Mutation Testing to Measure Test Suite Effectiveness

🔍 What is a Mutation?

A **mutation** is a small syntactic change in your source code meant to mimic a common developer mistake. Examples:

Original Code	Mutated Code	Mutation Type		
`a + b`	`a - b`	Arithmetic operator		
`if (x > y)`	`if (x >= y)`	Relational operator		
`return true`	`return false`	Boolean return		
`x && y`	`x		y`	Logical operator

These are injected one at a time to create **mutants** (modified versions of your code).

📊 Mutation Score

$$ \text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants}} \times 100 $$

- **High score (90–100%)**: Strong test suite
- **Low score**: Weak coverage or missing assertions

✅ Example (Python with `mutmut`)

Original Function (example.py):

```
def is_even(n):
    return n % 2 == 0
```

Test (test_example.py):

```
def test_is_even():
    assert is_even(4)
    assert not is_even(3)
```

Run Mutation Testing:

```
pip install mutmut
mutmut run
mutmut results
```

Output:

```
1 killed, 1 survived, 0 timeout, 0 incompetent
```

You now know at least one mutant was **not** caught by the test suite.

⚠️ Caveats

- **Performance**: It runs tests multiple times (once per mutant), so it’s slower than normal test runs.
- **False Positives**: Some mutants are **equivalent** (they don't change the behavior), and no test could kill them.
- **Noise**: Over-mutation or trivial changes may generate noise. Focus on core modules or critical logic paths.

📦 Integration with Pytest

Some tools like `mutmut` integrate well with `pytest`. You can set it to use your pytest runner with:

```
mutmut run --runner "pytest test_example.py"
```

Conclusion

Mutation testing is a **powerful quality assurance technique** that challenges the assumptions made by developers when writing tests. It helps ensure your tests are not just **covering lines**, but also catching **real defects**.