*Technoscience Journal for Community Development in Africa* 5:1 (2026) 11–23
ISSN 2659-0573

*Research Article*

# Hybrid framework for software change impact analysis using data-flow analysis and optimised latent dirichlet allocation

**Shakirat Ronke Yusuff,[1] Amos O. Bajeh,[2] Jumoke Falilat Ajao[1]**

[1]Department of Computer Science, Kwara State University, Malete, Ilorin, Nigeria
[2]Department of Computer Science, University of Ilorin, Ilorin, Nigeria

**Abstract**: Software systems continually evolve, making change management a vital yet costly part of software lifecycle processes. Change Impact Analysis (CIA) is essential to foresee the effects of modifications, minimizing unintended side effects and managing ripple impacts. Existing CIA techniques, however, often struggle to balance precision and recall, impacting their effectiveness. Many rely exclusively on either structured or unstructured source code data, missing out on insights offered by the other, which limits impact prediction accuracy. To address this, a hybrid framework for static source code CIA has been proposed, combining analyses of both structured and unstructured code information at the method level. This framework employs data-flow analysis for structured data and an optimized Latent Dirichlet Allocation (LDA) model for unstructured data. The LDA hyperparameters are fine-tuned using a differential evolution algorithm to approach optimal settings. The resulting impact sets from both analyses are merged through a union operation to capture comprehensive change effects. Experimental evaluation on three open-source Java projects demonstrated that this hybrid approach doubled the performance compared to using structural analysis alone, while preserving comparable precision and recall. Consequently, this framework offers the potential to reduce maintenance costs and optimize resource allocation. Future work may explore alternative LDA variants and optimization techniques to further enhance CIA accuracy.

**Keywords:** Differential Evolution, Change Impact Analysis, Latent Dirichlet Allocation, data-flow analysis, Optimisation Algorithm

## 1. Introduction

The pervasiveness of software systems in modern society necessitates their continuous evolution to remain useful in their operational environment. This evolution, partly driven by bug fixes, new user requirements, and technological advancements, makes software change an inevitable part of software maintenance (Bajeh et al., 2019). However, as software systems evolve, they tend to increase in complexity, and implementing changes becomes a difficult task (Angerer et al., 2019). Thus, a seemingly trivial change can impact numerous other system entities due to the intricate dependencies encoded within the software, potentially affecting the system's overall functionality if not properly managed (Ramu and Reddivari, 2025; Agrawal and Singh, 2020).

Software Change Impact Analysis (CIA) is a vital activity for assessing and identifying the potential consequences of implementing software changes. CIA is carried out to ensure a software system's effectiveness and reliability are not compromised by preventing side effects or managing ripple effects that could arise from implementing software change (Dai et al., 2022). The CIA process typically involves identifying an initial Starting Impact Set (SIS) based on a change request, which is then used to identify an Estimated Impact Set (EIS). The accuracy of a CIA technique is dependent on how much of the EIS reflects the Actual Impact Set (AIS), that is, the set of entities truly modified (Yusuff et al., 2021). However, a primary challenge in CIA is the trade-off between precision and recall (Cai, 2018). A high number of False Positives (FP) (low precision)

* Corresponding author:
Email: Shakirat.yusuff@kwasu.edu.ng

KWASU PRESS AND PUBLISHING

forces engineers to waste time on irrelevant tasks, while a high number of False Negatives (FN) (low recall) can lead to missed impacts and the introduction of new bugs. To address these issues, hybrid CIA approaches that combine multiple analysis methods have gained prominence, leveraging the strengths of each to improve overall accuracy.

Prior researches have independently explored structured and unstructured source code data for CIA. Structural analysis reveals syntactic dependencies, such as method calls and data flows, in the structured source code data using techniques like control-flow and dependency graph analysis (Angerer et al., 2019). Conversely, unstructured data, such as identifiers and comments, provide conceptual insights that structural analysis may overlook (Poshyvanyk et al., 2009). However, depending solely on one type of source code data makes CIA less effective because each type captures different types of dependencies. For example, structured data captures functional dependencies, while unstructured data captures contextual insights. Integrating both structured and unstructured data, however, has been indicated to provide a more comprehensive understanding of software dependencies, thereby addressing the precision-recall trade-off more effectively (Yusuff et al., 2021), which is crucial for improving software reliability and performance in complex systems.

This study proposes a hybrid framework for source code CIA that synergizes data-flow analysis with an optimised Latent Dirichlet Allocation (LDA) model using DE. This will be achieved by analysing static structured and unstructured source code data with the aim of delivering a balanced and effective CIA method that mitigates maintenance expenses while improving impact prediction, ultimately leading to more efficient software development processes.

The rest of the paper is laid out as follows: section two presents a review of the literature, section three discusses the methodology, and section four presents the results, while the implications of the presented results and conclusion are presented in sections five and six, respectively.

## 2. Literature Review

The literature on Software Change Impact Analysis (CIA) highlights its essential role in managing the complexities of evolving software systems often driven by bug fixes, feature additions, and technological advances. To this end, a wide range of CIA techniques have continuously emerged in the software engineering research domain due to their importance to software development, maintenance, and evolution. Hattori et al. (2008) proposed the use of precision and recall metrics for assessing the accuracy of CIA techniques. The study associated the concept of false positives and false negatives with the precision and recall metrics, respectively. Empirical experiments carried out on the Impala tool using three software systems revealed that there is a trade-off between precision and recall, with precision values increasing as recall decreases and vice versa. Consequently, prior studies have highlighted the importance of combining multiple source code information in software engineering tasks (Abdu et al., 2024; Yusuff et al., 2021; Wang et al., 2018).

In an early attempt at integrating multiple sources of source code information for CIA, Kagdi et al. (2010) combined conceptual and evolutionary coupling techniques based on the Latent Semantic Indexing (LSI) model and source code commits in multiple versions, respectively. The empirical experiment of the technique was carried out on four open-source systems, which revealed a significant improvement of up to 45% compared to the baseline techniques. Similarly, Gethers et al. (2012) presented an integrated approach that performs CIA based on a combination of information retrieval, dynamic analysis, and software repository mining. The technique employed an LSI model to capture conceptual couplings and only utilized dynamic analysis and repository mining if additional information, such as execution trace and an initial source code entity, was verified for change and available. By combining all the information derived from the analyses, a final impact set is computed. Empirical evaluation on four open-source software systems confirmed the hypothesis that a combination of multiple source code information can significantly improve accuracy over baseline techniques. However, the trade-off between precision and recall was not addressed in the study.

Alenezi and Magel (2014) presented a coupling metric based on a combination of structural and textual analysis. The authors combined structural couplings with LSI for predicting conceptual coupling between source code entities. Likewise, Wang et al. (2018) combined the LSI model and a neural network-based information retrieval (IR) technique called ‘Doc2Vec’ to address the challenges of using the LSI model to analyze source code data. Experimental results on the three open-source software systems revealed a significant improvement in F-measure when compared to using either method independently. However, the authors did not utilize structured source code information indicated as relevant for CIA tasks. This study also seeks to address

the shortcomings of LSI by utilizing the LDA model for textual analysis and integrating structural analysis for a more holistic CIA. While Savage et al. (2010) developed a tool called Topic XP to assist software developers to gain an overview of the software system during maintenance by extracting unstructured source code information using an LDA model, the tool could only extract and visualize unstructured source code information. It was not designed to change propagations that result from implementing a change request.

Although Dai et al. (2022) did not combine multiple source code information, they presented a CIA technique based on the analysis of multiple types of variables of C programs within and across the methods in the program. As a first step, the change code was split into multiple variables. Thereafter, a corresponding mechanism for analysing each type of variable was proposed to ensure the precision of the estimated impacts. Empirical evaluation carried out on five C programs revealed its ability to analyze change impacts at statement and method level of granularity accurately.

Additionally, various metaheuristic algorithms have been used in the literature for tuning LDA hyperparameters in an effort to achieve "near-optimal" solutions on software engineering data. Panichella (2019) compared the performance of five search algorithms: Differential Evolution (DE), Genetic Algorithm (GA), Particle Swarm Optimisation (PSO), Simulated Annealing (SA), and Random Search for tuning the LDA model parameters on source code data for identifying duplicate bugs. Empirical evaluation carried out on seven Java projects revealed that there is no dominant search algorithm for tuning the LDA parameters since each meta-heuristic algorithm was shown to have comparable performance across different projects. However, PSO and Random Search were found to be less effective than the other meta-heuristic algorithms employed in the study. Also, Yarnguy and Kanarkard (2018) presented an approach called ACO-LDA to find "near-optimal" values for LDA hyperparameters α and β using Ant Colony Optimisation (ACO). ACO-LDA was evaluated on the standard topic modelling dataset from UCI based on the perplexity metric. Experimental results revealed that ACO-LDA outperformed the untuned LDA model. Panichella, McMillan, et al. (2013) presented LDA-GA that automatically calibrates LDA hyperparameters: number of Gibbs sampling (n), K, α, and β using GA. The best LDA model parameters were returned from the chromosomes with the highest fitness. The technique was shown to significantly improve the performance of LDA models in software engineering tasks. Similarly, Agrawal et al., (2018) presented an approach called LDADE, which utilized the DE algorithm to tune the LDA hyperparameters: k, α and β. The result from experiments on different implementations of the LDA model and different datasets indicated a significant improvement in the stability of the LDADE model compared to the untuned LDA and the LDA model tuned on GA. The evaluation was based on the topic similarity and F-measure on various supervised and unsupervised tasks which revealed the effectiveness of the technique.

Despite the effort at improving the effectiveness of CIA, prior studies did not place emphasis on addressing the precision-recall tradeoff. Consequently, this study aims to address the precision-recall tradeoff by proposing a hybrid framework for combining structural and textual analysis for CIA in a way that change impacts from both analyses are captured.

## 3. Materials and methods

This section outlines the datasets, algorithms, experimental procedures and performance metrics employed in the study. Three main stages were involved, which are structural analysis, textual analysis and the hybrid analysis, which combines both structural and textual analysis using the union operation. The proposed hybrid framework developed is depicted in Figure 1.

Figure 1 illustrates the workflow of the proposed hybrid software change impact analysis framework. The process begins with data preparation; thereafter, structural analysis is carried out by extracting program dependencies from the structured source code data and applying reachability analysis and a distance criterion to obtain the Estimated Impact Set for structural analysis (EISs). The Differential Evolution (DE) optimisation algorithm was used to tune LDA hyperparameters to obtain a "near optimal" configuration on the dataset. Thereafter, textual analysis was performed on the unstructured source code data using the optimised LDA model to obtain the Estimated Impact Set ($EIS_t$). Finally, the $EIS_s$ and $EIS_t$ were combined based on the union operation to identify the final estimated impact set ($EIS_f$), which was evaluated using the precision, recall and F-measure metrics to assess the effectiveness of the hybrid framework.

### *3.1. Dataset Preparation*

The dataset utilized in this study was retrieved from http://www.cs.wm.edu/semeru/data/icse2012-impact-analysis. Three open-source Java projects, jEdit, JabRef and ArgoUML, were selected to allow for comparison with related studies. For each project, the dataset included the source code, detailed change request

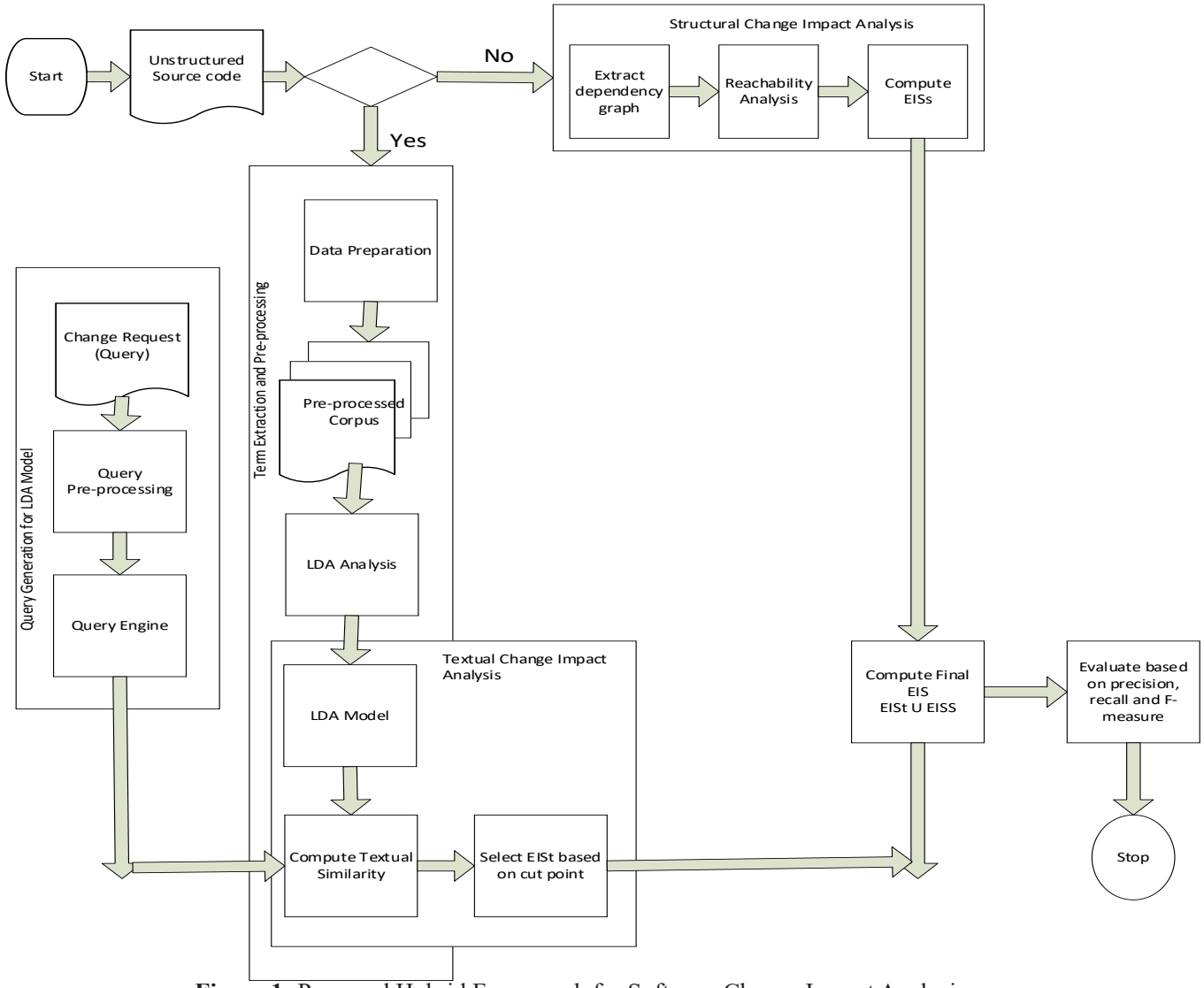


**Figure 1:** Proposed Hybrid Framework for Software Change Impact Analysis

descriptions (bug reports), and corresponding "gold set" files which list the methods modified to resolve each change. A summary of the dataset is presented in Table 1.

### *3.2. Structural Analysis on Structured Source Code*

Structured source code data was analyzed using data-flow analysis to generate the *Estimated Impact Set (EISs)* by extracting the program dependency graph to represent the structural dependencies between methods. The CIA process was then initiated from a Starting Impact Set (SIS), which consisted of at most three methods randomly selected from the gold set (AIS) for a given change request. Next, reachability analysis was performed on the dependency graph to identify all direct and indirect methods dependent on the selected SIS. To limit the number of false positives, the search was constrained by a depth parameter. Finally, the union of all identified methods (excluding those in the SIS) was recorded to form the final EISs. This process is summarized in Algorithm 1.

Algorithm I: Estimating Impacts based on Structural Analysis

Input: Dependency graph as DG, Starting Impact Set as SIS, depth, method mi
Output: Estimated Impact Set from structural analysis (EISs)

Begin:

1. Initialize storage S= { } (for EISs)
2. while maximum $m_j$ not reached do
3. for each method $m_i$ in DG do
4. if method $m_j$ in SIS then
5. DEP trace all dependencies of mj

**Table 1:** Overview of the dataset characteristics

| Software Project | Version | Lines of Code | Methods | Average gold set | **Change Request** |
|---|---|---|---|---|---|
| JEdit | 4.3 | 103,896 | 6, 413 | 4 | 272 |
| JabRef | 2.6 | 74,182 | 4, 607 | 7 | 40 |
| ArgoUML | 0.22 | 148,892 | 11, 000 | 5 | 132 |

```
6.                  apply depth criteria
7.                 S = {DEP}
8.               endif
9.          end for
10.   endwhile
11.  return EISs
```

End

### *3.3. Textual Analysis on Unstructured Source Code*

Textual analysis was performed on unstructured source code data to generate the Estimated Impact Set ($EIS_t$) The subsequent subsections detail the steps employed for textual analysis.

#### *3.3.1. Extraction of unstructured source code data*

Unstructured source code data, including identifiers and comments present in each software project: jEdit, *JabRef* and *ArgoUML,* were extracted at a method level of granularity, being one of the most important program entities for source code analysis. On the other hand, numbers, punctuation, and special characters were removed since they do not bring relevant information to the CIA task. After the extraction of the relevant terms, contiguous terms in each document of the generated text corpus corresponded to the implementation of a method in the software project. As such, the total number of documents in each text corpus equalled the number of methods in the corresponding software project.
Similarly, the terms from the short and long change request descriptions data were extracted to obtain queries to be used for querying the optimised LDA model during the CIA process as obtained in information retrieval tasks.

#### *3.3.2. Data preprocessing*

Data preprocessing is crucial to the generation of LDA models and text-mining applications in general, as it helps to remove noise and improve performance. Typically, the selection of pre-processing steps is a function of the data and the type of analysis. The following Natural Language Processing (NLP) preprocessing steps identified in previous studies were applied to the generated text corpora and queries (Banitaan and Alenezi, 2015; Alenezi, 2015)

i. Tokenization: Multi-word identifiers were split based on the identified camel case naming convention used in the software projects to obtain a "list of strings", also known as tokens. For instance, the multiword identifier *"gridLayout"* was split into *"grid"* and *"Layout"*.
ii. Case Normalization: All tokens in the text corpus for each software project were converted to lower case letters to eliminate case-sensitivity since there can be multiple instances of a token in different parts of a document expressed differently. For example, the tokens "*grid*, "*Grid"* and "*GRID"* were all converted to "*grid"* for each instance of the word in the corpus. The output of this step is a list of normalized strings.
iii. Stop-words Removal: Common English language words like 'the', 'and', 'is', 'of', and so on, which do not contribute to the semantics of the software projects and are found to be irrelevant for topic modelling, were removed using the stop-words package from the Natural Language Toolkit Kit (NLTK) library. Additionally, Java programming language keywords, which were also found to have no significant impact on the task, were removed. This step is important for maximizing the processing speed and reducing the amount of storage space required during analysis.
iv. Stemming: Terms in the corpus were then reduced to their root form to limit the number of terms with related meanings while preserving the original meaning of the terms. For example, the words 'encode', 'encoded' and 'encoding' were all reduced to the root word 'encod'. This step

further helps to improve the processing speed during analysis.

v. Pruning: Pruning was done by removing terms with two or fewer characters, as they were considered non-informative to the task.

### *3.3.3. Generation of optimised LDA model*

Since the LDA model was originally designed for natural language texts, as such, using it on source code data which have been found to be more repetitive requires a careful selection of hyperparameters for the LDA algorithm. This study utilized the Differential Evolution (DE) optimisation algorithm process for generating an optimised LDA for tuning LDA. The steps for generating the optimised LDA model are shown in Figure 2.

As shown in Figure 2, the preprocessed text corpus and queries were transformed into a D×W vector representation (Document-Term matrix) as a requirement for the LDA algorithm. Also, as a prerequisite for LDA model training, LDA hyperparameters (k, α, β) shown to significantly influence the generation of the LDA model were tuned using DE based on the parameter settings recommended in the literature (Panichella, 2021; 2019). Topic coherence was used as a fitness function to evaluate the LDA model in obtaining "near optimal" values. Topic coherence have shown to preserve the semantics of unstructured source code information and is computed in Equation 1. Subsequently, the D×W matrix and "near optimal" LDA hyperparameters were passed as inputs to the LDA algorithm. The LDA algorithm through statistical inference converted the D×W, into two lower dimensional matrices: D×K (document-topic matrix) and K×W (topic-term matrix) where D represents the number of documents in the text corpus; K, the number of topics and W, the vocabulary size.

Parameter configurations for LDA and DE algorithms are shown in Tables 2 and 3, respectively, while the DE process for tuning LDA parameters K, α and β is presented in Algorithm 2.

Table 2: LDA Hyperparameter Configuration

| LDA Parameters | Description | Tuning Range |
|---|---|---|
| k | Number of Topics | [20, 100] |
| α | Topic distribution over documents | Asymmetric and auto **in** the range [0, 1] |
| β | Topic distribution over words | Symmetric and auto **in** the range [0, 1] |

As shown in Table 3, the number of topics k is fixed between the range of 20 and 100 with an increment step of 6. Values for α were chosen from asymmetric and auto in the range [0, 1] while values for β were selected from symmetric and auto in the range [0, 1]. Asymmetric values were generated based on a fixed normalized asymmetric prior to 1.0/(Number of Topics). auto values were learned directly from the input data while symmetric values corresponded to a scalar value for a symmetric prior over topic/word probability.

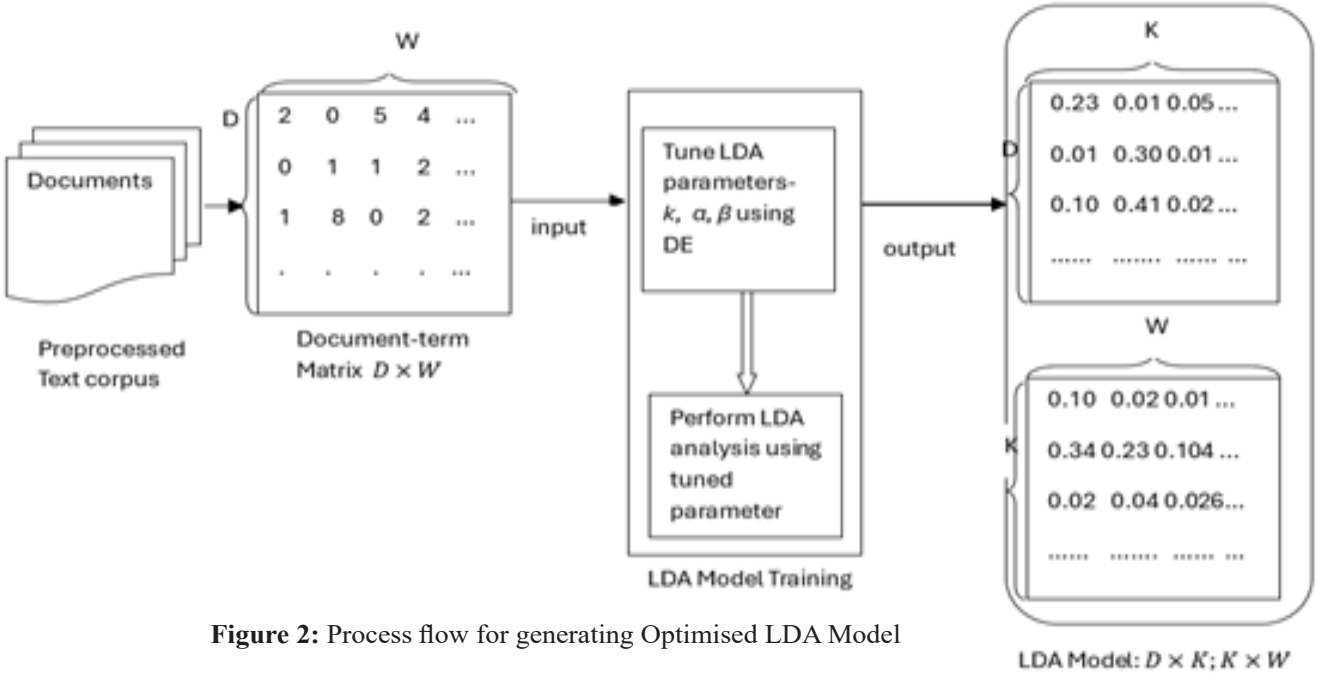


**Figure 2:** Process flow for generating Optimised LDA Model

Table 3: Differential Evolution parameter settings

| Differential Evolution Parameters | **Value** |
|---|---|
| Population size (np) | 10 |
| Crossover probability (cr) | 0.9 |
| Differential weight factor (f) | 0.8 |
| Number of generations (iter) | 5 |
| Maximum fitness evaluations (MaxFES) | 50 |

Algorithm: The Differential Evolution process for tuning LDA parameters K, α and β
Input: np=10, f=0.8, cr=0.9, iter = 5, MaxFES, Data={BoW corpus, K, α and β}
Output: best{topic_coherence}, LDA_Model
Begin:
1. currentGeneration = { }
2. Initialize all np individuals with random positions in the search space
3. for i = 0 to np-1 do
4. LDA_Score =topicCoherenceScore(Population[i], n, Data)
5. currentGeneration.add([Population[i], LDA_Score)
6. end for
7. for i = 0 to iter do
8. New_Generation = { }
9. for j = 0 to np-1 do
10. Si = Extrapolate (Population[j], Population, cr, f, np)
11. if LDA_Score(Si) ≥ currentGeneration[j][i] then
12. NewGeneration.append(Si,LDA_Score (Si, n, Data))
13. else
14. NewGeneration.append([CurrentGeneration[j][0], CurrentGeneration[j][1])
15 end if
16. end for
17. CurrentGeneration = NewGeneration
18. end for
19. {topic_coherence} = bestSolution(currentGeneration)
20. LDA_Model =currentGeneration
21. return best{topic_coherence}, LDA_Model
22. function Extrapolate (old, pop, cr, f, np)
23 a, b, c = threeOthers(pop, old)
24 new f = emptySet
25 for i = 0 to np-1 do
26 if cr ≤ random() then
27 new f.append(old[i])
28 else
29 new f.append(trim(i,(a [i] +f × (b[i]- c[i])))
30 end if
31 end for
32 return new f
33 end function
End

The DE process is a continuous one and only terminates when the maximum number of fitness functions (MaxFES) are reached. As shown in Algorithm 2, the bestSolution function returns the LDA model that gives the best topic coherence scores for the tuned parameters: K, α and β. The topic coherence score is best when it is highest which indicates that words in the topics are highly correlated and is computed based on Equation 1.

$$\text{Topic Coherence}(z,D) = \text{mean}\{\text{score}(w_i, w_j, \epsilon)\} \quad (1)$$

Where; z is the set of words that best describes the sampled topic z; D is the document collection; score is a measure of the coherence between a pair of words. $w_i$ and $w_j$ represent a pair of words that describes the topic ( $w_i \in V$; $w_j \in V$; i, j ∈ 1... 10 except i = j); V is the vocabulary of words present in D while ϵ (epsilon) is a smoothing value that is chosen based on the dataset to prevent the occurrence of extreme values while guaranteeing real values.

Specifically, the coherence score of the generated LDA models in this study was computed using the CoherenceModel class in the genism module in Python. The coherenceModel class is based on the c_v measure, which uses a Normalized Pointwise Mutual Information (NPMI) for computing the co-occurrence of words in a topic as shown in Equation 2.

$$NPMI\left(w_i, w_j\right) = \left( \frac{log \frac{p\left(w_i, w_j\right) + \epsilon}{p(w_i).p\left(w_j\right)}}{-log\left(P\left(w_i, w_j\right) + \epsilon\right)} \right) \quad (2)$$

Where; $p(w_i,w_j)$ indicates the number of times both words $w_i$ and $w_j$ co-occur in a document, and ϵ (epsilon) is a smoothing value used to prevent the occurrence of extreme values.

### *3.3.4. Estimating Impact Sets based on the generated optimised LDA model*

In this step, the generated optimised LDA models and queries from the previous steps were utilized for textual analysis as follows:

i. Compute document similarity; and
ii. Derive the Estimated Impact Set from textual analysis (EISt) using specific cut points.

i. Compute document similarity

The similarity between the text corpus and queries was computed to identify methods that were likely to be impacted by implementing a change request. The identified methods constituted the $EIS_t$. The Jenson Shannon (JS) distance metrics was adopted

for computing document similarity as it is based on probability theory such as Jenson Shannon Divergence (JSD) and the Kullback-Leibler (KL) Divergence, which have been identified as valuable options for computing document similarity in probabilistic models like LDA (Panichella, 2021). The JS distance for any two discrete probability distributions P and Q based on the JSD and the KL divergence is as shown in Equation 3.

$$JSD(P \parallel Q) = 1/2\ D(P \parallel M) + 1/2\ D(Q \parallel M) \quad (3)$$

where; $M = 1/2\ (P + Q)$ (4)

and D is the Kullback-Leibler divergence computed as

$$D(P \parallel Q) = \sum_i P(i)\ \log(P(i)/Q(i)) \quad (5)$$

Substituting Equations 4 and 5 in Equation 3 results in Equation 6.

$$JSD(P \parallel Q) = 1/2 \sum [P(i)\ \log(P(i)/(1/2\ (P(i)+Q(i)))) + Q(i)\ \log(Q(i)/(1/2\ (P(i)+Q(i)))) ] \quad (6)$$

Jenson Shannon Distance is derived as the square root of the JSD, as shown in Equation 7.

$$JS\ Distance = \sqrt{JSD(P \| Q)} \quad (7)$$

The JS distance metric was used to obtain the similarity scores for the topic distribution of the new unseen document (query) P by comparing the divergence of their distributions against all the topic distributions of the corpus Q. A Smaller JS distance inferred statistically closer (similar) probability distributions. The list of documents was then ranked based on the obtained similarity scores which is usually very large and can significantly affect the accuracy of the $EIS_t$ since not all the documents in the list are truly impacted. This study utilized the cut point method which selected the top n methods from the list with n values set to 5, 10, 15 and 20 methods respectively to limit the size of $EIS_t$. Thereafter, the accuracy of the estimated impacts $EIS_t$ at cut points 5, 10, 15, 20 was evaluated based on precision, recall and F-measure metrics by comparing with the entities in the gold set representing the Actual Impact Set (*AIS*) in this study.

### *3.4. Hybrid Framework Combining Structural and Textual Analysis*

The final Estimated Impact Set (EISf) was computed by combining the results from both structural and textual analyses using the union operation as shown in Equation 8.

$$EIS_f = EIS_s \cup EIS_t \quad (8)$$

Computing the EISf based on the union operation ensured that the hybrid framework captured the impacts uniquely identified each analysis aiming to improve recall without severely degrading precision.

### *3.5. Performance Evaluation Metrics*

The accuracy of the CIA techniques: EISs, EISt and EISf was evaluated using precision, recall, and F-measure.

(i) Precision: This is expressed as the ratio of the correctly estimated software entities and the total estimated software entities.

$$Precision = |EIS \cap AIS| / |EIS| \quad (2)$$

(ii) Recall: This is expressed as the ratio between correctly estimated software entities and the total actual affected entities.

$$Precision = |EIS \cap AIS| / |EIS| \quad (3)$$

(iii) F-measure: F-measure defines the harmonic mean of the precision and recall values, and is expressed formally as:

$$F\text{-measure} = (2 \times Precision \times Recall)/(Precision + Recall) \quad (4)$$

The precision and recall metrics are two relative metrics with trade-offs as a result, the f-measure metric has been suggested to address the trade-off between precision and recall when evaluating the accuracy of CIA techniques (Hattori et al., 2008).

## 4. Results

This section presents the results of the experiments carried out on the three software projects, comparing the performance of the baseline techniques (textual and structural) with the hybrid framework proposed in this study. The results are presented separately for each of the techniques utilized.

### *4.1 Performance Evaluation of Baseline Techniques and Proposed Hybrid Framework*

The experiments presented in this section were taken from the change request descriptions and gold set files of each software project under study. The selection of the experiments was carefully selected based on gold set files with varying number of methods to eliminate bias. As such gold set files containing on average between 9 and 33 methods were selected for the experiments. Specifically, the experiments being reported here are jEdit software with Bug ID #1382388; Jabref with Bug ID #1297576 and AgroUML with Bug ID #2618 which were found to have been used in related studies.

### *4.1.1. Results for structural analysis*

EISs was evaluated based on Precision (P), Recall (R) and F-measure across all three software projects using the Bugs identified in section 4.1. The result of the comparison across all three software projects is shown in Table 4 with bold fonts indicating the best performance.

Table 4: Evaluation of EISs on all Software Projects

| Software project | $EIS_s$ | TP | FP | FN | P | R | F-measure |
|---|---|---|---|---|---|---|---|
| jEdit | 3 | 1 | 2 | 8 | 0.33 | 0.11 | 0.17 |
| JabRef | 5 | 2 | 3 | 31 | **0.60** | 0.10 | 0.17 |
| Agro-UML | 4 | 2 | 2 | 14 | 0.5 | **0.13** | **0.21** |

As shown in Table 4, the highest precision was obtained on the JabRef software project, which indicated that it identified the greatest number of truly impacted methods. The recall for the three software projects, jEdit, JabRef and AgroUML, was 0.11, 0.10 and 0.13, respectively, which indicated that the EISs was unable to correctly identify many truly impacted methods for each experiment. The F-measure, which addresses the trade-off between precision and recall, was greatly affected by the recall values and was obtained as 0.21, 0.17 and 0.17 for the jEdit, JabRef and ArgoUML software projects, respectively.

### *4.1.2. Results for Textual analysis*

Using the same Bug IDs utilized in structural analysis, experiments for textual analysis was conducted for all cut points n= 5, 10, 15 and 20, and evaluated based on the precision, recall and F-measure. Comparison of the performance of EISt across all cut points is presented in Figures 3, 4 and 5 for jEdit, JabRef and ArgoUML software projects respectively.

As shown in Figure 3, precision value of 0.6 was maintained across all software projects. JabRef consistently produced the best precision, recall and F-measure values was identified to achieve the best precision with values 0.6, 0.67 and 0.6 at cut points of 10, 15 and 20 methods respectively. ArgoUML trailed with precision values of 0.5, 0.47 and 0.45 at cut points of top 10, 15 and 20 methods respectively while jEdit achieved the lowest precision at cut points of top 10, 15 and 20 methods with values 0.3, 0.27 and 0.25 respectively.

As shown in Figure 4, the best recall value of 0.56 was obtained on ArgoUML when all 20 methods in the EISt were considered while the least recall value was obtained on JabRef at cut point 5 methods with a value of 0.10. jEdit topped the recall chart at cut point 10 methods with a value of 0.33 followed by ArgoUML with a value of 0.31. Both jEdit and ArgoUML achieved recall values of 0.44 at cut point 15 methods while JabRef had a recall value of 0.3.

Figure 5 showed that ArgoUML performed best when the top 10, 15 and 20 methods were considered with values 0.38, 0.45 and 0.5 respectively. jEdit achieved F-measure values of 0.31, 0.33 and 0.34 at cut points 10, 15 and 20 methods respectively while JabRef achieved F-measure of 0.28, 0.41 and 0.45 at the same cut points respectively. However, at cut point of 5 methods, jEdit performed best with a F-measure of 0.43 trailed by
ArgoUML with F-measure of 0.29 and JabRef with F-measure of 0.17

### *4.1.3. Results of comparison EISs, EISt, and EISf*

Experimental analysis revealed distinct strengths and weaknesses for each approach. While textual analysis (EISt) generally achieved marginally higher precision than structural analysis (EISs), structural analysis suffered from very low recall, as it consistently missed many true impacts. The hybrid framework (EISf) leveraged the strengths of the baseline techniques. By combining the impact sets of the baseline techniques, EISf significantly improved recall while maintaining competitive precision. Most importantly, EISf demonstrated the highest F-measure across all three projects, indicating a superior balance between precision and recall and thus the best overall performance. A summary of the results is presented in Table 5 with bold fonts indicating the best values.

Table 5: Comparison of the performance of EISs, EISt, and EISf for all software projects

| **Software project/ CIA technique** | **Pre-cision** | **Re-call** | **F-measure** |
|---|---|---|---|
| jEdit | | | |
| Structural Analysis ($EIS_s$) | 0.33 | 0.11 | 0.17 |
| Textual Analysis ($EIS_t$) | 0.36 | 0.41 | 0.35 |
| Hybrid Framework ($EIS_f$) | **0.37** | **0.53** | **0.41** |
| JabRef | | | |
| Structural Analysis ($EIS_s$) | 0.60 | 0.10 | 0.17 |
| Textual Analysis ($EIS_t$) | **0.62** | 0.24 | 0.33 |
| Hybrid Framework ($EIS_f$) | 0.57 | **0.30** | **0.38** |
| ArgoUML | | | |
| Structural Analysis ($EIS_s$) | 0.50 | 0.13 | 0.21 |
| Textual Analysis ($EIS_t$) | **0.51** | 0.38 | 0.41 |
| Hybrid framework ($EIS_f$) | 0.48 | **0.47** | **0.46** |

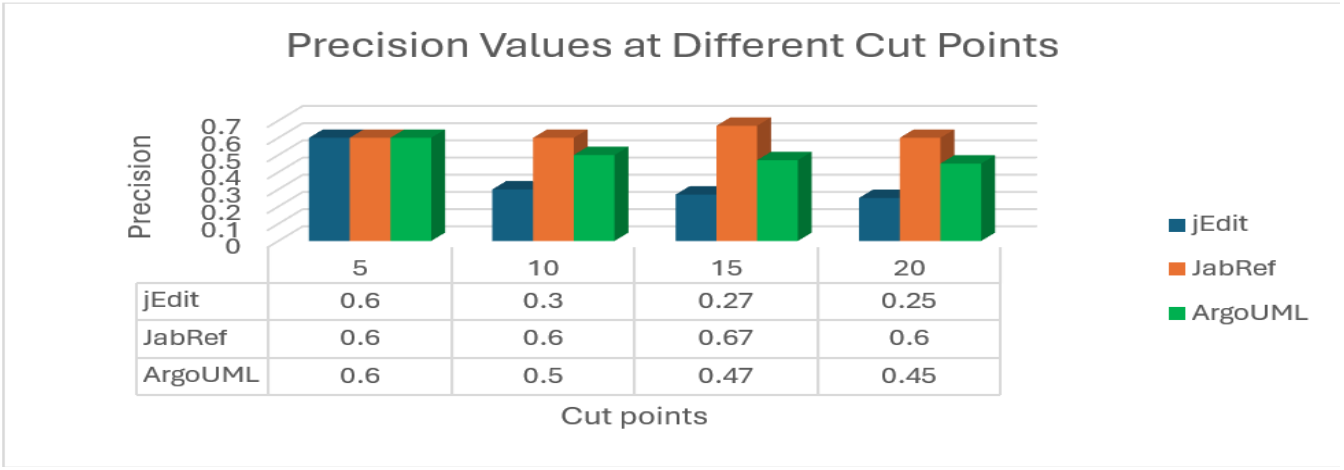


**Figure 3:** Precision for all Experiments at Different Cut Points

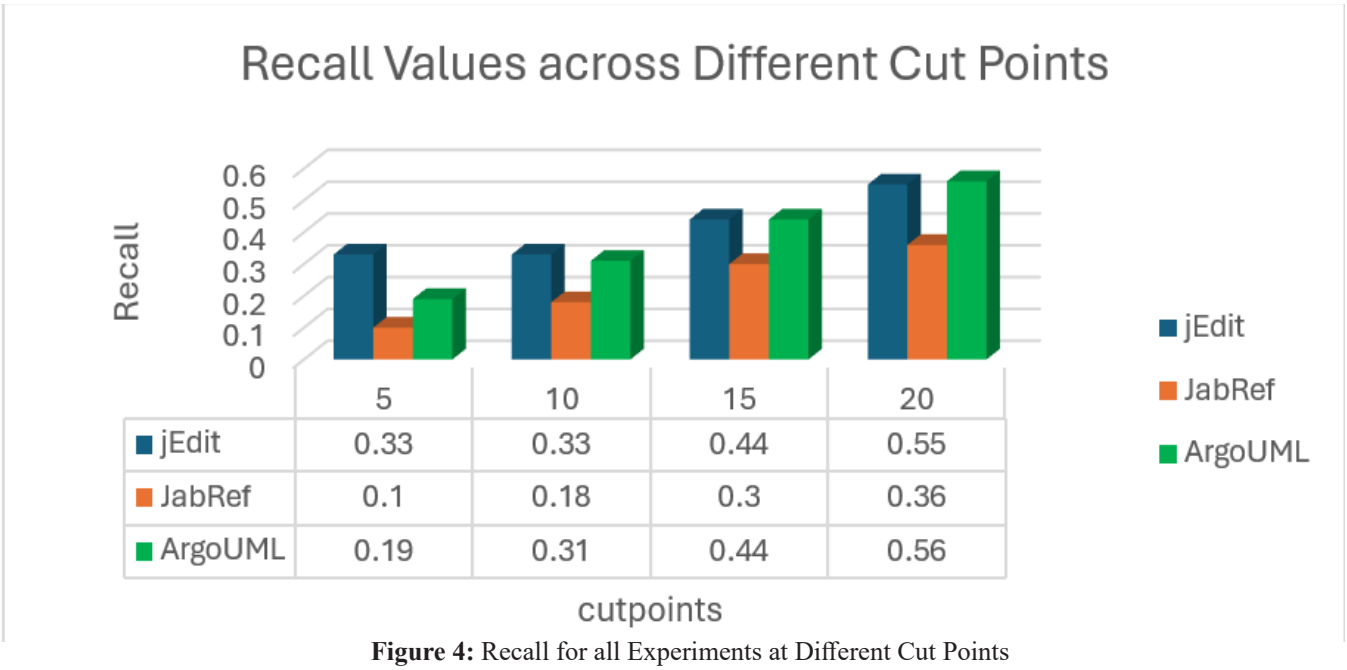


**Figure 4:** Recall for all Experiments at Different Cut Points

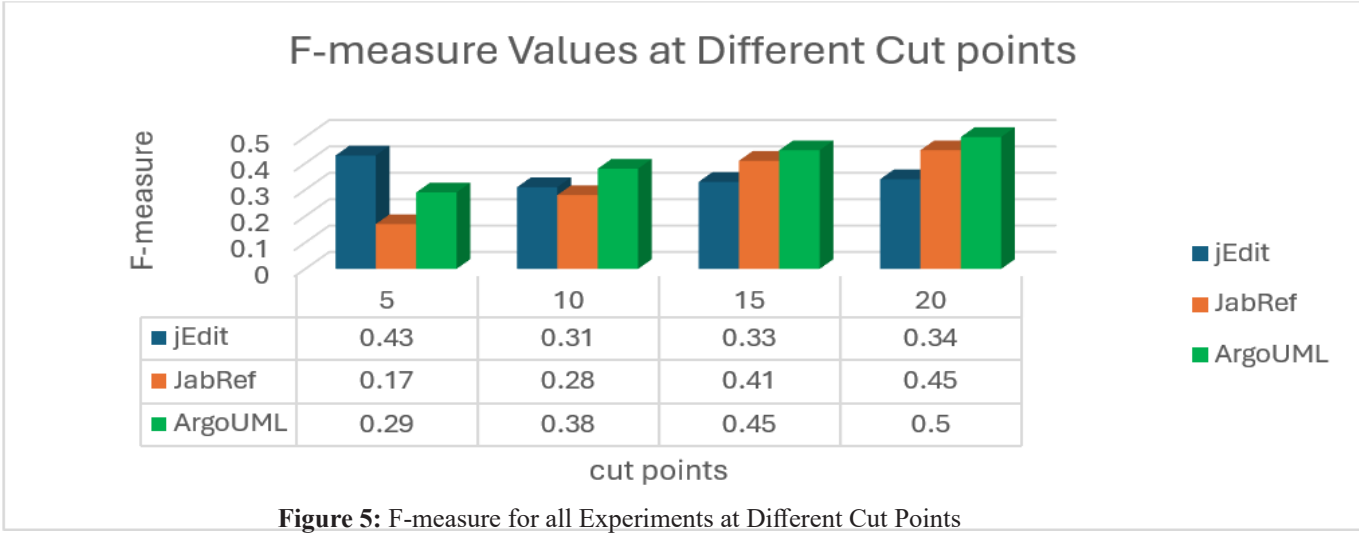


**Figure 5:** F-measure for all Experiments at Different Cut Points

### *4.2. Comparison of the Hybrid Framework with Existing Hybrid Techniques*

The performance of the proposed hybrid framework EISf was compared against two existing hybrid CIA techniques which used similar benchmark datasets, experiments and evaluation metrics. The best precision, recall, and f-measure values in percentages were considered for comparison where available. A summary of the results is shown in Table 3. Gethers et al. (2012) proposed an integrated impact analysis technique based on a combination of LSI model, dynamic analysis and historical analysis for managing software changes. Wang et al. (2018) proposed a hybrid CIA technique based on a combination of the LSI model and Doc2Vec for analysing unstructured source code data. The hybrid framework presented in this study utilized a combination of data-flow analysis and optimised LDA model to address the challenges of analysing a single source code data as well as the shortcomings of the LSI model.

Table 4.27: Performance evaluation of the hybrid framework with existing hybrid

| Evaluation Metrics | **Software Projects** | **Gethers *et al.*, 2012** | **Wang *et al.*, 2018** | **Hybrid Framework ($EIS_f$)** |
|---|---|---|---|---|
| Precision | *jEdit* | 18 | 10 | **57** |
| | *JabRef* | 14 | 24.98 | **67** |
| | *ArgoUML* | 17 | NA | **56** |
| Recall | *jEdit* | **75** | 45 | 42 |
| | *JabRef* | **48** | 24.82 | 42 |
| | *ArgoUML* | 41 | NA | **51** |
| F-measure | *jEdit* | 20 | 20 | **50** |
| | *JabRef* | 14 | 24.90 | **49** |
| | *ArgoUML* | 16 | NA | 51 |

Table 3 compares the performance of the proposed hybrid framework with existing hybrid approaches. Best values are displayed in bold. Columns having NA indicate that the studies did not use the metric for evaluating their technique.

## 5. Discussion

The results presented in section 3.0 provide evidence of the efficacy of combining textual and structural analyses over the baseline techniques, as suggested in Yusuff et al. (2021). The significant improvement in F-measure demonstrates that the hybrid framework was best at addressing the precision-recall trade-off. The strength of the hybrid framework lies in its ability to capture various source code dependencies inherent in different types of source code data. The optimised LDA model proved effective at identifying conceptual dependencies based on shared terminologies in identifiers and comments. This allowed for identifying relevant methods that may not have direct structural links but are functionally related. On the other hand, direct syntactic dependencies like method calls were identified through data-flow analysis. By combining the results of both analyses, the hybrid framework leveraged the orthogonal views of the system, leading to a more complete impact set and a marked improvement in recall. While this combination slightly lowered precision compared to the textual analysis approach in two projects, the gain in recall was substantial to produce a superior F-measure overall.

Comparison with existing hybrid techniques further validates the effectiveness of the hybrid framework, with precision values suggesting that the combination of data-flow analysis and an optimised LDA model is more effective at filtering out irrelevant methods than the combinations used in the compared studies. Notably, the hybrid framework achieved better results at relatively smaller cut-points (top 20 methods), whereas other previous studies mainly relied on much larger sets to achieve peak recall. This is a crucial practical advantage, as it reduces the number of candidate methods a developer must manually inspect, thereby saving time and reducing maintenance costs.

## 6. Conclusion

The study presented a hybrid framework for source code CIA that leverages both structured and unstructured source code information. By combining data-flow analysis and an optimised LDA model, the framework provided a holistic view of software dependencies without adversely impacting maintenance efforts.

Although the hybrid framework demonstrated significant improvements in addressing the precision-recall tradeoff, several limitations are evident. First, the validity of the results could be threatened by the inconsistent grammar used to express unstructured source code information in the form of comments and identifiers across the software projects utilized in the study. Second are the “near-optimal” values obtained from tuning LDA with DE on the software projects dataset. To mitigate the identified threats, the study applied NLP preprocessing techniques previously applied successfully on software engineering data

to help refine the data, thereby making it suitable for analysis. Also, since the LDA model was originally developed for natural language text, the study employed the use of the DE metaheuristic algorithm for tuning LDA hyperparameter configurations on the datasets as opposed to using the default LDA hyperparameter settings. Furthermore, since the experiments were carried out on Java software projects, care was taken to select software projects from different domains and of various sizes.

Consequently, future studies can address the identified limitations by working with other variants of LDA, optimisation algorithms and source code projects written in other programming languages to enhance the model's generalisability.

## References


Abdu, Z. Zhai, H. A. Abdo and R. Algabri (2024) Software Defect Prediction Based on Deep Representation Learning of Source Code From Contextual Syntax and Semantic Graph. in *IEEE Transactions on Reliability*, vol. 73, no. 2, pp. 820-834, June 2024, doi: 10.1109/TR.2024.3354965.

Agrawal, Anushree, and Singh, RK. 2020 Identification of co-change patterns in software evolution. *In Proceedings of the International Conference on Reliability, Infocom Technologies and Optimisation (Trends and Future Directions) (ICRITO)*, 781–785.

Agrawal, A., Fu, W., & Menzies, T. (2018) What is wrong with topic modeling? And how to fix it using search-based software engineering. *Information and Software Technology, 98*(February), 74–88. https://doi.org/10.1016/j.infsof.2018.02.005

Alenezi, M. (2015) Extracting high-level concepts from open-source systems. *International Journal of Software Engineering and Its Applications*, *9*(1), 183–190. https://doi.org/10.14257/ijseia.2015.9.1.16

Alenezi, M., & Magel, K. (2014) Empirical evaluation of a new coupling metric: Combining structural and semantic coupling. *International Journal of Computers and Applications*, *36*(1), 34–44. https://doi.org/10.2316/Journal.202.2014.1.202-3902

Angerer F, Grimmer A, Prähofer H, Grünbacher P. 2019 Change impact analysis for maintenance and evolution of variable software systems. Automated Software Engineering, 26(2), 417–461.

Bajeh AO, Olatunji MA, Oladele RO. 2019 Investigating self-regulation property of evolving open source systems: An empirical study. Journal of Sustainable Technology (JOST), 10(1), 68–76.

Banitaan, S., & Alenezi, M. (2015) Software Evolution via Topic Modeling: An Analytic Study. International Journal of Software Engineering and Its Applications (IJSEIA), 9(5), 43–52. https://doi.org/10.14257/ijseia.2015.9.5.05

Cai H. 2018 Hybrid program dependence approximation for effective dynamic impact prediction. *IEEE Transactions on Software Engineering*, *44*(4), 334–364.

Dai P, Wang Y, Jin D, Gong Y, Yang W. 2022 An improving approach to analysing change impact of C programs. *Computer Communications*, *182*, 60–71.

Gethers, M., Dit, B., Kagdi, H., & Poshyvanyk, D. (2012) Integrated impact analysis for managing software changes. *2012 34th International Conference on Software Engineering (ICSE)*, 430–440. https://doi.org/10.1109/ICSE.2012.6227172

Hattori, L., Guerrero, D., Figueiredo, J., Brunet, J., & Damasio, J. (2008) On the precision and accuracy of impact analysis techniques. Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008), 513–518. https://doi.org/10.1109/ICIS.2008.104

Kagdi, Huzefa, Gethers, M., Poshyvanyk, D., & Collard, M. L. (2010) Blending conceptual and evolutionary couplings to support change impact analysis in source code. *2010 17th Working Conference on Reverse Engineering*, 119–128. https://doi.org/10.1109/WCRE.2010.21

Panichella, A. (2021) A systematic comparison of search-Based approaches for LDA hyperparameter tuning. *Information and Software Technology*, *130*, 106411. https://doi.org/10.1016/j.infsof.2020.106411

Panichella, A. (2019) A Systematic Comparison of Search Algorithms for Topic Modelling—A Study on Duplicate Bug Report Identification. *11th Symposium on Search-Based Software Engineering*, *11664 LNCS*, 11–26. https://doi.org/10.1007/978-3-030-27455-9_2.

Panichella, A., McMillan, C., Moritz, E., Palmieri, D., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2013) When and how using structural information to improve IR-based traceability recovery. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 199–208. https://doi.org/10.1109/CSMR.2013.29.

Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, *14*, 5–32.

Savage, T., Dit, B., Gethers, M., & Posyvanyk, D. (2010) Topic XP: Exploring topics in source code using LDA. *Software Maintenance (ICSM), IEEE International Conference, 2010*, 1–6.

S. R. A. P. Ramu and S. Reddivari, "Change Impact Analysis Using Machine Learning: A Systematic Literature Review," 2025 IEEE International Conference on Information Reuse and Integration and Data Science (IRI), San Jose, CA, USA, 2025, pp. 19-24, doi: 10.1109/IRI66576.2025.00012.

Wang W, He Y, Li T, Zhu J, Liu J. 2018 An integrated model for information retrieval based change impact analysis. *Scientific Programming*, *2018*, 1–13.

Yarnguy, T., & Kanarkard, W. (2018) Tuning Latent Dirichlet Allocation Parameters using Ant Colony Optimisation. Journal of Telecommunication, Electronic and Computer Engineering, 10(1), 21–24.

Yusuff SR, Bajeh AO, Aro TO, Adewole K. 2021 Improving the accuracy of static source code based software change impact analysis through hybrid techniques: a review. *International Journal of Software Engineering and Computer Systems*. *7*(1), 57–66.