Analyzing Software Evolution Dynamics using Execution Path-aware Graph Divergence

Fitra Arifiansyah, Muhammad Zuhri Catur Candra* & Gusti Ayu Putri Saptawati

School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Jalan Ganesa No. 10 Bandung, 40132, Indonesia
*E-mail: mzc@itb.ac.id

Abstract. Software comprehension is a fundamental activity in software maintenance, and its complexity grows as systems evolve across releases. Call graphs (CG) are widely used to support this process because they capture the calling relationships among functions. However, obtaining meaningful comparisons between different versions of a CG remains challenging. Network Portrait Divergence (NPD) provides a graph invariant and computationally efficient metric for assessing structural differences by analyzing global distributions of node distances and neighborhood patterns. Although it is effective, NPD does not include execution semantics, even though execution paths often convey the behavioral changes that matter to developers. This study introduces a refinement of NPD that replaces neighborhood-oriented features with features derived from execution paths, represented through distributions of path lengths. The modified metric is evaluated in a controlled scenario using synthetic data, designed to distinguish the effects of structural changes including new call and changes to control flow. The results show that the proposed NPD is more responsive to modifications that influence execution behavior. Scenarios that create new execution paths and restructure control flow result in substantially higher divergence. These findings suggest that including execution path information offers a more behavior-oriented view of software evolution and complements topology-based approaches.

Keywords: *call graph analysis; divergence measurement; execution path; network portrait; software comprehension; software evolution.*

1 Introduction

Software comprehension is a critical prerequisite in various software development activities such as maintenance, testing, and quality management [1, 2]. Without a proper understanding of the software and its interactions, adding or removing features can introduce risks and unexpected behaviors in the system [3]. Activities such as maintenance, testing, and quality management are performed iteratively throughout the software development process, resulting in continuous growth of the system which is commonly referred to as software evolution. This evolution causes the size and complexity of software systems to increase rapidly over time. In each release version of the software, developers

Received March 5h, 2026, Revised May 16th, 2026, Accepted for publication June 10th, 2026.

often need to understand the system at different levels of granularity and the detail. This problem becomes even more complicated when investigating changes across multiple software releases. Software comprehension, particularly at the source code level, represents one of the most time-consuming activities in software development [4, 5]. An effective representation derived from the source code is a graph [6]. A graph can capture control flow, data flow, or dependency relationships within a program. Such a representation provides a clearer view of the overall structure of the code and the interactions between its components [7]. Using graphs as a tool for program comprehension has been shown to be more effective than traditional methods that rely solely on reading the code [8]. Among various types of graphs, a call graph is one of the most useful. A call graph is a visual representation of a software system in which each node represents a function, and each directed edge represents a function call [9, 10]. Static call graphs have proven to be effective in helping developers understand system interactions and structural relationships. For example, the tool in [11] can detect and visualize static interactions between classes in Java-based software. Another tool, Code2Graph [12], can automatically construct and visualize static call graphs for Python programs. Several studies [13, 14, 15, 7] have focused on creating visualization tools that enhance the graphical representation of call graphs to support program comprehension. There are other studies [16, 17, 18, 19] that create various visual representations of call graphs that are primarily used to generate graphical representations. Similarly, GraphEvo [3] utilizes static call graphs to visualize software evolution using a monolithic color-coded graph.

Execution paths are sequences of method call that describe how a program flows during execution [20]. It plays a central role in understanding software behavior [21]. When software evolves, these paths often change in subtle but meaningful ways, reflecting how new features, modifications, or refactoring change the system's logic. For developers, examining execution paths is therefore one of the most informative methods for understanding how a system behaves across different versions, particularly during maintenance and regression analysis where behavioral differences must be clearly identified.

Furthermore, due to the continuous evolution of software systems discussed earlier, program comprehension now faces a new challenge. This growth makes it increasingly difficult for developers to understand the interactions within the system, especially when examining how these interactions change across versions [18]. The challenge is intensified by the fact that the graphs produced during software evolution are not static, each software release represents a distinct snapshot in which nodes and edges may be added, removed, or modified. A dynamic graph can thus be viewed as a sequence of static graph snapshots across different time intervals [22]. To address this, graph summarization techniques are required. The goal of summarization is to simplify a graph into a higher-level

view while preserving essential information [23]. One common approach is clustering, particularly hierarchical clustering [13], which allows the granularity of the summarized graph to be adjusted according to the developer's comprehension needs. For example, HCSumm [24, 25] focuses on minimizing information loss in the summarization process. Summarizing dynamic graphs introduces additional complexity. Techniques designed for static graphs cannot be directly applied because reprocessing the entire graph from scratch at every change would be inefficient. Instead, summarizing information from previous versions should be reused [26]. One possible solution is micro clustering, in which clustering is performed incrementally at each time step, eliminating the need to recompute the entire graph whenever a node or edge changes.

In modern large-scale software development, collaboration among multiple developers is inevitable. Consequently, Version Control Systems (VCS) have become an essential tool that aids software comprehension [27]. The use of VCS transforms the generated source code graph into a dynamic one, as each commit represents a snapshot of the source code over time. One such metric is Network Portrait Divergence (NPD), which measures the divergence between two graphs by comparing global distributions of node distances and neighborhood patterns [28]. NPD is attractive and provides valuable structural insight because it is graph-invariant and does not require node correspondence [29].

However, NPD relies solely on topological features and does not incorporate execution semantics, even though execution paths are often the most meaningful indicators of behavioral changes in software [30]. Prior studies have shown that execution behavior and path level information provide a more faithful basis for analyzing software evolution [31, 32]. This may create a gap between what the metric measures and what developers truly need to understand. Motivated by this gap, this study addresses the following research question: "Does incorporating execution path information into NPD lead to divergence measurements that have better represent actual behavior and structural changes?"

This study hypothesizes that adding execution path will provide better divergence measurement that represents structural changes. To explore this idea, this study introduces execution path and path length as new features, replacing the original neighborhood and hop distance attributes in NPD. These execution path features aim to make the metric more aligned with the way software changes in software evolution.

The remains of this paper are organized as follows. Section 2 reviews related studies on GraphEvo, NPD, and execution path-based software evolution analysis. Section 3 present the proposed method, including the modified procedure for computing NPD based on execution path. Section 4 presents and

discusses the evaluation using several scenario setups and its findings. Finally, section 5 concludes the paper and proposes future directions to enhance this work.

2 Related Studies

Research on software evolution and software maintenance focuses on understanding how software changes over time due to maintenance activities, feature additions, and refactoring. Software repository analysis is a primary approach to studying this evolution, both in terms of code structure and the patterns of change that occur within it. Software evolution is reflected not only in the increase in code size, but also in changes in the relationships between components and their implications for the overall behavior of the system [33, 34]. This complexity makes understanding software evolution a significant challenge, especially when changes need to be analyzed across multiple versions.

In order to support program comprehension, various graph visualization and representation techniques have been developed. A systematic review shows that graph-based representations are one of the most common and effective approaches [35, 36, 37]. A call graph is one such graph-based representation that represents the calling relationships between functions and provides a clear interaction of software components. Several subsequent studies have highlighted how call graphs can be used to improve program comprehension, including through clustering, topological analysis, and network characteristic measurements [38, 39, 40].

In the context of software evolution analysis based on call graph, GraphEvo introduced as an approach to characterize and understand software changes between versions [3]. GraphEvo automatically constructs a static call graph for each software version and visualizes it to highlight the structural changes that occur. This approach aims to answer evolutionary questions such as what functions are added, removed, or modified, and how call relations change from one version to the next. A similar idea regarding graph divergence-based change analysis is also discussed in another paper [41], which shows that differences in the structure of the call graph can indicate significant changes in the system.

GraphEvo's workflow consists of two primary stages. The first stage builds call graphs for each software release. These graphs provide insights into how functions interact within the software. The second stage calculates various graph metrics across different software versions to measure how the structure evolves over time. Moreover, the tool provides comparison and feature analysis capabilities that help developers better understand code changes and execution differences across releases.

To measure structural differences between graphs, GraphEvo utilizes Network Portrait Divergence (NPD). NPD is a topology-based metric that compares two graphs through the global distribution of distances between nodes [29, 28]. This approach is graph invariant, independent of node order or graph size, thus allowing structural comparisons without explicit node mapping [42, 28]. The network portrait concept is also used in other domains, such as in the film domain and other networks that are not always based on a call graph [43, 44], showing that the distribution of distances and global connectivity patterns can effectively represent the structural characteristics of a network. NPD represents the statistical distribution of distance patterns in a network. A network portrait describes how many nodes have *k* neighbors at a distance *l*. Each row *l* corresponds to a specific hop distance from a node, while each column *k* indicates the number of nodes with k neighbors at that distance.

To compute divergence between software releases, GraphEvo first generates Network Portrait (NP) to determine how many other nodes can be reached from each node within n hops. The next step is to calculate the distances for 1-hop and 2-hop connections. Here, a hop refers to the distance between the source node and the next node connected by an edge. For each node, the number of neighbors reachable at each hop distance is identified. The hop distance is denoted by *l*, and *k* represents the number of neighbors within that distance. After the values of *k* and *l* are identified, a distribution is constructed to represent the number of nodes that fall under each hop. Once the probability distribution for each hop distance has been generated, the next step would be to merge the probability distributions from multiple software versions to calculate the Jensen–Shannon Divergence (JSD).

JSD is a measure of the difference between two probability distributions that are symmetric and bounded, making it easy to interpret [45]. The use of JSD as a measure of distribution difference is also often discussed in studies that utilize entropy and information-based approaches to compare network structures [43, 42, 46]. It reflects how many nodes in each graph have *k* neighbors at a distance *l*. In other words, JSD quantifies the difference between the (*l*, *k*) distributions of version 1 and version 2. This divergence represents the distance between two probability distributions, P and Q, derived from the graphs. It is based on the Kullback–Leibler (KL) divergence and is calculated using the following Eq. (1):

$$D_{JS}(P \parallel Q) = \tfrac{1}{2} D_{KL}(P \parallel M) + \tfrac{1}{2} D_{KL}(Q \parallel M) \quad (1)$$

where P is the probability distribution of the first graph, Q is that of the second graph, and M is the mixed distribution of P and Q, computed using the coefficient ½ for each. The advantage of JSD is that it is always bounded between 0 and 1, and the result is symmetric, meaning that $D_{JS}(P \mid \mid Q) = D_{JS}(Q \parallel P)$. This makes it

particularly useful for measuring structural divergence between different software versions in a consistent way. The higher the JSD value, the greater the structural difference between the two versions of the software.

3 Proposed Method

Although GraphEvo is effective in capturing global structural changes in call graphs, several studies have shown their limitations in representing program execution semantics. Several studies emphasize that program understanding is highly dependent on how functions are executed and how execution paths are formed, rather than only on static connectivity structures [47, 48]. Similarly, other research suggests that graph simplifications and structural metrics need to align with how humans understand program behavior [49].

Based on these findings, a gap between topology-based software evolution analysis and the need to more explicitly represent execution behavior, this work begins with a direct assumption that changes in a software system are often easier to observe through the execution paths that the program can follow, rather than through general topological information in its call graph. A small adjustment in the code, sometimes as minor as inserting a new function can shift how the program flows, even when the graph topology around the affected nodes remains almost unchanged. From this observation, we hypothesize that proposing execution path information into the Network Portrait Divergence (NPD) framework will produce divergence values that more accurately reflect the structural and behavioral differences between software versions.

3.1 Execution Path as a Feature

In this study, an execution path refers to the ordered sequence of function calls that starts from an initial point and terminates when no further call is reachable. Developers often rely on such paths to track feature behavior or follow a chain of calls during debugging, but the original NPD formulation does not explicitly use this information. In the original NPD, the network portrait is defined using two dimensions: l represents the hop distance from a node, and k represents the number of nodes reachable at that distance. This formulation is suitable for generic network comparison because it captures global neighborhood and distance patterns. However, it does not preserve the ordering of function calls, which is important in software execution.

To integrate execution path semantics, this study preserves the portrait structure but changes the meaning of its two dimensions. In the proposed formulation, l represents the length of an execution path, measured by the number of function calls in the sequence, while k represents the number of execution paths with

length *l*. Thus, each (*l*, *k*) pair no longer describes how many neighbors are reachable at a given hop distance, but how many execution paths share a given path length. The probability distribution and JSD calculation remain consistent with the original NPD, but the information encoded in the portrait is shifted from topological neighborhood structure to execution path behavior.

3.2 Modified Steps

The proposed divergence computation follows the general workflow of the original NPD, but replaces the neighborhood based portrait with an execution path based portrait. Instead of counting how many nodes are reachable at a particular hop distance, the proposed method counts how many execution paths have a particular path length. To make the procedure clearer and more reproducible, the complete computation is summarized in Algorithm 1.

Algorithm 1. Execution path-based NPD computation

Input: Two directed call graphs G1 = V1, E1 and G2=V2, E2
Output: Execution path-based divergence DJS(P || Q)

1. Identify entry nodes in each graph according to the selected entry point policy.
2. Enumerate simple execution paths from the entry nodes in G1 and G2.
3. Stop a path when no unvisited successor is reachable.
4. For each graph, compute the length l of each execution path.
5. Count the number of execution paths k for each path length l.
6. Construct execution path portraits P and Q using the resulting (l, k) entries.
7. Normalize P and Q into probability distributions.
8. Align the portrait keys of P and Q.
9. Compute the mixed distribution M = ½ (P + Q).
10. Compute DJS(P || Q) = ½ DKL (P || M) + ½ DKL (Q || M).
11. Return DJS(P || Q).

In this study, entry nodes are defined as nodes with zero in-degree in the extracted call graph. These nodes represent functions that are not called by other functions within the same graph and therefore serve as starting points for path enumeration. The same entry point policy is applied consistently to all compared versions to ensure comparability between portraits. If a software version introduces or removes entry nodes, the resulting divergence may reflect both structural changes and changes in the possible starting points of execution. This behavior is intentional in the current formulation because entry point changes also indicate changes in how execution paths may be formed.

The current implementation enumerates simple paths only. A simple path does not contain repeated nodes, which prevents infinite traversal in the presence of cycles or recursive calls. As a result, recursion and mutually recursive components are not unrolled indefinitely in this study. While this choice keeps path enumeration finite and reproducible, it may underrepresent recursive behavior.

From a computational perspective, the most expensive part of the proposed method is execution path enumeration. Since the method enumerates simple paths, the number of possible paths can grow rapidly in graphs with high branching factors. In the worst case, simple path enumeration may grow exponentially with the number of nodes. After the paths have been enumerated, constructing the execution path portrait is linear in the number of enumerated paths, because each path is assigned to its corresponding path length group. The alignment of portrait keys and the JSD computation are linear in the number of distinct (l, k) entries. Therefore, the scalability of the proposed method is mainly determined by the number of execution paths produced from the call graph.

3.3 Evaluation Plan

The approach will be evaluated through three scenarios:

1. A small illustrative example, used to show how the modified NPD behaves under controlled structural changes and then comparing the execution path-based NPD with the original GraphEvo
2. Synthetic graph datasets of various sizes and densities, allowing systematic comparison with the original formulation.
3. Real world case study, used to ensure that proposed NPD works as desired in real case.

These three evaluation settings are intended to provide a balanced view of how the proposed modification behaves in different conditions, ranging from simple, controlled structures to more realistic and irregular cases, and complemented by a real world case using a real project from open source repository. Through this combination, we expect to see whether the execution path-based portrait consistently produces divergence values that make sense relative to the actual changes in the code, and whether it offers a clearer picture of software evolution compared to the original NPD formulation.

4 Evaluation and Discussion

Based on the proposed method section, this study evaluates the proposed solution through three different settings. The first setting uses a small controlled example to make the calculation steps transparent and easier to follow. The second setting

relies on synthetic graphs of varying sizes and controlled modification scenarios to observe how the method responds when the graph structure becomes more complex. The third setting uses a real world case study based on selected modules from the Flask open source project to examine whether the proposed method can also capture changes in naturally evolving software. These three settings are discussed separately because they highlight different aspects of how the modified NPD behaves.

Since the proposed method directly modifies the portrait definition used in NPD, the original NPD is used as the primary baseline throughout the evaluation. This allows the experiment to isolate the effect of replacing neighbourhood based features with execution path based features while keeping the divergence computation consistent. Broader comparisons with other graph distance measures, such as graph edit distance, spectral distance, or degree distribution divergence, are outside the scope of the current study and are considered as future work.

Through these evaluation settings, our aim is to examine whether the execution path based portrait provides a more meaningful reflection of software evolution than the original formulation, particularly when changes affect execution paths, call ordering, and control flow structure.

4.1 Small Illustrative Example

To further test the proposed feature, this experiment was conducted using a simple example that includes execution paths that have more than two hops and introduces both new nodes and cyclic edges between connected nodes. We try to compare between the original NPD and the proposed execution path-based using the example. This case, illustrated in Figure 1, serves as the first test scenario.

Figure 1 Call graph example: version 1 (v_1) and version 2 (v_2).

Following the original steps, distances were calculated for 1-hop through 4-hop connections in both v_1 and v_2. The next step is to construct the distribution based

on the identifications summarized in Table 1 for each hop and version, both v_1 and v_2.

Table 1 Calculation of k for each l in v_1 and v_2.

v_1			v_2		
Node	**Next Node**	***k***	**Node**	**Next Node**	***k***
1-hop ($l = 1$)					
A	B, C, D	3	A	B, C, D	3
B	D	1	B	D	1
C	A, B	2	C	A, B	2
D	-	0	D	E	1
			E	-	0
2-hop ($l = 2$)					
A	B, D	2	A	B, D, E	3
B	-	0	B	E	1
C	B, D	2	C	B, D	2
D	-	0	D	-	0
			E	-	0
3-hop ($l = 3$)					
A	D	1	A	B, E	2
B	-	0	B	-	0
C	D	1	C	D, E	2
D	-	0	D	-	0
			E	-	0
4-hop ($l = 4$)					
			A	E	1
			B	-	0
			C	E	1
			D	-	0
			E	-	0

The resulting distributions are then presented in Table 2. After obtaining the distribution for each hop (l) as shown in Table 2, the next step is to construct the network portrait, which represents the set of (l, k) pairs.

Table 2 Distribution of k for each l in v_1 and v_2.

v_1		v_2	
k	**Count of node**	***k***	**Count of node**
1-hop ($l = 1$)			
0	1 (D)	0	1 (E)
1	1 (B)	1	2 (B, D)
2	1 (C)	2	1 (C)
3	1 (A)	3	1 (A)
2-hop ($l = 2$)			
0	2 (B, D)	0	2 (D, E)
2	2 (A, C)	1	1 (B)
		2	1 (C)
		3	1 (A)
3-hop ($l = 3$)			
0	2 (B, D)	0	3 (B, D, E)
1	2 (A, C)	2	2 (A, C)

v_1		v_2	
k	**Count of node**	***k***	**Count of node**
4-hop ($l = 4$)			
		0	3 (B, D, E)
		1	2 (A, C)

The same procedure is applied as described earlier in Section 3. Table 3 presents the probability values for each (l, k) pair generated in both versions, v_1 and v_2.

Table 3 Probability for each l, k in v_1 and v_2.

v_1			v_2		
(*l*, *k*)	**Count of node**	**Probability**	**(*l*, *k*)**	**Count of node**	**Probability**
(1,0)	1	0.083	(1,0)	1	0.050
(1,1)	1	0.083	(1,1)	2	0.100
(1,2)	1	0.083	(1,2)	1	0.050
(1,3)	1	0.083	(1,3)	1	0.050
(2,0)	2	0.167	(2,0)	2	0.100
(2,2)	2	0.167	(2,1)	1	0.050
(3,0)	2	0.167	(2,2)	1	0.050
(3,1)	2	0.167	(2,3)	1	0.050
			(3,0)	3	0.150
			(3,2)	2	0.100
			(4,0)	3	0.150
			(4,1)	2	0.100

The next step follows the same procedure as in the previous section, which involves combining the distributions of P (v_1) and Q (v_2), as shown in Table 4.

Table 4 Merge distribution between v_1 and v_2.

(*l*, *k*)	**Probability v_1**	**Probability v_2**
(1,0)	0.083	0.050
(1,1)	0.083	0.100
(1,2)	0.083	0.050
(1,3)	0.083	0.050
(2,0)	0.167	0.100
(2,1)	0	0.050
(2,2)	0.167	0.050
(2,3)	0	0.050
(3,0)	0.167	0.150
(3,1)	0.167	0
(3,2)	0	0.100
(4,0)	0	0.150
(4,1)	0	0.100

After that, we computed the JSD using equation (1), which resulted in a value of $D_{JS}(P \parallel Q)$ is 0.348. This NPD value indicates that structural changes occurred, but not in an extreme way, since it remains above the threshold of 0.3. In the

context of GraphEvo, this suggests a form of functional evolution, such as the addition of new functions or execution paths.

Once the analysis of Figure 1 using the original features of GraphEvo was completed, the next step was to perform the same analysis using the proposed features, namely execution path and path length. The overall procedure remains identical, except that the definitions of l and k differ under the proposed approach. The first step is identifying the execution paths for both v_1 and v_2. Based on Figure 1, the v_1 contains seven execution paths as follows:

- A → B → D
- A → C → B → D
- A → D
- B → D
- C → A → B → D
- C → A → D
- C → B → D

Meanwhile, the v_2 contains eight execution paths:

- A → B → D → E
- A → C → B → D → E
- A → D → E
- B → D → E
- C → A → B → D → E
- C → A → D → E
- C → B → D → E
- D → E

After identifying all execution paths, the lengths of the paths between v_1 and v_2 were then calculated, as shown in Table 5.

Table 5 Length of the path v_1 and v_2.

Node	$l = 1$		$l = 2$		$l = 3$		$l = 4$	
	v_1	v_2	v_1	v_2	v_1	v_2	v_1	v_2
A	1	-	1	1	1	1	-	1
B	1	-	-	1	-	-	-	-
C	-	-	2	-	1	2	-	1
D	-	1	-	-	-	-	-	-
E	-	-	-	-	-	-	-	-

The next step is to construct the distributions based on Table 5 to generate the Network Portraits (NP) for both v_1 and v_2. The results are presented in Table 6.

Table 6 The distribution of each l in v_1 and v_2.

v_1		v_2	
k	Count of node	k	Count of node
0	2 (C, D)	0	4 (A, B, C, E)
1	2 (A, B)	1	1 (D)
0	2 (B, D)	0	3 (C, D, E)
1	1 (A)	1	2 (A, B)
2	1 (C)	-	-
0	2 (B, D)	0	3 (B, D, E)
1	2 (A, C)	1	1 (A)
-	-	2	1 (C)
-	-	0	3 (B, D, E)
-	-	1	2 (A, C)

As in the previous example, the next step is to calculate the distribution of occurrences for each specific (l, k) pair in both v_1 and v_2. The results of this calculation are shown in Table 7.

Table 7 Probability for each l, k in v_1 and v_2.

(l, k)	v_1		v_2	
	Count of node	Probability	Count of node	Probability
(1,0)	2	0.167	4	0.200
(1,1)	2	0.167	1	0.050
(2,0)	2	0.167	3	0.150
(2,1)	1	0.083	2	0.100
(2,2)	1	0.083	-	-
(3,0)	2	0.167	3	0.150

(*l*, *k*)	v_1		v_2	
	Count of node	**Probability**	**Count of node**	**Probability**
(3,1)	2	0.167	1	0.050
(3,2)	-	-	1	0.050
(4,0)	-	-	3	0.150
(4,1)	-	-	2	0.100

Finally, the distributions of v_1 and v_2 are combined, and the result is presented in Table 8. After constructing the merged distribution of P (v_1) and Q (v_2), the next step is to calculate the divergence between them using equation (1). The resulting value of $D_{JS}(P \| Q)$ is 0.242. This indicates a measurable divergence, but it does not exceed the 0.3 threshold used in GraphEvo to indicate a major structural difference. When compared with the NPD value obtained from the original GraphEvo features (0.348), the proposed feature set produces a noticeably different outcome. Using the original features leads to the detection of a major structural change, while the proposed features produce a lower divergence value, implying that the change is less extensive. As illustrated in Figure 1, only one node and one edge were added. Therefore, the modification is relatively minor. However, because the proposed approach incorporates execution paths, the number of identified paths increases significantly, which in turn affects the divergence value. This observation explains the substantial difference between the NPD computed using the original GraphEvo features and that obtained with the proposed method.

Table 8 Merge distribution between v_1 and v_2.

(*l*, *k*)	v_1		v_2	
	Count of nodes	**Probability**	**Count of nodes**	**Probability**
(1,0)	2	0.167	4	0.200
(1,1)	2	0.167	1	0.050
(2,0)	2	0.167	3	0.150
(2,1)	1	0.083	2	0.100
(2,2)	1	0.083	0	0
(3,0)	2	0.167	3	0.150
(3,1)	2	0.167	1	0.050
(3,2)	0	0	1	0.050
(4,0)	0	0	3	0.150
(4,1)	0	0	2	0.100

4.2 Synthetic Graph Dataset

To better understand how the proposed execution path feature behaves under different types of structural changes, we evaluate it using three synthetic graph pairs representing small, medium, and large evolution scenarios. These cases are designed to simulate realistic patterns of software evolution there are minor corrections, incremental feature additions, and more substantial architectural extensions. By varying the number of nodes, edges, and call-graph relationships, we could observe how both divergence metrics there are the original NPD and the execution path-based variant, respond to differences in size and structure. This setup also allows us to examine whether the proposed method is consistent across graph scales and whether it captures behavioral changes that the neighborhood-based portrait may overlook. The characteristics of each graph pair and the result of divergence values are summarized in Table 9.

Across these three synthetic cases, the divergence values reveal clear differences in how the original NPD and the pro-posed execution path feature respond to structural changes. In the Small case, only a minor modification was introduced by replacing a recursive implementation of *modulus* with a direct computation. Although this change removes one call and eliminates an execution path, the overall structure of the program remains almost the same. As a result, the original NPD produces a small divergence (0.113), whereas the execution path features have a higher value (0.667), indicating that it is more sensitive to change that affects the flow of control.

Table 9 Divergence comparison in synthetic graph.

Graph Scale	Number of Nodes		Number of Edges		NPD	
	v_1	v_2	v_1	v_2	Original	Proposed
Small	6	6	5	3	0.113	0.667
Medium	9	8	9	9	0.006	0.473
Large	15	15	18	23	0.648	1.000

The Medium case highlights how a minor structural change can still influence execution behavior. The only difference between two versions is the addition of a call from *compute stats* to normalize, which introduces a new branch in the execution flow. While the original NPD captures this as a very small structural adjustment (0.006), suggesting little change at the graph topology level. However, the execution path features produce a noticeably higher divergence (0.473). This suggests that even a modest change in the ordering of function calls can alter how the program flows at runtime. In this case, the proposed method highlights behavioral differences that are not easily seen through the neighborhood-based portrait. Finally, the Large case shows the behavior of both metrics under more substantial evolution. The second version includes several new helper functions

and additional analysis and logging steps, expanding both the size and depth of the call graph. The original NPD reports a noticeable divergence (0.648), indicating significant structural growth. The execution path divergence again reaches 1.00, reflecting the introduction of multiple new execution chains and deeper call sequences. In this scenario, both metrics indicate meaningful change, but the execution path version more clearly captures the scale of behavioral expansion. In addition to these three cases, we created a set of controlled based on the small example to observe how each NPD (the original vs the proposed) responds to isolated structural modification. This additional evaluation is intended to provide insight into the sensitivity of both versions of NPD when the call graph changes in specific and well-defined ways. Two types of modifications were introduced: the first one is adding a new function which increases the number of nodes and the second one is adding new call function which increases the number of edges.

To support this additional evaluation, we defined several test scenarios that describe incremental modifications made to the base graph. The purpose of these scenarios is not to evaluate the correctness, but to examine how each NPD formulation responds when the graph is modified in a controlled manner. Table 10 shows the scenarios used in this sensitivity analysis, describing the type of modification applied.

The results of the controlled scenarios in Table 10 provide a clear view of how the two divergence metrics respond to isolated structural modifications in the call graph. Across all variations, the original NPD shows a steady and relatively conservative pattern: divergence increases when nodes or edges are added, but the magnitude generally reflects structural expansion rather than shifts in execution behavior. This outcome is consistent with its formulation as a topology-oriented metric that captures changes in neighborhood distributions and the overall graph structure.

In contrast, the proposed execution path-based NPD produces a wider and more expressive range of values. Scenarios that introduce new callable functions such as G2, G3, G6, and G8 lead to noticeably larger divergences because they create additional execution chains. Modifications that alter the internal flow of existing functions, including wrapping or branching (G4, G9, G10, G11), produce even sharper increases, indicating a higher sensitivity to changes in control-flow structure. These findings suggest that the proposed metric is more sensitive to changes that affect control flow organization.

Table 10 Sensitivity evaluation scenarios based on incremental structural and behavioral modifications.

sGraph ID	Diff Nodes vs G0	Diff Edge vs G0	Modification	Flow Impact Categories	NPD Original	NPD Proposed
G0	-	-	*baseline*	*baseline*	-	-
G1	1	0	add unused function	none	0.333	0.331
G2	1	1	new function called	new path	0.333	0.797
G3	1	2	add function calls	new path	0.308	0.797
G4	1	1	wrapper changes main flow	restructured path	0.413	0.595
G5	2	0	add two unused functions	none	0.333	0.393
G6	2	2	add two function calls	new path	0.667	0.827
G7	3	0	add three unused functions	none	0.333	0.442
G8	3	3	add three function calls	new path	0.667	0.848
G9	1	2	call preprocess before multiply	restructured path	0.308	0.797
G10	1	1	branching in multiply	restructured path	0.595	1
G11	2	2	wrapper and branching	restructured path	0.437	1

These scenarios represent changes in which the graph increases in size while the program's execution behavior remains unchanged. As shown in Figure 2, the original NPD produces small and relatively stable divergence values, indicating only minor structural adjustments. The proposed NPD, however, shows a proportional increase that follows the number of additional nodes. This occurs because the path-length distribution is normalized across all observed paths. Adding functions that are not reachable from the main entry point still affects this distribution by altering the relative weights of existing groups. The resulting divergence therefore reflects structural growth rather than behavioral change.

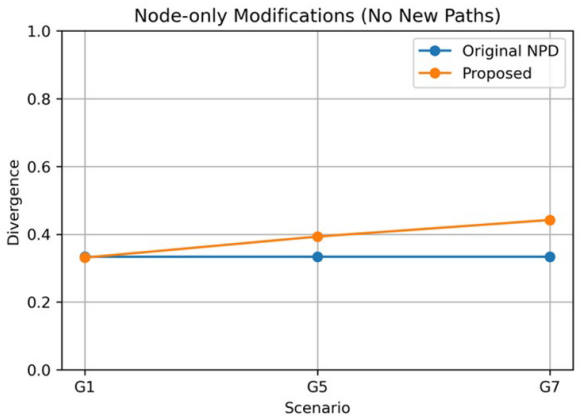

Figure 2 Divergence values for node only modifications.

When the scenarios are grouped according to their behavioral impact, as shown in Figure 3, a clear pattern emerges. Divergence values are smallest in cases where no new execution paths are created, increase when new paths are introduced, and reach their highest levels when internal branching or restructuring modifies the flow within existing functions. This progression aligns well with the intuitive severity of each type of modification. Scenarios such as G9 belong to this restructuring group, because the addition of a preprocessing step changes the ordering of the original flow rather than simply adding a new branch.

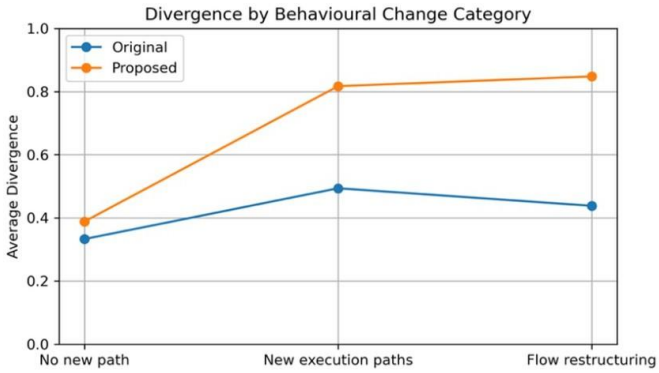

Figure 3 Divergence patterns across behavioral change categories.

Although one might initially expect unused function scenarios (G1, G5, G7) to yield zero divergence since they do not alter execution behavior, the results reveal a different pattern. The proposed metric constructs a distribution of path lengths and normalizes it across all observations. Adding unreachable functions

indirectly modifies this distribution by adjusting the total denominator, which in turn shifts the portrait even though the set of execution paths remains unchanged. Rather than representing a false signal, this behavior reflects structural evolution within the software. In practice, the addition of unused functions often corresponds to architectural preparation, refactoring activity, or scaffolding for future features. The steady increase in divergence across these cases indicates that the proposed metric reacts predictably to such changes.

Scenarios that combine new nodes with new edges (G2, G3, G6, G8) behave differently, producing substantially larger divergences. These cases affect both the size of the graph and its execution behavior, leading to a dual impact on the path-length distribution. Modifications to internal control flow (G4, G9, G10, G11) produce the largest divergences, highlighting the ability of the proposed method to capture execution-level structural changes rather than only static topology.

Taken as a whole, these patterns show that the two metrics emphasize different aspects of software evolution. The original NPD primarily reflects structural growth, responding mostly to increases in nodes and edges. The proposed execution path formulation gives stronger responses to modifications that affect how the program actually executes. As the scenarios become more complex, the proposed NPD consistently produces higher divergence values, which suggests that it offers enhanced sensitivity to execution-oriented change. This behavior is valuable when developers are concerned with how program logic evolves across versions rather than with structural properties alone.

The controlled scenarios demonstrate that the two versions of divergence metrics respond differently depending on the type of modification introduced into the call graph. The original NPD increases primarily in cases where the graph expands structurally, reflecting changes in node and edge counts. The proposed execution path-based NPD, however, shows a more graded response: small increases for node only modifications, moderate increases when new execution chains are introduced, and strong divergence when internal control-flow structures are altered. These observations confirm that the proposed formulation is more sensitive to behavioral changes, whereas the original emphasizes structural growth. This distinction sets the stage for interpreting the results of the small-example study and motivates the need for both metrics when analyzing software evolution. While these synthetic scenarios provide controlled evidence of the proposed NPD's sensitivity, they do not fully represent changes that occur in real software project. Therefore, an additional real world case study is conducted in the next section.

4.3 Real World Case Study

To address the limitation of relying only on illustrative and synthetic datasets, this study adds a real world case study using Flask (https://github.com/pallets/flask), an open source python web framework from the pallets project. Flask is a lightweight web server gateway interface (WSGI) designed to support both simple and complex web applications. This project was selected because it is a mature and widely used python project with clear release snapshot and naturally evolving program structure. These characteristics make Flask suitable for examining whether the proposed execution path-based NPD can capture changes that occur in real software evolution, rather than only in controlled synthetic settings.

The case study focuses on two core modules of this project, namely *app.py* and *ctx.py*. The *app.py* module was selected because it contains the main Flask application class and includes mechanisms related to request dispatching, response handling, error handling, URL routing, and application lifecycle management. These responsibilities make it a suitable target for analyzing execution path changes because modifications in this module may affect both function interactions and control flow. The *ctx.py* module was also selected because it manages application and request contexts, which are central to how Flask handles runtime state during request processing. Together, these two modules provide a file level view of both application flow and context management in Flask. The same extraction and divergence calculation program used in the synthetic evaluation in Section 4.2 was also used in this real world case study to ensure consistency across experimental settings.

Three release snapshots were selected: v2.0.0, v2.1.0, and v2.2.0. These versions allow two consecutive comparisons and one cumulative comparison, there are v2.0.0 to v2.1.0, v2.1.0 to v2.2.0, and v2.0.0 to v2.2.0. This setup is sufficient to observe both short-term changes between releases and accumulated changes across a longer interval. For each selected module and version, a static call graph was extracted and converted into the graph representation used by both divergence metrics. The structural profile of the extracted graphs is summarized in Table 11.

Table 11 Call graph profile of selected Flask modules across release version.

Module	Version	Nodes	Edges
app.py	2.0.0	68	79
	2.1.0	67	74
	2.2.0	72	80
ctx.py	2.0.0	30	39
	2.1.0	33	42
	2.2.0	31	41

The node and edge counts show that both selected modules show only moderate changes in graph size across the selected releases, making them suitable for examining whether execution path-based divergence can reveal changes beyond simple node and edge growth.

After profiling the extracted call graphs, divergence was computed for both consecutive and cumulative version pairs using the original NPD and the proposed execution path-based NPD. The results are presented in Table 12. The difference between the original and proposed NPD can be explained by the type of information encoded in each portrait.

Table 12 Divergence comparison for selected Flask module version pairs.

Module	Version Pair	Original NPD	Proposed NPD
app.py	2.0.0 to 2.1.0	0.062	0.841
	2.0.0 to 2.2.0	0.136	0.870
	2.1.0 to 2.2.0	0.166	0.950
ctx.py	2.0.0 to 2.1.0	0.111	0.543
	2.0.0 to 2.2.0	0.193	0.163
	2.1.0 to 2.2.0	0.139	0.689

The original NPD summarizes graph structure through neighborhood and hop distance distributions. This representation is effective for capturing global topological differences, but it does not preserve the ordering of function calls. Therefore, changes that modify execution sequences may remain weakly reflected when the overall graph topology remains similar. In contrast, the proposed formulation represents software behavior through execution path length distributions. Since each execution path corresponds to an ordered sequence of function calls, changes in call ordering, branching, or restructuring can directly alter the resulting probability distribution. Consequently, the proposed method becomes more responsive to behavior related changes, even when the number of nodes and edges changes only a few.

The real world results shown in Table 12 show that the original NPD and the proposed execution path-based NPD respond differently to naturally occurring changes in Flask. In *app.py,* the original NPD remains low across all version pairs, ranging from 0.062 to 0.166, which is consistent with the relatively small changes in node and edge counts. However, the proposed NPD metric produces much higher divergence values, ranging from 0.841 to 0.950. This indicates that although the overall topology of the module changes only moderately, the distribution of execution paths changes substantially. The result supports the claim that execution path information can reveal behavioral changes that are not strongly reflected by topology based comparison used by original version of NPD.

The *ctx.py* results provide a more nuanced interpretation. In two version pairs, v2.0.0 to v2.1.0 and v2.1.0 to v2.2.0, the proposed metric produces higher divergence than the original NPD, suggesting that the changes affect execution path structure stronger than global topology. However, in the cumulative comparison between v2.0.0 and v2.2.0, the proposed divergence is lower than the original NPD. This suggests that some structural changes observed in the call graph do not necessarily lead to large changes in execution path distribution. Therefore, the proposed method should not be interpreted as simply producing larger values, but rather as emphasizing changes that alter execution flow.

These findings strengthen the analysis by showing that the proposed method is sensitive to execution related changes while still producing lower values when structural modification does not substantially affect path distribution. The high values observed in *app.py* also explain the sensitivity of the method: execution paths encode ordered call sequences, so changes in call ordering, branching, or central function interactions can shift the normalized path length distribution and produce large JSD values. At the same time, the lower value in one *ctx.py* pair illustrates that the metric does not automatically amplify every structural change. This provides stronger empirical support for the claim that execution path-based NPD offers a behavior oriented complement to the original topology based NPD.

5 Conclusion

This study examined the use of execution path information as an enhancement to the Network Portrait Divergence (NPD) metric for analyzing structural and behavioral changes in evolving software systems. The motivation for this work comes from the observation that the original, neighborhood-based NPD treats call graphs primarily as structural artefacts, while software evolution often involves modifications that alter not only the shape of the graph but also the semantics of program execution. To address this gap, we introduced an alternative formulation of NPD that replaces neighborhood distributions with execution path and path length distributions, allowing divergence to be measured in a way that reflects changes in how the program behaves across versions.

The evaluation was conducted through three settings: a small illustrative example, controlled synthetic scenarios, and a real world case study using selected modules from the Flask project. Across the illustrative and synthetic settings, the proposed metric consistently showed stronger sensitivity to changes that introduce new execution chains or restructure internal control flow. The real world case study further strengthened this observation. In the Flask *app.py* module, the original NPD remained low despite changes across releases, while the proposed metric produced substantially higher divergence values, indicating changes in execution path distribution that were not strongly captured by

topology alone. The *ctx.py* module provided a clearer result, showing that the proposed metric does not simply produce larger values in all cases, but responds more strongly when changes affect execution path structure. These results indicate that the proposed formulation provides a behavior-oriented complement to the original NPD by capturing aspects of software evolution that are less visible through structural topology alone.

Although the proposed enhancement offers clearer behavioral sensitivity, several limitations must be acknowledged. In some restructuring scenarios, the divergence values produced by the execution path formulation tend to reach the upper limit of the scale, which reduces the granularity of the metric and makes it difficult to distinguish between different degrees of behavioral change. This suggests that additional weighting schemes or alternative normalization strategies may be required to maintain discriminative power across a wider variety of modifications. In addition, the current implementation enumerates simple paths without applying large scale pruning strategies. This may limit scalability when the call graph contains many branching structures or when the graph size becomes very large. Future refinement may therefore consider maximum path length constraints, fan out pruning, bounded depth traversal, cycle summarization, k-shortest path approximation, or sampling-based path enumeration to improve scalability and better handle recursive or cyclic call structures. Furthermore, interpretation of divergence values, whether produced by the original or the proposed metric, still relies on expert judgement. Divergence should be understood as an indicator of difference rather than an assessment of code quality, and the significance of any value may vary depending on the expectations of developers and the context of the project. These considerations point to several opportunities for future refinement, including adaptive thresholds, improved modelling of execution paths, broader validation across larger software systems, and expert supported interpretation.

Another limitation concerns the use of static call graphs. Static call graph extraction may introduce over approximated or infeasible edges, which can affect the resulting execution path portraits. In this study, the same extraction pipeline is applied consistently across versions to preserve comparability, but robustness across different call graph builders or extraction configurations has not yet been evaluated. Future work should therefore include sensitivity analysis using multiple call graph construction tools, alternative extraction settings, and dynamic traces to assess the influence of static call graph imprecision.

Overall, the findings demonstrate that incorporating execution path information into divergence analysis provides a meaningful extension to existing structural metrics. While the original NPD remains effective for capturing structural growth, the proposed formulation offers a complementary behavioral perspective

that can help developers better understand how program logic evolves across versions. Future research will aim to refine the model to improve its granularity, evaluate its scalability on larger software systems, and integrate qualitative insights through expert evaluation to strengthen its interpretability in practical software evolution studies.

Acknowledgement

This research is funded by Riset PPMI STEI 2026 – Riset KK, School of Electrical Engineering and Informatics ITB.

References

[1] Krüger, J., *Tackling Knowledge Needs during Software Evolution,* in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1244-1246, 2019.

[2] Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E. & Li, S., *Measuring Program Comprehension: A Large-scale Field Study with Professionals*, in IEEE Transactions on Software Engineering, **44**(10), pp. 951-976, 2017.

[3] Walunj, V., Gharibi, G., Ho, D.H. & Lee, Y., *Graphevo: Characterizing and Understanding Software Evolution using Call Graphs*, in 2021 IEEE International Conference on Big Data (Big Data), 2019.

[4] Ardigo, S., Nagy, C., Minelli, R. & Lanza, M., *M3tricity: Visualizing Evolving Software and Data Cities,* in 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 130-133, 2022.

[5] Chen, L., Lanza, M. & Hayashi, S., *Understanding Code Change with Micro-changes,* in International Conference on Software Maintenance and Evolution (ICSME), pp. 363-374, 2024.

[6] Mi, Q., Zhan, Y., Weng, H., Bao, Q., Cui, L. & Ma, W., *A Graph-based Code Representation Method to Improve Code Readability Classification,* Empirical Software Engineering, **28**(4), 87, 2023.

[7] Alanazi, R., Gharibi, G. & Lee, Y., *Automatic Hierarchical Clustering of Static Call Graphs for Program Comprehension,* in 2021 IEEE International Conference on Big Data (Big Data), 2018.

[8] Aponte J. & Cardenas, G.O., *Evaluating the Graph-based Visualization Technique: A Controlled Experiment,* Enfoque UTE, **8** (Suppl. 1), pp. 201-216, 2017.

[9] Grove, D., DeFouw, G., Dean, J. & Chambers, C., *Call Graph Construction in Object-oriented Languages,* in ACM SIGP**L**AN, 1997.

[10] Zhou, F., Fan, Y., Lv, S. Jiang, L., Chen, Z., Yuan, J., Han, F., Jiang, H., Bai, G. & Zhao, Y., *Fctree: Visualization of Function Calls in Execution,* Information and Software Technology, **175**, 2024.

[11] Vasa, R., Schneider, J.-G. & Nierstrasz, O., The *Inevitable Stability of Software Change,* in International Conference on Software Maintenance, pp. 4-13, 2007.

[12] Gharibi, G., Tripathi, R. & Lee, Y., Code*2graph: Automatic Generation of Static Call Graphs for Python Source Code* in International Conference on Automated Software Engineering, pp. 880-883, 2018.

[13] Alanazi, R., Gharibi, G. & Lee, Y., *Facilitating Program Comprehension with Call Graph Multilevel Hierarchical Abstractions,* Journal of Systems and Software, **176**, 2021.

[14] Alnabhan, M., Hammouri, A., Hammad, M., Atoum, M. & Al-Thnebat, O., *2d Visualization for Object-oriented Software Systems,* in 2018 International Conference on Intelligent Systems and Computer Vision (ISCV), pp. 1-6, 2018.

[15] Borowski, K., Balis, B. & Orzechowski, T., *Graph Buddy-an Interactive Code Dependency Browsing and Visualization Tool,* in Working Conference on Software Visualization (VISSOFT), pp. 152-156, 2022.

[16] Jin, W., Xu, S., Chen, D., He, J., Zhong, D., Fan, M., Chen, H., Zhang, H. & Liu, T., *Pyanalyzer: an Effective & Practical Approach for Dependency Extraction from Python Code,* in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1-12, 2024.

[17] Li, C. Pei, Y. Shen, Y. Lu, J. Fan, Y. Linghu, X. Tian, Y. & Wang, K., "*Pyvisvue3d3: Python Visualization from Hierarchy Tree to Call Graph*, SoftwareX, **26**, 2024.

[18] Mortara, J., Collet, P. & Dery-Pinna, A.-M., *Visualization of Object-oriented Software in a City Metaphor: Comprehending the Implemented Variability and its Technical Debt,* Journal of Systems and Software, **208**, 111876, 2023.

[19] Salis, V., Sotiropoulos, T., Louridas, P., Spinellis, D. & Mitropoulos, D., *A Replication Package for Pycg: Practical Call Graph Generation in Python,* in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*,* 200, 2021.

[20] Hora, A., *Monitoring the Execution of 14k Tests: Methods Tend to Have One Path that is Significantly More Executed,* in Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024), pp. 532-536, 2024.

[21] Ding, Y., Steenhoek, B., Pei, K., Kaiser, G., Le, W. & Ray, B., *Traced: Execution-aware Pre-training for Source Code,* in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24) pp. 1-12, 2024.

[22] Yang, Y., Yin, H., Cao, J., Chen, T., Hung, N. Q.V., Zhou X. & Chen, L., *Time-aware Dynamic Graph Embedding for Asynchronous Structural Evolution,* in IEEE Transactions on Knowledge and Data Engineering, **35**(9), pp. 9656-9670, 2023.

[23] Liu, Y., Safavi, T., Dighe, A. & Koutra, D., *Graph Summarization Methods and Applications,* in ACM Computing Surveys, **51**(3), pp. 1-34, 2018.
[24] Zhou, H., Liu, S., Shen, H. & Cheng, X., *Graph Summarization for Preserving Spectral Characteristics. In Society for Industrial and Applied Mathematics Ebooks,* Proceedings of the 2024 SIAM International Conference on Data Mining (SDM), 2024.
[25] Zhou, H., Liu, S., Shen, H. & Cheng, X., *Node Embedding Preserving Graph Summarization,* ACM Transactions on Knowledge Discovery from Data, **18**(6), pp. 1-19, 2024.
[26] Tsalouchidou, I., Bonchi, F., Morales, G.D.F. & Baeza-Yates, R., *Scalable Dynamic Graph Summarization,* in IEEE Transactions on Knowledge and Data Engineering, **32**(2), pp. 360-373, 2018.
[27] Brunner, T. & Porkoláb, Z., *Advanced Code Comprehension using Version Control Information,* IPSI Transactions on Internet Research, pp. 47-54, 2020.
[28] Shaikh, V., *Decision Support System for Pull Requests Review using Path-based Network Portrait Divergence and Visualization,* University of Missouri - Kansas City ProQuest Dissertations and Theses, 2022.
[29] Bagrow, J.P. & Bollt, E.M., *An Information-theoretic, All-scales Approach to Comparing Networks,* Applied Network Science, **4**, 45, 2019.
[30] Utture, A., Liu, S., Kalhauge, C.G. & Palsberg, J., *Striking a Balance: Pruning False-positives From Static Call Graphs,* in Proceedings of the 44th International Conference on Software Engineering, pp. 2043-2055, 2022.
[31] Al-Sharif, Z. & Jeffery, C., *Abstracttrace: The use of Execution Traces to Cluster, Classify, Prioritize, and Optimize a Bloated Test Suite,* Applied Sciences, **14**(23), 11168, 2024.
[32] Krause-Glau, A., Damerau, L., Hansen, M. & Hasselbring, W., *Visual Integration of Static and Dynamic Software Analysis in Code Reviews Via Software City Visualization,* Visualizing Software for Understanding and Analysis (VISSOFT), Flagstaff, AZ, USA, pp.144-149, 2024.
[33] D'ambros, M., Gall, H., Lanza, M. & Pinzger, M., *Analysing Software Repositories to Understand Software Evolution,* in Software Evolution, Springer, pp. 37-67, 2008.
[34] Rajlich, V., *Software Evolution and Maintenance,* in FOSE 2014: Future of Software Engineering Proceedings, pp. 133-144, 2014.
[35] Bani-Salameh, H., Ahmad, A. & Aljammal, A.H., *Software Evolution Visualization Techniques and Methods – a Systematic Review,* 2016 7th International Conference on Computer Science and Information Technology (CSIT), doi: 10.1109/CSIT.2016.7549475.

[36] Rufiange, S. & Melanc¸on, G., *Animatrix: A Matrix-based Visualization of Software Evolution,* in Proceedings of the 2nd IEEE Working Conference on Software Visualization, pp. 137-146, 2014.

[37] Chaikalis, T., Melas, G. & Chatzigeorgiou, A., *Seanets: Software Evolution Analysis with Networks,* in IEEE International Conference on Software Maintenance (ICSM), **286**, pp. 634-637, 2012.

[38] Alanazi, R.*, Advancements in Call Graph Methodologies for Enhanced Program Comprehension: A Review,* in IJCSNS International Journal of Computer Science and Network Security, **25**(11), pp.55-62, 2025.

[39] Keshani, M., Gousios, G. & Proksch, S., *Frankenstein: Fast and Lightweight Call Graph Generation for Software Builds,* Empirical Software Engineering, **29**(1), 2023. doi: 10.1007/s10664-023-10388-7.

[40] Tunalı, V. & Tüysüz, M.A.A., *Analysis of Function-call Graphs of Open-source Software Systems using Complex Network Analysis,* Pamukkale University Journal of Engineering Sciences, **26**(2), pp. 352-358, 2020.

[41] Sözer, H., *Call Graph Delta Analysis and Security Vulnerability Assessment with Static Analysis,* in 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 2412-2417, 2024.

[42] Domenico, M.D. & Biamonte, J., *Spectral Entropies as Information-Theoretic Tools for Complex Network Comparison,* Physical Review X, **6**(4), 041062, 2016.

[43] Chen, D., Shi, D.D., Qin, M., Xu, S.M. & Pan, G.J., *Complex Network Comparison Based on Communicability Sequence Entropy,* Physical Review. E, **98**(1), 012319, 2018.

[44] Lafhel, M., Cherifi, H., Renoust, B., Hassouni, M.E. & Mourchid, Y., *Movie Script Similarity using Multilayer Network Portrait Divergence,"* in Studies in computational intelligence, pp. 284-295, 2020.

[45] Hoyos-Osorio J.K. & Sanchez-Giraldo, L.G., *The Representation Jensen-Shannon Divergence,* 37th Conference on Neural Information Processing Systems, 2023.

[46] Endres, D. & Schindelin, J., *A New Metric for Probability Distributions,"* in IEEE Transactions on Information Theory, **49**(7), pp. 1858-1860, 2003.

[47] Bhattacharjee, A., *Hcpc: Human Centric Program Comprehension by Grouping Static Execution Scenarios,* TBD, 2021.

[48] Ilg, L. & Cito, J., *Codedetective: Enabling Isolated Code Execution,* Diploma Thesis in Software Engineering and Internet Computing, TU Wien, 2025.

[49] Stapleton, S., Gambhir, Y., LeClair, A., Eberhart, Z., Weimer, W. & Leach, K.*, A Human Study of Comprehension and Code Summarization,* in 2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC), pp. 2-13, 2020.