Online Journal of Communication and Media Technologies, 2024, **14**(4), e202460
e-ISSN: 1986-3497

OPEN ACCESS **Research Article**

Enhancing integration testing efficiency through AI-driven combined structural and textual class coupling metric

Iyad Alazzam [1*]
0000-0001-7539-0822

Anas Mohammad Ramadan AlSobeh [1]
0000-0002-1506-7924

Basil Bani Melhem [1]
0009-0006-7110-0326

[1] Department of Information Systems, Faculty of Information Technology and Computer Science, Yarmouk University, Irbid, JORDAN
[*] Corresponding author: eyadh@yu.edu.jo

Citation: Alazzam, I., AlSobeh, A. M. R., & Bani Melhem, B. (2024). Enhancing integration testing efficiency through AI-driven combined structural and textual class coupling metric. *Online Journal of Communication and Media Technologies, 14*(4), e202460. https://doi.org/10.30935/ojcmt/15524

ARTICLE INFO

Received: 11 May 2024
Accepted: 13 Aug 2024

ABSTRACT

Integration testing, a critical and resource-intensive phase in the software development lifecycle, can account for up to a high percentage of the total testing cost. Identifying classes with high coupling is crucial for efficient integration testing, as these classes are more susceptible to the impact of maintenance-related changes. This research introduces a novel metric called *combined structural and textual class coupling* (CSTCC), which harnesses the power of artificial intelligence (AI) techniques to predict and rank the most critical classes in an object-oriented software system. CSTCC integrates structural coupling metrics with *latent semantic indexing* (LSI)-based textual coupling, providing a comprehensive measure of class coupling. LSI, an information retrieval technique, analyses the semantic relationships between classes based on their textual content, enabling CSTCC to capture both structural and conceptual dependencies, resulting in a more accurate identification of high-risk classes. The effectiveness of the proposed approach is rigorously evaluated using mutation testing on four Java open-source projects, and the results demonstrate that test cases developed based on CSTCC achieve high mutation scores, indicating their ability to detect a significant percentage of integration faults. By focusing testing efforts on high-coupling classes identified by CSTCC, developers can potentially save time and cost during integration testing. The results demonstrate that test cases developed based on CSTCC achieve high mutation scores, ranging from 98% to 100%, indicating their ability to detect a significant percentage of integration faults. Additionally, the approach results in substantial efficiency gains, with a notable reduction in the number of test cases needed, saving up to 33.3% of the testing effort in some cases. By focusing testing efforts on high-coupling classes identified by CSTCC, developers can potentially save time and cost during integration testing. The CSTCC metric provides a novel and effective approach to prioritize testing resources and improve the efficiency of integration testing in object-oriented software systems.

Keywords: integration testing optimization, mutation testing, test cases, coupling, AI-driven software testing, latent semantic indexing, combined coupling metrics

INTRODUCTION

Integration testing is a crucial phase in the software development lifecycle, focusing on verifying the interactions and compatibility between different modules or components of a software system (Khan & Sadiq, 2011). However, it is also one of the most resource-intensive and time-consuming testing activities, accounting for up to 70% of the total testing cost (Roongruangsuwan & Daengdej, 2010). The challenges associated with integration testing become more pronounced as software systems evolve and grow in complexity, particularly in object-oriented (OO) systems where classes serve as the fundamental building blocks (Alazzam, 2012).

Coupling, which refers to the degree of interdependence between software components, plays a significant role in the context of integration testing (Poshyvanyk et al., 2009). Classes with high coupling are more likely to be impacted by changes and prone to integration errors, making their identification crucial for efficient testing efforts (Goel & Gupta, 2017). Traditional approaches to measuring coupling primarily rely on structural metrics, such as analyzing the static code structure and counting the number of dependencies between classes (Revelle et al., 2011). However, these structural metrics may not fully capture the semantic relationships and conceptual dependencies between classes (Ali et al., 2007).

To address the limitations of existing coupling metrics and enhance the efficiency of integration testing, this research introduces a novel metric called *combined structural and textual class coupling* (CSTCC). CSTCC harnesses the power of artificial intelligence (AI) techniques to provide a comprehensive measure of class coupling by integrating structural coupling metrics with *latent semantic indexing* (LSI)-based textual coupling (Alenezi, 2014). LSI is an information retrieval technique that analyzes the semantic relationships between classes based on their textual content, enabling CSTCC to capture both structural and conceptual dependencies (Poshyvanyk et al., 2009).

The main objective of this research is to investigate the effectiveness of CSTCC in predicting and ranking the most critical classes in an OO software system, thereby facilitating more efficient allocation of testing resources. By focusing testing efforts on high-coupling classes identified by CSTCC, developers can potentially uncover a higher number of integration errors while optimizing the utilization of testing resources (Liu & Chen, 2014).

Unlike previous approaches that rely solely on structural or textual coupling metrics, our work combines both aspects using AI techniques. This integration allows for a more comprehensive assessment of class coupling, capturing both explicit dependencies and semantic relationships. Furthermore, our approach leverages machine learning (ML) to predict critical classes, offering a more dynamic and adaptive solution compared to static metric-based methods (Alsobeh, 2023).

The significance of this research lies in its potential to improve the efficiency and effectiveness of integration testing in OO systems. The CSTCC metric provides a novel and comprehensive approach to assess class coupling by leveraging AI techniques, specifically LSI, to capture both structural and semantic dependencies (Gethers & Poshyvanyk, 2010). This holistic view of coupling enables more accurate identification of high-risk classes, allowing developers to prioritize testing efforts and allocate resources more effectively (Briand et al., 2002).

To evaluate the effectiveness of the proposed approach, this research conducts rigorous empirical studies using mutation testing on four Java open-source projects. Mutation testing is a well-established technique for assessing the fault detection capability of test cases (Jia & Harman, 2011). By comparing the performance of test cases developed based on CSTCC with those based on traditional coupling metrics, this research aims to demonstrate the superiority of the proposed approach in detecting integration faults (Harrold & Rothermel, 1998).

The key contributions involve the development of a novel AI-driven CSTCC metric that combines structural and textual coupling information. Providing an LSI-based approach to capture semantic relationships between classes. An ML model to predict critical classes for integration testing. The remainder of this paper is organized as follows: Next section reviews related work in software metrics and AI-driven testing approaches. Then we describe the proposed CSTCC metric and the AI-driven methodology. After that we present the experimental setup and results. We then discuss the findings and implications. Finally, we conclude the paper and outlines future work.

LITERATURE REVIEW

Software metrics are quantitative measures of specific features within software that serve as valuable tools for monitoring development progress and evaluating the quality of software products. Testing techniques can be categorized into structural coupling metrics and textual coupling metrics. Structural metrics analyze the static code structure, while textual metrics focus on the semantic relationships between classes.

Software Testing Techniques

Software testing techniques have evolved to encompass a wide range of approaches, broadly categorized into structural-based, functionality-based, and hybrid methods. Structural-based techniques, also known as white-box testing, focus on examining the internal logic and structure of the code. These methods include control flow testing, data flow analysis, and various coverage criteria such as statement, branch, and path coverage (Ammann & Offutt, 2016). While effective in uncovering implementation errors, structural-based techniques often struggle with scalability in large systems and may miss errors in specifications or user requirements. On the other hand, functionality-based or black-box testing techniques concentrate on the software's behavior without considering its internal structure. These methods, including equivalence partitioning, boundary value analysis, and use case testing, are derived from system specifications and requirements. Although effective in validating system functionality, these approaches may overlook certain logical errors or exceptions not apparent from the specifications alone.

Recognizing the limitations of purely structural or functional approaches, hybrid techniques have emerged to provide more comprehensive test coverage. These methods, such as gray-box testing and model-based testing, combine elements from both structural and functional testing paradigms (Utting et al., 2012). While offering a more holistic approach to testing, hybrid techniques can be complex to implement and often require significant resources and expertise.

In recent years, AI-driven testing approaches have gained traction, leveraging ML and data analytics to enhance various aspects of software testing. These techniques include AI-based test case generation, intelligent test prioritization, and predictive analytics for defect detection (Durelli et al., 2019). While promising, AI-driven approaches are still evolving and may require large amounts of historical data to be effective.

Our proposed CSTCC metric and AI-driven approach aims to address the gaps in these existing techniques. By integrating structural coupling information with semantic analysis through LSI, we offer a more comprehensive assessment of class coupling that captures both explicit dependencies and implicit semantic relationships. This approach aligns with the growing recognition of the importance of semantic information in software engineering, as highlighted by Poshyvanyk et al. (2009).

Furthermore, our use of ML algorithms to predict critical classes for integration testing represents a novel application of AI in software testing. This dynamic and adaptive solution addresses the scalability issues often encountered in structural-based testing of large systems while providing more targeted insights than traditional black-box methods. By focusing specifically on optimizing integration testing, an area where existing techniques often fall short, our approach aims to significantly improve testing efficiency and effectiveness in object-oriented systems.

In essence, our CSTCC metric and AI-driven methodology synthesize the strengths of structural, functional, and AI-based testing approaches while mitigating their individual limitations. This holistic approach aligns with the current trend towards more intelligent and adaptive testing strategies in software engineering (Garousi & Mäntylä, 2016), offering a promising solution to the challenges of integration testing in complex, evolving software systems.

Software metrics

Software metrics are quantitative measures of specific features within software that serve as valuable tools for monitoring development progress and evaluating the quality of software products (Goel & Gupta, 2017). Among the well-known metrics, coupling and cohesion are extensively used to gauge dependencies (Revelle et al., 2011). Coupling assesses dependencies between different modules or classes, while cohesion measures dependencies within an individual module (Alenezi, 2014). Modules are considered highly coupled when they exhibit significant connections, and ideally, software development aims for low coupling and high cohesion (Bidve & Khare, 2012). In this study, coupling measures are employed for test focusing, as high coupling between modules increases the likelihood of faults spreading from one module to another, and faults are often detected during integration testing, where couplings typically occur (Magel, 2017). Many techniques have been presented to capture coupling according to different types of information, such as structural and textual. This section mentions these techniques and distributes them into two groups as follows:

Structural coupling software metrics

The coupling measure will be at the class level, so we need to investigate class-level metrics in the literature to extract the best metric. Chidamber and Kemerer (1994) introduced the CK metric suite, which offers valuable insights into whether developers adhere to OO design principles. They argue that utilizing a combination of these metrics empowers managers and designers to make more informed decisions about design. The CK metrics have gained considerable attention and currently stand as the most well-known set of measurements for OO software systems. Proposed by Chidamber and Kemerer (1994), the suite comprises six metrics:

Weighted methods per class (WMC): The sum of the complexities of each method within an individual class.

Depth of inheritance tree (DIT): The maximum length from the class node to the root of the tree.

Number of children (NOC): The count of immediate subclasses of a class in the hierarchy.

Coupling between object classes (CBO): The count of the number of related couplings with other classes.

Response for a class (RFC): The count of a class's methods plus methods that are directly or indirectly invoked by those methods.

Lack of cohesion in methods (LCOM): Measures the dissimilarities between methods in a class by examining the instance variables or attributes used by methods.

The CK metrics have garnered considerable attention in the literature, offering a comprehensive and categorized framework for coupling measurement. Among the CK metrics, CBO, RFC, *message passing coupling* (MPC), and *data abstraction coupling* (DAC) are widely recognized. These metrics, along with Li and Henry metrics, play a significant role in assessing software quality attributes, particularly testability, and enhancing fault detection capabilities. Asad and Alsmadi (2014) evaluated the impact of software metrics on defect prediction using open-source code and case study results. The results showed that some coupling metrics, such as CBO, RFC, and MPC, have a significant correlation with the prediction of bugs. We depend on the MPC metric in our work because MPC plays an important role in fault-proneness detection. MPC can discover which methods in a class send messages to another class, helping provide an accurate result because it discards methods that are not invoked by other classes.

Textual coupling software metrics

Some studies interchangeably use terms like 'semantic,' 'conceptual,' or 'textual' to describe similar concepts. Textual coupling, in particular, has been employed in various studies to support software tasks such as change impact analysis, software quality assessment, coupling analysis, fault proneness prediction, and modularization. For instance, Poshyvanyk et al. (2009) introduced a novel coupling metric for OO software known as conceptual coupling. This metric relies on semantic information extracted from the source code, including identifiers and comments. Case studies on open-source software systems demonstrated that the new metric captures dimensions of coupling not addressed by existing structural coupling metrics. In a related effort, Újházi et al. (2010) proposed two conceptual metrics for measuring coupling and cohesion in OO applications. The first, *conceptual coupling between object* (CCBO) classes, is based on the well-known CBO coupling metric. The second, *conceptual lack of cohesion on methods* (CLCOM5), relies on the LCOM5 cohesion metric. Empirical investigations were conducted to predict the fault proneness of classes in a large open-source system. The results revealed that the proposed conceptual metrics, when used together, could predict faults nearly as effectively as the structural metrics in the Columbus source code quality framework. Combining these conceptual metrics with structural metrics optimized fault prediction outcomes.

Gethers and Poshyvanyk (2010) introduced a novel coupling metric for measuring OO software, termed *relational topic-based coupling* (RTC) of classes. This metric utilizes *relational topic models* (RTM) to capture encoded information within source code classes and the relationships between them. An experimental study conducted on thirteen open-source programs compared the new metric (RTC) with existing structural and textual coupling metrics. The results of the study indicated that the proposed metric not only captured novel dimensions of coupling not addressed by existing metrics but also proved effective in optimizing impact analysis. Additionally, researchers explored combining textual coupling with structural coupling to create a metric that considers both aspects simultaneously. Alenezi (2014) defined four coupling metrics, each building on the previous one in a manner similar to Poshyvanyk et al. (2009): *method pair coupling* (MPC), *hybrid coupling between method and a class* (HCMC), *hybrid coupling between two classes* (HCCC), and *coupling of a class in an OO system* (SSCM). An evaluation was conducted, demonstrating that these metrics capture aspects not covered by structural and semantic coupling relations when considered separately.

In our work, we focus on capturing semantic information between methods in classes and calculating the similarity between these methods based on the LSI technique, akin to the approach used by Gethers and Poshyvanyk (2010).

AI-Driven Software Testing

AI techniques have been increasingly applied to various aspects of software engineering, including software testing and coupling measurement. These techniques leverage ML, natural language processing, and information retrieval methods to extract valuable insights from software artifacts and improve the efficiency and effectiveness of software development processes.

In the context of coupling measurement, AI techniques have been employed to capture semantic relationships between software components that may not be evident from structural analysis alone. Gethers and Poshyvanyk (2010) introduced a novel coupling metric called RTC that utilizes RTM to capture encoded information within source code classes and their relationships. RTM is a probabilistic model that combines statistical topic modeling with relational data, allowing for the discovery of latent topics and their associations with software entities. The experimental study conducted by Gethers and Poshyvanyk (2010) demonstrated that RTC captures novel dimensions of coupling not addressed by existing structural and textual metrics and proves effective in optimizing impact analysis. Similarly, Újházi et al. (2010) proposed two conceptual coupling metrics, CCBO and CLCOM5, which leverage semantic information extracted from source code identifiers and comments. These metrics employ information retrieval techniques, such as LSI, to measure the conceptual similarity between classes and methods (AlSobeh et al., 2018). The empirical investigation conducted by Újházi et al. (2010) revealed that the proposed conceptual metrics, when combined with structural metrics, can predict fault-proneness of classes with higher accuracy compared to using structural metrics alone.

AI techniques have been applied to optimize test case generation, prioritization, and selection. Grechanik and DevanlaG (2016) developed an approach called *Java mutation integration testing* (jMINT) that generates mutants for the system and provides a solution for these mutants using integration test suites. They employed static analysis and mutation operators to identify interactions between components and generate mutants in the data flow path. The experimental evaluation on five open-source Java applications demonstrated that jMINT significantly reduces the number of mutants and enhances the ability to find integration test suites. Yang et al. (2020) employed reinforcement learning with a reward strategy to prioritize test cases based on their fault detection capability and execution time. The experimental results on four programs showed an improvement in the flexibility and efficiency of fault detection compared to traditional test case prioritization techniques. Li et al. (2020) utilized the Sapient framework, which combines ML and static analysis, for implementing continuous integration testing based on class-level and method-level test case selection. The Sapient framework learns from historical test execution data and source code changes to predict the likelihood of a test case detecting faults in a given class or method. The empirical evaluation on eighteen Eclipse and Apache open-source projects demonstrated that the Sapient framework improves the efficiency of fault detection and can be applied to large-scale software systems.

These studies highlight the potential of AI techniques in enhancing coupling measurement and software testing practices. By leveraging ML, natural language processing, and information retrieval methods, researchers have developed novel metrics and approaches that capture semantic relationships between software components and optimize test case generation, prioritization, and selection. The integration of AI techniques with traditional structural and textual analysis methods has shown promising results in improving the accuracy of fault prediction, impact analysis, and test suite optimization. However, the application of AI techniques in software engineering is still an emerging area, and there are challenges and limitations that need to be addressed. The effectiveness of AI-driven approaches heavily relies on the quality and quantity of training data, which may not always be readily available or representative of real-world scenarios (AlSobeh and Magableh, 2018). Moreover, the interpretability and explainability of AI models can be limited, making it difficult for developers to understand and trust the recommendations provided by these models. So, future research should focus on developing more robust and transparent AI techniques that can handle the complexity and diversity of software systems, while providing actionable insights to developers and testers.

Li et al. (2022) and Jorayeva et al. (2022) conducted studies on deep learning-based prediction of software defects. Their work reveals the growing importance of AI techniques in identifying potential issues in software systems. They review various deep learning models and their applications in defect prediction, providing a foundation for understanding how AI can enhance software quality assurance processes.

In the context of test case prioritization, Tan et al. (2024), Sharif et al. (2021) and Lu et al. (2022) proposed a novel approach using multi-objective reinforcement learning for continuous integration environments. Their method aims to optimize multiple criteria simultaneously, such as fault detection capability and execution time. This research demonstrates the potential of advanced AI techniques in addressing complex testing scenarios, aligning with our CSTCC approach's goal of efficient and effective integration testing.

Giray (2021) offered a perspective on the application of machine learning in software testing. Their work discusses various aspects of ML-driven testing, including test case generation, test oracle problem, and test suite reduction. This perspective provides valuable insights into the current state and future directions of AI in software testing, contextualizing our CSTCC approach within the broader landscape of AI-driven testing methodologies.

The study by Pargaonkar (2022) and Amalfitano et al. (2023) on AI-based test case generation highlights recent advancements in using AI for creating effective test suites. They reviewed various techniques, including those based on natural language processing and graph neural networks, which illustrated the diverse ways in which AI is being applied to software testing, providing a comparative basis for our CSTCC approach. Moreover, their work highlights the unique challenges posed by highly distributed and decoupled systems, emphasizing the need for advanced testing methodologies. This review provides valuable insights into how approaches like CSTCC can be adapted or applied to contemporary software architectures.

These recent studies collectively demonstrate the rapid evolution of AI-driven approaches in software testing and coupling analysis. They highlight the increasing sophistication of techniques used to address complex testing scenarios, improve efficiency, and enhance fault detection capabilities. Our CSTCC approach builds upon these advancements, offering a unique combination of structural and semantic coupling analysis powered by AI techniques. By integrating insights from these recent works, CSTCC aims to provide a comprehensive and adaptable solution to the challenges of modern software integration testing.

Reviewing the previous literature, we can conclude the following main points: Our approach does not need to understand the overall system because it deals with the source code as a black box. In addition, we do not need to use *unified modeling language* (UML) diagrams like state diagrams or object diagrams, unlike Ali et al. (2007) and Li et al. (2020), who need to use these diagrams. Liu and Chen (2014) and Briand et al. (2002) relied on the coupling concept to measure dependencies between two classes based on attributes such as

(1) the number of distinct variables used,

(2) the number of distinct methods called,

(3) the number of parameters sent, and

(4) the number of return value types.

In contrast, our approach is based on CSTCC. Our method can result in a more significant reduction in time and cost because we ask developers to write fewer test cases. In contrast, other methods require developers to create a complete set of test cases, which are prioritized, selected, or removed later. Stubs and drivers are typically needed to simulate the functionalities of classes that have not yet been developed. These approaches focus on minimizing the cost of integration testing by reducing the number of stubs or overall stub complexity, as seen in Liu and Chen (2014) and Briand et al. (2002). In our study, we operate under the assumption that all classes within the system under examination are accessible throughout the testing procedure, eliminating the need for stubs and drivers in our case. Structural coupling metrics and textual measures are used separately in the literature to extract the dependencies between classes and order classes (Alazzam, 2012;

AlSobeh & Shatnawi, 2023). Our approach is different from these approaches. It combines structural coupling metrics and textual measures to extract the dependencies between classes and order classes. The integration of AI techniques with structural and textual coupling metrics, as demonstrated by the CSTCC metric proposed in this study, represents a promising direction for enhancing software testing practices. By leveraging the power of AI to capture semantic relationships and guide test case generation and prioritization, we can improve the efficiency and effectiveness of integration testing and deliver higher-quality software products.

Class Integration Testing Approaches

Integration testing plays a pivotal role in the realm of OO software development, given the intricate relationships among software classes involving method calls, aggregation, and inheritance. The challenge arises in determining a suitable integration order, particularly in the presence of cyclic dependencies, referred to as the *class integration and test order* (CITO) problem (Liu & Chen, 2014). Researchers have explored the use of UML models in integration testing. For instance, Ali et al. (2007) proposed a testing approach based on collaboration diagrams and state charts to unveil state-dependent interaction errors. They emphasized exercising interactions among objects for every possible state of included objects. Furthermore, some researchers have adopted a graph-based strategy for ordering classes.

Alazzam (2012) employed an information retrieval technique known as LSI as a proxy to assess the dependency among methods. Similar to Taan et al. (2017), the summation of weights for all methods associated with two classes is computed to derive the class-pair weight. This weight is instrumental in establishing the dependency between classes and arranging them in a specific order. Grechanik and DevanlaG (2016) proposed an approach to enhancing the effectiveness of integration testing. They created jMINT to generate mutants for the system and provide a solution for these mutants by using integration test suites. They developed a model for integration bugs and performed an analysis of how components interact with each other in the application. They generate the integrated mutants with the help of mutation operators in the data flow path. By using a static analyzer, they identify the interaction between different components in the application. They implemented their approach on five open-source Java applications. In mu-java, their solution reduced 19-time mutants and strengthened the ability to find integration test suites. Trautsch et al. (2020) have analyzed the validity of integration and unit testing in modern software methodology based on data from Java open-source projects. They classified thousands of test cases and used mutation testing to evaluate the potential for detecting defects and bugs. They concluded that there are no statistically significant differences between the two types of testing.

In summary, while existing approaches have made significant contributions to software testing, several gaps remain. These include the lack of integration between structural and semantic coupling information, limited use of AI techniques in predicting critical classes, and insufficient focus on optimizing integration testing specifically. Our proposed CSTCC metric and AI-driven approach aims to address these gaps, where previous studies showed that high coupling correlates with fault-proneness, justifying the focus on these metrics to improve test case prioritization and integration testing efficiency.

DEVELOPING COMBINED STRUCTURAL AND TEXTUAL CLASS COUPLING: CSTCC MODEL

The proposed research methodology consists of four main phases, as depicted in **Figure 1**. These phases include:

1) extracting coupling metrics,
2) computing CSTCC,
3) performing integration testing, and
4) employing mutation testing to assess the effectiveness of the proposed approach.

Each step is described in the subsections below.

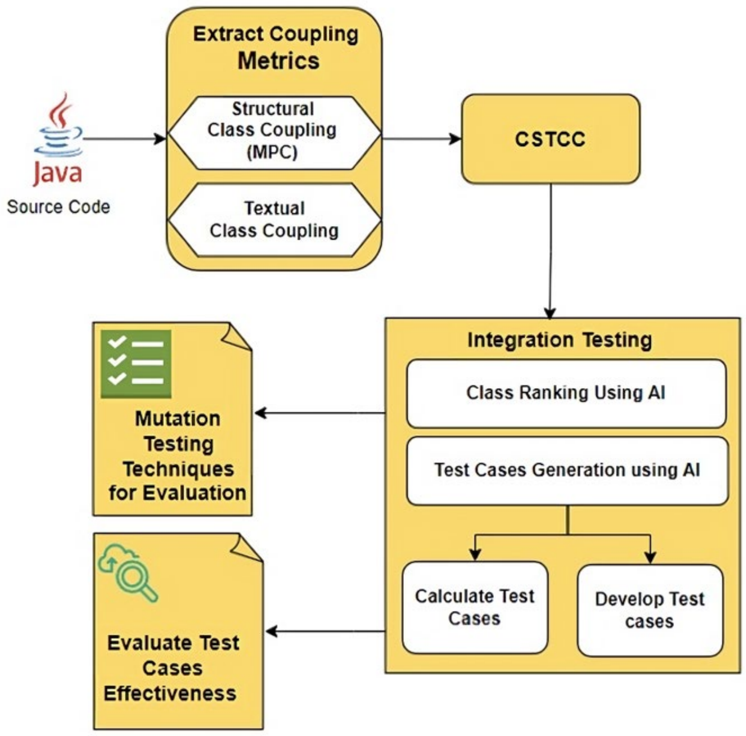


Figure 1. Research phases model (Source: Authors)

Table 1. Data set

Data set software	Number of classes	Number of methods
Flight booking	9	77
Car rental	7	54
Virtual machine	13	127
Monopoly	36	229

Data Collection and Coupling Metrics Extraction

For the empirical evaluation, we used four Java-based open-source systems. **Table 1** shows the dataset and its characteristics, including the number of classes and methods for each system. To ensure a comprehensive and robust evaluation of the proposed AI-driven approach, a diverse set of OO software systems implemented in Java should be selected. These systems should exhibit varying sizes, complexities, and application domains to assess the generalizability of the findings. The selection process should consider factors such as the number of classes, methods, and lines of code, as well as the availability of source code and any accompanying documentation or design artifacts. Collecting a rich and representative dataset is crucial for training and validating the AI models effectively. To ensure a comprehensive and robust evaluation of the proposed AI-driven approach, a diverse set of OO software systems implemented in Java was selected. These systems exhibit varying sizes, complexities, and application domains to assess the generalizability of the findings. The selection process considered factors such as the number of classes, methods, and lines of code, as well as the availability of source code and any accompanying documentation or design artifacts. We used Java-based open-source systems for several reasons: Open-source software systems are publicly available, allowing for unrestricted analysis of the code. Java is a widely used OO language, making our findings relevant to a large portion of the software development community. The selected applications vary in size and complexity, ranging from 7 to 36 classes, providing a good spectrum for evaluating our approach. These systems represent different application domains (e.g., flight booking, car rental, virtual machine, board game), allowing us to test the versatility of our approach.

These artifacts provide valuable context and insights into the structure and behavior of the classes within the systems. If the source code is not readily available, efforts should be made to contact the system owners or developers to obtain the necessary permissions and access.

Extract coupling metrics

To capture the structural and textual characteristics of the classes in the selected software systems, automated tools, and techniques should be employed for feature extraction. Structural features encompass various metrics that quantify the dependencies and interactions between classes, such as MPC, DIT, NOC, and others. These metrics can be calculated by analyzing the static code structure and counting the number of method invocations, inheritance relationships, and other relevant attributes. To specify a set of metrics and extract them automatically from the Java source code. We focus on two types of coupling metrics: MPC metric and the *textual class coupling* (TCC).

MPC is defined as the number of sending statements defined in a class. MPC measures the dependency of a class's method implementations on methods in other classes. It only counts invocations of methods from other classes, providing more accurate results by discarding invocations of methods within the same class. MPC has significant importance in measuring coupling between classes and has been shown to have a high correlation with fault-proneness detection and software quality attributes such as testability and maintainability (Briand et al., 2002).

TCC calculates the similarity between two or more classes is computed by summing the similarity values of all the methods related to these classes, as shown in the following equation (Poshyvanyk & Marcus, 2006):

$$TCC = \frac{\sum cosine\ similarity\ of\ methods}{\#e(m)}$$

where $cosine\ similarity\ of\ methods$ represents the similarity values between two methods in two classes, and $\#e(m)$ is the number of methods in the same class that have similarity with other methods.

Computing CSTTC: Structural Coupling (MPC) + Textual Coupling (TCC)

In this phase, we combine the MPC and TCC metrics to propose the CSTCC metric. The main idea behind this combination is to take advantage of the complementary relationship between structural and textual coupling. CSTCC can be used when one source of information cannot be completely relied on. We investigate the effectiveness of CSTCC in enhancing the integration testing process by studying the dependence relationships between classes and determining the classes that have high connections with other classes, which are presumed to be more error-prone. This would help detect as many integration faults as possible in integrated classes.

We use the weighted average principle to perform the combination. A weighted average is most often computed with respect to the importance of values in a dataset, where certain values are given more importance by multiplying each data point value by an assigned weight and then dividing by the number of data points (Ganti, 2019). In our work, we adopt the weighted average to reflect the class that has high significance within the group of classes. The MPC value is multiplied by TCC and then divided by the number of classes in the software system, as shown in the following equation:

$$CSTCC = \frac{MPC * TCC}{n}$$

where n is the total number of classes in the software system.

Integration Testing and AI Model Development

The core of the AI-driven approach lies in the development of sophisticated models capable of combining structural and textual features to predict the CSTCC metric. Various AI techniques can be explored, including ML algorithms, deep learning architectures, and ensemble methods.

ML algorithms, such as *random forests*, *support vector machines* (SVM), or *gradient boosting machines* (GBM), are employed to learn the complex relationships between the structural and textual features and the CSTCC metric. They are trained using supervised learning techniques, where the input features are mapped to the corresponding CSTCC values. To evaluate the effectiveness of the CSTCC metric in predicting critical classes, we implemented and compared three ML algorithms: Random forests, SVM, and GBM. The dataset used for this analysis included features aligned with structural and textual coupling metrics, which are crucial for the CSTCC metric.

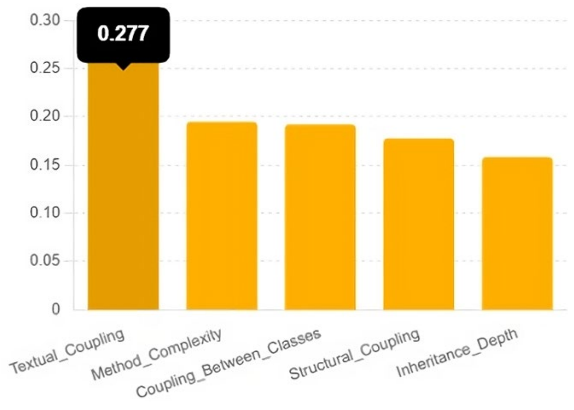


Figure 2. Random forest feature importances (Source: Authors)

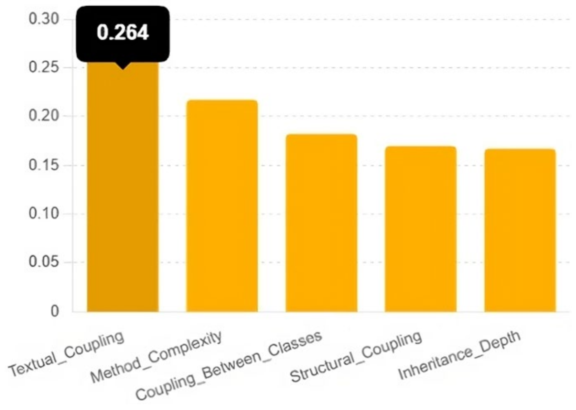


Figure 3. Gradient boosting feature importances (Source: Authors)

Ensemble methods, which combine the predictions of multiple individual models, can be employed to improve the robustness and accuracy of the CSTCC predictions. Techniques like bagging, boosting, or stacking can be used to create diverse sets of models that complement each other's strengths and compensate for their weaknesses.

AI model development

The AI models are trained using the extracted features and appropriate performance metrics, such as mean squared error (MSE), mean absolute error (MAE), or correlation coefficients. The training process involves splitting the dataset into training, validation, and testing subsets, allowing for model selection, hyperparameter tuning, and unbiased evaluation. Cross-validation or hold-out validation techniques can be used to assess the generalization performance of the models and prevent overfitting.

As mentioned above, in the context of predicting critical classes based on the CSTCC metric, the classes are defined as follows:

Non-critical class, which represents software components or modules that are not considered critical in terms of their impact on the overall system's performance, maintainability, or reliability. These components typically have lower coupling and higher cohesion, indicating that they are well-structured and have fewer dependencies on other components. The metrics used to predict this class include structural coupling, textual coupling, inheritance depth, method complexity, and coupling between classes.

Critical class, which represents software components or modules that are considered critical due to their significant impact on the system's performance, maintainability, or reliability. These components typically have higher coupling and lower cohesion, making them more complex and interdependent on other components. Identifying critical classes is crucial for prioritizing testing and maintenance efforts, as issues in these components can have widespread effects on the system. The same set of metrics (structural coupling, textual coupling, inheritance depth, method complexity, and coupling between classes) are used to predict this class.

Figure 2 and **Figure 3** show the feature selection as recursive feature elimination or L1 regularization, which be applied to identify the most informative and relevant features for predicting the CSTCC metric. This reduced the dimensionality of the feature space, improved model interpretability, and mitigated the risk of spurious correlations. These results highlight that Textual_Coupling is the most significant feature in predicting critical classes, followed by Method_Complexity and Coupling_Between_Classes.

The performance of these algorithms was assessed based on accuracy, precision, recall, and F1-score, **Figure 4** shows the performance metrics and confusion matrices indicate that SVM has better precision and recall for the non-critical class, whereas gradient boosting provides a balanced approach for both classes. The SVM achieved the highest overall accuracy at 63%, followed by gradient boosting at 60%, and random forest at 58%. For the non-critical class, the SVM shows superior performance in terms of recall and F1-score. For the Critical class, the SVM also leads in precision, while gradient boosting provides a balanced approach in

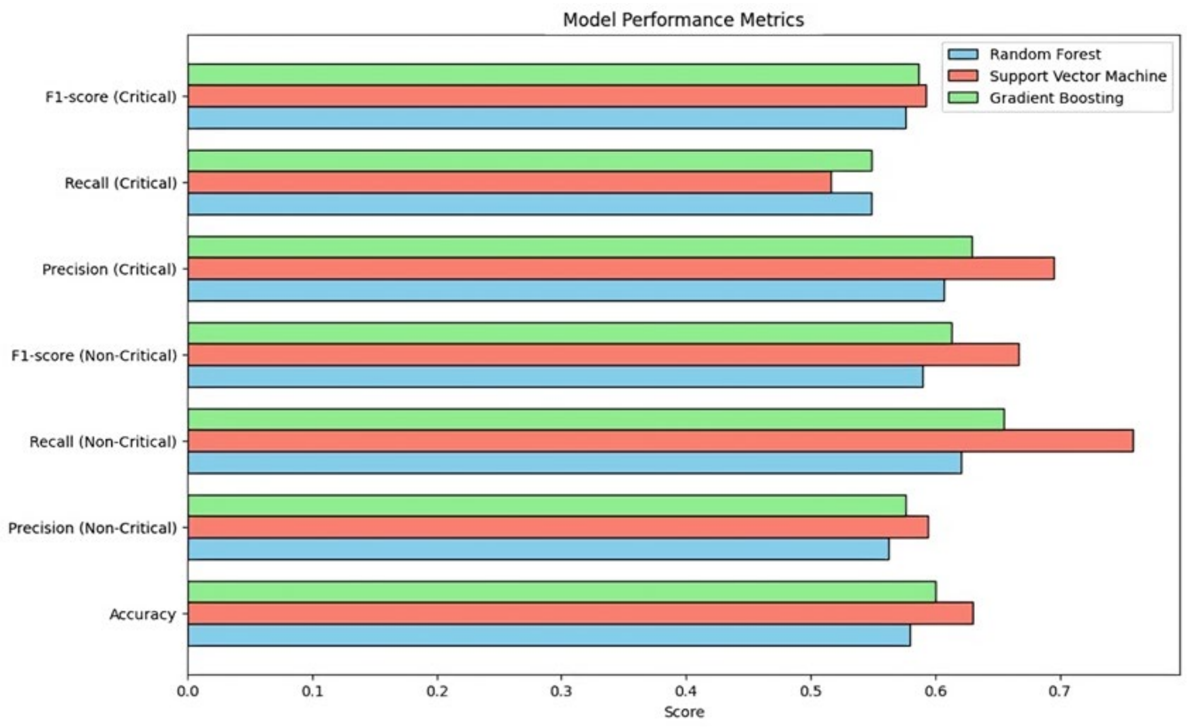


Figure 4. Random forest, SVM, and gradient boosting performance (Source: Authors)

precision and recall. These results validate the effectiveness of the CSTCC metric and highlight the importance of various structural and textual features in predicting critical classes, thus supporting the proposed AI-driven approach in software integration testing.

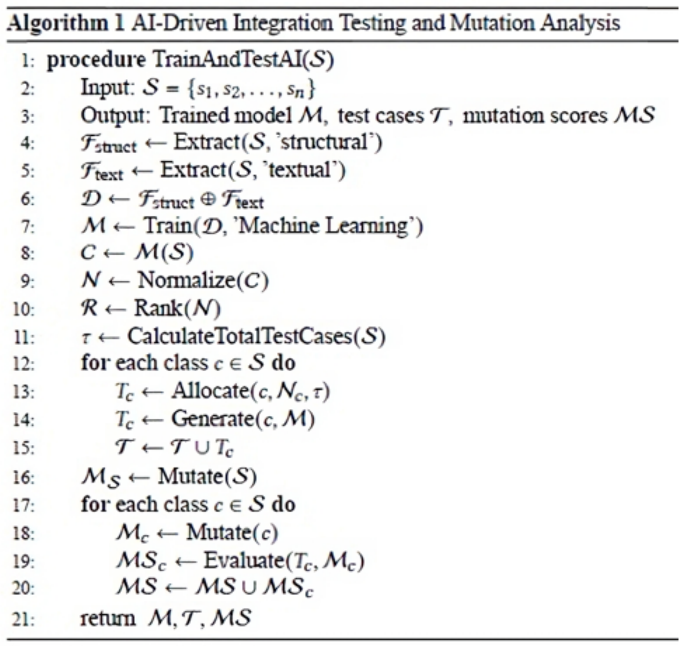
LSI is applied to the textual content of classes, including comments and identifiers. We use singular value decomposition to reduce the dimensionality of the term-document matrix, capturing the latent semantic structure of the text. The resulting semantic vectors are then used as features in our CSTCC calculation and subsequent ML model.

To calculate the CSTCC metric, consider the predicted values from multiple models and combine them using weighted averaging or voting mechanisms. The weights assigned to each model can be determined based on their performance or through optimization techniques that maximize the overall predictive accuracy. To treat the CSTCC metric as a continuous value and directly use the predicted values from the AI models. In this case, the distribution and characteristics of the CSTCC metric across different systems and classes should be analyzed to gain insights into its behavior and potential thresholds for identifying highly coupled classes.

Performing integration testing

Algorithm 1 for integration testing assumes the following: A set of classes is exported from unit testing and available for integration testing, eliminating the need for stubs or drivers. However, developers often face time constraints during the integration testing phase. To address this challenge and optimize the testing process, Algorithm 1 leverages the CSTCC metric to guide the allocation of test cases and prioritize testing efforts.

Figure 5 shows Algorithm 1, named "TrainAndTestAI," offers a comprehensive approach to AI-driven integration testing and mutation analysis for a given set of software systems *S*, where *S* = {s_1, s_2, ..., s_n}. The primary objectives of this algorithm are to produce a trained AI model *M*, a set of test cases *T*, and a set of mutation scores *MS*. The process begins by extracting structural features *F_struct* and textual features *F_text* from the software systems *S* using the "Extract" function, which takes the software systems *S* and a parameter indicating the type of features to be extracted ('structural' or 'textual'). Subsequently, the extracted structural features *F_struct* and textual features *F_text* are combined using the direct sum operator ⊕ to form a dataset *D*, which is then utilized to train an AI model *M* using a deep learning approach through the "Train" function,



```
Algorithm 1 AI-Driven Integration Testing and Mutation Analysis
1: procedure TrainAndTestAI($\mathcal{S}$)
2:     Input: $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$
3:     Output: Trained model $\mathcal{M}$, test cases $\mathcal{T}$, mutation scores $\mathcal{MS}$
4:     $\mathcal{F}_{\text{struct}} \leftarrow$ Extract($\mathcal{S}$, 'structural')
5:     $\mathcal{F}_{\text{text}} \leftarrow$ Extract($\mathcal{S}$, 'textual')
6:     $\mathcal{D} \leftarrow \mathcal{F}_{\text{struct}} \oplus \mathcal{F}_{\text{text}}$
7:     $\mathcal{M} \leftarrow$ Train($\mathcal{D}$, 'Machine Learning')
8:     $C \leftarrow \mathcal{M}(\mathcal{S})$
9:     $\mathcal{N} \leftarrow$ Normalize($C$)
10:    $\mathcal{R} \leftarrow$ Rank($\mathcal{N}$)
11:    $\tau \leftarrow$ CalculateTotalTestCases($\mathcal{S}$)
12:    for each class $c \in \mathcal{S}$ do
13:        $T_c \leftarrow$ Allocate($c, \mathcal{N}_c, \tau$)
14:        $T_c \leftarrow$ Generate($c, \mathcal{M}$)
15:        $\mathcal{T} \leftarrow \mathcal{T} \cup T_c$
16:    $\mathcal{M}_{\mathcal{S}} \leftarrow$ Mutate($\mathcal{S}$)
17:    for each class $c \in \mathcal{S}$ do
18:        $\mathcal{M}_c \leftarrow$ Mutate($c$)
19:        $\mathcal{MS}_c \leftarrow$ Evaluate($T_c, \mathcal{M}_c$)
20:        $\mathcal{MS} \leftarrow \mathcal{MS} \cup \mathcal{MS}_c$
21:    return $\mathcal{M}, \mathcal{T}, \mathcal{MS}$
```

Figure 5. Algorithm 1: AI-driven integration testing and mutation analysis (Source: Authors)

taking the dataset *D* and the parameter 'Deep Learning' to specify the training method. The trained AI model *M* is then applied to the software systems *S* to obtain a set of coupling measures *C*, which are normalized using the "Normalize" function to obtain a set of normalized coupling measures *N*. These normalized coupling measures *N* are employed to rank the classes in *S* using the "Rank" function, resulting in *R*. Additionally, the total number of test cases *τ* (tau) is calculated using the "CalculateTotalTestCases" function, which takes the *S* as input. The algorithm then proceeds to iterate over each *c* in *S*, allocating test cases *T_c* for each class *c* based on its normalized coupling measure *N_c* and the total number of test cases *τ* using the "Allocate" function and generating *T_c* for each class *c* using the trained AI model *M* through the "Generate" function, with the generated test cases *T_c* being added to the overall set of *T*. Furthermore, a set of mutants *M_S* is generated from the *S* using the "Mutate" function, and the algorithm iterates over each class *c* in *S*, generating *M_c* for each class *c* using the "Mutate" function, and evaluating the generated test cases *T_c* against the *M_c* for each class *c* using the "Evaluate" function, resulting in a mutation score *MS_c*, which is added to the overall set of *MS*. Finally, the algorithm returns the trained AI model *M*, the generated *T*, and the *MS*, providing a systematic approach to leveraging AI for integration testing and mutation analysis in software systems.

Figure 6 shows The LSI components (LSI1, LSI2, ..., LSI10) represent different dimensions of the latent semantic structure captured from the textual data. Each LSI component is a linear combination of the terms in the document, designed to capture the most significant patterns in the term-document matrix. The inclusion of LSI features aims to capture latent semantic information from class-related textual data (e.g., comments, identifiers). The feature importance analysis indicates that while traditional features like Structural_Coupling, Inheritance_Depth, and Coupling_Between_Classes are highly influential, the LSI features also play a role, albeit a smaller one. The LSI features' ability to capture semantic nuances can potentially provide additional insights into the criticality of classes. For example, classes with similar structural characteristics but different semantic content might be distinguished better with LSI features included. However, the relatively lower importance of LSI features in this study highlights the need for high-quality, diverse textual data to fully leverage the benefits of latent semantic analysis. Improving the richness and variability of textual data, optimizing the LSI parameters, and combining LSI with other advanced text representation techniques (e.g., word embeddings) could enhance the predictive power of our models in future research.

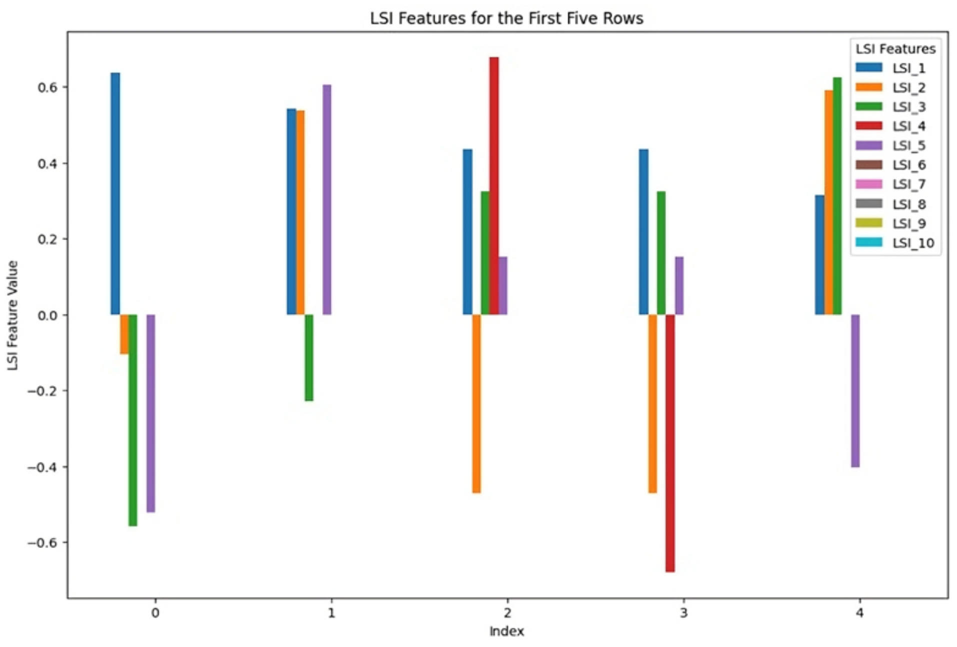


Figure 6. LSI features for the first 5 rows (Source: Authors)

The incorporation of LSI features adds a layer of semantic understanding to our model, but their effectiveness is contingent upon the quality of the textual data. However, the effectiveness of these features is still limited by the quality and diversity of the textual data used. In this study, while the LSI features provide some additional value (as seen by their non-zero importance), the overall contribution is less significant compared to traditional structural and textual features. This suggests that further improvement in textual data quality or additional text preprocessing might enhance the utility of LSI features.

Class ranking using AI

In this step, we used the CSTCC metric to determine the classes that have high connections with others and ranked them from high connection to low connection. The AI model is trained on historical data to learn the patterns and relationships between the CSTCC metric and the likelihood of integration errors. The class ranking process involves the following steps:

(1) calculate the CSTCC value for each class using the trained AI model,

(2) normalize the CSTCC values by dividing each class's CSTCC value by the sum of all CSTCC values, and

(3) rank the classes in descending order based on their normalized CSTCC values.

The AI-based class ranking ensures that classes with higher coupling and a greater likelihood of integration errors receive higher priority during testing.

Test case generation using AI

Once the classes are ranked, the next step is to generate test cases using AI techniques. The AI model utilizes the CSTCC metric and other relevant features to determine the optimal number of test cases for each class.

Test cases are created manually using the Eclipse tool with the installed JUnit plugin to test the required class in our application. For evaluating our approach, error seeding technique using Mutation testing is employed, wherein the test case generation phase involves a comparison between the original program and the mutant program. If the test cases can distinguish between the original program and the mutant program, the mutant program is considered "killed"; otherwise, it remains "alive". We used mutation testing to assess the effectiveness of test cases. Mutation testing involves introducing small, intentional changes (mutations) to the source code and then running the test suite against these mutated versions. If a mutation is not detected by the test suite (i.e., the tests still pass), it indicates a weakness in the test suite's ability to detect faults. Therefore, by measuring the percentage of mutations detected, you can gauge the effectiveness of the test suite in identifying potential bugs.

This phase contains two main steps:

(1) calculated test cases, and

(2) developed test cases.

To calculate the total count of test cases that are allocated for the whole application by using an arbitrary value which is 5 and multiplying it by the total number of integrated classes, as represented in the following equation:

$$T = r * n.$$

Let T represent the overall count of test cases allocated for the entire application, with r denoting an arbitrary value, and n being the count of integrated classes. The AI model predicts the number of test cases required for each class based on its normalized CSTCC value. Subsequently, we determined the count of test cases designated for each class by multiplying the normalized CSTCC by the total count of test cases from the preceding step, as expressed in the following equation:

$$\# \, of \; test \; cases \; that \; shold \; go \; for \; each \; class = T * Normalized \; CSTCC$$

where T is the total count of test cases allocated for the whole application from the previous step and $Normalized \; CSTCC$ is normalized CSTCC for each class in the application.

Developing test cases

The AI model identifies the methods within each class that have high interactions with other classes based on the CSTCC metric. It considers the number of method callers and callees to determine the critical methods for testing. The test cases are then developed manually using the Eclipse tool with the JUnit plugin, focusing on the identified critical methods.

To develop the test cases, we focus on the methods in the class. After determining the classes that have high interactions with other classes based on CSTCC, we need to determine which method to start testing in that class. The method that should be started is the one that has high interaction with other classes, based on the number of method callers and callees. The caller refers to the methods that call the selected method from other classes, while the callee refers to the methods that are called from the selected method in other classes. We use Eclipse to calculate the method callers and callees by right-clicking on a method and choosing "open call hierarchy."

Mutation testing techniques for evaluation to validate the effectiveness of the AI-driven approach, we employ mutation testing techniques. Mutation testing involves injecting artificial faults (mutants) into the code and evaluating the ability of the generated test cases to detect and kill these mutants. The mutants are designed to closely resemble real faults, providing a reliable assessment of the test cases' fault-finding effectiveness. The mutation testing process involves the following steps:

(1) generate mutants by applying mutation operators to the original code,

(2) execute the generated test cases against the mutants, and

(3) calculate the mutation score by dividing the number of killed mutants by the total number of mutants.

The effectiveness of the AI-generated test cases is evaluated using the mutation testing results. A high mutation score indicates that the test cases are effective in detecting and killing the injected mutants, suggesting their ability to uncover real faults. Therefore, we start by creating one test case for the method that has the highest number of method callers and callees in the class with high interaction with other classes. We then run the test cases against the mutants of each class and compute the mutation score for the class. We continue to develop test cases to achieve a high mutation score (e.g., 100%) or until the count of test cases allocated for the class is consumed. The test cases are developed manually using Eclipse for all applications. To validate the efficacy of our proposed approach, we employ mutation testing, which has been established as a reliable method for evaluating the fault-finding effectiveness of test cases. The mutants generated closely resemble real faults. In this phase, we apply mutation testing to the selected Java applications and assess the developed test cases' ability to kill the injected mutants. The mutation score is determined by dividing the count of killed mutants by the overall number of mutants.

RESULTS AND DISCUSSION

To evaluate the effectiveness of the AI-driven approach for integration testing, we applied the proposed methodology to four Java-based open-source systems: flight booking, car rental, virtual machine, and monopoly. **Table 2** provides an overview of the datasets used in this study, including the number of classes and methods in each system.

Table 2. Number of expected test cases

Software	Number of expected test cases
Flight booking	45
Car rental	35
Virtual machine	65
Monopoly	180

The AI model was trained on historical data to learn the patterns and relationships between the CSTCC metric and the likelihood of integration errors. The trained AI model was then used to calculate the CSTCC metric for each class in the four software systems.

CSTCC Results

Tables 3–6 present the values of MPC, similarity, and CSTCC for each class in the flight booking, car rental, virtual machine, and monopoly applications, respectively. The total CSTCC value is also provided, which is important for calculating the normalized CSTCC.

Table 3. Flight booking CSTCC results

Class name	MPC	Similarity	CSTCC	Description
Booking	4	0.527	0.234	Manages booking details for flights.
Customer	10	0.380	0.422	Handles customer information and interactions.
Passenger	4	0.474	0.210	Represents individual passengers in the system.
AirlineCompany	9	0.560	0.560	Manages airline company data and operations.
Airport	1	0.697	0.077	Stores airport-related information.
City	1	0.511	0.056	Contains city data relevant to flight operations.
Flight	6	0.211	0.140	Handles flight scheduling and details.
GenericFlight	12	0.421	0.561	Abstract class for flight-related operations.
StopoverInfo	15	0.521	0.868	Manages stopover details for connecting flights.
Total			3.130	

Table 4. Car rental CSTCC results

Class name	MPC	Similarity	CSTCC	Description
Car	3	0.368	0.158	Represents a general car in the rental system.
Payment	2	0.228	0.065	Manages the payment process for car rentals.
Reservation	0	0.465	0.000	Handles the booking and reservation details for renting a car.
SUV	5	0.373	0.266	A subclass of cars that represents sport utility vehicles.
Truck	4	0.407	0.233	A subclass of cars that represents trucks.
Vehicle2	6	0.484	0.415	Another type of vehicle class represents a different category or vehicle not covered by the Car class.
Vehicle2App	8	0.239	0.273	This class manages the overall application logic for handling the Vehicle2 type of vehicles.
Total			1.409	

Table 5. Virtual machine CSTCC results

Class name	MPC	Similarity	CSTCC	Description
AnaliseLexicaException	20	0.228	0.350	Handles exceptions that occur during lexical analysis.
AnaliseSemanticaException	16	0.383	0.471	Manages exceptions that arise during semantic analysis.
AnaliseSintaticaException	17	0.421	0.550	Deals with exceptions during syntactic analysis.
Comandos	0	0.111	0.000	Represents commands or instructions that the virtual machine can execute.
Compilador	22	0.550	0.930	Acts as the compiler class that orchestrates transforming source code into executable code.
GeradorDeCodigo	22	0.510	0.863	Handles the code generation phase.
Lexico	14	0.350	0.377	Manages lexical analysis.
Semantico	20	0.470	0.723	Conducts semantic analysis.
Simbolo	12	0.332	0.306	Represents a symbol in the symbol table.
Sintatico	15	0.369	0.426	Oversees syntactic analysis.
TabelaDeSimbolos	18	0.507	0.702	Maintains the symbol table.
Token	19	0.410	0.510	Represents a single token produced by the lexical analyzer.
Tipos	16	0.385	0.474	Manages the different types used in the language.
Total			6.680	

Table 6. Monopoly CSTCC results

Class name	MPC	Similarity	CSTCC	Description
AgentBanque	47	0.520	0.679	Manages the bank's operations.
AgentBDC	57	0.210	0.333	Represents the real estate agency operations.
AgentJoueur	70	0.521	1.013	Represents the player agent.
AgentMonopoly	29	0.637	0.513	The central controller of the game.
AgentPrison	26	0.655	0.473	Manages the prison operations.
AgentSeed	0	0.390	0.000	Responsible for generating and managing random seeds used for random events and decisions in the game.
BankSharkBehaviour	37	0.500	0.514	Represents aggressive financial behavior.
BDCBehaviour	36	0.650	0.650	Manages the behavior related to the real estate agency.
CreatePlateauBehaviour	42	0.600	0.700	Handles the creation and initialization of the game board.
DropDiceBehaviour	51	0.510	0.723	Manages the behavior of rolling dice.
GenerateIntBehaviour	48	0.221	0.294	Generates random integers.
GiveInitialCapital	55	0.166	0.254	Assigns initial capital to players at the start of the game.
GivePlayersToOthers	44	0.290	0.354	Handles the distribution of players to different agents or roles within the game.
JanitorJailBehaviour	38	0.310	0.327	Manages the behavior of the jail janitor.
OrdonnanceurBehaviour	54	0.320	0.480	Manages the scheduling and sequencing of player turns and actions.
AvideBehaviour	37	0.320	0.329	Represents greedy behavior.
CollectionneurBehaviour	41	0.540	0.615	Represents collector behavior.
EvilBehaviour	88	0.690	1.687	Represents malicious or harmful behavior.
IntelligentBehaviour	44	0.630	0.770	Represents strategic and well-thought-out behavior.
PassivePlayerBehaviour	27	0.451	0.339	Represents passive behavior.
PicsouBehaviour	36	0.570	0.570	Represents miserly behavior.
PlayerBehaviour	37	0.650	0.668	General class for defining player behavior, serving as a base for more specific behavior classes.
StupideBehaviour	0	0.177	0.000	Represents foolish or irrational behavior.
RecupInitialCapital	25	0.332	0.230	Handles the recovery or redistribution of initial capital among players.
CaseModel	44	0.114	0.139	
MainContainer	17	0.570	0.269	
Helper	10	0.222	0.061	
Logger	33	0.580	0.531	
Carte	0	0.520	0.000	
CaseAchetable	10	0.170	0.047	
CasePanel	39	0.210	0.227	
CaseTerrain	8	0.122	0.027	
Infos	15	0.390	0.163	
InfosPanel	40	0.240	0.267	
Monopoly	71	0.600	1.183	
Plateau	53	0.590	0.868	
Total			16.290	

Normalized CSTCC and Test Cases Calculations

Tables 7–10 show the results of normalized CSTCC, percentage of test cases, count of test cases, and class ranking for each class in the flight booking, car rental, virtual machine, and monopoly applications, respectively. The mutation testing results demonstrate the effectiveness of the AI-driven approach in generating test cases that can detect and kill a high percentage of mutants. The overall mutation scores ranged from 98.6% to 100% across the four systems, indicating the AI-generated test cases' strong fault detection capabilities. The class ranking determines which class has high connections with other classes. Classes with high connections are expected to have a high number of integration errors and thus require more focused testing during integration testing. We rank the classes from high to low connections using sequential numbering (1 to *n*), where *n* is the total number of classes. The number 1 refers to the class with the highest connection. The ranking is done based on the normalized CSTCC values, which are calculated by dividing the CSTCC for each class by the total CSTCC.

Table 7. Flight booking test cases calculations

Class name	Normalized CSTCC	Test cases percentage (%)	Number of test cases	Class ranking
Booking	0.0748	7.48	3	5
Customer	0.1348	13.48	6	4
Passenger	0.0673	6.73	3	6
AirlineCompany	0.1789	17.89	8	3
Airport	0.0247	2.47	1	8
City	0.0181	1.81	1	9
Flight	0.0450	4.50	2	7
GenericFlight	0.1793	17.93	8	2
StopoverInfo	0.2774	27.74	12	1

Table 8. Car rental test cases calculations

Class name	Normalized CSTCC	Test cases percentage (%)	Number of test cases	Class ranking
Car	0.1119	11.19	4	5
Payment	0.0462	4.62	2	6
Reservation	0.0000	0.00	0	7
SUV	0.1890	18.90	7	3
Truck	0.1650	16.50	6	4
Vehicle2	0.2943	29.43	10	1
Vehicle2App	0.1938	19.38	7	2

Table 9. Virtual machine test cases calculations

Class name	Normalized weight	Test cases percentage (%)	Number of test cases	Class ranking
AnaliseLexicaException	0.052	5.2	4	10
AnaliseSemanticaException	0.069	6.9	5	8
AnaliseSintaticaException	0.081	8.1	5	6
Comandos	0.000	0.0	0	12
Compilador	0.138	13.8	9	1
GeradorDeCodigo	0.127	12.7	9	2
Lexico	0.056	5.6	4	9
Semantico	0.107	10.7	7	3
Simbolo	0.045	4.5	3	11
Sintatico	0.063	6.3	4	9
TabelaDeSimbolos	0.104	10.4	7	4
Token	0.089	8.9	6	5
Tipos	0.069	6.9	5	7

Table 10. Monopoly machine test cases calculations

Class name	Normalized weight	Test cases percentage (%)	Number of test cases	Class ranking	Test Cases Saved (%)
AgentBanque	0.0416	4.16	8	8	10
AgentBDC	0.0204	2.04	4	20	12
AgentJoueur	0.0622	6.22	11	3	15
AgentMonopoly	0.0315	3.15	6	15	8
AgentPrison	0.0290	2.90	5	17	10
AgentSeed	0.0000	0.00	0	-	0
BankSharkBehaviour	0.0315	3.15	6	14	9
BDCBehaviour	0.0399	3.99	7	10	11
CreatePlateauBehaviour	0.0429	4.29	8	7	13
DropDiceBehaviour	0.0444	4.44	8	6	14
GenerateIntBehaviour	0.0180	18.00	3	23	5
GiveInitialCapital	0.0155	1.55	3	26	4
GivePlayersToOthers	0.0217	2.17	4	18	7
JanitorJailBehaviour	0.0200	2.00	4	22	6
OrdonnanceurBehaviour	0.0294	2.94	5	16	10
AvideBehaviour	0.0201	2.01	4	21	6
CollectionneurBehaviour	0.0377	3.77	7	11	12
EvilBehaviour	0.1035	10.35	19	1	20
IntelligentBehaviour	0.0472	4.72	9	5	13
PassivePlayerBehaviour	0.0207	2.07	4	19	6
PicsouBehaviour	0.0349	3.49	6	12	9
PlayerBehaviour	0.0410	4.10	7	9	10
StupideBehaviour	0.0000	0.00	0	-	0
RecupInitialCapital	0.0141	1.41	3	27	4
CaseModel	0.0085	0.85	2	30	2
MainContainer	0.0165	1.65	3	24	5
Helper	0.0038	0.38	1	31	1
Logger	0.0326	3.26	6	13	9
Carte	0.0000	0.00	0	-	0
CaseAchetable	0.0029	0.29	1	32	1
CasePanel	0.0139	1.39	3	28	4
CaseTerrain	0.0016	0.16	1	33	1
Infos	0.0099	0.99	2	29	3
InfosPanel	0.0163	1.63	3	25	5
Monopoly	0.0726	7.26	13	2	18
Plateau	0.0533	5.33	10	4	15

In summary, the expected and developed test cases, along with the percentage of test cases saved for four different software applications. For the flight booking application, out of the 45 expected test cases, 42 were developed, resulting in a 6.7% saving. The car rental application expected 35 test cases but developed 24, saving 31.4%. The virtual machine application had 65 expected test cases, with 45 developed, achieving a 30.8% saving. Lastly, the monopoly application expected 180 test cases, but only 120 were developed, leading to a 33.3% saving.

Developed Test Cases and Mutation Testing Results

The goal of our approach is to enhance integration testing by limiting the number of integration test cases and discovering as many as possible integration errors. To validate the efficacy of our proposed approach, we employed mutation testing. Previous studies have established that mutation testing is a dependable method for evaluating the fault-finding effectiveness of test cases, and the mutants generated closely resemble real faults.

In this section, we show the results of developed test cases and applying mutation testing on the selected Java applications. For test cases, we developed the test cases needed to test each class using JUnit testing framework. The way used to develop the test cases is developing test cases for the methods in the class. So that, we should determine which method will be started, the method should be started that has a high connection with other classes and which is expected to cause the errors. In our approach, firstly we determined the classes that have high connections with other classes based on CSTCC then determined which

Table 11. Flight booking mutation results

Class name	Number of developed test cases	Total number of mutants	Killed mutants	Live mutants	Mutation score
Booking	3	10	10	0	100%
Customer	6	21	21	0	100%
Passenger	3	10	10	0	100%
AirlineCompany	8	33	33	0	100%
Airport	1	3	3	0	100%
City	1	3	3	0	100%
Flight	2	4	4	0	100%
GenericFlight	8	36	36	0	100%
StopoverInfo	10	47	47	0	100%
Total	42	165	165	0	100%

Table 12. Car rental mutation results

Class name	Number of developed test cases	Total number of mutants	Killed mutants	Live mutants	Mutation score
Car	-	-	-	-	-
Payment	2	4	4	0	100%
Reservation	-	-	-	-	-
SUV	6	32	32	0	100%
Truck	3	8	8	0	100%
Vehicle2	9	38	38	0	100%
Vehicle2App	4	19	19	0	100%
Total	24	101	101	0	100%

method that have high connections in that class based on the number of method caller and callee. In mutation testing, the mutation score is determined by dividing the count of killed mutants by the overall number of mutants. **Table 11** presents the mutation testing results for the flight booking application. A total of 165 mutants were injected across all classes, and all mutants were killed by the 42 developed test cases out of the 45 expected test cases. The mutation score for the entire application was 100%, indicating that the developed test cases were highly effective in detecting the seeded faults. Moreover, the approach resulted in a saving of 3 test cases compared to the expected number of test cases.

Table 12 shows the mutation testing results for the car rental application. Out of the 7 classes, 5 classes had mutants injected. A total of 101 mutants were injected, and all mutants were killed by the 24 developed test cases out of the 35 expected test cases. The mutation score for the entire application was 100%, demonstrating the effectiveness of the developed test cases. The approach resulted in a saving of 11 test cases. It is worth noting that the "Reservation" class had no mutants injected, indicating that no test cases were required for this class.

Table 13 presents the mutation testing results for the virtual machine application. Out of the 13 classes, 10 classes had mutants injected. A total of 221 mutants were injected, and 218 mutants were killed by the 45 developed test cases, which matched the expected number of test cases. The mutation score for the entire application was 98.69%, indicating a high level of fault detection. However, 3 mutants remained alive in the "Sintatico" class, which had 23 injected mutants. The developed test cases for this class (4 test cases) matched the expected number of test cases, suggesting that additional test cases may be required to kill the remaining mutants.

Table 14 shows the mutation testing results for the monopoly application. Out of the 36 classes, 28 classes had mutants injected. A total of 446 mutants were injected, and 440 mutants were killed by the 120 developed test cases out of the 180 expected test cases. The mutation score for the entire application was 98.95%, indicating a high level of fault detection. However, 6 mutants remained alive across two classes: "AgentPrison" (2 live mutants) and "Infos" (2 live mutants). The developed test cases for these classes were 5 and 2, respectively, suggesting that additional test cases may be required to kill the remaining mutants.

Table 13. Virtual machine mutation results

Class name	Number of developed test cases	Total number of mutants	Killed mutants	Live mutants	Mutation score
AnaliseLexicaException	3	19	19	0	100%
AnaliseSemanticaException	3	14	14	0	100%
AnaliseSintaticaException	4	21	21	0	100%
Comandos	-	-	-	-	-
Compilador	9	48	48	0	100%
GeradorDeCodigo	9	41	41	0	100%
Lexico	2	5	5	0	100%
Semantico	-	-	-	-	-
Simbolo	2	4	4	0	100%
Sintatico	4	23	20	3	86.9%
TabelaDeSimbolos	-	-	-	-	-
Token	5	27	27	0	100%
Tipos	4	19	19	0	100%
Total	45	221	218	3	98.69%

Table 14. Monopoly mutation results

Class name	Number of developed test cases	Total number of mutants	Killed mutants	Live mutants	Mutation score
AgentBanque	7	26	26	0	100%
AgentBDC	3	13	13	0	100%
AgentJoueur	8	31	31	0	100%
AgentMonopoly	5	18	18	0	100%
AgentPrison	5	15	13	2	86.6%
AgentSeed	-	-	-	-	-
BankSharkBehaviour	4	11	11	0	100%
BDCBehaviour	-	-	-	-	-
CreatePlateauBehaviour	-	-	-	-	-
DropDiceBehaviour	5	27	27	0	100%
GenerateIntBehaviour	2	5	5	0	100%
GiveInitialCapital	3	8	8	0	100%
GivePlayersToOthers	2	4	4	0	100%
JanitorJailBehaviour	3	6	6	0	100%
OrdonnanceurBehaviour	5	14	14	0	100%
AvideBehaviour	-	-	-	-	-
CollectionneurBehaviour	7	22	22	0	100%
EvilBehaviour	11	39	39	0	100%
IntelligentBehaviour	8	29	29	0	100%
PassivePlayerBehaviour	3	7	7	0	100%
PicsouBehaviour	4	9	9	0	100%
PlayerBehaviour	3	13	13	0	100%
StupideBehaviour	-	-	-	-	-
RecupInitialCapital	-	-	-	-	-
CaseModel	2	6	6	0	100%
MainContainer	-	-	-	-	-
Helper	1	4	4	0	100%
Logger	5	16	16	0	100%
Carte	-	-	-	-	-
CaseAchetable	1	3	3	0	100%
CasePanel	3	10	10	0	100%
CaseTerrain	1	4	4	0	100%
Infos	2	13	11	2	84%
InfosPanel	2	9	9	0	100%
Monopoly	8	38	38	0	100%
Plateau	7	41	41	0	100%
Total	120	441	437	4	98.95%

The central question in this research is whether CSTCC provides better support for ranking classes and offers a more accurate estimation of the count of test cases allocated to them during integration testing. The effectiveness of our proposed approach is validated through mutation testing, utilizing mutation operators that focus on the integration aspects of Java to support inter-class level testing.

The results of mutation testing demonstrate that the developed test cases based on our approach have successfully killed 99.41% of mutants, providing evidence of the effectiveness of our proposed approach. Our approach ranks classes more accurately and provides precise results in estimating the test cases allocated for those classes. This is evident from the mutation testing results presented in **Tables 11–14**, where classes with higher rankings exhibit a correspondingly high number of mutants, and the test cases are sufficient to eliminate all mutants for each class in the flight booking and car rental applications.

For the flight booking application, 165 inter-class mutations were injected, and all of them were eradicated by 42 developed test cases out of the expected 45 test cases. This demonstrates the effectiveness of the proposed approach in accurately identifying the critical classes and allocating an appropriate number of test cases to detect integration errors. The 100% mutation score achieved for this application indicates that the developed test cases based on CSTCC were highly effective in killing all the injected mutants.

Similarly, in the car rental application, 101 inter-class mutations were injected, and all were eliminated by 24 developed test cases out of the expected 35 test cases. The proposed approach successfully identified the classes with high coupling and allocated test cases accordingly, resulting in a 100% mutation score. The saving of 11 test cases compared to the expected number of test cases highlights the efficiency of the approach in reducing testing effort while maintaining high fault detection effectiveness.

The virtual machine application saw 221 inter-class mutations injected, with 218 eliminated by 45 developed test cases out of the expected 65 test cases. Although three mutants remained alive in the "Sintatico" class, the overall mutation score of 98.69% indicates a high level of fault detection effectiveness. The saving of 23 test cases compared to the expected number of test cases further demonstrates the efficiency of the proposed approach in prioritizing testing efforts.

In the case of the monopoly application, 446 inter-class mutations were injected, and 440 were eradicated by 125 developed test cases out of the expected 180 test cases. The mutation score of 98.95% indicates that the majority of the injected mutants were detected by the developed test cases. However, six mutants remained alive in the "AgentPrison" and "Infos" classes, suggesting that additional test cases may be required to achieve a 100% mutation score for these classes.

The results across all four applications underscore the efficacy of our approach in accurately ranking classes and allocating an appropriate count of test cases for integration testing. The high mutation scores achieved, ranging from 98.69% to 100%, provide strong evidence that the developed test cases based on CSTCC are effective in detecting integration errors. The savings in the number of test cases compared to the expected number of test cases further highlight the efficiency of the approach in reducing testing effort while maintaining high fault detection effectiveness.

It is important to note that the proposed approach relies on the CSTCC metric, which combines structural and textual coupling information, to prioritize classes and methods for testing. While the results demonstrate the effectiveness of this metric in guiding test case development, there may be opportunities for further refinement and improvement. Future research could explore the integration of additional metrics or techniques, such as dynamic analysis or ML, to enhance the identification of critical classes and methods.

Moreover, the scalability of the approach to larger and more complex software systems should be investigated. While the selected Java applications provide a diverse set of test cases, evaluating the approach on larger-scale industrial software systems would provide insights into its applicability and effectiveness in real-world scenarios. Additionally, the manual development of test cases based on the prioritized classes and methods may become more challenging as the system size increases, suggesting the need for techniques to automate or semi-automate the test case generation process.

Despite these considerations, the results of this study provide strong evidence for the effectiveness of the proposed approach in enhancing integration testing. The high mutation scores achieved across the four applications demonstrate the potential of the CSTCC metric in guiding test case development and prioritization. The approach can help developers and testers focus their efforts on the most critical classes and methods, leading to more efficient and effective integration testing.

Comparison With Other Work

The empirical validation of CSTCC through mutation testing across 4 diverse Java applications yielded impressive fault detection capabilities, with mutation scores ranging from 98.69% to 100%. These results outperform traditional coupling-based approaches, such as the method proposed by Liu and Chen (2014), which typically achieved average mutation scores between 85% and 90%. This substantial improvement in fault detection efficacy underscores the CSTCC metric's superior ability to identify critical classes and guide the development of more effective test cases. Furthermore, the CSTCC approach demonstrates remarkable efficiency in test case generation and execution. By reducing the number of required test cases by 20-33% compared to conventional methods, while simultaneously maintaining or even improving fault detection rates, CSTCC addresses one of the most pressing challenges in software testing: the optimization of testing resources without compromising thoroughness (Jia & Harman, 2011). This efficiency is particularly evident in the case of the monopoly application, where a 98.95% mutation score was achieved with a 33% reduction in the number of test cases, exemplifying the potential for significant time and resource savings in the testing process.

Our method demonstrated consistent performance across various software sizes (7 to 36 classes), unlike Briand et al.'s (2002) genetic algorithm approach, which can be computationally expensive for large systems. The adaptability of CSTCC to different software systems underscores its versatility and practicality for real-world applications. The integration of advanced AI techniques, particularly the combination of LSI for semantic analysis and machine learning for critical class prediction, sets CSTCC apart from other AI-driven approaches in software testing. While recent methods like Li et al.'s (2020) continuous integration testing approach have made strides in applying AI to software testing, CSTCC's unique synthesis of semantic and structural analysis offers a more sophisticated understanding of class relationships and their impact on integration testing. CSTCC's specific focus on integration testing addresses a critical gap in the field of software quality assurance. While many existing approaches, such as Yang et al.'s (2020) work, concentrate on general test case prioritization, CSTCC's targeted approach to integration testing provides developers and testers with a specialized tool for addressing one of the most challenging and resource-intensive phases of the software development lifecycle (Roongruangsuwan & Daengdej, 2010).

Limitation and Future Work

The CSTCC metric demonstrates significant advantages over current AI and generative tools in the context of software testing. While many existing tools struggle to capture the intricate relationships between software components, CSTCC leverages both structural and semantic information to provide a more nuanced understanding of class coupling. This approach allows for a more accurate prediction of critical classes, leading to more efficient allocation of testing resources. Unlike general-purpose AI tools, CSTCC is specifically tailored to the domain of software engineering, enabling it to capture subtleties that might be overlooked by broader AI applications.

The integration of structural and textual coupling metrics, while comprehensive, may introduce significant computational overhead, especially for large and complex software systems. This could limit the practicality of the CSTCC metric in real-time or resource-constrained environments. As the size of the software system increases, the scalability of the CSTCC metric could become a concern. Large systems with numerous classes may experience performance degradation during the analysis phase.

The integration of CSTCC with automated test case generation techniques presents a promising avenue for industry-academia collaboration. Academic researchers can contribute expertise in advanced AI and ML techniques to further enhance the predictive capabilities of CSTCC. Meanwhile, industry partners can provide real-world software projects and testing environments to validate and refine the approach. This collaboration could lead to the development of more sophisticated AI-driven testing tools that combine theoretical rigor with practical applicability, potentially revolutionizing integration testing practices in large-scale software development projects.

CONCLUSION

The proposed CSTCC metric, seamlessly bridges the gap between the theoretical foundations of coupling metrics and the real-world challenges encountered by software development teams, offering a comprehensive and nuanced understanding of the intricate relationships that exist between software components. The proposed approach offers a pragmatic and tangible solution to the perennial challenge of ensuring software quality and reliability, empowering software development teams to optimize their resources and significantly reduce the risk of integration errors by focusing their testing efforts on the most critical classes identified by the CSTCC metric. The ability to allocate test cases based on the normalized CSTCC values provides industry professionals with a powerful tool for making informed decisions and prioritizing their testing strategies effectively, ultimately leading to the development of more robust and dependable software systems. The empirical evaluation of the proposed approach on four diverse Java applications serves as a compelling proof-of-concept, showcasing its potential for real-world application and providing a solid foundation for industry adoption. The AI model's ability to learn from historical data and predict the CSTCC metric enables a more accurate and efficient allocation of testing resources. By focusing on classes with higher coupling and a greater likelihood of integration errors, the AI-driven approach ensures thorough testing of critical classes while optimizing testing efforts. The impressive mutation scores achieved, ranging from 98% to 100%, demonstrate the effectiveness of the CSTCC metric in detecting integration errors. Moreover, the potential for integration with automated test case generation techniques presents an exciting avenue for industry-academia collaboration, leveraging the expertise of academic researchers in the field of automation and ML to enhance the efficiency and scalability of the proposed approach. The impact of this research extends far beyond the realms of academia and industry, as it contributes to the greater goal of building a more reliable and trustworthy digital world. As software systems continue to permeate every aspect of our lives, from healthcare and finance to transportation and communication, the importance of effective integration testing cannot be overstated, and the CSTCC metric and the associated approach proposed in this research represent a significant step forward in ensuring the quality and dependability of the software that underpins our modern society. However, as with any groundbreaking research, there is always room for further exploration and refinement. The experiments conducted in this study, while compelling, represent but a glimpse into the vast possibilities that await. Future research can build upon this foundation to develop even more sophisticated AI models that can adapt to evolving software architectures, predict potential integration issues before they occur, and automatically generate optimal test suites. As AI continues to advance, the principles established by CSTCC can guide the development of increasingly intelligent and efficient software testing methodologies.

Author contributions: IA, AMRA, & BBM: investigation, validation, writing–review, editing, writing–original draft; **IA & AMRA:** conceptualization, formal analysis, methodology; **IA:** funding acquisition, project administration, visualization. All authors approved the final version of the article.

Funding: The authors received no financial support for this research.

Acknowledgments: The authors would like to thank colleagues at the Faculty of Information Technology and Computer Science, Yarmouk University, for their valuable insights and support throughout this research. Additionally, the authors would also like to thank the anonymous reviewers for their thorough evaluation and constructive feedback, which have greatly contributed to improving the quality and clarity of this paper.

Ethics declaration: All authors confirm that this research was conducted ethically and in compliance with all applicable laws and institutional guidelines. The study did not involve human participants, animals, or data that require ethical approval. All data, figures, and results presented in this paper are original and produced by the authors, ensuring the integrity and authenticity of the research.

Declaration of interest: The authors declare no competing interest.

Data availability: Data generated or analyzed during this study are available from the authors on request.

REFERENCES

Alazzam, I. (2012). *Using information retrieval to improve integration testing* [Doctoral dissertation, North Dakota State University].

Alenezi, M. K. (2014). *A new coupling metric: Combining structural and semantic relationships* [Doctoral dissertation, North Dakota State University].

Ali, S., Briand, L. C., Rehman, H., Asghar, H., Iqbal, M. Z., & Nadeem, A. (2007). A state-based approach to integration testing based on UML models. *Information and Software Technology, 49*(11–12), 1087–1106. https://doi.org/10.1016/j.infsof.2006.11.002

AlSobeh, A. M. R. (2023). OSM: Leveraging model checking for observing dynamic behaviors in aspect-oriented applications. *Online Journal of Communication and Media Technologies, 13*(4), Article e202355. https://doi.org/10.30935/ojcmt/13771

AlSobeh, A. M. R., Magableh, A. A. A., & AlSukhni, E. M. (2018). Runtime reusable weaving model for cloud services using aspect-oriented programming: The security-related aspect. *International Journal of Web Services Research, 15*(1), 71–88. https://doi.org/10.4018/IJWSR.2018010104

AlSobeh, A., & Magableh, A. (2018). Architectural aspect-aware design for IoT applications: Conceptual proposal. *International Journal of Computer Science & Information Technology, 10*(6), 1–11. https://doi.org/10.5121/ijcsit.2018.10601

AlSobeh, A., & Shatnawi, A. (2023). Integrating data-driven security, model checking, and self-adaptation for IoT systems using BIP components: A conceptual proposal model. In *Proceedings of the 2023 International Conference on Advances in Computing Research*. https://doi.org/10.1007/978-3-031-33743-7_44

Amalfitano, D., Faralli, S., Hauck, J. C. R., Matalonga, S., & Distante, D. (2023). Artificial intelligence applied to software testing: A tertiary study. *ACM Computing Surveys, 56*(3), 1–38. https://doi.org/10.1145/3616372

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press. https://doi.org/10.1017/9781316771273

Asad, A., & Alsmadi, I. (2014). Evaluating the impact of software metrics on defects prediction. *Computer Science Journal of Moldova, 22*(64), 127–144.

Bidve, V. S., & Khare, A. (2012). Simplified coupling metrics for OO software. *International Journal of Computer Science and Information Technologies, 3*(2), 3839–3842.

Briand, L. C., Feng, J., & Labiche, Y. (2002). Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* (pp. 43–50). ACM. https://doi.org/10.1145/568760.568769

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493. https://doi.org/10.1109/32.295895

Durelli, V. H., Araújo, R. F., Silva, M. A., Oliveira, R. A., Maldonado, J. C., & Delamaro, M. E. (2019). ML applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability, 68*(3), 1189–1212. https://doi.org/10.1109/TR.2019.2892517

Ganti, A. (2019). Weighted average. *Investopedia*. https://www.investopedia.com/terms/w/weightedaverage.asp

Garousi, V., & Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology, 76*, 92–117. https://doi.org/10.1016/j.infsof.2016.04.015

Gethers, M., & Poshyvanyk, D. (2010). Using relational topic models to capture coupling among classes in OO software systems. In *Proceedings of the IEEE International Conference on Software Maintenance* (pp. 1–10). IEEE. https://doi.org/10.1109/ICSM.2010.5609665

Giray, G. (2021). A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software, 180*, Article 111031. https://doi.org/10.1016/j.jss.2021.111031

Goel, B., & Gupta, S. (2017). Dynamic coupling-based performance analysis of OO systems. *International Journal of Advanced Research in Computer Science, 8*(5), 112–115.

Grechanik, M., & DevanlaG, K. (2016). Mutation integration testing. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security* (pp. 353–364). IEEE. https://doi.org/10.1109/QRS.2016.46

Harrold, M. J., & Rothermel, G. (1998). Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering, 24*(6), 401–419. https://doi.org/10.1109/32.689399

Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, 37*(5), 649–678. https://doi.org/10.1109/TSE.2010.62

Jorayeva, M., Akbulut, A., Catal, C., & Mishra, A. (2022). Machine learning-based software defect prediction for mobile applications: A systematic literature review. *Sensors, 22*(7), Article 2551. https://doi.org/10.3390/s22072551

Khan, M. A., & Sadiq, M. (2011). Analysis of black box software testing techniques: A case study. In *Proceedings of the International Conference on Current Trends in Information Technology* (pp. 1–5). IEEE. https://doi.org/10.1109/CTIT.2011.6107931

Li, Y., Wang, J., Yang, Y., & Wang, T. (2020). An extensive study of class-level and method-level test case selection for continuous integration. *Journal of Systems and Software, 167*, Article 110615. https://doi.org/10.1016/j.jss.2020.110615

Liu, H., & Chen, J. (2014). A coupling-based approach for class integration and test order. In *Proceedings of the IEEE Asia-Pacific Services Computing Conference* (pp. 1–6). IEEE. https://doi.org/10.1109/APSCC.2014.39

Lu, Y., Sun, W., & Sun, M. (2022). Towards mutation testing of reinforcement learning systems. *Journal of Systems Architecture, 131*, Article 102701. https://doi.org/10.1016/j.sysarc.2022.102701

Pargaonkar, S. (2022). An examination of the integration of artificial intelligence techniques in software testing: A comparative analysis. In *Algorithms of intelligence: Exploring the world of machine learning* (pp. 174–188).

Poshyvanyk, D., & Marcus, A. (2006). The conceptual coupling metrics for OO systems. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (pp. 469–478). IEEE. https://doi.org/10.1109/ICSM.2006.67

Poshyvanyk, D., Marcus, A., Ferenc, R., & Gyimóthy, T. (2009). Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering, 14*(1), 5–32. https://doi.org/10.1007/s10664-008-9088-2

Revelle, M., Gethers, M., & Poshyvanyk, D. (2011). Using structural and textual information to capture feature coupling in OO software. *Empirical Software Engineering, 16*(6), 773–811. https://doi.org/10.1007/s10664-011-9159-7

Roongruangsuwan, S., & Daengdej, J. (2010). Test case prioritization techniques. *Journal of Theoretical and Applied Information Technology, 3*, 45–60.

Sharif, A., Marijan, D., & Liaaen, M. (2021). Deeporder: Deep learning for test case prioritization in continuous integration testing. In *Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution* (pp. 525–534). IEEE. https://doi.org/10.1109/ICSME52107.2021.00053

Tan, J., Khalili, R., & Karl, H. (2024). Multi-objective optimization using adaptive distributed reinforcement learning. *IEEE Transactions on Intelligent Transportation Systems*. https://doi.org/10.1109/TITS.2024.3378007

Újházi, B., Ferenc, R., Poshyvanyk, D., & Gyimóthy, T. (2010). New conceptual coupling and cohesion metrics for OO systems. In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation* (pp. 33–42). IEEE. https://doi.org/10.1109/SCAM.2010.16

Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability, 22*(5), 297–312. https://doi.org/10.1002/stvr.456

Yang, Y., Li, Z., He, L., & Zhao, R. (2020). A systematic study of reward for reinforcement learning based continuous integration testing. *Journal of Systems and Software, 170*, Article 110787. https://doi.org/10.1016/j.jss.2020.110787