Speeding up pytest runs

SF Python meetup
April 10, 2024

James Abel

Agenda

- pytest intro
- pytest-xdist
- CPUs
- msqlite
- pytest-fly

pytest intro

- **The pytest framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries**
- **Simple tests**: Writing tests is straightforward with pytest because you can use Python's built-in assert statement for test conditions.
- **Detailed info on failing assert statements**: When an assert fails, pytest provides detailed context to help you understand why.
- **Auto-discovery of test modules and functions**: pytest automatically discovers tests following its conventions, so you don't need to manually register tests.
- **Fixture support**: pytest offers powerful fixture support, which is a way to provide a fixed baseline upon which tests can reliably and repeatedly execute.
- **Parameterized testing**: You can easily parameterize tests to run the same test function with different inputs.
- **Plugins**: pytest has a vast ecosystem of plugins to extend its functionality for various needs, like parallel test execution, test coverage, and more.

Speeding up test runs

- `mock`-ing can run faster than using real services
  - AWS: moto
  - awsimple (76 tests):
    - moto mock: 43 sec
    - real AWS: 417 sec
    - 9.7x speedup!
  - Also, can facilitate CI
- Only re-run tests that are impacted by changed code, e.g., using `pytest-testmon`
- Only run meaningful tests (can be difficult)
- Caching (can be even more difficult …)
- *Parallelism* …

pytest-xdist

- **pytest-xdist is a plugin for the pytest framework that enables you to run tests in parallel, across multiple CPUs or even across different machines.**
- **Parallel execution**: Distribute tests across multiple CPUs to speed up the execution. This is particularly beneficial for large test suites or tests that perform time-consuming operations such as accessing the network or a database.
- **Distributed testing**: Run tests in a distributed manner across multiple machines to scale the testing process horizontally. This is useful in environments where the test suite is too large for a single machine or when you want to test in different environments simultaneously.
- **Load balancing**: Dynamically allocate tests to different CPUs or machines based on their current load, ensuring an even distribution of work and optimizing the overall test execution time.
- Generally, run-time feedback may be reduced vs. regular serial run
  - Due to how pytest-xdist is implemented, the -s/--capture=no option does not work.
  - https://pytest-xdist.readthedocs.io/en/latest/
- ***<u>Requires tests be independent</u>***
- **`-n X`** or **`–n auto`**
  - -n X explicitly defines number of workers, e.g., -n 4 for 4 workers
  - -n auto tells pytest-xdist to determine the number of workers

“Workers” vs. CPUs

- Not all “CPUs” or “Processors” are equal
  - We’re not talking about GPUs here … (at least yet)
- Long ago a CPU was a single processor. Now, generally at least in pairs.
- “Big” Cores (Performance cores)
  - Highest Single-Threaded performance, highest power, highest super-scaler, most execution units
- “Little” Cores (Efficiency cores)
  - Good performance/Watt
- SMT (Simultaneous Multi-Threading, AKA Intel® Hyper-Threading™ or HT)
  - 2 (or more) “virtual” processors presented to the OS from one core
  - Good for workloads where OS threads/processes don’t saturate a shared compute resource such as execution units
- Some platforms will have a mix
- Generally, set the number of workers to at least the number of “performance” cores
  - Assuming no inter-test dependencies
- Most platforms are power/thermal limited

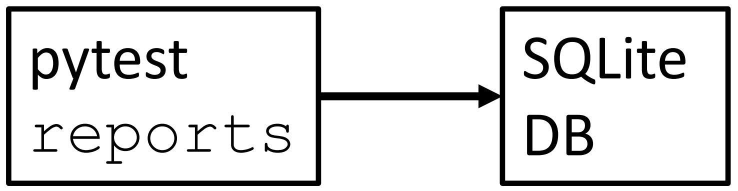
Hardware Configurations can impact Expected Performance

pytest-fly

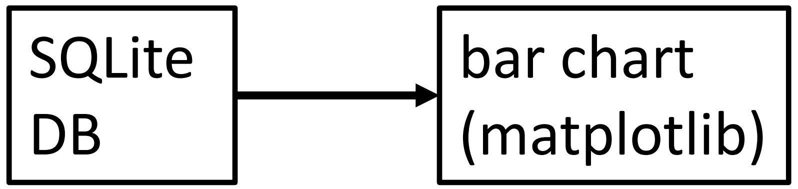
New!

- **Enables performance rollups and visualizations of pytest runs to aid the test developer to improve test performance and reliability**
- A pytest plugin that records pytest Reports to a local SQLite DB
  - Includes a basic visualization
- Installation
  - `pip install pytest-fly`
- Usage in pytest
  - `pytest --fly`
- Visualization
  - `python -m pytest_fly`

pytest run



visualization



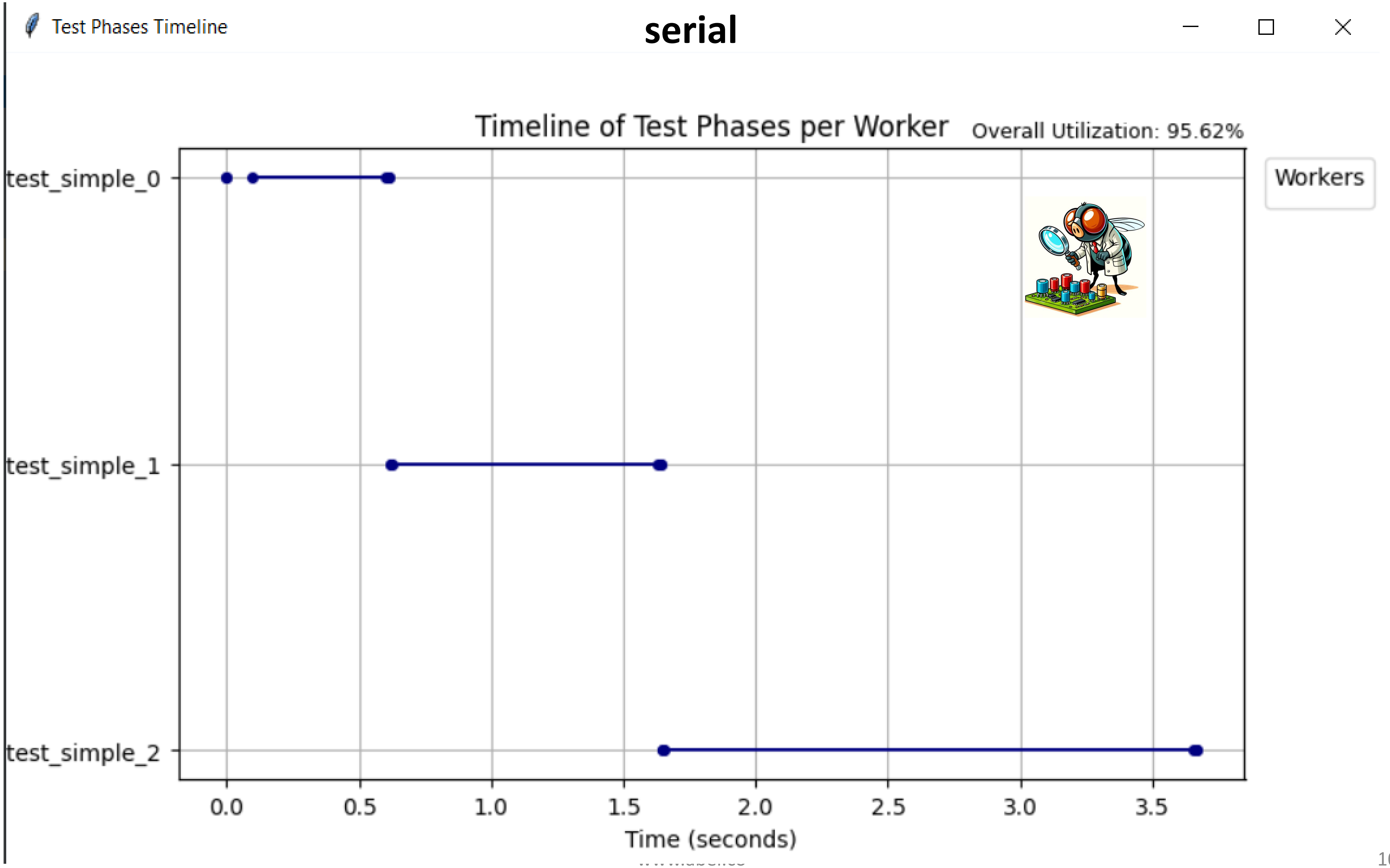
msqlite

- **m**ulti-threaded and **m**ulti-process access to SQLite DB (file)
- A separate python package on PyPI to facilitate using SQLite DBs in a parallel environment
- Supports a subset of SQLite usage models
- Meant for infrequent, short DB writes
- Locks the DB (file) for all accesses
- Automatic retries
- `pip install msqlite`
- https://github.com/jamesabel/msqlite

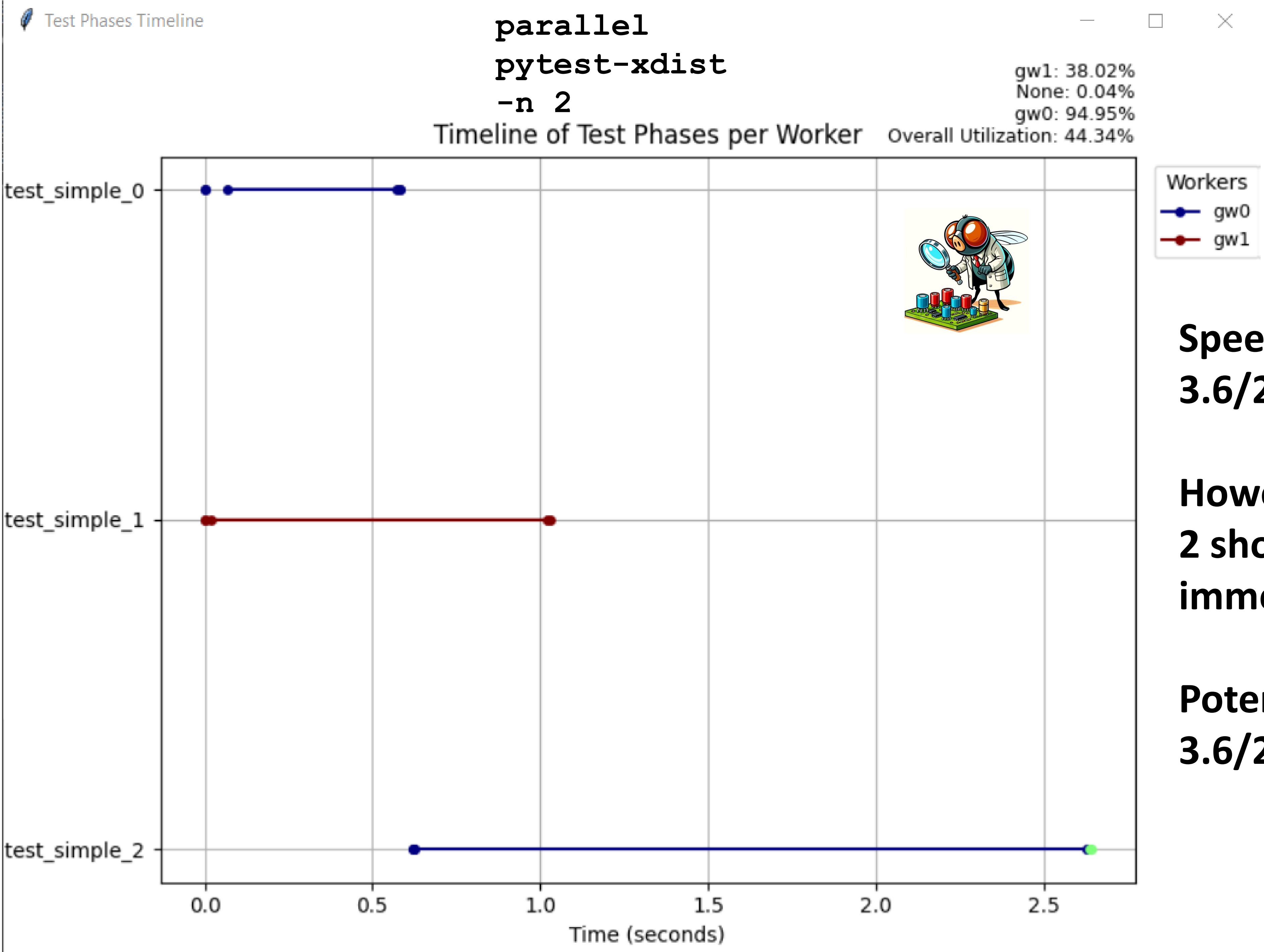
pytest Reports

- **nodeid**: A unique identifier for the test item. It's a string that represents the full path to the test, including the file name, class, and test function.
- **location**: A tuple containing the filesystem path to the test file, the line number where the test starts, and the test name. This provides an exact location of the test in the source code.
- **outcome**: A string indicating the outcome of the test, typically 'passed', 'failed', or 'skipped'.
- **duration**: The time taken to run the test, in seconds.
- **when**: The phase of the test execution this report represents. For TestReport, this can be 'setup', 'call', or 'teardown', indicating which phase the report is related to.

serial


Test Phases Timeline
Timeline of Test Phases per Worker
Overall Utilization: 95.62%
Workers
test_simple_0
test_simple_1
test_simple_2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Time (seconds)

```
parallel
pytest-xdist
-n 2
```



Speedup:
3.6/2.6=1.4x

However, test 2 should start immediately

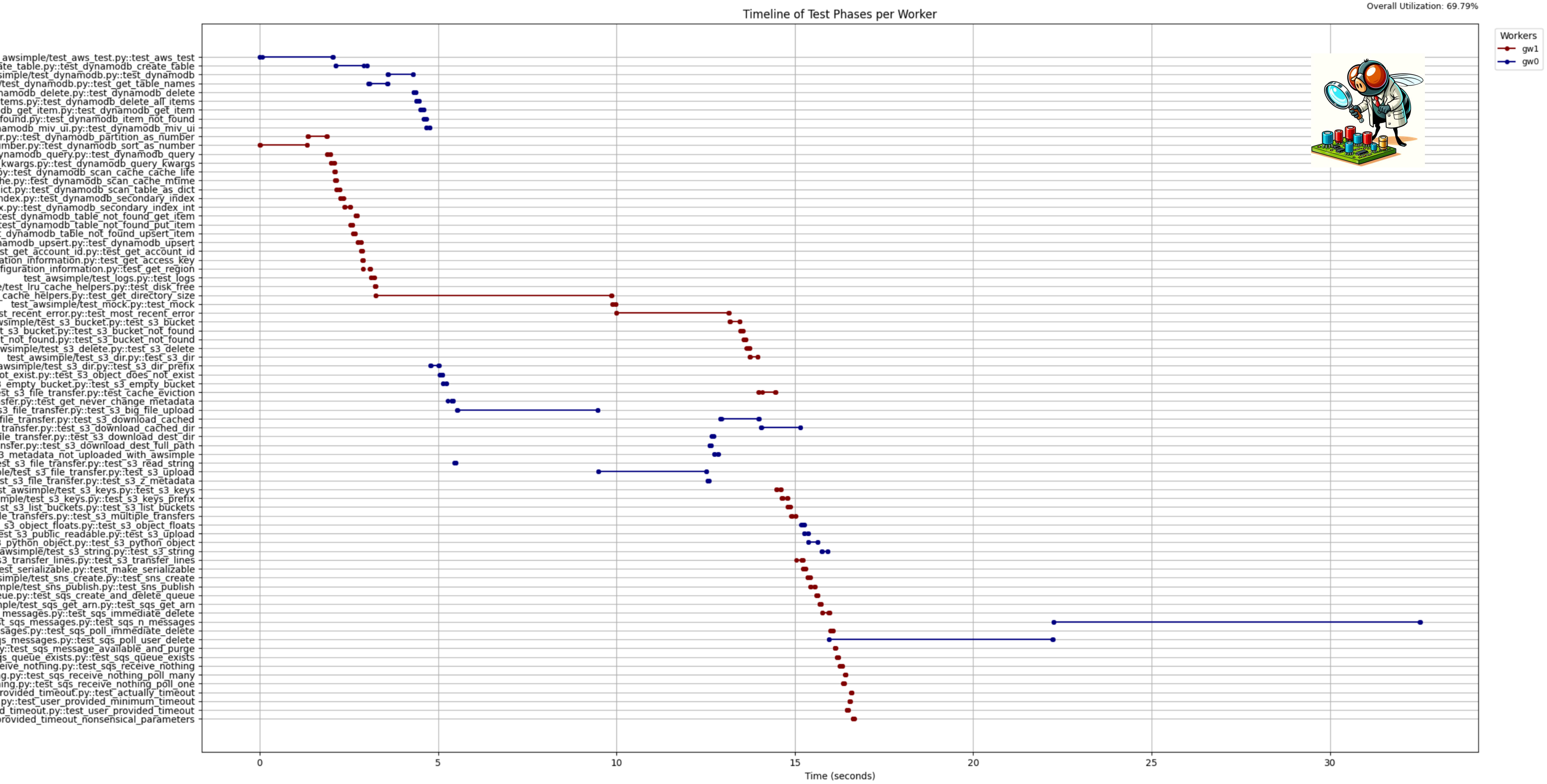
Potential:
3.6/2.0=1.8x

`awsimple` (mocked)

gw0: 95.46%

gw1: 44.12%

Overall Utilization: 69.79%



pytest-fly next steps

- Testing of pytest-fly itself
- Documentation
- Examples
- https://github.com/jamesabel/pytest-fly

BACKUP