Unit and Integration Testing Plan for Bond Analytics Scripts

This section outlines a robust testing framework for the callable and non-callable bond analytics scripts as described in the unified design document. It includes example unit tests and integration tests, best practices, and guidance on technology choices.

1. Testing Strategy Overview

- Unit Tests: Isolate individual calculation functions (e.g., YTM, duration, convexity) and utility methods using controlled (mock) inputs.
- Integration Tests: Validate full end-to-end runs—simulating the script's real execution with a given parameter set, ensuring interoperability of all components.

Recommended Tools:

- pytest: For clear, simple test syntax and fixtures.
- unittest.mock: For mocking QuantLib or user input where relevant.
- tox or CI pipeline: For automated repeated test execution.

2. Example Unit Tests

2.1 Non-CallableBondCalculator

```
python

import pytest
import QuantLib as ql
from bond_metrics_utils import calculate_ytm,
calculate_duration, calculate_effective_duration_convexity

def test_calculate_ytm_basic():
    # Setup: simple 1-year, 5% annual bond at par
    today = ql.Date(1, 1, 2022)
    ql.Settings.instance().evaluationDate = today
    sched = ql.Schedule(today, today + ql.Period(1, ql.Years),
ql.Period(ql.Annual),
                        ql.UnitedStates(), ql.Following,
ql.Following, ql.DateGeneration.Backward, False)
    bond = ql.FixedRateBond(2, 100, sched, [0.05],
ql.ActualActual())
    ytm = calculate_ytm(bond, 100)
    assert abs(ytm - 0.05) < 1e-6

def test_calculate_duration_types():
    # Setup as above
    ...
    ytm = 0.05
    mod_duration = calculate_duration(bond, ytm,
ql.Duration.Modified)
    mac_duration = calculate_duration(bond, ytm,
ql.Duration.Macaulay)
    # For 1y at par, both durations should equal 1
    assert abs(mod_duration - 1) < 1e-6
    assert abs(mac_duration - 1) < 1e-6

def test_effective_duration_convexity():
    ...
    ytm = 0.05
    eff_duration, eff_convexity =
calculate_effective_duration_convexity(bond, ytm, 1e-4)
    assert eff_duration > 0
    assert eff_convexity > 0
```

2.2 CallableBondCalculator

python

```
from callable_bond_module import CallableBondCalculator

def test_ytw_matches_known_call_date():
    # Setup bond with known worst call
    ...
    results =
CallableBondCalculator(...).calculate_yields_to_call()
    ytw, call_date = min(results, key=lambda x: x[1])  # assuming
(date, ytc)
    assert call_date == expected_worst_call_date
    assert abs(ytw - expected_ytw) < tolerance
```

2.3 Utilities

python

```
from bond_metrics_utils import build_quantlib_schedule

def test_build_quantlib_schedule_output():
    sched = build_quantlib_schedule(...)
    assert isinstance(sched, ql.Schedule)
```

3. Example Integration Tests

Integration tests should simulate the script end-to-end, checking both correct execution and the final printed/output values.

python

```
import subprocess
import sys

def test_non_callable_script_e2e(tmp_path):
    # Write a script with known parameters and expected outputs
    result = subprocess.run([sys.executable,
"non_callable_bond_metrics.py"], capture_output=True, text=True)
    assert "Yield to Maturity:" in result.stdout
    assert "Modified Duration:" in result.stdout
    # Optionally, parse the output to check numerical correctness
```

```
def test_callable_script_e2e(tmp_path):
    result = subprocess.run([sys.executable, 
"callable_bond_metrics.py"], capture_output=True, text=True)
    assert "Yield to Worst:" in result.stdout
    # Additional checks as above
```

4. Error Handling and Edge Cases

- Write tests to simulate bad inputs (e.g., maturity before issue, empty call schedules, prices too high for YTM).
- Ensure errors are reported as “N/A” and script does not crash.

python

```
def test_invalid_parameters_raise_or_report_na():
    # e.g. pass settlement_days larger than bond term
    ...
    with pytest.raises(ValueError):
        ... # or check str(output) for "N/A"
```

5. Automation & Best Practices

- Run `pytest` automatically on commit/pull request in CI (GitHub Actions, Travis, GitLab).
- Maintain >80% code coverage; focus on core business logic over print statements.
- Refactor code for testability (avoid hardcoded dates/params in calculation routines).

6. Documentation & Test Data

- Provide sample scripts/configs for both passing and failing tests.
- Document expected behaviors and edge-cases in `README.md` under a Testing section.

7. Summary Table: Types of Tests

Test Type	**Scope**	**Example Function**
Unit Test	Pure calculation routines	`calculate_ytm`, `calculate_duration`
Integration	Script workflow/outputs	Full main script execution
Error/Edge	Edge/corner cases	Bad params, pricing failures, call schedule