# Static Analysis using LLMs

## Project Report

**Tulasi Rama Raju Chittiraju – M16407796**

### Problem Statement

The classic static analysis tools like Spotbugs uses an already defined set of rules and pattern identification methods to detect vulnerabilities and code failures. This sometimes leads to limited coverage of the code base when faced with higher level logic flaws. Some of the LLMs and their APIs have shown to be capable in reasoning over code and natural language, which I find promising for better static analysis and code coverage. This project aims to see whether LLMs can enhance or complement existing static analysers by having better performance.

This project aims to see whether LLMs can enhance or complement existing static analysers by having better performance. LLM based analysis may generalize better, but its yet to be known if its precision and reliability is up to the mark and to be empirically validated.

### System Implementation

#### i. Tech Stack and Libraries used

> Python

> IntelliJ (Spotbugs)

> Ollama

> Llama 3.2

> Pandas (py)

#### ii. Data

The same codebase used for homework 2 was used for experimenting with LLMs to identify bugs. The hospital management system, including all the java code files are available at the given GitHub repository.

#### iii. Process

LLM-based static analysis pipeline - A Python script was implemented to recursively traverse the project source directory and read all java files. This merged code string was used to construct a detailed prompt instructing Llama 3.2 to behave as a static analyzer and return a JSON array of issues. The script then sent this prompt to the local Ollama API using the requests library, handled potential server errors, extracted the JSON array from the model's response, and saved the results into 'llm_issues.json'.

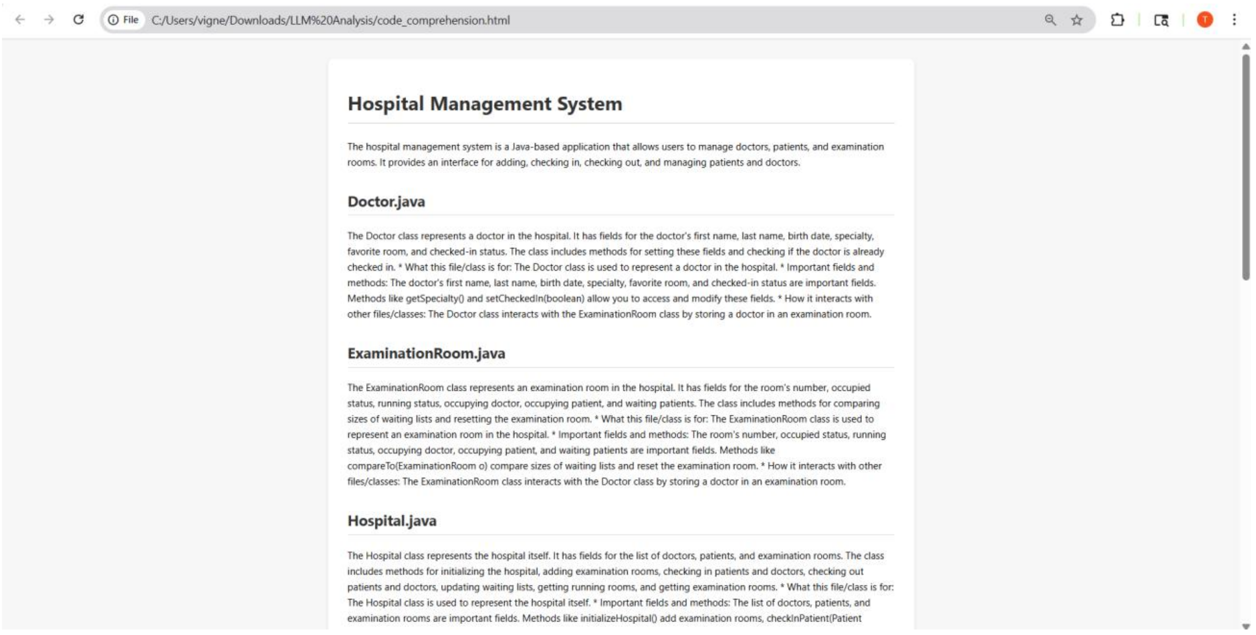
Reporting and comprehension layer - A second prompt used the combination of original project code and the JSON issues list to ask Llama 3.2 for a good HTML report with sections like root cause, impact, and suggested fixes for each issue. This report was saved as ‘llm_issues_report.html’. Another script, dedicated to code comprehension, generated a high-level explanation of each Java class and wrapped it into ‘code_comprehension.html’ for easy viewing in a browser.

SpotBugs output and comparing results - The SpotBugs XML was parsed, extracting file names, line numbers, bug types, categories, priorities, and ranks into ‘spotbugs_issues.csv’. The evaluation script then used pandas to load ‘llm_issues.json’ and ‘spotbugs_issues.csv’, normalize categories and severities (e.g., mapping multiple SpotBugs categories into CORRECTNESS or BAD_PRACTICE), and compute overlaps and differences between the two tools at the (file, category) level. The script also printed example overlapping issues and unique findings on each side, supporting a qualitative analysis of how LLM-based analysis complements traditional static analysis.

GitHub Repository Link - https://github.com/Tulasi-Raju/Static-Analysis-using-Llama-3.2

**Results**

**i. Code Comprehension HTML**


File
C:/Users/vigne/Downloads/LLM%20Analysis/code_comprehension.html

Hospital Management System

The hospital management system is a Java-based application that allows users to manage doctors, patients, and examination rooms. It provides an interface for adding, checking in, checking out, and managing patients and doctors.

Doctor.java

The Doctor class represents a doctor in the hospital. It has fields for the doctor's first name, last name, birth date, specialty, favorite room, and checked-in status. The class includes methods for setting these fields and checking if the doctor is already checked in. * What this file/class is for: The Doctor class is used to represent a doctor in the hospital. * Important fields and methods: The doctor's first name, last name, birth date, specialty, favorite room, and checked-in status are important fields. Methods like getSpecialty() and setCheckedIn(boolean) allow you to access and modify these fields. * How it interacts with other files/classes: The Doctor class interacts with the ExaminationRoom class by storing a doctor in an examination room.

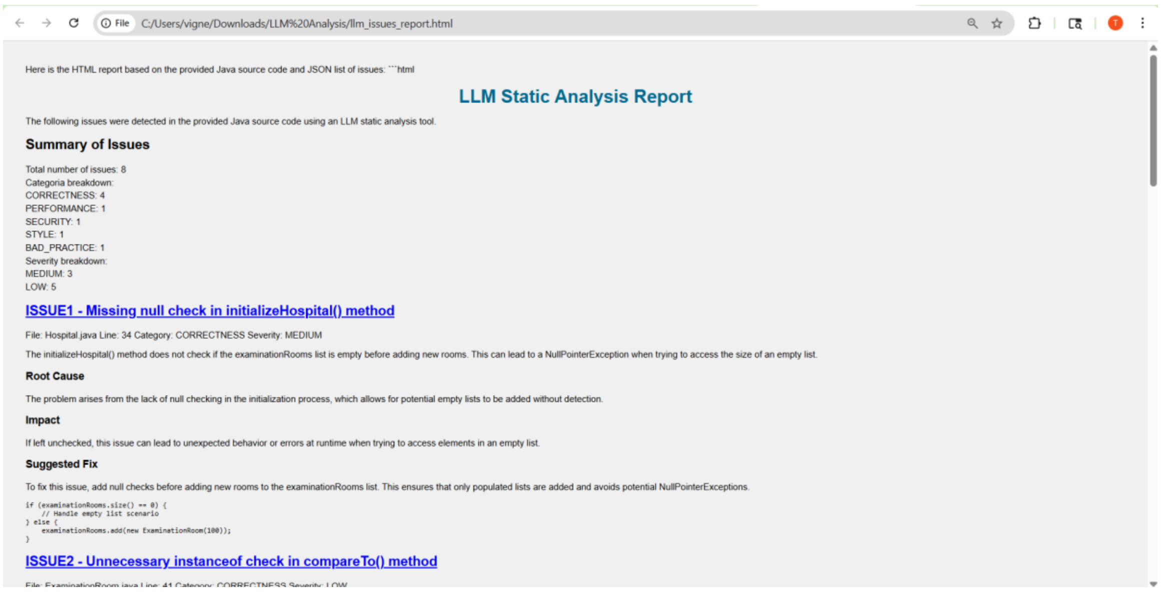
ExaminationRoom.java

The ExaminationRoom class represents an examination room in the hospital. It has fields for the room's number, occupied status, running status, occupying doctor, occupying patient, and waiting patients. The class includes methods for comparing sizes of waiting lists and resetting the examination room. * What this file/class is for: The ExaminationRoom class is used to represent an examination room in the hospital. * Important fields and methods: The room's number, occupied status, running status, occupying doctor, occupying patient, and waiting patients are important fields. Methods like compareTo(ExaminationRoom o) compare sizes of waiting lists and reset the examination room. * How it interacts with other files/classes: The ExaminationRoom class interacts with the Doctor class by storing a doctor in an examination room.

Hospital.java

The Hospital class represents the hospital itself. It has fields for the list of doctors, patients, and examination rooms. The class includes methods for initializing the hospital, adding examination rooms, checking in patients and doctors, checking out patients and doctors, updating waiting lists, getting running rooms, and getting examination rooms. * What this file/class is for: The Hospital class is used to represent the hospital itself. * Important fields and methods: The list of doctors, patients, and examination rooms are important fields. Methods like initializeHospital() add examination rooms, checkInPatient(Patient

### ii. LLM Issues HTML



C:/Users/vigne/Downloads/LLM%20Analysis/llm_issues_report.html

Here is the HTML report based on the provided Java source code and JSON list of issues: ```html

LLM Static Analysis Report

The following issues were detected in the provided Java source code using an LLM static analysis tool.

**Summary of Issues**

Total number of issues: 8
Categoria breakdown:
CORRECTNESS: 4
PERFORMANCE: 1
SECURITY: 1
STYLE: 1
BAD_PRACTICE: 1
Severity breakdown:
MEDIUM: 3
LOW: 5

ISSUE1 - Missing null check in initializeHospital() method

File: Hospital.java Line: 34 Category: CORRECTNESS Severity: MEDIUM

The initializeHospital() method does not check if the examinationRooms list is empty before adding new rooms. This can lead to a NullPointerException when trying to access the size of an empty list.

**Root Cause**

The problem arises from the lack of null checking in the initialization process, which allows for potential empty lists to be added without detection.

**Impact**

If left unchecked, this issue can lead to unexpected behavior or errors at runtime when trying to access elements in an empty list.

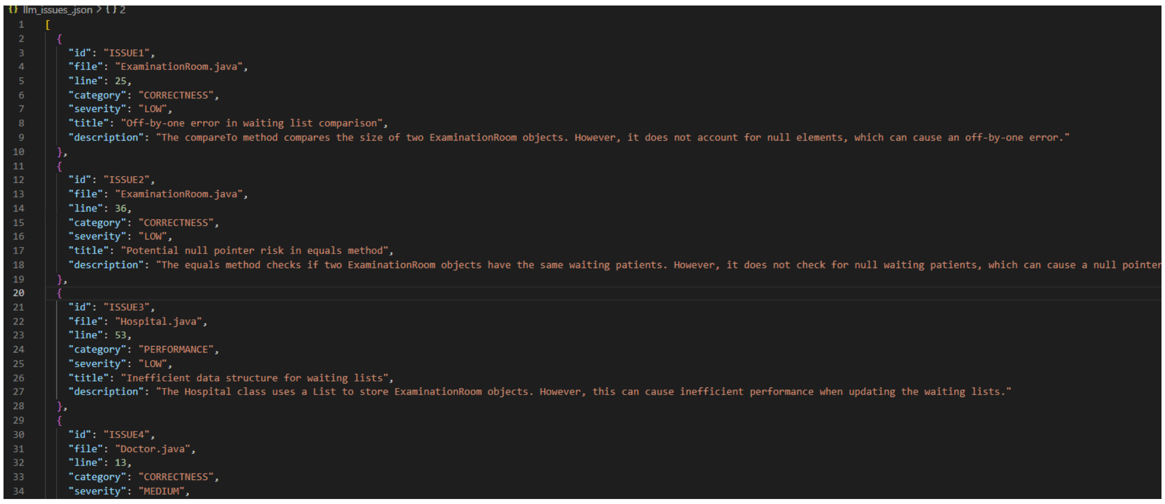
**Suggested Fix**

To fix this issue, add null checks before adding new rooms to the examinationRooms list. This ensures that only populated lists are added and avoids potential NullPointerExceptions.

```
if (examinationRooms.size() == 0) {
    // Handle empty list scenario
} else {
    examinationRooms.add(new ExaminationRoom(100));
}
```

ISSUE2 - Unnecessary instanceof check in compareTo() method

### iii. LLM Issues JSON



llm_issues_.json > { } 2

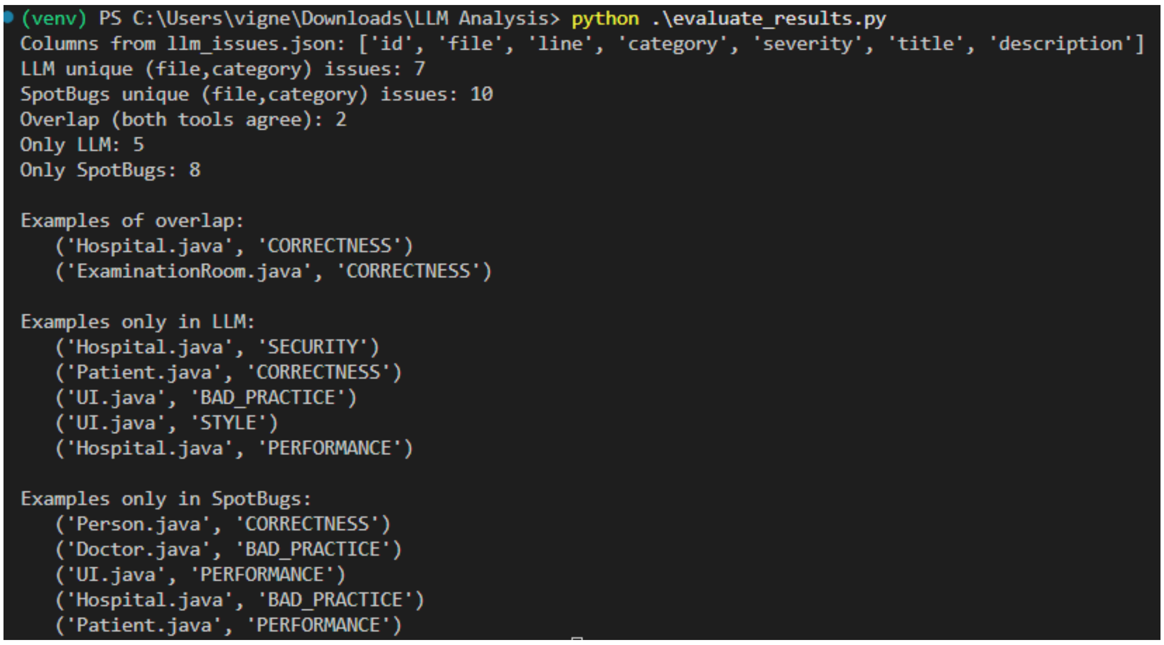
```
[
  {
    "id": "ISSUE1",
    "file": "ExaminationRoom.java",
    "line": 25,
    "category": "CORRECTNESS",
    "severity": "LOW",
    "title": "Off-by-one error in waiting list comparison",
    "description": "The compareTo method compares the size of two ExaminationRoom objects. However, it does not account for null elements, which can cause an off-by-one error."
  },
  {
    "id": "ISSUE2",
    "file": "ExaminationRoom.java",
    "line": 36,
    "category": "CORRECTNESS",
    "severity": "LOW",
    "title": "Potential null pointer risk in equals method",
    "description": "The equals method checks if two ExaminationRoom objects have the same waiting patients. However, it does not check for null waiting patients, which can cause a null pointer
  },
  {
    "id": "ISSUE3",
    "file": "Hospital.java",
    "line": 53,
    "category": "PERFORMANCE",
    "severity": "LOW",
    "title": "Inefficient data structure for waiting lists",
    "description": "The Hospital class uses a List to store ExaminationRoom objects. However, this can cause inefficient performance when updating the waiting lists."
  },
  {
    "id": "ISSUE4",
    "file": "Doctor.java",
    "line": 13,
    "category": "CORRECTNESS",
    "severity": "MEDIUM",
```

## Experimental Comparison

### Evaluate Results between Spotbugs and Llama 3.2

To evaluate the effectiveness of LLM-based static analysis, the issues identified by Llama 3.2 were compared with those reported by SpotBugs across the project. The results show that Llama 3.2 identified 7 unique (file, category) issue types, while SpotBugs identified 10, with an overlap of only 2 categories affecting the same files. This demonstrates that both tools detect different classes of problems, and therefore complement each other. Notably, Llama 3.2 flagged issues that SpotBugs did not, including security concerns in Hospital.java, style problems in UI.java, and performance or correctness risks not captured by traditional rule-based analysis.

Overall, the experiment shows that the LLM-based approach enhances static analysis coverage by catching conceptual, stylistic, and architectural issues that SpotBugs misses, while SpotBugs provides reliable pattern-based detection. Together, they provide a more complete picture of software quality than either tool alone.



```
(venv) PS C:\Users\vigne\Downloads\LLM Analysis> python .\evaluate_results.py
Columns from llm_issues.json: ['id', 'file', 'line', 'category', 'severity', 'title', 'description']
LLM unique (file,category) issues: 7
SpotBugs unique (file,category) issues: 10
Overlap (both tools agree): 2
Only LLM: 5
Only SpotBugs: 8

Examples of overlap:
   ('Hospital.java', 'CORRECTNESS')
   ('ExaminationRoom.java', 'CORRECTNESS')

Examples only in LLM:
   ('Hospital.java', 'SECURITY')
   ('Patient.java', 'CORRECTNESS')
   ('UI.java', 'BAD_PRACTICE')
   ('UI.java', 'STYLE')
   ('Hospital.java', 'PERFORMANCE')

Examples only in SpotBugs:
   ('Person.java', 'CORRECTNESS')
   ('Doctor.java', 'BAD_PRACTICE')
   ('UI.java', 'PERFORMANCE')
   ('Hospital.java', 'BAD_PRACTICE')
   ('Patient.java', 'PERFORMANCE')
```

**Note:** These results may vary as each generation of bug detection by running the same prompts multiple times with the LLMs can lead to multiple different sample size of solutions. So, to determine the exact metrics for comparison between Spotbugs and Llama 3.2 is inconclusive.

## Observations About LLM Behaviour

Llama 3.2 performed better when run on a GPU-enabled gaming laptop connected to power, due to higher clock speeds and available VRAM.

On battery, downclocking caused slower inference and occasionally less consistent answers.

LLM outputs remained mostly deterministic when prompts were strict and structured.

## Future Improvements

Better model with better computing power yields better results. We can quantify how these approaches complement each other by measuring overlaps, differences, precision, and recall of detected issues. Potential opportunity to create hybrid pipelines where LLMs post-process or refine traditional static analysis reports, generating more actionable developer guidance. Can optimize prompts to generate complete documentation of the bugs.