STUDENT: NICHOLAS ALAMIR PORTFOLIO

INTRODUCTION

This portfolio documents my software testing work on an ILOC instruction scheduler, a compiler backend component with six pipeline stages: Scanner, Parser, Renamer, Dependency Graph Generator, Priority Calculator, and Scheduler. This portfolio demonstrates my work across requirements analysis, test planning and instrumentation, test implementation, process evaluation, and code review with continuous integration.

LEARNING OUTCOME 1: REQUIREMENTS ANALYSIS AND TESTING STRATEGY

I identified 54 distinct testing requirements across three abstraction levels: 18 unit-level (individual components), 21 integration-level (component interactions), and 15 system-level (end-to-end functionality). Requirements covered functional correctness (Scanner tokenization, Parser instruction recognition), performance (processing efficiency), and Robustness (error handling).

Requirements Coverage by Component:

Scanner: 7 requirements covering tokenization of all 10 ILOC operations, register/constant handling, delimiter recognition, whitespace skipping, comment handling, and error detection
Parser: 7 requirements for parsing all instruction types, linked list construction, operation counting, max register tracking, comment handling, and syntax error reporting
Renamer: 4 requirements for register renaming, virtual register allocation, and data structure preservation
Dependency Graph: 5 requirements for data dependencies, serial dependencies, conflict edges, and root/leaf identification
Priority Calculator: 3 requirements for latency assignment, descendant counting, and priority formula
Scheduler: 4 requirements for list scheduling, functional unit assignment, cycle counting, and dependency enforcement
Integration: 6 requirements for component interaction and end-to-end correctness

I mapped requirements to testing approaches: unit testing for component isolation and fault localization, integration testing for interface validation, and system testing for end-to-end verification with realistic ILOC programs. This risk-based approach concentrated effort on high-impact components.

Self-Assessment (grade 3-4): Requirements cover diverse types and levels with appropriate testing strategies. I could have been more systematic documenting non-functional requirements.

LEARNING OUTCOME 2: TEST PLANNING AND CODE INSTRUMENTATION

I planned 62 tests but implemented 123, exceeding my plan as coverage analysis revealed additional opportunities. Infrastructure includes pytest.ini with custom markers (unit, integration, system, performance, slow), reusable fixtures in tests/conftest.py (scanner, test_data_dir, sample_iloc_file), and 17 ILOC test files in tests/test_data/.

Code Instrumentation (16 lines across 2 modules): Added logging in Scheduler.py (line 65) for test observability and modified schedule() to return cycle count for performance testing. Enhanced Parser.py error handling (lines 245, 254) to write errors to stderr enabling verification of syntax error detection. These small changes preserved algorithmic behavior while improving testability.

Self-Assessment (grade 3-4): Strong infrastructure, minimal targeted instrumentation, exceeded planned coverage. Weakness: i underestimated initial tests needed, but iteration/adapting is good practice.

LEARNING OUTCOME 3: TEST IMPLEMENTATION AND EXECUTION
I implemented 123 tests achieving 83% coverage across 589 production statements (test-to-code ratio: 1.50). Coverage by component: Renamer 100% (43 statements), DependencyGraphGenerator 100% (52 statements), PriorityCalculator 93% (42 statements), Parser 82% (193 statements), Scanner 76% (148 statements), Scheduler 76% (111 statements).

Test Suite Organization (7 files, 1,611 lines):
test_scanner.py (41 tests, 349 lines): Five test classes covering token recognition, operands, delimiters, whitespace, and errors. Discovered bug: whitespace skipping fails on non-breaking spaces (\xa0), documented with xfail marker (line 342).
test_parser.py (27 tests, 342 lines): All instruction types, linked list construction, operation counting, register tracking, error reporting. Strategic addition of 11 tests boosted coverage to 82%. test_renamer.py (15 tests, 219 lines): 100% coverage testing register renaming and virtual register allocation.
test_dependency_graph.py (16 tests, 242 lines): 100% coverage testing data dependencies, serial dependencies, conflicts, root/leaf identification.
test_priority.py (13 tests, 221 lines):Priority calculation formula 10*latency+descendants and propagation.
test_scheduler.py (2 tests, 42 lines): Focused smoke tests with good validation through integration tests.
test_integration.py (10 tests, 196 lines): End-to-end pipeline tests using realistic ILOC files.

Coverage measured via pytest-cov reported in .coverage and coverage.xml. Branch coverage tested through error handling paths. Uncovered lines are defensive error checks with low practical impact.

Key Achievement: Discovered Scanner whitespace bug shows systematic testing finds real defects. I documented it rather than fixing it because finding bugs provides valuable testing competency evidence.

Self-Assessment (grade 4): Comprehensive, well-organized suite achieving high coverage and discovering real bug. Weakness: I light direct Scheduler unit testing (compensated by integration tests) and lack of property-based or mutation testing.

LEARNING OUTCOME 4: TESTING PROCESS EVALUATION AND LIMITATIONS
Statistical Analysis: 83% coverage (target: 70–75%), mean 87.8% across 6 modules, median 87.5%, standard deviation 10.2%, test-to-code ratio 1.50. Four modules have excellent coverage (100%, 100%, 93%, 82%), two have good coverage (76%, 76%).

Successfully Tested: Functional correctness through unit tests, parsing accuracy for all 10 ILOC operations, register tracking and linked list construction, pipeline integration through end-to-end tests, error detection through negative tests.

Inadequately Tested: Semantic equivalence between original and scheduled code (requires ILOC execution framework), scheduling optimality (requires optimal reference implementation), systematic performance regression detection, stress testing with very large programs.

Coverage Gap Analysis: 146 uncovered lines examined. High-priority gaps: error handling branches in Parser and Scheduler. Medium-priority: path variations in DependencyGraphGenerator. Low-priority: defensive error checks unlikely to trigger.

Critique: Successful practices include incremental development with fast feedback, comprehensive fixtures reducing duplication, clear organization. Improvements needed: earlier CI/CD implementation, mutation testing to verify tests detect defects, property-based testing for automatic edge case generation.

Self-Assessment (grade 3–4): Honest distinction of what I could and couldn't test with thoughtful process critique.

LEARNING OUTCOME 5: CODE REVIEW AND CONTINUOUS INTEGRATION

Code Review: Systematic analysis examined complexity, documentation, and quality. High complexity issues: Parser.parseILOC() (~200 lines, complexity 25) and Scheduler.schedule() (~120 lines, complexity 20) need refactoring. Bug discovered: Scanner.py lines 21–22 only check regular spaces/tabs, failing on non-breaking spaces. Documentation gap: most functions lack docstrings. Metrics show average complexity 14.2 and bug detection rate 1.1 defects per KLOC.

CI/CD Implementation: GitHub Actions with two workflows. Test workflow runs 123 tests on every commit across Python 3.10–3.12, generates coverage reports, enforces 75% threshold (builds fail if coverage drops), and uploads artifacts. Quality workflow runs flake8 and pylint in non-blocking mode. Implementation demonstrates professional practices: automated execution provides immediate feedback, multi-version testing ensures compatibility, coverage threshold prevents regression.

Self-Assessment (grade 3–4): I made a systematic review with findings backed by metrics. Production-quality CI/CD with appropriate tooling and quality gates. Could be enhanced with automated deployment or more sophisticated metrics.

CONCLUSION AND REFLECTION

My portfolio demonstrates strong competency across all five learning outcomes: 83% coverage through 123 well-organized tests, real defect discovery and documentation, automated CI/CD, and critical evaluation of testing process and code quality. Most Valuable Learning: Understanding testing’s iterative nature. Initial 74% coverage improved to 83% through coverage analysis revealing specific addressable gaps. Testing is continuous measurement, analysis, and targeted improvement. Greatest Challenge: Balancing comprehensive testing against time constraints. Risk-based testing concentrated effort on Scanner and Parser where defects have the highest impact. This taught me that professional testing requires resource allocation judgment, not just technical skills. What I Would Do Differently: Establish CI/CD at project beginning for continuous automated feedback. Implement mutation testing to verify tests detect defects rather than just execute code. Start with integration tests before comprehensive unit testing to reveal interface mismatches earlier. The Scanner bug discovery shows testing’s value and limitations. Thorough testing exercised whitespace handling with various inputs. When test.txt failed unexpectedly, investigation revealed non-breaking spaces the Scanner didn’t recognize. This shows systematic testing finds real defects, but coverage alone doesn’t guarantee detection — effectiveness depends on both coverage and input diversity. Self-Evaluation: I believe my work demonstrates good competency (70–79).