

Article

ArchSense: Characterizing Component-Internal Implementation-Semantic Evolution via Code Representation

Tong Wang *, Guowei Jing and Hao Zhou 

School of Computer Science and Technology, Anhui University of Technology, Maanshan 243000, China

* Correspondence: tongwang_se@ahut.edu.cn

Abstract

Monitoring software architecture evolution is essential for maintaining system quality, but many practical change-detection pipelines still rely on syntactic differences. Such pipelines can produce large review streams in which minor edits, refactorings, and deeper implementation rewrites are mixed together. In this paper, we introduce ArchSense, a diagnostic framework for characterizing component-internal implementation-semantic evolution. ArchSense combines hierarchical structural alignment with method-level semantic displacement measurement. Instead of treating all code updates as equal, it uses vector representations from GraphCodeBERT to measure implementation-level semantic change intensity and then summarizes this signal within architectural components. We evaluate ArchSense on six large-scale open-source Java systems. The results show that semantic distance is positively associated with a three-annotator construct assessment of implementation change intensity. Additional disagreement analysis links high and low model decisions to concrete source code features, including control flow, call-set, return, exception, synchronization, and test-observable changes. A behavioral proxy check on 23,389 test method updates shows that the default threshold behaves as a conservative high-precision, low-recall review signal for implementation-semantic change. The validated scope of ArchSense is therefore component-internal diagnostic characterization for architecture-oriented review; connector topology, runtime communication, behavioral equivalence, and defect causality require complementary evidence.

Keywords: software architecture; implementation evolution; semantic code analysis; change characterization



Academic Editors: Flavio Canavero and Ioannis Hatzilygeroudis

Received: 14 April 2026

Revised: 17 June 2026

Accepted: 22 June 2026

Published: 1 July 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Understanding software architecture is essential for maintaining the quality of complex systems [1,2]. Contemporary research focuses on various aspects of evolution analysis, such as architectural view recovery, violation detection, and technical debt tracking [3,4]. Moreover, researchers also study how code changes affect architecture [5,6]. Repository mining and syntactic detection techniques are commonly used to identify modified parts [7,8]. These methods help researchers to understand system evolution, but they rely primarily on syntactic analyses of source code.

Software architecture consists of components, connectors, and configuration. Traditional evolution analysis often focuses on topological changes in connection relationships between components. Such topology is not determined only by internal implementation, because requirements, deployment constraints, and project decisions can also reshape architectural connections. The functional responsibilities and data-flow patterns inside a compo-

nent nevertheless form one important layer of architectural maintenance risk. A dramatic shift in implementation logic, such as a large-scale framework migration or a core algorithm rewrite, can change the reasoning burden and integration risk of a component, even when its public interface remains stable. Thus, measuring the intensity of internal semantic change provides a retrospective diagnostic which is observable for component-internal implementation evolution. It should not be interpreted as a complete model of architecture evolution, because connector roles, runtime communication, deployment configuration, and cross-component data-flow topologies are separate architectural dimensions.

The reliance on syntactic differences creates substantial challenges in practical software development [9,10]. Developers frequently perform small edits, refactorings, API cleanups, and localized bug fixes together with deeper implementation rewrites. These activities generate a high volume of syntactic churn. A syntactic change stream alone cannot be used to determine whether a method update is a minor local edit or a major implementation restructuring. This results in a problem where high-intensity implementation signals are hidden among lower-priority updates [11,12]. Moreover, existing techniques mostly give a binary view of changes. They report whether a component has changed, but fail to quantify the magnitude of the modification. A minor cleanup and a full algorithm rewrite may therefore be treated as equivalent change events.

These limitations raise the question of whether current detection methods can help architects prioritize the components that require attention. Furthermore, existing metrics such as number of lines of code or file churn have important limitations. They correlate with maintenance effort but do not reflect the semantic significance of evolution. Without this distinction, architects may focus on trivial code cleanups and ignore critical logical changes. This leads to inefficient resource allocation and increases the risk of architectural erosion.

This article addresses this research gap by characterizing semantic evolution inside architectural components. We introduce ArchSense, a framework that focuses on measuring semantic displacement rather than counting syntactic edits. Our approach evaluates how method logic is modified, how vectors shift in the representation space, and how components summarize these changes. The framework complements topology-based architecture analysis, refactoring detectors, and runtime behavioral validation with a component-internal semantic characterization layer. Specifically, ArchSense measures implementation-level semantic change intensity and treats it as an empirical diagnostic signal for component-level review prioritization. We treat semantic shifts at the method level as implementation-level observables and report their component-level patterns alongside manual construct assessment, source–feature disagreement analysis, behavioral proxy checks on test methods, and developer-documented historical records. By aggregating these signals, ArchSense summarizes the implementation activity occurring inside architectural component containers without claiming to fully characterize connectors, runtime communication, or behavior preservation.

The article’s contribution is as follows:

- (1) We propose a framework called ArchSense which measures component-internal implementation-semantic change intensity and examines how this signal can support component-level review triage. By incorporating structural analysis and neural embeddings, we offer a method to summarize how many implementation-semantically intensive changes have occurred inside architectural component containers.

- (2) We introduce a dual-metric matrix to categorize component-internal implementation evolution into distinct categories. This approach provides architects with quantitative evidence to differentiate broad low-intensity maintenance from concentrated high-intensity implementation restructuring.

(3) We provide empirical evidence from six large-scale systems, including manual construct assessment, source–feature disagreement analysis, test-observable behavioral proxies, and adaptive threshold calibration. The results support the use of semantic displacement as a review triage signal, while also showing that behavior-preserving refactoring and runtime equivalence require separate validation.

The rest of this paper is organized as follows: Section 2 presents the technical details of the ArchSense framework. Section 3 provides the experimental evaluation and discusses the findings from our large-scale study. Section 4 discusses the related work. Section 5 discusses the practical integration, inherent limitations, and validity threats. Finally, Section 6 concludes the paper and highlights several directions for future work.

2. Methodology

In this section, we present the technical details of the ArchSense framework. The main goal of the framework is to measure implementation-level semantic change intensity inside architectural components. We treat our metric as a computed implementation observable rather than as a direct measure of complete architectural evolution.

The measurement starts from source-code displacement rather than from predefined large-scale restructuring events. ArchSense computes semantic vector displacement from source code first and then studies whether this implementation-level signal is empirically related to changes that maintainers later describe as large refactoring, redesign, or system-level restructuring.

The rationale for this relation is based on maintenance reasoning cost. In large systems, developer-perceived major restructuring is usually not triggered by every edited line. It is more likely when updates change the local conditions that code reviewers, maintainers, and tests must reason about, including control flow, data dependencies, method interactions, return behavior, exception handling, synchronization, and core computation. These changes can increase review burden and can also require updates in surrounding tests or callers. Semantic displacement is therefore treated as a measurable proxy for implementation-restructuring pressure, not as a replacement for architectural topology or developer judgment.

Operationally, a source code modification may change the data flow and control logic within a method. GraphCodeBERT maps the old and new method versions into a semantic vector space, and the displacement between the two normalized vectors is used as the metric. This value is computed from source code and the embedding model, and it does not use architectural labels or developer documentation. Developer-documented records are not used to define the metric, train the encoder, calibrate the method-level distance values, assign the manual labels, or classify components. Whether or not this displacement co-varies with significant engineering activity is an empirical question. We treat it as a hypothesis and examine it in Section 3 using manual construct assessment, proxy-audits, source-level indicators, test-observable proxies, and selected developer-documented historical records. These sources are not behavioral oracles, and they do not establish runtime equivalence or defect causality. The developer records are used only as historical context for the implementation-level construct measured by the framework.

We use a coarse-to-fine analysis approach. The workflow has three main phases: structural abstraction and alignment, semantic intensity measurement, and component-level evolution characterization. These phases define a retrospective analysis pipeline for comparing two observed versions; they should not be interpreted as a causal model of how structural and semantic changes influence each other across future releases.

2.1. Phase 1: Structural Abstraction and Alignment

The first phase of our framework sets up a structural foundation for change detection [13]. To locate exactly where changes happen in the architecture hierarchy, we propose the Multi-Layered Architecture Tree (MLAT). This phase takes two versions of a software system as input and produces a set of candidate update pairs for semantic analysis. We define the software architecture not just as a flat collection of files but as a hierarchical structure that shows the system modular organization. We build the MLAT by parsing the source code and directory structure of the system. These four levels of abstraction strictly constitute the hierarchical layers of the MLAT model:

- Component layer: The root and high level subsystems representing the large functional units.
- Module layer: The intermediate containers such as Java packages that group related files.
- File layer: The source files acting as the physical units of implementation.
- Method Layer: The leaf nodes of the tree representing the basic functional units within files.

Each node n in the MLAT is described by a tuple $\langle L, T, \Sigma \rangle$, where L is its level, T is its type, and Σ is its signature. To make comparisons faster, we also calculate a structural hash for each node. This hash is calculated recursively from the node's own content and the hashes of its children. This ensures that any change in a sub-tree reaches the root. We created two trees, T_{old} and T_{new} , to represent the architectures before and after evolution.

For tracking architectural elements across versions, we use a hybrid alignment method that combines exact matching with similarity-based matching. This strategy allows us to map the nodes from T_{old} to T_{new} with high precision. We first perform a top-down exact pruning. We go through both trees, starting from the root. For any pair of nodes with the same signatures and hashes, we assume that the whole sub-tree below these nodes has remained unchanged. We mark these nodes as matched and remove their children from the search. This optimization significantly reduces the work for the next steps.

For nodes that are not pruned, we move to bottom-up similarity matching. In this step, we try to find the best match for each unmatched node. We define the similarity between node $u \in T_{old}$ and node $v \in T_{new}$ based on how many elements they share. Specifically, we calculate the Dice coefficient of their child sets:

$$Sim(u, v) = \frac{2 \times |Children(u) \cap Children(v)|}{|Children(u)| + |Children(v)|} \quad (1)$$

If the similarity score, $Sim(u, v)$, is higher than a set structural threshold, γ , we create a mapping link between u and v . Algorithm 1 describes this alignment procedure formally.

Based on the alignment map, we group structural evolution into three different types. Addition happens when a node in T_{new} has no match in T_{old} , and Deletion happens when a node in T_{old} has no match in T_{new} . These two cases represent clear changes to the architectural structure. The Update is identified when a node in T_{old} is matched to a node in T_{new} , but their content hashes are different. This shows that, while the architectural part remains, its internal implementation has changed.

Algorithm 1 Tree alignment algorithm**Require:** Base Tree T_{base} , Target Tree T_{target} , Structural Threshold γ **Ensure:** Set of Mapped Pairs M

```

1:  $M \leftarrow \emptyset$  {Stage 1: Exact Filter via Hash}
2: for all node  $u$  in  $T_{base}$ , node  $v$  in  $T_{target}$  do
3:   if  $Hash(u) == Hash(v)$  then
4:     Add  $(u, v)$  to  $M$ 
5:     Skip children of  $u$  and  $v$ 
6:   end if
7: end for {Stage 2: Similarity Scan}
8: for all unmapped node  $u$  in  $T_{base}$  do
9:    $BestCandidate \leftarrow Null$ 
10:   $HighestScore \leftarrow 0$ 
11:  for all unmapped node  $v$  in  $T_{target}$  do
12:     $CurrentScore \leftarrow Sim(u, v)$ 
13:    if  $CurrentScore > HighestScore$  then
14:       $HighestScore \leftarrow CurrentScore$ 
15:       $BestCandidate \leftarrow v$ 
16:    end if
17:  end for
18:  if  $HighestScore > \gamma$  then
19:    Add  $(u, BestCandidate)$  to  $M$ 
20:  end if
21: end for
22: return  $M$ 

```

2.2. Phase 2: Semantic Intensity Measurement

The objective of this phase is to quantify the intensity of each method update identified in Phase 1. Syntactic metrics like lines of code are poor indicators of component-level implementation impact. Therefore, we introduce a measurement framework based on the displacement of code in a semantic representation space. This displacement serves as a proxy for implementation evolution. The process involves three key steps: code representation, semantic distance calculation, and classification.

However, to understand why such a semantic measurement is necessary, we revisit the limitations of traditional architectural analysis. Traditional architectural analysis often focuses on external dependencies, such as changes to public interface signatures; while interface changes have immediate ripple effects across a given system, they only represent one dimension of architectural evolution. Modern component based systems also experience component-internal implementation evolution. A developer might completely rewrite a core algorithm while keeping the public interface unchanged. Traditional dependency analysis tools miss this rewrite because the external contract is unbroken. However the internal implementation complexity and the subsequent maintenance risk of that component can still shift substantially.

ArchSense is designed to capture this specific dimension of component-internal implementation-semantic evolution. We define a substantive evolution not by external behavioral equivalence but by the shift in implementation semantics and reasoning complexity. We measure the intensity of implementation level evolution to identify deep internal restructuring. This metric serves as a retrospective diagnostic signal for localized implementation drift. It complements traditional dependency-based architectural metrics rather than replacing them.

To capture the implementation logic of a method, we use a code-representation function, Φ , that maps a source code string, c , to a high-dimensional vector, V . The variable d denotes the dimensionality of the semantic vector space.

$$V = \Phi(c), \quad V \in \mathbb{R}^d$$

The function Φ is implemented using a pretrained neural model, GraphCodeBERT. We choose GraphCodeBERT over alternative models like CodeBERT or CodeT5 due to its specific inductive bias; while CodeT5 excels at generative tasks and CodeBERT relies heavily on natural language sequences, GraphCodeBERT is explicitly pretrained on data flow graphs. In software architecture, the internal data flow defines the operational pathways of a component. A significant modification to the data flow indicates a fundamental shift in how data is processed. This matches our objective to measure implementation evolution. The resulting vector displacement is an operational semantic proxy rather than a formal proof of behavioral equivalence.

This capability allows the model to separate substantive implementation-level changes from trivial programming noise. Superficial changes such as code styling, identifier renaming, or standardized API usage patterns result in negligible semantic displacement. Conversely, a pervasive restructuring of control and data flow, such as migrating a component to a new technology stack, results in a significant Euclidean distance; while such a migration might be viewed as a standardized process, it can alter the component-internal state semantics, reasoning complexity, and integration testing requirements. ArchSense is designed to quantify the intensity of this type of internal disruption. By focusing on implementation logic, our framework captures one class of evolutionary signals that may be associated with topological or restructuring changes, while acknowledging that topological changes can equally originate from external requirement shifts.

We perform the analysis at the method level. We chose this granularity to strike a balance between context richness and signal accuracy. Granularities coarser than a file or class are too broad. A single file often contains multiple different functionalities. Generating one vector for a large file can weaken the semantic signal of specific changes. Finer granularities, such as the statement level, lack the required context. Neural models need a complete logic flow in order to generate useful representations.

Methods are practical units for representing localized implementation logic and interactions. They are well adapted to the input length limit of pretrained models, which reduces the risk of cutting off complete logic fragments. Semantic shift at this level provides a fine-grained signal for implementation drift analysis. By limiting changes to particular methods, we keep stable code sections from masking the signal of evolutionary change. We perform a code-sanitization step before we apply the representation function. This step removes non-executable artifacts such as comments, annotations, and redundant whitespace.

After obtaining the vector representations V_{old} and V_{new} , we measure the displacement between them. The selection of the distance metric is a deliberate design choice to ensure metric stability. Pretrained code models often suffer from the anisotropy phenomenon. This property causes vectors to cluster in a narrow cone within the high-dimensional space. Under this distribution, angular metrics like cosine similarity become saturated and yield scores very close to 1.0. This prevents the framework from distinguishing the magnitude of logic modifications.

To address this issue, we normalize each vector to unit length. This normalization ensures that the magnitude of the vector is not affected by code length.

We then calculate the Euclidean distance between these normalized vectors. We select Euclidean distance for two specific engineering reasons: First, the nonlinear relationship between the two metrics expands the scale for vectors that are very close together. Due to the square root function, small angular differences near zero result in larger relative differences in Euclidean distance. This sensitivity helps to differentiate subtle shifts in the anisotropic space where cosine scores are compressed. Second, the Euclidean distance is a

proper metric that satisfies the triangle inequality. This property is essential for Phase 3, where we aggregate individual distances to calculate the average semantic displacement. The cumulative sum of Euclidean distances represents a physically meaningful path length in the vector space.

We define the semantic distance, d , for an update pair, as follows:

$$d(V_{old}, V_{new}) = \sqrt{\sum_{i=1}^d (V_{old}^{(i)} - V_{new}^{(i)})^2} \quad (2)$$

where d is the dimensionality of the vector space. This distance captures the absolute magnitude of implementation changes. It allows us to identify intensity differences between minor fixes and substantive redesigns.

According to the calculated distance, we divide method updates into two categories: trivial updates and substantive evolutions. A substantive evolution is defined as a modification that induces a significant displacement in the vector representation. For example, changing from a loop to a parallel stream is considered a substantive evolution even if it preserves the same input–output behavior. From a component-review perspective, such implementation changes can affect maintainability and reasoning complexity.

Conversely, a trivial update gives rise to negligible vector displacement, such as identifier renaming or formatting. These changes are treated as lower-priority signals from a component-review perspective. We use a distance threshold, δ , for this classification. An update is categorized as a substantive evolution when the distance is larger than or equal to δ ; otherwise, it is treated as a trivial update. The set of substantive evolutions is denoted by S , which is passed to the next phase for component-level characterization.

All embedding vectors are normalized to unit length before distance computation. Although Euclidean distance on normalized vectors is mathematically monotonic with cosine similarity, we adopt this formulation for its numerical stability and its suitability for threshold-based decision making. Euclidean distance provides a more interpretable distance scale for distinguishing negligible semantic drift from substantive semantic change.

2.3. Phase 3: Component Level Evolution Characterization

The final phase summarizes method-level measurements for each architectural component. This aggregation is a diagnostic summarization rather than a claim that architectural evolution is fully reducible to method-level changes. Components are primary structural containers in the architecture, while specific implementation modifications occur within methods. The cumulative semantic displacement within a component can therefore indicate a shift in local implementation responsibility or maintenance risk. It does not by itself model inter-component connectors, runtime communication, or data-flow topology across components. This multidimensional analysis allows ArchSense to categorize components into distinct implementation-semantic evolution patterns that should be interpreted together with topology-based architectural evidence.

A component represents a structurally cohesive unit of the architecture such as a package or a high-level module. Let C denote an architectural component. Let U_C represent the set of all updated methods within component C , identified in the structural analysis phase. Let T be the total change frequency which corresponds to the number of methods in set U_C . If T is zero, then the component is classified as stable.

For components with a non zero T , we characterize their evolution. A simple count of changes is insufficient because it conflates the scope of evolution with its intensity. A component with many small bug fixes is fundamentally different from a component

with a few major feature rewrites. Therefore, we introduce a dual-metric framework to independently quantify these two dimensions.

The first indicator is evolutionary breadth, which we denote as EB. EB captures the evolutionary density of a component by quantifying the proportion of updates that introduce substantive semantic change. A high EB indicates that most modifications in the component contribute to actual logical changes. A low EB suggests that the development activity is dominated by trivial refactorings. We define $EB(C)$ as follows:

$$EB(C) = \frac{|\{m \in U_C \mid d(m) \geq \delta\}|}{T} \quad (3)$$

The variable m represents an updated method pair in set U_C . Every pair m contains the method version before and after evolution. The function $d(m)$ calculates the semantic distance between these two versions. The symbol δ represents the semantic threshold. The numerator counts the number of method pairs where the semantic distance is greater than or equal to δ . Therefore, $EB(C)$ represents the ratio of substantive implementation evolutions to the total number of updates in component C . The value of EB ranges from 0 to 1.

The second indicator is average semantic displacement (ASD). This metric measures the overall intensity of the modifications by using the semantic distance as a weight. It serves as a proxy for the magnitude of the evolutionary effort. EB focuses only on substantive changes. In contrast, ASD considers the distance of all updates, including trivial ones. A high ASD indicates that the average modification is structurally complex or logically significant. A low ASD suggests that the changes are minor. We define $ASD(C)$ as follows:

$$ASD(C) = \frac{1}{T} \sum_{m \in U_C} d(m) \quad (4)$$

The diagnostic value of the dual-metric matrix arises from the orthogonality of its two dimensions. EB captures the scope of semantic change, which is how broadly a component's implementation is shifting across its methods. ASD captures the depth of semantic change, which is how intensely each modified method is being restructured. These dimensions are hypothesized to correspond to different review risk profiles. High EB is hypothesized to relate to integration and coupling risk from widespread method-level changes. High ASD is hypothesized to relate to regression risk from concentrated and intense internal restructuring. We treat these correspondences as hypotheses subject to empirical examination rather than as theoretical assertions. Their joint distribution defines four empirically distinguishable implementation-semantic behavior patterns. We examine these patterns with PLCR measurements, source-level indicators, test-observable proxies, and selected historical records in Section 3.3. The composite signal produced by this matrix is diagnostically valuable because neither dimension alone can distinguish the risk profiles that the combination reveals.

By combining EB and ASD, we construct an evolutionary matrix to classify the dominant component-internal implementation pattern. This matrix provides a multidimensional diagnosis that is more informative than any single metric. We define four distinct evolution categories based on a breadth threshold, α , and an intensity threshold, τ .

Category 1 is aggressive evolution. This category is defined by the condition where EB is greater than or equal to α and ASD is greater than or equal to τ . Within the scope of implementation-semantic behavior analysis, this state represents a pattern of widespread and intense semantic change within the component's implementation. The simultaneous high breadth and intensity indicate substantial internal restructuring. It is distinct from simple code modification. This pattern can signal elevated component-review risk, espe-

cially when it also affects public APIs or dependency paths. Therefore, these components require stricter review to ensure that the new implementation aligns with system goals.

Category 2 is active expansion. This category is defined by the condition where EB is greater than or equal to α and ASD is less than τ . Within the scope of implementation-semantic behavior analysis, this state reflects a pattern of widespread but shallow semantic change across the component's implementation, indicating growth in method coverage without deep internal restructuring. The changes are widespread but shallow. In software engineering terms this often points to modifications in cross cutting concerns. Examples include logging updates or parameter propagation. The measured semantic intensity remains low despite the high volume of touched methods. This category indicates broad but low-intensity implementation activity. The review priority is often related to integration testing rather than deep logic inspection.

Category 3 is hidden hotspots. This category is defined by the condition where EB is less than α and ASD is greater than or equal to τ . Within the scope of implementation-semantic behavior analysis, this state signifies a pattern of highly concentrated and intense semantic change confined to a small number of methods. The changes are confined to a small number of methods. However the displacement within those methods is high. This pattern suggests that review attention should focus on a small set of methods inside the component. Traditional frequency-based metrics can miss this review target because the change volume is low.

Category 4 is defensive maintenance. This category is defined by the condition where EB is less than α and ASD is less than τ . Within the scope of implementation-semantic behavior analysis, this state represents a pattern of low-scope and low-intensity semantic change, indicating limited substantive restructuring under the current threshold. Modifications in this category are typically preventative or corrective. Common examples are minor bug fixes or style standardization. From an implementation-review perspective, these components have lower semantic intensity under the current threshold. Identifying this category allows review teams to assign proportionally lighter implementation review effort, while still relying on tests and dependency analysis for behavioral and architectural safety.

2.4. Threshold Determination Criteria

The performance of ArchSense depends on four primary parameters: γ , δ , α , and τ . We treat these values as explicit operating choices rather than hidden constants. The structural threshold γ is set by a structural matching heuristic, while δ , α , and τ are examined through threshold sweeps, manual annotation, and project-level distributions. This data-driven approach makes the framework adaptable to different development styles and project scales.

The structural threshold, γ , controls the precision of node alignment in Phase 1. We set γ to 0.7 based on standard heuristics for structural matching. This value ensures that only node pairs with substantial structural overlap are mapped.

The semantic threshold, δ , defines the boundary for substantive implementation evolution at the method level. This study treats $\delta = 4.0$ as an explicit conservative operating point rather than as a universal semantic boundary. The threshold is examined through calibration-oriented threshold sweeps, the three-annotator manual annotation study reported in RQ1, and the project-specific calibration procedure in Algorithm 2. These analyses show that lower thresholds improve recall, while $\delta = 4.0$ prioritizes high-precision review signals. The sampling criteria, annotation fields, threshold-sweep results, and calibration rules are reported so that the threshold choice is explicit rather than hidden.

Algorithm 2 Project-specific threshold calibration

Require: Method-distance set D_P for project P ; optional audited labels Y_P ; review-budget quantile q ; target precision π

Ensure: Project-specific operating threshold δ_P

- 1: Compute distributional candidates, including $Q_q(D_P)$ and $Q_3(D_P) + 1.5 \cdot IQR(D_P)$.
- 2: **if** audited labels Y_P are unavailable **then**
- 3: Set $\delta_P = Q_q(D_P)$ for budgeted screening, or use $Q_3 + 1.5 \cdot IQR$ for conservative outlier screening.
- 4: **else**
- 5: Sweep candidate thresholds over the observed distance range.
- 6: For each candidate, compute precision, recall, and F1 against Y_P .
- 7: Select the candidate with maximum F1, or the candidate with maximum recall subject to precision $\geq \pi$.
- 8: **end if**
- 9: **return** δ_P

The thresholds α and τ define the evolutionary matrix at the component level. We determine these values using the median of the distribution for each target system. α is set to the median value of Evolutionary Breadth across all components. τ is set to the median value of average semantic displacement across all components. This approach provides a relative baseline for each project. It allows architects to identify components that exhibit evolutionary activity higher than the system average.

By using calibration-oriented sweeps, the manual annotation study, and distribution medians to examine thresholds, ArchSense reduces the bias of manual parameter tuning. This systematic process makes the classification results more reproducible and comparable across different software versions.

3. Experimental Evaluation

The objective of our experimental evaluation is to assess the effectiveness of ArchSense in measuring implementation-level semantic change intensity and in characterizing component-internal evolution patterns. We do not use developer issue records to define semantic displacement, calibrate thresholds, or assign component categories. The evaluation first examines manual construct assessment, source-level technical indicators, test-observable proxies, baseline comparisons, and threshold robustness. Developer issue records are then used only as qualitative historical context for selected observations. We designed four research questions that address the accuracy of the semantic measurement, the practical impact of the filtering mechanism, the analytical power of component-level characterization, and the robustness of threshold parameters.

RQ1: Alignment with manual construct assessment. How much does the semantic displacement measured by ArchSense align with manual assessment of implementation evolution at the method level?

RQ2: Impact of semantic characterization on change signal quality. How effectively does our semantic intensity measurement improve the quality of implementation-level change signals by filtering trivial updates from substantive semantic changes?

RQ3: Characterization of component-internal implementation evolution categories. How does the dual-metric matrix reveal implementation-semantic behavior patterns across different components? What technical and historical evidence helps contextualize these patterns?

RQ4: Threshold sensitivity and robustness. To what extent are the identified evolutionary categories sensitive to the selection of classification thresholds?

These research questions are designed to form a logical sequence of empirical assessment. RQ1 checks the construct validity of method-level semantic displacement as

an implementation-level measurement. RQ2 evaluates whether semantic filtering improves the quality of component-level change signals and whether this filtering introduces false-negative risk. RQ3 examines whether the dual-metric matrix reveals meaningful differences among components and then contextualizes the results using source-level indicators, test-observable proxies, API exposure measures, and selected historical records. RQ4 tests whether the conclusions remain stable under broader threshold perturbations and alternative embedding models.

We chose six large-scale open-source projects to check the effectiveness of ArchSense as shown in Table 1. Their application scenarios go from high throughput message queuing and real-time stream processing to distributed storage and resource scheduling. They are known for massive code bases, long-term histories, and strict industrial level maintenance standards. This choice allows us to check ArchSense across different architectural roles, which range from stable infrastructure cores to active functional engines. This ensures that our findings can be used across different kinds of software. Frequent changes in these systems give a various set of implementation changes and refactoring activities.

Table 1. Experimental subjects and their versions.

Project	Old Version	New Version	Scale KLOC
Flink	1.16	1.19	2095
Kafka	2.1	3.0	297
Hadoop-Common	2.7	2.10	261
Hadoop-HDFS	2.7	2.10	355
Hadoop-MapReduce	2.7	2.10	225
Hadoop-YARN	2.7	2.10	309

3.1. RQ1: Alignment with Manual Construct Assessment

The first research question evaluates the construct validity of the semantic displacement metric as a measure of implementation-level change intensity. We report a manual annotation study based on 300 stratified method pairs sampled from the method-level change records extracted from the studied project versions. Three annotators independently labeled the same samples using a common written guideline. Each sample contains the old method body, the new method body, the component, and the method signature. The annotation records a broad semantic-impact label, a five-level manual intensity label, a confidence value, a change-scope category, and a short rationale. The annotators did not see the GraphCodeBERT distance or the final threshold decision during labeling. The completed annotation records are available from the corresponding author upon reasonable request.

The annotation guideline distinguishes broad semantic impact from high-intensity substantive evolution. A method is labeled semantic if it changes control flow, data dependency, call interaction, returned behavior, exception behavior, synchronization, or core computation. A method is labeled non-semantic if the change is limited to formatting, comments, naming, equivalent local cleanup, or mechanical edits that preserve the reasoning structure of the method. The manual intensity label uses H1–H5, where H1–H3 represent non-semantic, minor, or moderate updates, and H4–H5 represent substantive changes that should be prioritized for component-level implementation review. These labels are produced by human annotators from source code pairs, not from the model distance, so they do not simply restate the model output. The manual study is not a behavioral oracle: it does not establish runtime input–output equivalence, performance effects, or later defect causality. It is used as construct-level evidence for implementation-semantic change intensity.

Before adjudication, the three annotators show strong agreement. Fleiss' κ is 0.979 for the three-way semantic-impact label and 0.847 for the unweighted five-level intensity

label. The mean pairwise quadratic-weighted Cohen's κ for H1–H5 is 0.973. Full three-annotator agreement is 98.33% for the semantic-impact label and 82.00% for the five-level intensity label, while 99.67% of samples fall within one neighboring intensity level across all three annotators. Table 2 reports the final adjudicated distribution of the five manual intensity labels.

Table 2. Three-annotator manual annotation distribution after adjudication.

Annotation View	H1	H2	H3	H4	H5
Samples	78	37	64	48	73

Across the 300 annotated samples, the final semantic-impact labels contain 185 semantic, 78 non-semantic, and 37 unclear updates. The Spearman correlation between semantic distance and the final H1–H5 manual intensity label is 0.705, indicating a moderate-to-strong positive association. This result supports the use of semantic displacement as an ordering signal, but it also shows that model distance should not be interpreted as an exact substitute for manual semantic labels.

We then evaluate the method-level threshold, δ , against the final manual labels by treating H4 and H5 as substantive evolution. Table 3 reports the operating-point behavior. The default threshold $\delta = 4.0$ achieves high precision at 84.06%, but the recall is 47.93%. This confirms that the default threshold is conservative: it prioritizes high-confidence review signals rather than maximizing recall. A lower threshold such as $\delta = 2.75$ increases recall to 66.12% and improves F1 on this manual annotation sample, but this is at the cost of lower precision.

Table 3. Threshold behavior against three-annotator manual labels.

δ	Accuracy	Precision	Recall	F1	False Negatives
2.75	79.33%	79.21%	66.12%	72.07%	41
4.00	75.33%	84.06%	47.93%	61.05%	63
5.00	66.33%	88.46%	19.01%	31.29%	98

To further examine model interpretability and disagreement, Table 4 cross-tabulates the threshold decision at $\delta = 4.0$ with the final manual H1–H5 labels. The table shows that the model and manual assessment agree for many high-intensity and low-intensity cases, but the disagreements are also informative. Model high/manual low cases include updates where the vector distance is high but the manual assessment treats the local evidence as non-semantic or only moderately semantic. Such cases suggest that the embedding can be sensitive to body restructuring, abstraction changes, or context shifts that do not clearly alter the implementation semantics visible within the method. Model low/manual high cases include localized control flow, API interaction, return behavior, or exception behavior changes that receive a high manual score even though the vector distance is below the conservative threshold. These cases explain why $\delta = 4.0$ has high precision but lower recall. They also delimit the metric's role as a review triage signal with identifiable disagreement modes.

Table 4. Disagreement analysis between threshold decision and manual intensity.

Model Decision at $\delta = 4.0$	Manual H4–H5	Manual H1–H3
High semantic displacement	58	11
Low semantic displacement	63	168

Table 5 reports a source code–feature attribution analysis for the same model/manual groups. The purpose is not to reverse-engineer GraphCodeBERT internally, but to make the observed high and low distance decisions inspectable through concrete code features. Model high/manual high cases have high rates of large textual edits, call-set changes, and control flow changes. Model high/manual low cases have high call-set change but no control, return, exception, or synchronization change in the extracted features, suggesting that some high distances reflect restructuring or representation shifts that are not judged as high-intensity semantic changes by the manual labels. Model low/manual high cases contain substantial control flow and call interaction changes, explaining why the conservative threshold misses some human-labeled high-intensity updates.

Table 5. Source–feature attribution for model/manual agreement and disagreement.

Group	Samples	Large Edit	Call-Set	Control	Return	Exception	Test-Observable
Model high/manual high	58	91.38%	96.55%	72.41%	15.52%	13.79%	18.97%
Model high/manual low	11	27.27%	90.91%	0.00%	0.00%	0.00%	54.55%
Model low/manual high	63	36.51%	88.89%	61.90%	3.17%	7.94%	26.98%
Model low/manual low	168	4.17%	73.21%	3.57%	0.60%	0.60%	35.71%

A follow-up source-level review examines twelve representative cases from the four cells in Table 4, with three cases selected from each cell. This case review provides source-level interpretability evidence, not runtime behavioral validation; no reviewed case included executable input–output or defect evidence. The model high/manual high cases typically involve changed control flow, data dependency, call interaction, return behavior, or exception handling. For example, one reviewed case adds replication waits and final block-count synchronization, changing the test execution path and failure sensitivity. Model high/manual low cases reveal overestimation patterns: one case removes a local method body because responsibility is likely moved through abstraction or interface extraction, while another only simplifies a generic failed-future expression through Java type inference and formatting cleanup. Model low/manual high cases reveal underestimation patterns: localized changes to high-availability validation paths or error response generation can affect externally checked behavior while remaining below the conservative threshold. Model low/manual low cases mostly involve moderate local test or resource-setup behavior, supporting the interpretation that low distance is usually a lower-priority review signal rather than a claim of behavioral equivalence.

The evaluation also includes a test-observable behavioral proxy check on all 23,389 changed methods located in test paths or test-like methods in the extracted source code pairs. This proxy is independent of GraphCodeBERT distance and manual labels. It marks a test method update as behavior-observable when assertion or verification lines, assertion APIs, exception behavior, return behavior, or substantial test control flow change. This is still not full runtime equivalence: it does not execute the systems or prove production behavior preservation. It provides a stricter check than source-level manual labels for whether a changed test method exposes an observable testing-behavior change. Table 6 shows that $\delta = 4.0$ has high precision but very low recall under this proxy. Therefore, ArchSense should not be claimed as a behavior-preserving refactoring filter. It is a conservative triage signal for implementation-semantic reviews.

Table 6. Test-observable behavioral proxy validation on test method updates.

Method	Samples	Accuracy	Precision	Recall	F1
ArchSense at $\delta = 4.0$	23,389	49.39%	72.30%	4.50%	8.47%
Syntactic edit ratio ≥ 0.35	23,389	52.22%	74.08%	12.63%	21.58%

A component-level production-to-test linkage analysis separates production methods from test methods within the same component and version pair. For each component, it compares production-side semantic intensity with the rate of test method updates that modify assertions, verification calls, exception checks, return checks, or test control flow. Table 7 reports the results for 53 components with at least 10 production updates and at least 5 test updates. Production ASD, production EB, and production P90 distance all show positive associations with test-observable change rates. Components in the top quartile of production ASD have a mean test-observable rate of 55.09%, compared with 34.22% for the remaining components. This analysis does not determine causal direction. The available version pair data cannot show whether production changes triggered test updates, whether tests were strengthened first and production code was then adapted, or whether both were changed as part of the same maintenance task. The result should therefore be read as behavior-facing co-change evidence, not as runtime input–output equivalence or causal proof.

Table 7. Production-to-test behavioral linkage.

Signal	N	ρ	p	Top ASD	Other
ASD	53	0.3832	0.0064	55.09%	34.22%
EB	53	0.3759	0.0036	55.09%	34.22%
P90 distance	53	0.2818	0.0412	55.09%	34.22%

The method testUnZip is selected through a predefined four-quadrant case selection strategy rather than as an isolated anecdote. We classify candidate examples by crossing syntactic edit ratios with semantic displacement: high-syntax/high-semantic, high-syntax/low-semantic, low-syntax/high-semantic, and low-syntax/low-semantic. testUnZip belongs to the high-syntax/low-semantic quadrant. This method update exhibits an extreme syntactic churn rate of 93.27%. A traditional syntactic baseline would identify this as a major change because almost the entire text has been rewritten. However, ArchSense assigns this method pair a semantic distance of only 1.96, placing it below the substantive threshold. This case illustrates one quadrant of our systematic case analysis: large textual change does not necessarily imply large implementation-semantic displacement.

3.2. RQ2: Impact of Semantic Characterization on Change Signal Quality

The second research question examines how semantic characterization affects the quality of change signals used for component-level review triage. The goal is to compare a syntactic change count with the proposed intensity-based measurement. We define signal quality as the extent to which reported changes correspond to substantive implementation evolution.

We apply ArchSense to six large-scale systems to measure the reduction in reported change signals. Figure 1 shows the signal volume on a logarithmic scale. The difference between the baseline bars and the ArchSense bars indicates that semantic characterization substantially reduces the review stream. Across the studied systems, the intensity-based measurement selects a much smaller high-distance subset from the methods reported by the syntactic baseline. This reduction occurs because many executable edits have low measured semantic intensity, including local cleanups and some refactoring-like changes.

It should be interpreted as review–volume reduction under a semantic threshold, not as proof that the unselected methods preserve runtime behavior.

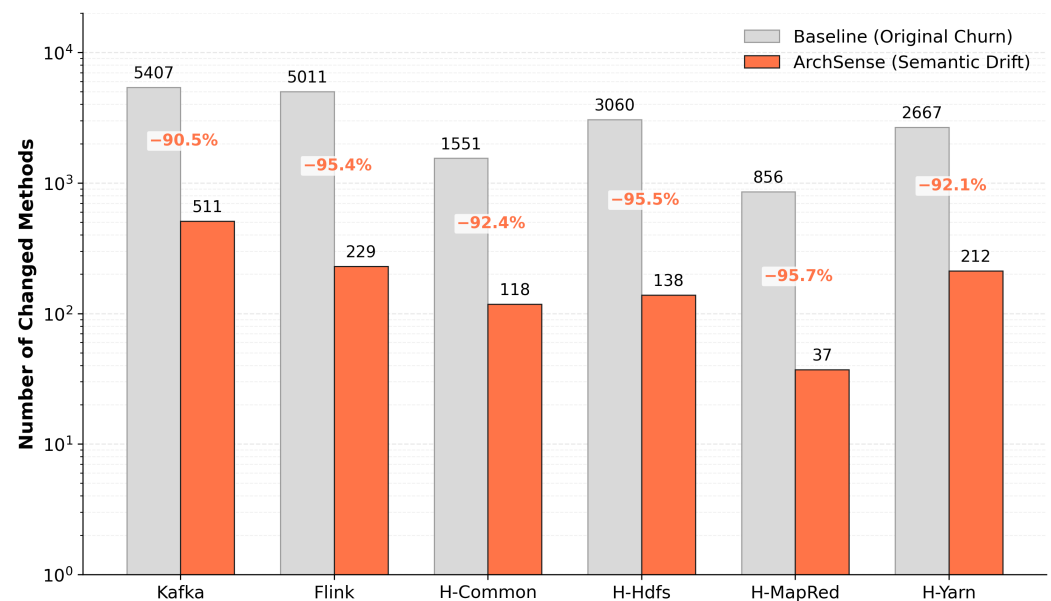


Figure 1. Comparison of change signal volume between baseline and ArchSense.

Table 8 summarizes the change signal volume under the default operating point. The baseline method reports a total number of raw signals, N , and ArchSense selects a smaller set of high-distance implementation-semantic signals, S . We define the syntactic baseline, N , as the total set of methods where developers modified the executable source code. To ensure a rigorous evaluation, we exclude non-executable changes such as whitespace, line breaks, and comments before calculating this baseline. Therefore, every modification in N represents an edit to implementation logic or program structure. We use this baseline because it reflects the manual review burden faced when every executable code edit is treated as a change signal. By using the sanitized executable method set as the baseline, we can evaluate how strongly ArchSense reduces the review stream under a conservative threshold. For example, in Flink, the baseline identifies more than 5000 changed methods, while ArchSense identifies 229 high-distance updates. This indicates that the threshold produces a compact review subset; it does not prove that all filtered-out edits preserve runtime behavior.

Table 8. Cross-project comparison of change signal efficiency.

Project	Baseline Signals, N	Substantive Signals, S	Reduction Rate
Kafka	5407	511	90.5%
Flink	5011	229	95.4%
Hadoop-Common	1551	118	92.4%
Hadoop-HDFS	3060	138	95.5%
Hadoop-MapReduce	856	37	95.7%
Hadoop-YARN	2667	212	92.1%

To examine whether this large reduction over-filters meaningful changes, we performed a supplementary rule-based proxy–audit on 264 stratified method pairs from the extracted method-level change records. This audit uses source code features independently of GraphCodeBERT distance, including control flow changes, method-call-set changes, return and exception behavior changes, and method size shifts. Under this audit, ArchSense obtains an accuracy of 86.36%, a precision of 87.27%, a recall of 62.34%, and an F1-score

of 72.73%. The false-negative count is 29 out of 264 samples. These results show that $\delta = 4.0$ is intentionally conservative: it strongly reduces review-volume, but it can miss some moderate semantic changes. Therefore, the filtering result should be interpreted as reducing the review stream with an explicit precision–recall trade-off, not as eliminating noise without loss. Table 9 reports the proxy-audit comparison.

Table 9. Supplementary proxy-audit for false-negative risk.

Method	Samples	Accuracy	Precision	Recall	F1
ArchSense	264	86.36%	87.27%	62.34%	72.73%
Syntactic Edit Ratio	264	91.67%	92.31%	77.92%	84.51%

The proxy-audit also reports statistical uncertainty and paired significance. The Wilson 95% confidence interval for ArchSense accuracy is 81.70–89.98%, while the corresponding interval for the syntactic edit ratio baseline is 87.71–94.43%. The paired exact McNemar test between ArchSense and the syntactic baseline gives $b = 24$, $c = 10$, and $p = 0.024307$. These results show that the proxy-audit should be interpreted as an operating-point trade-off: the syntactic baseline has higher recall on this proxy task, whereas ArchSense provides a more conservative high-precision review signal.

A direct comparison with RefactoringMiner 3.1.3 uses the official RefactoringMiner API method for directory/file-content comparison because the experimental inputs are released source directories rather than git histories. The analysis is restricted to Java files. Table 10 reports seven representative component–version pairs. Across these pairs, RefactoringMiner identifies 23,011 known refactoring instances. This result supports our positioning that RefactoringMiner and ArchSense answer different but complementary questions: RefactoringMiner detects named refactoring operations, while ArchSense measures semantic displacement intensity and aggregates it to component-level implementation summaries.

Table 10. Supplementary real RefactoringMiner runs on representative components.

Component	Version Pair	Old Java	New Java	Refactorings
Flink-Runtime	1.16.0–1.19.0	1676	1998	12,681
Hadoop-Common	2.7.0–2.10.2	773	1018	2659
Hadoop-Common	2.10.2–3.3.6	886	1307	1856
MapReduce-App	2.7.0–2.10.2	88	106	196
MapReduce-App	2.10.2–3.3.6	92	102	144
YARN-ResourceManager	2.7.0–2.10.2	347	663	3124
YARN-ResourceManager	2.10.2–3.3.6	475	722	2351

Beyond volume reduction ArchSense reveals the average intensity of the remaining signals. We measure ASD over the set of updated methods. A higher ASD indicates a higher average impact per change. Table 11 presents these values. The data reveals a fundamental distinction in evolutionary categories between storage infrastructure and processing engines.

Hadoop-HDFS shows the lowest intensity with an ASD of 0.56. This low value is consistent with the stability-oriented role of a storage layer. Many HDFS changes are defensive maintenance activities, such as adding checks or updating logging behavior. Such edits may alter the source text, but they produce only small shifts in the semantic vector space. ArchSense therefore characterizes this pattern as low-intensity maintenance.

Table 11. Signal intensity of updated methods.

Project	Substantive Evolutions S	Avg. Semantic Displacement ASD
Kafka	511	1.11
Flink	229	1.43
Hadoop-Common	118	0.95
Hadoop-HDFS	138	0.56
Hadoop-MapReduce	37	1.17
Hadoop-YARN	212	1.21

Flink and Hadoop-YARN show higher intensities. This pattern is consistent with their algorithmic responsibilities: Flink handles complex stream-processing windows, while YARN manages dynamic resource scheduling. Evolution in these systems often involves revising core algorithms to improve throughput or latency. Such modifications can substantially change intra-method control flow and data dependencies. The high ASD values indicate that changes in processing layers tend to represent more intensive evolution than the defensive maintenance observed in storage layers.

Hadoop-MapReduce is a relatively stable subsystem with few logical changes, but its ASD is still high at 1.17. This pattern suggests a category of concentrated fixes. Since this subsystem has limited feature evolution, the remaining changes are more likely to correspond to important bug fixes or security-related updates that require deeper logic modification. Unlike the broad but shallow maintenance pattern in HDFS, the maintenance pattern in MapReduce is narrow but relatively deep.

Overall, ArchSense filters out more than 90% of method-level change signals across the studied systems while preserving a measurable intensity signal for the remaining updates. This provides a quantitative basis for prioritizing component-level implementation review. Components with high ASD indicate substantial logic shifts that may require closer component-owner review, whereas low-ASD changes in stable components can be handled through lighter review processes when project policy accepts the recall risk. In this way, the framework converts raw code churn into a more interpretable signal for component-internal evolution management.

A small model ablation on 50 stratified method pairs examines whether the results depend uniquely on GraphCodeBERT. The ablation compares GraphCodeBERT with CodeBERT, UniXcoder, and CodeT5 using the same preprocessing and calibrates each model threshold on the same proxy-audit sample. GraphCodeBERT has the highest rank consistency with the original distance values because the main experiment also uses GraphCodeBERT. CodeBERT, UniXcoder, and CodeT5 achieve comparable proxy-audit F1 values after calibration. This result suggests that the overall framework is not tied to a single encoder, although the absolute threshold δ must be calibrated separately for each model. Table 12 reports the model-ablation results.

The same 50 method pairs are used to compare Euclidean distance with cosine distance. Because all embeddings are L2-normalized, the two metrics are mathematically monotonic. The empirical result is consistent with this property: for all four encoders, the Spearman correlation between Euclidean and cosine distance is 1.000, and the calibrated binary classifications are identical. This supports our use of Euclidean distance for ASD aggregation without suggesting that cosine distance would lead to different rankings on normalized embeddings. Table 13 summarizes this distance-metric comparison.

Table 12. Supplementary model ablation on stratified method pairs.

Model	Samples	Spearman w.r.t. Main Distance	Accuracy	F1
GraphCodeBERT	50	0.996	90.00%	85.71%
CodeBERT	50	0.778	96.00%	94.12%
UniXcoder	50	0.740	94.00%	90.91%
CodeT5	50	0.836	94.00%	91.43%

Table 13. Euclidean and cosine distance comparison on normalized embeddings.

Model	Samples	Spearman Correlation	Classification Agreement
GraphCodeBERT	50	1.000	100.00%
CodeBERT	50	1.000	100.00%
UniXcoder	50	1.000	100.00%
CodeT5	50	1.000	100.00%

3.3. RQ3: Characterization of Architectural Evolution Categories

The third research question investigates the semantic stability and implementation activity patterns across different functional domains. We analyze representative components from six major subsystems. To understand the functional role of these components, we group them into specific domains. We categorize Hadoop-Common, Hadoop-HDFS, and Flink-Runtime into the infrastructure domain. We categorize Hadoop-YARN into the resource management domain. We categorize Hadoop-MapReduce into the compute domain. We aim to identify distinct component-internal evolution categories by jointly examining evolutionary breadth, average semantic displacement, distributional intensity indicators, and API stability metrics across these domains.

ArchSense is a retrospective characterization tool that describes and categorizes the evolutionary state of a system between two observed versions. The RQ3 analysis is organized to avoid using developer issue records as the validation target. We first analyze the component-level metrics and source-level indicators, including control-change rate, call-set change rate, test-observable rate, API influence range, and call-chain depth. Selected issue records are then reported only as historical context for how maintainers described work in the same areas. These records are not behavioral ground truth, because they can reflect project management, release planning, and documentation practice as well as technical changes. We therefore do not use them to define the metric, assign categories, or prove representational validity.

To evaluate the externally visible exposure of these categories, we introduce the PLCR (Public Logic Change Rate). We define this metric as the proportion of substantive implementation evolutions that occur within public-facing methods. A higher PLCR indicates that component-internal evolution reaches more public-facing entry points, although it does not by itself prove external client impact. Given the set of substantive evolutions, $\mathcal{S}_C$, within a component, C , we calculate PLCR as:

$$PLCR = \frac{|\{m \in \mathcal{S}_C \mid m \text{ is public}\}|}{|\mathcal{S}_C|} \quad (5)$$

PLCR is a coarse exposure proxy because a public method is not necessarily used by external clients. We therefore add supplementary static indicators. The API influence range counts substantive public methods as externally visible entry points. Maximum call-chain depth approximates how far a substantive method can influence callers inside the component. Sampled P90 distance provides a distributional intensity indicator, so the analysis is not based only on the mean ASD. These indicators do not replace full dynamic

dependency analysis, but they reduce the risk of interpreting public visibility or simple averaging alone as external exposure. We use a sampling strategy for this analysis because exhaustive inspection of every component is impractical for large systems. To focus on component-relevant implementation changes, we select the most active components from each functional layer as representatives of core system functionality. For Hadoop, we select Hadoop-Common and Hadoop-HDFS to represent the infrastructure. We select Hadoop-YARN for resource management and Hadoop-MapReduce for computation. For Flink, we select the runtime engine. For Kafka, we select the client and stream modules.

Table 14 reports the supplementary static impact indicators for the highest-impact components in the extracted method-level change records. The Kafka streams and Hadoop-HDFS main components show the largest API influence ranges, while several components also reach the maximum sampled call-chain depth of five.

The sampled P90 column shows that components with similar average intensity can still differ in their upper-tail semantic displacement. This result gives additional context to PLCR and ASD: public method ratio or mean distance alone can understate the propagation risk when a small number of substantive public methods sit deep in local call chains.

Table 14. Supplementary API impact and call-chain indicators.

Component	Updates	Subst.	EB	ASD	API Range	Max Depth	P90 <i>d</i>
Kafka-Streams	2334	200	0.0857	2.30	6	5	4.28
Hadoop-HDFS Main	1515	80	0.0528	1.73	6	5	4.83
Hadoop-HDFS Tests	1443	53	0.0367	1.39	4	4	7.18
Hadoop-Common Tests	686	49	0.0714	1.75	3	3	2.77
Kafka-Clients	2204	275	0.1248	2.27	2	4	4.68

Figure 2 visualizes the evolutionary status of these components. The x axis represents evolutionary breadth. The y axis represents ASD. The size of each bubble corresponds to total change frequency. The color indicates the project ecosystem.

To ensure these categories are not isolated observations we perform a longitudinal stability analysis. We track the evolution status of each component across three consecutive versions. Our results show that 82 percent of the components keep the same classification. This consistency shows that the categories assigned by ArchSense reflect recurring component-internal implementation activity patterns rather than a one-time threshold artifact.

To examine the technical properties behind these visualizations, we perform a quantitative triangulation. We relate the visual categories to measures of evolutionary volatility and API stability. Table 15 summarizes this analysis for representative components. We then add source-level technical indicators before using developer records as qualitative historical context.

Table 15. Volatility and API stability validation.

Component	TCF	ASD Mean	ASD Std	PLCR	Category
Kafka-Clients	2204	2.267	1.28	12.5%	Aggressive Evolution
Hadoop-YARN	1392	1.841	1.28	5.2%	Aggressive Evolution
Flink-Runtime	5011	1.560	1.02	0.0%	Defensive Maintenance
Hadoop-HDFS	1515	1.735	1.14	0.0%	Defensive Maintenance
Hadoop-Auth	30	2.323	1.34	0.0%	Hidden Hotspot

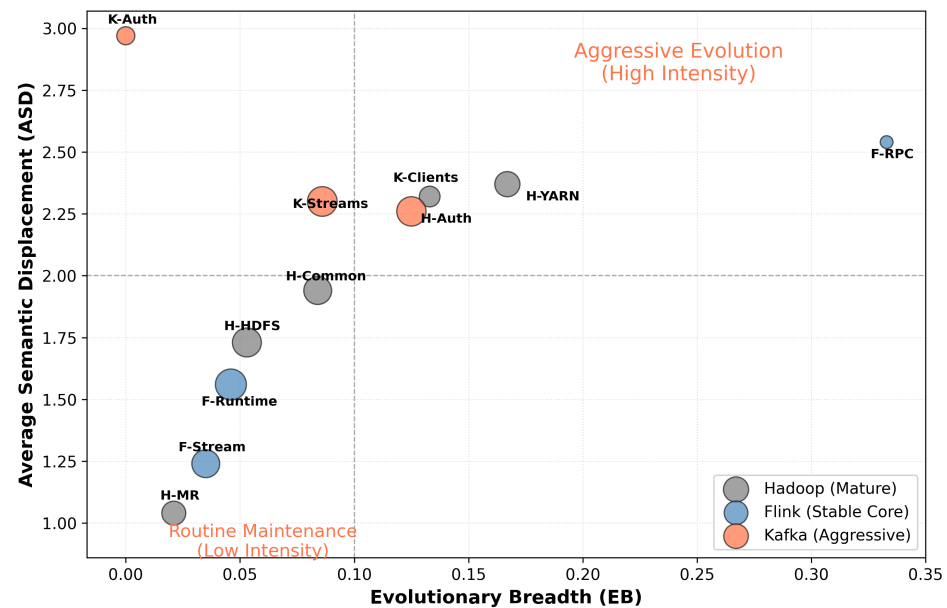


Figure 2. Evolutionary categories across projects.

Because developer records can contain social and project-management noise, we also evaluate whether high-ASD components show source-level technical indicators that are independent of developer issue records. Table 16 compares components in the top quartile of production ASD with the remaining components. High-ASD components have higher production control-change rates, higher call-set change rates, and higher test-observable change rates. The API influence and call-chain indicators are available only for the representative components in Table 15; they are therefore treated as descriptive exposure indicators, not broad statistical evidence. This result defines the role of JIRA in our interpretation: JIRA records are not used as ground truth, but only as qualitative context after the source-level technical indicators have been examined.

Table 16. Technical triangulation with source-level indicators.

Indicator	N	Top ASD	Other	ρ	p
Control-change rate	90	41.65%	27.50%	0.5052	0.0002
Call-set change rate	90	74.72%	66.80%	0.2724	0.0102
Test-observable rate	65	50.76%	29.88%	0.3213	0.0094
API influence range	8	2.00	3.50	0.2887	0.5080
Maximum call-chain depth	8	4.00	5.00	−0.2217	0.5780

After the technical analysis, we report selected developer issue records only as historical context. These records are socio-technical artifacts rather than objective architectural ground truth: they can reflect project priorities, documentation habits, release planning, and personnel decisions. They are not used to validate the metric, assign categories, or infer causal direction. Their role is limited to showing how maintainers described some of the same component areas after the quantitative and source-level analysis has already been completed.

The first archetype is aggressive evolution, exemplified by Kafka and YARN. These components show high semantic displacement exceeding 1.8. Their source-level indicators also show broad control flow and call-set changes, and Kafka has a PLCR of 12.5%, which suggests that part of the change may affect externally visible review surfaces. Selected historical records, such as KAFKA-7548 on consumer buffer management and YARN-5611 on life cycle management APIs, show that maintainers also documented substantial work

in these areas. These records are reported after the metric-based classification and are not used as objective proof of the technical cause. This combination of high ASD, elevated PLCR, and source-level change indicators serves as an implementation-level signal of boundary-relevant rewrites. This distinction is further elaborated in the hidden hotspot analysis below.

The second archetype is defensive maintenance, represented by HDFS and Flink. These layers exhibit lower semantic intensities despite massive change frequencies. Source-level indicators show that their changes are frequent but less concentrated in high-distance method updates than the aggressive evolution examples. A selected historical record, HDFS-14514, describes a complex bug fix in encryption zones. This record helps explain one maintenance context, but the category is assigned from the metric matrix and source-level indicators. ArchSense therefore treats these components as lower-priority implementation-semantic review zones under the current metric, not as proof that runtime behavior or external contracts are preserved.

The third archetype is hidden hotspot represented by the Hadoop-Auth component. This module exhibits a low change frequency of only 30 updates but reaches an average semantic displacement of 2.32 with a high volatility of 1.34. The zero PLCR in Table 15 indicates that the observed displacement is concentrated in internal methods under our static extraction, but it does not prove that the external contract is preserved. A selected historical record, HADOOP-10771, describes related work on security protocol rewriting and authentication handler consolidation. We use this record only to provide historical context for the affected component area. The metric-based result shows that substantive implementation-semantic drift can occur within a very small volume of code modifications. If this intensity were the result of random low-quality code, the modifications would be scattered rather than concentrated in a few core methods.

This phenomenon demonstrates that ArchSense does not merely count change frequency. Instead, it acts as a diagnostic layer that identifies implementation shifts which remain difficult to see through manual inspection of raw change volume. Frequency-based review heuristics often equate a high volume of modifications with high risk, while components with few updates may be assumed to be stable. ArchSense reduces this bias by decoupling syntactic volume from semantic intensity. It identifies concentrated implementation shifts that can occur beneath a stable public method surface. ASD alone does not distinguish between different types of rewrites; this discriminative function is supported by combining ASD with PLCR and the supplementary source-level indicators in Table 16. A component with high ASD and zero PLCR, as seen in Hadoop-Auth, is best interpreted as an internally concentrated implementation-semantic hotspot under static analysis. The dual-metric framework is designed to make this type of internally bounded review target visible, which frequency-based metrics cannot detect.

In summary this experiment shows the value of multidimensional characterization for component-level implementation management. Within the scope of implementation-level semantic analysis, ArchSense distinguishes between the volume of change and the depth of semantic restructuring, providing one complementary input to broader architectural assessment. This distinction supports more precise allocation of quality assurance resources based on empirically observed implementation patterns. Review teams can prioritize manual inspection of aggressive evolution and hidden hotspot components while using automated testing and routine checks for defensive maintenance zones. This approach shifts component monitoring from raw counting toward evidence-based review prioritization.

3.4. RQ4: Threshold Sensitivity and Robustness

The fourth research question examines the robustness of our evolutionary categories. A common risk in threshold based frameworks is that the classification results may be overly sensitive to parameter selection. If the identified categories disappear with threshold adjustments, the classification lacks scientific validity. To verify the robustness of ArchSense, we extended the original sensitivity analysis in two ways. First, we vary the method-level threshold δ from 2.8 to 5.2, corresponding to a wider perturbation range around the default value 4.0. Second, we vary the breadth threshold α and the intensity threshold τ by 10 percent of their original values.

Table 17 summarizes the stability of our classification matrix across these perturbations. We define stability as the percentage of components that maintain the same evolutionary category. Across all six projects, ArchSense demonstrates high classification stability. For instance, in the Hadoop-Common project, 100 percent of the components retained their classification under all tested threshold variations. In more volatile projects like Flink and Hadoop-YARN, the stability remains above 85 percent.

Table 17. Sensitivity analysis of evolutionary categories.

Scenario	Kafka	Flink	H-Common	H-HDFS	H-YARN	H-MapReduce
$\alpha + 10\%$	100%	96%	100%	100%	93%	93%
$\alpha - 10\%$	100%	100%	100%	100%	93%	93%
$\tau + 10\%$	85%	81%	100%	80%	86%	86%
$\tau - 10\%$	100%	96%	100%	96%	78%	85%

Table 18 reports both the global threshold sweep and the project-level calibration check. Panel A shows how the overall substantive-update volume changes as δ varies from 2.8 to 5.2. Panel B checks representative project-version pairs at $\delta = 3.6$ and $\delta = 4.0$ to verify project-level monotonicity and to show that the same absolute threshold can have different review-volume effects across projects.

The minor fluctuations observed in some component categories are expected in a continuous evolutionary spectrum. Components shifting between aggressive evolution and active expansion were located near the decision boundaries in the vector space. This transition confirms that our framework captures the continuous nature of implementation evolution rather than enforcing arbitrary divisions.

The high stability results indicate that the identified categories are stable component-internal implementation patterns under the tested threshold range. This suggests that the measured implementation activity of a component is not an artifact of a single threshold value. By using relative, data-driven thresholds, ArchSense effectively normalizes the varying scales of code churn across different systems.

The extended δ analysis shows the expected monotonic precision-recall behavior. Lower thresholds admit more method updates into the substantive set, while higher thresholds produce a stricter high-confidence subset. The default value $\delta = 4.0$ lies near the middle of this operating range and retains 1988 substantive updates from 38,884 executable method updates across the collected method-level change records. Because substantive updates are defined by $d(m) \geq \delta$, every project-level count must be monotonically non-increasing as δ increases; the computed curves satisfy this property. For example, Hadoop-Common 2.7.0–2.10.2 decreases from 172 updates at $\delta = 3.6$ to 118 at $\delta = 4.0$, and Hadoop-Common 2.10.2–3.3.6 decreases from 123 to 91. The relative drop from $\delta = 3.6$ to $\delta = 4.0$ ranges from 24.3% in Hadoop-MapReduce 2.10.2–3.3.6 to 41.4% in Flink 1.19.0–2.0.0, confirming that the same absolute threshold has different review-volume effects across projects. Overall, the sensitivity analysis confirms that the dual-metric classification is

informative under a range of parameter settings, while also making explicit that δ controls a review–volume trade-off rather than a universal semantic boundary.

Table 18. Threshold sensitivity and project-level calibration of δ .

Panel A: Overall threshold sweep				
	δ	Substantive Updates	Mean Substantive Rate	Mean Component EB
	2.8	5861	14.93%	0.149
	3.2	4154	10.74%	0.104
	3.6	2929	7.66%	0.068
	4.0	1988	5.25%	0.050
	4.4	1253	3.39%	0.031
	4.8	730	2.11%	0.020
	5.2	398	1.17%	0.012
Panel B: Project-level monotonicity check				
Project–Version Pair	$\delta = 3.6$	$\delta = 4.0$	Drop	Monotonic
Flink 1.16–1.19	616	409	33.6%	Yes
Flink 1.19–2.0	302	177	41.4%	Yes
Hadoop-Common 2.10–3.3	123	91	26.0%	Yes
Hadoop-Common 2.7–2.10	172	118	31.4%	Yes
Hadoop-HDFS 2.10–3.3	231	146	36.8%	Yes
Hadoop-HDFS 2.7–2.10	202	138	31.7%	Yes
Hadoop-YARN 2.7–2.10	293	212	27.6%	Yes
Kafka 2.1–3.0	721	511	29.1%	Yes

The default value $\delta = 4.0$ should not be interpreted as a project-independent universal constant. It is the conservative operating point used in this empirical setting. For a new project, the threshold should be calibrated using a small stratified sample or a rule-based proxy–audit. A practical calibration procedure is to sample method pairs across the project-specific distance distribution, label or audit their implementation impact, sweep candidate thresholds, and select the operating point according to the intended precision–recall trade-off. When manual labels are unavailable, a percentile-based threshold can be used as an initial screening value and then refined after review feedback. This adaptive calibration view is consistent with the role of δ as a review triage parameter rather than as an intrinsic semantic boundary.

To make this calibration step reproducible, Table 19 reports three concrete project-specific calibration rules. A budgeted quantile threshold selects the top $1 - q$ share of method distances for review. An IQR threshold selects high-distance outliers using $Q_3 + 1.5IQR$. When a small audited sample is available, a label-guided sweep selects the threshold that maximizes F1 or maximizes recall subject to a target precision. The example values confirm that project-specific thresholds differ substantially. For instance, the 90th percentile threshold is 2.3433 for Hadoop-MapReduce 2.10.2–3.3.6, but it is 3.9361 for Kafka 2.1.0–3.0.0. This result indicates that a single absolute threshold should not be treated as project-independent.

A supplementary threshold sweep uses the 264-method proxy–audit sample. This analysis is not used to replace the original calibration set, but it provides an additional check of the operating point. The best F1 on this proxy–audit occurs at $\delta = 2.75$, with precision 80.26%, recall 79.22%, and F1 79.74%. The default $\delta = 4.0$ has higher precision at 87.27% but lower recall at 62.34%. This confirms that the selected threshold is conservative and prioritizes high-confidence component-review signals over maximum recall. Table 20 reports the proxy–audit threshold sweep.

Table 19. Project-specific adaptive threshold calibration examples.

Project–Version Pair	$\delta = 4.0$ Rate	$Q_{90} \delta$	Q_{90} Rate	$Q_{95} \delta$	Q_{95} Rate	IQR δ
Flink 1.19–2.0	3.19%	2.9135	10.00%	3.6651	5.01%	3.7438
Hadoop-Common 2.7–2.10	7.61%	3.7337	10.06%	4.4437	5.03%	4.7765
Hadoop-HDFS 2.10–3.3	4.35%	3.1456	10.01%	3.8978	5.01%	3.9842
Hadoop-MapReduce 2.10–3.3	3.22%	2.3433	10.01%	3.3432	5.06%	2.5808
Hadoop-YARN 2.7–2.10	7.95%	3.7191	10.01%	4.3475	5.02%	5.1559
Kafka 2.1–3.0	9.45%	3.9361	10.01%	4.6250	5.01%	5.1896

Table 20. Supplementary proxy threshold sweep.

δ	Accuracy	Precision	Recall	F1	False Negatives	False Positives
2.75	88.26%	80.26%	79.22%	79.74%	16	15
4.00	86.36%	87.27%	62.34%	72.73%	29	7

The final analysis examines potential feedback loop effects. ArchSense follows a pairwise computational pipeline from structural alignment to method-level semantic measurement and component-level aggregation, but this pipeline should not be interpreted as a causal model of software evolution. In real systems, semantic and structural changes can be bidirectionally coupled across releases. To make this boundary empirical, Table 21 reports a lagged component-level association check across the consecutive Flink and Hadoop version intervals available in our dataset. Previous-interval ASD has only a weak and statistically unsupported association with next-interval ASD or next-interval test-observable change rate. This result delimits the model’s use to observed two-version differences; explaining feedback-driven system evolution requires longitudinal structural-semantic modeling. The table is a coarse association check at the version pair level. It cannot determine whether semantic changes drive structural changes, whether structural redesign later drives implementation rewrites, or whether both are coordinated by release planning.

Table 21. Lagged component-level association check.

Relation	N	ρ	p	High- t Next	Other Next
ASD _{t} vs. ASD _{$t+1$}	27	0.2515	0.2124	28.57%	25.00%
EB _{t} vs. EB _{$t+1$}	27	0.2118	0.2936	28.57%	25.00%
ASD _{t} vs. Test _{$t+1$}	27	0.2037	0.3956	31.52%	31.87%

This boundary motivates a systems-oriented future direction. A stronger model would treat structural topology, connector changes, runtime communication, and implementation semantics as co-evolving state variables across multiple versions. Such a model would allow structural and semantic changes to influence each other over time, while the present framework is limited to retrospective characterization of observed version pairs.

4. Related Work

This section reviews related work from three perspectives. First, we discuss studies on software architecture change analysis that primarily rely on structural information. Second, we review research on refactoring detection and behavior-preserving code changes. Third, we summarize work on pretrained code representations and semantic code similarity. Finally, we distinguish our work from existing studies.

4.1. Architecture Change Analysis

A large body of research investigates software architecture change analysis by examining structural dependencies and code organization. Repository mining techniques

are commonly used to identify architectural changes across versions. Mondal et al. [14] analyze architectural change commits using syntactic categories and natural language information. Their approach focuses on identifying locations of architectural change but does not distinguish behavior-preserving refactorings from semantically significant updates.

Architecture recovery techniques reconstruct architectural views from source code using static analysis or information fusion. Zhang et al. [15] propose a multi-source architecture recovery approach to extract architectural structures from evolving systems; while they are effective in recovering structural views, such approaches typically treat all detected changes as equally relevant, leading to high noise in cases where refactoring activity is frequent.

Other studies analyze architectural evolution in specific paradigms. Genfer and Zdun [16] investigate architectural evolution categories in microservice-based systems using static dependency analysis. Although these approaches provide insights into structural trends, they do not explicitly address semantic preservation or refactoring-induced noise.

Overall, existing architecture change analysis techniques primarily rely on structural differences. Semantic similarity between code versions is rarely considered, resulting in a large number of syntactically detected but semantically similar changes. This creates a gap between architecture-level change localization and the implementation-level meaning of the changed code.

4.2. Refactoring Detection and Behavior-Preserving Changes

Refactoring detection has been extensively studied as a means to identify behavior-preserving code transformations. RefactoringMiner [17] detects refactorings using AST differencing and rule-based heuristics, achieving high accuracy for known refactoring types. Subsequent studies extend these techniques to support additional refactoring categories or programming languages.

Recent work applies machine learning models to refactoring detection. For example, Li et al. [18] establish models to detect code changes. These approaches focus on identifying specific refactoring types rather than on reducing noise in higher-level change analysis.

While refactoring detection techniques are effective at recognizing known behavior-preserving transformations, they are typically designed as standalone tools. Their results are rarely integrated into software architecture change analysis pipelines. Moreover, refactoring detectors may miss unrecognized or composite refactorings, limiting their ability to comprehensively reduce noise in large-scale systems.

4.3. Semantic Code Representations and Similarity Analysis

Pretrained models for source code representation learning have shown strong performance on a variety of code understanding tasks. Models such as CodeBERT [19], GraphCodeBERT [20] and CodeT5 [21] learn vector representations that capture statistical and structural regularities in code. These representations enable semantic similarity analysis beyond surface-level syntax.

Semantic code representations have been widely used in applications such as code search, clone detection, and program repair. Several studies demonstrate that embedding-based similarity measures can identify functionally similar code fragments even when their syntactic forms differ.

However, most existing work applies semantic code similarity at the level of individual functions or code snippets. Its role in software architecture change analysis has not been explored. In particular, few studies investigate how semantic similarity can be used to filter syntactically detected changes and reduce refactoring-induced noise in architecture-level analysis.

The three strands of work reviewed above address complementary parts of the problem. Architecture change analysis provides structural context, refactoring detection explains known behavior-preserving transformations, and pretrained code models provide a representation space for implementation semantics. The key gap is not the absence of structural analysis or semantic models separately, but the lack of a scoped fusion layer between them. Existing architecture studies usually start from dependencies, modules, or recovered views, while semantic representation studies usually optimize local code tasks such as search, clone detection, or repair. These two lines of work therefore answer different questions. Structural methods can identify where the architecture changed, but they often cannot distinguish widespread shallow maintenance from concentrated deep implementation rewrites. Embedding-based methods can compare code snippets, but without component alignment they cannot explain how local semantic movement is distributed across architectural containers.

Existing fusion attempts remain limited for the purpose of this study. Architecture recovery methods can combine static dependencies and other project signals, but they do not quantify semantic intensity within matched methods. Refactoring detectors can identify known behavior-preserving transformations, but they are not designed to aggregate residual implementation-semantic movement at the component level. As a result, prior work does not directly answer whether a component has broad low-intensity maintenance, concentrated high-intensity rewriting, or boundary-relevant implementation change. This motivates the positioning of ArchSense in the following subsection.

4.4. Summary and Positioning

In summary, prior work on architecture change analysis focuses primarily on structural differences, while research on refactoring detection and semantic code representation addresses behavior preservation at the code level. The integration of semantic similarity analysis into architecture change analysis remains limited.

This paper bridges these areas by introducing a semantic-aware characterization framework that complements structure-based architecture change analysis. Rather than detecting refactoring types, proving behavior preservation, or inferring complete architectural intent, our approach uses pretrained code representations to measure component-internal implementation-semantic displacement. By focusing on triage and empirical analysis, this work provides a practical approach to make large change streams more inspectable for architecture-oriented review.

5. Discussion and Limitations

This section discusses the practical integration of ArchSense into development workflows, its limitations, and the threats to the validity of our study.

5.1. Practical Implications and Architectural Insights

The primary advantage of ArchSense is its ability to quantify component-internal implementation evolution. This capability provides review insights beyond simple change-frequency counting.

Architecture review often relies on change frequency to gauge risk. A component with thousands of modifications is perceived as highly volatile and risky. Conversely, a component with very few modifications is assumed to be stable. This frequency-based heuristic is effective for localized code reviews but fails at the architectural scale. It conflates the volume of syntactic churn with the intensity of semantic evolution.

Our framework addresses this limitation; while the method-level threshold is a calibrated operating point, the component-level aggregation reveals evolutionary patterns

that frequency-based inspection can miss. For example, ArchSense identified the hidden hotspot archetype. Components in this category such as Hadoop-Auth have extremely low change frequencies. Inspection based only on commit counts may ignore these components. However the model computes an exceptionally high average semantic displacement for them. This reveals a concentrated and substantial restructuring of core logic such as security protocols.

By decoupling the volume of change from the intensity of implementation change, ArchSense identifies component-internal semantic drift that can remain invisible to traditional frequency-based monitoring. This demonstrates that semantic representation models can provide useful review signals beyond simple frequency-based heuristics.

Furthermore this capability allows for integration into Continuous Integration and Continuous Deployment pipelines. ArchSense can act as a semantic triage tool during code review. When a developer submits a pull request, the framework calculates the semantic distance of the modified methods. Changes with a distance below the threshold can be routed to lighter implementation review only when project policy accepts the associated recall risk, while updates that exceed the threshold can be flagged for closer component-owner inspection. This triage focuses review attention on the most intense implementation-semantic signals, but it should not replace tests, refactoring detectors, or runtime behavioral validation.

The dual-metric classification matrix offers a multidimensional view of component-internal implementation activity. It enables development teams to distinguish between localized deep implementation restructuring and widespread low-intensity maintenance based on quantitative evidence. This can support technical debt and review risk management over long-term development cycles when combined with dependency, test, and runtime evidence.

5.2. Limitations and Future Work

ArchSense faces several limitations:

First, ArchSense measures implementation-level semantic change and does not model cross-component connector topology or runtime communication patterns. This defines the scope of the framework. A complete architectural model must incorporate inter-component interactions, because emergent architectural properties arise from these interactions. The present study isolates one component-internal dimension that existing tools do not address and examines it with manual construct assessment, source-level indicators, test-observable proxies, and selected historical context. ArchSense is designed to function as one diagnostic layer within a broader analytical toolkit, intended to be composed with topology-based dependency tools rather than used as a standalone architecture-evolution model. Changes that primarily alter communication protocols or external API dependencies between components whose internal method semantics remain similar fall outside our measurement scope and require complementary dependency-graph analysis. Therefore, conclusions drawn from ArchSense should be limited to component-internal implementation-semantic activity unless additional connector and runtime evidence is supplied. Future work will integrate our semantic displacement algorithm with architectural dependency graphs to develop a perspective that combines internal semantic signals with external topological change signals.

Second, the framework relies on the initial structural alignment. The alignment algorithm may produce mapping errors when developers perform complex operations such as method splitting or merging. Extreme structural churn can impact the subsequent semantic measurement. We plan to integrate more advanced mapping techniques to improve alignment accuracy.

Third, the accuracy of our categories relies on the quality of the pretrained embeddings. GraphCodeBERT captures intra-method data flow but it may struggle with highly specific domain frameworks. Our framework is primarily optimized for modifications in control flow and data dependencies. Therefore, certain types of updates result in lower semantic displacement than their actual component-level impact suggests. This specific failure scenario involves changes to configuration strings, resource identifiers, or constant literals. These elements often function as static values within the implementation logic; while the model identifies these as token modifications, they do not fundamentally alter the data flow graph of the method. Furthermore our method operates at the method level and does not account for changes in imports or configuration files which are also key elements of software architecture. Future research will evaluate newer models and incorporate inter component interaction analysis to further enhance semantic precision.

Fourth, scalability is a practical limitation. Full-project analysis requires parsing many Java files and encoding every changed method pair. The implementation therefore separates lightweight post-processing experiments from model-dependent embedding experiments and supports cached outputs. For very large repositories, batch encoding, persistent embedding caches, and incremental pull-request level analysis are necessary for deployment.

Fifth, the current validation is retrospective. Selected developer-documented events provide historical context, but they do not prove that semantic displacement causes later architectural restructuring. JIRA and similar records are socio-technical artifacts influenced by project management, release planning, communication practices, and documentation habits. They are therefore used only as qualitative historical context, not as objective technological ground truth. A stronger prospective validation would compute semantic displacement before the documented restructuring event and test whether it predicts future architectural changes. This predictive setting is left for future work, and the present study avoids causal claims.

Sixth, the framework follows a pairwise computational pipeline from structural alignment to method-level semantic measurement and component-level aggregation. This pipeline should not be interpreted as a causal model of software evolution. In real systems, semantic and structural changes can be bidirectionally coupled across releases: an implementation rewrite in one version may trigger structural reorganization in a later version, and a structural redesign may in turn drive implementation changes. As shown in Table 21, our lagged component-level check does not provide statistically supported evidence for such feedback effects. Capturing feedback loops requires a multi-version temporal model that jointly tracks structural dependencies, connector changes, runtime communication, and implementation-level semantic shifts. A future two-way coupling model should represent structure and semantics as co-evolving state variables, so that changes in one dimension can influence the other over time. This is beyond the two-version diagnostic setting studied here.

5.3. Threats to Validity

Construct validity concerns whether our metrics accurately capture implementation evolution. Vector displacement is an approximation of semantic change rather than a formal verification. A large distance might result from non-functional text changes rather than actual logic updates. We mitigate this risk by using a code sanitization pipeline to remove comments and annotations before embedding. GraphCodeBERT is sensitive to data flow structures. It may assign a large distance to a change that preserves external behavior but alters internal data flow. The three-annotator study therefore serves as a construct-level assessment of implementation-semantic change, not as behavioral-level validation. The

test-observable behavioral proxy experiment provides an additional check, but it still does not execute production systems or prove runtime equivalence. Accordingly, evolution intensity is defined as a measure of implementation restructuring and review effort rather than runtime behavioral equivalence.

Internal validity relates to the correctness of our inferences. A bias arises from the selection of distance thresholds. We address this risk by reporting threshold sensitivity analysis, proxy–audit results, and the three-annotator manual annotation study. These analyses show that the component categorizations are informative under a range of parameter settings, while also making clear that $\delta = 4.0$ is a conservative operating point rather than a project-independent constant. The manual study reports inter-annotator agreement before adjudication, but the annotators did not use dynamic runtime traces or complete system-wide call graphs; this remains a limitation of the validation context. Developer-documented JIRA records introduce another threat: they are socio-technical artifacts that may be affected by team priorities, release planning, and documentation habits. We therefore use them only as qualitative historical context rather than objective ground truth.

External validity concerns the generalizability of our results. Our study is limited to large-scale Java systems. These results may not apply to small scale applications or different programming paradigms. However the underlying neural model is trained on a multi lingual corpus. Therefore, the semantic analysis capability of ArchSense is theoretically adaptable to other programming languages.

6. Conclusions

In this paper, we proposed ArchSense, a framework designed to characterize component-internal implementation-semantic evolution. The framework shifts the focus from simple syntactic change detection to a quantitative assessment of implementation-level semantic displacement. By integrating hierarchical structural analysis with neural embeddings, our approach quantifies the magnitude of method-level implementation changes within a semantic representation space and summarizes this signal inside architectural components.

Our empirical evaluation on six large-scale software systems shows that semantic displacement is statistically associated with manual construct assessment of implementation change intensity. The dual-metric matrix characterizes distinct component-internal implementation activity patterns across functional domains. We found that components within the infrastructure domain exhibit lower-scope defensive maintenance patterns, while components in the resource management domain show stronger innovation-oriented implementation activity. These findings should be interpreted as component-internal implementation-semantic characterization rather than as a complete architectural model.

This work provides three primary contributions. First, it introduces a framework for measuring implementation-level semantic change intensity inside architectural components and examines it with manual construct assessment, source-level indicators, test-observable proxies, and selected historical context. Second, it shows that embedding based semantic distance can serve as a conservative review triage signal, while test-observable behavioral proxy results show that it should not be treated as proof of behavior preservation. Third, it provides an empirical characterization of implementation-semantic evolution that helps connect micro level code changes with component-level review reasoning.

Future work will address the current limitations of analyzing components in isolation. We plan to extend ArchSense with architectural dependency graphs so that dependency changes, call-graph patterns, and dependency strength can be analyzed alongside internal semantic shifts. We will also explore a two-way coupling model in which structural

topology and implementation semantics are treated as co-evolving signals across multiple versions. Further validation should include runtime behavioral evidence and prospective prediction. A leave-one-version-out evaluation could compute the semantic displacement distribution on versions V_1 through V_{n-1} , identify high-displacement components, and compare them with later JIRA records from the V_{n-1} to V_n period without post hoc selection. This design would test whether implementation-level semantic change intensity can anticipate developer-documented restructuring events before they are recorded, complementing the retrospective characterization methodology of the current work.

Author Contributions: T.W.: Conceptualization, Methodology, Software, Writing—Original Draft; G.J.: Supervision, Writing—Review and Editing, Validation, Investigation, Data Curation; H.Z.: Supervision, Project administration, Funding acquisition, Writing—Review and Editing, Validation, Investigation, Data Curation. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported partially by the National Natural Science Foundation of China (No. 62302012, 62401012), and the Anhui Provincial Department of Education, China (No. 2023AH051117).

Data Availability Statement: The source code, intermediate results, and datasets are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Mondal, A.K.; Schneider, K.A.; Roy, B.; Roy, C.K. A survey of software architectural change detection and categorization techniques. *J. Syst. Softw.* **2022**, *194*, 111505. [\[CrossRef\]](#)
2. Wang, T.; Li, B. Esarcost: Estimating repair costs of software architecture erosion using slice technology. *J. Syst. Softw.* **2024**, *208*, 111875. [\[CrossRef\]](#)
3. Cai, Y.; Kazman, R. Software design analysis and technical debt management based on design rule theory. *Inf. Softw. Technol.* **2023**, *164*, 107322. [\[CrossRef\]](#)
4. Li, R.; Liang, P.; Soliman, M.; Avgeriou, P. Understanding software architecture erosion: A systematic mapping study. *J. Softw. Evol. Process.* **2022**, *34*, e2423. [\[CrossRef\]](#)
5. Janke, M.; Mäder, P. 7 Dimensions of software change patterns. *Sci. Rep.* **2024**, *14*, 6141. [\[CrossRef\]](#) [\[PubMed\]](#)
6. Chondamrongkul, N.; Sun, J.; Warren, I. Formal software architectural migration towards emerging architectural styles. In *European Conference on Software Architecture*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 21–38.
7. Wang, T.; Li, B.; Zhu, L. A quality-driven iterative evolution approach for software architecture. *Int. J. Softw. Eng. Knowl. Eng.* **2023**, *33*, 231–255.
8. Jin, W.; Zhong, D.; Cai, Y.; Kazman, R.; Liu, T. Evaluating the impact of possible dependencies on architecture-level maintainability. *IEEE Trans. Softw. Eng.* **2022**, *49*, 1064–1085. [\[CrossRef\]](#)
9. Zakeri-Nasrabadi, M.; Parsa, S.; Jafari, S. Measuring and improving software testability at the design level. *Inf. Softw. Technol.* **2024**, *174*, 107511. [\[CrossRef\]](#)
10. Buschhaus, C.; Netz, L.; Rumpe, B.; Stachon, M.; Stüber, S. Semantic analysis of domain model evolution in model-driven software development. In *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*; IEEE: Piscataway, NJ, USA, 2025; pp. 99–103.
11. Hoang, T.; Kang, H.J.; Lo, D.; Lawall, J. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 518–529.
12. Liu, Z.; Tang, Z.; Xia, X.; Yang, X. Ccrep: Learning code change representations via pretrained code model and query back. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 17–29.
13. Wang, T.; Li, B. A software architecture multiple-level change detection method based on two-step maat matching. *Acta Electron. Sin.* **2023**, *51*, 694–700.
14. Mondal, A.K.; Roy, B.; Schneider, K.A. An exploratory study on automatic architectural change analysis using natural language processing techniques. In *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*; IEEE: Piscataway, NJ, USA, 2019; pp. 62–73.

15. Zhang, Y.; Xu, Z.; Liu, C.; Chen, H.; Sun, J.; Qiu, D.; Liu, Y. Software architecture recovery with information fusion. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*; ACM: New York, NY, USA, 2023; pp. 1535–1547.
16. Genfer, P.; Zdun, U. Exploring architectural evolution in microservice systems using repository mining techniques and static code analysis. In *European Conference on Software Architecture*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 157–173.
17. Tsantalis, N.; Ketkar, A.; Dig, D. Refactoringminer 2.0. *IEEE Trans. Softw. Eng.* **2020**, *48*, 930–950. [[CrossRef](#)]
18. Li, T.; Zhang, Y. Multilingual code refactoring detection based on deep learning. *Expert Syst. Appl.* **2024**, *258*, 125164. [[CrossRef](#)]
19. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; pp. 1536–1547.
20. Guo, D.; Lu, S.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D. Graphcodebert: Pre-training code representations with data flow. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Virtual Event, 3–7 May 2021.
21. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C.H. CodeT5: Identifier-aware unified pretrained encoder-decoder models for code understanding and generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2021; pp. 8696–8708.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.