429 Project Report Summaries - Part 3

Names of testers

Ying Fei Zang (261054638) ying.zang@mail.mcgill.ca

YunShan Nong (261055472) yunshan.nong@mail.mcgill.ca

Yu Tong Hu (261051311) yu.tong.hu@mail.mcgill.ca

Folder Directives

/Codes

- TodosTester.py
- ProjectsTester.py
- CategoriesTester.py

/Codes/Charts

- Category_Dynamic_Analysis.xlsx
- Todo_Project_Dynamic_Analysis.xlsx

/Videos

- CategoriesTester.mp4
- SonarCloud
- TodosTester_ProjectsTester.mp4

Summarizes deliverables

Inside `CategoriesTester.py`:

- This test is modified from `testCategories.py`.
- There is one test function called `test_dynamic_category`, which manipulates 10000 instances of `Categories` and collects data of Transaction Time, CPU % Use and Available Free Memory of each of the following three actions: Add, Change and Delete.
- Since 10000 rows of data is difficult for Excel to manage (we have tried and the file crashed very often), we decided to collect the data at an interval of 500 instances. So there are in total 20 rows of data.
- A csv file named `category.csv` is generated at the end of the execution of `test_dynamic_category` with Python Pandas library.

Inside `Category_Dynamic_Analysis.xlsx`:

- The file contains the data of `category.csv` and the graphs generated based on the data collected.

Inside `TodosTester.py`:

- This tester program creates, updates and deletes 10000 todos in the main method when we run it.
- The program calls `create_todos_with_random_data` which creates 10000 todos and collects data like sample time (s), CPU use percentage and memory use (MB) at an interval of 500 instances. The total transaction time for creating todos is the accumulative time it takes, the time is added every time this method is called. Also, to help to update and delete random todos, we need to keep track of all the ids of todos created before. For that, there is an array `created_ids` which we add the returned id of created todos.
- Similar methods to update and delete todos: `update_todos_with_random_data` and `delete_random todos`. Where data are printed at every 500 instances.
- Whenever we delete a todos, the method makes sure that the random id to todos is removed from the array `created_ids`.

Inside `ProjectsTester.py`:

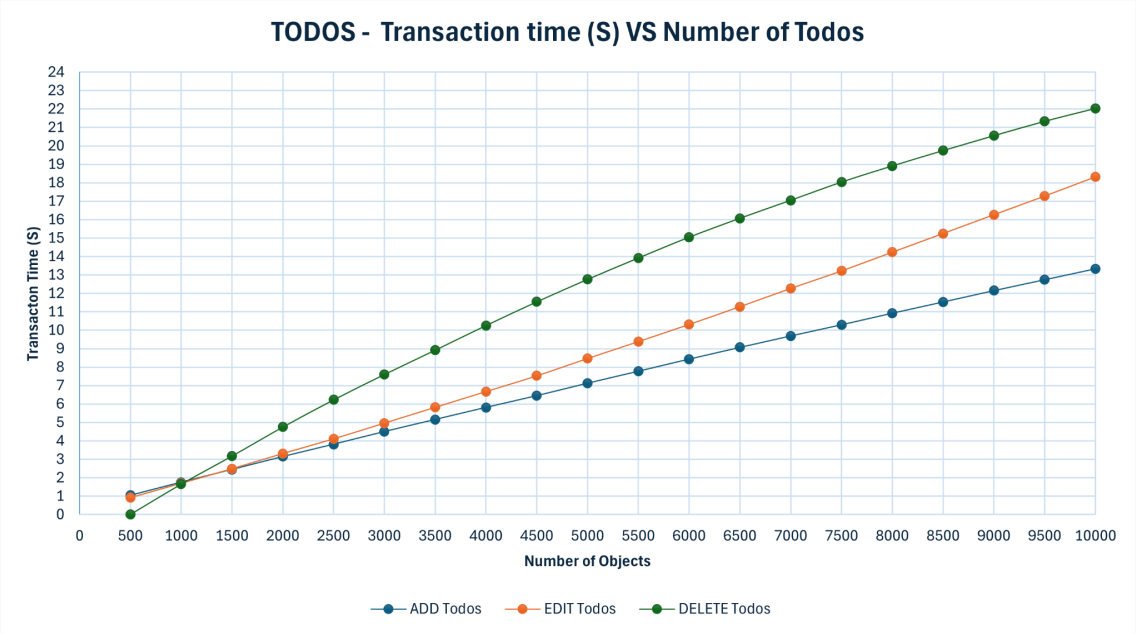
- Same structure with `TodosTester.py`, this tester program creates, updates and deletes 10000 instances of projects in the main method when we run it.

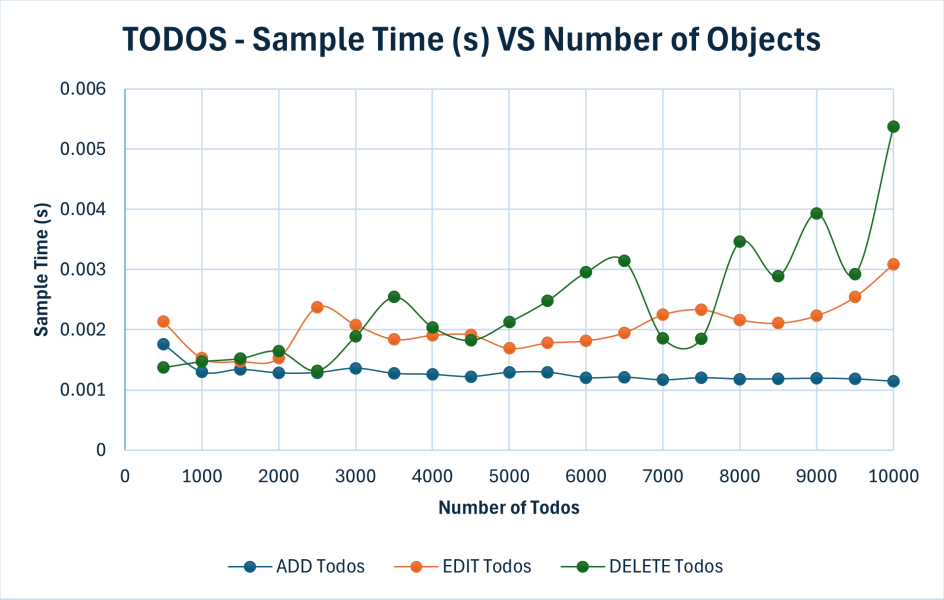
Inside `Todo_Project_Dynamic_Analysis.xlsx`:

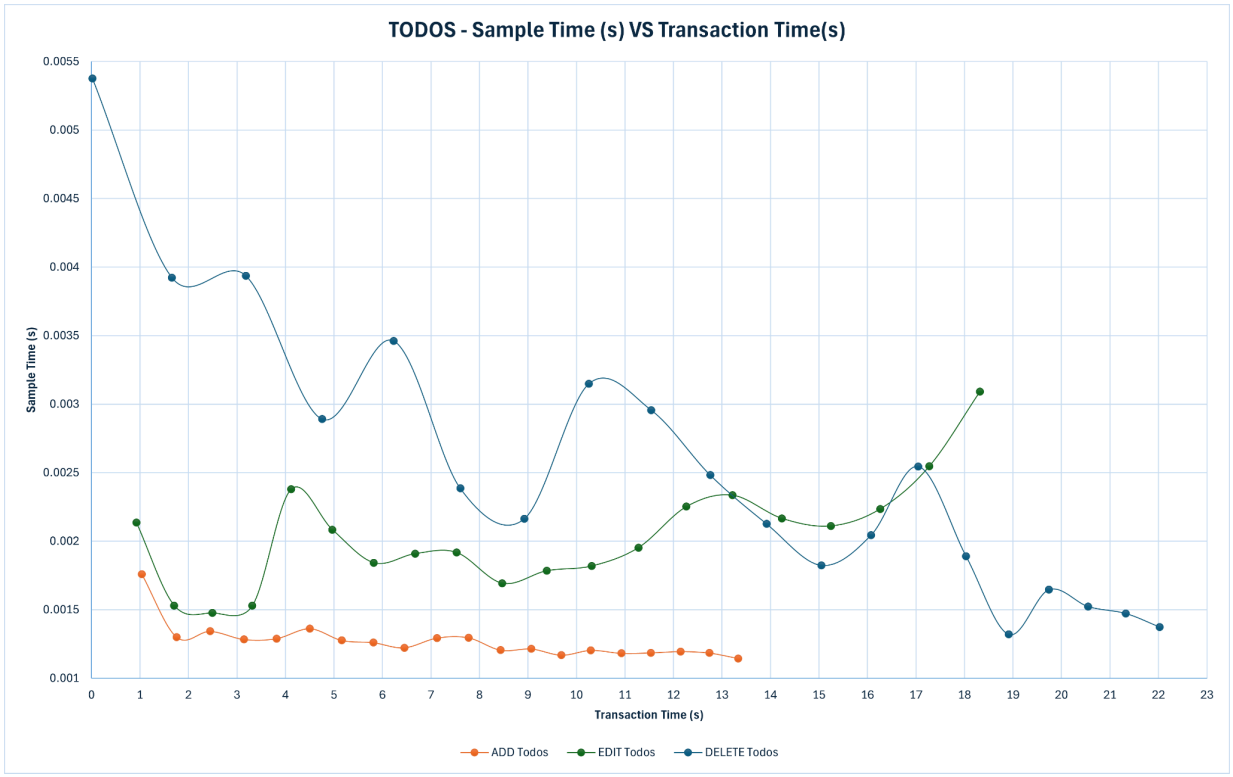
- This Excel files have 2 sheets, one for todos, the other one for projects.
- The data collected with `TodosTester.py` and `ProjectTester.py` have been exported to this file to generate different graphs. Including sample time, transaction time, CPU use and memory use for ADDING, EDITING, DELETING todos and projects.

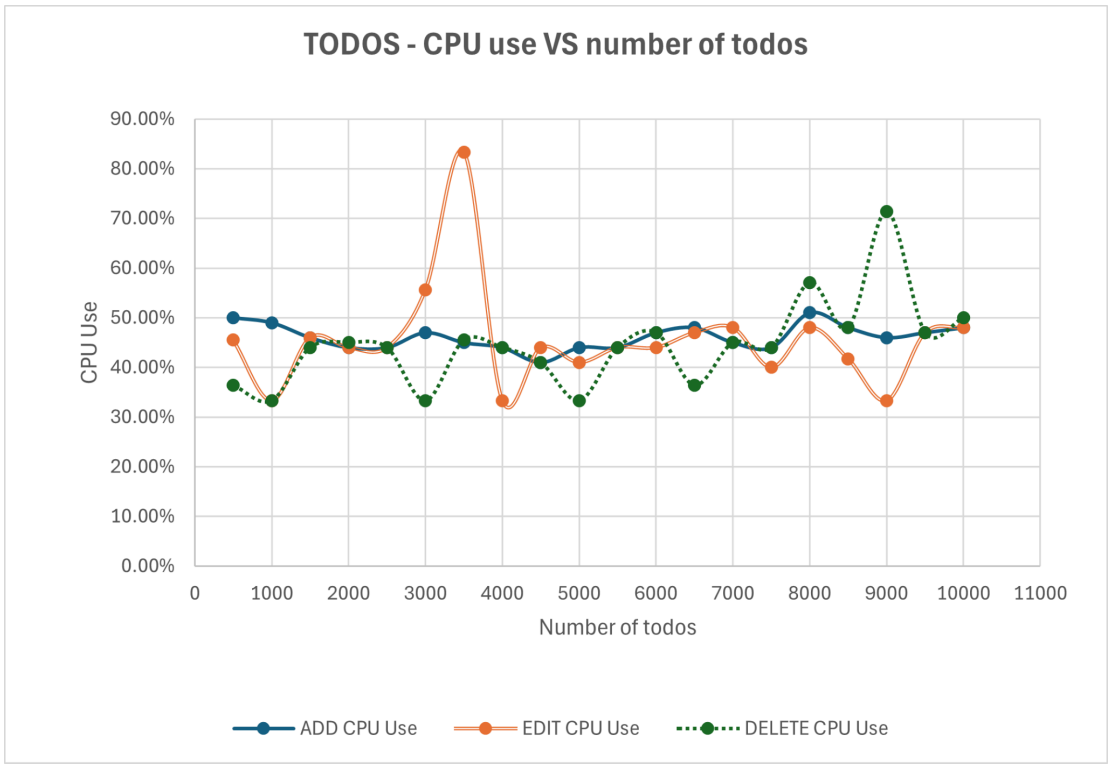
Describes implementation of performance test suite

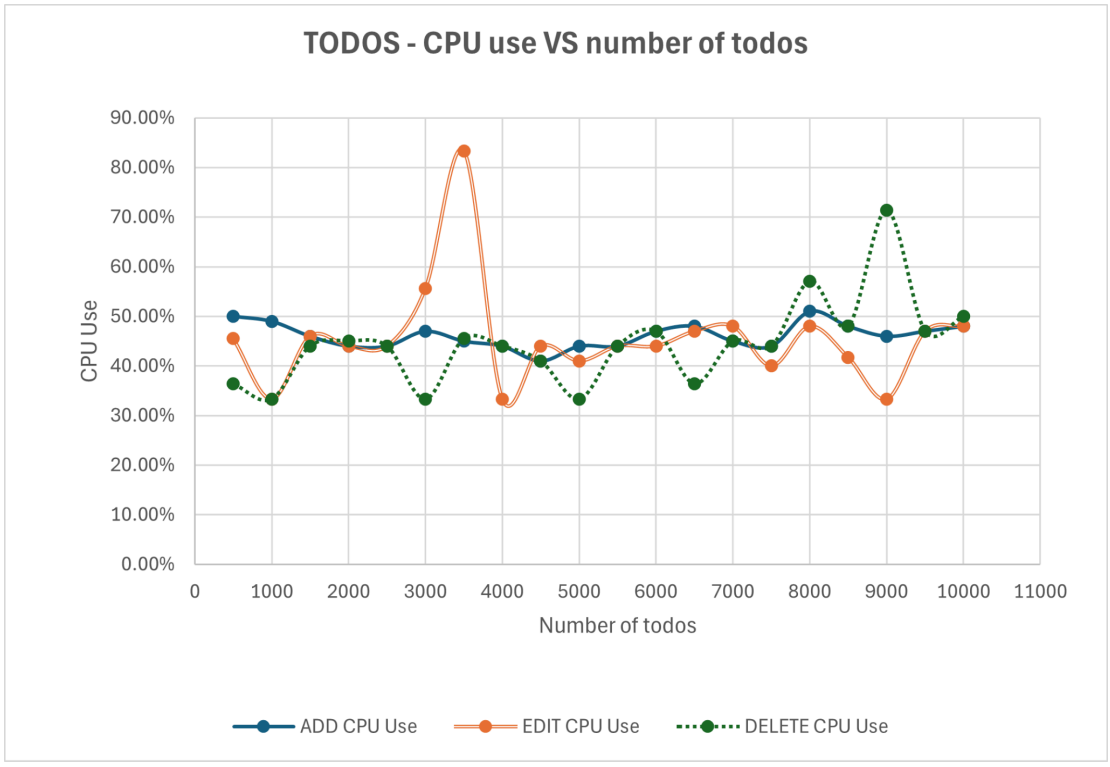
Todos Charts:



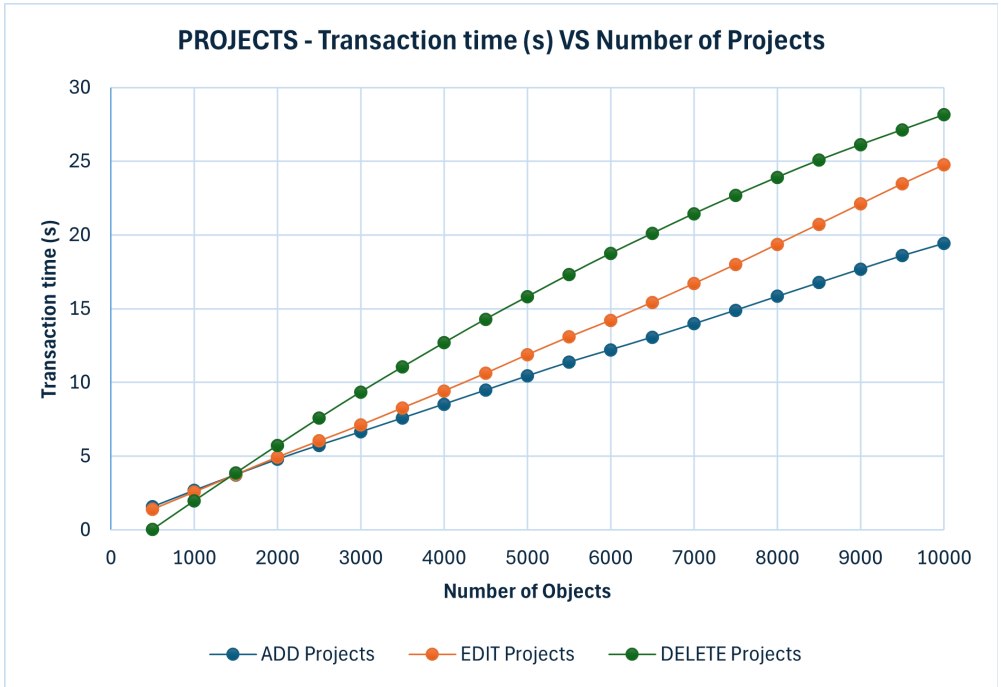


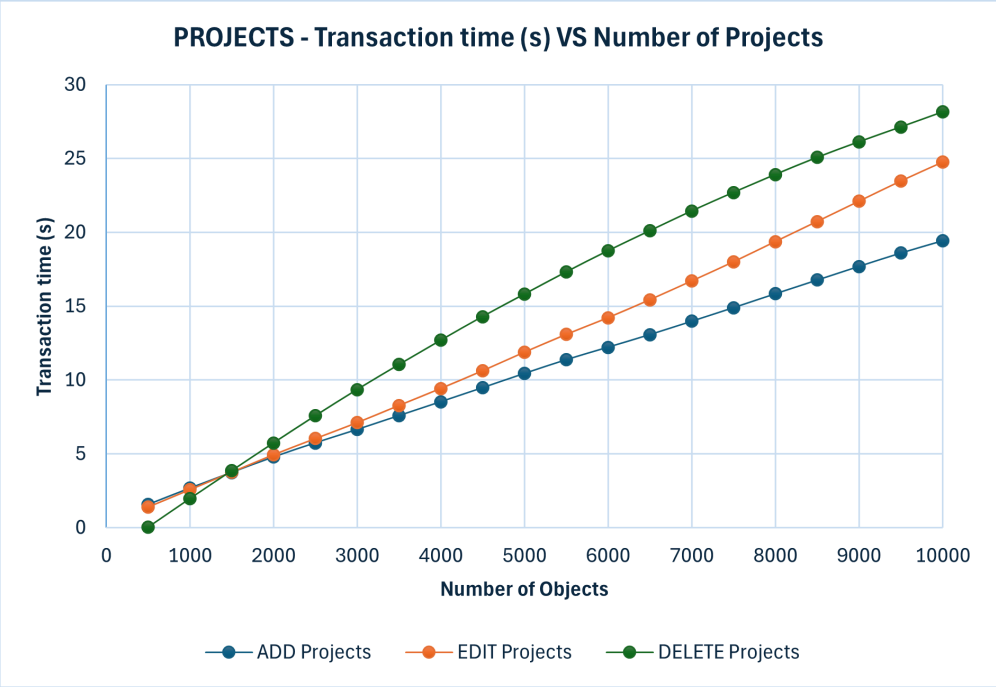

TODOS - Sample Time (s) VS Transaction Time(s)
Sample Time (s)
0.0055
0.005
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Transaction Time (s)
ADD Todos
EDIT Todos
DELETE Todos

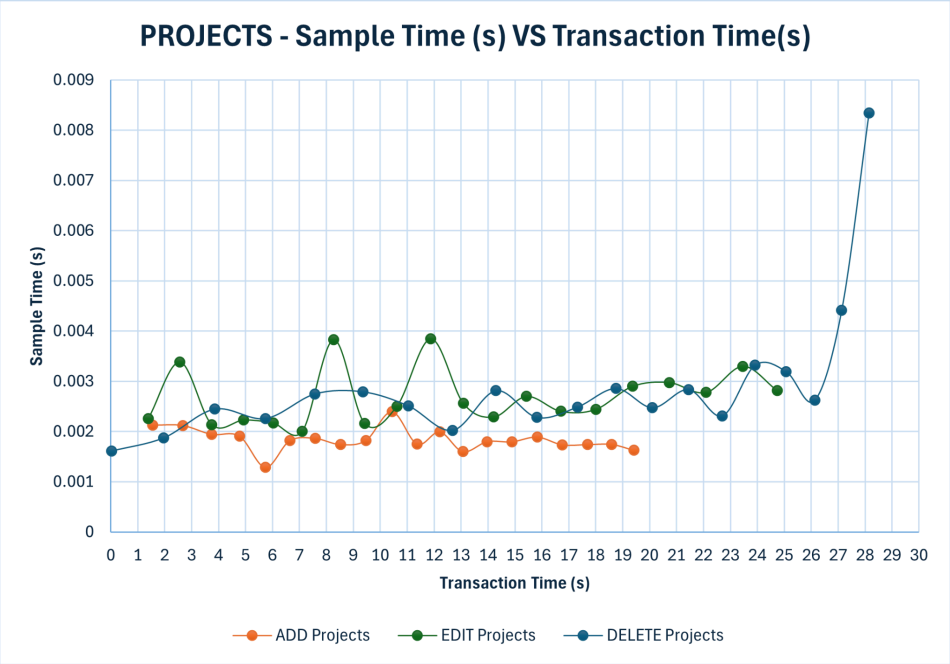

TODOS - CPU use VS number of todos
CPU Use
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
0 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000
Number of todos
ADD CPU Use
EDIT CPU Use
DELETE CPU Use

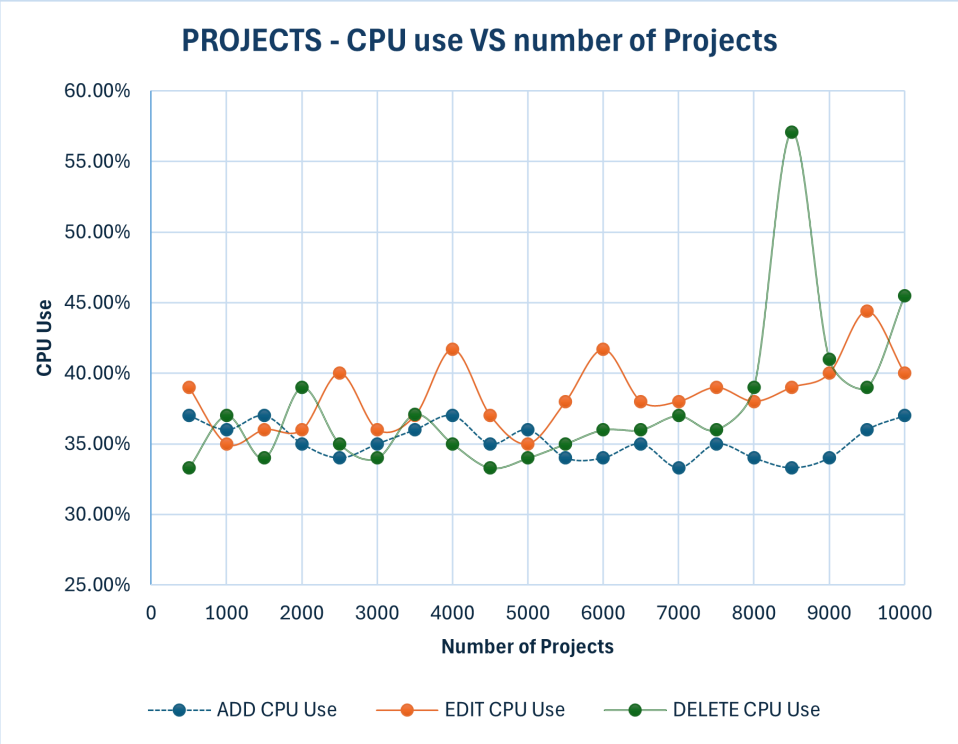


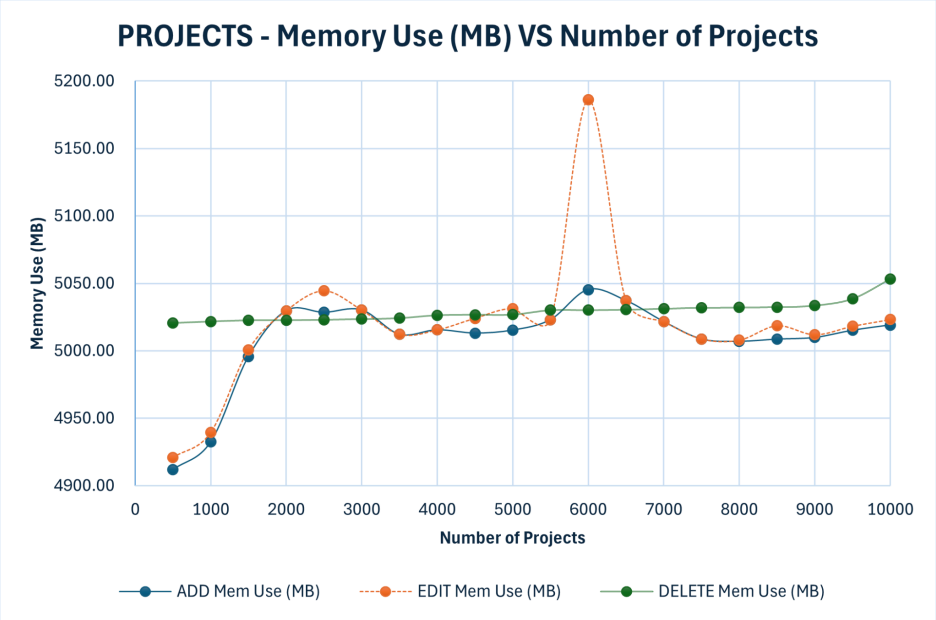
Projects Charts:



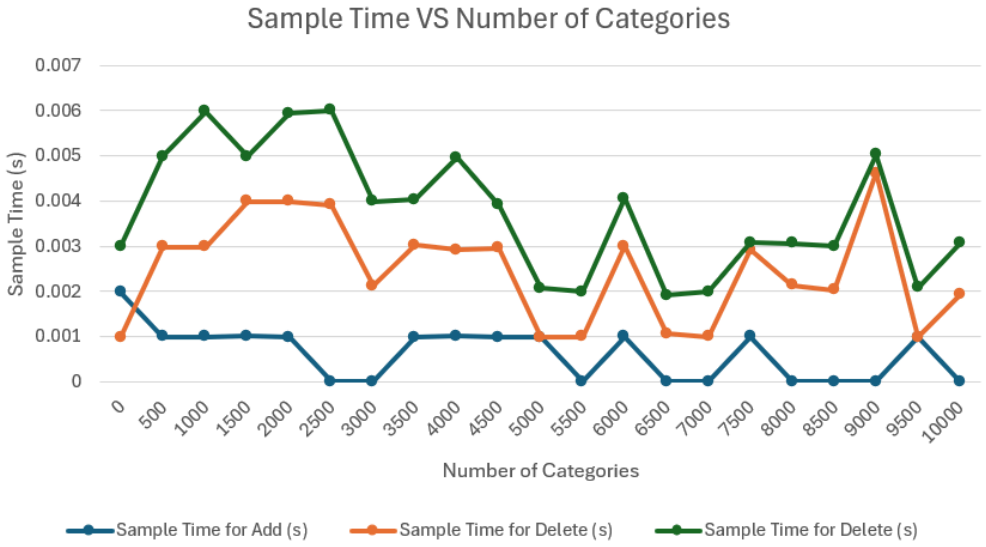

PROJECTS - Transaction time (s) VS Number of Projects
Transaction time (s)
0
5
10
15
20
25
30
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of Objects
ADD Projects
EDIT Projects
DELETE Projects

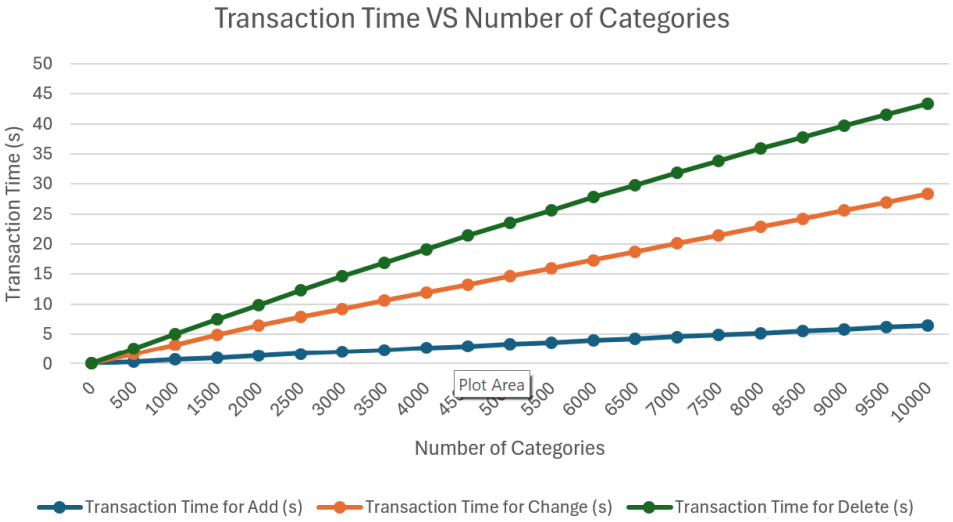

PROJECTS - Sample Time (s) VS Transaction Time(s)
Sample Time (s)
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
0.009
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Transaction Time (s)
ADD Projects
EDIT Projects
DELETE Projects

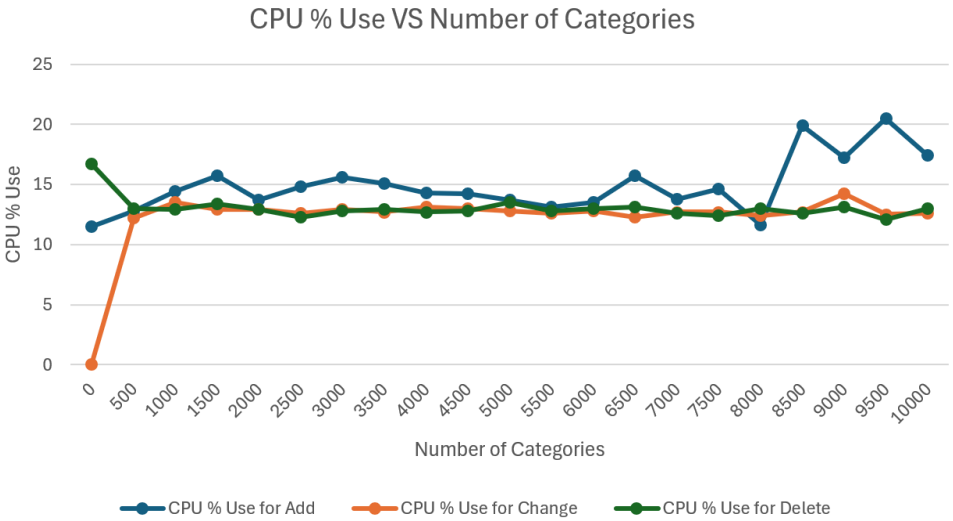


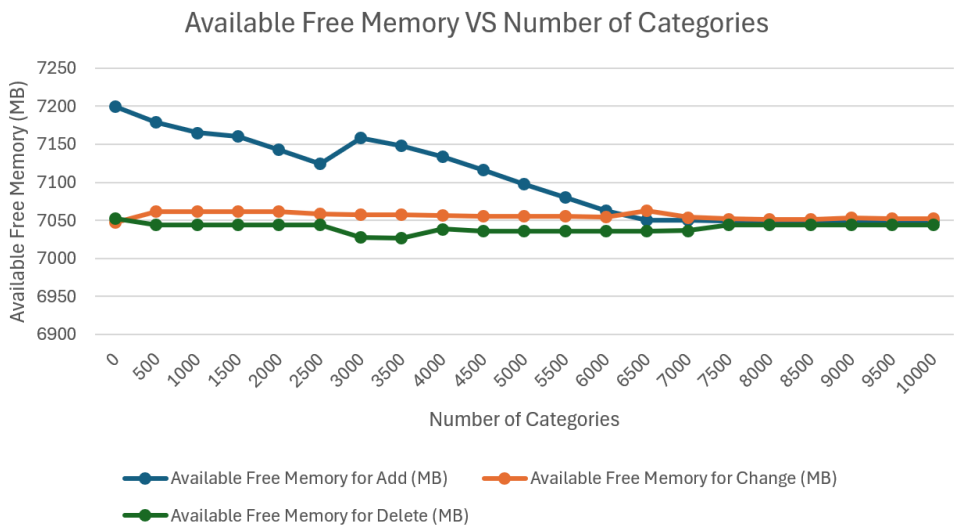


Categories Charts:


Sample Time VS Number of Categories
Sample Time (s)
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000 8500 9000 9500 10000
Number of Categories
Sample Time for Add (s)
Sample Time for Delete (s)
Sample Time for Delete (s)


Transaction Time VS Number of Categories
Transaction Time (s)
50
45
40
35
30
25
20
15
10
5
0
0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000 8500 9000 9500 10000
Plot Area
Number of Categories
Transaction Time for Add (s)
Transaction Time for Change (s)
Transaction Time for Delete (s)


CPU % Use VS Number of Categories
CPU % Use
25
20
15
10
5
0
0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000 8500 9000 9500 10000
Number of Categories
CPU % Use for Add
CPU % Use for Change
CPU % Use for Delete

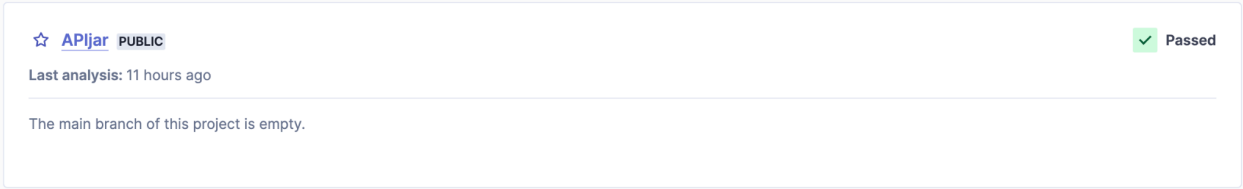


Recommendations for Codes Enhancement & Performance Risks:

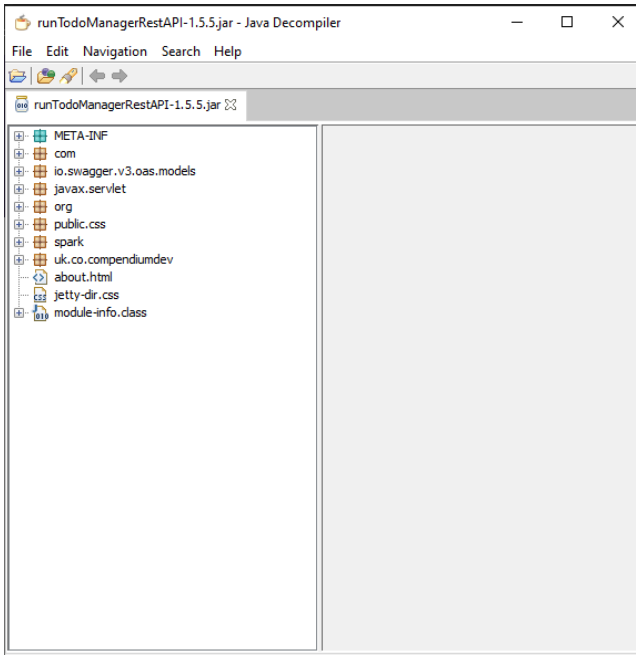
- We can observe that in all of the “Transaction Time VS Number of XXXXXX” graphs, the amount of transaction time increases in the order of Add, Change, Delete, which is logical since Add only requires putting the instance at the back of the query, but Change and Delete require searching the instance with specific id. To increase the performance, one may enhance the code by implementing searching algorithms such as binary search. It should be doable since we know that the id of newly added instances in the query is always bigger than all the previous id.
- For each graph related to CPU Use, There would always be some odd peaks that rise either in the middle of the curve or at the end of the curve. The unstable CPU percent use could be a potential performance risk that the developer should look into.

Describes implementation of static analysis with Sonar Cube community edition.

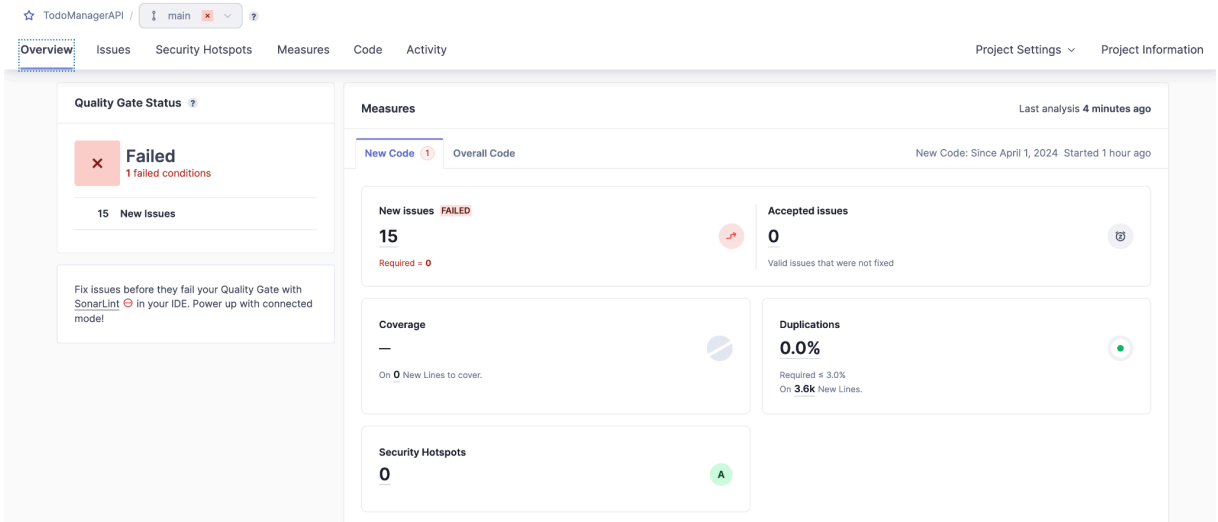
The implementation of static analysis with Sonar Cube was done on runTodoManagerRestAPI-1.5.5.jar file at first, but the result was unclear since it didn't provide a code when doing the analysis on the jar file only.



We then tried to decompile the jar file with JD-GUI and analyze those files with Sonar Cube.



The results of the analysis were still unclear.



We did more research about this software and found out that there exist another software called Sonar Cloud which does similar job with Sonar Cube.

[SonarCloud vs SonarQube]

“If your whole toolchain is already using online services (e.g. GitHub+Travis, or Bitbucket Pipelines, or Azure Pipelines online) then it likely means SonarCloud is a good fit (you’ll be leveraging native integrations we offer with these online tools, and wouldn’t have to maintain an on-prem installation when you’re used to consuming online services).

- If you build/test/package your application(s) on-prem, than fitting in an on-prem product like SonarQube likely makes more sense, as you’d likely want to avoid having a CI setup that spans across on-prem and cloud, with all of the technical considerations that this might imply (e.g. firewalls, NATs etc.).

A quick note too, to make it very clear from a static code analysis benefit point of view engine: SonarCloud runs the same Static Code Analysis engine as SonarQube Developer Edition.”

We were able to fork the original github code version of the API and make changes to be able to run the analysis: https://github.com/YuTongHu1/thingifier

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
thingifier-root							
challenger	4,407	0	0	392	0	0.0%	1.2%
challengerAuto	70	0	0	34	0	—	0.0%
ercoremodel	2,841	5	0	342	5	0.0%	0.0%
examplemodels	249	0	0	27	0	0.0%	0.0%
standAloneTodoListManagerRestApi	86	0	0	3	0	0.0%	0.0%
standAloneTodoListManagerRestApiAuto	47	0	0	5	0	—	0.0%
standAloneTodoListRestApi	84	0	0	0	0	0.0%	0.0%
swaggerizer	41	0	0	3	0	—	0.0%
thingifier	5,743	6	0	604	1	0.0%	0.6%
thingifierapp	115	0	0	34	0	0.0%	0.0%
todoManagerRestAuto	63	0	0	12	0	—	0.0%
pom.xml	33	0	0	0	0	—	0.0%

Include recommendations for improving code based on the result of the static analysis.

There are multiple recommendations, but we will only name the important ones. There are many instances where a literal is constantly repeated in the code. For example, the string “todos” is repeated 18 times in ChallengerApiResponseHook.java. It is best to define a

constant because duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences.
Additionally, some lines of code are commented out and should be removed because they distracts the focus from the actual executed code. It creates a noise that increases maintenance code. And because it is never executed, it quickly becomes out of date and invalid, they can be retrieved from source control history if required. The same thing goes for unused imports at the top of some files.
Another issue that could be easily be solved and improve the quality of the code is the loop over a `Map`, an object that maps keys to values. A map cannot contain duplicate keys, which means each key can map to at most one value. When both the key and the value are needed, it is more efficient to iterate the `entrySet()`, which will give access to both instead of iterating over the `keySet()` and then getting the value. If the `entrySet()` method is not iterated when both the key and value are needed, it can lead to unnecessary lookups. This is because each lookup requires two operations: one to retrieve the key and another to retrieve the value. By iterating the `entrySet()` method, the key-value pair can be retrieved in a single operation, which can improve performance.
More about loops, there are instances where we are able to reduce the number of break and continue statements in a for loop. This is an example of a noncompliant for loop with multiple continue statements.

```
for (int i = 1; i <= 10; i++) {
  if (i % 2 == 0) {
   continue;
  }
  if (i % 3 == 0) {
    continue;
  }
 // ...
}
```

This is the solution to a compliant for loop

```
for (int i = 1; i <= 10; i++) {
  if (i % 2 == 0 || i % 3 == 0) {

   continue;
  }
  // ...
}
```

The use of **`break`** and **`continue`** statements increases the complexity of the control flow and makes it harder to understand the program logic. In order to keep a good program structure, they should not be applied more than once per loop. This rule reports an issue when there is more than one **`break`** or **`continue`** statement in a loop. The code should be refactored to increase readability if there is more than one.

There are also multiple lamba expression in the code. The right-hand side of a lambda expression can be written in two ways:

1. Expression notation: the right-hand side is as an expression, such as in

   ```
   (a, b) → a + b
   ```

2. Block notation: the right-hand side is a conventional function body with a code block and an optional return statement, such as in

   ```
   (a, b) → {return a + b;}
   ```

By convention, expression notation is preferred over block notation. Block notation must be used when the function implementation requires more than one statement. However, when the code block consists of only one statement (which may or may not be a **`return`** statement), it can be rewritten using expression notation. This convention exists because expression notation has a cleaner, more concise, functional programming style and is regarded as more readable.

Onto some of the test files, this line of code `Assertions.assertTrue(response.body.equals(""));` leads to some unclearness. Testing equality or nullness with JUnit's `assertTrue()` or `assertFalse()` should be simplified to the corresponding dedicated assertion.

Noncompliant code example

```
Assert.assertTrue(a.equals(b));

Assert.assertTrue(a == b);

Assert.assertTrue(a == null);

Assert.assertTrue(a != null);

Assert.assertFalse(a.equals(b));
```

Compliant solution

```
Assert.assertEquals(a, b);

Assert.assertSame(a, b);

Assert.assertNull(a);

Assert.assertNotNull(a);

Assert.assertNotEquals(a, b);
```

To sum up, the static analysis highlights areas where our code can be improved. This includes defining constants for repeated words, removing unused code, optimizing loops, and simplifying assertions in tests. By making these changes, our code will become easier to understand, maintain, and perform better overall.