# 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 |