Test Approach Document:
Yet Another Metrics Collector (YAMeC)

C. Marcus DiMarco & Brendan Gibbons

MSCS710 Project

Spring 2025

Contents

1. Introduction

This document outlines our testing strategy for the Yet Another Metrics Collector (YAMeC) application. Given our tight deadline, we're focusing on critical components while maintaining a practical approach to ensure basic functionality works correctly.

2. Testing Scope

2.1 In Scope

- Unit tests for C++ system metrics collection
- Unit tests for JNI bridge functionality
- Manual testing for web interface
- Basic integration tests to verify data flow

2.2 Out of Scope

- Extensive performance testing
- Automated UI testing
- Cross-platform testing (focusing on Windows only)
- Stress/load testing

3. Test Environments

Development: Our local machines are running the target OS (Windows)

Testing tools: Google Test (for C++), JUnit (for Java)

4. C++ Unit Testing Approach

4.1 Testing Framework

We'll use Google Test for C++ unit testing due to its simplicity and widespread adoption.

4.2 Test Cases for C++ System Calls

Basic Metric Retrieval Tests

- Test that CPU usage returns a valid percentage (0-100%)
- Test that memory metrics return non-negative values

- Test that disk and NIC metrics return for each partition/disk and network card on the system
- Test that process-level metrics are retrieved for each active process

Hardware Data Retrieval Tests

- Test that the hardware data is retrieved for each applicable hardware device from the getHardwareInformation methods

Mock Tests

- Create mock system APIs to test response handling
- Test error handling when system calls fail

4.3 Sample Test Code Approach

```
// Example approach for testing CPU metrics

TEST(SystemMetricsTest, CpuUsageReturnsValidPercentage) {

   double cpuUsage;

   int status = monitor.getCpuUsage(&cpuUsage);

   EXPECT_TRUE(status == 0) << “Retrieval failed. Status code ” << status << “ was returned.”;


   EXPECT_GE(cpuUsage, 0.0) << “CPU Usage returned less than 0% (” << cpuUsage >> “).”;

   EXPECT_LE(cpuUsage, 100.0) << “CPU Usage returned greater than 100% (” << cpuUsage >> “).”;

}

// Example approach for testing memory metrics

TEST(SystemMetricsTest, TotalMemoryReturnsNonNegativeValue) {

unsigned long long physicalBytesAvailable,  virtualBytesCommitted;

double committedPercentUsed;

   int status = monitor.getMemoryCounters(&physicalBytesAvailable, &virtualBytesCommitted, &committedPercentUsed);
```

```
    EXPECT_TRUE(status == 0) << “Retrieval failed. Status code ” << status << “ was returned.”;


    EXPECT_GT(physicalBytesAvailable, 0) << “Physical memory available returned equal to 0 bytes”;

    EXPECT_GT(virtualBytesCommitted, 0) << “Virtual bytes committed returned equal to 0 bytes”;

    EXPECT_GE(committedPercentUsed, 0) << “Committed percent used returned less than 0%”;

    EXPECT_LE(committedPercentUsed, 100.0) << “Committed percent used returned greater than 100%”;

}
```

5. JNI Bridge Testing

5.1 Testing Framework

JUnit will be used for Java-side testing of the JNI bridge.

5.2 Test Cases for JNI Bridge

Library Loading Tests

- Test that the native library loads successfully
- Test behavior when library is missing or incompatible

Data Type Conversion Tests

- Test that C++ values are correctly converted to Java types
- Test boundary values for numeric conversions

Error Handling Tests

- Test handling of C++ exceptions in Java
- Test behavior when native methods return error codes

5.3 Sample Test Code Approach

// Example approach for testing JNI library loading

```
@Test
public void testNativeLibraryLoads() {
    // This test will fail if the library can't be loaded
    assertDoesNotThrow(() -> {
        SystemCpuMetric metric = monitor.getCpuMetrics();
    });
}

// Example approach for testing data type conversion
@Test
public void testCpuUsageReturnsValidValue() {
    SystemCpuMetric metric = monitor.getCpuMetrics();
    double cpuUsage = metric.getUsage();
    assertTrue(cpuUsage >= 0.0 && cpuUsage <= 100.0);
}
```

6. Java Service Layer Testing

6.1 Test Cases for Metrics Service

Service Method Tests

- Test that service layer properly wraps native method calls
- Test that returned metrics map contains expected values
- Test error handling when native methods fail

6.2 Sample Test Code Approach

```
// Example approach for testing metrics service
@Test
public void testGetCpuMetricsReturnsExpectedValues() {
```

```
  SystemCpuMetric cpuMetrics = monitor.getCpuMetrics();


   assertNotNull(cpuMetrics);

  double usage = cpuMetrics.getUsage();

   assertTrue(usage >= 0.0 && usage <= 100.0);
}

// Add similar tests and assertions for other expected metrics and device information
```

7. Database Testing

7.1 Test Cases for Database Operations

Schema Validation Tests

- Test that tables are created with expected schema
- Test that foreign key constraints work correctly

CRUD Operation Tests

- Test inserting metric records
- Test retrieving metric records
- Test updating metric records
- Test querying metrics by timestamp ranges

8. Manual Web UI Testing

8.1 Test Cases for Web Interface

Functional Tests

- Verify metrics are displayed correctly
- Verify page refreshes update metrics
- Verify UI elements are properly rendered

Usability Tests

- Verify metrics are displayed in a readable format
- Verify units are clearly indicated

- Verify error states are clearly communicated

8.2 Detailed Web UI Test Checklist

Test Case	Steps	Expected Result
Metrics Display	1. Launch application 2. Navigate to web UI	All metrics are displayed with values and proper units
Refresh Functionality	1. Launch application 2. Note initial values 3. Refresh page after 30 seconds	Metrics values are updated to reflect current system state
Error Handling	1. Simulate error condition (e.g., by stopping metrics service) 2. Refresh page	Error is clearly displayed to user with helpful message

9. Bug Tracking

Bugs will be tracked using GitHub Issues. Each issue should include:

- Detailed steps to reproduce
- Expected vs. actual behavior
- Severity (Critical, High, Medium, Low)
- Screenshots if applicable

10. Conclusion

This testing approach balances our need for quality with our tight timeline. By focusing on critical components and high-risk areas, we can ensure basic functionality works correctly while acknowledging limitations in our testing scope.

The goal is to verify that:

- System metrics are collected correctly
- Data flows properly through the JNI bridge
- The web UI correctly displays the collected metrics