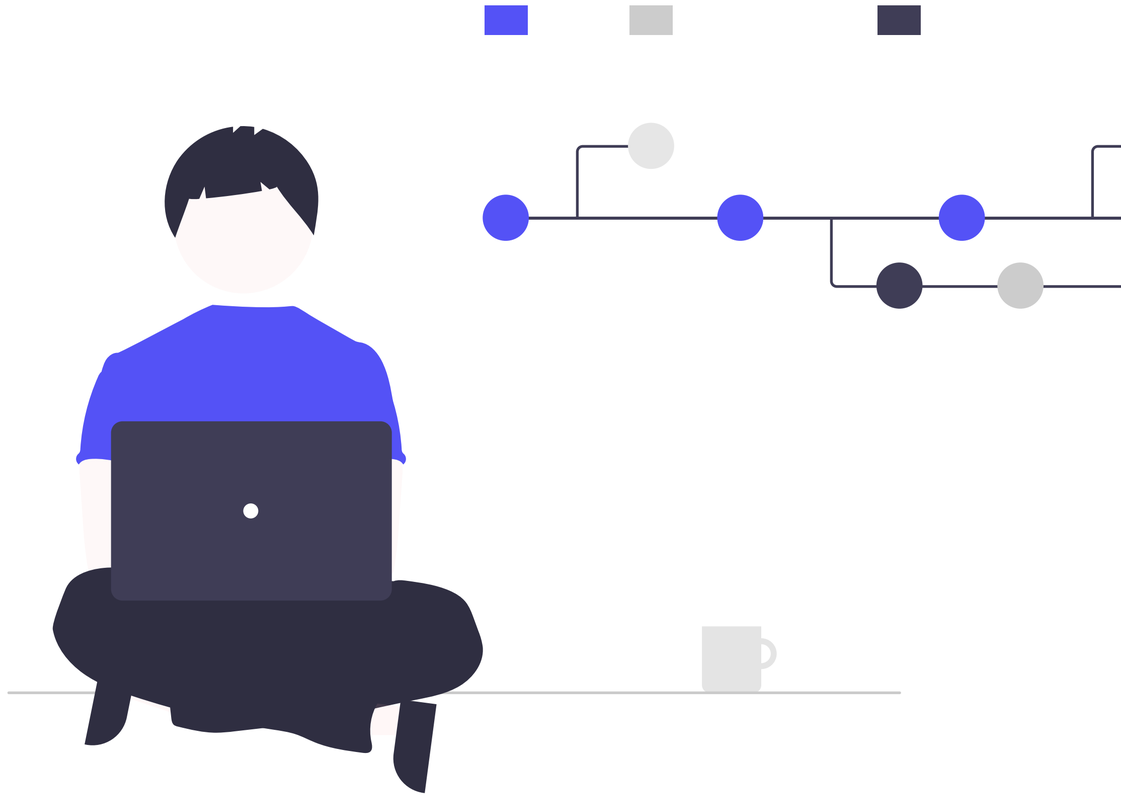
200% speed up

By spawning **multiple** processes for each test case, we decreased time spent with a factor of **2**.

```
python brutetest.py

2 failed, 407 passed in 5.17s
...
2 failed, 407 passed in 8.05s

Elapsed time: 6.38 s

$
```

The **heuristic** approach

Version-control



HOW: UTILIZING SOTA

```python

def get_imports(module_path: Path):
    modules_before = set(sys.modules)
    if not module_path.exists():
        return set()

    path = module_path.resolve().absolute()
    spec = spec_from_loader("module.name", SourceFileLoader("module.name", str(path)))
    if spec and spec.loader:
        mod = module_from_spec(spec)
        spec.loader.exec_module(mod)

    modules_after = set(sys.modules)

    modules_found = modules_after.difference(modules_before)

    for m in modules_found:
        sys.modules.pop(m)

    return modules_found


def glob_tests(root_dir: Path, test_folder: Path):

    test_imports: list[PickData] = []

    sys.path.append(str(root_dir))
    for path in test_folder.glob("test_*.py"):
        imports = get_imports(path)
        test_imports.append(
            PickData(
                path=path,
                dependants=set(filter(lambda x: x.startswith(root_dir.name), imports)),
            )
        )

    return test_imports
```

pytest-testmon

@tarpas

pytest-picked

@anapaulagomes

HOW: UTILIZING SOTA

```python
def get_imports(module_path: Path):
    modules_before = set(sys.modules)
    if not module_path.exists():
        return set()

    path = module_path.resolve().absolute()
    spec = spec_from_loader("module.name", SourceFileLoader("module.name", str(path)))
    if spec and spec.loader:
        mod = module_from_spec(spec)
        spec.loader.exec_m

    modules_after = set(sy

    modules_found = module

    for m in modules_found
        sys.modules.pop(m)

    return modules_found


def glob_tests(root_dir: P

    test_imports: list[Pic

    sys.path.append(str(ro
    for path in test_folde
        imports = get_impo
        test_imports.apper
            PickData(
                path=path,
                dependants
            )
        )

    return test_imports
```

```python
def get_unstaged_paths(root_dir: Path):
    raw_output = subprocess.run(["git", "status", "--short"], capture_output=True)
    output = raw_output.stdout.decode(encoding="utf-8")
    pattern = re.compile(r"([ MTAU\?]{2}) ([^ \-\>]+)(?: \-\>(.+))?")
    filepaths = output.splitlines()

    paths = set()
    for line in filepaths:
        match = pattern.match(line)
        if match:
            code, orig_path, path = match.groups()
            if path:
                p = root_dir / path

            else:
                p = root_dir / orig_path

                path = orig_path

            if p.is_file() and is_py(p):
                paths.add(".".join(p.with_suffix("").relative_to(root_dir).parts))
    return paths
```

HOW: UTILIZING SOTA

```python
def get_unstaged_paths(root_dir: Path):
    raw_output = subprocess.run(["git", "status", "--short"], capture_output=True)
    output = raw_output.stdout.decode(encoding="utf-8")
    pattern = re.compile(r"([ MTAU\?]{2}) ([^ \-\>]+)(?: \-\>(.+))?")
    filepaths = output.splitlines()

    paths = set()
    for line in filepaths:
        match = pattern.match(l
        if match:
            code, orig_path, pa
            if path:
                p = root_dir /

            else:
                p = root_dir /

                path = orig_pat

            if p.is_file() and
                paths.add(".".j
    return paths
```

```python
    test_folder = Path().parent.parent / "tests/"
    root_dir = get_root_from_test_folder(test_folder)

    with open(os.devnull, "w") as f:
        with redirect_stdout(f):
            tests = glob_tests(root_dir, test_folder)
            unstaged_paths = get_unstaged_paths(root_dir)

            tests_of_interest = set()
            for data in tests:
                deps: set[str] = unstaged_paths.intersection(data.dependants)
                if deps:
                    tests_of_interest.add(root_dir / data.path)

    run_tests(tests_of_interest)
```

WHAT | WHY | THE PROBLEM | HOW | TAKEAWAYS

∞ speed up

We only tests relevant testcases from unstaged files accordingly to git version control.

```
  git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   algorithms/arrays/flatten.py
        modified:   algorithms/unix/path/split.py
        modified:   tests/test_array.py
$
```

∞ speed

We only tests relevant testcases from unstaged files accordingly to git version control.

```
  py tooling/yore.py
================== test session starts ==================

collected 32 items

tests/test_unix.py ....                                [ 12%]
tests/test_array.py ............................       [100%]
================== 32 passed in 0.23s ===================
$
```

∞ speed

We only tests relevant testcases from unstaged files accordingly to git version control.

TAKEAWAYS

01

Looking at code coverage is a good thing and good tools are available.

02

Adding coverage analysis to your CI/CD pipeline **may** introduce overhead.

03

The naive approach to parallelization is beneficial but “stupid”.

04

Using a heuristic to reduce redundant analysis is **key** to improving efficiency.

Thank You

Questions?