International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 5, Issue 4, 330-344, 2024
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V5I4P129

Original Article

Intelligent Software Architecture Recovery Using Large Language Models and Graph Neural Networks for Legacy System Modernization

Dr. Ajit Kumar Pasayat[1], Dr. Suman Majumder[2], Dr. Hrudaya Kumar Tripathy[3], Dr. Madhabananda Das[4]
[1, 2]Assistant Professor, Computer Engineering, KIIT Deemed to be University, Bhubaneswar, Odisha, India.
[3]Professor, Computer Engineering, KIIT Deemed to be University, Bhubaneswar, Odisha, India.
[4]Senior Professor, Computer Engineering, KIIT Deemed to be University, Bhubaneswar, Odisha, India.

Abstract *- Legacy software systems remain central to banking, healthcare, insurance, government, retail, telecommunications, and enterprise resource planning environments, yet many of these systems evolve for years without corresponding architectural documentation. The absence of reliable architectural knowledge increases maintenance cost, slows digital transformation, weakens software reliability, and creates substantial risk during cloud migration, microservices decomposition, regulatory remediation, and DevOps modernization. Traditional software architecture recovery techniques rely heavily on static dependencies, clustering heuristics, call graphs, package structures, and manual expert interpretation. Although these methods provide useful structural views, they often fail to capture the semantic intent, business capability alignment, hidden cross-cutting concerns, and modernization feasibility of legacy modules. This paper proposes an intelligent architecture recovery framework that combines Large Language Models (LLMs) and Graph Neural Networks (GNNs) to recover, explain, and operationalize architectural knowledge from legacy software repositories. The proposed framework constructs a heterogeneous software knowledge graph from source code, build files, configuration artifacts, database scripts, dependency metadata, API traces, commit history, and documentation fragments. LLMs are used to infer semantic roles, business capabilities, domain concepts, anti-patterns, and modernization constraints from code and natural language artifacts, while GNNs learn structural and semantic representations of software entities to identify architectural components, boundary violations, dependency risks, and migration candidates. The paper presents a research-oriented framework, algorithmic workflow, graph schema, training strategy, evaluation protocol, and modernization decision model. Unlike purely review-based studies, the contribution of this paper is a proposed recover-and-modernize architecture intelligence pipeline that transforms legacy software comprehension into actionable modernization recommendations. The framework is intended to support architects and engineering teams in microservices decomposition, cloud migration planning, technical debt prioritization, service boundary discovery, impact analysis, and reliability improvement. The study argues that hybrid LLM-GNN architecture recovery can bridge the gap between structural reverse engineering and semantically grounded modernization governance.*

Keywords *- Software Architecture Recovery, Legacy System Modernization, Large Language Models, Graph Neural Networks, Software Knowledge Graph, Reverse Engineering, Microservices Migration, Technical Debt, Software Comprehension, Architecture Reconstruction.*

1. Introduction

Legacy software systems represent accumulated business knowledge, operational rules, integration logic, and regulatory constraints that are often not fully captured in formal documentation. In many organizations, mission-critical systems have been maintained for years through incremental changes, emergency patches, vendor customizations, database workarounds, configuration overlays, and tightly coupled integrations. As a result, the implemented architecture diverges from the originally intended architecture, making it difficult for engineering teams to understand component boundaries, data ownership, control-flow dependencies, hidden coupling, and modernization risks. Architecture recovery has therefore become a fundamental problem in software engineering because it enables maintainers to reconstruct high-level design knowledge from low-level implementation artifacts.

Early design recovery research emphasized the importance of reconstructing design abstractions from source code and informal human knowledge to support maintenance and reuse [1]. Later architecture recovery and software clustering studies formalized the use of dependency information, module coupling, and subsystem patterns to recover architectural views from existing implementations. However, modern enterprise systems pose additional challenges. A legacy platform may include Java, COBOL, C#, JavaScript, SQL, Python scripts, batch jobs, REST APIs, message queues, XML configuration, YAML

pipelines, infrastructure-as-code, and vendor-specific rule engines. In such environments, architecture is not located in a single artifact; it is distributed across code, data, deployment, configuration, runtime behavior, and institutional memory.

Recent advances in artificial intelligence, machine learning, and decision intelligence for software lifecycle governance have created new opportunities to support architecture-centered modernization [2]. In particular, LLMs have shown strong capability in code understanding, natural language explanation, code summarization, and software maintenance support. At the same time, GNNs offer a principled way to represent software systems as graph-structured data, where classes, methods, services, APIs, tables, queues, commits, developers, and deployment units can be modeled as interconnected entities. The central thesis of this paper is that intelligent architecture recovery requires both semantic reasoning and graph representation learning. LLMs can infer what software entities mean, while GNNs can learn how those entities interact structurally.

Traditional clustering-based architecture recovery approaches often group modules based on dependency similarity, cohesion, or coupling. These approaches remain valuable, but they may misclassify components when structural proximity does not match business meaning. For example, utility modules, shared database access layers, logging libraries, authentication components, and message adapters often create dense dependency patterns that distort clustering results. Conversely, two modules may belong to the same business capability even when their direct code dependency is weak because they communicate through a database table, event topic, batch file, or external service. A hybrid LLM-GNN approach can address this limitation by combining semantic context with multi-relational structural evidence.

The proposed framework, named LLM-GNN Architecture Recovery and Modernization Intelligence, treats architecture recovery as a multi-stage representation learning and decision support problem. First, software artifacts are parsed and transformed into a typed software knowledge graph. Second, LLM-based semantic extraction identifies domain concepts, functional roles, architectural patterns, technical debt indicators, and modernization constraints. Third, GNN-based learning generates node, edge, and subgraph embeddings that capture structural coupling and semantic cohesion. Fourth, a boundary recovery module identifies candidate architecture components and service boundaries. Fifth, a modernization intelligence layer ranks extracted components according to migration readiness, risk, ownership clarity, technical debt, and business criticality.

The research motivation is practical. Enterprise modernization initiatives frequently fail or slow down because teams begin migration before understanding the actual architecture. A system may appear modular at the package level while remaining tightly coupled through shared database tables, synchronous call chains, implicit configuration dependencies, or hidden business rules. The proposed framework reduces this uncertainty by generating explainable architectural views before modernization decisions are made. It does not replace architects; rather, it augments them by generating evidence-based recommendations, traceable dependency maps, and modernization hypotheses.

The rest of the paper is organized as follows. Section II presents background and related work. Section III defines the research problem. Section IV introduces the proposed framework. Section V describes the software knowledge graph construction process. Section VI explains the LLM semantic recovery layer. Section VII presents the GNN architecture recovery layer. Section VIII discusses modernization decision intelligence. Section IX defines evaluation methodology. Section X presents technical discussion and practical relevance. Section XI addresses limitations and risks. Section XII concludes the paper and identifies future research directions.

2. Background and Related Work

Software architecture recovery is closely related to reverse engineering, software comprehension, design recovery, clustering, remodularization, and modernization planning. The earliest design recovery literature framed the problem as reconstructing higher-level abstractions from implemented systems, combining code analysis with domain knowledge and human interpretation [1]. This remains highly relevant because architecture recovery is not merely a mechanical extraction of dependency graphs; it is an interpretive process that connects implementation facts to design intent.

Clustering-based recovery methods became a major research direction because source code dependencies can be transformed into graphs or matrices that support automated grouping. The Bunch tool, for example, used search-based clustering to identify module decompositions that improve cohesion and reduce coupling [3]. Such approaches demonstrated that architecture recovery can be partly automated, but they also revealed important limitations. Clustering algorithms may generate different decompositions depending on input features, similarity metrics, and stopping criteria. They may also produce clusters that are structurally plausible but semantically weak.

Comprehension-driven clustering attempted to overcome purely metric-driven limitations by using subsystem patterns and naming information to support meaningful recovery. The ACDC algorithm is a representative example because it emphasized program comprehension rather than only mathematical optimization [4]. This direction is important for legacy modernization because migration teams require component explanations, not only dependency partitions. A cluster that cannot be explained in business or architectural terms is difficult to validate and difficult to operationalize.

Hierarchical clustering for software architecture recovery provided another important foundation by analyzing clustering choices, dependency characteristics, and architecture extraction goals [5]. However, hierarchical methods still depend heavily on the selected representation of the system. If the model includes only static dependencies, it may ignore runtime behavior. If it includes only package-level dependencies, it may overlook database coupling, event flows, and configuration dependencies. In modern systems, the architecture recovery problem is therefore multi-view and heterogeneous.

Research on AI-driven lifecycle governance and defect prediction has shown that machine learning can support software quality, automation, and project decision-making when integrated into the lifecycle rather than treated as an isolated prediction tool [6]. This idea is significant for architecture recovery because recovered architecture should not be a static diagram created once and forgotten. It should become part of continuous software governance, CI/CD risk assessment, refactoring planning, and technical debt monitoring.

LLMs introduced a new dimension to software engineering automation because they can connect code-level artifacts to natural language explanations. Transformer-based architectures established the foundation for modern language models by using attention mechanisms to learn contextual representations [7]. For software systems, this capability is useful because source code contains both formal syntax and informal meaning expressed through identifiers, comments, naming conventions, configuration labels, and documentation fragments. Architecture recovery benefits when these semantic signals are interpreted alongside structural dependencies.

Code-specific language models such as CodeBERT further demonstrated that programming language and natural language can be jointly represented for software understanding tasks [8]. This is directly relevant to architecture recovery because legacy systems often contain incomplete documentation, outdated comments, and inconsistent naming practices. A model trained to reason across code and text can help infer domain roles from mixed artifacts, such as method names, API descriptions, SQL procedures, and commit messages.

GraphCodeBERT extended code representation learning by incorporating data-flow information, showing that structural program information can improve code understanding [9]. This insight supports the central premise of the present paper: software architecture recovery cannot depend on tokens alone. Data flow, control flow, call dependencies, inheritance, API invocation, database access, event publication, and deployment relationships must also be represented. A hybrid approach can therefore combine LLM-based semantics with graph-based structure.

Graph neural networks provide a general framework for learning representations over graph-structured data [10]. In software architecture recovery, this is valuable because software systems are naturally relational. Classes call methods, methods access tables, services expose APIs, components consume events, build files reference packages, and commits modify related files. GNNs can propagate information across these relationships, allowing the model to learn architectural neighborhoods rather than isolated artifact features.

Graph convolutional networks introduced scalable neural methods for semi-supervised learning on graphs by aggregating local neighborhood information [11]. This idea maps well to architecture recovery because many software entities have no explicit architecture label. A small number of architect-validated labels can be propagated through the graph to infer candidate components, layers, bounded contexts, and risk zones. This allows architecture recovery to become a human-in-the-loop learning process.

GraphSAGE introduced inductive representation learning, allowing graph models to generalize to unseen nodes by learning aggregation functions over local neighborhoods [12]. This is especially important for software systems because repositories change continuously. New files, new APIs, new services, and new database tables appear as teams deliver features. A recovery model must adapt to evolving software graphs without requiring complete retraining for every change.

Graph attention networks introduced attention-based aggregation over graph neighborhoods [13]. In architecture recovery, attention is useful because not all dependencies have equal architectural significance. A method call to a logging library should not influence component recovery in the same way as a synchronous call to a payment processor or a direct dependency on a shared customer table. Attention mechanisms can help the model emphasize architecturally meaningful edges.

Recent enterprise software studies have also highlighted the importance of AI-enabled governance, secure microservices architecture, and predictive analytics in operational domains. For example, secure healthcare prescription processing and cloud-native microservices work show that architecture decisions must account for compliance, scalability, and deployment constraints [14]. These concerns are directly relevant to modernization because recovered architecture must support not only code comprehension but also security, privacy, operational reliability, and regulatory traceability.

Continuous integration and delivery research for banking systems demonstrates that machine learning can assist risk detection in real deployment environments [15]. This connects architecture recovery to DevOps modernization because recovered architectural views can be used to identify risky deployments, dependency-sensitive changes, and components that require additional validation. Architecture recovery should therefore be integrated into delivery pipelines rather than treated as a one-time reverse engineering exercise.

Software defect prediction studies further show that structural and historical software signals can be used to reason about quality risks [16]. When architecture recovery is linked with defect prediction, a modernization team can identify not only component boundaries but also unstable components, fault-prone modules, and areas where refactoring could reduce future maintenance cost. This provides a quality-aware modernization perspective.

Graph-based modeling of service dependencies has also been applied to failure propagation in distributed systems [17]. This is highly aligned with the proposed approach because legacy modernization often transforms monolithic or tightly coupled systems into distributed services. If service boundaries are selected without understanding dependency propagation, modernization can create new reliability risks rather than reduce old ones.

The literature therefore supports three major observations. First, architecture recovery has a strong foundation in reverse engineering and clustering, but purely structural approaches are insufficient for semantically complex enterprise systems. Second, LLMs can improve semantic interpretation of code and documentation, but they require grounding to avoid hallucinated architectural explanations. Third, GNNs can learn from software dependency graphs, but they require meaningful node and edge features to recover architecture in a way that aligns with business capabilities. The proposed framework is designed around these observations.

3. Problem Statement and Research Objectives

The central problem addressed in this paper is the automated recovery of semantically meaningful, structurally valid, and modernization-ready architecture views from legacy software systems. The problem is difficult because architecture is rarely represented explicitly in legacy repositories. Instead, architectural knowledge is distributed across source code, build scripts, database schemas, configuration files, API contracts, deployment descriptors, logs, ticket histories, and expert memory. Existing recovery methods typically focus on one or two artifact types and therefore provide incomplete views.

Let a legacy software system be represented as a set of artifacts ($A = \{a_1, a_2, \ldots, a_n\}$), where each artifact may correspond to a class, method, package, API, service, table, configuration file, build dependency, message topic, or deployment unit. Let ($R = \{r_1, r_2, \ldots, r_m\}$) represent relationships among artifacts, such as calls, imports, reads, writes, publishes, consumes, inherits, deploys-with, commits-with, or depends-on. The architecture recovery task is to infer a higher-level decomposition ($C = \{c_1, c_2, \ldots, c_k\}$), where each recovered component or bounded context groups related artifacts based on structural cohesion, semantic coherence, domain meaning, and modernization feasibility.

The first research objective is to define a heterogeneous software knowledge graph that can represent both structural dependencies and semantic software concepts. Traditional dependency graphs often model only files and call relationships. In contrast, a modernization-oriented graph must include business rules, data entities, external systems, runtime endpoints, ownership signals, and deployment constraints. This richer representation enables more accurate recovery of architectural units.

The second research objective is to design an LLM-based semantic extraction layer that interprets source code, comments, identifiers, API names, SQL scripts, configuration files, and documentation fragments. The LLM layer should infer domain concepts, functional responsibilities, architectural roles, anti-patterns, and migration constraints. However, because LLMs can produce plausible but incorrect explanations, the framework must ground LLM outputs in extracted artifacts and graph evidence.

The third research objective is to develop a GNN-based recovery model that learns architecture-relevant representations of software entities. The GNN must support node classification, edge importance estimation, graph clustering, link prediction, and subgraph-level risk scoring. It must also support human feedback because enterprise architecture recovery requires validation by architects, senior engineers, and domain experts.

The fourth research objective is to connect recovered architecture to modernization decision intelligence. A recovered component is useful only if it helps answer practical questions: Can this module become a microservice? Which database tables create coupling risk? Which APIs should be grouped? Which batch jobs block migration? Which components should be refactored first? Which parts require regulatory review? This paper therefore treats architecture recovery as the first stage of modernization governance, not as a diagram-generation task.

The fifth research objective is to propose an evaluation protocol that measures both technical recovery quality and modernization usefulness. Many architecture recovery methods are evaluated only through clustering metrics. However, modernization teams require additional evidence, including explanation quality, boundary stability, migration feasibility, impact analysis accuracy, and expert acceptance. The proposed protocol includes quantitative and qualitative dimensions.

The novelty of the paper lies in integrating LLM semantic reasoning, GNN structural learning, and modernization decision ranking into a single architecture recovery pipeline. The framework is not limited to source-code clustering; it produces explainable, evidence-grounded, and decision-oriented architecture intelligence.

4. Proposed LLM-GNN Architecture Recovery Framework

The proposed framework consists of six major layers: artifact ingestion, knowledge graph construction, semantic extraction, graph representation learning, architecture recovery, and modernization intelligence. Each layer transforms raw legacy artifacts into progressively higher-level architecture knowledge.

The artifact ingestion layer collects source code repositories, build files, package manifests, database schemas, migration scripts, API specifications, configuration files, deployment manifests, commit history, issue metadata, and available documentation. It supports multiple programming languages and artifact types because legacy enterprise systems are rarely homogeneous. In a Java system, for example, the framework may parse Spring controllers, service classes, repository classes, Maven dependencies, SQL files, YAML configuration, Kubernetes manifests, Kafka topics, and REST contracts.

The knowledge graph construction layer transforms extracted facts into a heterogeneous graph. Nodes represent artifacts and concepts. Edges represent structural, semantic, historical, and operational relationships. A class node may have edges to method nodes, API endpoint nodes, table nodes, configuration nodes, and commit nodes. A table node may be connected to multiple services, revealing shared data coupling. A message topic node may connect producers and consumers, revealing event-driven dependencies. This graph becomes the central representation for downstream recovery.

The semantic extraction layer uses LLMs to generate grounded descriptions of artifacts. For each class, method, package, endpoint, table, configuration block, and documentation fragment, the model produces structured semantic tags such as business capability, technical role, data ownership, risk indicator, and modernization note. For example, a class named ClaimEligibilityProcessor may be tagged as “eligibility evaluation,” “claims processing,” “business rule execution,” and “candidate domain service.” These tags become node features in the graph.

The graph representation learning layer trains GNN models over the software knowledge graph. Node embeddings capture local and global relationships among artifacts. Edge embeddings capture dependency significance. Subgraph embeddings represent candidate architectural units. The model can be trained using semi-supervised labels from known packages, architect-validated component boundaries, domain vocabulary, issue ownership, and historical modification patterns. The GNN layer is responsible for learning architecture-aware similarity that goes beyond textual naming similarity.

The architecture recovery layer converts embeddings and graph metrics into recovered architectural views. It identifies candidate components, layers, bounded contexts, technical services, shared infrastructure modules, database-centric clusters, and integration hubs. It also flags boundary violations, cyclic dependencies, architectural erosion, and unstable coupling. The output is not a single static diagram; it is a set of ranked architectural hypotheses with evidence.

The modernization intelligence layer evaluates each recovered component according to migration readiness, coupling risk, data ownership clarity, business criticality, defect history, deployment independence, and operational constraints. This layer produces recommendations such as “extract as service,” “refactor before extraction,” “retain as shared library,” “split database ownership,” “introduce anti-corruption layer,” or “defer migration due to high coupling.” The final output supports architecture modernization planning.

A key design principle is evidence traceability. Every recovered component must be linked back to the underlying facts that justify it. If the framework proposes that a set of classes belongs to the “claims eligibility” domain, it must show supporting evidence such as class names, method summaries, API endpoints, database tables, commit patterns, and call relationships. This prevents black-box modernization recommendations and supports architect validation.

Another design principle is human-in-the-loop correction. Architects may approve, reject, merge, split, or rename recovered components. These corrections are stored as graph labels and used to improve future predictions. This is especially important because legacy systems often contain organization-specific naming conventions and domain concepts that generic models may not understand initially.

The framework also includes an architectural memory mechanism. Once a component boundary is validated, it becomes a reference architecture element. Future code changes are compared against this recovered baseline to detect drift, boundary violations, or emerging coupling. This turns architecture recovery into continuous architecture governance.

5. Software Knowledge Graph Construction

The software knowledge graph is the foundation of the proposed framework. It represents the legacy system as a typed, attributed, multi-relational graph (G = (V, E, X, T)), where (V) is the set of nodes, (E) is the set of edges, (X) represents node and edge features, and (T) represents node and edge types. Unlike simple dependency graphs, the proposed graph combines static, semantic, historical, runtime, and deployment information.

Node types include repository, package, file, class, method, function, API endpoint, database table, database column, stored procedure, message topic, queue, configuration key, build artifact, deployment unit, container image, environment, issue, commit, developer, business concept, architectural pattern, and modernization constraint. Edge types include contains, imports, calls, inherits, implements, reads, writes, exposes, consumes, publishes, configured-by, deployed-with, modified-with, owned-by, related-to, violates, depends-on, and candidate-boundary.

Static analysis extracts package dependencies, class dependencies, method calls, inheritance relationships, interface implementations, annotations, API mappings, dependency injection relationships, and build dependencies. For Java systems, tools such as abstract syntax tree parsers and bytecode analyzers can be used to recover method calls and class-level dependencies. For dynamic languages, static extraction is less complete, so the graph must incorporate additional signals such as tests, runtime traces, and configuration references.

Data dependency extraction is essential because many legacy systems are coupled through databases rather than explicit service calls. A module may not call another module directly, but both may read and write the same table. The graph therefore models database tables as first-class nodes. Read and write edges distinguish query-only dependencies from ownership-like dependencies. A table written by multiple components receives a high shared-data-coupling score, which is important for modernization planning.

Runtime dependency extraction captures API calls, message flows, queue interactions, batch schedules, and service-to-service communication. Runtime traces can reveal dependencies that static analysis misses, especially in systems using reflection, dynamic configuration, dependency injection, or external workflow engines. These runtime edges are weighted by frequency, latency, failure rate, and operational criticality when such information is available.

Historical dependency extraction uses commit history and issue metadata. Files that change together over time may belong to the same functional area even if static dependencies are weak. Conversely, files with strong static dependencies but different change histories may indicate accidental coupling. Commit co-change relationships therefore provide valuable evidence for component recovery and modernization risk estimation.

Semantic nodes are generated by the LLM layer. These include business concepts, domain labels, technical roles, and modernization constraints. For example, if several classes mention “eligibility,” “claim,” “network,” and “benefit,” the framework may create or strengthen a business concept node representing claims eligibility. Artifacts are connected to semantic nodes through evidence-weighted related-to edges.

The graph construction process must normalize identifiers because legacy systems often use inconsistent naming conventions. Abbreviations, acronyms, vendor prefixes, old module names, and inconsistent casing can reduce semantic matching quality. The framework applies tokenization, acronym expansion, lemmatization, domain dictionary mapping, and embedding-based similarity to normalize identifiers. LLMs can assist by proposing candidate expansions, but all expansions must be grounded in repository evidence.

Each node receives a feature vector composed of structural features, semantic features, historical features, and operational features. Structural features include degree, dependency type counts, fan-in, fan-out, centrality, cycle participation, and layer indicators. Semantic features include LLM-generated labels, code embeddings, documentation embeddings, and domain concept scores. Historical features include change frequency, co-change clusters, defect-linked commits, and developer ownership concentration. Operational features include runtime invocation frequency, failure rate, latency, and deployment frequency.

The resulting graph provides a unified representation for architecture recovery. It supports queries such as: Which components write to the same database tables? Which services depend on the same legacy utility? Which packages change together? Which APIs belong to the same business capability? Which modules are central integration hubs? Which candidate service boundaries would minimize coupling while preserving semantic cohesion?

6. LLM-Based Semantic Architecture Recovery

The LLM layer addresses a critical limitation of traditional architecture recovery: the difficulty of inferring meaning from structure alone. In legacy systems, business capabilities are often encoded in identifiers, comments, SQL names, API routes, configuration keys, and operational terminology. A dependency graph may show that two classes interact, but it does not explain whether the relationship represents business orchestration, data access, security enforcement, logging, transformation, validation, or integration.

The proposed semantic recovery process begins with artifact chunking. Source files are divided into meaningful units such as classes, methods, endpoint handlers, SQL procedures, configuration sections, and documentation paragraphs. Each chunk is associated with location metadata, dependency context, and surrounding artifact information. The LLM is not asked to summarize an isolated fragment without context; instead, it receives bounded evidence that includes imports, class names, method signatures, comments, call relationships, and related files.

For each artifact chunk, the LLM generates structured outputs rather than free-form prose. The output schema includes functional responsibility, domain concept, architectural role, data entities used, external dependencies, risk indicators, modernization notes, and confidence level. For instance, a method may be classified as "business validation," "eligibility decision," "database lookup," or "integration adapter." A class may be classified as "domain service," "controller," "repository," "utility," "orchestration component," or "legacy compatibility adapter."

The LLM layer also performs domain vocabulary induction. It extracts recurring business terms and maps them to candidate capabilities. This is useful when documentation is incomplete. If the system repeatedly contains terms such as "claim," "benefit," "member," "plan," "eligibility," and "coverage," the framework can infer domain clusters. These clusters are not accepted automatically; they are converted into graph features and validated through structural evidence.

Another important function is anti-pattern detection. The LLM can identify signs of god classes, transaction scripts, duplicated business rules, hard-coded configurations, mixed concerns, obsolete comments, and unclear ownership. However, the framework avoids relying on LLM judgment alone. Anti-pattern predictions are verified through measurable graph features such as high fan-in, high fan-out, long dependency chains, high co-change frequency, or multi-domain semantic overlap.

The LLM layer also supports architecture explanation generation. After the GNN identifies a candidate component, the LLM generates a human-readable explanation based only on retrieved evidence from the graph. This produces outputs such as: "This candidate component appears to implement claim eligibility evaluation because its classes expose eligibility APIs, read member benefit tables, apply plan rules, and change together in eligibility-related commits." The explanation is grounded in graph facts to reduce hallucination.

Prompt design is treated as a research concern. The framework uses constrained prompts that require evidence references, structured JSON outputs, confidence scores, and uncertainty markers. The model is instructed to distinguish between explicit evidence and inferred meaning. For example, if a class name contains "Payment" but no payment API, table, or business rule evidence exists, the LLM should mark the interpretation as uncertain. This evidence discipline is necessary for enterprise adoption.

The LLM outputs are transformed into graph features. Semantic tags become node attributes. Domain relationships become edges. Confidence scores become edge weights. Explanations are stored as metadata linked to recovery decisions. In this way, the LLM does not replace the graph; it enriches the graph with semantic signals.

This design also helps address privacy and security concerns. In regulated environments such as healthcare and banking, source code may not be permitted to leave enterprise boundaries. Therefore, the framework can be deployed with local or private code models, retrieval filtering, redaction, and access controls. AI-driven fraud detection and secure governance studies emphasize the importance of auditable decision pathways in regulated domains [18]. The same principle applies to architecture recovery: recommendations must be traceable, reviewable, and governed.

The semantic layer is especially valuable for modernization because it can identify business-aligned service candidates. Microservices should not be extracted merely from packages; they should align with business capabilities, data ownership, deployment independence, and operational accountability. LLM-based semantic recovery helps bridge technical artifacts and domain architecture.

7. GNN-Based Structural and Semantic Recovery

The GNN layer learns architecture-aware representations from the heterogeneous software knowledge graph. The objective is to identify components, boundaries, dependency risks, and modernization candidates by combining structural and

semantic evidence. While LLMs infer meaning from localized artifacts, GNNs propagate information across relationships and learn system-level patterns.

The framework supports several GNN tasks. The first task is node classification, where nodes are assigned architectural roles such as domain service, controller, repository, utility, integration adapter, shared infrastructure, configuration artifact, or database-owned entity. Labels may come from naming conventions, package structures, known framework annotations, architect validation, or manual seed examples. Semi-supervised learning is appropriate because only a small fraction of nodes may be labeled.

The second task is edge classification or edge scoring. Not all dependencies are architecturally equal. The GNN estimates whether an edge represents strong coupling, weak coupling, infrastructure dependency, data ownership dependency, runtime integration, or boundary violation. Edge scoring helps distinguish harmless utility usage from problematic cross-domain access.

The third task is graph clustering. Node embeddings are grouped into candidate architectural components using algorithms such as community detection, constrained clustering, or embedding-space clustering. The difference from traditional clustering is that embeddings incorporate many evidence types: code structure, semantic tags, runtime traces, data access, change history, and deployment metadata. This produces more modernization-relevant clusters.

The fourth task is link prediction. Link prediction estimates missing relationships, such as undocumented dependencies, likely co-change relationships, or implicit coupling between components. In legacy systems, some dependencies may not be visible through static analysis because of reflection, configuration, dynamic dispatch, or external data exchange. Link prediction can guide further inspection.

The fifth task is subgraph risk scoring. A candidate component is treated as a subgraph and evaluated for extraction readiness. The model considers internal cohesion, external coupling, database ownership, cycle participation, runtime criticality, defect history, and semantic clarity. The output is a modernization readiness score.

The GNN architecture can combine relational graph convolution, attention, and inductive sampling. Relational modeling is needed because a calls edge differs from a writes-table edge or a co-changes-with edge. Attention is useful because high-value dependencies should receive more influence than low-value dependencies. Inductive sampling enables the model to scale to large repositories and evolving systems.

The learning objective combines supervised and self-supervised losses. Supervised losses use known labels for artifact roles and architect-validated components. Self-supervised losses include masked node attribute prediction, edge type prediction, contrastive learning between related artifacts, and subgraph consistency learning. For example, two files that change together and share domain terms may be treated as positive pairs, while unrelated utilities may be treated as negative pairs.

A simplified training objective can be expressed as:

$$[\ = 1\ \{\text{node}\} + 2\ \{\text{edge}\} + 3\ \{\text{cluster}\} + 4\ \{\text{contrastive}\} + 5\ \{\text{feedback}\}\ ]$$

Where (*{node}) trains architectural role prediction, (*{edge}) trains dependency significance prediction, (*{cluster}) encourages coherent component grouping, (*{contrastive}) improves representation quality, and (_{feedback}) incorporates architect corrections.

The GNN layer also computes explainability signals. Attention weights, influential neighbors, high-weight dependency paths, and feature importance scores are extracted to support architectural explanation. For example, if a node is assigned to a "billing" component, the explanation may show that the decision was influenced by database access to billing tables, API routes containing billing terms, commit co-change with billing modules, and LLM semantic tags. Explainability is essential because architecture recovery outputs must be reviewed by humans before modernization decisions.

The framework also supports boundary violation detection. A boundary violation occurs when a component directly accesses another component's internal data, bypasses an API, depends on implementation classes rather than interfaces, or creates cyclic dependency chains. The GNN identifies such patterns by detecting cross-cluster edges with high architectural significance. This is useful for modernization because boundary violations often block microservices extraction.

The GNN layer is not intended to generate a single final architecture automatically. Instead, it generates ranked candidate decompositions with evidence. Architects can compare alternatives using metrics such as modularity, coupling reduction,

semantic cohesion, data ownership clarity, and migration cost. This design respects the reality that architecture recovery involves both automated inference and expert judgment.

8. Modernization Decision Intelligence

Recovered architecture becomes valuable when it supports modernization decisions. The proposed framework therefore includes a modernization intelligence layer that converts recovered components into actionable recommendations. This layer evaluates each component across five dimensions: structural independence, semantic coherence, data ownership, operational readiness, and business criticality.

Structural independence measures how easily a component can be separated from the legacy system. It considers external fan-out, incoming dependencies, cycle participation, shared libraries, framework coupling, configuration coupling, and dependency depth. A component with high internal cohesion and low external coupling is more suitable for extraction. A component embedded in dependency cycles requires refactoring before modernization.

Semantic coherence measures whether a recovered component represents a clear business capability or technical concern. A semantically coherent component has consistent domain vocabulary, related API endpoints, aligned database entities, and explainable responsibilities. Low semantic coherence suggests that the cluster may be an accidental grouping or a technical layer rather than a modernization unit.

Data ownership measures whether the component owns its data or shares tables with many other components. Shared database coupling is one of the most common barriers to microservices migration. A component that writes to a table also written by several unrelated modules may require database refactoring, event-driven synchronization, or anti-corruption layers before extraction.

Operational readiness measures whether the component can be deployed, tested, monitored, and scaled independently. It considers test coverage, build dependencies, runtime traces, deployment units, configuration isolation, logging, monitoring, and failure history. A component with strong architectural boundaries but weak operational readiness may be suitable for modular refactoring before service extraction.

Business criticality measures the importance of the component to enterprise operations. Criticality can be inferred from runtime usage, business documentation, issue severity, stakeholder mapping, transaction volume, and regulatory relevance. High-criticality components may require phased modernization, parallel run validation, and stronger rollback mechanisms.

The modernization intelligence layer ranks components using a multi-criteria decision model. Each component receives a score for extraction readiness, refactoring urgency, risk exposure, and expected modernization value. Recommendations are generated in categories: immediate service candidate, refactor-before-extraction candidate, data-decoupling candidate, shared-platform candidate, high-risk legacy core, and defer-with-monitoring candidate.

This layer also supports modernization planning scenarios. In a cloud migration scenario, the framework identifies components with minimal infrastructure coupling and clear deployment boundaries. In a microservices migration scenario, it identifies business-aligned components with strong data ownership. In a technical debt reduction scenario, it identifies high-coupling modules with frequent defects and unstable change patterns. In a DevOps modernization scenario, it identifies components whose dependency risks should be included in CI/CD gates.

Research on AI-driven decision intelligence for project management supports the idea that software lifecycle governance should combine predictive analytics, architecture context, and automated testing signals [19]. The present paper extends that principle to architecture recovery by using recovered architecture as a governance asset. Architecture intelligence should influence backlog prioritization, refactoring strategy, test planning, release risk analysis, and modernization sequencing.

Healthcare and pharmacy automation studies also show that modernization frameworks must handle compliance, workflow reliability, and operational continuity [20]. This is relevant because legacy modernization often occurs in regulated systems where uncontrolled refactoring can affect patient safety, financial integrity, privacy, or legal reporting. The proposed framework therefore emphasizes traceability, explainability, and human validation.

Federated learning and privacy-preserving governance studies further highlight the need to protect sensitive data while enabling intelligent analysis [21]. Legacy architecture recovery may involve proprietary code, regulated data schemas, and confidential business rules. The framework can support private deployment, artifact redaction, access control, and model governance to ensure that architecture recovery does not create new information leakage risks.

The modernization intelligence layer produces three primary outputs. The first output is an architecture recovery graph containing components, dependencies, semantic labels, and risk scores. The second output is a modernization roadmap that ranks candidate refactorings and migrations. The third output is an evidence report that explains why each recommendation was produced. These outputs allow architecture recovery to become a practical modernization instrument rather than a theoretical exercise.

9. Methodology and Evaluation Protocol

This paper proposes a research methodology suitable for evaluating the LLM-GNN architecture recovery framework in enterprise or open-source legacy systems. The evaluation is designed to measure not only clustering quality but also semantic accuracy, modernization usefulness, explanation quality, and expert acceptance.

The first evaluation question is whether the framework recovers architecture components that align with expert understanding. This can be assessed using architect-labeled components as ground truth where available. Metrics include adjusted Rand index, normalized mutual information, precision, recall, F1-score, and boundary agreement. However, because architecture ground truth may be incomplete or subjective, the evaluation should include expert review rather than relying only on numerical metrics.

The second evaluation question is whether LLM semantic features improve recovery quality. This can be tested through ablation studies. A baseline model using only static dependencies is compared with a model using static and historical dependencies, a model using graph features without LLM semantics, and the full LLM-GNN model. Improvement in component coherence, boundary accuracy, and explanation acceptance indicates the value of semantic enrichment.

The third evaluation question is whether GNN learning improves over traditional clustering. Baselines may include package-based decomposition, hierarchical clustering, ACDC-style clustering, Bunch-style clustering, and embedding-free community detection. The proposed model should be evaluated against these baselines using both structural metrics and expert judgments. Prior comparative architecture recovery studies show that different techniques can produce substantially different outcomes depending on system characteristics [22].

The fourth evaluation question is whether the recovered architecture supports modernization decisions. This requires task-based evaluation. Architects and senior engineers can be asked to use framework outputs to identify candidate services, shared database risks, cyclic dependencies, boundary violations, and migration blockers. Evaluation metrics include recommendation usefulness, time-to-understanding, defect-prone component identification, and agreement with expert modernization plans.

The fifth evaluation question is whether explanations are trustworthy. Explanation quality can be evaluated through evidence completeness, correctness, conciseness, and actionability. Experts should be able to trace each recommendation back to source files, graph edges, semantic tags, runtime traces, and historical evidence. If an explanation cannot be verified, the recommendation should not be accepted.

The sixth evaluation question is whether the framework remains stable under system evolution. A modernization tool must handle new commits, new dependencies, new services, and changing business rules. Temporal evaluation can train the model on one repository snapshot and test recovery quality on later snapshots. Boundary stability and drift detection performance are important metrics.

The evaluation dataset should include systems with different architectures, languages, and modernization challenges. Suitable candidates include modular monoliths, layered enterprise systems, service-oriented systems, database-centric legacy platforms, and partially migrated microservices. For each system, artifact extraction should include code, configuration, database schemas, API contracts, build files, and version history.

The evaluation should use mixed methods. Quantitative metrics provide reproducibility. Expert review provides practical validity. Case study analysis provides modernization insight. This is necessary because architecture recovery is partly technical and partly interpretive. A recovered architecture that scores well on clustering metrics but is rejected by architects has limited practical value.

The methodology also includes risk controls. LLM outputs are stored separately from verified graph facts. Human reviewers can mark outputs as accepted, rejected, or uncertain. Sensitive artifacts can be redacted before model processing. The framework logs prompts, retrieved evidence, generated tags, model versions, and validation decisions. This governance approach supports auditability.

10. Technical Discussion

The proposed framework addresses several technical challenges in architecture recovery. The first challenge is heterogeneity. Legacy systems combine multiple languages, frameworks, databases, deployment models, and integration mechanisms. A single parser or clustering algorithm cannot capture this complexity. The software knowledge graph provides a flexible representation that can absorb heterogeneous artifacts while preserving typed relationships.

The second challenge is semantic ambiguity. Identifiers may be abbreviated, inconsistent, outdated, or misleading. A package named common may contain critical business rules. A class named Util may perform domain-specific transformations. A database table may have a cryptic name inherited from decades-old design. LLMs help interpret these artifacts, but their outputs must be validated through graph evidence.

The third challenge is architectural erosion. Over time, systems accumulate shortcuts, dependency violations, duplicated logic, and shared data access. Traditional architecture documents may describe an intended architecture that no longer matches implementation. The proposed framework recovers implemented architecture while also detecting divergence from expected patterns.

The fourth challenge is scale. Enterprise repositories may contain thousands of files and millions of edges. The GNN layer must therefore support sampling, batching, incremental updates, and graph partitioning. Inductive graph learning is useful because modernization projects are continuous, and the system graph changes as new code is added.

The fifth challenge is explainability. Architecture recovery recommendations influence costly modernization decisions. A black-box model that proposes service boundaries without evidence is unlikely to be trusted. The proposed approach uses LLM-generated explanations grounded in graph evidence, GNN attention signals, dependency paths, and modernization metrics.

The sixth challenge is integration with engineering workflows. Architecture recovery should not be a disconnected research tool. It should integrate with repository scanning, CI/CD pipelines, architecture review boards, cloud migration planning, test impact analysis, and technical debt dashboards. Research on predictive analytics and Redis-backed optimization in pharmacy inventory contexts shows that intelligent systems become useful when embedded into operational workflows rather than isolated from them [23].

The seventh challenge is governance. AI-generated architecture recommendations may influence refactoring budgets, migration priorities, and risk decisions. Therefore, the framework records decision lineage. Each recommendation has evidence, model confidence, human feedback, and revision history. This aligns with regulated enterprise needs where explainability and auditability are essential.

The eighth challenge is modernization trade-off analysis. A component may be technically extractable but low in business value. Another component may be high value but risky due to data coupling. The framework therefore separates recovery confidence from modernization priority. A recovered component can be architecturally valid but still unsuitable for immediate extraction.

The ninth challenge is continuous evolution. Architecture recovery should not occur only at the beginning of a modernization initiative. Once recovered, the architecture graph can serve as a baseline for drift detection. If a new commit introduces a dependency from the payment domain to the customer profile database without approved API access, the system can flag a potential boundary violation.

The tenth challenge is cross-domain generalization. A model trained on one enterprise system may not directly generalize to another because domain vocabulary and architecture conventions differ. The proposed framework uses human-in-the-loop learning and domain vocabulary induction to adapt to each system. This design reduces dependence on a universal architecture taxonomy.

11. Practical Enterprise Relevance

The proposed framework has direct relevance for enterprise modernization programs. Many organizations want to migrate from monoliths to microservices, move workloads to cloud platforms, introduce DevOps automation, reduce technical debt, and improve software reliability. However, modernization often begins with incomplete architectural understanding. The proposed framework provides a systematic method to recover and operationalize architecture knowledge before major transformation decisions are made.

In banking systems, architecture recovery can identify transaction processing boundaries, fraud detection dependencies, batch settlement flows, shared database tables, and compliance-sensitive modules. Machine learning risk detection in CI/CD

for banking systems demonstrates that intelligent risk evaluation can support real deployment governance [15]. Architecture recovery can extend that governance by identifying which components create release risk.

In healthcare systems, architecture recovery can identify prescription workflows, eligibility logic, claims processing modules, audit trails, privacy-sensitive data paths, and compliance-critical services. Secure prescription processing research emphasizes the importance of HIPAA-compliant microservices architecture, cloud deployment, and secure operational design [14]. The proposed framework can help determine which legacy components are ready for cloud-native modernization and which require security remediation first.

In retail systems, architecture recovery can identify order management, inventory, fulfillment, returns, pricing, promotions, and customer account boundaries. Legacy retail systems often contain hidden coupling through shared databases and batch integrations. Graph-based recovery can expose these dependencies before service extraction.

In insurance and pharmacy benefit management systems, business rules may be distributed across rule engines, SQL procedures, Java services, spreadsheets, and manual configuration. LLM semantic extraction can identify domain rules, while GNN dependency learning can show which modules are structurally connected. This is particularly useful for preventing modernization efforts from breaking hidden rule dependencies.

In government and public sector systems, modernization must preserve auditability, accessibility, security, and continuity of service. Architecture recovery can support staged transformation, risk documentation, and evidence-based procurement planning. It can also reduce dependency on a small number of long-tenured experts who hold undocumented system knowledge.

The framework also supports software quality improvement. By combining architecture recovery with defect history and change frequency, teams can identify unstable architectural hotspots. Comparative machine learning studies for defect prediction show that predictive models can support software quality assessment when trained on relevant software metrics [24]. Architecture-aware defect analysis can further help teams identify whether faults are concentrated in poorly bounded components.

The framework supports refactoring prioritization. Not every technical debt item should be addressed immediately. The modernization intelligence layer ranks refactoring opportunities according to business value, coupling risk, defect history, and migration feasibility. This helps organizations avoid expensive refactoring that produces limited modernization benefit.

The framework supports architecture governance in CI/CD. Once recovered components are validated, dependency rules can be generated automatically. For example, a rule may state that the “billing” component cannot directly write to “member profile” tables. Pull requests violating this rule can trigger review. In this way, recovered architecture becomes enforceable architecture.

The framework also supports onboarding and knowledge transfer. New developers can explore recovered components, dependency maps, semantic summaries, and modernization notes. This reduces the knowledge gap created by outdated documentation. Architecture recovery therefore becomes both a modernization tool and a software comprehension tool.

12. Threats to Validity and Limitations

The proposed framework has several limitations. The first limitation is dependence on artifact quality. If source code is incomplete, build files are missing, runtime traces are unavailable, and documentation is outdated, recovery quality may decrease. The framework can still operate on available artifacts, but confidence scores should reflect evidence gaps.

The second limitation is LLM uncertainty. LLMs can infer plausible semantic labels that may not be correct. This risk is addressed through evidence grounding, structured outputs, confidence scores, and human validation. However, LLM-generated interpretations should never be treated as authoritative without review.

The third limitation is graph extraction incompleteness. Static analysis may miss dynamic calls, reflection, dependency injection, runtime-generated queries, and external integrations. Runtime traces and configuration analysis can reduce this gap, but complete dependency recovery is difficult in complex systems.

The fourth limitation is ground truth ambiguity. Architecture is partly subjective. Different architects may choose different service boundaries depending on business strategy, team structure, deployment goals, and regulatory constraints. Therefore, evaluation should consider multiple valid decompositions rather than assuming a single perfect architecture.

The fifth limitation is scalability. Very large enterprise repositories may create graphs with millions of nodes and edges. Efficient graph storage, sampling, incremental updates, and distributed training may be necessary. Practical deployment requires engineering optimization beyond conceptual model design.

The sixth limitation is privacy and intellectual property risk. Source code and architecture information are sensitive. The framework should support private model deployment, strict access control, audit logging, artifact redaction, and governance policies. Studies on privacy-preserving fraud detection reinforce the importance of operational governance when AI is applied to sensitive enterprise environments [21].

The seventh limitation is modernization recommendation risk. A model may correctly identify a component but incorrectly estimate its migration feasibility if organizational factors are missing. Team ownership, vendor contracts, budget constraints, release windows, and regulatory deadlines may affect modernization decisions. Human architects must therefore remain responsible for final decisions.

The eighth limitation is language and framework coverage. A framework optimized for Java and Spring may not immediately generalize to COBOL, mainframe batch systems, SAP customization, or low-code platforms. Extending artifact parsers and semantic prompts for additional ecosystems is required.

The ninth limitation is evaluation cost. Expert validation is time-consuming. However, architecture recovery cannot be properly evaluated without expert judgment. A practical evaluation strategy should combine automated metrics with selective expert review of high-impact components.

13. Future Research Directions

Future research should investigate benchmark datasets for architecture recovery that include source code, dependency graphs, runtime traces, expert-labeled architecture, and modernization outcomes. Current datasets often lack the full artifact diversity required for enterprise modernization research. A shared benchmark would allow more rigorous comparison among clustering, LLM-based, GNN-based, and hybrid methods.

Another direction is multi-agent architecture recovery. Specialized agents could parse code, analyze databases, inspect runtime traces, detect security risks, summarize business rules, and evaluate modernization options. These agents would collaborate through the software knowledge graph while preserving evidence traceability.

Future work should also explore causal architecture analysis. GNNs can identify correlations among dependencies, defects, and change patterns, but modernization decisions require causal reasoning. For example, teams need to know whether breaking a dependency will reduce defects or simply move complexity elsewhere.

Another research direction is continuous architecture governance. Once a recovered architecture is validated, it can become an enforceable policy model. CI/CD pipelines can detect boundary violations, database ownership conflicts, cyclic dependencies, and unauthorized API usage. This connects architecture recovery to software delivery governance.

Future studies should examine integration with automated refactoring. Architecture recovery can identify candidate refactorings, but transformation requires safe code changes, tests, and migration patterns. LLMs may assist in generating refactoring plans, while GNNs may estimate impact. However, automated refactoring must be carefully validated.

Future research should investigate architecture recovery for AI-enabled systems. Modern software increasingly includes machine learning pipelines, feature stores, model serving endpoints, data drift monitors, and AI governance artifacts. Traditional architecture recovery does not fully capture these elements. Extending the graph schema to include data pipelines and model dependencies is necessary.

Finally, future work should explore explainable modernization economics. Architecture recovery can identify technical candidates, but executives require cost-benefit analysis. Combining recovered architecture with effort estimation, risk modeling, defect prediction, and business impact scoring could provide a stronger decision foundation.

14. Conclusion

This paper proposed an intelligent software architecture recovery framework that combines Large Language Models and Graph Neural Networks for legacy system modernization. The framework addresses the limitations of traditional architecture recovery by integrating semantic interpretation, heterogeneous graph modeling, graph representation learning, explainable component recovery, and modernization decision intelligence. LLMs provide semantic understanding of code, documentation, identifiers, configuration, and domain concepts. GNNs learn structural and semantic relationships across software knowledge

graphs. Together, they enable architecture recovery that is more aligned with business capabilities, data ownership, operational dependencies, and modernization feasibility.

The proposed framework treats architecture recovery not as a one-time diagram-generation exercise but as a continuous governance capability. It supports microservices decomposition, cloud migration, technical debt prioritization, boundary violation detection, onboarding, release risk assessment, and architecture drift monitoring. It also emphasizes evidence traceability, human-in-the-loop validation, privacy-aware deployment, and explainable recommendations.

The primary contribution of the paper is a research-oriented hybrid architecture recovery model that connects code comprehension, graph intelligence, and modernization planning. By combining semantic reasoning with graph learning, the proposed approach provides a path toward more reliable, explainable, and actionable legacy modernization. As enterprises continue to modernize critical systems, intelligent architecture recovery will become an essential capability for reducing transformation risk and preserving business continuity.

References

[1] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *Computer*, vol. 22, no. 7, pp. 36–49, July 1989, doi: 10.1109/2.30731.

[2] S. D. Sivva, R. R. Thalakanti, S. S. G. Bandari, and S. D. R. Yettapu, "AI-Driven Decision Intelligence for Agile Software Lifecycle Governance: An Architecture-Centered Framework Integrating Machine Learning Defect Prediction and Automated Testing," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 4, pp. 167–172, 2023, doi: 10.63282/3050-9246.IJETCSIT-V4I4P118.

[3] B. S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, Mar. 2006, doi: 10.1109/TSE.2006.31.

[4] V. Tzerpos and R. C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering," in *Proceedings of the Seventh Working Conference on Reverse Engineering*, 2000, pp. 258–267, doi: 10.5555/832307.837118.

[5] O. Maqbool and H. A. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, Nov. 2007, doi: 10.1109/TSE.2007.70732.

[6] Gunda SK, Yettapu SDR, Bodakunti S, Bikki SB, "Decision Intelligence Methodology for AI-Driven Agile Software Lifecycle Governance and Architecture-Centered Project Management," 2023 Mar. 30, vol. 4, no. 1, pp. 102–108, doi: 10.63282/3050-9262.IJAIDSML-V4I1P112.

[7] A. Vaswani et al., "Attention Is All You Need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017, pp. 5998–6008.

[8] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547, doi: 10.18653/v1/2020.findings-emnlp.139.

[9] D. Guo et al., "GraphCodeBERT: Pre-Training Code Representations with Data Flow," in *Proceedings of the International Conference on Learning Representations*, 2021.

[10] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, Jan. 2009, doi: 10.1109/TNN.2008.2005605.

[11] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *Proceedings of the International Conference on Learning Representations*, 2017.

[12] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Advances in Neural Information Processing Systems*, vol. 30, 2017, pp. 1024–1034.

[13] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," in *Proceedings of the International Conference on Learning Representations*, 2018.

[14] Gudi, S. R., "Design and Evaluation of Secure Microservices Architecture for HIPAA-Compliant Prescription Processing on AWS and OpenShift," *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 2, pp. 144–149, 2024, doi: 10.63282/3050-9262.IJAIDSML-V5I2P116.

[15] Thalakanti, R. R., and S. S. Goud Bandari, "Intelligent Continuous Integration and Delivery for Banking Systems using Machine Learning Driven Risk Detection with Real World Deployment Evaluation," *International Journal of AI, BigData, Computational and Management Studies*, vol. 5, no. 4, pp. 168–175, 2024, doi: 10.63282/3050-9416.IJAIBDCMS-V5I4P118.

[16] S. K. Gunda, "Comparative Analysis of Machine Learning Models for Software Defect Prediction," in *2024 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS)*, Chennai, India, 2024, pp. 1–6, doi: 10.1109/ICPECTS62210.2024.10780167.

[17] Mutyam, N., "Graph-based modeling of service dependencies for predicting failure propagation in distributed systems," *International Journal of Multidisciplinary Evolutionary Research*, vol. 5, no. 1, pp. 113–116, 2024, doi: 10.54660/IJMER.2024.5.1.113-116.

[18] Bandari, S. S. G., S. D. Sivva, and R. R. Thalakanti, "Regulatory Grade Fraud Detection using Explainable Artificial Intelligence with Auditable Decision Pathways and Empirical Validation on Banking Data," *International Journal of*

Artificial Intelligence, Data Science, and Machine Learning, vol. 5, no. 3, pp. 139–147, 2024, doi: 10.63282/3050-9262.IJAIDSML-V5I3P115.

[19] Gunda, S. K. G., "The Future of Software Development and the Expanding Role of ML Models," *International Journal of Emerging Research in Engineering and Technology*, vol. 4, no. 2, pp. 126–129, 2023, doi: 10.63282/3050-922X.IJERET-V4I2P113.

[20] Gudi, S. R., "AI-Driven Fax-to-Digital Prescription Automation: A Cloud-Native Framework Using OCR, Machine Learning, and Microservices for Pharmacy Operations," *International Journal of Emerging Research in Engineering and Technology*, vol. 5, no. 1, pp. 111–116, 2024, doi: 10.63282/3050-922X.IJERET-V5I1P113.

[21] Thalakanti, R. R., S. S. Goud Bandari, and S. D. Sivva, "Federated Learning for Privacy Preserving Fraud Detection across Financial Institutions: Architecture Protocols and Operational Governance," *International Journal of Emerging Research in Engineering and Technology*, vol. 5, no. 2, pp. 108–114, 2024, doi: 10.63282/3050-922X.IJERET-V5I2P111.

[22] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing Software Architecture Recovery Techniques Using Accurate Dependencies," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 69–78, doi: 10.1109/ICSE.2015.22.

[23] Gudi, S. R., "Leveraging Predictive Analytics and Redis-Backed Caching to Optimize Specialty Medication Fulfillment and Pharmacy Inventory Management," *International Journal of AI, BigData, Computational and Management Studies*, vol. 5, no. 3, pp. 155–160, 2024, doi: 10.63282/3050-9416.IJAIBDCMS-V5I3P116.

[24] S. K. Gunda, "Fault Prediction Unveiled: Analyzing the Effectiveness of Random Forest, Logistic Regression, and KNeighbors," in *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, Erode, India, 2024, pp. 107–113, doi: 10.1109/ICSSAS64001.2024.10760620.

[25] N. Anquetil and T. C. Lethbridge, "Experiments with Clustering as a Software Remodularization Method," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, 1999, pp. 235–255, doi: 10.5555/832306.837051.

[26] Gudi, S. R., "Enhancing Reliability in Java Enterprise Systems through Comparative Analysis of Automated Testing Frameworks," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 2, pp. 151–160, 2023, doi: 10.63282/3050-9246.IJETCSIT-V4I2P115.

[27] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, Jan. 2021, doi: 10.1109/TNNLS.2020.2978386.

[28] Sivva, S. D., "An end-to-end AI-based systems engineering paradigm for lifecycle governance, predictive quality assurance, automation economics, and cybersecurity intelligence," *Journal of Frontiers in Multidisciplinary Research*, vol. 4, no. 1, pp. 600–604, 2023, doi: 10.54660/.JFMR.2023.4.1.600-604.

[29] T. B. Brown et al., "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[30] Balerao, M., "A converged artificial intelligence architecture for innovation, software lifecycle optimization, and cybersecurity risk mitigation," *International Journal of Multidisciplinary Futuristic Development*, vol. 4, no. 1, pp. 117–120, 2023, doi: 10.54660/IJMFD.2023.4.1.117-120.

[31] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[32] Manga I, Sivva SD, Manga VK, "The Adaptive Intelligence in Cloud Systems: A Unified Architecture for AI Enhanced Observability and Automated Root Cause Analysis," 2024 Mar. 30, vol. 5, no. 1, pp. 160–166, available at: https://ijaidsml.org/index.php/ijaidsml/article/view/366.

[33] S. D. Sivva, R. R. Thalakanti, S. S. G. Bandari, and S. D. R. Yettapu, "AI-Driven Decision Intelligence for Agile Software Lifecycle Governance: An Architecture-Centered Framework Integrating Machine Learning Defect Prediction and Automated Testing," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 4, pp. 167–172, 2023, doi: 10.63282/3050-9246.IJETCSIT-V4I4P118.

[34] Bandari, S. S. G., Sivva, S. D., and Thalakanti, R. R., "Regulatory Grade Fraud Detection using Explainable Artificial Intelligence with Auditable Decision Pathways and Empirical Validation on Banking Data," *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 3, pp. 139–147, 2024, doi: 10.63282/3050-9262.IJAIDSML-V5I3P115.