Software Design Document: System Metrics Collector Application

C. Marcus DiMarco & Brendan Gibbons

MSCS710 Project

Spring 2025

Version	Description	Date
1.0.0	Initial version	Feb 8, 2025

1. Introduction

1.1. Purpose

This document outlines the software design for a system metrics collector application. The application is designed to collect system-level metrics (CPU usage, memory usage, etc.) from the operating system using native C/C++ code and present this data to users through a locally hosted web interface. The application will be built using Java, Maven, and Spring Boot, leveraging JNI for interfacing with the C/C++-based metric collection module.

1.2. Goals

- Accurate Metric Collection: Provide accurate and up-to-date system metrics.
- Cross-Platform Compatibility (Desired): Design the architecture to be adaptable to different operating systems (Linux, Windows, macOS), even if initial implementation focuses on one.
- User-Friendly Web Interface: Offer a simple and intuitive web interface to view collected metrics.
- Modular Design: Employ a modular architecture to facilitate maintainability, testability, and future extensibility.
- Performance Efficiency: Minimize the performance impact of the metric collection process on the system.
- Intelligent Suggestions: Notes on current performance and resource issues and advise the user on potential actions to remedy them. Best practices are also suggested based on the status of the system.

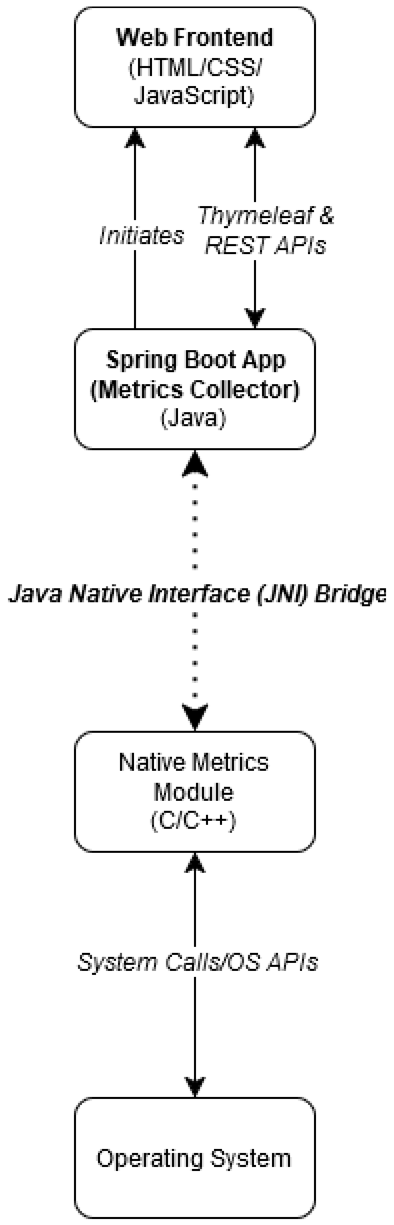
1.3. Target Audience

This document is intended for:

- Software developers involved in the implementation of the metrics collector application.
- Testers who will verify the application's functionality and performance.
- Operations and deployment teams who will be responsible for deploying and maintaining the application.
- Stakeholders interested in the technical design and architecture of the application.

2. Overall System Architecture

The application adopts a layered architecture, separating concerns and promoting modularity. The main components and their interactions are depicted below:


Web Frontend (HTML/CSS/ JavaScript)
Initiates
Thymeleaf & REST APIs
Spring Boot App (Metrics Collector) (Java)
Java Native Interface (JNI) Bridge
Native Metrics Module (C/C++)
System Calls/OS APIs
Operating System

Components:

- Web Browser (User Interface): The client-side component, a standard web browser, used by users to access and view the collected metrics through the web application.
- Spring Boot Application (metrics-collector-app): The core Java application.
    - Hosts the web application (using Thymeleaf).
    - Provides REST API for metric data.
    - Manages the overall application logic and workflow utilizing an MVC (Model, View, Controller) design pattern.
    - Interacts with the JNI Bridge to retrieve metrics.
- JNI Bridge (metrics-collector-jni - JNI part in C++): A C++ layer acting as an intermediary between the Java application and the native C/C++ metrics module.
    - Provides JNI functions callable from Java.
    - Translates data between Java and C/C++ data types.
    - Calls functions in the C/C++ Metrics Module.
- C/C++ Metrics Module (metrics-collector-jni – C/C++ part in system_metrics.cpp): Native C/C++ code responsible for collecting system metrics.
    - Utilizes OS-specific system calls or APIs to gather metric data (e.g., /proc/stat, GetSystemTimes).
    - Focuses solely on metric collection logic, optimized for performance if needed.
- Operating System: The underlying operating system providing the data sources (files, APIs) for metric collection.

3. Module Design

3.1. metrics-collector-app Module (Spring Boot Application)

3.1.1. Responsibilities:

- Web Application Hosting: Serve the web UI using an embedded web server (Apache Tomcat in Spring Boot).
- REST API Endpoint: Provide an API (/api/metrics) to expose metrics in JSON format.
- Data Presentation Logic: Prepare metric data for display in the web UI and API responses (View).
- JNI Interaction: Load the native library and call JNI methods to fetch metrics.
- Error Handling: Manage potential errors, especially related to JNI library loading and native method calls. This would include logging.

3.1.2. Key Components:

- <organization_name>.metrics.app.MetricsCollectorApplication: The main Spring Boot application class, responsible for application initialization.
- <organization_name>.metrics.app.controller.MetricsController:
  - Handles HTTP requests.
  - Possible functions:
    - getMetricsApi(): REST endpoint returning metrics as JSON (@ResponseBody).
    - getMetricsPage(): Controller for the web UI, prepares data for the Thymeleaf template.
- <organization_name>.metrics.app.service.MetricsService:
  - Abstraction layer for metric retrieval logic.
  - Possible functions:
    - getSystemMetrics(): Calls JNI methods via SystemMetricsNative to get metrics. Processes and packages the metrics data into a Map.
- <organization_name>.metrics.app.jni.SystemMetricsNative:
  - Java class declaring native methods that correspond to JNI functions in the metrics-collector-jni module.
  - Handles loading the native library (System.loadLibrary("metricsjni")).
- Thymeleaf Templates (src/main/resources/templates/):
  - index.html: Thymeleaf template to display metrics in a user-friendly HTML page.
- Static Resources (src/main/resources/static/):
  - css/style.css: CSS for basic styling of the web UI.
  - js/app.js: JS for fetching updates and adding dynamic behavior to the client.
- application.properties: Configuration file (port, application name, etc.).

3.1.3. Interfaces:

- REST API: /api/metrics (GET) - Returns JSON payload of metrics.
- Web UI: / (GET) - Serves the HTML page displaying metrics.
- JNI Interface: Interaction with metrics-collector-jni via native methods defined in SystemMetricsNative.java.

3.2. metrics-collector-jni Module (JNI and C/C++ Code)

3.2.1. Responsibilities:

- System Metric Collection: Implement platform-specific logic to gather system metrics using C/C++ code.
- JNI Function Implementation: Provide C++ JNI functions that Java can call.
- Data Conversion: Convert data from C/C++ data types to Java-compatible data types within JNI functions.
- Native Library Building: Compile C and C++ code into a shared library (.so or .dll).

3.2.2. Key Components:

- src/main/c/system_metrics.cpp:
    - Contains C/C++ functions for metric collection. Example functions may include: get_cpu_usage(), get_total_memory().
    - Platform-dependent code will reside here, possibly using preprocessor directives (#ifdef, #endif) for different OS support.
- src/main/jni/SystemMetricsJNI.cpp:
    - C++ file containing JNI wrapper functions.
    - Example JNI functions:
        - Java_<organization_name> _metrics_app_jni_SystemMetricsNative_getCpuUsage(),
        - Java_<organization_name> _metrics_app_jni_SystemMetricsNative_getTotalMemory().
    - These functions call corresponding C/C++ functions from system_metrics.cpp and handle JNI environment interaction and data type conversion.
- src/main/jni/SystemMetricsJNI.h (Optional - can be generated):
    - JNI header file defining function signatures for JNI implementations.
- pom.xml: Used by the Maven build manager to manage project settings and configurations, including dependencies, build settings, version, developer details, and so on.

3.2.3. Interfaces:

- JNI Interface: Functions exposed to Java, defined in SystemMetricsNative.java and implemented in SystemMetricsJNI.cpp.

- C/C++ Function Interface: Internal C/C++ functions within system_metrics.c module for metric collection, called by JNI functions.
- OS System APIs/Files: Interaction with the operating system to retrieve metrics (e.g., system calls, reading files from /proc, Windows API calls).

4. Data Design

4.1. Metrics Collected:

The application will initially collect the following system metrics:

- CPU Usage: Percentage of CPU time currently in use.
- Total Memory: Total system RAM available.
- Used Memory/Free Memory
- Disk Usage (Total, Used, Free for specific partitions)
- Network Usage (Bytes sent/received on interfaces)

4.2. Data Structures:

- C/C++ Code: Native C/C++ data types will be used to collect metric values (e.g., double for CPU usage percentage, long long for memory in bytes).
- JNI Bridge: Data will be converted between C/C++ native types and JNI types (e.g., jdouble, jlong).
- Java Application: Metrics will be stored in java.util.Map within the MetricsService. The keys will be metric names (e.g., "cpuUsage", "totalMemory"), and values will be Java objects representing the metric values (e.g., Double, Long).
- JSON API Response: Metrics exposed via the REST API will be formatted as a JSON object, mirroring the Map structure from the Java service layer.
- Thymeleaf Template Data: The Map of metrics will be passed as a model attribute to the Thymeleaf template for rendering in the web UI.

4.3. Data Flow (Metric Retrieval):

1. Java code in MetricsService calls static native methods in SystemMetricsNative.java (e.g., SystemMetricsNative.getCpuUsage()).
2. This triggers a JNI call to the corresponding JNI function in SystemMetricsJNI.cpp (e.g., Java_com_example_metrics_app_jni_SystemMetricsNative_getCpuUsage()).
3. The JNI function in C++ calls the appropriate C/C++ function in system_metrics.c (e.g., get_cpu_usage()).

4. The C/C++ function uses OS-specific methods to collect the metric data from the operating system.
5. The C/C++ function returns the metric value to the JNI function.
6. The JNI function converts the C/C++ data type to a JNI type and returns it back to the Java SystemMetricsNative class.
7. The Java MetricsService receives the metric value and stores it in the metrics Map.
8. The MetricsController retrieves the metrics Map and passes it to the web UI or returns it as JSON.

4.4. Database Design:

Processes each have data recorded separately at a specific timestamp. Key data about processes are kept in their own records. Specific metrics on a process collected at a specific time are saved to a Metrics record, with each being given a timestamp of when they are running (MetricsInterval). If the user chooses to condense Metrics records after a certain period to save on storage, they are condensed into AggregatedProcessMetrics, which belong to a specific AggregatedInterval.

The idea of this database design is to leverage foreign keys to automatically group data together and make providing total or average metrics easier for any given time.

4.4.1. Application

- Records constant data about a specific process
- Id (PK)
- Name
- Path (directory)

4.4.2. Process

- Records constant data about a specific process
- Id (PK)
- ApplicationId (FK)
- OsProcessId (int)
- ParentProcessId (FK) (nullable)
- SpawnTime (timestamp)

4.4.3. ProcessMetrics

- A record which contains metrics as collected at a specific timestamp
- Id (PK)
- ProcessId (FK)

- CpuUsage
- GpuUsage
- MemoryUsage
- DiskUsage
- NetworkUsage
- Timestamp
- *Performance metrics over the interval of time specified*

4.4.4. SystemMetrics

- A record which contains metrics concerning the operating system, or system as a whole, rather than a specific application or process
- Id (PK)
- Timestamp
- Other metrics which would not be found within ProcessMetrics (Swap memory usage, VRAM, etc.)
    - If this is not used, the ProcessMetrics table can be used and the ProcessId foreign key can be made nullable
- *Performance metrics over the interval of time specified*

4.4.5. MetricsInterval

- Specifies the time which a selection of metrics were collected
- Id (PK)
- Timestamp (Should be unique) – Time which these metrics were recorded
- *Any metrics which cannot be attributed to specific processes & cannot be derived from them*

4.4.6. AggregatedInterval

- Specifies a period in which metrics were collected from and then aggregated into single records
- Id (PK)
- StartTime – Timestamp
- EndTime – Timestamp
- *Any metrics which cannot be attributed to specific processes & cannot be derived from them*

4.4.7. AggregatedProcessMetrics

- A record representing the aggregated system and performance metrics of a process over a specified interval of time.
- Id (PK)
- IntervalId (FK from AggregatedInterval)
- ProcessId (FK from Process)
- *Averaged performance metrics over the interval of time from the Aggregate*

4.4.8. Cpu

- Id (PK)
- Name
- Base speed
- Cores
- Logical Processors
- Virtualization (Boolean)
- L1 cache size
- L2 cache size
- L3 cache size

4.4.9. CpuMetrics

- Id (PK)
- CpuId (FK)
- Clock Speed
- Utilization
- Temperature

4.4.10. Gpu

- Id (PK)
- Name
- Dedicated Memory
- Shared Memory

4.4.11. GpuMetrics

- Id (PK)
- GpuId (FK)
- Dedicated Memory Usage
- Shared Memory Usage

- Temperature
- Graphics Engine Version
- Utilization

4.4.12. Memory

- Id (PK)
- Speed
- Formfactor
- Capacity
- Page Space
- Swap Space
- Slots Used
- Slots Total

4.4.13. MemoryMetrics

- Id (PK)
- MemoryId (FK)
- Memory Used
- Compressed Memory
- Cached Memory

4.4.14. Disk

- Id (PK)
- Name
- Capacity
- Type
- Drive Letters

4.4.15. DiskMetrics

- Id (PK)
- DiskId (FK)
- Active Time (%)
- Response Time
- Read Speed
- Write Speed
- Swap Available
- Swap Used

- Page File Size
- Page File Used

4.4.16. Nic

- Id (PK)
- Name
- Type

4.4.17. NicMetrics

- Id (PK)
- NicId (FK)
- Connection Type (Protocol)
- Send Speed
- Receive Speed

5. User Interface Design

5.1. Web UI Description:

The web UI will be a simple, locally hosted web page accessible through a web browser. It will display the collected system metrics in a tabular format.

5.2. UI Elements:

- Page Title: "System Metrics"
- Metric Table:
  - Two columns: "Metric" and "Value".
  - Rows for each metric (CPU Usage, Total Memory, etc.).
  - Metric values will be displayed with appropriate units (e.g., "%" for CPU usage, "MB" or "GB" for memory).
- Error Display: If there are errors during metric collection (especially JNI related), an error message will be displayed prominently on the page.

5.3. UI Technology:

- Thymeleaf: Server-side templating engine to dynamically generate the HTML page with metric data.
- HTML: Structure of the web page.

- CSS: Basic styling for readability (e.g., table formatting, font styles).
- JavaScript will be added for dynamic updates (auto-refreshing metrics) or more interactive visualizations (charts). However, the initial design will focus on a simple static page refreshed by the user.

6. JNI Design

6.1. JNI Function Naming Convention:

JNI function names will follow the standard JNI naming convention:

Java_<package_name>_<class_name>_<method_name>

Example:

Java_<organization_name> _metrics_app_jni_SystemMetricsNative_getCpuUsage

6.2. JNI Function Signatures:

JNI function signatures will map Java method parameter and return types to corresponding JNI types. For this application:

- Java methods will be static, native methods in SystemMetricsNative.java.
- Return types will be primitive types (double, long) or potentially strings for error messages.
- No arguments are expected for the initial metric retrieval methods. JNI functions will typically have JNIEnv* env and jclass clazz as arguments.

6.3. Data Type Conversion:

JNI layer will handle conversion between Java and C/C++ data types:

- double (Java) <--> jdouble (JNI) <--> double (C/C++)
- long (Java) <--> jlong (JNI) <--> long long (C/C++) (or appropriate C/C++ integer type)
- String handling (if needed for error messages or future text-based metrics) will involve JNI string functions (NewStringUTF, GetStringUTFChars, ReleaseStringUTFChars).

6.4. Error Handling in JNI:

- The C/C++ code will be designed to handle potential errors during system calls gracefully (e.g., checking return values, handling file access issues).
- If errors occur in the C/C++ code, the JNI functions should:
  - Return a special value (e.g., -1 for numeric metrics, or NULL if returning a String) or
  - Throw a Java exception (less preferred for performance in frequent metric retrieval but might be suitable for critical errors).
- Java code in MetricsService will check for error values or handle potential exceptions from JNI calls and display error messages in the web UI.

7. C/C++ Code Design (System Metrics Collection)

7.1. Metric Collection Methods:

The C/C++ code in system_metrics.cpp will utilize OS-specific methods for each metric. For example, on Windows, metrics can be collected utilizing the following metrics:

- CPU Usage: GetSystemTimes fills a buffer for the time the CPU is idle, the time spent running kernel code, and the time spent running user code. Utilization can be measured by polling it over some interval of time, taking the difference between the returned times, and calculating the percentage of time which the system is not idle.
- Total Memory: GlobalMemoryStatusEx fills a buffer value which indicates the status of the memory, including the percent of memory usage, the size and free space of the paging file, the size and free space of physical memory, and the size and free space of virtual memory

7.2. Performance Considerations:

- Metric collection functions in C/C++ should be designed to be efficient and minimize overhead.
- Minimize file I/O and system call overhead.
- Consider caching or sampling strategies if very frequent metric updates are required and performance becomes a concern (though for a simple application, this might not be necessary initially).

7.3. Platform Dependency Management:

- Use preprocessor directives (#ifdef, #elif, #endif) extensively in system_metrics.cpp to isolate platform-specific code sections.
- Create separate source files or folders for platform-specific implementations if the code difference is substantial.
- Build processes (Maven, build scripts) should be configured to compile the correct C/C++ source files based on the target operating system during native library compilation.

8. Non-Functional Requirements

- Performance: Metric collection should be fast and have minimal impact on overall system performance. Web UI should load quickly and be responsive.
- Reliability: The application should be robust and handle errors gracefully, especially in the JNI and native metric collection layers.
- Maintainability: Modular design and clear code structure will enhance maintainability.
- Security: For a locally hosted application, security is less critical, but best practices should still be followed (e.g., avoid storing sensitive information, be mindful of potential vulnerabilities in web UI dependencies if any are added in the future).
- Platform Support: Initial development may target a specific platform (e.g., Windows). Design should facilitate adding support for other platforms (Linux, macOS) in the future.

9. Technology Stack and Tools

- Programming Languages: Java, C, C++
- Database Layer: SQLite
- Build Tool: Maven
- Framework: Spring Boot (for Java application), Thymeleaf (for web UI templating)
- JNI: Java Native Interface
- Operating Systems (Target): Initially Windows (with potential for Linux and macOS support)
- Development Tools:
    - IDE (IntelliJ IDEA)
    - JDK (Java Development Kit)
    - GCC/G++ (or platform-specific C/C++ compilers)

- Maven

10. Build and Deployment

10.1. Build Process:

1. Compile Native Library (metrics-collector-jni):
   - Maven build in metrics-collector-jni module will use exec-maven-plugin or a more integrated approach to compile C and C++ code into a shared library (.so or .dll). Platform-specific compilation settings will be configured (e.g., compiler flags, libraries to link).
   - The native library will be placed in metrics-collector-jni/target/native/.
2. Package Spring Boot Application (metrics-collector-app):
   - Maven build in metrics-collector-app module will package the Java code, resources, and dependencies into an executable JAR file using spring-boot-maven-plugin.
   - Native Library Packaging: The native library from metrics-collector-jni/target/native/ should be included in the Spring Boot JAR artifact. This could be done by copying it into the JAR structure using Maven plugins during the package phase, perhaps placing it within a native/ directory inside the JAR.
3. Parent POM (metrics-collector-parent): Orchestrates the build process for both modules.

10.2. Deployment Considerations:

- Running the Application: To run the packaged JAR, the Java Virtual Machine needs to be able to find the native library.
  - java.library.path: Setting the java.library.path system property when launching the JAR is a common approach. This property needs to point to the directory where the native library resides (either externally on the filesystem or within the JAR).
  - Library Loading from JAR (Advanced): A more robust deployment approach would be to extract the native library from within the JAR to a temporary location at runtime and then load it. This involves more complex JAR manipulation and native library extraction logic.
- Platform-Specific Deployment (Future Enhancement): For different operating systems, different native libraries will be built (.so for Linux, .dll for Windows, .dylib

for macOS). The deployment process needs to account for this, potentially requiring platform-specific packaging and distribution.

11. Future Enhancements (Optional)

- Expanded Metric Set: Add more system metrics (disk I/O, network statistics).
- Metric History and Storage: Integrate an open-source monitoring toolkit (e.g., utilizing InfluxDB or Prometheus) to store historical metric data, enabling trend analysis and time-series visualizations.
- Advanced Web UI:
    - Real-time metric updates using WebSockets or Server-Sent Events (SSE).
    - Interactive charts and graphs for data visualization (using JavaScript charting libraries).
    - User configuration for selecting metrics to display and refresh intervals.
- Alerting: Implement alerting based on metric thresholds.
- Remote Monitoring: Enable remote access to the application and metric data with appropriate security measures (authentication, authorization).
- Containerization: Utilize Docker to ensure availability and portability of the application, regardless of user environment. This would ensure consistent packaging and configuration.

12. Open Issues and Risks

- JNI Complexity: JNI development can be complex and error-prone. Careful attention to memory management, data type conversion, and exception handling are required in the JNI layer.
- Platform Dependency: Developing and maintaining platform-specific C/C++ code for metric collection adds complexity. Thorough testing on target platforms is essential.
- Native Library Build Integration: Setting up a robust and cross-platform build process for the native library within the Maven build environment can be challenging.
- Error Handling in Native Code: Robust error handling in the C/C++ code is critical to prevent crashes or unexpected behavior.
- Security Considerations for Future Enhancements: If remote access or more advanced features are added, security will become a more significant concern.