AEM JOURNAL.ORG

Constructing a Supply Chain Structure Complexity Scoring Standard by Processing Open Source Component Dependency Graphs Using GCN

Heng Xia[1], Liang Meng[2]
[1]Guangxi New Electric Power Investment Group Wuzhou Power Supply Company, Wuzou 543000, Guangxi, China
[2]Digital Intelligence Operation Center of Guangxi Power Grid Co., LTD, Nanning 534000, Guangxi, China
Corresponding author: Heng Xia (e-mail: 18767164710@163.com)

ABSTRACT This paper proposes a multi-attribute graph modeling method based on attention-enhanced GCN to analyze open source component dependency graphs, addressing biases in complexity scores caused by intricate dependency structures and missing version semantics. By integrating multi-attribute dependency information with graph neural networks, this approach improves the accuracy of identifying risks within engineering software supply chains. A multi-attribute dependency graph is constructed with components as nodes and dependencies as edges, integrating dependency types, version constraints, and maintenance metadata as node and edge features. A version-aware edge weighting mechanism is then designed to calculate version intersection coverage by parsing semantic version expressions, enhancing the semantic expression of graph compatibility. Furthermore, a hierarchical edge-attention GCN module is applied, leveraging multi-head attention to dynamically learn propagation weights for critical dependency paths and enhance feature aggregation for high-risk delivery chains. Finally, a fully connected layer generates a fine-grained complexity score that incorporates structural depth, dependency breadth, version fragmentation, and maintenance health, and gradient attribution is used to make the score interpretable. Experiments show that in terms of scoring discrimination ability, the F1-score of the proposed method in large-scale (>500 nodes) scenarios is 0.85 ± 0.018, and the accuracy is 0.88 ± 0.015, which effectively copes with structural complexity; in terms of project scoring consistency assessment, the Spearman rank correlation coefficient in high-fragmentation ($\geq$3 major versions) scenarios is 0.90 ± 0.016, and the KL divergence is 0.32 ± 0.025, which alleviates the problem of missing version semantics.

INDEX TERMS open-source components, dependency graph, graph convolutional network, supply chain complexity, engineering software

I. INTRODUCTION

THE widespread reuse of open-source components has led to a highly interconnected software ecosystem within manufacturing and technical fields, where the structural characteristics of these digital dependencies directly affect the security and maintainability of industrial systems. This architectural complexity determines the overall stability and operational resilience of the modern industry when integrating advanced digital tools [1,2]. Components form a multi-level topological network through explicit dependencies and implicit transfers, in which version constraint logic and maintenance status are intertwined within the graph structure, forming a complex co-evolutionary system [3]. Accurately characterizing the organizational laws of this network requires moving beyond the limitations of simple topological analysis and integrating multidimensional attributes such as semantic compatibility, dependency type weights, and ecological activity [4-6]. Constructing a semantically aware scoring system can not only reveal potential cascading failure paths but also provide a quantitative basis for architectural optimization, dependency updates, and risk response [7,8].

Current models and evaluation systems for open source component dependency graphs suffer from a deep gap between structural representation and semantic integration. Most methods simplify dependency relationships into binary connections, ignoring the logical structure and compatibility intent of the version constraint expressions carried by the edges, resulting in a severe sparsity of semantic information on the edges in the graph [9-10]. Version numbers, as key attributes, are often treated as discrete labels or simple numerical vectors, failing to reflect the asymmetric compatibility ranges defined by operators such as tilde and caret under the SemVer specification, leading to misjudgments of the actual upgradeable ranges between components [11-12]. Node features are often limited to basic metadata, lacking dynamic modeling of maintenance cadence, release patterns, and community response delays, making it difficult to reflect the potential decline in long-term component availability [13]. Graph neural networks generally employ a homogenized neighborhood aggregation mechanism during feature propagation, failing to distinguish between the roles of direct and transitive dependencies in risk transmission, nor do they apply a learnable mechanism for stratified weighting of the influence of different dependency types [14-16]. The generation phase often relies on linear combinations or empirical formulas, lacking a decoupled mapping structure from graph representation to complexity dimensions, making it difficult to trace output results back to specific dependency paths or structural patterns [17-18]. More critically, existing frameworks generally lack the ability to attribute score composition, failing to answer the core governance question of "why a component is complex" [19-20]. These issues collectively restrict the evolution of complexity assessment from static description to dynamic explanation, limiting its practical utility in dependency optimization and architectural decision-making. This paper aims to establish a structural complexity metric for open source component supply chains, overcoming the limitations of traditional metrics in semantic expression and propagation modeling. Its core contributions are reflected in four key areas: First, it proposes a multi-attribute dependency graph construction paradigm that unifies component metadata, dependency types, and version constraint expressions to form a graph representation that combines both structural and semantic information. Second, it designs a version intersection coverage calculation model that generates continuous edge weights by parsing SemVer expressions, accurately reflecting the actual compatibility potential between components and compensating for the information loss caused by discrete judgments. Third, it develops a hierarchical edge-attention GCN (Graph Convolutional Network) module that dynamically learns the propagation importance of each dependency edge during message delivery, particularly focusing on features of high-risk paths and improving the model's sensitivity to critical links. Fourth, it constructs a four-dimensional complexity scoring system that measures structural nesting depth, dependency radius, version distribution dispersion, and maintenance health. Using gradient attribution techniques, it traces the scoring back to the source, achieving an interpretable mapping from scoring results to specific dependency paths. This method not only provides a fine-grained risk profile but also supports dimensionally broken down optimization recommendations, assisting developers in identifying refactoring priorities. The experimental part verifies the model's advantages in complexity discrimination, risk prediction accuracy, and attribution consistency on mainstream open source ecological data sets, and proves its practical value in supply chain governance.

II. RELATED WORK

There have been various modeling attempts in academia for structural analysis of dependency graphs. Early studies used static graph traversal methods to count topological features such as in-degree, out-degree, and path length to identify core nodes and potential bottlenecks [21-22]. Subsequent work applied complex network theory to calculate indicators such as betweenness centrality and clustering coefficient, attempting to reveal the distribution of component influence in the network [23-24]. In the field of machine learning, some studies have attempted to represent components as low-dimensional vectors and use Node2Vec or DeepWalk for embedding learning, which has been widely used in community structure theory and network detection [25-26]. In terms of version processing, Ajibode A's research revealed that the 52,227 pre-trained language models released on Hugging Face had problems such as confusing naming, insufficient documentation transparency, and arbitrary version control, emphasizing the need to establish standardized and semantic versioning practices [27]. Kathi S R proposed an AI-assisted method that automatically repairs and prioritizes dependency vulnerabilities in enterprise-level Java systems through dependency analysis, vulnerability information, and machine learning models, thereby improving the efficiency and effectiveness of security maintenance [28]. However, these methods generally regard version constraints as Boolean judgments, lack a quantitative assessment of the degree of overlap between version intervals, and cannot reflect the size of the actual upgrade space. At the same time, most models assign equal weights to all adjacent edges during message transmission, ignoring the differentiated impact of different dependency types and version matching quality on risk propagation. More importantly, the existing scoring system mostly stays at the macro-aggregation level, lacks decoupled modeling of structural depth, dependency breadth, version fragmentation, and maintenance status, and is difficult to provide actionable optimization suggestions. Overall, current research still has obvious gaps in semantic parsing accuracy, dynamic weight allocation, and fine-grained indicator generation.

The development of graph neural networks has provided a new path for modeling complex dependency structures [29-30]. In recent years, attention mechanisms have been applied to the GNN (Graph Neural Network) architecture, enabling the model to selectively aggregate neighbor information. For example, GAT (Graph Attention Networks) adjusts the importance of edges through learnable attention coefficients, demonstrating superior performance in social networks and recommendation systems [31-32]. In the field of software engineering, some studies have used similar structures to identify vulnerable code modules and trace defect propagation paths using attention weights [33]. Another type of work explores joint encoding strategies for multi-attribute graphs, mapping multiple features of nodes and edges to a unified representation space to enhance the model's ability to integrate heterogeneous information [34-35]. It is worth noting that the hierarchical attention structure performs well in capturing long-range dependencies, allowing the model to focus on local and global patterns at different levels, and has been applied to the field of traffic prediction[36]. Despite this, existing applications have mostly focused on isomorphic graphs or simple attribute graphs, failing to specifically design for the dynamic nature of version intersections and the sparsity of maintenance metadata unique to open source dependency graphs. While attempts have been made to employ edge-attention mechanisms, these mechanisms often rely on manually set thresholds or static rules and lack the ability to adaptively learn from data. Furthermore, most models output classification labels or risk ratings, lacking explicit decomposition and attribution analysis of the scoring components, limiting their practical applicability in governance. In light of this, this paper constructs a version-aware multi-attribute graph modeling framework, combining hierarchical edge-attention GCNs with gradient attribution techniques to achieve end-to-end generation from raw dependency data to interpretable complexity scores.

III. DEPENDENCY GRAPH MODELING BASED ON ATTENTION-ENHANCED GCN

Figure 1 illustrates an open-source supply chain complexity assessment system based on an attention-enhanced graph convolutional network. Starting with multi-source data input, the system constructs a multi-attribute graph containing node metadata and dependency edge semantics. Its core innovation lies in version-aware edge weight calculation, which parses semantic version constraints into continuous compatibility metrics. The hierarchical attention GCN module dynamically learns the importance of dependency paths, capturing complex propagation patterns through a multi-head mechanism. Ultimately, it outputs interpretable scores based on four dimensions: structural depth, dependency breadth, version fragmentation, and maintenance health. Combined with an interpretability layer, this provides a quantitative basis for supply chain governance.

A. BUILDING A MULTI-ATTRIBUTE OPEN SOURCE COMPONENT DEPENDENCY GRAPH

1) Multidimensional Metadata Encoding Mechanism for Node Features

In the construction of the multi-attribute dependency graph, each node represents an open source component, and its semantic representation consists of a set of maintainability indicators extracted from real ecological data. To characterize the long-term evolution trend and governance quality of the component, the initial feature vector of the node is generated by aggregating metadata of multiple dimensions. The maintainer activity is measured by the density of code submissions per unit time, which serves as the core metric. It is worth noting that a high submission frequency alone does not directly equate to high maintenance quality; this work effectively suppresses noise signals caused by emergency fixes or development turbulence by introducing Pull Request response latency and Issue closed-loop stability as adjustment factors. This combined indicator reflects the response consistency of the project maintenance subject. The update frequency uses a sliding window to count the logarithmic mean of the intervals between version releases in the past 12 months, and is processed by minimum smoothing and interval normalization to eliminate noise interference caused by sudden changes in the release rhythm. Vulnerability history information is extracted by linking public records in the national vulnerability database to extract component-related security defect entries, and an exponential decay function is applied to perform weighted accumulation on the time span. The expression is:

$$V_i = \sum_{v \in C_i} e^{-\lambda \cdot \Delta t_v} \tag{1}$$

V_i represents the cumulative vulnerability strength of a component; C_i is the set of all associated security incidents; Δt_v is the time difference between the incident disclosure time and the current moment (in months); and λ is the decay rate parameter, which controls the rate at which the impact of historical risks decays over time. These indicators are normalized and concatenated to form a high-dimensional node attribute vector. Furthermore, a binary flag is applied to encode the lifecycle status of a component, including whether it is under maintenance, marked as a core dependency by the official ecosystem, or has been deprecated, further enhancing the semantic differentiation of nodes within the structure. All features are organized at a unified scale to form the initial input representation of each node in the graph, ensuring that the subsequent graph neural network can effectively capture the dynamic behavior patterns of components in the supply chain.

2) Structural Attribute Modeling of Dependency Edges

Dependencies connect nodes as directed edges, from the depender to the dependee, fully preserving call semantics. Each edge carries three structural attributes: dependency type, call strength, and version constraint semantics. Dependency type is determined based on the hierarchical position of the dependency declaration. By parsing the differences between direct references and nested dependency paths in the project configuration file, edges are classified as direct or transitive and embedded in the edge feature space in a one-hot encoding format. Call strength is obtained by statically analyzing the frequency of references to external component APIs (Application Programming Interfaces) in the source code. The raw counts are logarithmically compressed and Z-score normalized to suppress the over-dominance of frequently called components in graph propagation. It should be noted that the call intensity in this work is based on static source code analysis and may not cover scenarios such as reflection or runtime dynamic loading; however,

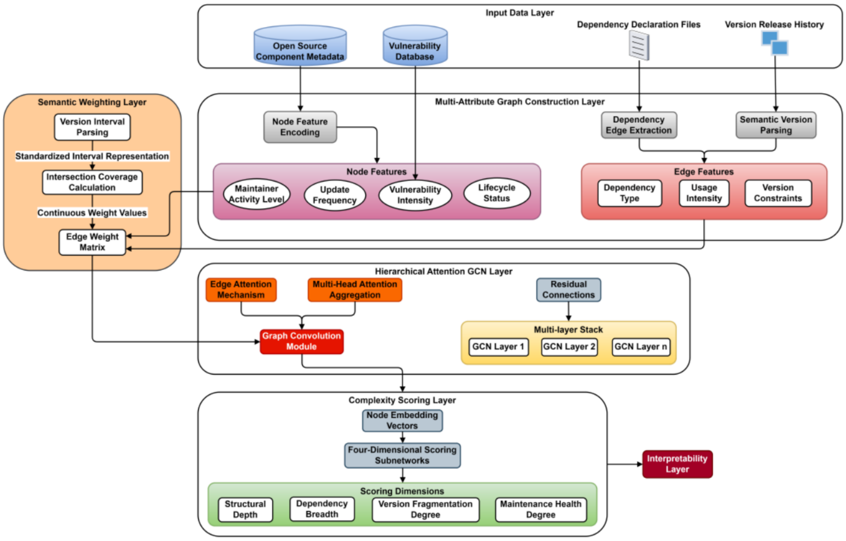

FIGURE 1. Open Source Supply Chain Complexity Assessment System

in the dependency relationships of mainstream open-source components, explicit API references remain the main form of dependency coupling, and this metric, used in conjunction with other edge attributes, can mitigate the bias caused by incomplete coverage to some extent.

The extraction of version constraint semantics relies on the grammatical parsing of SemVer specification expressions, using a finite state machine to identify operator patterns and map them into a mathematical description of version compatibility intervals. The parsing result includes the minimum acceptable version, the maximum acceptable version, and the interval open and closed flag, forming a four-dimensional real number vector. After being concatenated with other edge attributes, this vector is independently normalized and linearly projected into a unified feature space to form an edge attribute tensor. Ultimately, the entire dependency graph is defined by the node feature matrix, edge connection index, and edge attribute tensor, forming a multi-attribute heterogeneous graph with both topological structure and semantic details. This modeling method embeds key governance signals such as version strategy, usage intensity, and dependency hierarchy into the underlying structure of the graph, providing a fine-grained input foundation for subsequent semantically-aware risk propagation modeling.

B. DESIGNING A VERSION-AWARE EDGE WEIGHT CALCULATION MECHANISM

1) Parsing and Standardizing Semantic Version Ranges

To achieve a precise semantic understanding of version constraints in dependency relationships, this paper adopts a syntax-driven parsing strategy to convert raw version expressions into computable numeric intervals. All input version constraint strings are first preprocessed with regular expressions to remove redundant modifiers and illegal characters to ensure format consistency. Then, based on the state transition rules defined in the SemVer specification, a finite state automaton is constructed to identify and classify operators, including common patterns such as the caret (^), tilde (~), and ranges (>=, <, =). Each type of operator triggers the corresponding interval generation logic, and the corresponding minimum and maximum acceptable versions are calculated using the preset version semantic rule engine.

For major version locking class constraints, the maximum boundary is set to the previous version with the major version number incremented; for minor version locking class constraints, the minor version increment is used as the upper limit. All generated intervals are uniformly represented in closed and open form, that is, left-closed and right-open intervals, to avoid compatibility misjudgments caused by boundary processing ambiguity. The interval endpoints are mapped to real number vectors through dotted decimal encoding to facilitate subsequent numerical operations. This process converts discrete text expressions into structured mathematical descriptions, making the version strategies between different components comparable and operational, and solving the semantic loss problem caused by treating versions as labels in traditional methods. This standardized representation serves as the basis for subsequent intersection calculations, ensuring that compatibility evaluation is conducted within a unified semantic framework.

2) Quantification of Version Intersection Coverage and Generation of Edge Weights

After completing the resolution of the version ranges of both dependent parties, it further calculates the degree of overlap in their actual compatible ranges, which serves as the core basis for edge weights. Assuming that the acceptable version range declared by the relying party is $[a_{\min}, a_{\max}]$, and the version coverage range actually released by the dependent party is $[b_{\min}, b_{\max}]$, then the intersection of the two is defined as:

$$I = [\max(a_{\min}, b_{\min}), \min(a_{\max}, b_{\max})] \tag{2}$$

If the lower bound of the intersection is not less than the upper bound, it is determined that there is no valid intersection and the weight is reset to zero; otherwise, the ratio of the intersection length to the length of the interval declared by the relying party is calculated to form a normalized coverage index:

$$w = \frac{\min(a_{\max}, b_{\max}) - \max(a_{\min}, b_{\min})}{a_{\max} - a_{\min}} \tag{3}$$

w is the version compatibility strength weight of the dependency edge, with a value range between [0,1], reflecting the proportion of the actual satisfyable version space to the declared range. This indicator not only captures the degree of overlap of the version range but also implies the influence of the strictness of the constraint - when the declared range is narrow, a small version mismatch causes the weight to drop significantly, thereby amplifying the risk signal of potential upgrade conflicts. To enhance robustness against edge cases, a small threshold is applied. When the denominator is less than this value, it is regarded as an exact match to prevent division by zero anomalies. The weights of all edges are quantile-normalized globally to eliminate systematic offsets caused by differences in version release density between different ecosystems. Finally, the weight matrix is embedded in the adjacency structure of the graph, replacing the traditional binary connection identifier to form a weighted adjacency representation. In subsequent graph convolution operations, this weight directly participates in the weighted aggregation of features during message passing, giving dependency paths with high version compatibility a higher propagation priority in information flow. Through this mechanism, the model can dynamically distinguish between "nominal dependencies" and "actually operational dependencies," effectively alleviating the problem of false connections caused by version mismatches and improving the graph's ability to accurately depict real-world supply chain behavior. Here, "actually operational dependencies" does not refer to runtime behavior, but rather to dependencies with nonzero version overlap coverage in the static dependency declaration, i.e., dependencies with actual satisfiability at the version semantic level; in contrast, "nominal dependencies" only refers to declarations existing in the configuration file, regardless of whether they can actually be satisfied. This weight construction method elevates semantic compatibility from a Boolean judgment to a continuous metric, providing a fine-grained structural basis for complexity assessment.

Figure 2 shows a scatter plot of the correlation between dependency type, call intensity, and version coverage. The horizontal axis shows normalized call intensity, and the vertical axis shows calculated version intersection coverage. Blue dots represent direct dependencies, and red dots represent transitive dependencies. The data shows a very clear distribution pattern that fully aligns with theoretical expectations: blue dots (direct dependencies) are prominently clustered in the upper right quadrant of the graph. Specifically, direct dependencies have version coverage greater than 0.4 (high compatibility) and call intensity greater than 0.5 (high usage intensity). This indicates that developers actively maintain and call stable and compatible direct dependencies, which are core to project functionality. In contrast, red dots (transitive dependencies) are mostly distributed in the lower left quadrant of the graph. Transitive dependencies mostly have version coverage less than 0.4 (low compatibility) and call intensity less than 0.5 (low usage intensity). These are often indirectly imported dependencies that have not been carefully evaluated, applying potential compatibility risks. This stark contrast demonstrates the absolute necessity of distinguishing dependency types in edge features.

Figure 3(a) shows a continuous response curve of the "intersection coverage weight" as the starting point of the dependent component's release interval changes. The horizontal axis represents the minimum version number of the actual release interval, which ranges from 1.0 to 2.2, covering the typical migration interval from major version 1 to 2. The vertical axis shows the proposed intersection coverage weight and the traditional Boolean matching degree. The data shows that when the minimum version number is lower than 1.2, the intersection coverage weight increases as it rises, reaching a peak after the minimum version number is 1.2. It then decreases after the minimum version number is greater than 1.6 as the release interval gradually moves outside the declared range, forming an asymmetric bell-shaped response. This change reveals that compatibility is not a sudden change, but rather has a gradual overlap and decay period. Traditional Boolean methods show a hard jump in this period and fail to reflect the true compatibility potential. Figure 3(b) shows a weighted adjacency matrix heat map of a five-node open source component subgraph. The data is an explicitly defined 5×5 matrix, no longer relying on random generation. The horizontal axis represents the dependency initiator component, and the vertical axis represents the dependent target component. Matrix elements represent the version compatibility weights from the initiator component to the dependent component. The dependency weight of initiator component 2 on dependent target component 1 is 0.65, indicating strong compatibility. The dependency weight of initiator component 4 on dependent target component 1 is 0.78, the highest in the graph, indicating a close match between their version strategies and stable maintenance. The dependency weight of initiator component 5 on dependent target component 3 is 0.12, approaching the failure threshold, indicating potential version fragmentation or maintenance lags. Edge weights constructed using version intersection coverage are not only mathematically rigorous but also clearly visualize differences in the actual compatibility potential between components, supporting fine-grained modeling of subsequent complexity scores.

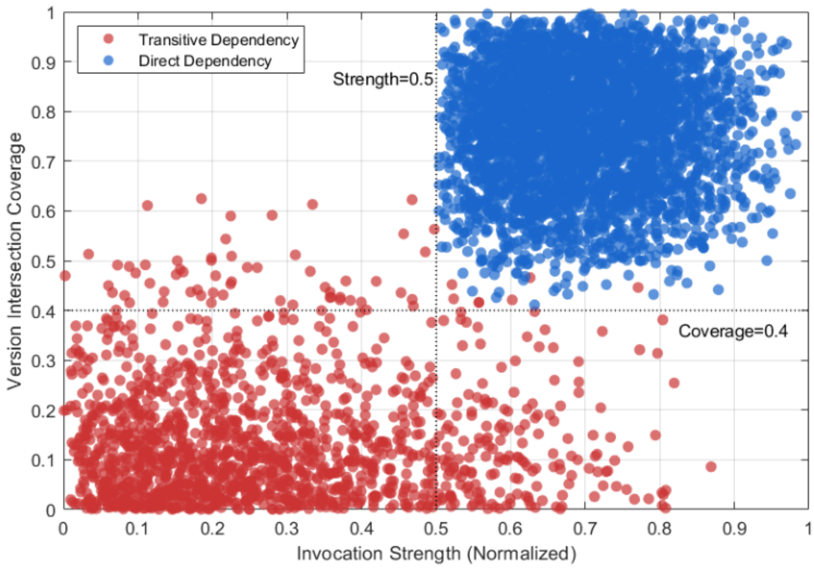

FIGURE 2. Dependency Attribute Association Analysis

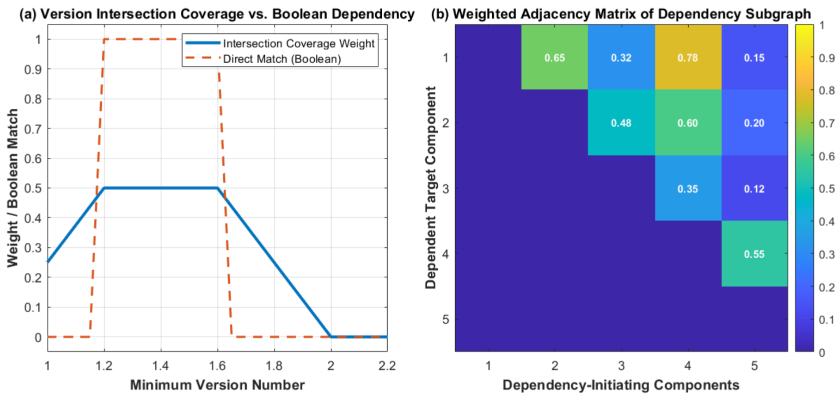

FIGURE 3. Visualization of Version Compatibility and Dependency Weight

C. IMPLEMENTING THE HIERARCHICAL ATTENTION GRAPH CONVOLUTION MODULE

In each layer of the graph convolution operation, to achieve dynamic regulation of the propagation strength of the dependency path, an attention calculation structure that depends on edge attributes is designed. This mechanism takes the edge feature vector between node pairs as input, including version compatibility weights, dependency type encodings, and call strength normalization values. It generates attention precursor representations through learnable linear transformations. Specifically, for any directed edge-connected node, its edge feature vector is mapped to a low-dimensional latent space, processed by a nonlinear activation function, and then spliced with the current layer embeddings of the source and target nodes to form a joint attention input. The spliced vector is then weightedly projected through another set of trainable parameters to output a scalar attention score. The process can be expressed as:

$$a_{ij} = \mathbf{w}^\top \sigma \left(\mathbf{W}_1 [\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}, \mathbf{e}_{ij}] \right) \tag{4}$$

a_{ij} is the unnormalized node-to-node attention coefficient; $\mathbf{h}_i^{(l)}$ and $\mathbf{h}_j^{(l)}$ are the hidden states of nodes i and j at layer l, respectively; $\mathbf{e}_{ij}$ is the attribute vector of the corresponding edge; $\mathbf{W}_1$ is the shared transformation matrix; σ is the LeakyReLU (Leaky Rectified Linear Unit) activation function; and $\mathbf{w}$ is the attention weight vector. The attention scores of all adjacent edges are normalized on the set of incoming edges of the target node using the Softmax function to ensure that the weight distribution has an interpretable probabilistic meaning.

To further enhance the model's ability to capture complex dependency patterns, a multi-head attention mechanism is applied to perform the aforementioned attention calculation process in parallel across multiple independent subspaces. Each attention head is equipped with an independent parameter matrix, allowing the model to learn differentiated dependency importance distributions across different representation subspaces. The weighted aggregation results of the outputs from each head are concatenated and then mapped back to the original hidden dimensions via a linear transformation to form the final layer output representation:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\Big\|_{k=1}^{K}\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(k)}\mathbf{W}_2^{(k)}\mathbf{h}_j^{(l)}\right)\right)\mathbf{W}_o \quad (5)$$

$\alpha_{ij}^{(k)}$ is the normalized weight calculated by the attention head; $\mathcal{N}(i)$ is the node i's neighbor set; $\|$ represents the vector concatenation operation; $\mathbf{W}_2^{(k)}$ is the feature transformation matrix for each head; and $\mathbf{W}_o$ is the output projection matrix. Furthermore, during inter-layer transport in the four-layer GCN, residual connections are used for gating and fusing the state of the previous layer's nodes with the output of the current layer, thereby mitigating the gradient diffusion problem that may result from four-layer deep stacking. The gating mechanism is controlled by a learnable parameter that dynamically adjusts the proportion of historical states retained based on the information gain of the current layer.

TABLE 1. Core Configuration of the Hierarchical Edge Attention GCN Module

Parameter Category	Parameter Name	Value
Network Architecture	Number of GCN Layers	4
	Number of Attention Heads	8
	Node Hidden Dimension	128
	Edge Feature Dimension	64
Training Configuration	Learning Rate	0.001
	Dropout Rate	0.3
	Weight Decay	5×10^{-4}
	Batch Size	32
	Early Stopping Patience	30
Attention Mechanism	LeakyReLU Slope	0.2

Table 1 lists the core configuration of the layered edge attention GCN module. The network consists of four layers, each employing an 8-head attention mechanism. The node hidden dimension is set to 128, and the edge feature dimension is 64, balancing expressiveness and computational efficiency. During training, a learning rate of 0.001 and a dropout rate of 0.3 are used to prevent overfitting; weight decay is set to 5×10^{-4} for enhanced regularization. The batch size is 32, with 30 epochs of early stopping to ensure training stability. LeakyReLU activation is used in the attention calculation, with a slope of 0.2. The overall parameter settings have been fine-tuned over multiple rounds, ensuring that the model remains lightweight while effectively supporting the fine-grained modeling of complex dependency paths and accurately identifying high-risk transmission chains.

D. GENERATING AN INTERPRETABLE COMPLEXITY SCORE VECTOR

1) Four-Dimensional Decoupled Complexity Score Generation Mechanism

After the graph neural network completes multi-layer feature propagation, each component node corresponds to a high-dimensional embedding vector that combines its structural context within the dependency graph with multi-attribute semantic information. To achieve a refined depiction of supply chain complexity, a structured output layer is designed to map the final layer node embeddings to four interpretable complexity dimensions. Specifically, a set of independent, fully connected subnetworks is used to process the embedding vectors separately, each dedicated to generating a scoring component for a specific dimension. This ensures that the various indicators do not interfere with each other during the learning process, resulting in decoupled modeling. The structural depth is evaluated by the sub-network at the nesting level of the node in the dependency call chain, and by capturing the longest and shortest path distribution between it and the top-level application, it quantifies the depth of its position in the supply chain; the dependency breadth reflects the radiation range of the component referenced by different projects or modules, and combines the in-degree statistics and cross-subgraph connectivity indicators to measure its ecological coupling strength. The version fragmentation is determined by analyzing the degree of dispersion of the version constraint intersection coverage distribution of all its upstream dependencies. The standard deviation and range information are implicitly encoded in the feature propagation path, extracted by the corresponding sub-network, and normalized for output. Maintenance health is based on the dynamic evolution pattern of the node's own maintenance metadata, and evaluates the comprehensive performance of its update regularity, vulnerability repair delay, and community response level. Although the various complexity dimensions are correlated in reality, the use of independent sub-networks aims to achieve dimensional decoupling and attribution clarity of the scoring results; it is worth noting that, since all sub-networks share the same context-aware embedding extracted by the same graph neural network, the potential dependencies between dimensions have been fully modeled at the feature level. The output of each sub-network is compressed to the [0,1] interval by Sigmoid transformation to form a standardized score:

$$s_d = \sigma(\mathbf{w}_d^{\top}\mathbf{h}_i^{(L)} + b_d) \quad (6)$$

s_d represents the score of the d-th complexity dimension; $\mathbf{h}_i^{(L)}$ is the final embedding of node i at layer L; $\mathbf{w}_d^{\top}$ and b_d are the learnable weight vector and bias term corresponding to that dimension, respectively. This design avoids the dimensional confusion caused by traditional linear weighted combinations, focusing each scoring component on a specific governance dimension, improving the semantic clarity of the results and their optimization guidance value.

2) Gradient Sensitivity-Based Rating Attribution Analysis Method

To enhance the traceability and governance support capabilities of scoring results, a gradient-based attribution mechanism is applied to locate the graph structural elements that contribute most to the scoring output. In the specific implementation, each complexity score is regarded as a differentiable function of the original input features and the graph structure, and an automatic differentiation system is used to calculate the gradient amplitude of the score relative to the weight of each dependent edge. The larger the absolute value of the gradient, the more significant the impact of the edge on the final score in the feature propagation path. Through the backpropagation chain rule, the gradient information is traced back step by step along the network layer to locate the key dependency path:

$$g_{ij} = \left| \frac{\partial s_d}{\partial w_{ij}} \right| \tag{7}$$

g_{ij} represents the attribution strength of the edge's dimension score, and w_{ij} is the edge's weight in the adjacency matrix (i.e., version intersection coverage). This gradient value, after global normalization, can be used to rank and select the top k dependency edges that contribute most to complexity. Furthermore, the attribution strengths are mapped back to the original dependency graph, generating a heatmap visualization that highlights the specific dependencies that lead to unusual structural depth or increased version fragmentation. This method requires no additional training or proxy models, leveraging the differentiable structure of the backbone network for efficient attribution, ensuring that interpretations are strictly consistent with the model's decision logic. This mechanism allows developers to precisely identify specific dependencies that contribute to high complexity, such as a transitive dependency that applies highly fragmented version constraints or a component that hasn't been updated for a long time, which significantly reduces the maintenance health score. This attribution capability transforms abstract scores into concrete governance actions, significantly improving the practical applicability of complexity assessments for dependency refactoring, upgrade prioritization, and architectural optimization.

IV. SUPPLY CHAIN COMPLEXITY EVALUATION BASED ON MULTIDIMENSIONAL INDICATORS

A. EXPERIMENTAL DATA

The experimental data are derived from real dependency graphs of mainstream open-source ecosystems, covering the three major platforms of PyPI, npm, and Maven. Data collection is performed by parsing the metadata repositories of each ecosystem to obtain explicit dependencies between components. This data is then expanded using static dependency trees to extract transitive dependencies and construct a complete directed dependency graph. Version constraint information is accurately extracted from configuration files, retaining the original SemVer expressions to support subsequent semantic parsing. Maintenance metadata includes component release history, commit frequency, maintainer response latency, and CVE (Common Vulnerabilities and Exposures) vulnerability records, all of which are linked and supplemented via the GitHub API and the NVD (National Vulnerability Database). To ensure representative evaluation, it samples components from various ecosystems stratified by popularity and ecosystem role, ultimately constructing a multi-attribute graph dataset containing over 12,000 component nodes and 85,000 dependency edges. All graph structures are segmented at the project level to ensure no overlapping dependency paths between subgraphs and prevent information leakage. During data preprocessing, version expressions are standardized and mapped uniformly into computable compatibility intervals. Sparse features are also smoothed and normalized. This dataset combines scale diversity, ecosystem heterogeneity, and semantic integrity, providing a solid foundation for evaluating model performance under different dependency patterns.

B. MULTIDIMENSIONAL ANALYSIS OF OPEN SOURCE COMPONENT MAINTENANCE STATUS

To analyze the maintenance status of open source components from multiple dimensions, the evaluation process first extracts multidimensional features based on the actual metadata of the open source ecosystem, including maintainer activity, update frequency, and vulnerability history. Maintainer activity is calculated by calculating the density of code commits and weighting the issue resolution cycle. Update frequency is calculated using a sliding window to calculate the logarithmic mean of the release interval and normalize it. Vulnerability intensity is recorded in a linked security database, and an exponential decay model is applied to perform a time-weighted accumulation of historical vulnerabilities. The Pearson coefficient matrix is used for feature correlation analysis to verify the independence and complementarity of signals in each dimension. Finally, the feature encoding mechanism's ability to represent real-world component behavior is verified through visualization of distribution patterns and statistical goodness-of-fit.

Figure 4 analyzes the maintenance patterns and security risk characteristics of open source components. Figure 4(a) shows the joint distribution of maintainer activity and update frequency. The horizontal axis represents normalized maintainer activity, and the vertical axis represents normalized update frequency. Most data points are clustered in the lower left corner (low activity, low update frequency), forming a distinct cluster. However, a small number of components exhibit both high activity and high update frequency. This demonstrates that in the open source ecosystem, most components evolve slowly, while core components exhibit high activity, validating the effectiveness of feature extraction in distinguishing component activity levels. Figure 4(b) shows the empirical data histogram and exponential fit of the probability distribution of cumulative vulnerability strength. The horizontal axis represents the normalized cumulative vulnerability strength value, and the vertical axis represents the corresponding probability density. The blue histogram shows that the vulnerability strength values of most components are close to less than 0.4 (safe). As the strength value increases, the probability density decreases sharply (in line

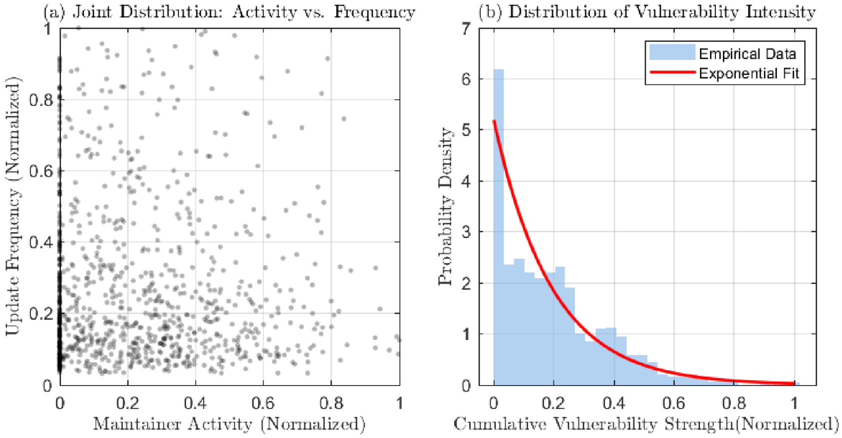

FIGURE 4. Multidimensional Analysis of Component Node Characteristics

with the exponential distribution law, red fitting line). Only a very small number of components have high vulnerability strengths. This reflects the design intention of applying an exponential decay function to perform a weighted summation of historical vulnerabilities: recent, high-frequency vulnerabilities have a much greater impact on the current security status of a component than isolated vulnerabilities from older times, allowing the feature to effectively capture the real-time security risks of components.

C. VERSION COMPATIBILIITY AND OVERLAP ANALYSIS IN DEPENDENCY NETWORKS

During the evaluation process, it first defines the version ranges for each pair of dependencies, including the version range declared by the relying party and the released version range of the dependent party. Next, it calculates the size of the intersection of these two intervals, and based on the ratio of the intersection length to the total length of the declaring interval, it derives the compatibility weight of each pair of dependencies. The larger the intersection, the greater the compatibility between the versions. Subsequently, it aggregates the weights of all dependency pairs and analyzes the compatibility distribution between different dependency pairs. Finally, based on the intersection coverage calculation method, it generates a visualization of the interval intersections and compares the matching of the version constraints of different dependency pairs with the actual releases, ultimately evaluating the overall version compatibility of the system.

In Figure 5(a), the horizontal axis represents version numbers, and the vertical axis represents different dependency pairs. The blue line represents the acceptable version range declared by the relying party, the red line represents the version range actually released by the dependent party, and the thick green line represents the actual intersection of the two. Six dependency pairs are set in the data. The first pair declares a range of [1.0, 3.0], while the dependent's range is [2.0, 4.0]. There is an overlap between the two in [2.0, 3.0]. The third and fourth pairs have no intersection at all, so the green line is missing. Visualizing these intersections clearly demonstrates the degree of match between different dependency strategies, thereby intuitively reflecting potential compatibility risks. Figure 5(b) shows the distribution of version compatibility weights. The vertical axis of the bar chart represents weight, while the horizontal axis represents the sequence number of different dependency pairs. Data calculation results show that: the weight of the first group of dependency pairs is 0.5, indicating that half of the declared range can be satisfied by the actual versions; the weight of the second group is less than 0.2, indicating that only a small number of versions meet the requirements; the weight of the third group is 0.0 (almost no compatibility); and the weight of the sixth group reaches 1, indicating that they have high compatibility. Overall, the weight values vary significantly between different dependency pairs, revealing the differences in the impact of the strictness of dependency declarations and the release policy of the dependent party on compatibility. The data shows that dependency topology information alone is not sufficient to characterize risk; the semantic intersection of version constraints must also be considered, which can provide a fine-grained basis for complexity scoring.

D. COMPARISOM OF SCORING DISCRIMINATIVE ABILITY AT DIFFERENT ECOLOGICAL SCALES

To systematically validate the model's adaptability in heterogeneous ecosystems, it selects 5,000 representative component subgraphs from the PyPI and npm ecosystems as the evaluation benchmark, ensuring coverage of supply chain structures of varying scale and complexity. Using a stratified sampling strategy, it divides the subgraphs into three scale ranges based on the number of nodes: small (<100 nodes), medium (100-500 nodes), and large (>500 nodes). It retains 500 independent samples from each scale category to ensure statistical significance. For each subgraph, it calculates the binary classification performance of the model output based on a manually annotated list of highly complex components (determined by dual thresholds of structural depth and version fragmentation). The core evaluation metrics used are F1-score and accuracy, with

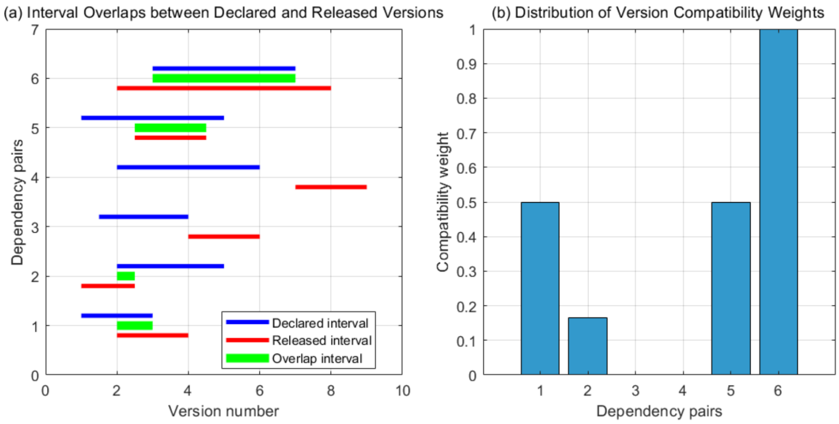

FIGURE 5. Version Compatibility and Overlap Analysis in Dependency Networks

5-fold cross-validation used to mitigate data segmentation bias. Comparative experiments included the popular standard GCN, GAT, GraphSAGE (Graph Sample and aggreGatE), and the proposed GCN model. All models are trained on the same preprocessed data, and hyperparameters are uniformly tuned using Bayesian optimization.

Figure 6 compares the discriminative performance of the proposed method and mainstream graph neural network models on subgraphs of varying sizes from the PyPI and npm ecosystems. All models achieve relatively optimal discriminative performance on medium-sized graphs. This is attributed to the balance between the number of nodes and connection density achieved by such graph structures: sufficient topological complexity to support semantic learning, while avoiding information dilution or oversmoothing during feature propagation due to overscaling. In contrast, small-scale graphs are limited by insufficient contextual information and struggle to capture deep dependency patterns. Large-scale graphs, on the other hand, suffer from path redundancy and noise accumulation, which impair the accuracy of identifying key paths. The proposed GCN model significantly outperforms the baseline method across all scales, with a particularly strong performance in large-scale scenarios, achieving an F1-score of 0.85 ± 0.018 and an accuracy of 0.88 ± 0.015. This demonstrates that its hierarchical edge attention mechanism effectively enhances sensitivity to long-range dependency chains and enhances feature selection. Furthermore, the proposed model achieves a significantly smaller error margin, demonstrating the enhanced stability and reproducibility of its predictions. The data verify the robustness and generalization ability of the designed version of perception weight and multi-attribute fusion strategy in heterogeneous ecosystems.

E. PROJECT SCORING CONSISTENCY ASSESSMENT

To validate the ability of complexity scores to reflect actual version management quality, an external consistency assessment protocol based on real dependency structures is constructed. In the Node.js ecosystem, projects are categorized into three levels of version fragmentation: high ($\geq$3 major versions), medium (2 major versions), and low (only 1 major version), based on the number of different major versions of the same component in their dependency tree. 400 representative projects are selected for each category to ensure balanced coverage of the sample in terms of ecological distribution, functional categories, and maintenance cycles. Using this classification as an external benchmark, the Spearman rank correlation coefficient between the version fragmentation scores generated by each model and the actual fragmentation level is calculated to measure their ranking consistency. The KL (Kullback-Leibler) divergence is also applied to assess the degree of divergence between the probability distribution of the scores and the actual category distribution. This dual-metric system comprehensively characterizes the semantic fidelity of the scores from the perspectives of ordinal alignment and distributional approximation. All models underwent 5-fold cross-validation, and their hyperparameters are independently tuned. This design effectively isolates structural biases and accurately assesses the models' ability to perceive version management quality.

Figure 7 shows the consistency evaluation results between version fragmentation scores generated by different graph neural network models and the actual dependency management quality in the Node.js ecosystem. Using two complementary metrics, the Spearman rank correlation coefficient and KL divergence, the system measures the ordinal alignment and distributional approximation of the score output to the true version distribution. Experimental results show that all models exhibit lower correlation and distributional bias on low-fragmentation projects. This is attributed to the clear dependency structure and unified versioning strategy of such projects, which allows the models to more accurately capture their governance characteristics. However, as the degree of fragmentation increases, version conflicts and compatibility gaps increase, making it significantly more difficult to match the scores with the true state. Notably, the proposed GCN model significantly outperforms baseline methods in all categories and demonstrates greater robustness in high-fragmentation scenarios, with a Spearman rank correlation coefficient of 0.90 ± 0.016 and a KL divergence

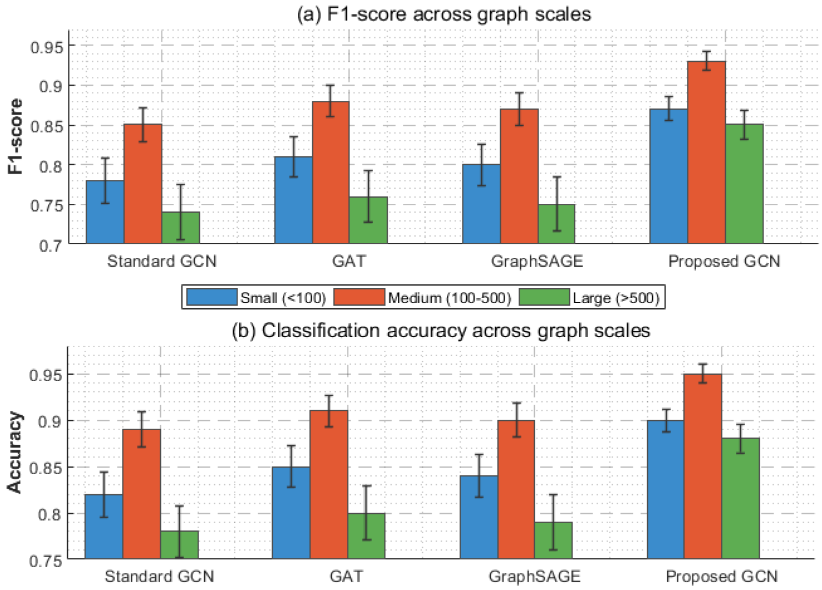

FIGURE 6. Scoring Ability

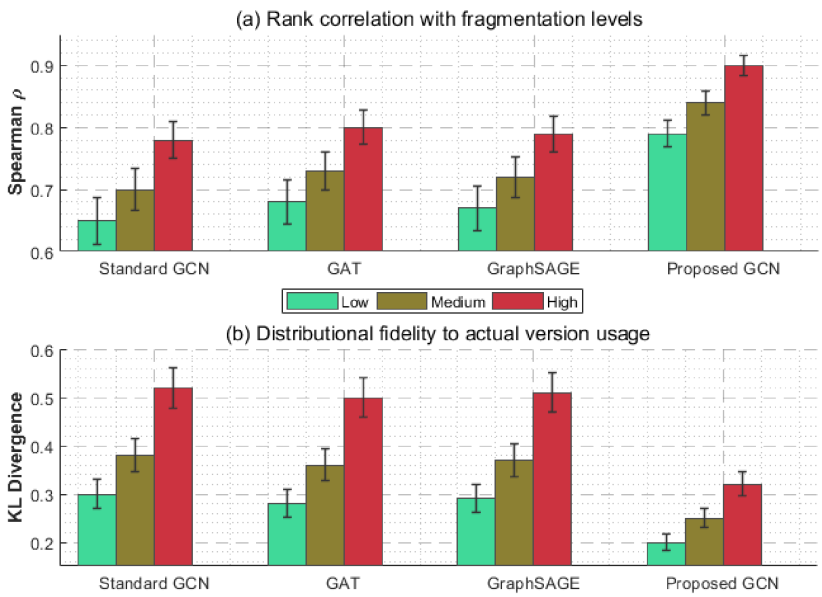

FIGURE 7. Item Scoring Consistency Assessment

of 0.32±0.025, indicating that its version-aware edge weight mechanism and hierarchical attention structure effectively enhance its ability to parse complex version logic. Furthermore, the model achieves a smaller error margin, reflecting its higher predictive stability across different project types.

F. CRITICAL PATH IDENTIFICATION PERFORMANCE

To validate the model's ability to identify supply chain risk propagation paths, a path restoration evaluation framework based on real vulnerability associations is constructed. The importance score of each edge in the graph obtained in gradient attribution analysis is used as the ranking criterion, and the top K high-risk edges output by each model are extracted to form the predicted path set. Precision@K and Recall@K are used as core evaluation metrics to measure the degree of overlap between the predicted edge set and the real vulnerability propagation paths: the former reflects the purity of the top K prediction results, while the latter assesses the proportion of the real critical path covered. The K value is set to 10, 20, and 30 based on typical dependency tree depths to cover common call chain lengths. All models used a unified preprocessing graph structure and normalization strategy to ensure evaluation fairness.

TABLE 2. Critical Path Identification Performance

Model	K	Precision@K (mean ± std)	Recall@K (mean ± std)
Standard GCN	10	0.42 ± 0.038	0.31 ± 0.042
	20	0.38 ± 0.035	0.48 ± 0.039
	30	0.35 ± 0.033	0.59 ± 0.037
GAT	10	0.48 ± 0.036	0.36 ± 0.040
	20	0.44 ± 0.033	0.53 ± 0.036
	30	0.40 ± 0.031	0.61 ± 0.034
GraphSAGE	10	0.45 ± 0.037	0.34 ± 0.041
	20	0.41 ± 0.034	0.50 ± 0.038
	30	0.37 ± 0.032	0.58 ± 0.036
Proposed GCN	10	0.63 ± 0.024	0.52 ± 0.028
	20	0.58 ± 0.021	0.69 ± 0.025
	30	0.54 ± 0.019	0.76 ± 0.022

[Note: std is the abbreviation of Standard deviation]

Table 2 compares the performance of different graph neural network models for identifying real vulnerability propagation paths, using the precision@K and recall@K metrics for the first K high-weight edges. Experimental results show that as the value of K increases, the recall of all models increases, while the precision gradually decreases, reflecting the noise applied by the expansion of the candidate edge set. The proposed GCN model significantly outperforms standard GCN, GAT, and GraphSAGE across all K settings, achieving a precision of 0.63 ± 0.024 and a recall of 0.52 ± 0.028 at K=10, demonstrating that its hierarchical edge attention mechanism effectively focuses on the most critical propagation links. More notably, this model exhibits a lower standard deviation in five-fold cross-validation, indicating that its predictions are more stable and reproducible, and are not significantly affected by local graph structure perturbations. This low standard deviation further validates the robustness of the version-aware weighting and attention-driven feature aggregation strategy for risk transmission path identification. Compared with the baseline model, the proposed method not only improves the coverage of real vulnerability paths but also enhances the credibility of key edge sorting, providing a more operational decision-making basis for priority repair and dependency reconstruction in actual scenarios.

G. TESTING THE GENERALIZABILITY OF SCORESIN A CROSS-ECOSYSTEM MIGRATION SCENARIO

To validate the model's ability to generalize scores across heterogeneous open source ecosystems, a cross-domain migration experiment is designed to evaluate its adaptability to unseen ecosystems. The source domain is the PyPI ecosystem, characterized by high version update frequency and moderate dependency density; the target domain is the Maven ecosystem, characterized by low update frequency and high dependency nesting depth, typical of enterprise-grade applications. After the model is fully trained to convergence in the source domain, all parameters are frozen and the model is directly deployed on the target domain's dependency graph to predict complexity scores without any fine-tuning or retraining. Using the target domain's true scores (generated based on manual annotation and statistical benchmarks) as a reference, prediction bias is quantified using three metrics: root mean squared error (RMSE), mean absolute error (MAE), and coefficient of determination (R^2). This protocol rigorously simulates real-world migration scenarios, testing the model's robustness to ecosystem-specific noise and its ability to universally represent cross-domain structural patterns, revealing the applicability boundaries of different architectures for cross-ecological governance tasks.

TABLE 3. Rating Generalization

Model	RMSE (mean ± std)	MAE (mean ± std)	R^2 (mean ± std)
Standard GCN	0.286 ± 0.032	0.214 ± 0.025	0.682 ± 0.041
GAT	0.263 ± 0.029	0.197 ± 0.023	0.721 ± 0.038
GraphSAGE	0.271 ± 0.031	0.203 ± 0.024	0.708 ± 0.039
Proposed GCN	0.192 ± 0.018	0.141 ± 0.016	0.834 ± 0.026

Table 3 shows the generalization performance of each model in a cross-ecosystem migration scenario, using metrics such as RMSE, MAE, and coefficient of determination (R^2). The source domain is PyPI, and the target domain is Maven. Experimental results show that all models experience varying degrees of performance degradation in cross-domain prediction, reflecting the distribution shift challenge caused by differences in update frequency and dependency structure between ecosystems. Lower RMSE and MAE indicate that the model's estimate of target domain complexity is closer to the true distribution, while higher R^2 values demonstrate its ability to effectively capture structural patterns in the high dependency density of the Maven ecosystem. The proposed GCN model significantly outperforms standard GCN, GAT, and GraphSAGE across all three metrics, achieving the lowest standard deviation, with an R^2 of 0.834±0.026. This demonstrates that its predictions are not only less biased but also more stable across different data partitions. The small standard deviation reflects the model's robustness to inter-domain heterogeneity, preventing significant fluctuations due to differences in training and testing environments. This performance is attributed to its multi-attribute edge feature modeling and hierarchical attention mechanism, which enables it to extract common risk propagation patterns across ecosystems while suppressing ecosystem-specific noise. The results verify that this method has good transferability in actual supply chain governance and is suitable for cross-platform dependency risk assessment and the construction of a unified scoring system.

V. CONCLUSION

This paper addresses the risk assessment bias in industry digital frameworks caused by dependency structure complexity and missing version semantics within integrated open-source component ecosystems. By refining these technical evaluations, the research ensures greater operational stability for advanced manufacturing and technical engineering systems. It proposes a multi-attribute dependency graph modeling method based on attention-enhanced graph convolutional networks (GCNNs). By constructing a multi-dimensional graph representation that integrates dependency types, version constraints, and maintenance metadata, and designing an edge weighting mechanism driven by version intersection coverage, it significantly enhances the graph's ability to express compatibility semantics. A hierarchical edge attention (GCN) module is applied to dynamically focus on high-risk propagation paths and enhance feature aggregation of critical dependency chains. Ultimately, a fine-grained complexity score is generated that incorporates structural depth, dependency breadth, version fragmentation, and maintenance health, and gradient attribution is used to make the score interpretable. Experiments demonstrate that this method exhibits excellent scoring discrimination, scoring consistency, path identification, and transfer generalization across multiple ecosystems and scales. In large-scale scenarios, the F1-score is 0.85 ± 0.018, and the accuracy is 0.88 ± 0.015. In highly fragmented scenarios, the Spearman rank correlation coefficient is 0.90 ± 0.016, and the KL divergence is 0.32 ± 0.025. This research not only provides a quantifiable and traceable assessment framework for industry open-source software risks, but also, through semantically aware and attribution-driven design, enhances the practical guiding value of complexity metrics in manufacturing system optimization and architecture governance, providing technical support for building a highly trusted digital engineering ecosystem.

AUTHOR CONTRIBUTIONS

Conceptualization –Heng Xia and Liang Meng; methodology – Heng Xia and Liang Meng; investigation – Heng Xia and Liang Meng; writing-original draft preparation – Heng Xia and Liang Meng. All authors have read and agreed to the published version of the manuscript.

CONFLICTS OF INTEREST

The authors declare no conflict of interest.

FUNDING

This research received no external funding.

ACKNOWLEDGEMENTS

Not applicable.

REFERENCES

[1] C. Paton, J. Braa, A. Muhire, et al., "Open source digital health software for resilient, accessible and equitable healthcare systems," Yearbook of medical informatics, vol. 31, no. 01, pp. 067-073, 2022, doi: 10.1055/s-0042-1742508.

[2] S. Iqbal and M. Hamamreh J, "A comprehensive tutorial on how to practically build and deploy 5G networks using open source software and general-purpose, off-the-shelf hardware," RS Open J. Innov. Commun. Tech, vol. 2, no. 6, pp. 1-28, 2021, doi: 10.46470/03d8ffbd.4ccb7950.

[3] S. Xu, Y. Gao, L. Fan, et al., "Lidetector: License incompatibility detection for open source software," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 1, pp. 1-28, 2023, doi: 10.1145/3518994.

[4] A. Arora and C. Garman, "Analysis of software bill of materials tools," Cyber Security: A Peer-Reviewed Journal, vol. 6, no. 4, pp. 334-355, 2023, doi: 10.69554/ALLH3848.

[5] B. Schroeder A, A. Dobson E T, T. Rueden C, et al., "The ImageJ ecosystem: Open source software for image visualization, processing, and analysis," Protein science, vol. 30, no. 1, pp. 234-249, 2021, doi: 10.1002/pro.3993.

[6] J. Aggarwal and M. Kumar, "Software metrics for reusability of component based software system: a review," Int. Arab J. Inf. Technol, vol. 18, no. 3, pp. 319-325, 2021, doi: 10.34028/iajit/18/3/8.

[7] A. D'Astous, G. Cereza, D. Papp, et al., "Shimming toolbox: an open source software toolbox for B0 and B1 shimming in MRI," Magnetic resonance in medicine, vol. 89, no. 4, pp. 1401-1417, 2023, doi: 10.1002/mrm.29528.

[8] H. Wang, S. Yu, C. Chen, et al., "Beyond accuracy: an empirical study on unit testing in open source deep learning projects," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 4, pp. 1-22, 2024, doi: 10.1145/3638245.

[9] T. Bock, A. Schmid, and S. Apel, "Measuring and modeling group dynamics in open source software development: A tensor decomposition approach," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 31, no. 2, pp. 1-50, 2021, doi: 10.1145/3473139.

[10] W. Meng, F. Zaiter, Y. Zhang, et al., "Logsummary: Unstructured log summarization for software systems," IEEE Transactions on Network and Service Management, vol. 20, no. 3, pp. 3803-3815, 2023, doi: 10.1109/TNSM.2023.3236994.

[11] A. Bemis K, C. Föll M, D. Guo, et al., "Cardinal v," 3: a versatile open source software for mass spectrometry imaging analysis. Nature Methods, vol. 20, no. 12, pp. 1883-1886, 2023, doi: 10.1038/s41592-023-02070-z.

[12] M. Souppaya, K. Scarfone, and D. Dodson, "Secure software development framework (ssdf) version 1.1," NIST Special Publication, vol. 800, no. 218, pp. 800-218, 2022, doi: 10.6028/NIST.SP.800-218.

[13] K. Shah and V. Prabhakar T, "Construction of a digital twin framework using free and open source software programs," IEEE Internet Computing, vol. 26, no. 5, pp. 50-59, 2021, doi: 10.1109/MIC.2021.3051798.

[14] S. Peldszus, D. Brugali, D. Strüber, et al., "Software reconfiguration in robotics," Empirical Software Engineering, vol. 30, no. 3, pp. 1-51, 2025, doi: 10.1007/s10664-024-10596-9.

[15] P. Kumar, N. Singh S, and S. Dawra, "Software component reusability prediction using extra tree classifier and enhanced Harris hawks optimization algorithm," International Journal of System Assurance Engineering and Management, vol. 13, no. 2, pp. 892-903, 2022, doi: 10.1007/s13198-021-01359-6.

[16] M. Mustaqeem and M. Saqib, "Principal component based support vector machine (PC-SVM): a hybrid technique for software defect detection," Cluster Computing, vol. 24, no. 3, pp. 2581-2595, 2021, doi: 10.1007/s10586-021-03282-8.

[17] J. Saxon, J. Koschinsky, K. Acosta, et al., "An open software environment to make spatial access metrics more accessible," Journal of Computational Social Science, vol. 5, no. 1, pp. 265-284, 2022, doi: 10.1007/s42001-021-00126-8.

[18] C. Osborne, F. Daneshyan, R. He, et al., "Characterising open source co-opetition in company-hosted open source software projects: the cases of PyTorch, TensorFlow, and transformers," Proceedings of the ACM on Human-Computer Interaction, vol. 9, no. 2, pp. 1-30, 2025, doi: 10.1145/3710944.

[19] A. Ram and K. Chakraborty S, "Analysis of software-defined networking (sdn) performance in wired and wireless networks across various topologies, including single, linear, and tree structures," Indian Journal of Information Sources and Services, vol. 14, no. 1, pp. 39-50, 2024, doi: 10.51983/ijiss-2024.14.1.3926.

[20] K. Blind and T. Schubert, "Estimating the GDP effect of Open Source Software and its complementarities with R&D and patents: evidence and policy implications," The Journal of Technology Transfer, vol. 49, no. 2, pp. 466-491, 2024.

[21] M. Ferreira, M. Monteiro, T. Brito, et al., "Efficient static vulnerability analysis for javascript with multiversion dependency graphs," Proceedings of the ACM on Programming Languages, vol. 8, no. PLDI, pp. 417-441, 2024.

[22] Y. Zhuo, J. Chen, G. Rao, et al., "Distributed graph processing system and processing-in-memory architecture with precise loop-carried dependency

guarantee," ACM Transactions on Computer Systems (TOCS), vol. 37, no. 1-4, pp. 1-37, 2021.
[23] S. Kumar J, B. Archana, K. Muralidharan, et al., "Graph Theory: Modelling and Analyzing Complex System," Metallurgical and Materials Engineering, vol. 31, no. 3, pp. 70-77, 2025.
[24] D. Clementel, A. Del Conte, M. Monzon A, et al., "RING 3.0: fast generation of probabilistic residue interaction networks from structural ensembles," Nucleic acids research, vol. 50, no. W1, Art. no. W651-W656, 2022.
[25] Y. Zhang and M. Tang, "A theoretical analysis of deepwalk and node2vec for exact recovery of community structures in stochastic blockmodels," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 46, no. 2, pp. 1065-1078, 2023.
[26] T. Zhou, R. Pan, J. Zhang, et al., "An attribute-based Node2Vec model for dynamic community detection on coauthorship network," Computational Statistics, vol. 40, no. 1, pp. 177-204, 2025.
[27] A. Ajibode, A. Bangash A, R. Cogo F, et al., "Towards semantic versioning of open pre-trained language model releases on hugging face," Empirical Software Engineering, vol. 30, no. 3, pp. 1-63, 2025.
[28] R. Kathi S, "AI-Assisted Dependency Vulnerability Resolution in Large-Scale Enterprise Systems," International Research Journal of Advanced Engineering and Technology, vol. 2, no. 07, pp. 8-18., 2025.
[29] X. Cheng, H. Wang, J. Hua, et al., "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 3, pp. 1-33, 2021.
[30] Z. Liu, P. Qian, X. Wang, et al., "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," IEEE Transactions on Knowledge and Data Engineering, vol. 35, no. 2, pp. 1296-1310, 2021.
[31] W. Tang, H. Sun, J. Wang, et al., "Identifying users across social media networks for interpretable fine-grained neighborhood matching by adaptive gat," IEEE Transactions on Services Computing, vol. 16, no. 5, pp. 3453-3466, 2023.
[32] J. Jiang, P. Guo, X. Xu, et al., "Social perception with graph attention network for recommendation," ACM Transactions on Recommender Systems, vol. 4, no. 1, pp. 1-21, 2025.
[33] C. Wen X, C. Gao, J. Ye, et al., "Meta-path based attentional graph learning model for vulnerability detection," IEEE Transactions on Software Engineering, vol. 50, no. 3, pp. 360-375, 2023.
[34] Y. Zeng, Y. Fang, W. Luo, et al., "Accelerating Skyline Path Enumeration with a Core Attribute Index on Multi-attribute Graphs," Proceedings of the ACM on Management of Data, vol. 3, no. 3, pp. 1-26, 2025.
[35] H. Chen, J. Jiang, and N. Zheng, "Learning to infer unseen single-/multi-attribute-object compositions with graph networks," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 45, no. 10, pp. 12022-12037, 2023.
[36] L. Huang, X. Liu X, Q. Huang S, et al., "Temporal hierarchical graph attention network for traffic prediction," ACM Transactions on Intelligent Systems and Technology (TIST), vol. 12, no. 6, pp. 1-21., 2021.