B.Comp. Dissertation

Advancing automated code review with JSReview: A hybrid framework combining learning and pattern-based approaches for JavaScript

By Bikramjit Dasgupta

Department of Information Systems and Analytics
School of Computing
National University of Singapore
2022/2023

B.Comp. Dissertation

Advancing automated code review with JSReview: A hybrid framework combining learning and pattern-based approaches for JavaScript

By Bikramjit Dasgupta

Department of Information Systems and Analytics
School of Computing
National University of Singapore
2022/2023

Project ID: H202340
Advisor: Dr. Lek Hsiang Hui
Deliverables:
Report: 1 Volume

Abstract

Peer code reviews are a crucial stage in any software engineering project, where incoming source code is manually analyzed for defects and adherence to best practices to ensure the highest quality of the entire project. However, this manual process is often time and resource-intensive, as well as error-prone and inconsistent. To address these challenges, this project proposes and implements an automated code review framework that reduces human effort while enhancing the quality of the review process – beyond linters. Employing a novel approach that combines machine learning and pattern-based analysis, this hybrid framework is designed to be practically comprehensive, targeting areas of bug detection, code smell identification, vulnerability hotspot assessment, and maintainability risks – which are the focal points of manual code review processes. By integrating the framework with a new GitHub CI/CD platform, this project demonstrates its applicability and potential impact in real-world scenarios. Overall, the quantitative and qualitative evaluations of the framework reveal promising outcomes, paving the way for further refinement and extension of both the core concepts and technical implementation.

Subject descriptors:

D.2.5 Testing and debugging

D.2.7 Distribution, maintenance, and enhancement

I.2.6 Learning

Keywords:

Automatic Code Review, Machine Learning, Software Engineering, Software Quality, Vulnerability Detection, Defect Detection, CI/CD

Implementation software and hardware:

JavaScript ES6, Python 3.9, Acorn, PyTorch, Scikit Learn, Fastify, MongoDB

Acknowledgement

I would like to express my deepest gratitude to my advisor, Dr. Lek Hsiang Hui, for giving me unwavering and crucial direction with this project at every stage. His expertise and direction played a pivotal role in shaping the outcomes of this project, enabling me to work on something that strongly resonated with my passions and interests.

I am also profoundly grateful to the faculty and fellow students at the School of Computing for this 4-year journey of learning that finally culminated in this project.

Contents

1. Introduction

1.1. The problem

Developing high-quality software necessitates meticulous attention to detail throughout all stages of the software development lifecycle. A critical component of this lifecycle is the peer code review process, where other engineers in a team review incoming revisions by manually analyzing the code [1]. This is made easier with dynamic automated testing, which speeds up the process to detect runtime defects through pre-defined runtime tests to verify outcomes. However, automated testing methods still possess inherent limitations, which necessitates the continued use of static peer code analysis to verify code quality, maintainability concerns, vulnerabilities, and adherence to best practices or organizational standards. These aspects often require the expertise and insights of seasoned software engineers who can assess the code's overall design, architecture, and quality.

While manual static code reviews are undeniably effective, they entail a unique set of challenges. The process can be time and resource intensive, especially large-scale projects where revisions are frequent. As with any human-driven procedure, inconsistencies and errors may occur, leading to disparities in code quality assessments. Moreover, manual reviews necessitate the involvement of experienced engineers, who possess the expertise to evaluate and verify code quality but may be out of reach for smaller or inexperienced teams. Linters are frequently used during development, but they primarily analyze surface level information regarding the stylistic aspects of source code. Given these limitations, there is a pressing need for an automated static code review (ACR) solution capable of consistently and effectively streamlining the process. Such a solution can accelerate the software development lifecycle by minimizing human effort while maintaining high standards of code quality.

1.2. Motivation

The motivation behind this project is rooted in the modern challenges and limitations of code reviews, considering the ever-growing complexity and scale of modern software engineering projects. As the software development landscape evolves, there is an increasing demand for an efficient, reliable, and automated solution to cope with this growing complexity [2]. Automation in this area therefore remains a crucial area in need of improvement, as code reviews are still “done by manual work in the industry” (He et al. [3]).

Moreover, the advantages of an ACR solution extend to a wide range of beneficiaries, encompassing not only large software engineering teams but also students and individual developers. For sizable teams working on progressively complex projects, implementing an ACR system can significantly decrease the effort and manpower required for code reviews, thereby streamlining the overall development process. In the context of educational environments, students stand to gain valuable insights through immediate feedback on their code, thereby expediting the learning process and enhancing their programming aptitude. For individual developers, an ACR framework provides an invaluable supplementary perspective, serving to verify code quality and ultimately elevate the overall caliber of the project.

1.3. Objectives

This project therefore aims to design and implement a an ACR system that comprehensively automates the review process effectively. Through a hybrid framework that incorporates learning and pattern-based analysis, this project targets all major aspects of the static code review method holistically. These include the four core aspects of ACR: detecting potentially logical bugs, code smells that indicate deeper underlying issues, maintainability risks and vulnerable hotspots of complexity within code.

While the fundamental design of the proposed framework is intended to be adaptable across multiple programming languages, for the purposes of this project, the framework will target JavaScript. This is due to its susceptibility to errors and its vital role as the backbone of modern web applications.

In addition, this project also aims to create a Continuous Integration and Continuous Deployment (CI/CD) platform that leverages the proposed framework, to help software engineering teams. Integrated with GitHub, this platform would be used to scan repositories and incoming pull requests through the framework. This platform will therefore demonstrate the practical integration of the core ACR framework within real-world software development workflows.

2. Related work

2.1. Literature review

2.1.1. Static code analysis and ACR

Although linters such as ESLint have gained widespread popularity amongst most JavaScript developers for static evaluation [4], the high requirement of human effort persists due to their surface level analysis into syntax and style. For instance, every commit in Mozilla projects require reviews from two independent reviewers to ensure consistency and efficacy [5].

To go beyond linting, two primary research directions in ACR have emerged in recent years to reduce this tediousness of manual reviews. The first concentrates on creating infrastructural tools such as reviewer recommendation systems or specialized interfaces to aid review workflows like Gerrit [6]. Thongtanunam et al. [7] for instance have proposed a reviewer recommendation system called RevFinder, arguing that the difficulty of finding suitable reviewers adds 12 days to the code review process on average. This sentiment is echoed by Zanjani et al. [8], presenting a similar approach named cHRev to automate selecting reviewers, demonstrating a high degree of success in streamlining processes and increasing quality of reviews.

The second direction for ACR is tackling the direct automation of the analysis and approval of source code and revisions to existing codebases, reducing the role of peer reviewers. This includes DeepReview [9] and SimAST-GCN [10] – which are predictive models to directly approve or reject revisions in source code – bypassing the requirement of human reviewers. However, due to challenges with efficacy, other reviewers like Tufano et al. [11] have proposed assistive learning-based models that detect and patch bugs. Similarly, machine learning is also explored for vulnerability detection, with Ferenc et al. [12] developing a dataset for binary classification of vulnerabilities in functions. Li et al. [13] and Chakraborty et al. [14] have also proposed deep learning-based approaches to detect vulnerabilities with success – with an

accuracy of 95 percent for the latter. Code smells on the other hand have been investigated with more of a metric and pattern-based approach in ACR, with Fard and Mesbah [15] proposing an effective technique to detect smells specific to JavaScript.

2.1.2. Automatic program repair

Taking ACR further, Automatic Program Repair (APR) is an emerging field of research focusing on patch generation for defects. Rather than aid the evaluation process at the end of development, APR takes a more active role in helping developers fix their code within development. Addressing recent advances, Goues et al. [16] highlight that APR is "fundamentally a search problem", with the search goal being a set of new changes to a program for an error, without introducing new defects. Other approaches in this field include employing supervised learning or transfer learning using language models [11]. Although beyond the scope of this project, the learnings about bug detection, localization and search techniques are valuable insights from APR that are applicable in ACR.

2.1.3. JavaScript

Often termed as the "lingua franca" of the web, JavaScript is heavily employed in both web applications and backend applications, despite having flaws with regards to being severely unstructured, unpredictable, and error-prone [17]. As one of the most dynamic languages with typing and runtime behavior, it becomes a highly challenging language to statically analyze for vulnerabilities and defects [18]. This is only worsened by the "jungle of JavaScript frameworks" [19], as the primary package registry NPM distributes upwards of 1.3 million packages [20] – each with slightly different standards, expectations, and conventions due to the language's unopinionated nature. Consequently, this therefore underscores the requirement for a JavaScript-specific ACR solution to address its inherent limitations and risks.

2.2. Current commercial solutions for static analysis and ACR

Despite the extensively growing research fields of ACR and APR, not many commercial solutions exist. This section first details the use of linters used by most developers for static analysis, and later into two major attempts at bringing ACR to developers.

2.2.1. Linters

JavaScript linters such as ESLint and JSLint have been instrumental in improving the code quality and maintainability by identifying syntax errors, stylistic issues, and potential pitfalls during the development process [4]. Through an integration with most IDEs, these linters highlight issues and display messages while developers write code. However, their limitations lie in their surface level focal point, leaving deeper logic-related problems or vulnerabilities undetected. Although their rule-based approach is effective in highlighting virtually all syntax-related issues, they neglect complex patterns deeper in the structure of the code.

2.2.2. DeepScan

DeepScan [21] is a static analysis tool that aims to go beyond traditional JavaScript linters. It uses a data flow analysis technique to detect more intricate issues, such as runtime errors and logical flaws, that might go unnoticed by rule-based linters. By matching with large dictionaries of pattern-based issues, this approach enhances its ability to identify problematic code patterns and potential vulnerabilities. Although DeepScan provides an improvement over linters, it still has limitations. These include neglecting deeper bugs and vulnerable areas that cannot easily be detected through a pattern-based approach due to the complexity, intricacy, and uniqueness of each case.

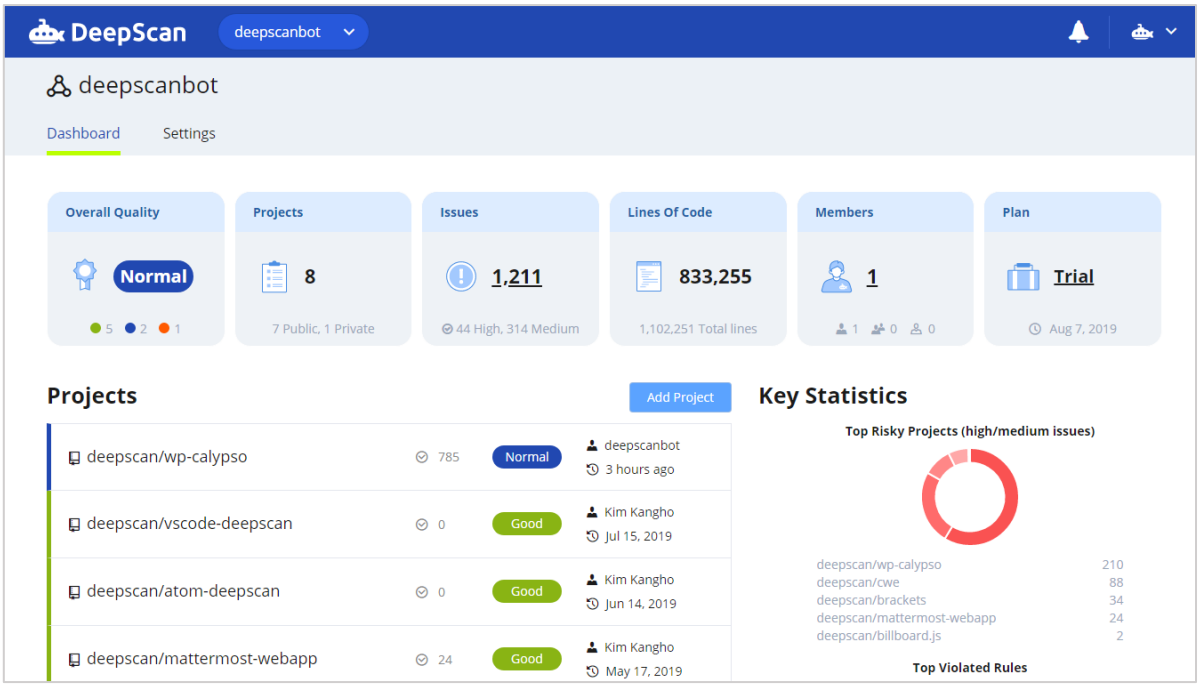

Figure 1: Using a dashboard and report for smells and vulnerabilities

To use DeepScan, users link their project repositories via GitHub, and DeepScan scans their repository through their engine, presenting the severities and locations of pattern-based issues.

2.2.3. SonarQube

Taking DeepScan further, SonarQube [22] is a popular static analysis solution for most major languages, which scans source code for defects, vulnerabilities and smells through a pattern-based approach. SonarQube's strengths lie in its integration with CI/CD pipelines for real-time feedback, and the extensively large ruleset. Additionally, it allows to customize existing rules – for a very practical solution as each organization has different standards. Despite these strengths, SonarQube shares the same limitations as DeepScan, such as relying on predefined rules and patterns for detecting issues, and struggling to identify complex patterns in the structure of the code.

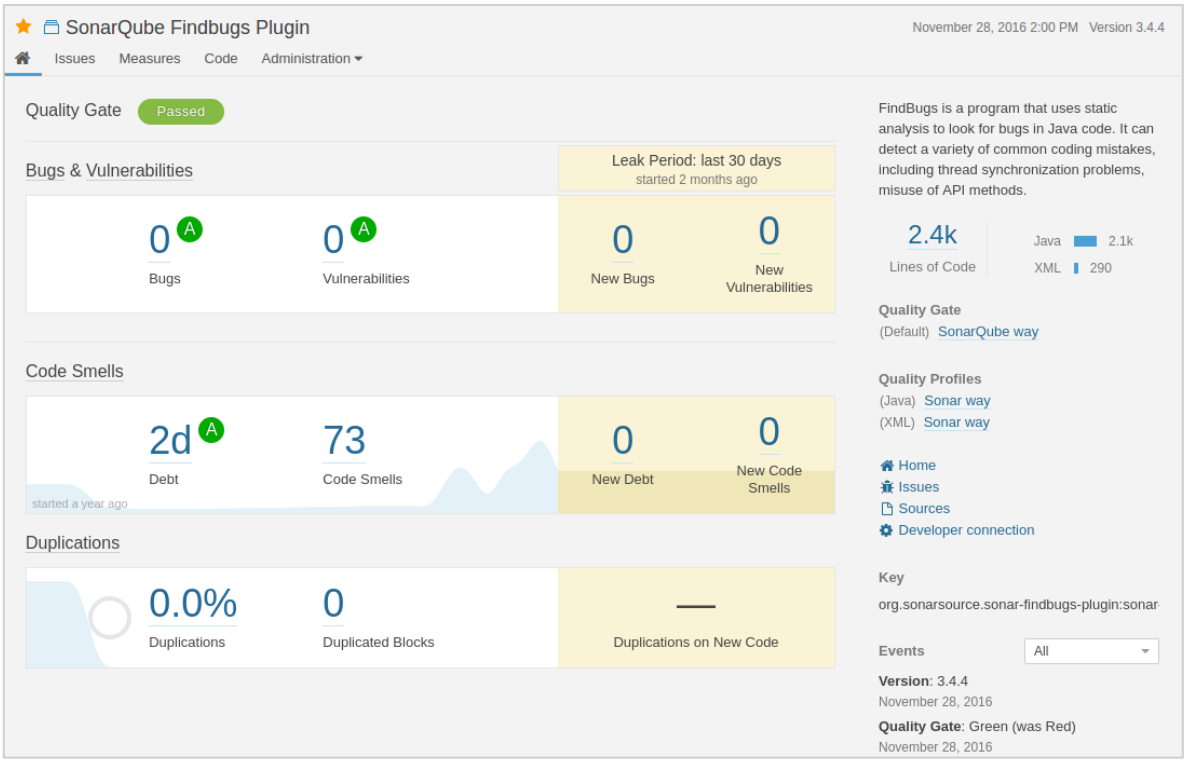

Figure 2: SonarQube's dashboard for bugs, vulnerabilities, and smells

Like DeepScan, users link their repositories to the platform for scans and tracking common quality metrics such as number of defects, days of technical debt, duplication ratio and introduction of new issues with revisions. All new pull requests are then analyzed automatically as they come in, with a detailed report using these metrics for incoming code. Users may configure existing rules through a comprehensive settings page, depending on what their organization requires.

3. Requirements

3.1. Functional requirements

To meet all objectives, the essential functional requirements of the ACR framework are geared towards providing a comprehensive static analysis to detect the following aspects of source code. This includes four aspects for holistic ACR: code smells, potential bugs, maintainability risks and vulnerable hotspots, as follows. This project will aim to reduce human effort instead of attempting to completely remove it – due to efficacy concerns.

3.1.1. Code smells

These are rule-based indications of potential underlying issues that may undermine code standards, readability, understandability, and extensibility. The framework should detect and pinpoint developers and reviewers to help them better investigate the implications of detected smells. The framework should detect these smells below (Figure 3), which are some prominent smells in the JavaScript language [23]. Apart from unused variables, this list of smells is specifically targeted to go beyond common linters that most developers use within IDEs.

Code smell	Description
Large functions	Functions that are too long and contain excessive logic, making them hard to grasp.
Deep nesting	Excessive level of nested blocks, reducing maintainability and readability.
God functions	Functions that perform too many tasks, violating the single responsibility principle.
God objects	Objects containing excessive amount of functionality, undermining readability.
Unused variables	Variables declared but never used in the future.
Magic literal numbers	Unexplained constants in code that are difficult to read and maintain.

Callback hell	Excessive nested function callbacks that make code much harder to change.
Type coercion	Implicit but dangerous conversion of types.
Excessive chaining	Long chains of function calls, which make code hard to read and change down the line.
Dead code	Bloat code that is never executed.
Excessive global variables	Overuse of global variables, that lead to collisions or unexpected behavior.
Excessive comments	Redundance or over-reliance on comments.
Misuse of null/undefined	Interchangeable use of null and undefined, which can lead to bugs and confusion in code.
Inheritance overuse	Deep hierarchies, that make code harder to understand and maintain.

Figure 3: List of code smells

3.1.2. Potential bugs

The ACR framework must be proficient in uncovering potential logical bugs through static analysis, to help developers in their debugging process. It should especially be able to detect more implicit bugs with complex underlying patterns and relationships.

3.1.3. Maintainability risks

The ACR framework must detect areas of code that undermine the long-term maintainability of projects – especially with large codebases [24].

Maintainability risk	Description
Duplicated code	Copy-pasted code, that undermine scalability and can lead to unpredictable issues.
Lack of documentation	Missing documentation for classes and functions that make it harder to understand code.
Complex conditionals	Exceptions that are not properly caught or handled.

Improper variable scoping	Variables scoped at unnecessarily higher scopes or levels than required, making changes risky

Figure 4: List of maintainability risks

3.1.4 Vulnerable hotspots

Complex areas of code, that are particularly susceptible to vulnerabilities, need to be detected to help developers perform additional checks to ensure those parts of code are safe [25, 12].

3.2. Non-functional requirements

Apart from functional requirements, there are a few key non-functional requirements that are vital to the usability and practicality of this framework:

- **Extensibility**: The framework should provide a foundation that can be built upon in the future, for other languages or functionalities.
- **Resilience for JSReview**: The framework should be tolerant to a wide diversity of JavaScript projects. This is key, as many teams have their own standards and best practices for JavaScript due to how unopinionated the language is. There is also a plethora of frameworks, particularly with regards to third party node.js packages – which follow their own guidelines and structures.
- **Consistency**: Standards enforced must be practicable, consistent, and as objective as possible.
- **Modularity**: Different components of the larger framework should be modular in nature – with minimal dependence on other components. Loose coupling would ensure extensibility of the framework in the future.

4. Design and methodology

This section details how the proposed framework is designed and implemented to meet all requirements in section 3.

4.1. Overall framework design

To derive a comprehensive and modular design, we may first observe two distinct categories among the analysis objectives outlined in the functional requirements. The detection of code smells and maintainability risks are governed through a predefined set of rules, while bug and vulnerable hotspot detection are more ambiguous, with complex patterns in the code.

1. For **smells** and **maintainability risks**, pattern recognition will be done through Abstract Syntax Trees (ASTs) to analyse the underlying structure of code effectively.
2. Strictly pattern-based analysis on the other hand may not be suitable for the **bugs** and **vulnerable hotspots,** due to the necessity for a more in-depth examination of each case. Consequently, employing machine learning techniques to uncover these underlying complex patterns, which are not explicitly discernible, could substantially enhance the efficacy of detection in both categories. However, AST-based structural analysis would still be required to extract important sections like functions and code blocks from the input and will also help directly detect some specific bugs that have specific patterns.

To account for this duality, the framework's architecture will first pre-process all incoming code to AST representations to retain their structural, relational, and semantic aspects for all downstream tasks – since both categories above need AST-based extraction of key details like blocks, values, and software metrics. Downstream, this AST representation will go through the following pattern-based analytics, before being passed to the learning-based analytics for vulnerabilities and bugs:

1. **Closure/block analysis**: The AST will be recursively searched to according to closures and code-blocks, to analyse them against the code smell and maintainability risk patterns. Through this, function nodes will also be extracted to be passed further downstream to both the software metric extraction stage, and the bug detection and repair model for analysis.
2. **Value lifecycle analysis**: Variables, function arguments and class attributes will then be tracked via recursive walks through the AST, to also match them against code smell and maintainability risk patterns.
3. **Software metric extraction**: Extracted functions from the closure/block analysis will be further analysed to get key software metrics – to then be passed downstream to the vulnerable hotspot classifier as feature vectors. More details about this in section 4.4.

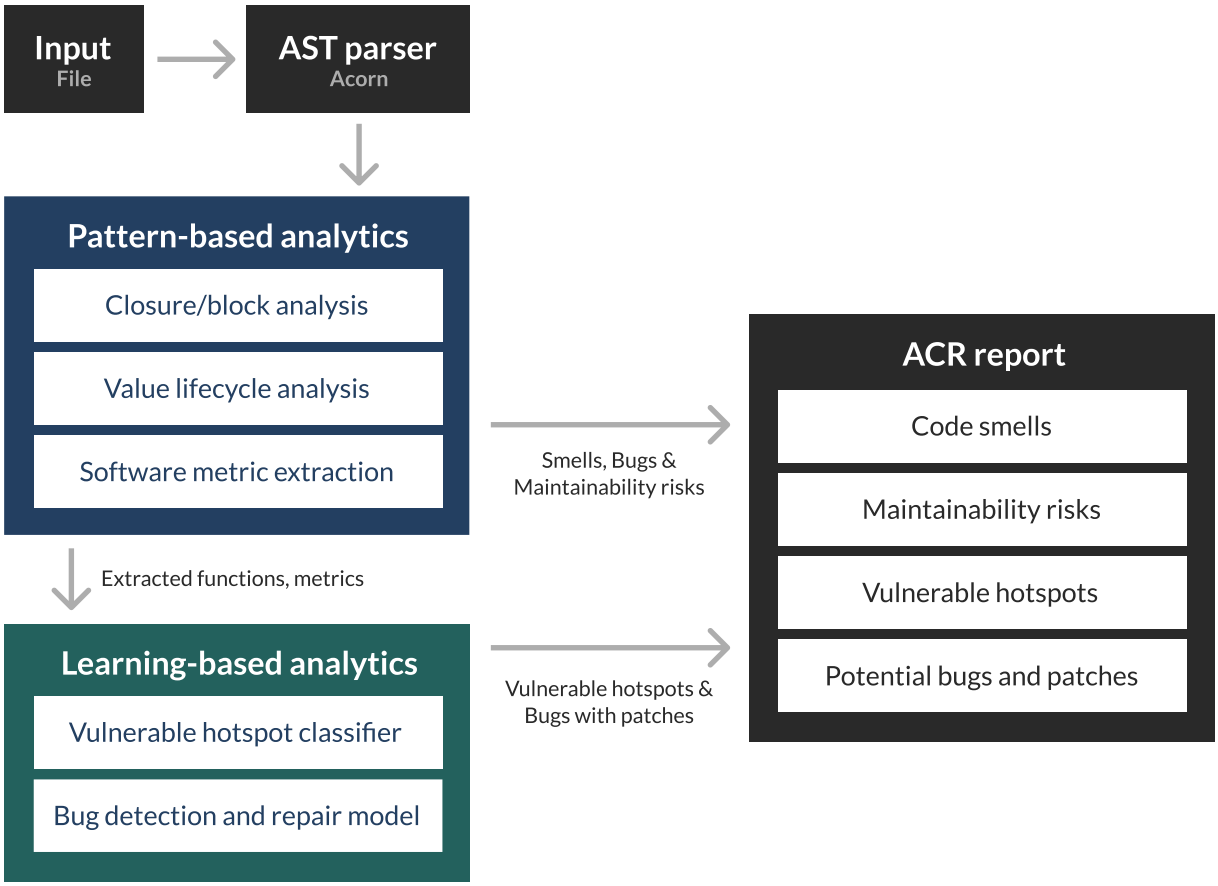

Figure 5: Overall framework design for JSReview

4.2. Detecting code smells

The list of code smells in section 3.1 have their own individual patterns to match AST nodes against, during both the closure/block and value lifecycle analysis stages.

Code smell	Patterns in ASTs
Large functions	Function nodes with a high number of statements or lines of code.
Deep nesting	Block nodes with a high level of recursive nesting.
God functions	Function nodes that perform multiple tasks or have excessive arguments and variables.
God objects	Class nodes that have a high number of properties or methods.
Unused variables	Value nodes that are not called or used.
Magic literal numbers	Nodes containing unexplained numerical constants.
Callback hell	Nodes with multiple deeply nested asynchronous function callbacks.
Type coercion	Node lifecycles with implicit type conversions from originally inferred types.
Excessive chaining	Nodes with long chains of method calls or property access, reducing readability and maintainability.
Dead code	Nodes that are never accessed.
Excessive global variables	Value nodes that are declared high in scope and used excessively in different parts of the tree.
Excessive comments	Nodes with a high density of comments, suggesting code may be difficult to understand or maintain.
Misuse of null/undefined	Conditional and value nodes that treat null and undefined values interchangeably.
Inheritance overuse	Class nodes with deep inheritance hierarchies.

Figure 6: Code smells and patterns (closure analysis in white, value lifecycle in gray)

In both closure/block and value lifecycle analyses, each node is analyzed and tracked according to the patterns through a recursive walk across the nodes with a depth-first traversal. Flagged nodes will be stored with location and type of violation.

4.3. Detecting maintainability risks

Like in code smells, pattern matching on AST nodes will also be used to flag maintainability risks. This pattern matching is done exclusively in the closure/block analysis stage.

Maintainability risk	Patterns in ASTs
Duplicated code	Nodes with identical or highly similar structures with each other. Checked with a Tree Edit Distance (TED) algorithm to match structures [26].
Lack of documentation	Function and class nodes without documentation.
Complex conditionals	Deeply nested or convoluted structures of conditional nodes.
Improper variable scoping	Variable nodes scoped higher than required.

Figure 7: Maintainability risks and patterns

4.4. Detecting vulnerable hotspots

Given the success of machine learning in the field of vulnerability detection, this project will utilize machine learning methods to effectively flag hotspots in code that have complex logic and might be vulnerable.

4.4.1. Dataset and metric-based classification

To develop an effective model to detect hotspots, a suitable dataset and classification approach are required. Few options were considered:

- Using the Snyk Vulnerability Database [27] to source for labels and code, for a multi-classification task. However, the database evaluated JavaScript packages as a whole, instead of individual chunks of code, undermining localization of defective code in the resulting dataset.

- Exploring the GitHub archive to obtain code chunks from pull requests containing identified vulnerabilities and their respective patches was considered. However, this proved less feasible, as commit messages explicitly mentioning vulnerabilities were found to be relatively scarce. This made it challenging to gather a substantial dataset for a text classification task.
- Lastly, using a third-party dataset provided by Ferenc et al. [12] proved to be feasible, as it focused on JavaScript hotspots by sourcing real-world vulnerabilities from the Snyk platform, Node Security Project, and patches from GitHub. The extracted data and features then were combined into a unified dataset for binary classification for vulnerable hotspots in code, given software metrics. These include cyclomatic complexity, Halstead metrics, cyclomatic density, and parameter information, to detect if individual functions were hotspots (Figure 8).

4.4.2. Binary classification models

Although Ferenc et al. [12] explored using a few models for binary classification in their research, this project conducted further feature engineering and scaling to enhance model performance, reduce feature noise, and alleviate multicollinearity problems. To accomplish this, a Standard Scaler was used to standardize features to transform them to have a mean of 0 and standard deviation of 1, helping ensure that models were not influenced by disbalanced magnitudes. Additionally, mutual information-based feature selection was conducted to select features based on their linear and non-linear relationships with the target variable. It also outperformed the usage of chi-squared and ANOVA F-value tests for feature selection for the baseline KNN model (since this classification algorithm was recommended by the original researchers due to its good performance). Figure 8 (next page) shows the list of features selected for this task.

Metric	Description
CYCL	Cyclomatic Complexity
PARAMS	Number of Parameters
HOR_D	Distinct Halstead Operators
HOR_T	Total Halstead Operators
HON_D	Distinct Halstead Operands
HON_T	Total Halstead Operands
HLEN	Halstead Length
HVOC	Halstead Vocabulary Size
HDIFF	Halstead Difficulty
HVOL	Halstead Volume
HEFF	Halstead Effort
HBUGS	Halstead Bugs
HTIME	Halstead Time
CYCL_DENS	Cyclomatic Density

Figure 8: Selected features for hotspot classification

The escomplex package [28] was tested and therefore selected to extract these features from all incoming input code before scaling and prediction. This processing is integrated at the final stage of the pattern-based analytics process (Figure 5).

As for the models, a wide range of binary classification models were selected for testing and evaluation against the baseline KNN model. Hyper-parameter tuning was conducted via a cross-validated grid-search approach, prior to the final comparisons, which are explained in detail in section 5.1. Besides the baseline KNN model, the models explored were: Logistic Regression, Decision Tree Classifier, Random Forest, XGBoost, Bagging and AdaBoost. Although most models greatly outperformed the model performances in Ferenc et al., they all exhibited a sizeable lack of recall, while being very high in precision – resulting in encouraging but misleading F1 scores.

For instance, XGBoost performed the best with an ROC AUC of 0.96 and F1 of 0.75 but had a recall of only 0.67. Upon closer inspection the binary target variable seemed to be severely disbalanced with less positives than negatives, with a ratio of 12% for positives. Under-sampling negatives did increase XGBoost F1 and recall scores to 0.76

and 0.71 respectively, but reduced ROC AUC and precision to 0.93 and 0.82 respectively. The issue here was the moderately small size of the dataset with 12,125 rows, which undermined the under-sampling efforts. On the other hand, oversampling significantly decreased performance on the test set due to overfitting issues, undermining the feasibility of this approach.

4.4.3. Novel self-balancing weighted random forest approach

Due to the issue with dataset size, sampling, and performance, two different new approaches were explored.

The first of which is a modified weighted bagging (WB) approach, where each tree is given an ROC AUC weight for voting power based on their performance. While traditional bagging samples a proper subset from data for training, it leaves out data that is not sampled for each individual tree. This out-of-sample data is therefore leveraged in this approach to test each tree for its AUC score, which serves as the voting power to be used before prediction. While this outperformed traditional bagging for ROC AUC scores, the recall score remained low – even after experimenting with F1/recall scores as weights instead of ROC AUC.

The second approach is a self-balancing weighted random forest (SWRF) approach, that took WB even further. Instead of modifying a bagging model, this modified the traditional random forest algorithm with a few key changes:

- **Balanced training data for each tree**: In SWRF, each tree is trained on a balanced random sample created by under-sampling the majority class and over-sampling the minority class. Although individual trees train on under-sampled data, the overall model effectively trains on the entire dataset by utilizing different balanced samples across the ensemble.
- **ROC AUC-based weighting of trees**: Like the WB approach, each tree's contribution to the final prediction is determined by its ROC AUC weight. As

previously, instead of using a separate validation set, the un-sampled portion of the data for each tree is used to compute the AUC weight, ensuring that no additional data is wasted solely for weight calculation.

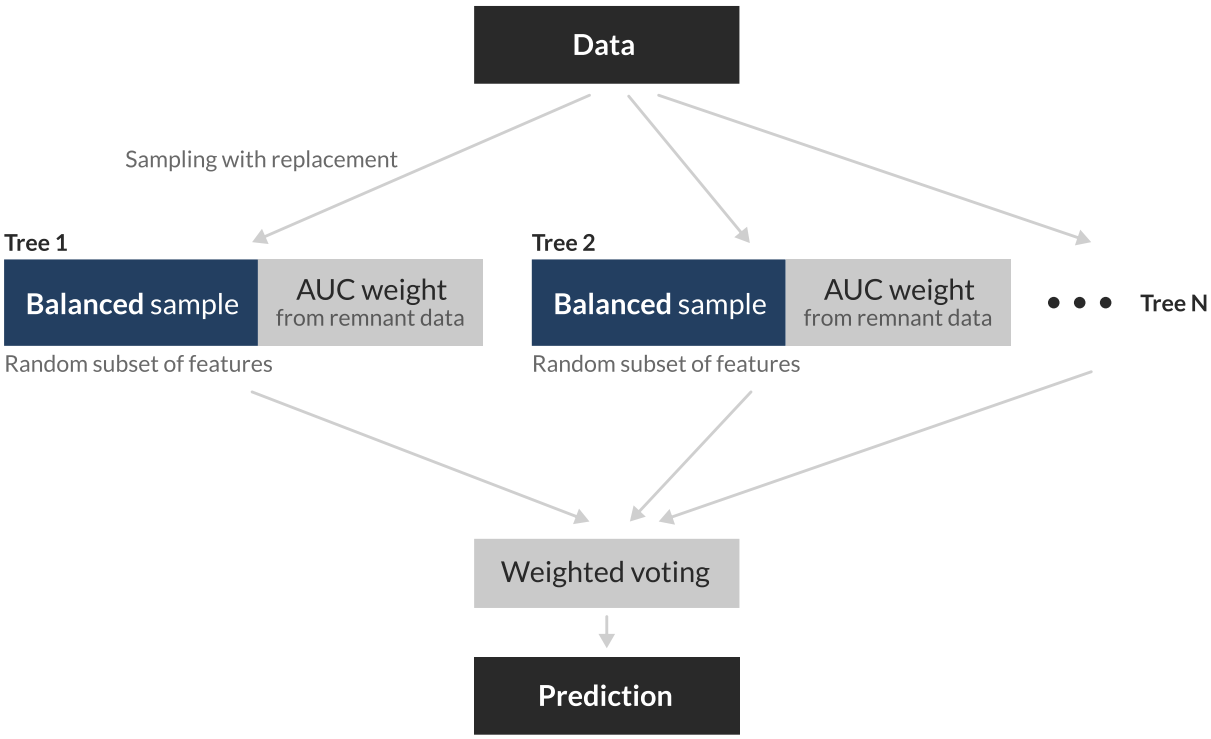

Figure 9: Abstract insight into how balanced sampling and weighting works in SWRF

The primary benefit of this approach is to prevent under-sampling or oversampling original data, as each tree is trained on different balanced samples – such that the whole ensemble model trains on the entire training dataset. This effectively combats the issue of the dataset size when it comes to oversampling and under-sampling seen earlier. The secondary benefit is the weighted voting mechanism – which ensures votes are determined by tree performance rather than being equal. Performance is further explored later in section 5.1.1.

On the mechanism of balanced sampling on the tree-level, the number of samples for each binary class is first determined as follows:

$$N_{minority}^{train} = \alpha \cdot N_{minority}$$

$$N_{majority}^{train} = \min\left(\beta \cdot N_{minority}^{train}, N_{majority}\right)$$

where $N_{majority}$ and $N_{minority}$ are number of total rows in majority and minority classes in the dataset respectively, while $N_{majority}^{train}$ and $N_{minority}^{train}$ are the number of rows for majority and minority respectively that each tree is trained on. Using these numbers, samples from majority and minority classes ($S_{majority}^{train}$, $S_{minority}^{train}$) are randomly sampled from the overall model training data for each individual tree. As shown, two hyperparameters are introduced because of this approach, on top of traditional random forest:

- **Minority class ratio (α)**: Proportion of the minority class (Cn) randomly sampled for each tree, with replacement. This ensures that trees train on different subsamples of the minority class in the overall data.
- **Majority class multiplier (β)**: Multiplier factor that determines the size of the majority class (Cm) sample, relative to the minority class sample for each tree. Majority class sample size is therefore computed by multiplying minority class sample size by β.

On the mechanism of evaluating ROC AUC scores for each tree, overall training data not being used in $S_{majority}^{train}$ and $S_{minority}^{train}$ are used for each tree. There are no leakage issues here for either the tree or SWRF, as it just uses the leftover remnants after each tree samples its training data, from the data used for training SWRF. These weights are used for probability estimates for binary classification as follows:

$$P(y = 1) = \frac{\sum_{i=1}^{n} \omega_i \cdot p_i}{\sum_{i=1}^{n} \omega_i}$$

where ω_i, p_i, n are tree weights (ROC AUC scores), tree predictions and number of trees respectively. A simple 0.5 threshold is used on the probability to get prediction $\hat{y}$.

4.5. Detecting bugs and recommending patches

For the purposes of bug detection, we can first identify two major groups of bugs: those that follow specific patterns, and those that do not. The first category necessitates a pattern-based approach where pieces of code are analyzed through their ASTs for pre-defined bug patterns. This is different from smells, as smells indicate a deeper issue, but might not necessarily be issues themselves – while bugs are the incorrect or directly problematic use of code. The second category however requires a deeper analysis of complex patterns through machine learning. The table below details the pattern-based bugs this project detects, and their patterns.

Pattern-based bug	Patterns in ASTs
Incorrect operator use	Binary expression nodes with incorrect operators
Function call before definition	Call expression nodes with callee names found before function declaration nodes
Usage of reserved words	Identifier nodes matching a dictionary of reserved words
Invoking non-functions	Call expression nodes where callee is not a function declaration identifier
Mismatched destructuring	Variable declarator nodes with array or object in the id property, but mismatched type in init property
Unhandled promises	Call expressions with ‘then’ usage, but no ‘catch usage’, or a lack of ‘then’ in the first place

Figure 10: *Pattern-based bugs and their AST patterns*

These pattern-based bugs will be detected in the closure/block analysis stage. The following sections detail how the second category with more complex bugs will leverage machine learning for detection and finding potential patches.

4.5.1. Potential datasets and methods

To detect more complex bugs beyond pattern-based ones, several dataset options were considered for detection using machine learning methods. One approach involved scraping StackOverflow for common JavaScript questions and corresponding answers – such that the answers would serve as the fixes, while the questions the buggy code. This immediately proved infeasible because many questions did not necessarily include buggy code, and many answers did not include chunks that were direct fixes for the questions (more explanations than code chunks).

Another approach involved scraping the GitHub archive for pull requests containing bug patches, by searching for keywords such as "bug" and "patch" in comments and storing code chunks before and after the patch. However, this method had challenges in terms of quality, as randomly scraping pull requests also resulted in gathering data from low-quality pull requests and repositories. Consequently, the initial performance of a natural language LSTM baseline model for classification was suboptimal.

To address these concerns, an alternative approach was adopted, utilizing the BugsJS benchmark dataset [29], which comprises of a more curated list of commit hashes and comments before and after patches in GitHub pull requests. Through this list and the GitHub API, a dataset of buggy and corresponding patched functions was created by scraping the listed commit hashes. The data was then preprocessed by cleaning comments, removing indentations and new lines, and standardizing prefix stubs for function declarations. The size of the dataset eventually reached 42,627 pairs of buggy and fixed code. This dataset was then employed for two tasks in the following sections: binary defect classification in code, and sequence-to-sequence patch generation.

4.5.2. Encoder models

A basic encoder-LSTM model was initially chosen as the baseline to compare datasets and compare further models. The LSTM architecture was chosen over a traditional RNN due to its ability to capture long-term relationships in larger functions. However, performance was unsatisfactory with an ROC AUC of 0.62.

4.5.3. Transfer learning

With successes demonstrated with using pre-trained models and transfer learning to detect and patch Java bugs with Tufano et al. [11], a similar approach was considered using BERT, RoBERTa, CodeBERT and CodeT5. Among these models, BERT, RoBERTa and CodeBERT were trained for an encoder-only binary classification task to indicate whether a particular function is buggy. CodeBERT had better performance than BERT and RoBERTa, primarily due to it being trained on CodeSearchNet [30].

CodeT5, however, was trained for both bug detection and patch generation due to its encoder-decoder architecture. Although also pre-trained on CodeSearchNet, it outperformed CodeBERT [31]. Since CodeT5 allows multi-task learning, a singular model was trained for both detection and patch generation tasks with the prefixes “Defect:“ and “Refine:” respectively, to output a Boolean string for bug classification, and the patched function for patch generation.

For each input file into the framework’s engine, functions are first extracted through the closure/block analysis stage using AST nodes, then special characters, indentations and new lines are removed prior to tokenization. For patched code outputs from CodeT5, they are first processed via an AST parser, and re-rendered as a string, to ensure it has natural indentations and new lines instead of the raw form. If any output for patched code has syntax or structural errors, the AST parser will also pick it up while attempting to parse, preventing it from being sent as output.

5. Evaluation of JSReview

Due to the hybrid nature of the framework and the intertwining of learning and pattern-based methods, evaluation is done in two stages: quantitative and qualitative. The quantitative section details the performance metrics for the machine learning models employed in bug and vulnerability detection, while the qualitative section evaluates the standards and feasibility of the predominantly pattern-based methods, as well as sample outputs from the learning-based models.

5.1. Quantitative evaluation

5.1.1. Vulnerable hotspot detection

To assess the performance of the binary classification models using the vulnerability dataset (where a positive means an existence of a hotspot), the key metrics that were analyzed were ROC AUC, F1 score, precision, and recall. The following table summarizes the performances of the tested model, using the same training dataset, and the same preserved test dataset for evaluation for every model. These performances are also following the hyperparameter tuning, scaling and feature selection – and represent the best scores achieved for each model.

Model	ROC AUC	F1 Score	Precision	Recall
Logistic Regression	0.66	0.41	0.77	0.28
K Nearest Neighbors	0.88	0.70	0.88	0.58
Decision Tree	0.72	0.64	0.71	0.58
Bagging	0.90	0.73	0.83	0.65
Random Forest	0.94	0.76	**0.90**	0.66
AdaBoost	0.89	0.69	0.87	0.57
XGBoost	0.96	0.75	0.85	0.67
WB (new approach)	0.95	0.74	0.87	0.63
SWRF (new approach)	**0.98**	**0.80**	0.88	**0.72**

Figure 11: Vulnerability classification model performances (best in bold)

SWRF outperformed other models in ROC AUC, F1 Score, and Recall. This is expected due to its active efforts in weighting each tree based on AUC score, and self-balancing individual data samples for trees. Random forest had the highest precision.

Overall, the effectiveness of most models is encouraging, as it proves that real-world vulnerable hotspots in code have an underlying pattern that can be predicted by extracting software metrics.

5.1.2. Bug detection and patching

Like the evaluation of models for the vulnerability classification task, the train-test split of data was controlled and kept consistent throughout the process of testing all models. Unlike vulnerability classification however, the results were less clear-cut, with a lot of variance and unpredictability with hyperparameter tuning for each model. Nevertheless, this table details the performance of each model for the bug prediction binary classification task, where a positive means the existence of a bug.

Model	ROC AUC	F1 Score	Precision	Recall
LSTM	0.57	0.32	0.25	0.46
BERT	0.63	0.54	0.48	0.61
RoBERTa	0.63	0.57	0.52	0.63
CODEBERT	0.68	0.64	0.55	0.77
CodeT5	**0.77**	**0.73**	**0.60**	**0.93**

Figure 12: *Bug detection binary classification model performances (best in bold)*

CodeT5 outperformed other models in this binary classification task to indicate presence of bugs, but it remains biased towards high recall and low precision, despite attempts to change the architecture to include multi-layer perceptrons. This creates the challenge that not every flagged function includes bugs – but it still serves as a useful tool to guide developers to potential areas in code that might be buggy. Patch generation using the CodeT5 model on the other hand can mostly be evaluated qualitatively to understand its relevance and performance.

5.2. Qualitative evaluation

5.2.1. Code smells and maintainability risks

Due to the high degree of consistency and predictability that comes with purely pattern-based approaches employed by code smell and maintainability risk detection tasks, their outcomes are always of high quality. The only flaw here is the limitation that comes as a result, as they cannot go beyond these predefined patterns.

Nevertheless, these two tasks were evaluated through the manufacturing of 9 pieces of code for each smell and maintainability risk – with various variants – to see how well it copes with each. The outcomes were mostly as expected, with the exact lines being flagged for almost every test. 4 total tests out of the 72 did not pass due to a few bugs with code duplication, god object and inheritance overuse checks – but were soon debugged – after which the checks worked as expected.

After this test, all smell and maintainability checks were run against 2 major repository files of React and moment.js, which yielded positive results as it was able to indicate a lot of true positives, with a handful of false positives due to some difficult edge cases in the patterns – with the overall TPR-FPR ratio of 0.942 upon counting.

5.2.2. Patch generation using CodeT5

Like smells and maintainability risks, the React and moment.js repository was scanned to evaluate patch generation, with every extracted function from original files run against the CodeT5 detection and patch pipeline for heuristic review. While the detection yielded mostly true positives upon manually checking through flagged functions, the patch generation sometimes worked well – especially when the bugs were common such as using logical operators between different types, forcing a type coercion or impossibility to be equal. But for more implicit bugs it failed and output the exact same function as the input. Two perfect examples taken from testing using synthetically manufactured bugs are shown in Figure 13.

Perfect example 1: Returning unexpected data format

```
function calculateTotal(items) {
  let total = 0;
  for (let i = 0; i < items.length; i++) {
    total += items[i].price;
  }
  return total.toFixed(2); // returning a string instead of a number
  return total;
}
```

Perfect example 2: Invalid loop conditions

```
function findMax(arr) {
  let max = arr[0];
  for (let i = 1; i <= arr.length; i--) { // loop will crash
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] > max) {
       max = arr[i];
    }
  }
  return max;
}
```

Figure 13: Perfect patches from the patch generation task

5.2.3. Relevance of vulnerability classification

Despite having excellent quantitative performance with the vulnerability classification models in section 5.1.1, more exploration was done to verify the relevance of the dataset, and what exactly a positive meant. Upon testing across synthetic functions, React and moment.js repository files, the positives mostly belonged to functions that contained very long, complex, or convoluted logic – that was difficult to understand by just reading the code. This was encouraging, as complex pieces of code would intuitively increase the risk of vulnerabilities through a difficulty in maintenance, increased risk of human error, and difficulty in best-practice enforcement. One typical synthetic example that was flagged by the SWRF model is shown in Figure 14.

Vulnerable hotspot positive example

```
function validateUserInput(input, options) {
  const { allowHtml, maxLength, minLength, allowedChars } = options;
  let sanitizedInput = input.trim();
  if (!allowHtml) {
    sanitizedInput = sanitizedInput.replace(
      /<[^>]*>/g, ''
    );
  }
  if (
    minLength &&
    typeof minLength === 'number' &&
    sanitizedInput.length < minLength
  ) {
    throw new Error(
      `Input length must be at least ${minLength}`
    );
  }
  if (
    maxLength &&
    typeof maxLength === 'number' &&
    sanitizedInput.length > maxLength
  ) {
    sanitizedInput = sanitizedInput.slice(0, maxLength);
  }
  if (allowedChars) {
    const regex = new RegExp(`^[${allowedChars}]+$`);
    if (!regex.test(sanitizedInput)) {
      throw new Error('Invalid characters in input');
    }
  }
  return sanitizedInput;
}
```

Figure 14: Example of a function flagged by SWRF for being a vulnerable hotspot

6. CI/CD platform using JSReview

Using the JSReview framework's engine as its core, a GitHub CI/CD and repository scanning platform was developed to demonstrate the real-world use of the framework. This platform is intended to be used by small to medium sized professional software engineering teams, who do not have dedicated software testers and might find it difficult reviewing code in terms of their opportunity cost – as it takes precious time away from actual development.

6.1. Platform objectives

The primary objectives of this platform encompass the reduction of human intervention and effort in the following key areas:

- **Comprehensive project quality evaluation**: The platform aims to thoroughly assess the overall quality of a project, identifying and flagging potential issues within the repository that warrant attention or verification.
- **Streamlined pull request review process**: By assisting developers in refining their code revisions and aiding reviewers in identifying flaws and defects in incoming pull requests, the platform seeks to minimize the time and effort for both parties in this crucial aspect of the development process.

6.2. Platform requirements

To effectively fulfill its objectives, the platform must satisfy several tangible requirements:

- **GitHub repository integration**: Ability for users to effortlessly add repositories to begin scanning and tracking their repository and incoming pull requests automatically.

- **Comprehensive repository report**: Users must have access to a clear and concise overview of all identified issues across the four JSReview categories - code smells, bugs, vulnerable hotspots, and maintainability risks - following a repository-wide scan. The evaluation outcomes should be intuitive, clear, concise, and actionable for developers to improve their project.
- **Automating pull request scans as a CI/CD check**: The platform should automatically run newly revised code through the JSReview engine upon initiations of pull requests, so developers and reviewers are able to access evaluation reports.

6.3. System design

To achieve these requirements, a streamlined system architecture was derived to leverage the JSReview framework to support all use cases in a straightforward manner.

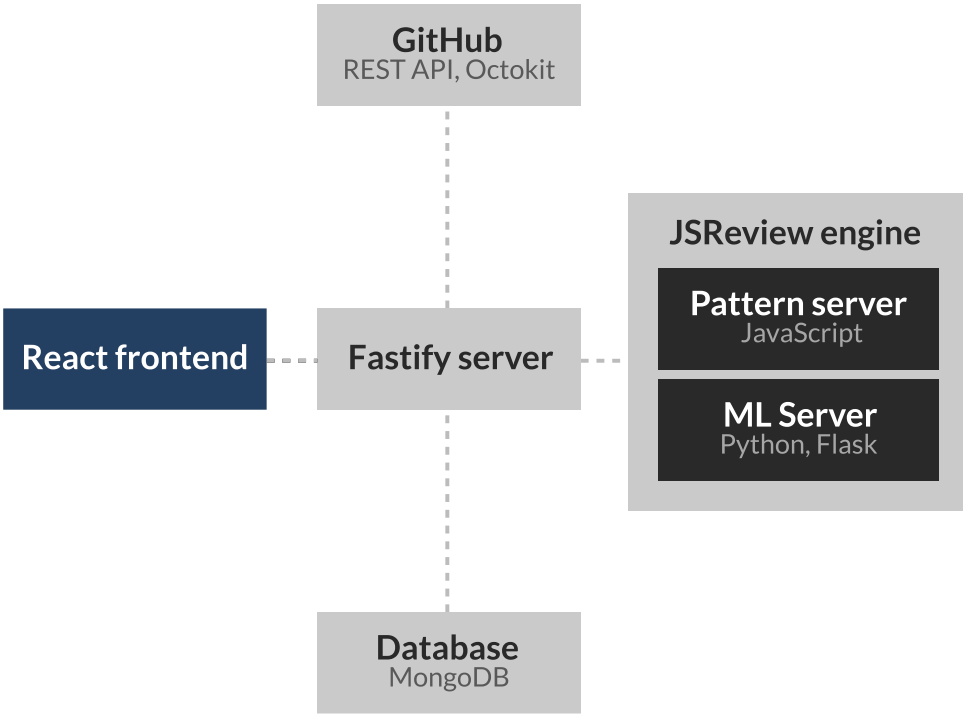

Figure 15: Overview of the platform's architecture

At the core of the architecture sits a JavaScript Fastify application server, which connects to the JSReview engine servers, the MongoDB database, the React (Vite) frontend, and GitHub REST API and Octokit interfaces. JavaScript was chosen for this server due to its native integration with the MongoDB query language, support for AST parsing of code, and sufficient integration with GitHub's Octokit Node package. This server also listens to pull requests and commit changes on a repository via webhook events with Octokit, and initiates scans on updates by calling upon the JSReview engine for analysis through the process described in section 4.

As for the database, all user data and scans are stored in MongoDB collections. The database has two collections to achieve all its functionality: "users" and "scans". User data, and scan histories based on each user are stored in the former, while repository and pull request scans are stored in the latter. MongoDB was chosen due to the complexity and semi-structured nature of both GitHub data as well as the scans that JSReview outputs. SQL-based databases are less feasible with JSReview, as it might lead to overly complex table structures, and rows with many null values. This is because scans have positives for some fields, negatives for others – and the entire data structure that the implementation of the JSReview engine outputs is deeply nested. Figure 16 explains this data structure.

JSReview framework engine's output format in JSON, for one scan

```
{
  _id: <user-repo-hash>,
  type: <repo or pull>,
  timestamp,
  files: [
    {
      path,
      smells: [{ start, end, type }],
      bugs: [{ start, end, type }],
      maintainability: [{ start, end, type }],
      hotspots: [{ start, end }]
    }
  ]
}
```

Figure 16: JSON format of output from JSReview (start/end refer to line numbers)

As for the implementation of the framework's core engine, JavaScript was chosen for pattern-based analysis and AST-related tasks, as it is significantly more effective and straightforward to process JavaScript ASTs, and match patterns against ASTs using JavaScript and Acorn, compared to Python ports of JavaScript parsers like esprima. Machine learning tasks were done in a Python sever instead, due to the extensive support in terms of ML packages like scikit-learn and PyTorch.

6.4. Interface design

This section will detail some key interface screenshots to visualize how the users will interact with the system to take advantage of JSReview's capabilities. This includes the flow from authenticating via a GitHub account, adding repositories, browsing these added repositories, checking repository scan results, and checking pull request scan results. These are some key screens instead of an exhaustive set.

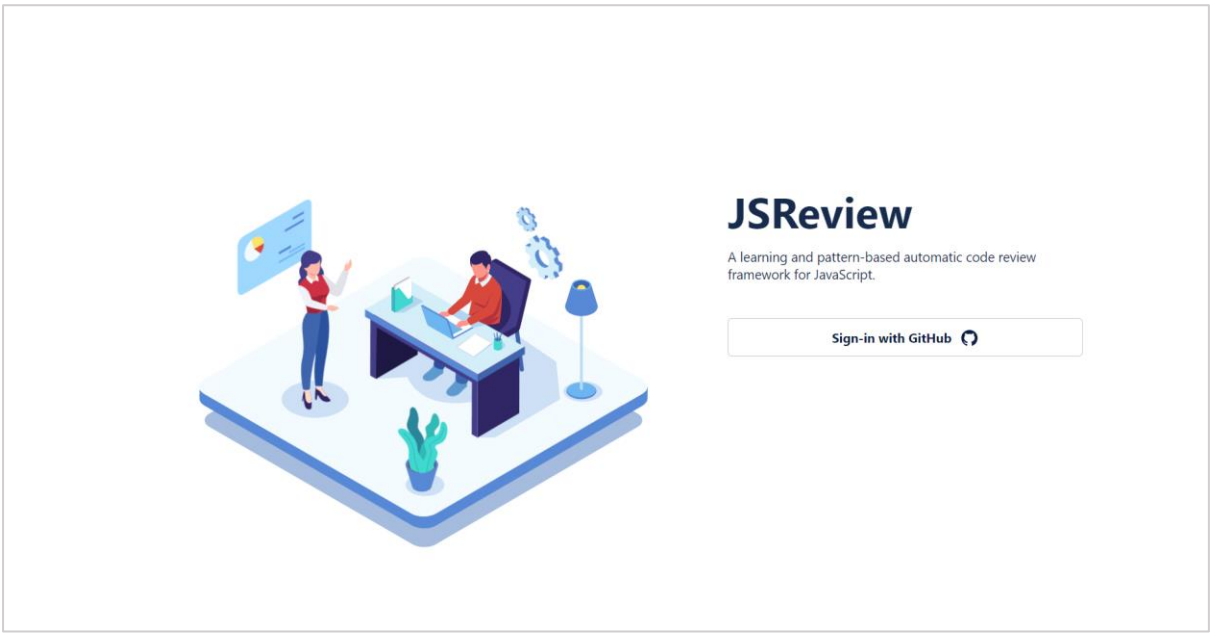

Figure 17: Signing in via GitHub

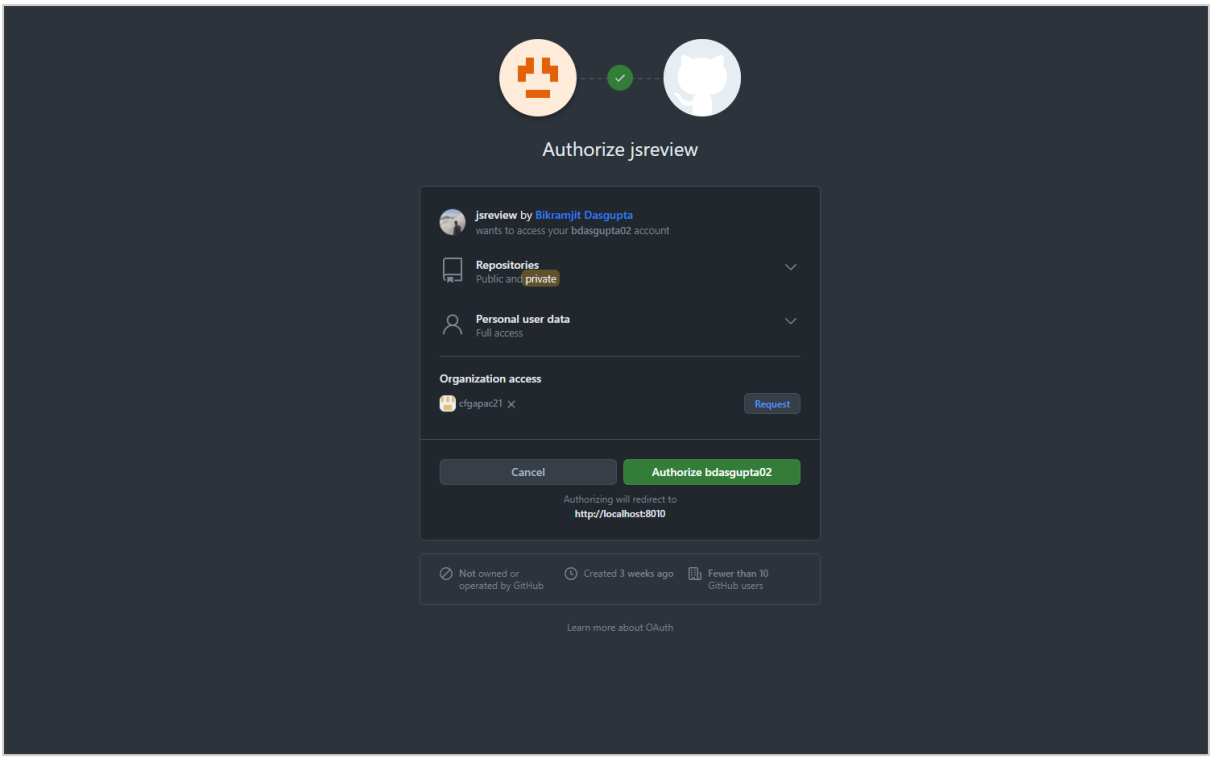

Figure 18: Authorizing via GitHub during signing in

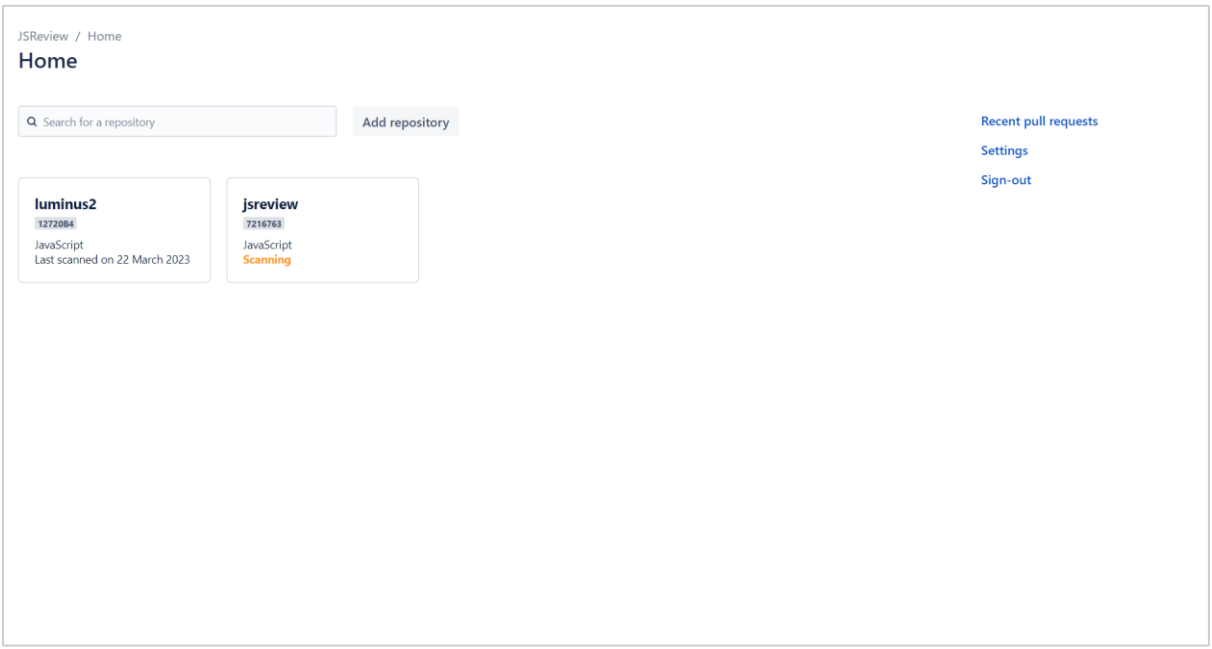

Figure 19: List of added repositories and statuses for scanning, and button to add new

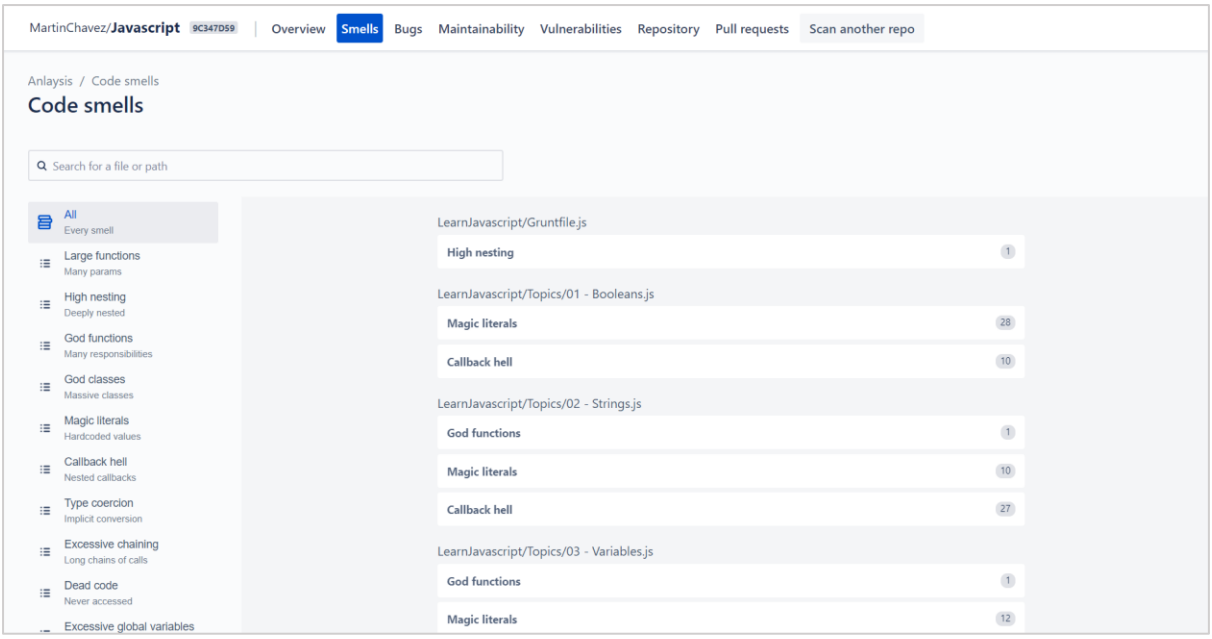

Figure 20: Report of different smells from the repository-level scan

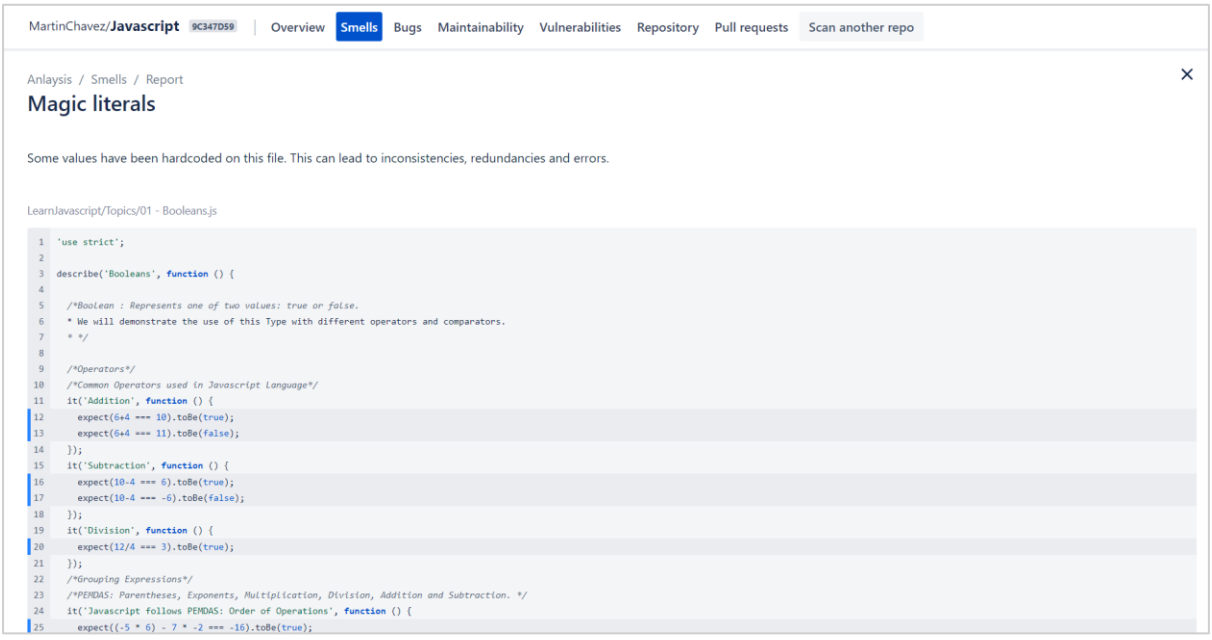

Figure 21: Clicking on the details of each file's smells brings up the code view, detailing locations of smells from one file, for one category of smells

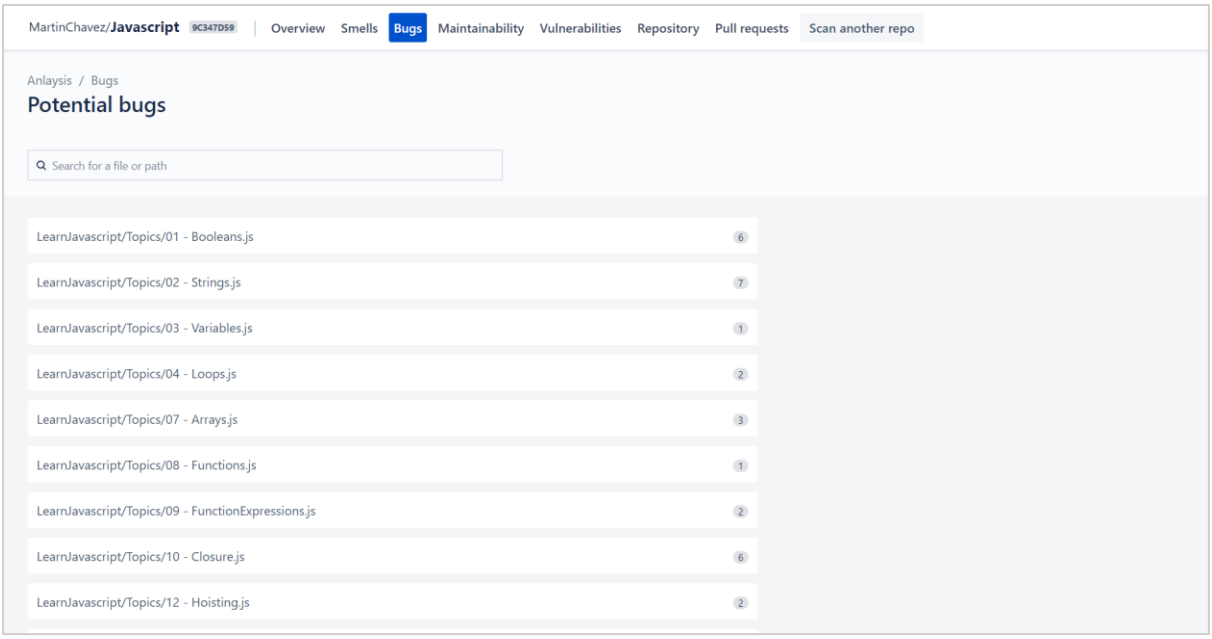

Figure 22: List of bugs from the repository-wide scan

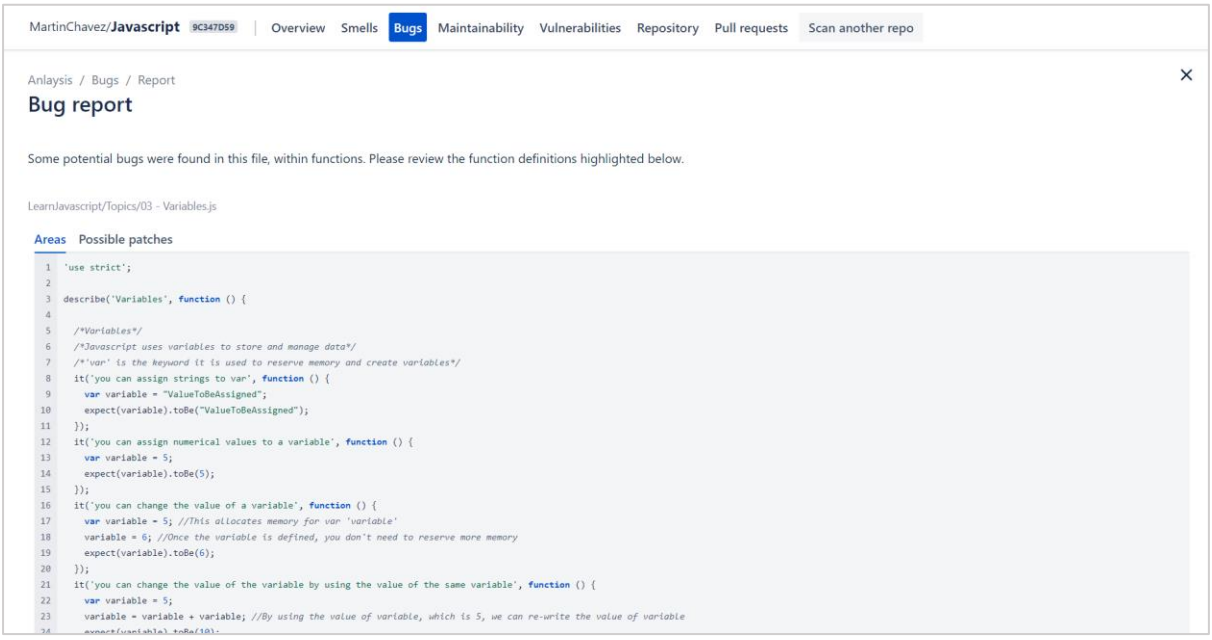

Figure 23: Details of each file, with any bugs it has inside. Potential patches can be accessed through this page via the tabs

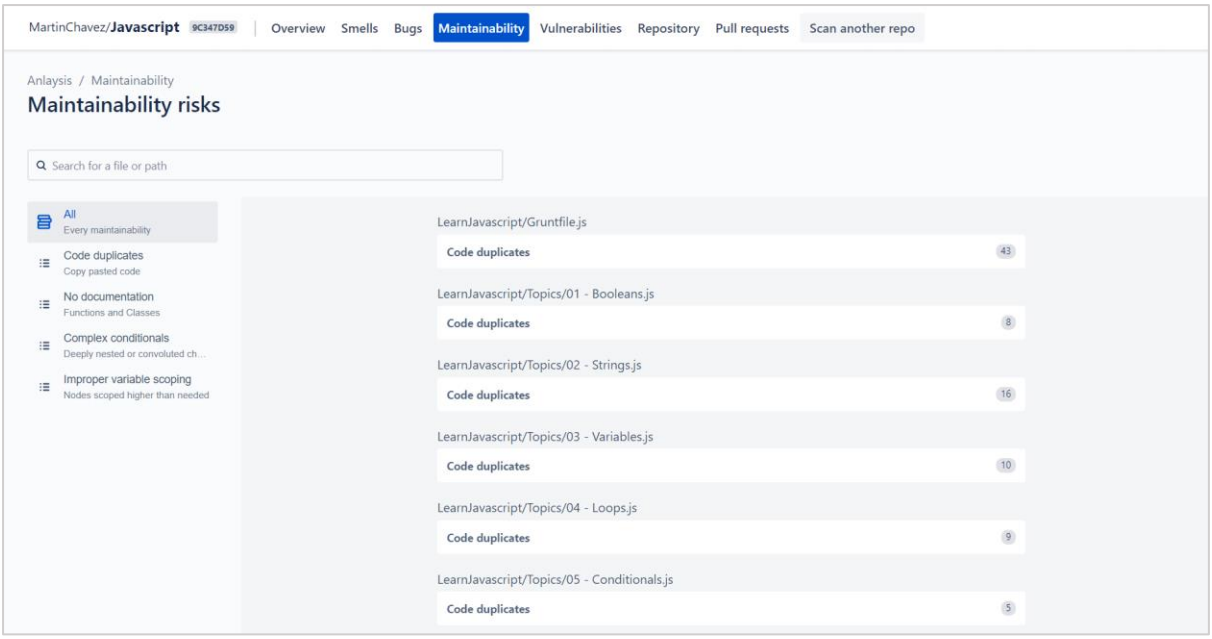

Figure 24: List of maintainability risks from repository-wide scan

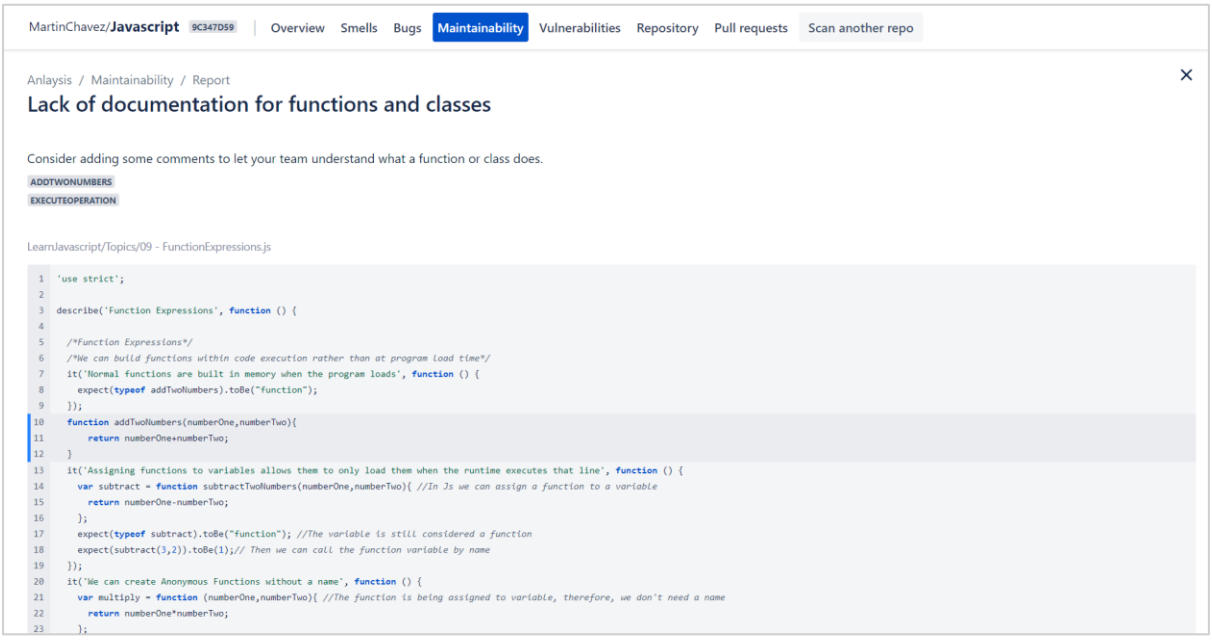

Figure 25: Details of maintainability risks, per risk, per file

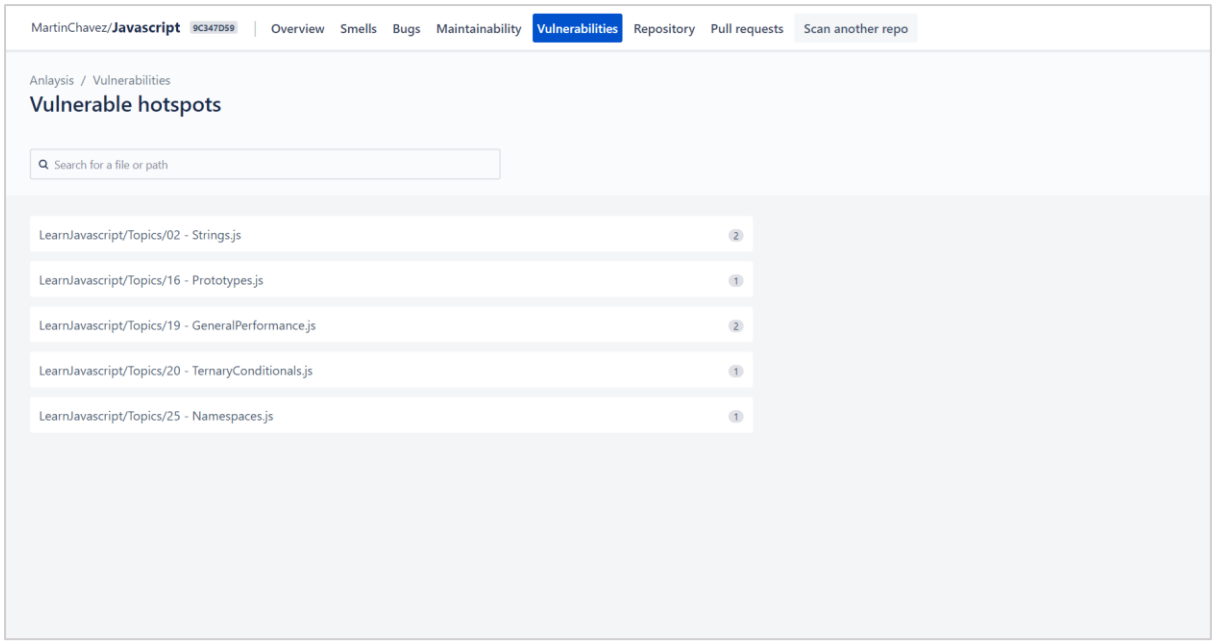

Figure 26: List of vulnerable hotspots found in the repository. Clicking on one will lead to a similar page as the bug report page for details

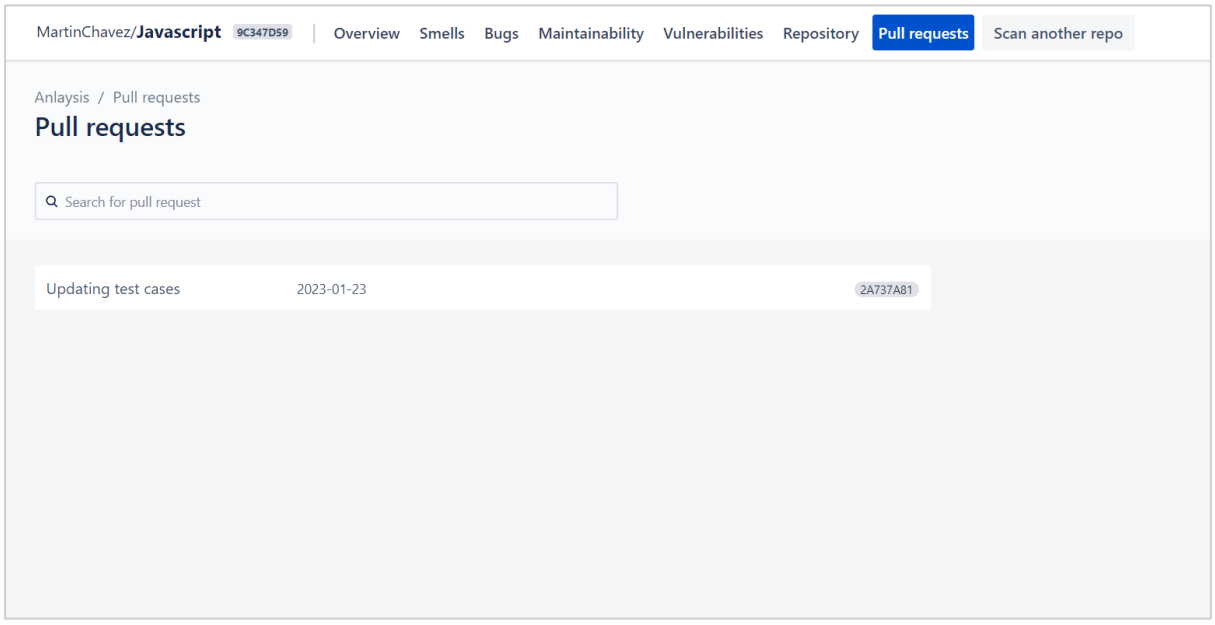

Figure 27: List of scanned pull requests. These are automatically scanned when created on GitHub, and opening details leads to the previously shown scan details page, but specific to the scans of revised changes, instead of the whole repository

6.5. Usability and benefits

The interface is inspired by other CI/CD quality checking tools and dashboards, including SonarQube. This is because developers have a pre-existing mental model on how these products work – so they may use this platform with greater ease and familiarity. The interface is also designed with visibility in mind, without going too much into nested page structures that increase the clicks required to reach the code detail pages to conduct reviews. As shown in 6.4, all pages explaining the locations and types of issues are file-centric, as one file may have multiple instances of the same issue. This is based on the idea that it is more intuitive for developers to search for issues on a file-basis instead of an issue-basis.

To evaluate the benefits of this platform, three toy projects with different JavaScript frameworks with synthetically created issues were created. Mock pull requests for each were then created to self-evaluate the efficacy of the platform compared to manual reviews. The platform on average had the ratio of clicks taken to review of 0.34 when compared with manual reviews. Time taken to review was also decreased by 47% - calculated by time taken to access and find issues. The core JSReview engine successfully picked up most issues, leading to a faster search time.

6.6. Learnings from the platform

This platform serves as a practical example of how JSReview can be integrated with a practical tool that developers may actually use. It details how the results can be interpreted through a usable frontend, and the different ways the same core engine can be used (in this case for both repository-wide scans, as well as pull-request scans).

Similarly, the core framework can be used across other projects as well. Some ideas include a VSCode plugin to scan projects within IDEs during development, and an educational platform for students where the engine is used to teach them about issues with their code, and best practices associated with each type of issue.

7. Limitations

The JSReview framework, while offering valuable insights and assistance in JavaScript code analysis, has a few key limitations that could be addressed in future iterations. One such limitation is the reliance on static analysis techniques, which may miss logical issues in code that are detectable through automated testing. While the framework is intended to complement developer-driven automated testing, its dependence on static analysis makes it less comprehensive.

Another limitation is the proof-of-concept nature of bug prediction and patching. Despite demonstrating promising performance, the bug prediction and patching components of the framework currently serve more as a proof of concept than a final reliable implementation. This is a result of their weaknesses in precision and the scale/quality of the dataset used. The datasets for both bugs and hotspots also may not cover the entire range of scenarios due to their datasets, leading to some false negatives.

Additionally, the performance of the JSReview engine is limited by its relatively slow scanning speed, with an estimated processing rate of 500 lines per minute. The primary cause of this issue is the inference duration of the bug detection and patching model. This could be problematic for large-scale projects or time-sensitive workflows, especially for the case of the CI/CD platform.

Finally, model generalizability may be undermined by the sheer variance of JavaScript project standards across organizations and individuals. This variability presents a challenge in creating a universally applicable tool, as it must cater to a wide range of coding practices and styles. Increasing dataset scale, and accounting for generalized variations of patterns would be a step forward to alleviate this concern.

8. Conclusion

This project has proposed the design and implementation of the JSReview framework, which represents a step forward in a practical application of ACR techniques for the real-world. By targeting the core aspects of static ACR – bugs, smells, maintainability, and vulnerability – JSReview evaluates code quality through a multifaceted spectrum. Likewise, the integration of learning and pattern-based techniques together allows the framework in reaching its objectives in these four areas. While JSReview has its limitations, its performance across the four domains is encouraging, and it therefore services as the foundation for future development, hoping to inspire refinement in the rapidly evolving realm of ACR.

9. Future work

Beyond combatting limitations from section 7, JSReview provides a plethora of opportunities for future development. One such opportunity is the extension of the framework to other programming languages or technology stacks. Given that the fundamental principles of the framework's overall design are language agnostic, the hybrid and modular structure can be replicated or further refined for other languages.

Another promising direction for future investigation is the exploration of alternative models for specific tasks within the framework. For example, large language models could be employed for bug detection and patch generation, while search-based models could be utilized for vulnerability detection. The integration of additional ACR aspects into the framework's stages could lead to a more comprehensive evaluation process, encompassing a broader range of code quality dimensions.

Overall, the modular nature of the framework, and its abstract design principles, create a lot of room for future development and refinement.

References

[1] O. Kononenko, O. Baysal and M. W. Godfrey, "Code review quality: how developers see it," *ICSE '16: Proceedings of the 38th International Conference on Software Engineering,* pp. 1028-1038, 2016.

[2] S. Carey, "Complexity is killing software developers," InfoWorld, 1 November 2021. [Online]. Available: https://www.infoworld.com/article/3639050/complexity-is-killing-software-developers.html.

[3] J. He, L. Wang and J. Zhao, "Supporting Automatic Code Review via Design," *2013 IEEE Seventh International Conference on Software Security and Reliability Companion,* 2013.

[4] K. F. Tómasdóttir, M. Aniche and A. V. Deursen, "The Adoption of JavaScript Linters in Practice: A Case Study on ESLint," *IEEE Transactions on Software Engineering,* vol. 46, no. 8, pp. 863-891, 2020.

[5] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," 2006.

[6] Google, "Gerrit," [Online]. Available: https://www.gerritcodereview.com/. [Accessed 9 March 2023].

[7] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida and K. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER),* 2015.

[8] M. B. Zanjani, H. Kagdi and C. Bird, "Automatically Recommending Peer Reviewers in Modern Code Review," *IEEE Transactions on Software Engineering,* vol. 42, no. 6, pp. 530-543, 2016.

[9] H. Li, S. Shi, F. Thung, X. Huo and B. Xu, "DeepReview: A view: Automatic code r omatic code review using deep multi-instance view using deep multi-instance," *Advances in knowledge discovery and data mining: 23rd Pacific-Asia Conference, PAKDD,* pp. 318-330, 2019.

[10] B. Wu, B. Liang and X. Zhang, "Turn tree into graph: Automatic code review via simplified AST driven graph convolutional network," *Knowledge-Based Systems,* vol. 252, 2022.

[11] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk and G. Bavota, "Using Pre-Trained Models to Boost Code Review Automation," *Proceedings of the 44th International Conference on Software Engineering,* pp. 2291-2302, 2022.

[12] R. Ferenc, P. Hegedus, P. Gyimesi, G. Antal, D. Ban and T. Gyimothy, "Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions," *IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE),* 2019.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for," 2018.

[14] S. Chakraborty, R. Krishna, Y. Ding and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?," *IEEE Transactions on Software Engineering,* vol. 48, no. 9, pp. 3280-3296, 2022.

[15] A. M. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript Code Smells," *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM),* 2013.

[16] C. L. Goues, M. Pradel, A. Roychoudhary and S. Chandra, "Automatic Program Repair," *IEEE Software,* vol. 38, no. 4, pp. 22-27, 2021.

[17] N. A, "Evaluation of JavaScript Quality Issues and Solutions for Enterprise Application Development," *Lecture Notes in Business Information Processing,* vol. 200, 2015.

[18] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet and R. Berg, "Saving the world wide web from vulnerable JavaScript," *ISSTA '11: Proceedings of the 2011 International Symposium on Software Testing and Analysis,* pp. 177-187, 2011.

[19] D. Graziotin and P. Abrahamsson, "Making Sense Out of a Jungle of JavaScript Frameworks," *Lecture Notes in Computer Science,* vol. 7983, pp. 334-337, 2013.

[20] M. Heller, "InfoWorld," 1 October 2022. [Online]. Available: https://www.infoworld.com/article/3210589/what-is-nodejs-javascript-runtime-explained.html. [Accessed 8 January 2023].

[21] "DeepScan," [Online]. Available: https://deepscan.io/. [Accessed 21 January 2023].

[22] "SonarQube," [Online]. Available: https://www.sonarsource.com/products/sonarqube/. [Accessed 6 February 2023].

[23] A. Saboury, M. P, F. Khomh and G. Antoniol, "An empirical study of code smells in JavaScript projects," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER),* 2017.

[24] J. Padolsey, Clean Code in JavaScript: Develop reliable, maintainable, and robust JavaScript, 2020.

[25] G. Lin, S. Wen, Q. L. Han, J. Zhang and Y. Xiang, "Software Vulnerability Detection Using Deep Neural Networks: A Survey," *Proceedings of the IEEE ,* vol. 108, no. 10, 2020.

[26] S. S and P. M, "A New Perspective on the Tree Edit Distance," *International Conference on Similarity Search and Applications,* pp. 156-170, 2017.

[27] "Snyk Vulnerability Database," [Online]. Available: https://security.snyk.io/. [Accessed 9 February 2023].

[28] "escomplex," npm, [Online]. Available: https://www.npmjs.com/package/escomplex. [Accessed 5 March 2023].

[29] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc and A. Mesbah, "BugsJS: a Benchmark of JavaScript Bugs," *12th IEEE Conference on Software Testing, Validation and Verification (ICST),* 2019.

[30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *Findings of the Association for Computational Linguistics,* p. 1536–1547, 2020.

[31] Y. Wang, W. Wang, S. Joty and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Conference on Empirical Methods in Natural Language Processing*, 2021.