Creating unit tests for GlideinWMS using AI tools

Ilya Baburashvili
CUNY Hunter College
695 Park Ave. New York , NY 10065

FERMILAB-POSTER-24-0158-STUDENT

Introduction

GlideinWMS is a workload management system that uses distributed computing to complete tasks, also known as jobs. It is particularly useful for high-throughput computing that's used in research projects. It relies on Glideins, which are pilot jobs that pull jobs from a queue and provide resources for their completion, based on the jobs requirements. These decisions are made based on resource availability and job requirements. We used new AI tools to add unit tests to GlideinWMS.

Acknowledgments

This research was supported in part by the U.S. Department of Energy (DOE), Omni Technology Alliance Internship Program. The program is championed by the DOE's Office of Chief Information Officer (OCIO) and represents a partnership with the leadership of the Office of Economic Impact and Diversity, the Office of Science, the Office of Nuclear Energy, and the National Nuclear Security Agency. The program is administered by the Oak Ridge Institute for Science and Education.

```python
def test_count_slots(source_instance, setup_metrics_dir_for_test):
    expected_values = load_expected_values()
    with mock.patch.object(
        htcondor_query.CondorStatus, "fetch", return_value=utils.input_from_file(FIXTURE_F
    ), mock.patch.object(source_instance, "get_metric_values", return_value=expected_value
        source_instance.load()
        metric_values = source_instance.get_metric_values()["slots_status_count"]
        for label, expected_value in expected_values["Slots"].items():
            assert metric_values[label] == expected_value


def test_count_cores(source_instance, setup_metrics_dir_for_test):
    expected_values = load_expected_values()
    with mock.patch.object(
        htcondor_query.CondorStatus, "fetch", return_value=utils.input_from_file(FIXTURE_F
    ), mock.patch.object(source_instance, "get_metric_values", return_value=expected_value
        source_instance.load()
        metric_values = source_instance.get_metric_values()["cores_count"]
        for label, expected_value in expected_values["Cores"].items():
            assert metric_values[label] == expected_value


def test_count_memory(source_instance, setup_metrics_dir_for_test):
    expected_values = load_expected_values()
    with mock.patch.object(
        htcondor_query.CondorStatus, "fetch", return_value=utils.input_from_file(FIXTURE_F
    ), mock.patch.object(source_instance, "get_metric_values", return_value=expected_value
        source_instance.load()
        metric_values = source_instance.get_metric_values()["memory_count"]
        for label, expected_value in expected_values["Memory"].items():
```

Method and Materials

The past several years have seen a significant growth in the availability of AI tools. This includes LLMs such as ChatGPT, Github Copilot, Amazon You, Anthropic Claude, and others. First, we needed to research and test various AI models to select the most suitable one for creating unit tests. This would take into account things like cost, its ability to produce working code and explanations.

I selected ChatGPT, Claude, and Github Copilot. This was due to the availability of free browser versions of ChatGPT and Claude and a free trial version of Github Copilot, as well as their ease of use.

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

Results

Claude and ChatGPT were both effective in creating unit tests when provided with code for functions that were being tested. However, changes still needed to be made to make the unit tests pass. In addition, the browser versions of both LLMs can't access the entire repository. This makes their use more difficult since they aren't able to access the entire codebase, making it hard to produce working code from the start.

Conclusion

Claude and ChatGPT are both reliable AI tools for creating unit tests and adding other functionality to a large scale project like GlideinWMS. However, given their limitations, other AI tools like Github Copilot might be more suitable for working with a large codebase. Comparing those is something that needs to be looked into in the future.

```python
def test_fetch(condor_q, monkeypatch):
    expected_result = {(1, 0): {"Owner": "user1"}}

    # Patch fetch_using_exe
    monkeypatch.setattr(condor_q, "fetch_using_exe", lambda *args, **kwargs: expected_resu
    result = condor_q.fetch(format_list=[("Owner", "s")])
    assert result == expected_result

    # Patch fetch_using_bindings
    monkeypatch.setattr(condor_q, "fetch_using_bindings", lambda *args, **kwargs: expected_
    monkeypatch.setattr("condor_lib.USE_HTCONDOR_PYTHON_BINDINGS", True)
    result = condor_q.fetch(format_list=[("Owner", "s")])
    assert result == expected_result

def test_init(mock_schedd_cache):
    # Test with schedd_name and pool_name
    condor_q = CondorQ(schedd_name="schedd1", pool_name="pool1", schedd_lookup_cache=mock_
    assert condor_q.schedd_name == "schedd1"
    mock_schedd_cache.getScheddId.assert_called_with("schedd1", "pool1")

    # Test with None schedd_lookup_cache
    condor_q = CondorQ(schedd_name="schedd1", pool_name="pool1", schedd_lookup_cache=None)
    assert isinstance(condor_q.schedd_lookup_cache, local_schedd_cache.NoneScheddCache)

def test_complete_format_list(condor_q):
    format_list = [("Owner", "s")]
    expected_format_list = [("Owner", "s"), ("ClusterId", "i"), ("ProcId", "i")]
    assert condor_q.fetch(format_list=format_list) == expected_format_list
```