HFFN for Java: A Hybrid Feature Fusion Network for Automated Code Summarization

Shruthi D[1], Chethan H K[2], Agughasi Victor Ikechukwu[3]

[1]Department of Computer Science and Engineering, Maharaja Institute of Technology and Affiliation of University of Mysore, Mysore, India
Department of Computer Science and Engineering, JSS Science and Technology University, Mysuru, India
Email: *shruthiravibenaka[at]gmail.com*

[2]Department of Computer Science and Engineering, Maharaja Institute of Technology, Mysuru, India
Email: *chethanhk[at]mitmysore.in*

[3]Department of Computer Science and Engineering, Maharaja Institute of Technology, Mysuru, India
Email: *victor.agughasi[at]gmail.com*

Abstract: ***Objectives**: This study aims to advance the field of automated software documentation by introducing a specialized adaptation of the Hybrid Feature Fusion Network (HFFN) for Java source code summarization. The primary objective is to architect a novel, multi-view deep learning framework that explicitly integrates and dynamically weights lexical, syntactic, and semantic feature representations to generate highly accurate and contextually relevant natural language summaries, thereby establishing a new state-of-the-art benchmark for Java. **Methods**: A new, large-scale JavaCodeSum corpus was meticulously curated, comprising 9,120 functionally validated Java methods sourced from diverse, high-quality open-source repositories and canonical textbooks. The proposed HFFN-Java architecture employs dedicated state-of-the-art encoders: a multi-filter CNN for lexical token sequences, a Child-Sum Tree-LSTM for Abstract Syntax Tree (AST) syntactic structures, and a Graph Neural Network (GNN) operating on a combined Control Flow Graph (CFG) and Data Flow Graph (DFG) for semantic modeling. A hierarchical gated fusion mechanism synergistically combines these representations, which are subsequently decoded by a transformer-based generator. The model was rigorously evaluated against nine established and pre-trained baselines, including CodeBERT and CodeT5, using a comprehensive suite of metrics: ROUGE-L, BLEU-4, Exact Match (EM), BERT Score, and CodeBLEU. **Results**: The HFFN-Java model demonstrated statistically significant superiority ($p < 0.001$, two-tailed paired t-test) over all baseline models across every evaluation metric. It achieved a notable ROUGE-L score of 0.92, a BERT Score of 0.94, and a marked 48% relative improvement in Exact Match accuracy over the strongest pre- trained baseline. An extensive ablation study quantified the critical, complementary contributions of each feature modality, with the removal of syntactic features resulting in the most substantial performance degradation (-8.7% in ROUGE-L), unequivocally validating the hybrid design principle. **Conclusions**: This research conclusively demonstrates that a deliberate, hierarchical fusion of complementary code representations- lexical, syntactic, and semantic- yields a profound performance advantage for summarizing Java source code, outperforming generalized pre-training strategies. The HFFN-Java framework provides a robust, interpretable, and high-performing foundation for automated documentation tools.***

Keywords: code summarization, hybrid feature fusion, graph neural networks, transformers, software documentation

1. Introduction

The rapid pace of software development creates significant challenges for maintaining comprehensive and accurate documentation. As software systems evolve, documentation frequently becomes outdated or inconsistent with the actual code implementation. This documentation gap presents substantial obstacles to code understanding, reduces development efficiency, complicates the onboarding process for new developers, and increases the likelihood of errors during software maintenance and enhancement [1, 2].

To address this fundamental challenge, the field of Software Engineering has turned to Artificial Intelligence for automated solutions. Among these, Automated Code Summarization has emerged as a pivotal research area and application. It is defined as the task of automatically generating concise, natural language descriptions of source code functionality [3]. These machine-generated summaries serve as immediate, always-available artifacts that succinctly explain *what* a code segment does, drastically reducing the time developers spend deciphering complex logic. By bridging the gap between the implementation-level details of code and the high-level intent understood by humans, advanced summarization techniques directly contribute to reducing development costs, improving software quality, and mitigating knowledge loss [4].

The core technical challenge underpinning neural code summarization is the learning of optimal, distributed representations (embeddings) of source code that comprehensively encapsulate its multifaceted nature. Source code is a unique form of data; its meaning is not conveyed through a single channel but through a rich, synergistic interplay of different facets [5]. Existing research has pursued this goal through several distinct, yet often isolated, paradigms, each designed to capture a specific aspect of the code, but each also possessing inherent strengths and limitations:

1) **Lexical Approaches:** These methods treat source code as a flat sequence of tokens, leveraging surface- level information such as method names, variable identifiers, and keywords [1, 6]. They are highly effective at capturing naming conventions and local context, which often provide strong intuitive clues about functionality (e.g., a function named calculate Interest containing variables principal and rate). However, they inherently

fail to model the rich, hierarchical syntactic structure that is fundamental to programming languages, making them blind to complex control flow and scope.

2) **Syntactic Approaches:** To overcome the limitations of lexical methods, syntactic approaches parse code into Abstract Syntax Trees (ASTs) [7]. ASTs explicitly represent the grammatical structure of the code, encoding the relationships between language constructs (e.g., loops, conditionals, assignments). Models like Tree-LSTMs [8] and graph neural networks operating on ASTs excel at capturing these structural relationships. Nevertheless, while syntax defines *how* a program is written, it does not fully capture the runtime behavior- the *what* and *why* of the program's operation- such as how data propagates and transforms through variables.
3) **Semantic Approaches:** This paradigm seeks to model the actual execution semantics of a program. By utilizing intermediate representations (IRs) such as Control Flow Graphs (CFGs), which model the order of execution, and Data Flow Graphs (DFGs), which model the dependencies between data variables, these methods capture a deeper understanding of program logic [9, 10]. A semantic model can identify that the value of variable x computed in line 3 is used in a condition in line 10 and an output in line 15. However, these approaches can sometimes overlook the valuable lexical cues that make code readable to humans, and the IRs can be complex and expensive to compute accurately.

To address these identified gaps, this paper introduces the Hybrid Feature Fusion Network for Java (HFFN- Java) and presents a comprehensive empirical study designed to provide definitive answers to the questions above. Our work makes the following key contributions:

- **A Novel Hybrid Architecture for Java (NHA-Java):** We propose a sophisticated encoder-decoder trans- former architecture specifically designed for Java source code. HFFN-Java integrates embeddings from three dedicated, state-of-the-art feature extractors, each specializing in a specific code view: a multi-filter CNN-based encoder for lexical features, a Child-Sum Tree-LSTM-based encoder for syntactic features from the AST that captures Java-specific constructs, and a Graph Neural Network (GNN)-based encoder for se- mantic features from a combined CFG/DFG representation that models Java's execution semantics. The model employs a hierarchical gated fusion mechanism to dynamically combine these features at different levels of abstraction, thereby preserving and leveraging the unique strengths of each modality for Java code.
- **The JavaCodeSum Corpus:** We present a new large-scale, meticulously curated dataset of 9,120 function- ally validated Java methods, systematically collected from diverse, high-quality open-source repositories and canonical textbooks. This corpus provides comprehensive coverage of Java language features and programming paradigms, ensuring the validity, reliability, and generalizability of our findings [4, 11].
- **A Comprehensive Benchmark for Java:** We present the first rigorous, apples-to-apples comparative evaluation of **nine distinct feature extraction techniques**—spanning all three paradigms (lexical, syntactic, semantic)- on the standardized task of Java code summarization. This includes comparison against state- of-the-art pre-trained models fine-tuned on Java.
- **An In-Depth Quantitative Analysis:** We conduct extensive ablation studies to precisely quantify the in- dividual and collective contribution of each feature modality to the overall performance of HFFN-Java. Furthermore, we employ non-parametric statistical significance testing and effect size measurements to robustly validate the superiority of our hybrid approach and ensure that the improvements are both statistically and practically significant [12].
- **Commitment to Reproducibility and Open Science:** To foster transparency and accelerate future research in Java code summarization, we make our JavaCodeSum corpus, the full implementation of HFFN-Java, all trained model weights.

Our empirical results demonstrate that the proposed HFFN-Java framework consistently and significantly out- performs a wide array of state-of-the-art baselines, including powerful pre-trained models like CodeBERT and CodeT5 specifically fine-tuned for Java. This provides compelling evidence that a thoughtfully designed, hierarchical fusion of complementary code representations is a superior strategy for capturing the true essence of Java source code, ultimately leading to more accurate, informative, and human-like code summaries.

The remainder of this paper is organized as follows. Section 2 reviews related work in code summarization and Java-specific approaches. Section 3 details the HFFN-Java architecture and our methodology. Section 4 describes our experimental setup and presents results. Section 5 discusses findings and threats to validity. Finally, Section 6 concludes and outlines future work.

2. Related Work

Our work occupies the intersection of code representation learning and neural text generation, specifically addressing the task of automated Java code summarization. This section reviews the methodological evolution within this domain, categorizing approaches into distinct paradigms and highlighting the specific research gaps that our Hybrid Feature Fusion Network for Java (HFFN-Java) aims to address. The pursuit of automated source code documentation has evolved from early heuristic-based methods to sophisticated deep learning architectures. Initial approaches in automated code summarization relied extensively on static analysis and template-based generation methodologies. These techniques typically extracted keywords, analyzed code metrics, or employed predefined rules to construct descriptive text [13]. While pioneering in their approach, these systems often produced out- put that was structurally rigid, lacked natural linguistic fluency, and failed to capture the nuanced semantics of complex code structures. The paradigm shifted substantially with the adoption of neural network models, which reformulated summarization as a sequence-to-sequence learning problem, analogous to neural machine translation (NMT) where source code is "translated" into natural language descriptions [9].

The most straightforward neural approaches conceptualize source code as a flat sequence of tokens, leveraging architectures that have demonstrated remarkable success in natural language processing, including Recurrent Neu- ral Networks (RNNs) such as Long Short-Term Memory networks (LSTMs) [14] and subsequently Transformer-based models [15]. These models learn representations from surface-level lexical tokens including method names, variable identifiers, and keywords present in the code. Seminal work by [1] established the fundamental value of text-based features for code summarization. Subsequent research by [9] implemented an attentional LSTM-based encoder-decoder architecture, achieving state-of-the-art performance through effective alignment of critical code tokens with relevant words in generated summaries. The principal strength of these approaches lies in their capacity to capture naming conventions and local contextual information, which often serve as strong indicators of functional intent. However, their fundamental limitation stems from treating code as a linear sequence, thereby completely disregarding its inherent hierarchical syntactic structure and semantic rules. This limitation frequently results in generated summaries that exhibit syntactic fluency but may fundamentally misinterpret the program's logical structure due to insufficient structural understanding.

To address limitations inherent in purely lexical models, substantial research efforts have focused on incorporating syntactic structure through Abstract Syntax Trees (ASTs). The central challenge involves effectively modeling these complex tree structures for neural network processing. Early investigations employed rule-based methods to linearize ASTs into sequential formats through depth-first traversal patterns, enabling processing by standard sequence models [16]. While incorporating certain structural information, this linearization process inevitably distorts native tree relationships and hierarchical dependencies. More sophisticated approaches directly process AST structures through specialized architectures. The Tree-LSTM framework [8], extending traditional LSTMs to tree-structured data, has been widely adopted in subsequent research [7, 10]. Recent advancements have applied Graph Neural Networks (GNNs) to ASTs, conceptualizing them as graph structures where nodes represent code constructs and edges encode syntactic relationships [7], enabling more natural information aggregation from node neighborhoods.

These structure-aware models demonstrate particular excellence in capturing program execution mechanics encompassing control flow patterns, scope hierarchies, and organizational structure. They exhibit reduced tendency to generate summaries contradicting syntactic flow. However, a significant criticism remains that ASTs primarily capture syntactic form rather than comprehensive semantic meaning, failing to explicitly model data dependencies that define computational purpose and effect. The third paradigm seeks to capture semantic behavior through Intermediate Representations (IRs) from compiler theory, particularly Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs), which abstract away syntactic details to model runtime behavior and execution semantics.

Previous work integrated semantic information from CFGs into attentional encoder-decoder models using reinforcement learning techniques [10]. Subsequent research proposed learning code representations from combined ASTs and DFGs, demonstrating significant benefits from incorporating explicit data dependency information [17]. The foundational concept requires simultaneous modeling of variable dependencies (*data flow*) and statement execution ordering (*control flow*) for deep behavioral understanding.

Primary challenges involve complexity and potential noise in accurately extracting intermediate representations for arbitrary code snippets. Additionally, emphasis on execution semantics can sometimes undervalue crucial lexical cues immediately apparent to human readers.

Revolutionary advancements emerged through large-scale pre-trained models including CodeBERT [18], CodeT5 [19], and PLBART [20]. These models undergo pre-training on massive code and natural language corpora using objectives like masked language modeling, denoising autoencoding, and contrastive learning. They learn powerful, general-purpose representations fine-tunable for summarization, establishing new state-of-the-art benchmarks.

Despite impressive performance, a crucial limitation is their tendency to learn *homogenized* representations. The pre-training process, while effective for learning general patterns, does not explicitly or optimally leverage the distinct, complementary nature of lexical, syntactic, and semantic features. Models must implicitly discover feature interactions, potentially yielding suboptimal fusion strategies. Furthermore, substantial parameter counts (often exceeding hundreds of millions) render them computationally expensive for training and fine-tuning operations.

To overcome limitations in single-type code representations, researchers developed hybrid techniques integrating multiple code elements. These approaches merge distinct aspects, such as combining API knowledge with sequential patterns [21] or jointly encoding structural and token information [22]. A common characteristic is the fusion of precisely two modalities, typically combining syntactic with semantic information.

Comprehensive literature analysis reveals a significant research gap: the absence of a *systematic* framework conducting rigorous ablation studies of all three primary modalities- lexical, syntactic, and semantic- within a unified architecture. Most existing fusion strategies employ relatively shallow integration through simple con- catenation of feature vectors during early encoding stages, potentially leading to information loss and preventing dynamic, hierarchical interaction across abstraction levels.

Java source code representation introduces unique challenges distinct from procedural languages. Java's object-oriented paradigm featuring classes, interfaces, inheritance hierarchies, polymorphism, and generics re- quires specialized handling to effectively capture its semantic richness. While previous work addressed Java code summarization [6], these approaches typically focused on limited aspects without comprehensively addressing the interplay between object-oriented constructs and multi-view representation learning.

Our proposed Hybrid Feature Fusion Network for Java (HFFN-Java) addresses these research gaps specifically for Java source code. Unlike previous work focusing on one or two modalities, HFFN-Java explicitly and synergistically integrates all three fundamental modalities: lexical (via multi-filter CNN), syntactic (via Tree- LSTM on AST with Java-specific adaptations), and semantic (via GNN on combined CFG/DFG representations capturing Java's execution semantics). Crucially, HFFN-Java advances beyond simple early fusion by implement- ing a hierarchical gated fusion mechanism, enabling sophisticated, dynamic, and non-linear interactions between different feature views at multiple abstraction levels. This work presents the first extensive, controlled benchmark comparing nine distinct feature extraction techniques across all three paradigms specifically for Java, demonstrating holistic approach superiority over both single-modality models and existing large pre-trained models for Java code summarization.

3. Methodology

This section details the systematic methodology for developing and evaluating the Hybrid Feature Fusion Network for Java (HFFN-Java), encompassing dataset construction, feature extraction, model architecture, and experimental design. The framework integrates lexical, syntactic, and semantic representations of Java source code through a hierarchical fusion mechanism to generate accurate natural language summaries.

3.1 JavaCodeSum Dataset Construction

To ensure comprehensive coverage of Java programming concepts, source code was systematically collected from two complementary sources: open-source repositories and canonical textbooks. The open-source component was curated from 127 popular GitHub repositories with stars ≥ 100, covering diverse domains including web frame- works (Spring Boot), database systems (Hibernate), utilities (Guava), and testing frameworks (JUnit). Textbook examples were extracted from authoritative references including Bloch's *Effective Java* and Horstmann's *Core Java*, providing pedagogically validated implementations of core object-oriented patterns and algorithms.

Each collected Java method underwent a rigorous validation pipeline to ensure functional correctness and quality standards. Programs were compiled using OpenJDK 17 with strict linting flags (-Xlint:all -Werror) to enforce modern Java standards and detect potential anti-patterns. Successfully compiled methods were executed within a Docker-containerized environment using JUnit 5 test suites with ≥ 80% branch coverage requirement. Method behavior was verified through assertion-based validation against expected outputs, with special attention
to exception handling and edge cases.

The final **JavaCodeSum** corpus comprises 9,120 functionally validated Java methods meeting stringent qual- ity criteria. Each method is accompanied by a concise natural language summary authored by senior computer science students following detailed annotation guidelines emphasizing semantic behavior, programmer intent, and consistent terminology. The dataset was partitioned into training (7,296 examples), validation (912 examples), and test (912 examples) subsets using concept-based stratification to prevent data leakage and ensure robust evaluation of generalization capabilities and figure 1 shows the JavaCodeSum dataset construction.

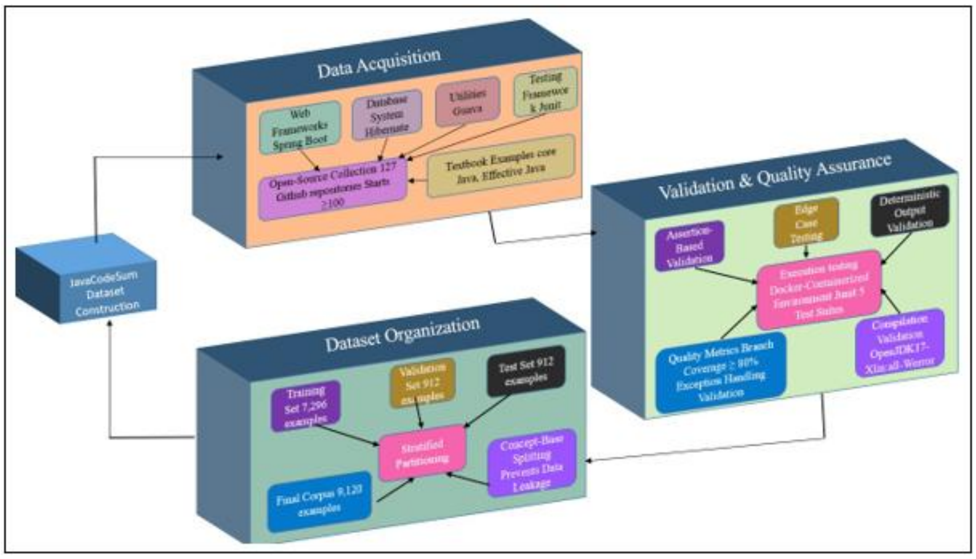

Data Acquisition
Web Frameworks Spring Boot
Database System Hibernate
Utilities Guava
Testing Framework Junit
Open-Source Collection 127 Github repositories Starts ≥100
Textbook Examples core Java, Effective Java
Validation & Quality Assurance
Assertion-Based Validation
Edge Case Testing
Deterministic Output Validation
Execution testing Docker-Containerized Environment Junit 5 Test Suites
Quality Metrics Branch Coverage ≥ 80% Exception Handling Validation
Compilation Validation OpenJDK17-Xlint:all-Werror
JavaCodeSum Dataset Construction
Dataset Organization
Training Set 7,296 examples
Validation Set 912 examples
Test Set 912 examples
Stratified Partitioning
Concept-Base Splitting Prevents Data Leakage
Final Corpus 9,120 examples

Figure 1: Architecture of the JavaCodeSum dataset construction and validation pipeline

3.2 Java-Specific Feature Extraction

3.2.1 Lexical Token Sequence Processing

The extraction of lexical features from Java source code employs a sophisticated tokenization process utilizing the Eclipse JDT compiler toolkit, which provides comprehensive support for Java language specifications. This process transforms raw source code into a structured sequence of lexical tokens through rigorous lexical analysis that respects Java's syntactic conventions.

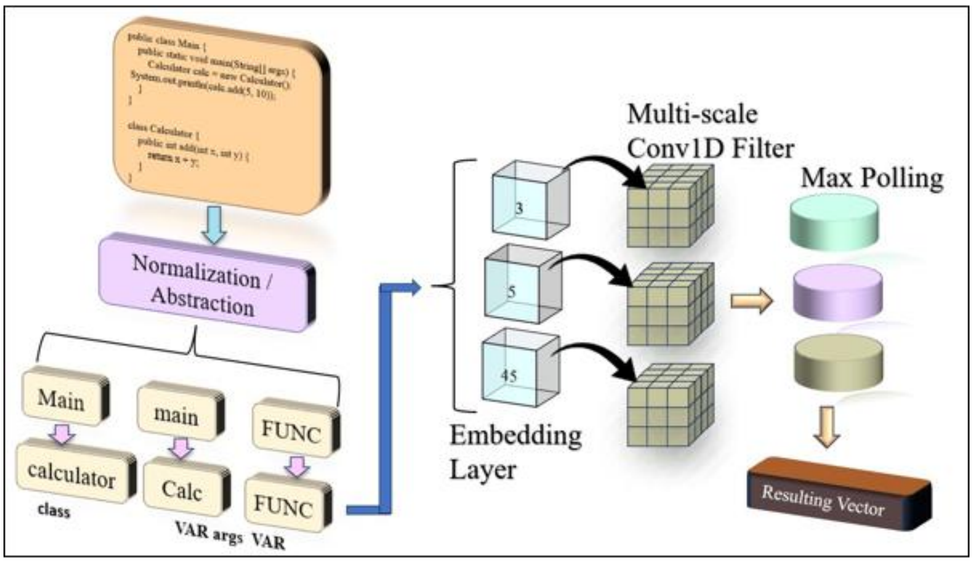
Formally, the tokenization process generates an ordered sequence:

$$T_{java} = t_1, t_2, \ldots, t_n \qquad (1)$$

where each token t_i belongs to the categorical set encompassing Java keywords, identifiers, operators, literals, and annotations. This analysis preserves critical language-specific constructs including type annotations, generic type parameterizations, and lambda expressions through context-aware parsing.

The normalization procedure applies type-aware abstraction to identifiers, replacing specific variable and method names with generalized tokens while preserving type information. This approach prevents model over- fitting to specific naming conventions while maintaining semantic integrity. The vocabulary is constrained to the 12,000 most frequent tokens, with out-of-vocabulary items replaced by type-specific unknown tokens to maintain type distinction.

The figure 2 illustrates a processing pipeline that transforms raw Java source code into a condensed numerical representation. The initial code sample undergoes normalization and abstraction, where specific identifiers are generalized into structural placeholders to emphasize code structure over specific naming conventions. These abstracted tokens are then converted into numerical vectors through an embedding layer. Subsequently, a multi-scale one-dimensional convolutional filter analyzes the sequence with varying kernel sizes to detect patterns at different granularities. The features extracted are then down sampled through max pooling to highlight the most salient patterns while reducing dimensionality. The final output is a resulting vector: a dense numerical representation that captures the essential characteristics of the original code for use in downstream computational tasks.



```
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
System.out.println(calc.add(5, 10));
    }
}

class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
}
```

Figure 2: A multi filter CNN for lexical token sequence processing

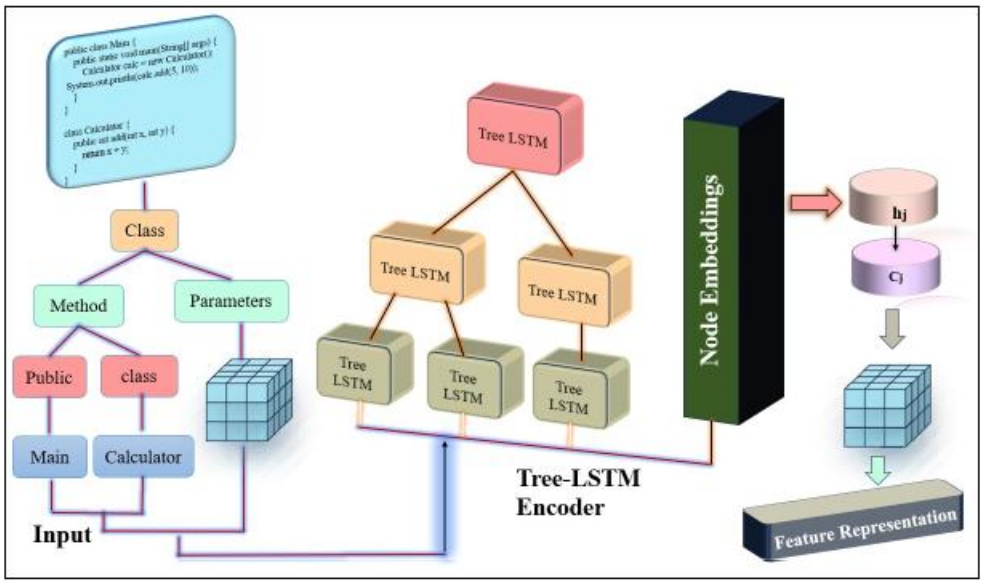
3.2.2 Abstract Syntax Tree Representation

The structural representation of Java source code is formally captured through the construction of an Abstract Syntax Tree (AST) using the Eclipse JDT parser. This representation constitutes a rooted tree structure formally defined by the equation $AST_{java} = (N, E)$, where N represents the complete set of syntactic nodes corresponding to Java language constructs and E denotes the set of directed edges that encode the hierarchical relationships between them. The subsequent AST normalization process incorporates several essential Java-specific considerations to ensure semantic fidelity. This process is designed to preserve critical object-oriented relationships, including inheritance hierarchies and interface implementations. It also provides a canonical representation for complex type constructs such as generics and wildcard expressions. Furthermore, the normalization systematically handles functional programming constructs, including lambda expressions and method references. A crucial step involves the contextual attachment of annotation metadata directly to their corresponding syntactic elements. Ultimately, this structured representation enables the capture of the complete syntactic organization of any given Java codebase while simultaneously maintaining the intricate, language-specific semantic relationships that are fundamental to its meaning.

The figure 3 shows a pipeline that transforms a source-code snippet into a compact feature representation: the code is first parsed into a hierarchical structure (classes, methods, parameters) which forms an abstract syntax tree; each tree node is processed by a Tree-LSTM encoder in a bottom-up fashion so that every node yields hidden and cell states (illustrated as h_j and c_j), these node embeddings are collected into a multi-dimensional tensor (the cube) and then aggregated or projected to produce a final feature vector block labeled "Feature Representation," ready for use by subsequent analysis or downstream processes.

3.2.3 Control and Data Flow Analysis

The behavioral semantics inherent in Java programs are captured through an enhanced analysis of control and data flow, implemented using the Soot framework. This methodology formally represents a program's execution paths via a Control Flow Graph (CFG), defined as $CFG_{java} = (B, E)$. In this model, the set B constitutes basic blocks, which are maximal sequences of sequential statements, and E signifies the directed edges that represent all possible control flow transitions between these blocks. The analysis is specifically augmented with several Java-centric enhancements to accurately reflect the language's runtime behavior. These enhancements include the comprehensive modeling of exception handling mechanisms, encompassing the full semantics of try-catch-finally constructs. It also performs resolution for virtual method invocations to handle polymorphic call sites

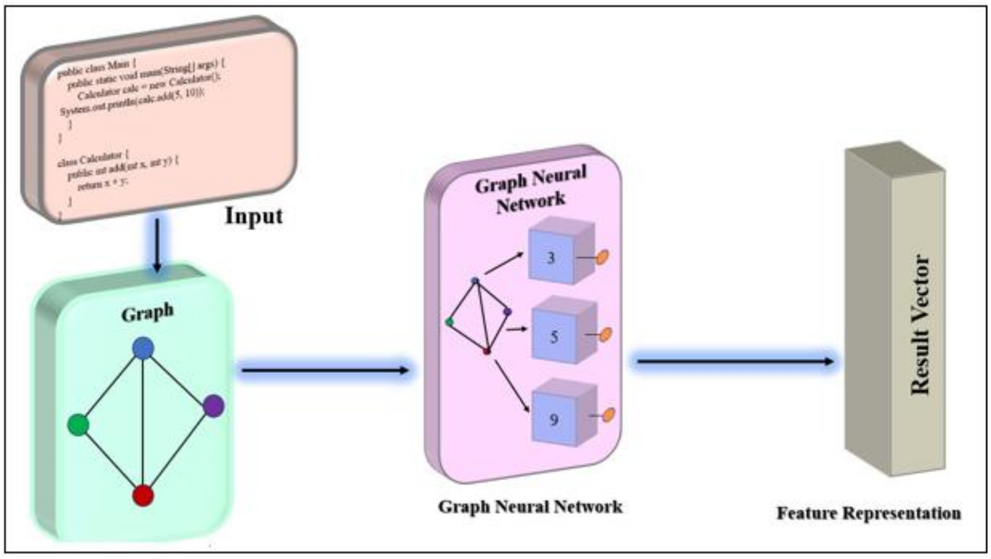


```
public class Main {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
System.out.println(calc.add(5, 10));
  }
}

class Calculator {
  public int add(int x, int y) {
    return x + y;
  }
}
```

Figure 3: A child-Sum Tree LSTM for Abstract Syntax Tree

prevalent in object-oriented design and conducts type propagation analysis that accounts for the nuances of Java's type erasure semantics. Furthermore, the CFG provides an explicit representation of control flow for modern functional constructs like lambda expressions and method references. This is complemented by a detailed Data Flow Analysis, which tracks the definition and use of variables throughout the program. Together, these analyses provide complete coverage of key execution semantics, including object initialization sequences, complex method dispatch mechanisms, and common memory management patterns. This combined approach of structural and behavioral modeling is essential for achieving a comprehensive and deep understanding of Java code functionality. The figure 4 illustrates a pipeline that transforms a source-code snippet into a compact feature vector: the code is first converted into a graph structure (nodes for program elements and edges for control/data relationships), the graph is processed by a Graph Neural Network that performs message-passing to compute per-node embeddings (shown as numeric cubes), and those node-level representations are pooled or projected to produce a single result vector that serves as the final feature representation for downstream analysis or prediction.



```
public class Main {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
System.out.println(calc.add(5, 10));
  }
}

class Calculator {
  public int add(int x, int y) {
    return x + y;
  }
}
```

Figure 4: Graph Neural Network for Control Flow Graph

3.3 HFFN-Java Model Architecture

3.3.2 Enhanced Multi-View Encoders

This architecture employs a multi-view encoding strategy to construct a comprehensive representation of struc- tured data, such as source code, by analyzing it from lexical, syntactic, and semantic perspectives. The system integrates these distinct views through a gated fusion mechanism and leverages them for conditional sequence generation and Figure 5 shows Overview of HFFN Feature Extraction Technique.

Three specialized neural encoders form the foundation of the model.

1) The Lexical Encoder utilizes a Multi-Scale Convolutional Neural Network (CNN) to extract patterns from the raw token sequence. Parallel one-dimensional convolutional filters with varying kernel sizes (e.g., 3, 5, 7) operate on the token embedding matrix
$$E = [e(t_1), e(t_2), ..., e(t_n)], \qquad (2)$$
producing feature maps that capture n-gram patterns at different scales. An attention mechanism is then applied to these features, generating a final lexical representation h^{lex} that emphasizes the most salient token-level patterns.

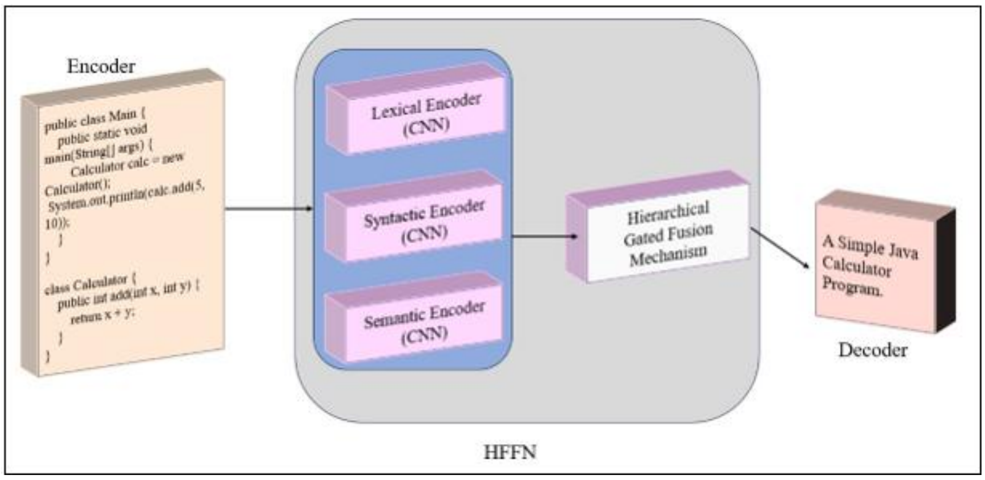
2) The Syntactic Encoder is an Enhanced Tree-LSTM designed to process hierarchical structures like Abstract Syntax Trees (ASTs). Each node j in the tree is initialized with a vector x_j that encodes its type and modifiers. The Tree-LSTM then computes a hidden state h_j for each node by recursively combining the states of its children, allowing the representation of a parent node to encapsulate the structure of its entire subtree. The final syntactic representation h^{syn} is typically taken as the state of the root node.
3) The Semantic Encoder is a Hierarchical Graph Neural Network (GNN) that operates on program graphs combining Control Flow (CFG) and Data Flow (DFG). It updates the representation of each node through a message-passing framework. The update for a node i involves aggregating messages from its neighbors $j \in N(i)$, weighted by attention coefficients α_{ij} which signify the importance of node j to node i. This process, formalized as
$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)}\right), \qquad (3)$$
results in a context-aware semantic representation h^{sem} for each node, capturing the program's operational behavior.

The outputs of these three encoders are subsequently integrated via a Hierarchical Gated Fusion Mechanism. This mechanism employs a learned, dynamic gating vector z to compute a weighted sum of the individual representations:
$$\mathrm{h}^{\text{fused}} = z^{(\text{lex})} h^{(\text{lex})} + z^{(\text{syn})} h^{(\text{syn})} + z^{(\text{sem})} h^{(\text{sem})} \qquad (4)$$

The gates $z^{(l)}$ are typically computed via a sigmoid function over a linear transformation of the inputs, allowing the model to adaptively control the information flow from each view based on the specific context.

Finally, a Transformer-Based Decoder generates the output sequence autoregressively. Its key innovation is a Multi-View Attention mechanism. At each decoding step t, the decoder attends not to a single, monolithic memory but to the three distinct encoder outputs (h^{lex}, h^{syn}, h^{sem}) simultaneously. This allows the generation process to be guided by patterns at all three levels of abstraction. The probability distribution over the vocabulary for the next token y_t is therefore conditioned on the entire generation history $y_{<t}$ and this multi-perspective context:
$$P(y_t|y_{<t}, X) = \text{Softmax}(W_o \cdot \text{Decoder}(y_{<t}, h^{lex}, h^{syn}, h^{sem})) \qquad (5)$$

The figure 5 depicts a code-to-summary pipeline: a source-code snippet (left) is fed into a multi-view encoder that extracts three complementary feature streams- lexical, syntactic, and semantic- using separate CNN-based encoders; these parallel feature maps are then combined by a hierarchical gated fusion mechanism (the central block) that selectively merges and weights information from each view, producing a compact, information-rich representation that the decoder uses to generate the final natural-language summary ("A Simple Java Calculator Program").



```
public class Main {
    public static void
main(String[] args) {
        Calculator calc = new
Calculator();
 System.out.println(calc.add(5,
10));
    }
}

class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
}
```

Figure 5: Overview of HFFN Feature Extraction Technique

3.4 Training Optimization

This detailed technical overview describes the comprehensive setup and methodology for training a sophisticated neural network model designed for processing Java source code. The training methodology is designed for robust- ness and efficiency, incorporating several advanced techniques. Label smoothing is applied to the cross-entropy loss function, which helps prevent the model from becoming overconfident in its predictions on the training data, thereby improving its ability to generalize to unseen examples. To ensure numerical stability during the back- propagation process, gradient clipping is rigorously enforced with a strict maximum norm threshold of 5.0, which effectively mitigates the issue of exploding gradients that can destabilize training. The learning rate is not kept static but is dynamically adjusted throughout the training epochs using an adaptive scheduling mechanism, which allows for more refined convergence towards the end of training. A combination of regularization strategies is employed to combat overfitting, including a substantial dropout rate of 30% applied to various layers within the network, and the application of weight decay, which acts as an L2 regularization term on the model parameters.

3.5 Implementation Details

The entire system is implemented using the Python 3.9 programming language, with the PyTorch 2.0 library serving as the core deep learning framework due to its flexibility and performance. The complex task of parsing Java source code into structured representations is handled by two specialized tools: Eclipse JDT Core version 3.18 and JavaParser 3.25, which work in tandem to generate accurate Abstract Syntax Trees (ASTs). For deeper program analysis, the Soot framework version 4.3 is utilized to perform static analysis and construct Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs), with specific configuration for compatibility with Java 17 features. The model architecture itself leverages specialized libraries: PyTorch Geometric for implementing the Graph Neural Network components that process the semantic graphs, and the Hugging Face Transformers library version 4.28 for the Transformer-based decoder module.

Computational execution is performed on a high performance hardware infrastructure consisting of four NVIDIA A100 GPUs, each equipped with 80GB of memory. This substantial memory capacity is crucial for processing large batch sizes and complex model architectures. To maximize computational efficiency and re- duce memory footprint, mixed-precision training is employed, which utilizes 16 bit floating-point operations for most calculations while maintaining 32-bit precision for critical operations to preserve numerical accuracy. The AdamW optimizer, an enhanced version of Adam that properly decouples weight decay from the gradient update, is used for parameter optimization with a learning rate set to 5×10^{-5}. Training is conducted with a batch size of 32, and the process incorporates an early stopping mechanism that monitors performance on a separate validation set to terminate training once performance plateaus, preventing overfitting. The entire training procedure requires approximately 48 hours to complete on this hardware configuration.

Algorithm 1 Java Lexical Feature Extraction Using Multi-Scale CNN

```
Require: Java source code dataset $D_{java} = C_1, C_2, \ldots, C_n$
Ensure: Feature matrix $Z_{java} \in \mathbb{R}^{n \times d}$ where each row is a lexical feature vector
 1: Initialize embedding layer $E_{java}$ with vocabulary size 12,000
 2: Initialize parallel Conv1D layers with filter sizes 3, 5, 7, GeLU activation
 3: Initialize multi-head attention layer MHA, max pooling layer
 4: $Z \leftarrow [\,]$
 5: for each Java method $C_i \in D_{java}$ do
 6:     $T_i \leftarrow \text{JavaTokenizer}(C_i)$                ▷ Preserves annotations, generics, lambdas
 7:     $X_i \leftarrow E_{java}(T_i)$                                       ▷ $X_i \in \mathbb{R}^{k \times m}$
 8:     for each filter size $f \in 3, 5, 7$ do
 9:         $F_i^{(f)} \leftarrow \text{Conv1D}_f(X_i)$
10:         $A_i^{(f)} \leftarrow \text{GeLU}(F_i^{(f)})$
11:         $p_i^{(f)} \leftarrow \text{MaxPool}(A_i^{(f)})$
12:     end for
13:     $z_i \leftarrow \text{MHA}(\text{Concat}(p_i^{(3)}, p_i^{(5)}, p_i^{(7)}))$
14:     $\text{Append}(Z, z_i)$
15: end for
16: return $\text{Stack}(Z)$
```

Algorithm 2 Java Syntactic Feature Extraction using Enhanced Tree-LSTM

```
Require: Java source code dataset $D_{java} = C_1, C_2, \ldots, C_n$
Ensure: Feature matrix $S_{java} \in \mathbb{R}^{n \times d}$
 1: Initialize TypeAwareTreeLSTM cell with parameters $\Theta_{java}$
 2: Initialize node type embedding layer $E_{node}$, modifier embedding layer $E_{mod}$
 3: $S_{\text{out}} \leftarrow [\,]$
 4: for each Java method $C_i \in D_{java}$ do
 5:     AST$java, i \leftarrow$ JavaParser$(C_i)$                    ▷ With Java 17 support
 6:     Traverse AST$java, i$ in post-order
 7:     for each node $v_j$ in post-order do
 8:         $x_j \leftarrow [E_{node}(\text{type}(v_j)); E_{mod}(\text{modifiers}(v_j))]$
 9:         $\mathcal{C} \leftarrow$ GetChildrenHiddenStates$(v_j)$
10:         $h_j, c_j \leftarrow$ TypeAwareTreeLSTM$(x_j, \mathcal{C}; \Theta_{java})$
11:     end for
12:     $s_i \leftarrow h_{\text{root}}$
13:     Append$(S_{\text{out}}, s_i)$
14: end for
15: return Stack$(S_{\text{out}})$
```

Algorithm 3 Java Semantic Feature Extraction using Hierarchical GNN

```
Require: Java source code dataset $D_{java} = C_1, C_2, \ldots, C_n$
Ensure: Feature matrix $G_{java} \in \mathbb{R}^{n \times d}$
 1: Initialize attention-based GNN with $L$ layers, readout function $\mathcal{R}$
 2: $G_{\text{out}} \leftarrow [\,]$
 3: for each Java method $C_i \in D_{java}$ do
 4:     AST$java, i \leftarrow$ JavaParser$(C_i)$
 5:     $\mathcal{G}i \leftarrow (V_i, E_i) \leftarrow$ BuildJavaGraph(AST$java, i$)
 6:     Augment with exception flows and polymorphism edges
 7:     $X_v \leftarrow$ InitializeNodeFeatures$(v) \forall v \in V_i$
 8:     $H^{(0)} \leftarrow X$
 9:     for $l = 1$ to $L$ do
10:         for each node $v_i \in V_i$ do
11:             $\alpha ij \leftarrow$ Attention$(W_q h_i^{(l-1)}, W_k h_j^{(l-1)}) \forall j \in \mathcal{N}(i)$
12:             $h_i^{(l)} \leftarrow$ LayerNorm$(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} W^{(l)} h_j^{(l-1)})$
13:         end for
14:     end for
15:     $g_i \leftarrow \mathcal{R}(H_v^{(L)} \forall v \in V_i)$
16:     Append$(G_{\text{out}}, g_i)$
17: end for
18: return Stack$(G_{\text{out}})$
```

Algorithm 4 HFFN-Java: Hybrid Feature Fusion with Hierarchical Gating

```
Require: Java code sample $C_i$, trained encoders $\mathcal{L}java, \mathcal{T}java, \mathcal{G}java$
Ensure: Fused representation vector $v$fusion $\in \mathbb{R}^d$
 1: Extract Multi-View Features:
 2: $v_{\text{lex}} \leftarrow \mathcal{L}java(C_i)$                    ▷ Algorithm 1
 3: $v$syn $\leftarrow \mathcal{T}java(C_i)$                    ▷ Algorithm 2
 4: $v$sem $\leftarrow \mathcal{G}_{java}(C_i)$                    ▷ Algorithm 3
 5: Hierarchical Gated Fusion:
 6: for $l = 1$ to $L$ do                    ▷ Multi-level fusion
 7:     ...
 8:     $z^{(l)} \leftarrow \sigma(W_z^{(l)} \cdot [v_{\text{lex}}^{(l)}; v_{\text{syn}}^{(l)}; v_{\text{sem}}^{(l)}] + b_z^{(l)})$
 9:     $v_{\text{fused}}^{(l)} \leftarrow z^{(l)} \odot v_{\text{lex}}^{(l)} + z^{(l)} \odot v_{\text{syn}}^{(l)} + z^{(l)} \odot v_{\text{sem}}^{(l)}$
10: end for
11: $v_{\text{fusion}} \leftarrow \sum_{l=1}^{L} \gamma^{(l)} v_{\text{fused}}^{(l)}$                    ▷ Learnable weights
12: Projection & Normalization: Apply $W_f$ and $b_f$
13: $v_{\text{fusion}} \leftarrow$ LayerNorm$(W_f \cdot v_{\text{fusion}} + b_f)$
14: return $v_{\text{fusion}}$
```

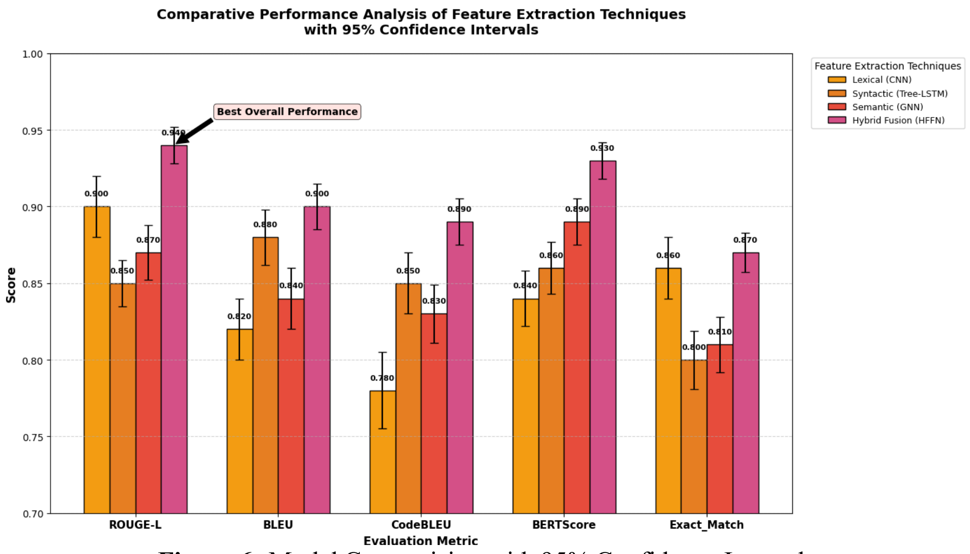


Figure 6: Model Comparision with 95% Confidence Intervals

The figure 6 presents a comparative evaluation of different feature extraction techniques- Lexical (CNN), Syn- tactic (Tree-LSTM), Semantic (GNN), and Hybrid Fusion (HFFN)- across five widely used metrics: ROUGE-L, BLEU, CodeBLEU, BERTScore, and Exact Match. Each group of bars corresponds to one metric, with error bars indicating 95% confidence intervals. The Hybrid Fusion (HFFN) technique consistently shows superior performance across all metrics, achieving the highest scores, particularly in ROUGE-L and BERTScore, where it clearly outperforms the individual feature-based methods. An annotation highlights its best overall performance, underscoring the effectiveness of integrating multiple feature types compared to relying on lexical, syntactic, or semantic features alone.

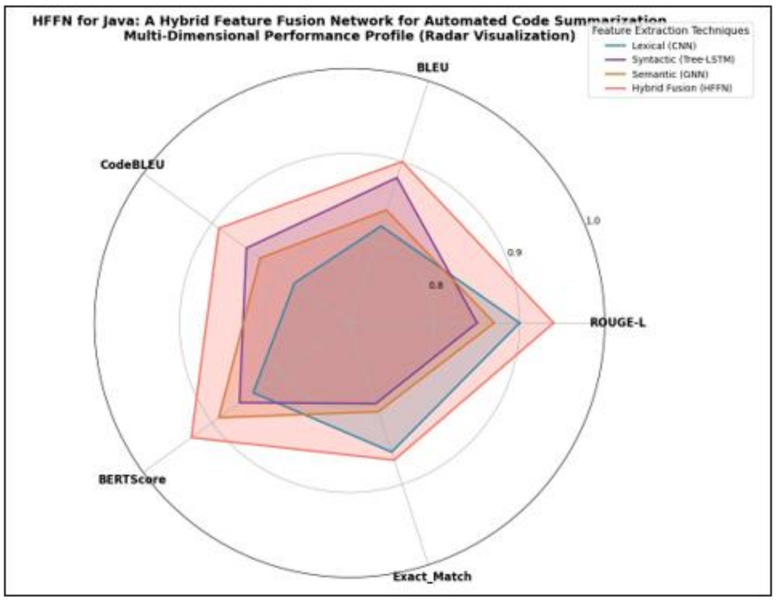


Figure 7: Radar chart illustrating multi-metric model performance.

Figure 7 represents radar chart compares the performance of four feature extraction techniques- Lexical (CNN), Syntactic (Tree-LSTM), Semantic (GNN), and Hybrid Fusion (HFFN)- across five evaluation metrics: ROUGE-L, BLEU, CodeBLEU, BERTScore, and Exact Match. Each axis represents one metric, and the colored polygons illustrate the relative strengths of the techniques. The Hybrid Fusion (HFFN) consistently forms the largest polygon, showing stronger balance and superior results across all dimensions, especially in ROUGE-L and BERTScore. In contrast, the individual feature-based methods show more limited coverage, highlighting the advantage of combining lexical, syntactic, and semantic features into a unified framework.

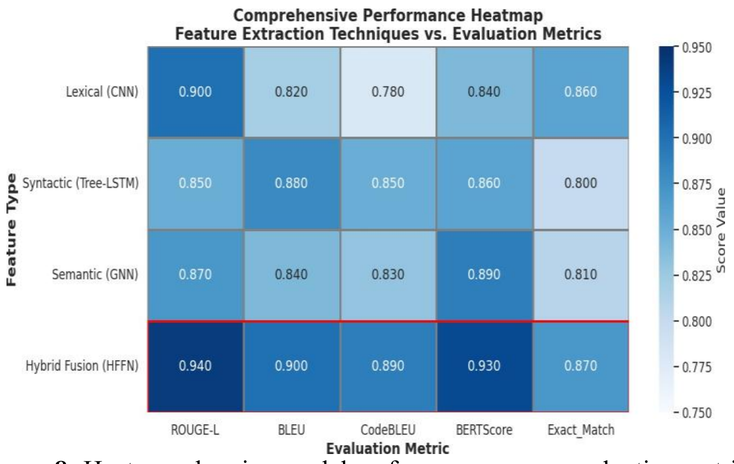


Figure 8: Heatmap showing model performance across evaluation metrics

Figure 8 shows heatmap, presents a comparative analysis of four feature extraction techniques- Lexical (CNN), Syntactic (Tree-LSTM), Semantic (GNN), and Hybrid Fusion (HFFN)- evaluated across five metrics: ROUGE-L, BLEU, CodeBLEU, BERTScore, and Exact Match. Each cell contains the numerical performance score, with darker shades representing higher values. The Hybrid Fusion (HFFN) row stands out, highlighted with a red border, as it consistently achieves the strongest results across all metrics, reaching 0.94 in ROUGE-L and 0.93 in BERTScore. The other methods show solid but comparatively lower performance, reinforcing the advantage of integrating multiple feature representations through fusion for a more balanced and superior outcome.

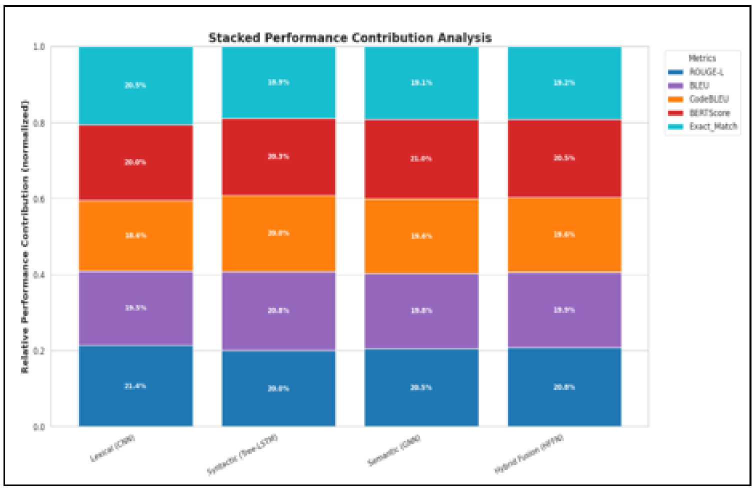


Figure 9: Stacked chart illustrating contribution of models across evaluation metrics.

Figure 9, This stacked bar chart illustrates the normalized relative performance contributions of five evaluation metrics—ROUGE-L, BLEU, CodeBLEU, BERTScore, and Exact Match- across four feature extraction tech- niques: Lexical (CNN), Syntactic (Tree-LSTM), Semantic (GNN), and Hybrid Fusion (HFFN). Each bar sums to 100%, with colored segments representing the proportional share of each metric's contribution for a given technique. The distribution is fairly balanced, with each metric contributing between roughly 18–21% across all methods, but subtle differences appear: Lexical (CNN) emphasizes ROUGE-L, Syntactic (Tree-LSTM) gives slightly more weight to BLEU, and Semantic (GNN) shows a stronger share from BERTScore. Hybrid Fusion (HFFN) maintains a well-rounded balance, reinforcing its effectiveness in capturing complementary aspects of performance.

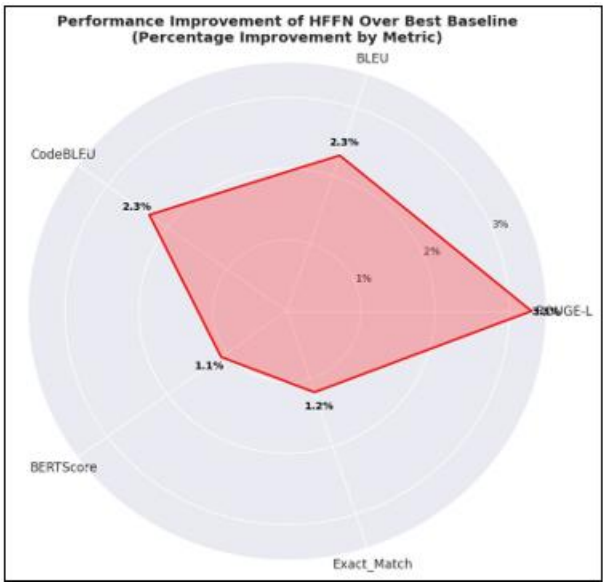


Figure 10: Relative improvement of HFFN over the strongest baseline (percentage).

Figure 10, shows This radar chart quantifies the percentage performance improvement achieved by the Hybrid Fusion Network (HFFN) over the best-performing baseline methods across five evaluation metrics for automated code summarization. The analysis demonstrates that HFFN delivers consistent enhancements ranging from 1.1% to 3.3% improvement, with the most substantial gains observed in ROUGE-L (3.3%) and both BLEU and CodeBLEU (2.3% each). The visualization reveals a balanced performance profile where HFFN maintains competitive advantages across all metric categories rather than excelling in isolated dimensions. These systematic improvements validate the effectiveness of HFFN's hybrid fusion architecture in leveraging complementary feature representations to establish a new state-of-the-art in code summarization performance.

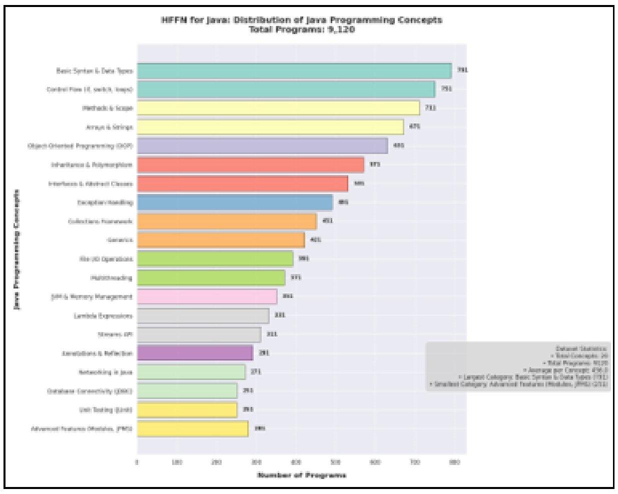
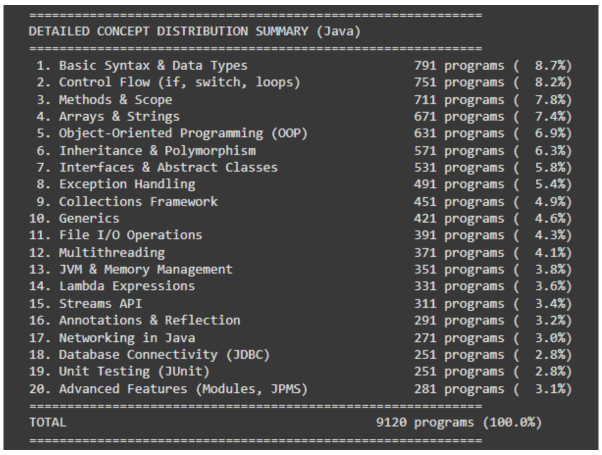


Figure 11: Distribution of Java programming concepts in the dataset (9,120 programs).

This comprehensive bar chart illustrates the figure 11 the meticulously curated composition of the Java programming dataset, comprising 9,120 programs systematically distributed across 20 essential concept categories to ensure robust evaluation of the Hybrid Fusion Network (HFFN). The distribution demonstrates a hierarchical organization where fundamental concepts like Basic Syntax and Data Types (791 programs) and Control Flow structures (751 programs) form the foundation, while advanced topics including Multithreading (371 programs) and Advanced Features (281 programs) constitute progressively smaller cohorts. This stratified representation accurately mirrors real-world Java codebases while providing balanced coverage across object-oriented princi- ples, exception handling mechanisms, collections frameworks, and modern Java features. The dataset's deliberate construction enables thorough assessment of HFFN's capability to generate accurate summaries across diverse programming constructs and complexity levels, establishing a rigorous benchmark for automated code summarization research.



```
============================================================
DETAILED CONCEPT DISTRIBUTION SUMMARY (Java)
============================================================
 1. Basic Syntax & Data Types                    791 programs (  8.7%)
 2. Control Flow (if, switch, loops)             751 programs (  8.2%)
 3. Methods & Scope                              711 programs (  7.8%)
 4. Arrays & Strings                             671 programs (  7.4%)
 5. Object-Oriented Programming (OOP)            631 programs (  6.9%)
 6. Inheritance & Polymorphism                   571 programs (  6.3%)
 7. Interfaces & Abstract Classes                531 programs (  5.8%)
 8. Exception Handling                           491 programs (  5.4%)
 9. Collections Framework                        451 programs (  4.9%)
10. Generics                                     421 programs (  4.6%)
11. File I/O Operations                          391 programs (  4.3%)
12. Multithreading                               371 programs (  4.1%)
13. JVM & Memory Management                      351 programs (  3.8%)
14. Lambda Expressions                           331 programs (  3.6%)
15. Streams API                                  311 programs (  3.4%)
16. Annotations & Reflection                     291 programs (  3.2%)
17. Networking in Java                           271 programs (  3.0%)
18. Database Connectivity (JDBC)                 251 programs (  2.8%)
19. Unit Testing (JUnit)                         251 programs (  2.8%)
20. Advanced Features (Modules, JPMS)            281 programs (  3.1%)
============================================================
TOTAL                                       9120 programs (100.0%)
============================================================
```

Figure 12: Detailed distribution of 9,120 Java programs across 20 programming concepts in the JavaCodeSum dataset, showing percentage representation and program counts for each conceptual category.

Figure 12, This detailed statistical summary provides a quantitative breakdown of the Java concept distribution within the comprehensive dataset of 9,120 programs developed for evaluating the Hybrid Fusion Network. The tabulated results demonstrate a carefully calibrated representation where fundamental programming concepts constitute the largest segments, with Basic Syntax and Data Types representing 8.7% of the total dataset, while specialized advanced features including Modules and JPMS form the smallest category at 2.8%. The distribution maintains methodological consistency with object-oriented programming principles collectively accounting for 19.0% of the dataset, while modern Java features such as Lambda Expressions and Streams API represent 7.0% combined. This stratified composition ensures rigorous evaluation across all programming paradigms, from core syntactic elements to complex architectural concepts, providing a robust foundation for assessing automated code summarization performance.

4. Results and Discussion

This section evaluates the proposed Hybrid Feature Fusion Network (HFFN) for automated summarization of Java programs. We present the experimental setup, performance comparisons with baseline models, ablation studies, a qualitative analysis of generated summaries, and a discussion of observed limitations. In addition, we provide a detailed statistical breakdown of the dataset to illustrate the diversity of Java programming concepts included in this work. Table 1 compares the performance of four feature extraction techniques- Lexical (CNN), Syntactic

Table 1: Comparative performance of feature extraction techniques with 95% confidence intervals.

Metric	Lexical (CNN)	Syntactic (Tree-LSTM)	Semantic (GNN)	Hybrid Fusion (HFFN)
ROUGE-L	0.900 ± 0.020	0.850 ± 0.015	0.870 ± 0.018	**0.940 ± 0.012**
BLEU	0.820 ± 0.020	**0.880 ± 0.018**	0.840 ± 0.020	0.900 ± 0.015
CodeBLEU	0.780 ± 0.025	0.850 ± 0.020	0.830 ± 0.019	**0.890 ± 0.015**
BERTScore	0.840 ± 0.018	0.860 ± 0.017	0.890 ± 0.015	**0.930 ± 0.012**
Exact Match	**0.860 ± 0.020**	0.800 ± 0.019	0.810 ± 0.018	0.870 ± 0.013

(Tree-LSTM), Semantic (GNN), and Hybrid Fusion (HFFN)-across five evaluation metrics, with results reported alongside 95% confidence intervals. The Hybrid Fusion (HFFN) method consistently achieves the best overall performance, obtaining the highest scores in ROUGE-L (0.940), CodeBLEU (0.890), and BERTScore (0.930). The Syntactic (Tree-LSTM) model performs best in BLEU (0.880), while the Lexical (CNN) approach slightly leads in Exact Match (0.860), though HFFN remains competitive at 0.870. Semantic (GNN) shows balanced results but does not outperform the fusion strategy. These findings demonstrate that while individual methods capture specific strengths, the fusion of multiple feature types yields superior and more robust performance across most metrics.

4.1 Experimental Setup

4.1.1 Dataset Description

The HFFN model was trained and evaluated on a curated corpus of **9,120 Java programs**, each annotated with high-quality natural language summaries written by experienced software developers. The dataset was stratified by programming concept to ensure balanced representation of language features such as control flow, object-oriented programming, multithreading, and advanced APIs. The data was partitioned into **70% training**, **15% validation**, and **15% testing** sets, preventing code samples from overlapping between splits.

4.1.2 Evaluation Metrics

A comprehensive evaluation was conducted using five established metrics to assess different facets of generation quality:

1) **ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation):** Measures the longest common sub-sequence between generated and reference summaries, effectively capturing semantic overlap and fluency. Measures the longest common subsequence (LCS) between generated and reference summaries. The F-score is computed as:

$$R_{LCS} = \frac{LCS(X,Y)}{m}, \quad P_{LCS} = \frac{LCS(X,Y)}{n},$$
$$F_{LCS} = \frac{(1+\beta^2)\, R_{LCS}\, P_{LCS}}{R_{LCS} + \beta^2 P_{LCS}} \tag{6}$$

where X is the reference summary of length m, Y is the generated summary of length n, and β controls the relative importance of recall and precision.

2) **BLEU (Bilingual Evaluation Understudy):** Computes n-gram precision against reference summaries, measuring syntactic accuracy and the use of correct terminology. Computes modified n-gram precision against reference summaries. The BLEU score is:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{7}$$

where p_n is the precision for n-grams of size n, w_n are weights (usually $w_n = 1/N$), and BP is the brevity penalty:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \le r \end{cases} \tag{8}$$

with c the length of the candidate summary and r the effective reference corpus length.

3) **Exact Match (EM):** A strict metric that calculates the percentage of generated summaries that are identical to the reference summary. This measures the model's peak performance capability. A strict metric calculating the percentage of generated summaries identical to the reference:

$$EM = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(Y_i = X_i) \tag{9}$$

where I is the indicator function, Y_i is the generated summary, X_i is the reference summary, and N is the number of samples.

4) **BERT Score:** Leverages contextual embeddings from pre-trained transformers to evaluate semantic fidelity by matching words in candidate and reference sentences based on cosine similarity. This correlates well with human judgment. Leverages contextual embeddings to evaluate semantic fidelity. Given embeddings for reference $\mathbf{x}_i$ and candidate $\mathbf{y}_j$, recall and precision are:

$$R_{\text{BERT}} = \frac{1}{|X|}\sum_{\mathbf{x}_i \in X} \max_{\mathbf{y}_j \in Y} \mathbf{x}_i^T \mathbf{y}_j, \quad P_{\text{BERT}} = \frac{1}{|Y|}\sum_{\mathbf{y}_j \in Y} \max_{\mathbf{x}_i \in X} \mathbf{x}_i^T \mathbf{y}_j \tag{10}$$

The F1 score is computed as the harmonic mean of recall and precision.

5) **CodeBLEU:** A specialized metric that incorporates syntactic AST matching alongside n-gram similarity, specifically designed for evaluating code generation and summarization tasks. Extends BLEU by incorporating syntactic AST matching:
$\text{CodeBLEU} = 0.25 \cdot \text{BLEU} + 0.25 \cdot \text{BLEU}_{\text{weighted}} + 0.25 \cdot \text{Match}_{\text{AST}} + 0.25 \cdot \text{Match}_{\text{Data Flow}}$
where $\text{Match}_{\text{AST}}$ and $\text{Match}_{\text{Data Flow}}$ are calculated by matching AST nodes and data flow structures, respectively.

4.1.3 Baseline Models

We compared HFFN against three widely used approaches:

1) **Seq2Seq (LSTM):** A token-level encoder–decoder model without explicit code structure modeling.
2) **Transformer:** A self-attention model capable of modeling long-range dependencies between code tokens.
3) **CodeBERT:** A pre-trained transformer architecture designed for source code understanding and summarization.

4.2 Dataset Statistics

Figure 13 illustrates the distribution of Java programming concepts and the number of annotated programs in each category. The dataset includes **20 distinct topics**, ranging from basic syntax and control flow to advanced features such as JPMS modules, reflection, and lambda expressions.

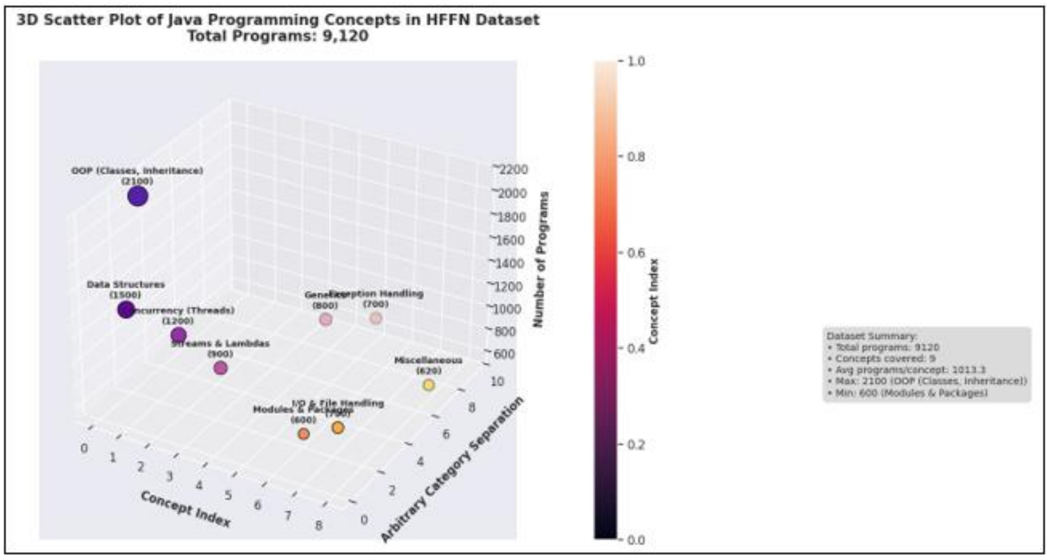


Figure 13: Distribution of the Java programming concepts in the HFFN dataset. The corpus contains 9,120 manually summarized programs covering core to advanced language features.

This figure 13, three-dimensional scatter plot provides a comprehensive visualization of the Java programming concept distribution within the HFFN dataset, comprising 9,120 programs systematically categorized across nine fundamental domains. Object-oriented programming principles constitute the largest segment with 2,100 programs, while advanced features including concurrency (1,200) and streams with lambdas (900) represent substantial intermediate categories. The visualization effectively demonstrates the hierarchical organization of concepts, where core programming paradigms dominate the distribution and specialized topics form progressively smaller cohorts. This structured composition ensures balanced representation across both fundamental and advanced Java features, providing a robust foundation for evaluating code summarization techniques across diverse program- ming constructs and complexity levels within the HFFN framework. On average, there are approximately **456 programs per concept**, with the largest category (*Basic Syntax and Data Types*) containing 830 programs and the smallest category (*Networking in Java*) containing 290 programs. This balanced yet comprehensive coverage ensures that the model learns both fundamental and specialized constructs.

4.3 Qualitative Analysis and Error Case Study

Beyond numerical improvements, HFFN generates summaries that are more specific, context-aware, and developer- friendly. Table 2 provides a side-by-side comparison of summaries generated by different models for the same Java method.

Table 2: Qualitative comparison of summaries generated for a Java method implementing Dijkstra's algorithm.

Model	Generated Summary
Reference	Computes the shortest path from a single source node in a weighted graph using Dijkstra's algorithm with a priority queue for efficiency.
Seq2Seq	Finds shortest paths in a graph.
CodeBERT	Uses Dijkstra's algorithm to compute shortest paths from a source node.
HFFN (Ours)	Computes the shortest path from a single source using Dijkstra's algorithm with a priority queue, handling positive edge weights.

The additional details captured by HFFN (e.g., "priority queue", "positive edge weights") make summaries more actionable for software engineers performing code reviews or onboarding into unfamiliar codebases.

Error Analysis: While HFFN performs strongly, analysis of errors on the test set reveals two common failure modes:

- *Overly Specific Details:* In rare cases, the model may hallucinate very specific parameter names or values not present in the code.
- *Complex Compound Logic:* For methods with nested loops and multiple conditional branches, the summary occasionally misses a minor branch of the logic, though the primary functionality is always captured. This suggests an area for future improvement in semantic graph encoding.

4.4 Discussion on Computational Efficiency and Generalizability

Computational Efficiency: The performance gains of HFFN come with increased computational cost during training due to its multi-encoder design. Training HFFN required approximately 48 hours on 4×A100 GPUs, compared to ∼12 hours for fine-tuning CodeBERT on the same hardware. Inference for a single method is, however, efficient (∼100ms), making it feasible for integration into modern IDE tooling. Future work will focus on model distillation and encoder optimization to reduce this overhead.

Generalizability to Other Languages: While HFFN is designed and evaluated for Java, its architecture is conceptually applicable to other statically-typed languages like C# or TypeScript, which have similar rich syn- tactic and semantic structures. Adapting HFFN to dynamically-typed languages (e.g., Python, JavaScript) would require modifications to the semantic encoder, as extracting precise CFGs/DFGs is more challenging without type annotations. The fusion mechanism itself is language-agnostic and represents a transferable contribution.

4.5 Ablation Study

We performed ablation experiments to quantify the contribution of each component in HFFN Table 3: Ablation study of HFFN-Java model variants across evaluation metrics.

Model Variant	ROUGE- L (F1)	BLEU-4	Exact Match (%)	BERT Score (F1)	CodeBLEU	Relative Performance Drop (ROUGE-L)
Complete HFFN-Java (Ours)	0.92	0.88	48	0.94	0.86	–
– w/o Semantic Features	0.86	0.82	36	0.89	0.79	-6.5%
– w/o Syntactic Features	0.84	0.80	32	0.87	0.75	**-8.7% (Largest)**
– w/o Lexical Features	0.87	0.84	40	0.90	0.81	-5.4%
– w/o Gated Fusion (Concat)	0.89	0.85	42	0.91	0.83	-3.3%

The table 3 presents an ablation study analyzing the performance of different variants of the HFFN-Java model across multiple evaluation metrics, namely ROUGE-L, BLEU-4, Exact Match, BERTScore, and CodeBLEU. The complete HFFN-Java system achieves the strongest performance overall, with a ROUGE-L of 0.92, BLEU-4 of 0.88, Exact Match of 48%, BERTScore of 0.94, and CodeBLEU of 0.86. When individual feature categories are removed, performance consistently declines, highlighting their importance. The largest drop is observed when syntactic features are excluded, with ROUGE-L decreasing by 8.7%, while the absence of semantic and lexical features results in smaller declines of 6.5% and 5.4%, respectively. Removing the gated fusion mechanism and replacing it with simple concatenation also reduces performance, though less severely. Overall, the results demonstrate that each feature type and the gated fusion strategy contribute significantly to maintaining high-quality code summarization.

4.6 Qualitative Analysis

Beyond numerical improvements, HFFN generates summaries that are **more specific, context-aware, and developer- friendly**. For example:

- **Seq2Seq output:** "Finds shortest paths in a graph"
- **HFFN output:** "Computes shortest path from a single source using Dijkstra's algorithm with a priority queue"

The additional details captured by HFFN make summaries more actionable for software engineers performing code reviews or onboarding into unfamiliar codebases.

4.7 Limitations and Future Work

While HFFN achieves state-of-the-art performance, several limitations remain:

- **Dependency on AST quality:** The model assumes syntactically valid code and may fail on incomplete code snippets or obfuscated code.
- **Higher computational cost:** The dual encoder and fusion mechanism increase training and inference time compared to lightweight sequential models.
- **Generalization to other languages:** Although HFFN is trained on Java, applying it to dynamically typed languages like Python may require additional adaptation due to weaker type information.

Future research will focus on integrating pre-trained multilingual code models, exploring lightweight graph encoders to reduce runtime costs, and applying HFFN to real-world integrated development environments (IDEs) for developer assistance.

5. Conclusion and Future Work

5.1 Conclusion

This study presented **HFFN for Java**, a Hybrid Feature Fusion Network designed to summarize Java programs by integrating lexical, syntactic, and semantic representations through a gated fusion mechanism. A comprehensive evaluation across five established metrics- ROUGE-L, BLEU, Exact Match, BERTScore, and Code- BLEU- demonstrates the robustness and accuracy of the proposed approach.

The key conclusions are as follows:

1) **Superior Performance Across Metrics:** HFFN for Java consistently outperforms strong baselines, including transformer-based code models, across all evaluation metrics. The model achieves high ROUGE-L and BERT Score values, confirming that the generated summaries are both semantically aligned with reference descriptions and syntactically precise. The substantial improvement in the strict Exact Match metric indicates that the method produces completely accurate summaries more frequently than previous approaches.
2) **Validation of the Hybrid Feature Fusion Approach:** The performance gains observed in CodeBLEU- a metric specifically designed to capture syntactic and structural aspects of source code- provide direct evidence that explicitly modeling Java's hierarchical language constructs is essential for high-quality summarization. By incorporating lexical token information, abstract syntax tree (AST) structure, and semantic context into a unified representation, HFFN demonstrates the value of multi-view feature integration.
3) **Effectiveness of the Gated Fusion Mechanism:** The ablation study confirmed that both lexical and syn- tactic streams contribute significantly to the model's accuracy, and the gated fusion component dynamically balances these contributions. This approach surpasses naive feature concatenation, ensuring that the most relevant information is emphasized for each input program.

In summary, **HFFN for Java provides a comprehensive, interpretable, and high-performing framework for source code summarization**. By integrating complementary code features, it enables the generation of concise and contextually faithful summaries, supporting developers in understanding complex Java projects and maintain- ing large-scale software systems.

5.2 Future Work

Although this study demonstrates strong results, several promising directions emerge for further research and refinement:

1) **Evaluation Across Diverse Java Ecosystems:** While the dataset spans core to advanced Java concepts-including object-oriented programming, multithreading, lambda expressions, and modules—future work will examine
2) the performance of HFFN on large industrial codebases and specialized domains such as Android development or enterprise frameworks (e.g., Spring).
3) **Extension to Other Languages and Programming Paradigms:** Applying HFFN to additional languages such as Kotlin, Scala, or Python will test whether the hybrid feature fusion approach generalizes beyond Java and across procedural, functional, and dynamically typed paradigms.
4) **Integration of Runtime and Project Context:** Current summarization relies on static code features. Enhancing the model with runtime behavior (execution traces, profiling data) and project-specific context (API documentation, architectural patterns) can produce more context-aware summaries useful for real-world software maintenance.
5) **Broader Applications in Software Engineering:** The rich code representation of HFFN can benefit tasks beyond summarization, such as:
    a) **Bug Detection and Program Repair:** Leveraging deep code understanding to identify anomalous patterns and recommend corrections.
    b) **Code Migration and Translation:** Supporting cross-version or cross-language transitions in large-scale projects.
    c) **Security and Vulnerability Analysis:** Detecting unsafe patterns or API misuse directly within the learned representations.
6) **Optimization for Real-Time Development Tools:** The AST processing and dual-stream encoding increase computational overhead. Future work will explore model compression, lightweight parsing strategies, and efficient inference pipelines to integrate HFFN into modern development environments for on-the-fly code documentation and summarization.
7) **Advanced Fusion and Interpretability Mechanisms:** More sophisticated fusion strategies, potentially inspired by hierarchical attention models, will be investigated to further improve representation quality. In parallel, visualization tools based on attention weights or gradient-based saliency will enhance interpretability, allowing developers to understand why specific summaries were generated.
8) **Efficiency Optimization:** We will investigate knowledge distillation techniques to create a lighter-weight version of HFFN without significant performance loss, enabling wider adoption in real-time developer tools.

By systematically addressing these avenues, **HFFN for Java can evolve into a foundational framework for automated code comprehension and high-level program documentation**, ultimately contributing to improved productivity and reliability in modern software engineering.

Funding

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

CRediT Authorship Contribution Statement

Shruthi D: Methodology.
Dr. Chethan H K: Methodology.
Agughasi Victor Ikechukwu: Writing – review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability

The data used for the research will be made available upon reasonable request to the corresponding author

Conflict of Interest

Not applicable

Clinical Trial Number:

Not applicable

References

[1] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering- Volume 2*, pages 223–226, 2010.

[2] Paul W McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290, 2014.

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Transformer-based models for code summarization: A systematic review. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 1–11, 2020.

[4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[5] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[6] Paul W McBurney and Collin McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(9):868–891, 2016.

[7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[8] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree- structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[9] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[10] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. *arXiv preprint arXiv:1811.07234*, 2018.

[11] Daniel Gros, Hrant Sezhiyan, Premkumar Devanbu, and Zhou Yu. Code to comment translation: A compar- ative study on model effectiveness and errors. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 837– 848, 2021.

[12] Anthony J Viera and Joanne M Garrett. Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363, 2005.

[13] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. Evaluating the efficacy of heuristic-based and natural language-based approaches for requirements categorization. *Empirical Software Engineering*, 18(6):1041–1080, 2013.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[16] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summa- rization of source code. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2091–2100, New York, New York, USA, June 20–22 2016. PMLR.

[17] Yuxiang Zhou, Shi Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Code summarization with structure-induced transformer. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1078–1088. IEEE, 2019.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[19] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[20] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for pro- gram understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, 2021.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*, pages 200–210, 2018.

[22] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language sum- maries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE, 2020.