Intelligent Methods for Software Architecture and Business Logic Recovery: Current State and Prospects for Application

Vasyl Tsurkan [1,†], Andrii Ponepaliak[2,*,†]

[1] 2 National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 37, Prospect Beresteiskyi (former Peremohy), Kyiv, Ukraine, 03056, Ukraine
[2] Donetsk National Technical University, 76, Sambirska Str., Drohobych, Lviv region, 82111, Ukraine

Abstract

This article critically examines intelligent methods for recovering software architecture and business logic in polyglot enterprise systems. The central issue is evidence control. Existing approaches can identify relationships between software artifacts, but these relationships often differ in origin, reliability, and analytical meaning. A direct structural dependency is not the same as a semantic association. A configuration reference is not the same as a language-model-generated hypothesis. The review covers classical architecture recovery methods, semantic code representations, large language models, graph-based program representations, and graph neural network approaches. Each group contributes a useful but limited form of knowledge. Classical recovery methods provide traceable structural dependencies; semantic models reveal potentially related artifacts; large language models formulate interpretive explanations; and graph-based models place artifacts within a broader system context. Yet none of these approaches is sufficient on its own when business logic is scattered across implementation, configuration, data-access, and integration layers. The article argues that business logic recovery should be treated not only as relationship detection, but as the organization of heterogeneous evidence. This distinction is crucial. If confirmed dependencies, semantic similarities, configuration links, and generated interpretations are merged without separation, the recovered model may appear complete while losing analytical transparency. The review therefore identifies the need for a traceable semantic-graph representation, contributing to the field of systems analysis and mathematical modeling of complex software systems.

Keywords

Architecture Recovery, Business Logic Recovery, Program Comprehension, Enterprise Systems, Code Embeddings, Large Language Models, Graph Neural Networks, Semantic Gap, Business Knowledge Decay.

1. Introduction

Software maintenance rarely begins with a complete understanding of the system being modified. In the context of modern information technology, managing this software evolution requires advanced methodologies. Specifically, within the domain of systems analysis and mathematical modeling of complex systems, the task of architecture and business logic recovery represents a critical challenge. In long-lived systems, developers often work with partial documentation, outdated architectural assumptions, and implementation details that no longer correspond to the original design. To make safe changes, they must reconstruct how the system is organized, which components carry its main responsibilities, and how these components interact during execution and evolution. This is a cognitive task as much as a technical one. In program comprehension research, such work is associated with building a mental model of the system, without which even local modifications may produce unintended architectural or behavioral effects [1]. Later studies have extended this view by examining theories, methods, and tools that support software comprehension at different levels of abstraction [2].

[1]RS-2026: Resilient Systems: Secure Digital Technologies and Critical Infrastructure, June 30, 2026, Drohobych, Ukraine
* Corresponding author.
† These authors contributed equally.
v.v.tsurkan@gmail.com (V. Tsurkan); andrii.ponepaliak.asp@donntu.edu.ua (A. Ponepaliak);
0000-0003-1352-042X (V. Tsurkan); 0009-0007-3514-2841 (A. Ponepaliak)

Architecture recovery and architecture reconstruction address this problem by deriving higher-level views of a system from its existing artifacts. They help move from implementation-level elements to a more coherent understanding of components, dependencies, and architectural organization [3]. In software re-engineering, this reconstructed view supports the analysis, transformation, and evolution of legacy systems [4]. Yet architecture alone does not explain everything. A recovered dependency can show that two components are connected, but it does not necessarily reveal the business purpose of that connection.

This limitation becomes especially visible in polyglot enterprise systems. Business logic in such systems is rarely located in one class, module, or service. It may be distributed across source code, configuration files, database queries, API contracts, workflow definitions, access-control mechanisms, and external integrations. A domain rule that appears simple at the business level may depend on several technical artifacts at once. For example, a pricing rule, eligibility check, or order-status transition can involve a service method, an SQL query, a configuration parameter, a validation rule, and a response from an external system. The code is only part of the evidence.

For this reason, business logic recovery requires more than classical structural reconstruction. It also requires semantic interpretation, contextual analysis, and a way to distinguish different kinds of relationships between artifacts. A direct method call, a configuration reference, a semantic similarity, and a language-model-generated explanation do not have the same analytical status. Similarity is not equivalence. An explanation is not proof. If these relationships are merged without distinction, the recovered model may look comprehensive while hiding uncertainty.

The aim of this article is to critically review intelligent methods that can support the recovery of software architecture and business logic in polyglot enterprise systems. The review covers classical architecture recovery methods, semantic and transformer-based code representations, large language models, graph-based program representations, and graph neural network approaches. The central argument is that the main challenge is not only to detect relationships between software artifacts, but to preserve their evidential status. Business logic recovery should therefore be treated as an evidence-aware reconstruction process in which structural facts, semantic associations, configuration links, and interpretive hypotheses remain traceable and distinguishable.

2. Relevance of Research

The relevance of this research is shaped by the way enterprise software systems evolve. They rarely remain aligned with their initial architectural design. Over time, new requirements, changing development teams, framework migrations, integrations, and local technical decisions gradually transform the system. Documentation usually lags behind. Architectural knowledge becomes fragmented.

For developers and architects, this creates a practical problem. Maintaining such systems requires more than reading isolated classes, methods, or services. A change in one place may affect validation logic, data access, configuration, API behavior, or integration with an external system. The visible dependency is only one part of the maintenance risk. The business consequence of that dependency is often harder to recover.

Architecture recovery and reconstruction remain relevant because they provide a way to derive structural views from existing software artifacts [3]. In software re-engineering, these views support the understanding and transformation of legacy systems [4]. They help identify components, dependencies, and architectural erosion. This is necessary. It is not sufficient.

The limitation becomes clearer when the object of recovery is business logic rather than architecture alone. Enterprise business rules are often implemented across several technical layers. A discount rule may involve service logic and database queries. An access restriction may depend on annotations, configuration, and runtime security mechanisms. A payment or order-processing scenario may cross internal services, message queues, and external APIs. In such cases, structural reconstruction can show where artifacts are connected, but it does not fully explain what business behavior they implement together.

This is why intelligent methods have become increasingly relevant for software analysis. Machine learning for code can reveal semantic regularities across program artifacts [6]. Large language models can connect implementation fragments with natural-language context, identifiers, comments, documentation, and task descriptions [7]. Graph-based representations can preserve the context in which artifacts interact. These methods broaden the analytical perspective beyond direct dependencies.

However, their growing relevance also creates a methodological risk. Semantic similarity, generated explanations, graph proximity, and confirmed structural dependencies are not the same kind of evidence. If they are combined without distinction, the recovered model may become less reliable, not more. The problem is not only to find more relationships. The problem is to know what kind of relationship has been found.

Therefore, this research is relevant because it addresses a gap between technical recovery and evidence-aware interpretation. Polyglot enterprise systems require models that can connect heterogeneous artifacts while preserving the origin, type, and reliability of each recovered relationship. Such models are important for maintenance, modernization, program comprehension, and business logic recovery, especially when domain knowledge is no longer fully available in documentation or in the memory of the original developers.

3. Statement of the Problem

The problem addressed in this article is the lack of an integrated approach to business logic recovery that combines structural reconstruction, semantic interpretation, and evidence control. Existing architecture recovery methods can reconstruct components, dependencies, call relations, control-flow structures, and selected data-flow relations [3]. These results are valuable. They are also incomplete.

The incompleteness appears at two levels. First, many meaningful relationships in enterprise systems are not expressed as direct calls or explicit dependencies. They may be introduced through dependency injection, inversion of control, aspect-oriented programming, reflection, runtime binding, configuration files, framework conventions, or external integration rules. As a result, a recovered call graph or dependency graph may represent only part of the actual system behavior.

Second, even a correctly recovered technical relationship does not automatically explain its business role. A service may call another service, access a repository, validate a field, or execute an SQL query. These facts show technical interaction. They do not by themselves show whether the corresponding fragments implement credit-limit validation, discount calculation, access control, order blocking, compliance checking, or another domain rule. The business meaning remains implicit.

This creates a deeper methodological gap. Business logic in polyglot enterprise systems is not confined to source code. It may be distributed across implementation logic, database queries, object-relational mappings, configuration parameters, API contracts, test cases, documentation fragments, and external services. An approach that analyzes only one artifact type therefore produces a partial view of the system. Business logic recovery requires a representation that can connect heterogeneous artifacts without reducing them to the same analytical category.

The central difficulty is evidential. A direct method call, a data-flow dependency, a configuration reference, a semantic similarity, and an explanation generated by a large language model have different origins and different levels of reliability. They should not be represented as equivalent facts. A structural dependency can often be traced to a concrete implementation element. A semantic association is probabilistic. A model-generated explanation is interpretive. These differences matter.

If such relationships are merged without distinction, the recovered model may become misleading. It may suggest that a business rule has been reconstructed, although part of the reconstruction is based on weak semantic similarity or an unverified language-model hypothesis. In that case, the model does not reduce uncertainty. It hides it.

Therefore, the research problem is the transition from technical reconstruction to evidence-aware business logic recovery. This requires determining how structural dependencies, configuration links, semantic associations, graph context, and model-generated interpretations can be combined while preserving their type, source, and degree of confidence. The article addresses this problem by comparing classical, semantic, language-model-based, and graph-based approaches and by identifying the need for a traceable semantic-graph representation of business logic.

4. Research Results

4.1. Classical Architecture Recovery Methods and Their Limitations

Classical architecture recovery methods provide the structural starting point for understanding an existing software system. Their purpose is to reconstruct the implemented architecture from available artifacts when original design descriptions are incomplete, outdated, or absent [3]. In software re-engineering, such reconstruction supports the transition from low-level implementation details to a more coherent view of components, dependencies, and architectural organization [4].

This view is grounded in established concepts of software architecture. Perry and Wolf describe architecture through elements, form, and rationale: the system parts, the relationships that organize them, and the reasoning behind design decisions [11]. Shaw and Garlan emphasize high-level organization, architectural styles, and principles of component composition [12]. These perspectives show that architecture recovery is not only about detecting technical links. It is about reconstructing the organizing structure of a system.

In practice, classical recovery usually relies on static software analysis. Abstract syntax trees, control-flow graphs, call graphs, dependency graphs, data-flow graphs, and module dependency graphs help identify program elements and the relationships between them [3]. These representations can show which components exist, which methods invoke one another, how data move through operations, and which modules form the main dependency structures. This is a major strength. The recovered relationships are usually traceable to concrete implementation elements: files, methods, statements, variables, signatures, imports, or configuration entries.

Traceability makes classical recovery valuable for maintenance. A developer can inspect a recovered dependency and verify where it comes from. A method call can be linked to a line of code. A module dependency can be connected to an import or a build configuration. A data-flow relation can be checked against variables and assignments. The evidence is technical and concrete.

However, structural evidence is not the same as business understanding. A call graph may show that one method invokes another, but it does not explain why this interaction exists in the domain. A dependency graph may reveal that a service uses a repository, but it does not show whether the corresponding logic implements pricing, authorization, validation, order processing, or another business function. The structure is visible. The business meaning is not.

This limitation is central for business logic recovery. In program comprehension, developers do not only identify software elements; they build a mental model of how the system behaves and why particular elements matter [1]. Classical recovery contributes to this model by exposing technical organization. Yet it does not determine the domain role of recovered relationships. A technically correct dependency can still be semantically secondary from the perspective of business behavior.

Code property graphs partially extend the expressive power of structural analysis. A code property graph combines syntactic structure, control flow, and data-flow information within a single graph representation [5]. This allows a richer analysis of how values move through a program, which functions participate in an operation, and which conditions influence a result. For security analysis and technical program understanding, this is a strong representation. Still, it remains a technical model. A subgraph can show the path of data through validation logic, but it does not automatically state whether this path corresponds to a credit-limit rule, a discount policy, or an access restriction.

The limitations become stronger in polyglot enterprise systems. Many meaningful relationships are not expressed as direct calls. They may be introduced through dependency injection, inversion

of control, aspect-oriented programming, reflection, runtime binding, annotations, configuration files, or framework conventions. Static analysis can recover what is explicitly represented. It is less reliable when interaction is mediated by infrastructure, configuration, or runtime mechanisms.

Another problem is noise. Enterprise systems contain many technically important but business-secondary relationships: logging, caching, serialization, transaction handling, middleware, adapters, and framework utilities. These elements can dominate structural graphs because they are widely connected. Yet high connectivity does not necessarily mean high domain relevance. Without semantic filtering, the recovered architecture may become technically accurate but analytically overloaded.

Therefore, classical architecture recovery should be treated as a necessary foundation, not as a complete solution. It provides verifiable structural evidence and helps reconstruct the implemented organization of a system. But business logic recovery requires another step: interpretation of what these relationships mean in a domain context. This step cannot be derived from structure alone.

4.2. Semantic Representations of Code and Program Artifacts

Each author must be defined separately for accurate metadata identification. Multiple authors may share one affiliation. Authors' names should not be abbreviated; use full first names wherever possible. Include authors' e-mail addresses whenever possible.

Classical architecture recovery exposes technical structure, but it does not capture all forms of meaning that are relevant for business logic recovery. Source code contains more than calls, branches, and dependencies. It also contains identifiers, naming conventions, comments, recurring implementation patterns, data structures, API descriptions, and domain-specific vocabulary. These elements can provide semantic signals that are not fully represented in structural graphs.

Machine learning for code treats software as a structured data source in which textual, syntactic, and contextual features can be learned and compared [6]. Deep learning approaches have extended this direction to tasks such as code search, classification, summarization, generation, and program comprehension [13]. For business logic recovery, their main value is not that they prove the meaning of a fragment. They help identify candidate relationships that structural analysis may miss.

A first important group of methods connects programming languages with natural language. This is relevant because business meaning is often expressed indirectly: in method names, class names, comments, API descriptions, error messages, test names, and documentation fragments. CodeBERT showed that pre-trained models can jointly represent code and natural language, supporting tasks such as code search and code-description matching [14]. CodeT5 further emphasized the semantic role of identifiers in code understanding and generation [16]. This matters in enterprise systems, where names of services, DTOs, enum values, database fields, and configuration keys often carry domain information.

These models can make business-related fragments easier to find. A developer may search for "customer eligibility", "discount calculation", or "payment verification" even when the exact wording in the implementation differs. The result is not a recovered business rule. It is a lead.

A second group of methods enriches code representation with structural or behavioral signals. Pure textual similarity is fragile: two fragments may use similar words but implement different behavior, while the same business rule may be expressed through different syntactic forms. GraphCodeBERT addresses this problem by incorporating data-flow information into pre-trained code representations [15]. Tree-based neural models use syntactic structure as a source of program features [19]. Inst2vec represents code through intermediate program forms and is oriented toward operational aspects of program semantics [20]. These approaches reduce the dependence on surface-level tokens and names.

Still, structural enrichment does not automatically produce domain interpretation. A data-flow-aware representation can show that a value participates in validation or calculation. It does not by itself determine whether the validation concerns access rights, credit limits, tax rules, or compliance constraints. The business role remains to be inferred from additional context.

A third direction concerns unified and cross-modal representations of software artifacts. PLBART applies sequence-to-sequence pre-training to program understanding and generation tasks by combining code and natural-language information [17]. UniXcoder develops a unified cross-modal representation that supports different code-related tasks and usage modes [18]. Such models are useful for enterprise analysis because business behavior may be described in several forms at once: implementation code, API contracts, comments, tests, configuration keys, and documentation. A recovery method must be able to compare these forms without assuming that they are identical.

Semantic code search illustrates this practical value clearly. CodeSearchNet demonstrated how natural-language queries can be matched with relevant code implementations [21]. In business logic recovery, this enables a shift from identifier-based search to behavior-oriented search. Instead of searching only for a known method name, an analyst can search for a business action, rule, or scenario. This is especially useful when domain knowledge has been scattered across a large codebase or partially lost over time.

Contemporary surveys of machine learning techniques applied to source code show that effective program modeling usually depends on a combination of textual, syntactic, structural, and semantic features [22]. For enterprise systems, this conclusion is important. The unit of analysis cannot be limited to a method or class. Business-relevant evidence may appear in source files, SQL queries, configuration fragments, API specifications, test cases, logs, error messages, and documentation. Semantic representations help connect these artifacts at the level of meaning.

Their evidential status, however, is limited. Semantic proximity is not business equivalence. Two artifacts may refer to the same domain entity but belong to different business processes. Two methods may look similar because they share technical vocabulary, not because they implement the same rule. Conversely, two parts of the same business scenario may look dissimilar because they operate at different layers of the system.

Therefore, semantic representations should be treated as a layer of candidate discovery, not as a source of confirmed recovery results. They expand the search space, reveal possible relationships, and help group artifacts that deserve further inspection. But they do not remove the need for traceability. For business logic recovery, the next step is to connect semantic signals with structural evidence, configuration links, graph context, and interpretive explanations. Only then can semantic relatedness become part of a controlled recovery process.

4.3. Large Language Models for Business Logic Interpretation

Semantic representations can reveal that software artifacts may be related. They do not explain why this relation matters in a business context. This is where large language models become relevant. Their value lies not in replacing structural recovery, but in translating technical fragments into candidate domain interpretations.

The methodological basis of modern large language models is the transformer architecture, which made it possible to model long-range dependencies in sequences and became a foundation for scalable language models [23]. Later work showed that such models can perform a wide range of tasks by using patterns learned during pre-training and information provided in the prompt [24]. In software engineering, this has led to applications in code generation, explanation, documentation, testing, refactoring, bug analysis, and maintenance support [7].

For business logic recovery, however, the role of large language models should be defined more narrowly. They are not autonomous recovery tools. They are interpretive instruments. A model can summarize a method, explain the likely role of a configuration parameter, compare code with a requirement description, or suggest that several artifacts may belong to the same business scenario. These outputs can make scattered evidence readable. They do not make it verified.

This distinction is important in enterprise systems. A business operation may involve an API endpoint, service logic, validation rules, database access, configuration, and an external integration. Structural analysis can recover some of these links when they are explicitly represented. A language model can add another layer: it can infer a possible domain explanation from names, parameters,

comments, messages, and local context. For example, it may suggest that several fragments participate in access verification, payment processing, order-status transition, discount calculation, or policy enforcement. Such explanations are useful because they reduce the distance between implementation details and business vocabulary.

But interpretation is not proof. The main limitation of large language models is that their output is probabilistic and context-dependent. A model does not execute the system, inspect runtime behavior, or verify whether a generated explanation corresponds to the actual implementation. It produces a plausible interpretation based on the provided context and learned regularities. This creates the risk of hallucination, that is, the generation of statements that sound coherent but are not factually grounded [8].

The risk increases when the input is incomplete. If a model receives only an isolated method, it may overinterpret names and produce a domain explanation that ignores related calls, configuration entries, tests, or API behavior. If the input is dominated by infrastructure code, it may give excessive importance to logging, caching, transactions, adapters, or framework mechanisms. The problem is not only that the model can be wrong. The problem is that the explanation may look convincing.

For this reason, LLM-generated interpretations should be stored and used as hypotheses. They need source attribution: which artifacts were included in the prompt, which fragments supported the explanation, and which assumptions remained unverified. They also need grounding. A proposed business interpretation becomes analytically useful only when it can be compared with structural dependencies, data-flow relations, configuration links, tests, documentation, runtime traces, or expert validation.

This leads to a practical principle: LLM output should explain evidence, not replace it. In business logic recovery, a generated summary can help formulate a candidate rule, name a scenario, or connect low-level artifacts with domain terminology. Yet the recovered model must still show where the supporting evidence comes from. Without this link, the explanation remains detached from the system.

Large language models therefore occupy a specific position in the recovery process. They connect semantic signals with domain-level language and help make heterogeneous artifacts interpretable. At the same time, their outputs must remain separate from confirmed structural facts. A method call, a configuration reference, and an LLM-generated explanation may all contribute to understanding the same business scenario, but they do not carry the same evidential weight. This distinction prepares the ground for graph-based representations, where structural facts, semantic associations, and interpretive hypotheses can coexist without being collapsed into one category.

4.4. Graph-Based and GNN-Based Models

Semantic representations and large language models help identify related artifacts and formulate candidate interpretations. However, business logic recovery also requires context. A method, query, configuration entry, or API operation rarely has a stable business meaning in isolation. Its role becomes clearer through the artifacts around it.

Graph-based program representation addresses this need by modeling software as a set of entities and relationships. In such a model, software artifacts are treated not only as separate units, but as parts of a connected structure. This is important for architecture recovery and business logic recovery because business behavior is often distributed across several interacting elements. Graph-based representations make it possible to analyze code elements together with their dependencies, data flows, and interaction patterns [9].

For enterprise systems, the graph should not be limited to source-code entities. A useful representation may include implementation artifacts, data-access elements, configuration fragments, API operations, tests, and external integrations. These elements do not have the same nature. They should not be forced into a homogeneous model. A service method, an SQL query, a YAML parameter, and a REST endpoint may participate in the same business scenario, but they represent different kinds of evidence.

This is where heterogeneous graphs become especially relevant. They allow different node and edge types to remain explicit. A direct method call, a data-flow relation, a configuration binding, an API dependency, and a semantic association should be distinguishable inside the graph. Recent work on heterogeneous program graphs supports this direction by showing how program representations can preserve different types of software entities and relationships [10]. This distinction is not only technical. It determines whether the recovered model remains interpretable.

The graph is not the evidence itself. A graph can organize evidence, expose neighborhoods, and reveal paths between artifacts. It can show that a validation method is connected with an endpoint, a repository query, a configuration flag, and a test case. Such a subgraph may suggest a business rule or scenario. Still, the interpretation depends on what the edges mean and where they come from. A graph built from static dependencies has a different evidential basis than a graph enriched with semantic similarity or LLM-generated explanations.

Graph neural networks extend graph-based representation by learning from the structure of neighborhoods. Instead of describing an artifact only by its local content, a GNN can take into account the surrounding nodes and relationships. This is useful when the business role of an artifact is not obvious from its name or body. A method with a neutral name may become meaningful because it is connected to payment processing, customer status, authorization checks, or order transitions.

Different GNN architectures support this idea in different ways. Graph convolutional networks aggregate information from neighboring nodes and therefore capture local structural context [25]. Graph attention networks add a weighting mechanism that can assign different importance to different neighbors [26]. This matters in software graphs, where not every connection is equally informative. A relation to a domain repository or validation component may carry more business meaning than a relation to a logger or serialization utility.

Other approaches are useful for large and evolving systems. GraphSAGE supports inductive representation learning and can generalize to nodes that were not present during training [27]. This is relevant for enterprise systems whose modules, services, and integrations change over time. Gated graph sequence neural networks propagate information through a graph by repeatedly updating node states [28]. Such iterative propagation can be useful when business meaning emerges across several connected steps, for example from an API request to validation, data access, and integration with an external service.

In business logic recovery, graph-based and GNN-based models can support several tasks. They can help classify artifacts by probable role, detect similar subgraphs, group fragments that belong to the same scenario, or rank elements by domain relevance. Their strength is contextuality. They do not look only at an isolated artifact. They analyze its position in the system.

This strength also creates a risk. A graph model can only learn from the representation it receives. If dependency injection, runtime binding, configuration links, message queues, or external integrations are missing from the graph, the model cannot recover them reliably. If infrastructure nodes are overrepresented, the model may learn technical centrality instead of business relevance. A highly connected element is not necessarily a domain-significant one.

Therefore, graph-based and GNN-based approaches should be treated as a contextual layer, not as a complete solution. They are valuable because they can represent business logic as a distributed subgraph of heterogeneous artifacts. But their reliability depends on the completeness of the graph, the quality of node and edge typing, and the preservation of relationship origin. A direct dependency, a semantic association, and an interpretive hypothesis may coexist in one graph. They must not become the same kind of edge.

For this reason, graph modeling should be integrated with evidence control. The graph should preserve not only what is connected, but also why the connection exists, how it was detected, and how reliable it is. Only under this condition can graph-based analysis support business logic recovery rather than merely produce another layer of structural complexity.

4.5. Comparative Assessment of the Reviewed Methods

The reviewed approaches produce fundamentally different types of knowledge about a software system, including structural facts, semantic associations, contextual inferences, and interpretive hypotheses. Therefore, they should be compared not only in terms of their ability to identify relationships, but also with regard to the traceability of those relationships, their semantic interpretability, and their suitability for recovering distributed business logic. To systematize the findings of the review, six criteria were used: structural traceability, semantic interpretability, coverage of heterogeneous artifacts, consideration of system-level context, transparency of evidential status, and suitability for business logic recovery. The approaches were assessed on an ordinal scale from 0 to 3, where 0 denotes no direct support for the respective property, 1 denotes limited support, 2 denotes substantial but incomplete support, and 3 denotes strong support. These scores represent the results of a qualitative analytical comparison rather than experimentally measured effectiveness.

Table 1
Recovery methods comparing

Criterion	M1	M2	M3	M4	M5
Structural traceability	3	1	1	2	3
Semantic interpretability	1	2	3	2	3
Coverage of heterogeneous artifacts	1	2	3	3	3
Consideration of system-level context	2	2	2	3	3
Transparency of evidential status	3	1	1	2	3
Suitability for business logic recovery	1	2	2	2	3
Structural traceability	3	1	1	2	3
Semantic interpretability	1	2	3	2	3

Method codes: M1 – classical structural methods and code property graphs; M2 – semantic and transformer-based models; M3 – large language models; M4 – graph-based models and graph neural networks; M5 – hybrid semantic–graph representation.

The comparison shows that classical structural methods provide the highest degree of traceability because their outputs can be mapped to specific methods, statements, calls, and data-flow paths [3,

5]. Their main limitation is their restricted ability to explain the business meaning of the recovered dependencies. Semantic models extend the analysis by exploiting identifiers, comments, natural-language descriptions, and learned code representations, but their outputs predominantly take the form of candidate associations [6, 13]. Large language models are more effective in connecting technical artifacts with domain terminology; however, their explanations remain hypotheses until they are supported by source code, tests, configuration files, or other evidence [7, 8]. Graph-based models and GNNs capture neighborhoods, paths, and subgraphs, although their reliability ultimately depends on the completeness and typing quality of the input graph [9, 10].

The relationship between traceability and interpretability is illustrated in Figure 1. The horizontal axis represents the extent to which the origin of a result can be identified and verified against software artifacts, whereas the vertical axis reflects the ability to relate a technical implementation to a business function or rule.

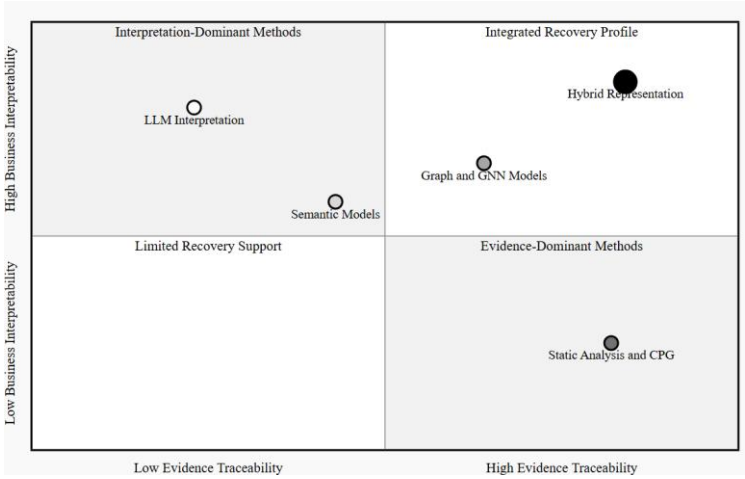

Figure 1: Positioning of business logic recovery methods according to their levels of traceability and interpretability.

Static analysis and CPGs occupy the area of high traceability and relatively low interpretability. Large language models occupy the opposite position. Semantic and graph-based approaches lie between these two extremes because they extend the analytical context but do not eliminate the need for validation. The hybrid representation is positioned in the target area because it is intended to combine structural verifiability, semantic context, and explicit control over the evidential status of relationships. The positioning shown in the figure is qualitative and does not represent the results of a quantitative experiment.

Static analysis can confirm the calls between the services and the dependency on the repository. Configuration analysis can identify the relationship with the threshold parameter. A semantic model may associate the test, the configuration key, and the calculation method with the discount concept even when no direct dependency exists between them. A large language model may formulate the corresponding candidate rule in natural language, while a graph-based representation may combine the relevant artifacts into a single subgraph. Only a typed hybrid model, however, preserves the distinction between a confirmed method call, a configuration relationship, a probable semantic association, and an interpretive hypothesis.

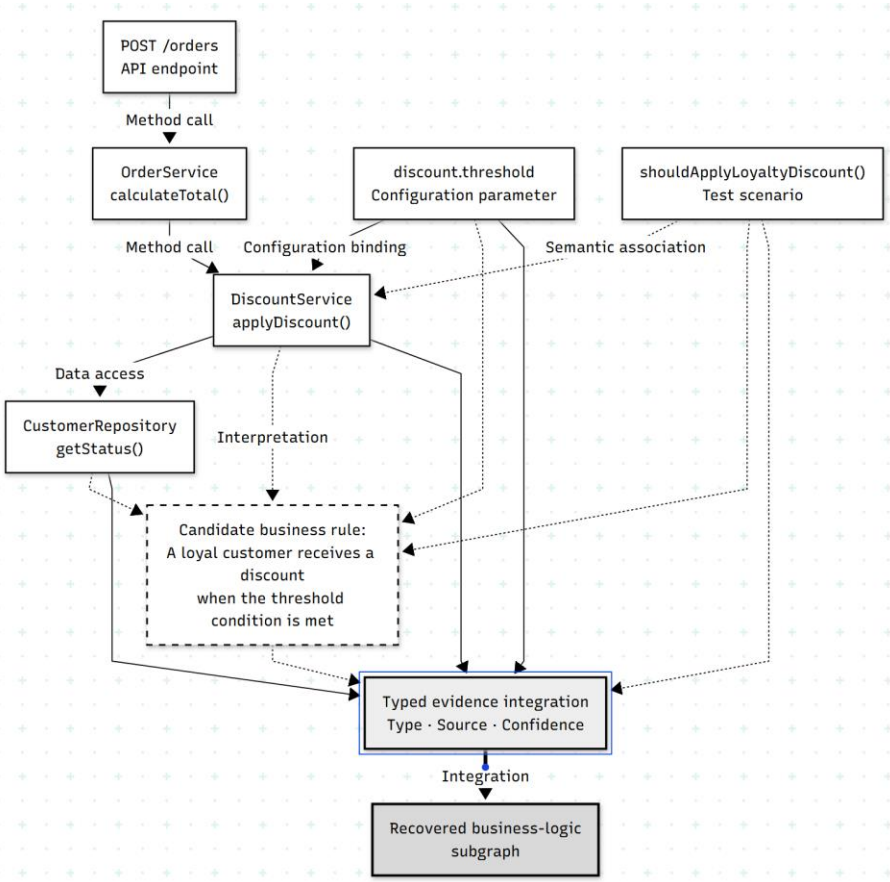

Figure 2: Illustrative recovery of a distributed business rule using different types of evidence.

In the figure 2, solid edges denote relationships that can be directly verified against software artifacts, whereas dashed edges represent semantic associations and interpretive inferences. The example is illustrative and is intended to demonstrate the analytical roles of the respective method groups rather than to report the results of an empirical evaluation.

The comparison reveals three principal patterns. Traceability and interpretive capability are distributed across different classes of methods. Structural approaches produce verifiable results but provide limited insight into their business meaning, whereas semantic and language-based models support interpretation more effectively but require additional validation. Identifying a larger number of relationships does not necessarily increase the reliability of the recovered model. Semantic search, graph-based inference, and language models broaden artifact coverage, but they also increase the proportion of probabilistic results. Model completeness and the evidential reliability of individual elements should therefore be assessed separately. The reviewed methods are functionally complementary. Static analysis provides the structural foundation, semantic models identify implicit associations, large language models generate candidate domain-level explanations, and graph-based approaches organize these results within a system-level context. The objective is therefore not to select a single universal method, but to integrate the methods in a controlled manner while preserving the type, provenance, and confidence level of each relationship.

Accordingly, the comparative assessment supports the need for an integrated representation that combines heterogeneous software artifacts without collapsing structural facts, semantic associations, and interpretive hypotheses into a single undifferentiated category.

4.6. Toward a Hybrid Semantic-Graph Representation of Business Logic

The comparative assessment shows that business logic recovery cannot rely on a single analytical layer. Structural recovery, semantic modeling, language-model interpretation, and graph-based

context each contribute different forms of knowledge. The problem is how to combine them without erasing their differences. A hybrid semantic-graph representation should address this problem directly.

In such a representation, business logic is not treated as a single method, class, module, or service. It is represented as a distributed structure of related artifacts. These artifacts may belong to different layers of the system: implementation logic, configuration, data access, API behavior, tests, documentation, and external integrations. The purpose of the graph is to make these relations visible. But visibility is not enough.

A useful hybrid representation should preserve at least three dimensions of every relationship. The first is type: whether the relationship is a method call, data-flow dependency, configuration binding, API relation, semantic association, or interpretive hypothesis. The second is source: whether it was obtained from static analysis, configuration parsing, semantic search, graph inference, LLM interpretation, tests, documentation, or expert validation. The third is confidence: whether the relationship is confirmed, probable, weak, or unverified. Without these dimensions, the graph becomes only a larger dependency map.

This distinction is essential because different links play different analytical roles. A direct call can support structural reconstruction. A configuration reference can explain framework-mediated behavior. A semantic association can indicate a candidate relation between artifacts. An LLM-generated explanation can formulate a possible domain interpretation. These signals can complement one another, but they cannot replace one another. Evidence must remain typed.

The hybrid graph should also distinguish between domain-relevant and infrastructure-dominated relationships. Enterprise systems contain many technical links created by logging, caching, transaction management, serialization, middleware, adapters, and framework utilities. These elements are important for execution, but they may obscure the structure of business behavior if treated as equally meaningful. Therefore, the representation should allow such nodes and edges to be marked, filtered, weighted, or interpreted separately.

Another requirement is traceability. Each recovered business-logic fragment should be linked back to the artifacts that support it. If a subgraph is interpreted as a discount rule, access-control scenario, payment check, or order-status transition, the model should show which methods, queries, configuration entries, tests, or documentation fragments justify this interpretation. This is especially important when AI-based components participate in the recovery process. A generated explanation is useful only when the reader can inspect the evidence behind it.

A hybrid semantic-graph representation can therefore be understood as an evidence organization framework. It does not simply collect relationships between artifacts. It records how these relationships were found, what they mean, and how reliable they are. This makes it possible to combine deterministic structural facts with weaker semantic and interpretive signals while keeping uncertainty explicit.

Such a representation may support several practical tasks. It can help locate fragments that participate in the same business scenario, detect scattered or duplicated rules, identify weakly documented domain behavior, and support modernization or refactoring decisions. It can also help developers navigate from a business-level question to the technical artifacts that implement it. The value is not only analytical. It is navigational.

At the same time, the proposed direction remains a research task rather than a completed solution. It requires formal definitions of node and edge types, confidence levels, evidence sources, and validation procedures. It also requires empirical evaluation on real enterprise systems, where business logic is heterogeneous, partially implicit, and often affected by framework behavior. Since some stages may involve AI-based components, issues of reliability, testing, and maintenance of such systems should also be considered in software engineering practice [29].

Therefore, a hybrid semantic-graph representation should be developed as a traceable and evidence-aware model of business logic. Its main contribution would not be the mere integration of structural, semantic, graph-based, and language-model-based methods. Its contribution would be

controlled integration: the ability to connect heterogeneous artifacts while preserving the difference between confirmed facts, candidate associations, and interpretive hypotheses.

5. Conclusions

This article has examined intelligent methods that can support software architecture and business logic recovery in polyglot enterprise systems. The review shows that existing approaches provide useful but incomplete views of the same problem. Classical architecture recovery reconstructs explicit structural relationships. Semantic code models reveal potentially related artifacts. Large language models formulate candidate interpretations. Graph-based and GNN-based approaches add contextual information about artifact interactions. Ehese contributions are complementary. They are not equivalent.

The main conclusion is that business logic recovery should be understood as an evidence-aware reconstruction process. The central difficulty is not only to detect relationships between software artifacts, but to determine what kind of evidence each relationship represents. A direct dependency, a configuration reference, a semantic association, and a generated explanation differ in origin, reliability, and analytical meaning. If these differences are ignored, the recovered model may look complete while concealing uncertainty.

This is especially important for enterprise systems in which business behavior is distributed across implementation logic, configuration, data access, API contracts, tests, documentation, and external integrations. In such systems, no single analytical method can reconstruct business logic in a reliable and transparent way. Structural methods need semantic interpretation. Semantic models need grounding. LLM-generated explanations need verification. Graph-based models need typed nodes, typed edges, and explicit evidence sources.

The reviewed literature therefore points toward a hybrid semantic-graph representation of business logic. Such a representation should combine structural facts, semantic associations, configuration links, graph context, and interpretive hypotheses while preserving their different evidential status. Its value lies not in aggregating more relationships, but in organizing them with traceability, relationship typing, source attribution, and confidence levels.

Further research should focus on the formalization and empirical validation of this representation in real enterprise systems. This includes defining artifact types, edge types, evidence categories, confidence models, and validation procedures. Particular attention should be paid to systems where business logic is scattered, partially implicit, and mediated by frameworks or runtime mechanisms. Under these conditions, business logic recovery becomes more than a task of reconstructing technical dependencies. It becomes a controlled process of explaining how enterprise software implements domain behavior.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT-5.5 in order to: Grammar and spelling check. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] von Mayrhauser, A., and A. M. Vans. "Program Comprehension During Software Maintenance and Evolution." *Computer* 28.8 (1995): 44–55. DOI: 10.1109/2.402076.

[2] Storey, M.-A. "Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future." *Software Quality Journal* 14 (2006): 187–208. DOI: 10.1007/s11219-006-9216-4.

[3] Ducasse, S., and D. Pollet. “Software Architecture Reconstruction: A Process-Oriented Taxonomy.” *IEEE Transactions on Software Engineering* 35.4 (2009): 573–591. DOI: 10.1109/TSE.2009.19.

[4] Byrne, E. J. “A Conceptual Foundation for Software Re-Engineering.” *Proceedings of the Conference on Software Maintenance* (1992): 226–235. DOI: 10.1109/ICSM.1992.242539.

[5] Yamaguchi, F., N. Golde, D. Arp, and K. Rieck. “Modeling and Discovering Vulnerabilities with Code Property Graphs.” *IEEE Symposium on Security and Privacy* (2014): 590–604. DOI: 10.1109/SP.2014.44.

[6] Allamanis, M., E. T. Barr, P. Devanbu, and C. Sutton. “A Survey of Machine Learning for Big Code and Naturalness.” *ACM Computing Surveys* 51.4 (2018): Article 81. DOI: 10.1145/3212695.

[7] Hou, X., Y. Zhao, Y. Liu, Z. Yang, K. Wang, et al. “Large Language Models for Software Engineering: A Systematic Literature Review.” *ACM Transactions on Software Engineering and Methodology* 33.8 (2024): Article 220, 1–79. DOI: 10.1145/3695988.

[8] Ji, Z., N. Lee, R. Frieske, et al. “Survey of Hallucination in Natural Language Generation.” *ACM Computing Surveys* 55.12 (2023): Article 248. DOI: 10.1145/3571730.

[9] Allamanis, M., M. Brockschmidt, and M. Khademi. “Learning to Represent Programs with Graphs.” International Conference on Learning Representations (ICLR) (2018). DOI: 10.48550/arXiv.1711.00740.

[10] Zhang, K., W. Wang, H. Zhang, G. Li, and Z. Jin. “Learning to Represent Programs with Heterogeneous Graphs.” *Proceedings of ICPC 2022: 378–389*. DOI: 10.1145/3524610.3527905.

[11] Perry, D. E., and A. L. Wolf. “Foundations for the Study of Software Architecture.” *ACM SIGSOFT Software Engineering Notes* 17.4 (1992): 40–52. DOI: 10.1145/141874.141884.

[12] Garlan, D., and M. Shaw. “An Introduction to Software Architecture.” *Advances in Software Engineering and Knowledge Engineering* (1993). DOI: 10.1142/9789812798039_0001.

[13] Le, T. H. M., H. Chen, and M. A. Babar. “Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges.” *ACM Computing Surveys* 53.3 (2020): Article 62, 1–38. DOI: 10.1145/3383458.

[14] Feng, Z., D. Guo, D. Tang, et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” *Findings of EMNLP* (2020): 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

[15] Guo, D., S. Ren, S. Lu, et al. “GraphCodeBERT: Pre-training Code Representations with Data Flow.” International Conference on Learning Representations (ICLR) (2021). DOI: 10.48550/arXiv.2009.08366.

[16] Wang, Y., W. Wang, S. Joty, and S. C. H. Hoi. “CodeT5: Identifier-Aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.” *EMNLP* (2021): 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.685.

[17] Ahmad, W. U., S. Chakraborty, B. Ray, and K.-W. Chang. “Unified Pre-training for Program Understanding and Generation.” *NAACL-HLT* (2021): 2655–2668. DOI: 10.18653/v1/2021.naacl-main.211.

[18] Guo, D., S. Lu, N. Duan, et al. “UniXcoder: Unified Cross-Modal Pre-training for Code Representation.” *ACL* (2022): 7212–7225. DOI: 10.18653/v1/2022.acl-long.499.

[19] Mou, L., G. Li, L. Zhang, T. Wang, and Z. Jin. “Convolutional Neural Networks over Tree Structures for Programming Language Processing.” *AAAI* (2016): 1287–1293. DOI: 10.1609/aaai.v30i1.10139.

[20] Ben-Nun, T., A. S. Jakobovits, and T. Hoefler. “Neural Code Comprehension: A Learnable Representation of Code Semantics.” 2018. DOI: 10.48550/arXiv.1806.07336.

[21] Husain, H., H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.” 2019. DOI: 10.48550/arXiv.1909.09436.

[22] Sharma, T., et al. “A Survey on Machine Learning Techniques Applied to Source Code.” *Journal of Systems and Software* (2024): Article 111934. DOI: 10.1016/j.jss.2023.111934.

[23] Vaswani, A., N. Shazeer, N. Parmar, et al. “Attention Is All You Need.” 2017. DOI: 10.48550/arXiv.1706.03762.
[24] Brown, T. B., B. Mann, N. Ryder, et al. “Language Models Are Few-Shot Learners.” 2020. DOI: 10.48550/arXiv.2005.14165.
[25] Kipf, T. N., and M. Welling. “Semi-Supervised Classification with Graph Convolutional Networks.” 2016. DOI: 10.48550/arXiv.1609.02907.
[26] Veličković, P., G. Cucurull, A. Casanova, et al. “Graph Attention Networks.” 2017. DOI: 10.48550/arXiv.1710.10903.
[27] Hamilton, W. L., R. Ying, and J. Leskovec. “Inductive Representation Learning on Large Graphs.” 2017. DOI: 10.48550/arXiv.1706.02216.
[28] Li, Y., D. Tarlow, M. Brockschmidt, and R. Zemel. “Gated Graph Sequence Neural Networks.” 2015. DOI: 10.48550/arXiv.1511.05493.
[29] Martínez-Fernández, S., J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, and S. Wagner. “Software Engineering for AI-Based Systems: A Survey.” ACM Transactions on Software Engineering and Methodology 31.2 (2022): Article 37e, 1–59. DOI: 10.1145/3487043.