Group 98:
Samantha Ines Perez Hoffman (261039555)
Lucy Zhang (261049310)

Summary of Deliverables

The ECSE 429 Software Validation Term project is divided into the following three deliverables:

- Part A: Exploratory Testing of Rest API
- Part B: Story Testing of Rest API
- Part C: Non-Functional Testing of Rest API

Part A: Exploratory Testing of Rest API

We submitted this part of the project on October 8, 2024. In this stage, we were able to run the REST API To-Do List application locally, do some exploratory testing, write a unit test suite, and write a report about our work and findings. Following this part, we were able to use our notes and reports to continue on our work in testing the REST API To-Do List application.

Part B: Story Testing of Rest API

This part was also completed, and it was submitted on November 5, 2024. We were able to test the REST API To-Do List application through story testing. We wrote ten different user stories within Gherkin feature files and ran those user stories using Cucumber. We had identified the expected behavior of the application from the perspective of the user and verified those behaviors using step definitions. We included all details and observations within our report for Part B.

Part C: Non-Functional Testing of Rest API **(this report)**

In this part, we focus on non-functional testing using dynamic analysis and static analysis. For dynamic analysis, we are documenting the cpu usage, available free memory, and time taken to add, delete, and edit a certain number of objects. Since we are a team of two, we will be more specifically testing for Todo and Project objects.

To write the code for creating, editing, and deleting the objects, we adapted the code from our unit tests from Part A. We also created a function to document the data into a csv file into the test resources directory. The code was written in java.

As for testing using static analysis, we used a static analysis tool called Sonar Cube. After setting up this tool, we were given an output of different recommendations to improve the code of the REST API To-Do List application.

As a summary, in this part, we were able to do the following:

1. Adapt unit tests from Part A to make POST, PUT, and DELETE API calls on Todo and Project objects
2. Record the cpu usage, available free memory, and time taken to make API calls for a varying number of objects
3. Create charts using the recorded data
4. Setup Sonar Cube static analysis tool and run static analysis on the application
5. Write a report

The report will include all charts and discussions of the recorded data. The discussions include recommendations to improve the code based on the data observed from both dynamic and static analysis. A video of all the performance tests running will also be attached.

Implementation of Performance Test Suite

We were able to implement the performance test suite within 3 different java files: ProjectPerformanceTests.java, TodoPerformanceTests.java, and PerformanceTestUtils.java. The two files ProjectPerformanceTests.java and TodoPerformanceTests.java include the methods to make the API calls (adapted from Part A unit tests) as well as tests that run those API calls and measure the transaction time, memory usage, and cpu usage on a varying number of objects. The number of objects we tested were 1, 10, 50, 100, 500, and 1000 objects, so we tested the performance of creating, editing, and deleting those numbers of objects.

To measure the transaction time, we simply just used System.currentTimeMillis() to get the start and end time of the operation. As for the memory usage and cpu usage, we used methods from the interface OperatingSystemMXBean to get those data points.

As for recording those data points, we had a method within the PerformanceTestUtils.java file to write the data into a new csv file for every type of operation on a specific object e.g. edit projects. The csv file was outputted within the src/test/resources/performance-test-results directory with a given file name to the method. We used a simple java file writer to create this new csv file. These CSV files were used to make new charts using google sheets.

Charts

Using the data gathered from dynamic analysis, we were able to produce the following charts for creating, editing, and deleting todos and projects:

1. Transaction Time (ms) vs. Number of Objects
2. Memory Usage (MB) vs. Number of Objects
3. CPU Usage (%) vs. Number of Objects

All charts are attached in the appendices for readability purposes of this report. The same charts are also uploaded within the src/test/resources/performance-test-results/charts directory within our repo.

Todo API Group Charts

- Create todo objects: Appendix A, Figures 1-3.
- Edit todo objects: Appendix A, Figures 4-6.
- Delete todo objects: Appendix A, Figures 7-9.

Project API Group Charts

- Create project objects: Appendix B, Figures 1-3.
- Edit project objects: Appendix B, Figures 4-6.
- Delete project objects: Appendix B, Figures 7-9.

Recommendations for Code Enhancements Based on Dynamic Analysis

There are three types of data we are analyzing: transaction time, memory usage, and cpu usage.

Transaction Time Analysis

From observing the Transaction Time (ms) vs. Number of Objects charts for create, edit, and delete todo and project objects, we had noticed that the transaction time increases linearly based on the number of objects. This could become costly when the user decides to perform actions on more objects.

Our recommendation is that some of the operations could be easily optimized to constant time by using a data structure like a hashmap. If the code is written in java, adding to a hashmap would be an O(1) operation as well as getting the value to a hashmap with a key (which would be the object id). After getting whatever value attached to the id, we would be able to edit or delete the object in constant time.

Memory Usage Analysis

After looking at the Memory Usage (MB) vs. Number of Objects charts, we were able to observe that the memory usage mostly stays fairly constant as the number of objects increases. The only two standouts were the memory usage for editing a todo and deleting a project. Those seemed to linearly increase after 500 objects.

To potentially improve memory usage, we could optimize each operation where no unnecessary memory is being retained. Another way to optimize memory usage is through using appropriate data structures for different usages.

CPU Usage Analysis

After analyzing the CPU Usage (%) vs. Number of Objects charts, we noticed that there is a significant high CPU usage at the beginning, and the number drops shortly after and plateaus. The high CPU usage at the beginning could potentially be dangerous for users that frequently restart their machine, because it likely means that the code does not do well with CPU usage during startup. As for the lower CPU usage that plateaus even as the number of objects increase, it could potentially mean that the code is bottlenecked somewhere, which could potentially lead to spikes in latency.

Some recommendations we could give to improve CPU usage is through firstly optimizing the startup CPU usage. We know that writing code to run in a more parallel manner could help with this. Another improvement we can make is investigating the potential bottleneck for the CPU usage to plateau. Being able to use the CPU effectively and finding the reason why the CPU usage plateaus could improve the overall throughput of the application.

Performance Risks

Here are some performance risks in terms of transaction time, memory usage, and CPU usage.

Transaction Time

As discussed in the previous section, the runtime seems to linearly increase i.e. O(n). This could be dangerous as the number of objects we are operating in increases. This could lead to spikes in latency.

Memory Usage

Memory usage doesn't have any significant performance risks, but it is important to check that there are not any memory leaks or unnecessary data retention because we want to optimize memory usage.

CPU Usage

The high CPU usage at startup could be dangerous if a system restarted the application at a frequent rate. There could also be delays if there exists a bottleneck and a large number of operations are sent to the application.

Implementation of Static Analysis

For the static analysis portion of this project, we utilize a tool called Sonar Cube (we used the community edition of the tool). This allowed us to study the source code of the Rest API to do list manager.

In order to use this tool, we first were required to install the Sonarqube community edition along with the Sonar-Scanner. We then configured the Sonarqube environment variable in our computer's bash_profile file. Afterwards we were able to start the Sonarqube tool and log in into the Sonarqube dashboard. In the dashboard, we created a new project following the instructions the program gave us. After the project was set up, we were able to run the static analysis of the source code using the command below from the "thingifier-1.1.0-apichallenges/" folder in our local terminal .

```
mvn clean verify sonar:sonar \ -Dsonar.projectKey=ECSE429-PartC \
-Dsonar.projectName='ECSE429-PartC' \ -Dsonar.host.url=http://localhost:9000 \
```

After running the analysis, we were able to look through the Sonarqube scan results from localhost:9000 in our browser. These results we'll be discussed in further detail in the next section of the report in the shape of recommendations.

Recommendations for Code Enhancements Based on Static Analysis

The static analysis performed using Sonarqube revealed multiple recommendations that could be addressed in order to improve the program's code. Resolving these issues will enhance the code in various ways making it easy to maintain as well as improving its performance, readability and even resolving some issues. When running Sonarqube, it generated 783 issues: 89 high severity issues, 240 medium severity issues and 454 low severity issues with an estimated prediction of 8d 6h of effort to address all of them. We'll provide below a detailed breakdown of the main recommendations Sonarqube generated organized by severity from high to low.

High Severity Issues

Sonarqube has revealed 89 high severity issues that should ideally be addressed first. In terms of error handling, it has highlighted multiple snippets of code where there is a missing enum constant or a missing default case to switch statements. These are important to address as they can potentially lead to unhandled scenarios which could then bring runtime errors or unexpected behaviour. In terms of design, sonarqube has highlighted multiple opportunities where it is best to define a constant instead of duplicating a literal value. For instance, this happens a lot when it comes to defining resource paths on the web app like "todos/" or "application/xml". Resolving that issue would improve code maintainability. Finally, the static analysis tool has also found multiple instances where we could refactor a method to reduce its cognitive complexity. It seems sometimes the code is written in an overcomplicated way that can potentially lead to bugs and a tough coding experience to work in.

Medium Severity Issues

Sonarqube has revealed 240 medium severity issues that should be ideally addressed whenever there is some time to spare. There are multiple instances of bad practices done throughout the codebase. For instance, it's best practice to use a logger instead of System.out and to use the "equals()" method when comparing Strings and Boxed types. It's also best practice to add the missing deprecated javadoc tag when code is deprecated. Sonarqube has also highlighted multiple cases of unused code such as unused method parameters and commented lines of code. Removing unnecessary code can help clean out the codebase a lot and make it easy to maintain. It also offered a potential solution to improve performance through iterating over the "entrySet" instead of the "keySet". Furthermore, the static analysis tool also recommended avoiding the use of generic exceptions by instead using a personalized exception for the problem in question. This would facilitate debugging by helping find the source of the issues.

Low Severity Issues

Sonarqube has revealed 454 low severity issues that are not super important to address right away but potentially in the future if there are no more pressing issues to deal with. For starters, there are multiple imports declared throughout the codebase that are unused, so it would be best practice to remove them for code cleanliness. The tool also spotted multiple fields with bad naming practices so it is recommended to go over all the field names and modify them accordingly. Lastly, there are a few ways that we could improve the program's performance. For instance, we could use an EnumMap instead of a regular Map or use a StringBuilder instead of String in the code snippets where we're in the process of constantly modifying a String.

Appendix A: Operations on Todo Object Charts

Figure 1: Transaction Time (ms) vs. Number of Objects for Create Todo

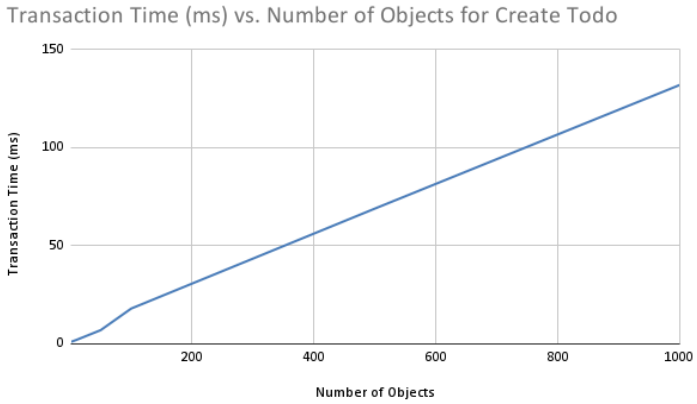


Figure 2: Memory Usage (MB) vs. Number of Objects for Create Todo

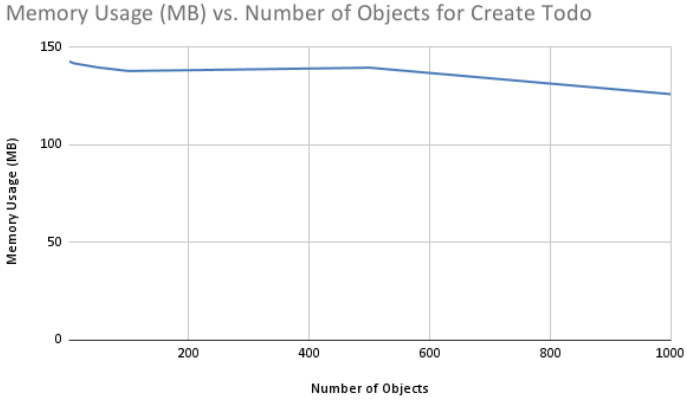


Figure 3: CPU Usage (%) vs. Number of Objects for Create Todo

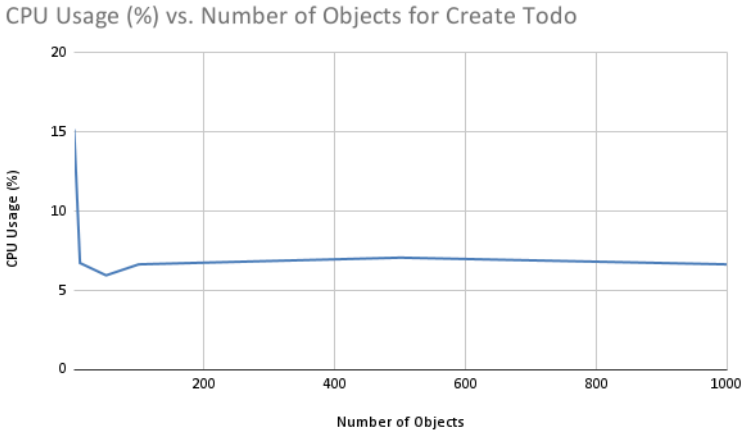


Figure 4: Transaction Time (ms) vs. Number of Objects for Edit Todo

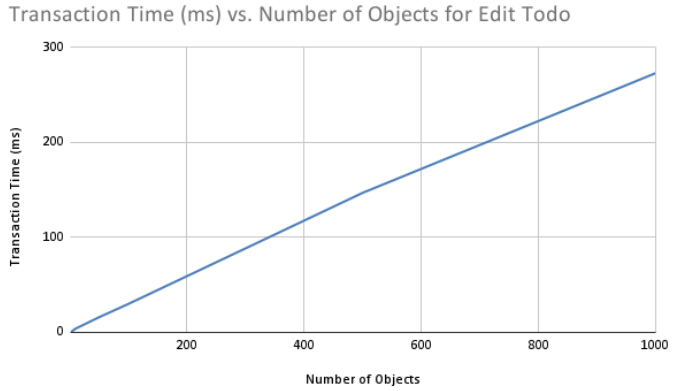


Figure 5: Memory Usage (MB) vs. Number of Objects for Edit Todo

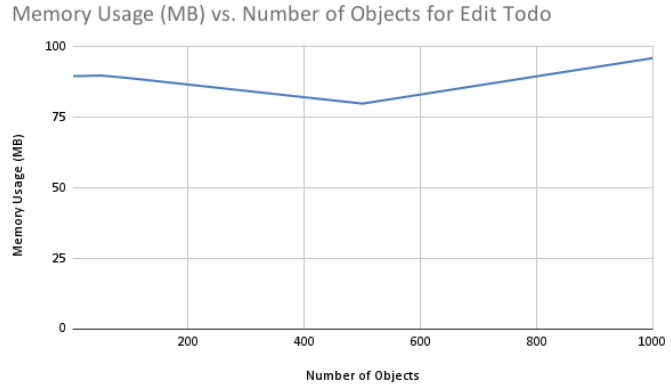


Figure 6: CPU Usage (%) vs. Number of Objects for Edit Todo

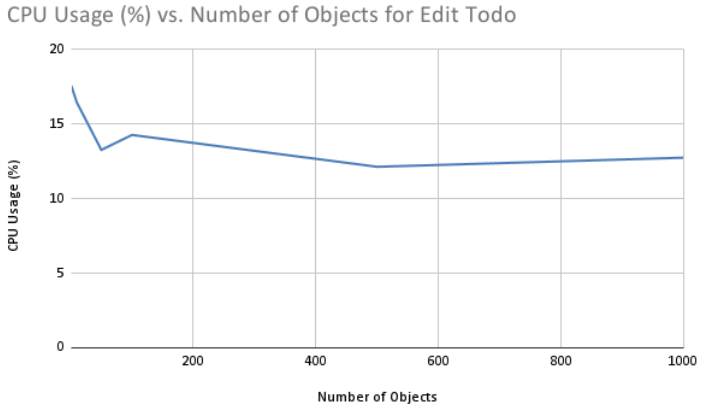


Figure 7: Number of Objects and Transaction Time (ms) for Delete Todo

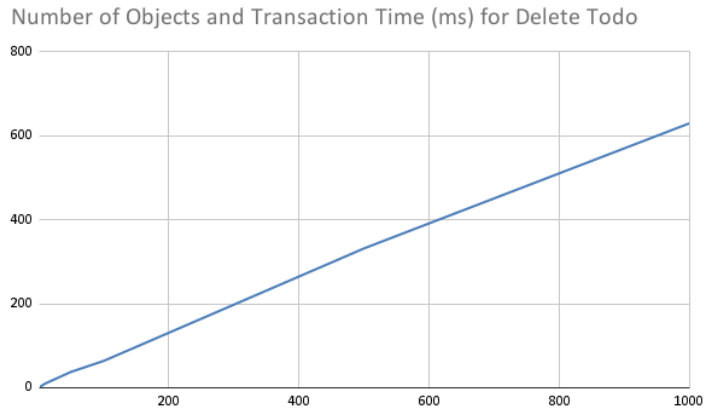


Figure 8: Memory Usage (MB) vs. Number of Objects for Delete Todo

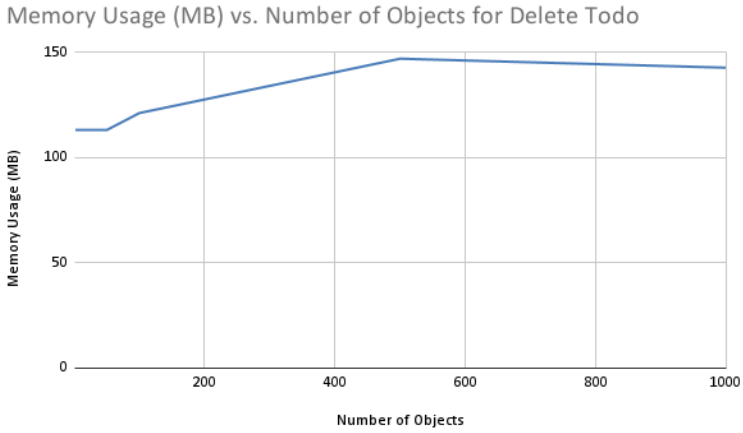
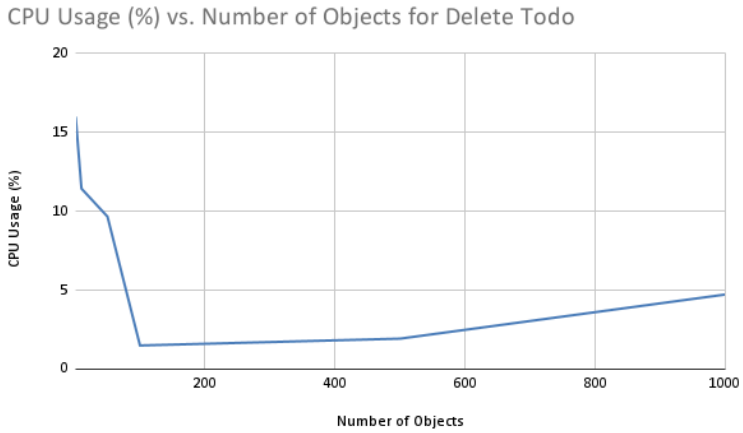


Figure 9: CPU Usage (%) vs. Number of Objects for Delete Todo



Appendix B: Operations on Project Object Charts

Figure 1: Transaction Time (ms) vs. Number of Objects for Create Project

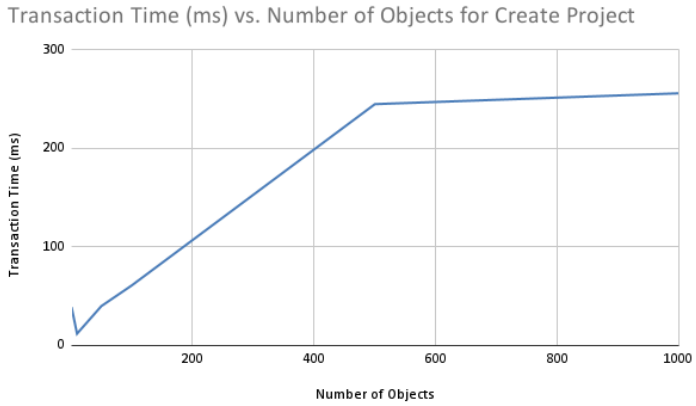


Figure 2: Memory Usage (MB) vs. Number of Objects for Create Project

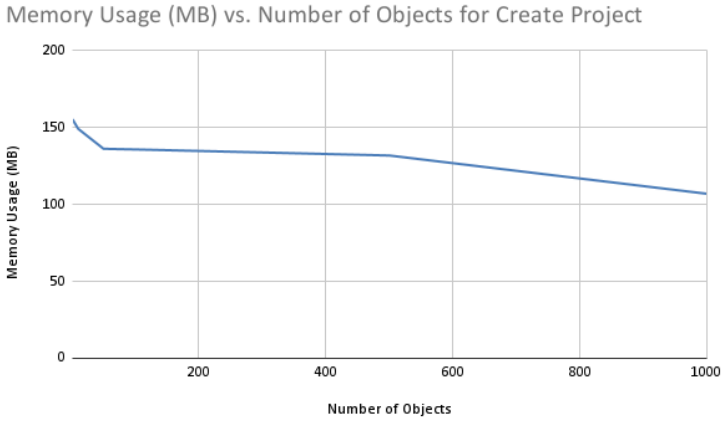


Figure 3: CPU Usage (%) vs. Number of Objects for Create Project

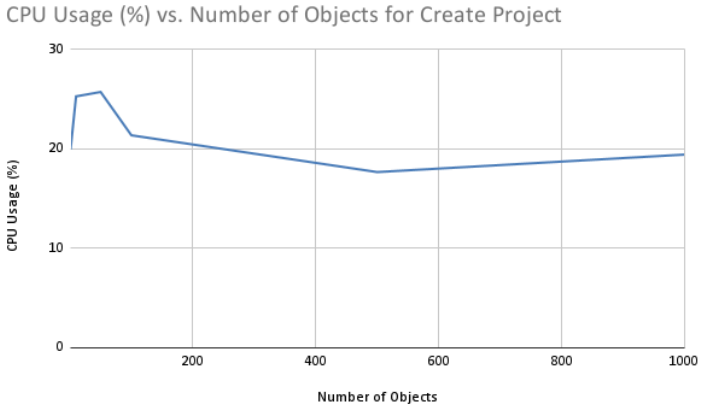


Figure 4: Transaction Time (ms) vs. Number of Objects for Edit Project

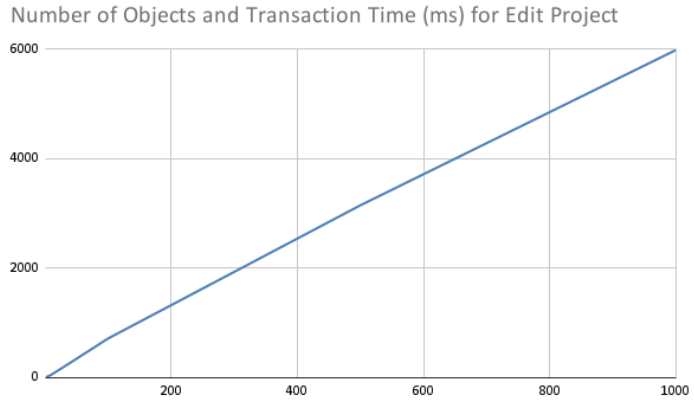


Figure 5: Memory Usage (MB) vs. Number of Objects for Edit Project

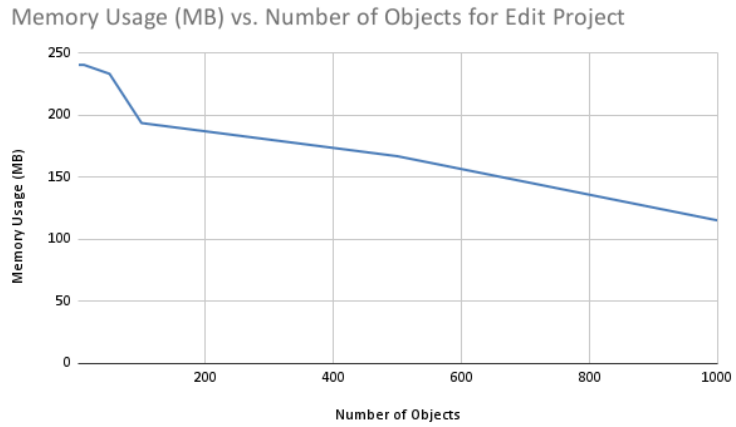


Figure 6: CPU Usage (%) vs. Number of Objects for Edit Project

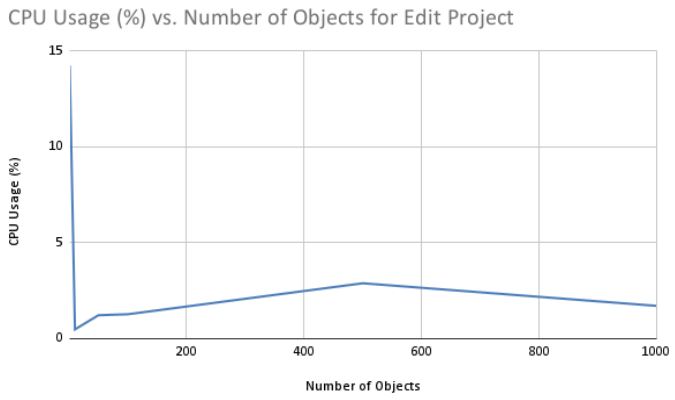


Figure 7: Number of Objects and Transaction Time (ms) for Delete Project

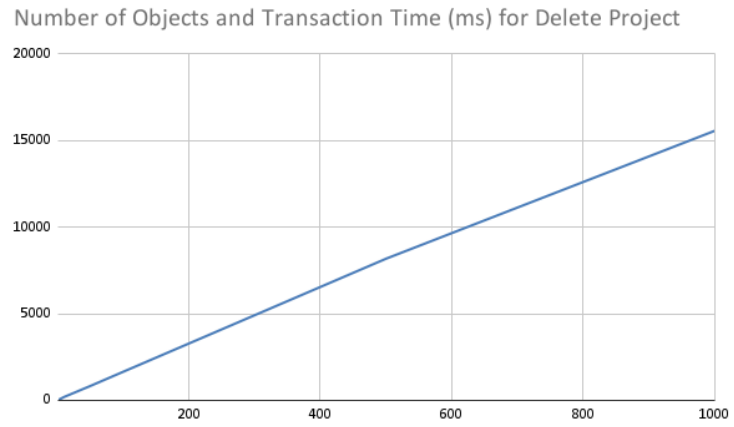


Figure 8: Memory Usage (MB) vs. Number of Objects for Delete Project

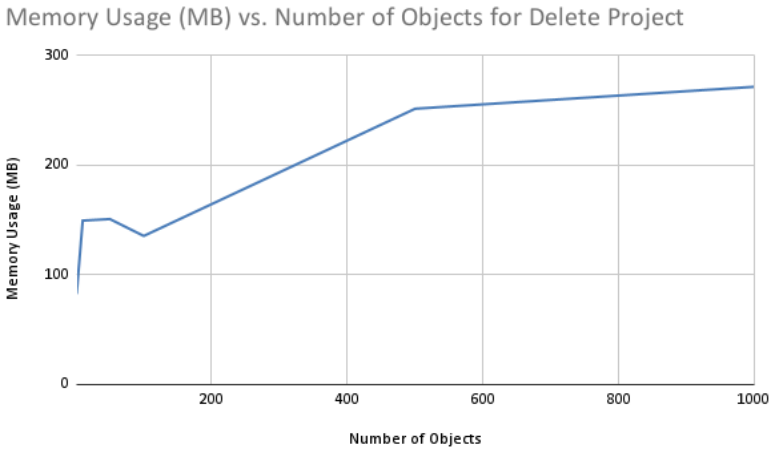


Figure 9: CPU Usage (%) vs. Number of Objects for Delete Project