Name	**Student ID**	**Email Address**
Anastasiia Nemyrovska	261114550	anastasiia.nemyrovska@mail.mcgill.ca
Mohammed Elsayed	261053266	mohammed.elsayed2@mail.mcgill.ca

ECSE 429 SOFTWARE VALIDATION TERM PROJECT

PART C - NON-FUNCTIONAL TESTING OF REST API

1. SUMMARY OF DELIVERABLES

For the last part of the project, performance and static analyses were conducted.

As part of performance testing, experiments were conducted to measure the time taken to add, delete, and modify objects as their count increased. System resource utilization (CPU and memory) were tracked during each experiment.

As part of static analysis, source code was reviewed with SonarQube's community edition, focusing on identifying code smells, technical debt, and risks related to complexity or maintainability. Recommendations for improving code quality and reducing potential future risks were made.

Key updates to the unit test suite and tools include:

- *RandomObjectGenerator.java*: Incorporated JavaFaker to populate API objects with random data for performance testing.
- *PerformanceUtils.java*: Added utility methods to monitor CPU usage and available memory during performance experiments using OperatingSystemMXBean and ManagementFactory.
- *CsvWriter.java*: Introduced functionality to log performance results to CSV files, namely todo_performance_results.csv and project_performance_results.csv.

The updated unit test suite can be found in the repository.

A video of the tests being executed in random order can be found here.

2. IMPLEMENTATION OF PERFORMANCE TEST SUITE

2.1 Todo

2.1.1 Summary of Results

The performance testing of the todos endpoint reveals that transaction times for all operations increase steadily as the number of objects grows (see Figure 1). CPU usage for these operations appears inconsistent, with occasional spikes, especially during Update operations (see Figure 2). Update operations, particularly with the POST method, stand out as consuming more memory compared to others, with significant spikes at various object counts (see Figure 3). These irregularities highlight inefficiencies in how the system handles transactions.

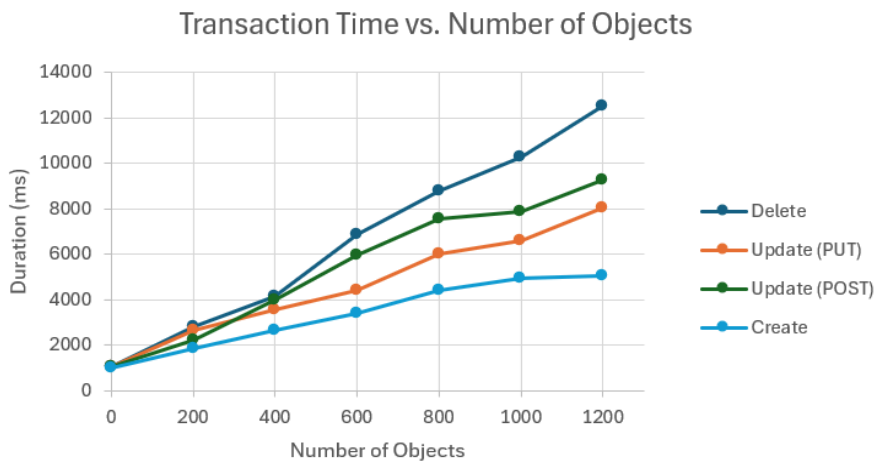

Figure 1. The scatter chart of transaction time versus number of manipulated “todo” objects.

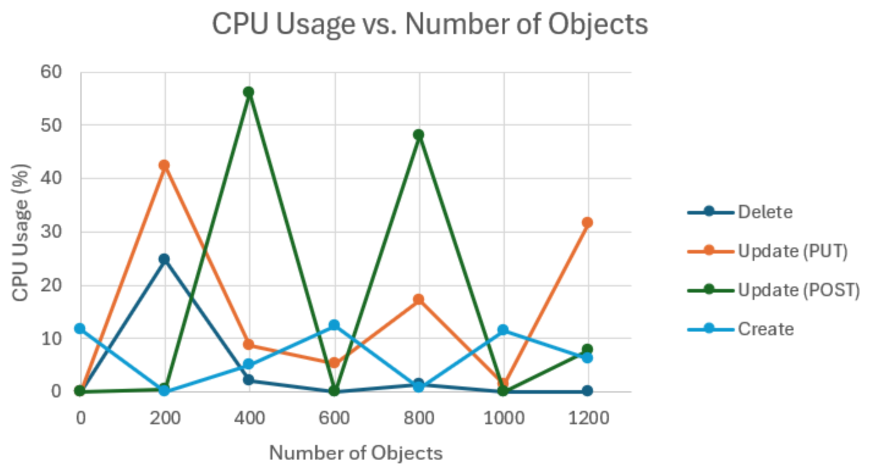

Figure 2. The scatter chart of CPU usage versus number of manipulated “todo” objects.

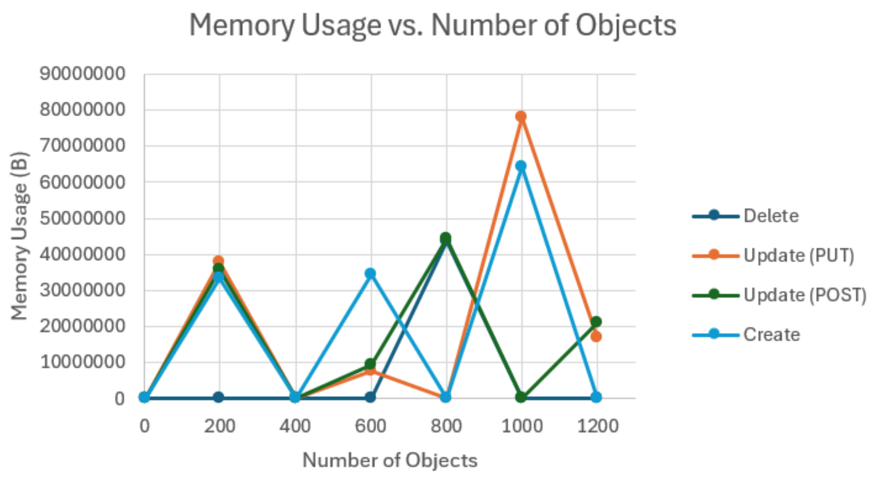

Figure 3. The scatter chart of memory usage versus number of manipulated "todo" objects.

2.1.2 Recommendations for Code Enhancements

To address these performance issues, optimizing the Update operations (POST) should be prioritized, focusing on managing memory usage to reduce the observed spikes. Reviewing data structures and algorithms for efficiency in handling bulk updates may mitigate this issue. Database queries should be optimized, ensuring proper indexing and batch processing. Introducing asynchronous processing for Create and Delete operations could help reduce transaction times and improve overall performance during high-load scenarios.

2.1.3 Performance Risks

The most significant risks for the todos endpoint include the memory spikes observed during Update operations, which could lead to system instability under heavy loads. The inconsistent CPU usage indicates inefficient processing paths or potentially redundant computations, which could worsen as the system scales. Lastly, the steady increase in transaction times poses a risk to user experience, especially when dealing with a large number of objects, as operations might become noticeably slower.

2.2 Project

2.2.1 Summary of Results

The project endpoint demonstrates a proportional increase in transaction times across operations as the number of objects grows, with Delete operations being particularly time-intensive (see Figure 1). CPU usage remains relatively low but shows occasional spikes, notably during Create operations (see Figure 2). Memory usage for Create operations is marked by significant fluctuations, particularly at higher object counts, suggesting inefficiencies in how these operations handle larger workloads (see Figure 3). Overall, the endpoint shows clear signs of scalability challenges as object numbers increase.

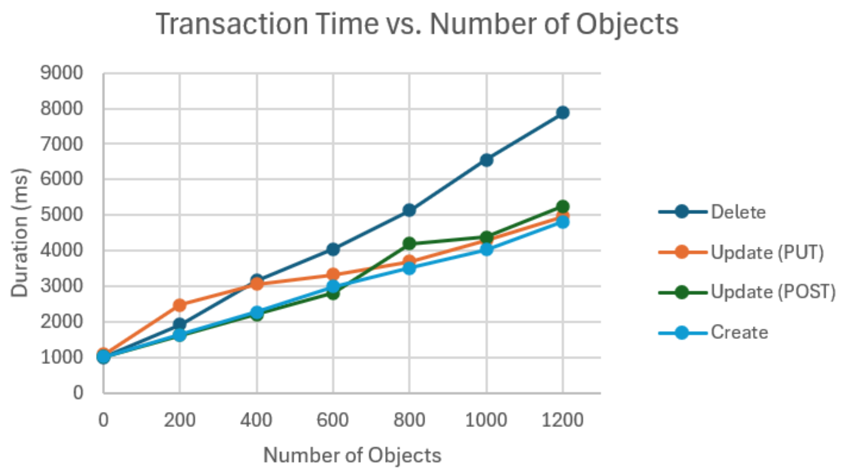

Figure 4. The scatter chart of transaction time versus number of manipulated "project" objects.

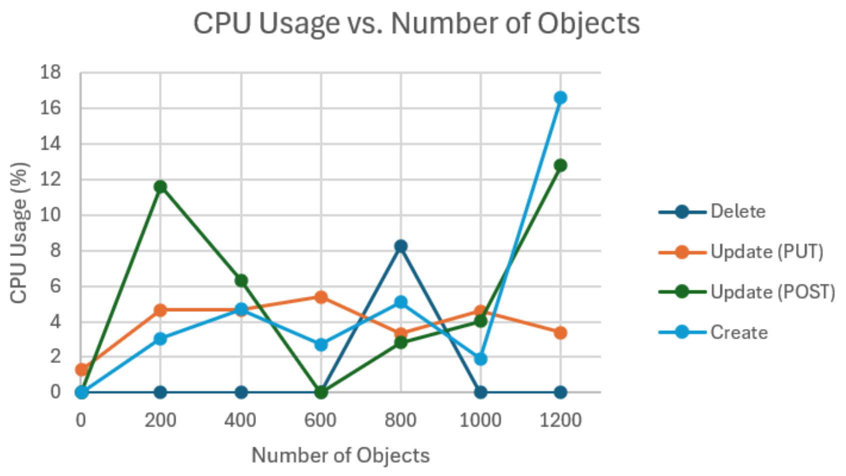

Figure 5. The scatter chart of CPU usage versus number of manipulated "project" objects.

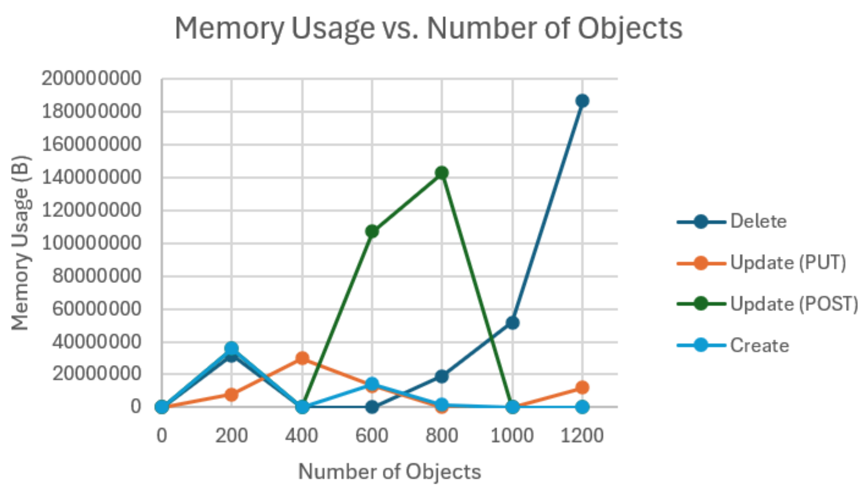

Figure 6. The scatter chart of memory usage versus number of manipulated "project" objects.

2.2.2 Recommendations for Code Enhancements

Optimizing the Delete operation should be a primary focus, as its high transaction times could be improved by implementing batch deletion or reducing redundant steps in the deletion pipeline. Memory overhead during Create operations can be addressed by implementing memory pooling or optimizing serialization and deserialization processes. Additionally, profiling tools should be used to identify specific bottlenecks in both Create and Delete operations.

2.2.3 Performance Risks

The high transaction times for Delete operations present a risk of timeout errors or degraded system responsiveness in real-world scenarios with large datasets. Similarly, the memory usage spikes during Create operations could result in crashes or unresponsiveness during periods of high activity. Although CPU usage is relatively low overall, the occasional spikes suggest inefficiencies that might worsen with increased usage, potentially leading to performance bottlenecks under heavy loads.

3. IMPLEMENTATION OF STATIC ANALYSIS WITH SONARQUBE

To implement static analysis using the SonarQube Community Edition, the application was set up by downloading and extracting the SonarQube ZIP package. Once installed, the SonarQube server was started locally, and the web interface was accessed to configure the analysis. The project source code was scanned using the Maven Sonar Scanner. A sonar-project.properties file was configured to define the project key.

The complexity, code statement count, summary of issues highlighted by the analysis, and recommendations to improve the overall codebase are provided below. Duplicate issues have been omitted. For easy navigation, the issues have been divided into subcategories by their severity (high, medium, low) and type (bug/code smell).

Complexity

Cyclomatic Complexity: 1,784
Cognitive Complexity: 1,467

Size
New Lines: 0
Lines of Code: 9,428
Lines: 13,390
Statements: 4,172
Functions: 779
Classes: 137
Files: 140
Comment Lines: 994
Comments (%): 9.5%

Issues

Severity: High

Type: *Bug*

Save and re-use this "Random": instead of creating a new Random instance each time, the code should reuse a single Random instance.

Type: *Code Smell*

Refactor this method to not always return the same value: ensure methods consistently produce varying outputs depending on inputs.
Refactor this method to reduce its Cognitive Complexity: methods exceed the allowed complexity, making them hard to read and maintain, and should be simplified by breaking into smaller functions.
Define a constant instead of duplicating literals: avoid hardcoding repeated strings by creating constants.
Make the method "static" or remove unnecessary usage of fields: static context is more efficient for methods that don't rely on instance variables.
Correct "&" to "&&": use logical AND for proper conditional evaluation and avoid potential issues.

Add missing enum constants or a default case in switch statements: ensure switch statements are exhaustive or have a fallback case for better reliability.
Add at least one assertion to test cases: test cases lack validation; include assertions to ensure code correctness and robustness.
Reduce the duplication of UI elements like HTML snippets: move duplicated code into reusable constants.

Severity: Medium

Type: *Bug*

Strings and Boxed types should be compared using equals(): avoid using == for Strings or Boxed types (e.g., Integer, Double) as it checks reference equality instead of value equality.
A "NullPointerException" could be thrown: variables like objectValue and relationshipToUse are nullable and could lead to runtime exceptions if not properly checked before use; add null checks to ensure reliability.
The return value of "format" must be used: ignoring the return value of a method like format can lead to unintended behavior; ensure the result is captured and utilized correctly.

Type: *Code Smell*

Replace this use of System.out by a logger: use a logging framework instead of System.out.
Remove this block of commented-out lines of code: redundant commented-out code should be removed.
Remove this unused method parameter: unused parameters should be eliminated.
Remove this unused private method: methods should be deleted if they are unused to avoid dead code.
Remove this useless assignment: avoid redundant assignments to variables like challenger or response, which serve no purpose in the logic.
Merge this if statement with the enclosing one: simplify nested or sequential if conditions to make the logic clearer and more maintainable.
Provide the parameterized type for this generic: specify types explicitly for generic objects to avoid runtime errors and improve readability.
Define and throw a dedicated exception instead of using a generic one: avoid using generic exceptions for specific errors; use custom exception types to improve error handling and debugging.
Remove this use of Thread.sleep(): avoid using Thread.sleep() as it is generally considered a bad practice in tests and production code.
Either remove or fill this block of code: empty or incomplete code blocks should be either implemented or removed to avoid confusion.

Replace repeated tests with a single parameterized one: consolidate repetitive tests into parameterized tests for better maintainability and efficiency.
Add a private constructor to hide the implicit public one: add private constructors to utility classes to prevent instantiation.
String contains no format specifiers: remove or correct cases where strings are improperly formatted without specifiers, avoiding potential confusion.
Iterate over the entrySet instead of the keySet: use entrySet for iterating over maps to improve performance by avoiding redundant key lookups.
Remove unused imports/fields: eliminate unused private fields to reduce clutter.
Reduce the number of assertions in a single test: Refactor tests with too many assertions for better readability and focus.

Severity: Low

Type: *Bug*

Cast one of the operands of this multiplication operation to a "long": casting one of the operands to a long ensures that the operation is performed using 64-bit arithmetic, which prevents overflow and improves reliability.

Type: *Code Smell*

Use isEmpty() to check for empty collections: using isEmpty() is more readable and aligns with Java best practices.
Convert maps to EnumMap: maps that use enums as keys can be replaced with EnumMap for better performance and readability.
Complete TODO comments: numerous files contain TODO comments that need to be resolved.
Use StringBuilder for Concatenations: replace string concatenation with StringBuilder for better performance.
Static final fields: multiple fields are recommended to be made static final to improve immutability and readability.
Remove public modifiers in JUnit tests: public modifiers are unnecessarily used in JUnit test methods.
Replace redundant type specifications with the diamond operator (<>): explicit type arguments can be replaced with the diamond operator for better readability.
Replace if-then-else statements with single return statements: simplifying conditionals improves readability.
Refactor long or complex methods: long methods with high complexity are harder to maintain and debug.
Reorder modifiers for specification compliance: modifiers are not in the standard order as per Java Language Specification.

Refactoring redundant variables: temporary variables used unnecessarily before return, immediately return expressions instead of assigning them to variables.
Rename packages for consistency: some package names do not adhere to standard naming conventions and should be renamed.
Refactor "Brain Methods": large, complex methods with high nesting and variable count are hard to maintain; break them into smaller, focused methods to improve clarity.
Remove empty statements: empty statements (;) serve no purpose and reduce code readability; remove these for cleaner code.
Remove redundant boolean expressions: overly complex boolean expressions can be simplified into primitive boolean checks for clarity and maintainability.
Avoid redundant fields in classes: fields like args are better suited as local variables; refactor to keep fields minimal and relevant.
Standardize method and variable naming: non-standard naming in methods or variables reduces readability; rename to follow conventions and improve consistency.
Refactor for validation: missing or unclear validations (e.g., null checks) can lead to runtime errors; add necessary validations to ensure robustness.