CSC490 Assignment A6 - Testing Your Project

Armaan Rehman Shah - 1009641309
Boaz Cheung - 1007673607
Alice Sedgwick - 1009301355
Yuan Yu - 1008782195

November 28, 2025

1 Part One: Showing Code Coverage

We used `pytest-cov` to generate the code coverage for this project. The coverage report can be found in the comments: `https://github.com/UofT-CSC490-F2025/Immigreat/pull/32`

1.1 Configuration Files

- **pytest.ini**: Configures pytest with coverage options including branch coverage, multiple report formats (HTML, XML, JSON, terminal), and test markers (unit, integration, slow).
- **requirements-dev.txt**: Specifies all testing dependencies including `pytest>=7.4.0`, `pytest-cov>=4.1.0`, `moto>=4.2.0` for AWS mocking, and other testing utilities.

1.2 GitHub Workflow

We implemented a comprehensive GitHub Actions workflow at .github/workflows/test-coverage.yml that:

1. **Triggers on:**
   - Push to main and dev branches
   - All pull requests
   - Manual workflow dispatch
2. **Test Matrix:** Runs tests on Python 3.13
3. **Coverage Generation:** Automatically generates multiple coverage report formats:
   - Terminal output with missing lines
   - HTML report (archived as artifact)
   - XML report for external services
   - JSON report for programmatic access
4. **PR Integration:** Automatically posts coverage statistics as comments on pull requests, including:
   - Overall coverage percentage
   - Per-module coverage breakdown
   - Comparison with previous coverage (when available)
5. **Artifact Storage:** Archives HTML coverage reports for 30 days

1.3 Coverage Reports

Our implementation generates comprehensive coverage documentation:

- **TESTING.md**: Complete testing guide with instructions for running tests, understanding coverage reports, and writing new tests.
- **TEST_COVERAGE_SUMMARY**: Detailed summary of test coverage implementation showing how we meet both Part One and Part Two requirements.

1.4 Key Features

1. **Automated CI/CD**: Every commit and PR automatically runs the full test suite with coverage analysis
2. **Branch Coverage**: Tracks not just line coverage but also branch coverage to ensure all conditional paths are tested
3. **Multiple Report Formats**: Supports terminal, HTML, XML, and JSON formats for different use cases
4. **README Integration**: Coverage badges and testing section in README.md display current coverage status
5. **Developer Workflow**: Local testing matches CI environment exactly, ensuring consistency

2 Part Two: Test Coverage Breadth

2.1 Code Coverage

Our project achieves **96.38%** overall code coverage across 1,259 statements. The coverage report can be found in the comments: `https://github.com/UofT-CSC490-F2025/Immigreat/pull/32`

2.2 Module-by-Module Coverage

2.3 Test Suite Composition

Our comprehensive test suite includes **315 test cases** organized as follows:

- **Unit Tests (312 tests)**:
  - Data Ingestion: 62 tests (test_data_ingestion.py, test_data_ingestion_helpers.py)
  - RAG Pipeline: 41 tests (test_rag_pipeline.py, test_rag_pipeline_advanced.py, test_rag_pipeline_edges.py)
  - Web Scrapers: 125+ tests across multiple files
  - Forms Scraper: 60+ tests (core, advanced, edges)
  - Database Admin: 13 tests (test_db_admin_lambda.py)
  - Utilities: 21 tests
- **Integration Tests (3 tests)**: End-to-end workflow testing (test_rag_integration.py)

2.4 Testing Infrastructure

- **Fixtures** (tests/conftest.py): Comprehensive shared fixtures including:
  - AWS service mocks (S3, Bedrock, Secrets Manager, RDS)
  - Database connection mocks
  - Sample data fixtures
  - Environment variable management
- **Mocking Strategy**: Uses `moto` for AWS services, `pytest-mock` for enhanced mocking, and `unittest.mock` for general mocking
- **Test Markers**: Organized with pytest markers for selective test execution:
  - `@pytest.mark.unit`: Fast unit tests
  - `@pytest.mark.integration`: Slower integration tests
  - `@pytest.mark.slow`: Long-running tests

3 Part Three: Test Coverage Depth

3.1 Selected Module: RAG Pipeline

We chose src/model/rag_pipeline.py as our most complex module for in-depth testing because it:

1. Integrates multiple AWS services (Bedrock, RDS, Secrets Manager)
2. Implements sophisticated retry logic with exponential backoff
3. Handles complex query processing with vector similarity search
4. Performs semantic reranking and facet expansion
5. Manages database connections with error handling
6. Processes both direct and HTTP Lambda invocations

3.2 Comprehensive Test Suite

Our RAG pipeline testing is distributed across three files with extensive edge case coverage:

- test_rag_pipeline.py
- test_rag_pipeline_advanced.py
- test_rag_pipeline_edges.py

For details on why the performed tests are important, please refer to comments within the module: **src/model/rag_pipeline.py**

3.3 Test Categories Summary

Category	**Count**	**Focus**
Positive Tests	18	Happy path functionality, successful operations
Edge Cases	15	Boundary conditions, empty inputs, malformed data, missing configs
Failure Modes	13	Error handling, exceptions, validation failures, network errors
Total	**46**	**Comprehensive coverage of all scenarios**

Table 1: Test Distribution by Category (RAG Pipeline + Related Modules)

4 Part Four: Two Memorable Bugs

Video: https://utoronto-my.sharepoint.com/