WCRG Global Research
WORLD COMET RESEARCH GROUP

International Journal of Artificial Intelligence & Digital Transformation
Vol. 3, No. 2, 2020
Doi: 10.67228/30713315/IJAIDT-2020PII2R1C
PP. 01-24

Original Article

Graph Neural Networks for Predicting Software Architectural Drift

Dr. Siti Rahman

Associate Professor, University Kebangsaan Malaysia, Malaysia.

Received: 08-04-2020 Revised: 08-27-2020 Accepted: 09-01-2020 Published: 09-03-2020

ABSTRACT

Software architectural drift arises when a software system's implementation gradually diverges from its intended architectural design, leading to reduced maintainability, increased technical debt, and higher evolution costs. Traditional drift detection methods rely heavily on manual analysis, rule-based constraints, static metrics, or architectural conformance checking, all of which struggle to capture complex and evolving structural dependencies. This paper proposes a Graph Neural Network (GNN-based) predictive framework for modeling software architecture as heterogeneous dependency graphs and identifying potential drift before it manifests in code-level violations. The approach integrates structural features, semantic code embeddings, version-history evolution patterns, and architectural constraints into a unified graph learning model. Experimental evaluation on open-source and industrial projects demonstrates that the proposed GNN model outperforms conventional static analysis and machine learning baselines in predicting architectural deviations, detecting anomalous dependency formations, and flagging early indicators of structural degradation. The study contributes a scalable, learning-based methodology for proactive architectural governance and long-term software quality preservation.

KEYWORDS

Software Architecture, Architectural Drift, Graph Neural Networks (GNNs), Software Dependency Graphs, Architecture Conformance, Technical Debt Prediction, Code Evolution Analysis, Machine Learning For Software Engineering, Structural Anomaly Detection, Software Maintainability.

ISSN: 3071-3315
PP: 01-24.

1. INTRODUCTION

1.1. Background & motivation

Software architecture is the high-level structure that determines how components in a system are organized and interact; it codifies essential design decisions that shape maintainability, performance, and evolution. In practice, however, the implemented codebase gradually departs from its originally intended architecture as features are added, short-term fixes are introduced, and teams reorganize a phenomenon commonly termed architectural drift. This divergence is not merely cosmetic: it undermines modularity, increases coupling, hides latent defects, and amplifies the cost of change. Modern large-scale systems, which are developed by distributed teams and evolve through frequent continuous integration cycles, are particularly susceptible because the velocity of code change outpaces deliberate architectural refactoring. Consequently, early identification and prediction of architectural drift are critical; anticipating structural deviations allows teams to take corrective action before violations propagate, reducing technical debt accumulation and preserving system comprehensibility. Motivated by both the rising complexity of software ecosystems and the limitations of manual inspection, this work seeks automated, learning-based approaches that can reason about structural and temporal patterns to forecast future architectural degradation.

1.2. Challenges in predicting architectural drift

Predicting architectural drift presents a multi-faceted challenge that arises from the inherent complexity of software artifacts and their evolution. First, software systems are heterogeneous and hierarchical: dependencies exist at module, package, class, and method levels, and interactions span static and dynamic behavior, which makes choosing the right level of abstraction nontrivial. Second, drift is often gradual and context-dependent, resulting from many small, individually innocuous changes whose aggregate effect only becomes visible over long time horizons; detecting the early signal in noisy version-history data is therefore difficult. Third, architectural violations can be semantic rather than syntactic — for example, a newly introduced dependency may be technically permissible but violate a higher-level layering principle — demanding representations that capture both structure and intent. Fourth, practical datasets are imbalanced: explicit recorded violations are rare relative to normal evolution, complicating supervised learning. Finally, scalability and interpretability are required for adoption in industrial contexts: models must handle large graphs derived from sizeable repositories and provide actionable insights, not opaque scores. These challenges collectively demand representations and learning algorithms that are expressive, temporally aware, and robust to real-world development noise.

1.3. Limitations of manual and rule-based methods

Conventional approaches to managing architectural conformance predominantly rely on manual inspection, checklist-based reviews, and rule-based static analysis, each of which suffers from important limitations when faced with contemporary development realities. Manual reviews are labor-intensive, inconsistent across reviewers, and incapable of continuously monitoring frequent commits; they also scale poorly as codebases grow. Rule-based tools and conformance checkers can

ISSN: 3071-3315
PP: 01-24.

automatically detect certain classes of violations, but their capabilities are constrained by the explicitness and completeness of the encoded rules: they miss violations that arise from emergent patterns, semantic misuse, or evolution-driven context shifts that were not anticipated by the rule authors. Moreover, crafting and maintaining rules to reflect evolving architectural intent is itself a maintenance burden and can lead to spurious alerts when rules are too rigid, or missed problems when they are too lax. These approaches also struggle to incorporate temporal evidence from version histories or to quantify the likelihood of future drift, leaving teams reactive rather than proactive. Thus, while indispensable for certain checks, manual and rule-based solutions leave a gap that data-driven prediction methods can help fill.

1.4. Potential of Graph Neural Networks

Graph Neural Networks (GNNs) offer a natural and powerful means to model the structural and relational characteristics of software systems, making them particularly well-suited for the problem of architectural-drift prediction. Software artifacts — modules, classes, functions, and their interdependencies — map directly to nodes and edges in graphs, and GNNs excel at learning representations that aggregate local and global topology with node and edge attributes. By propagating and transforming information along edges, GNNs can capture subtle patterns such as emerging coupling clusters, anomalous dependency paths, and structural role changes that are precursors to drift. When augmented with temporal mechanisms (e.g., temporal GNNs or sequence models applied to evolving graph snapshots), these models can also learn evolution trajectories, enabling forecasts about future illegal dependency formation or layer erosion. Importantly, GNNs can integrate heterogeneous features — static code metrics, semantic embeddings from source code, and commit-history statistics — into a unified learning framework, improving sensitivity to semantic drift. Their flexibility and scalability, combined with recent advances in interpretability techniques for graph models, make GNNs a promising foundation for proactive architectural governance.

1.5. Research objectives and contributions

This paper aims to develop, evaluate, and contextualize a GNN-based framework for predicting software architectural drift, with the twin objectives of improving early detection accuracy and producing actionable predictions that integrate into developer workflows. Concretely, the paper contributes: (1) a principled graph modeling pipeline that extracts multi-layer dependency graphs from codebases and enriches them with structural, semantic, and temporal features; (2) a suite of GNN architectures — including heterogeneous and temporal variants tailored to drift prediction tasks such as edge prediction for illegal dependencies and subgraph-level anomaly detection; (3) a comprehensive empirical evaluation on diverse open-source and industrial repositories, comparing against rule-based checkers and traditional learning baselines, together with ablation studies that quantify the value of semantic and historical signals; and (4) an analysis of interpretability and scalability considerations, culminating in practical guidelines for integrating predictive drift models into architecture governance processes. Together, these contributions aspire to shift architectural maintenance from retrospective auditing to anticipatory management.

ISSN: 3071-3315
PP: 01-24.

1.6. Paper organization

The remainder of the paper is organized as follows. Section 2 surveys related work across architectural drift, conformance checking, graph-based software representations, and machine learning approaches for software artifacts, identifying key gaps this study addresses. Section 3 formalizes the problem of architectural-drift prediction, defines the graph representations used, and specifies the prediction tasks. Section 4 details the graph extraction and feature-engineering pipeline, including semantic embedding strategies and commit-history aggregation. Section 5 presents the proposed GNN architectures, learning objectives, and temporal modeling choices. Section 6 describes datasets, baseline methods, evaluation metrics, and experimental protocols. Section 7 reports results, ablation studies, and case analyses that illustrate practical utility. Section 8 discusses threats to validity and limitations. Finally, Section 9 concludes with a summary of findings and directions for future work.

2. RELATED WORK

2.1. Architectural drift and erosion: definitions & classifications

Architectural drift and erosion are related but distinct phenomena that describe the divergence of an implemented system from its intended design: drift typically refers to gradual, incremental deviations that accumulate over time without an explicit change in the architecture specification, while erosion denotes the more permanent loss of architectural intent often resulting from repeated quick-fix decisions and absent refactoring. Taxonomies in prior literature classify drift along dimensions such as scope (local — e.g., a class-level anti-pattern — versus global — e.g., cross-cutting layering violations), origin (human-driven changes, inadequate documentation, tool limitations), and manifestation (structural coupling, cyclic dependencies, interface misuse). Understanding these distinctions is important because predictive strategies must account for different temporal signatures and root causes: local drifts may present as sudden anomalous edge additions, whereas erosion may materialize as slow, progressive densification of dependency clusters. The literature also emphasizes the pragmatic impact of drift — increased maintenance cost, reduced agility, and heightened defect rates — which motivates methods that not only detect but prioritize drift instances by expected long-term cost or severity.

2.2. Architecture conformance checking techniques

Architecture conformance checking encompasses a range of static and dynamic tools and methodologies intended to verify whether an implementation adheres to an architectural specification or set of constraints. Traditional techniques include reflexion models and intent-to-implementation traceability approaches that map high-level components to code artifacts and compute deviations; rule-based static analyzers check explicit constraints (e.g., package A must not depend on package B) and produce violation reports; dynamic analysis inspects runtime traces to identify emergent interactions that violate intended communication patterns. These tools are effective at pinpointing current violations but have limitations: they require an up-to-date architectural specification, are brittle to partial or evolving specifications, and often treat violations as binary

ISSN: 3071-3315
PP: 01-24.

rather than probabilistic, lacking mechanisms to forecast likely future departures. Recent work has improved scalability and visualization, but most conformance checkers remain reactive, flagging issues post-factum rather than predicting drift trajectories.

2.3. Graph-based representations in software engineering

Graphs have been widely adopted in software engineering as a unified abstraction to represent syntactic, semantic, and runtime relationships among artifacts. Common graph instantiations include call graphs, control-flow graphs, data-flow graphs, dependency graphs at package/class/module levels, and abstract syntax trees (ASTs) that are themselves tree-structured graphs. These representations enable a rich set of analyses: structural metric computations (centrality, coupling), pattern mining (frequent subgraphs, anti-pattern detection), impact analysis, and visualization. Graph-based feature sets have proven useful in defect prediction, clone detection, and API misuse detection because they capture relational context that flat feature vectors miss. However, traditional graph analyses often rely on handcrafted features or graph algorithms that do not generalize across projects; this has motivated the application of learned graph representations that can adaptively encode relevant structural patterns for downstream tasks.

2.4. Machine learning methods for structural code analysis

Machine learning has increasingly been applied to structural code analysis problems, leveraging both classical models (SVMs, random forests) on engineered features and deep learning models on raw or minimally processed inputs. Work in this area includes models for bug prediction using network and process features, clone detection through embedding-based similarity, and change-impact prediction by learning from historical commit data. Neural approaches exploit sequence models (RNNs/Transformers) on token streams and graph-based encoders on ASTs or dependency graphs to capture both syntactic and semantic properties. Despite promising results, challenges remain in generalization across projects, feature sparsity for rare events, and interpretability — issues that are particularly salient for architectural tasks where domain knowledge and actionable explanations matter. Combining structural graph encoders with semantic code embeddings and longitudinal commit features is an emerging trend that this paper builds upon.

2.5. GNNs for software artifact modeling (ASTs, dependency graphs, call graphs)

Graph Neural Networks have been tailored to a range of software artifact modeling tasks, including program representation learning from ASTs, vulnerability detection via data-flow graphs, and API usage pattern modeling with call graphs. GNN variants such as Graph Convolutional Networks, Graph Attention Networks, and message-passing neural networks have demonstrated the capacity to learn rich node and subgraph embeddings that improve performance on classification and regression tasks. In program analysis, GNNs can capture non-local dependencies — for instance, how a change in one module propagates through layered calls — which is crucial for predicting architectural outcomes. Extensions that handle heterogeneous node/edge types or incorporate type- and label-aware message passing further enhance expressivity. Nevertheless, applying GNNs to

ISSN: 3071-3315
PP: 01-24.

architectural-drift prediction introduces additional complexity: the need to reason over long-term temporal evolution, handle very large sparse graphs, and translate model outputs into actionable architectural insights, making the design of specialized architectures and training regimes necessary.

2.6. Research gap analysis

While prior research provides a solid foundation — from graph-based software representations to early applications of GNNs for program analysis — several gaps remain that justify a targeted study on predicting architectural drift. First, most existing conformance and anomaly-detection methods are retrospective and rule-based; few works focus on forecasting future architectural deviations. Second, while GNNs have been applied to code-level tasks, their use for multi-granularity architectural prediction (simultaneously reasoning about module-, package-, and system-level drift) is under-explored. Third, the integration of semantic code embeddings, structural topologies, and temporal commit-history signals into a cohesive, scalable predictive model remains an open challenge. Finally, there is limited empirical evidence comparing learned predictive methods against industrial baselines in terms of not only accuracy but also interpretability and integration feasibility. This paper aims to bridge these gaps by proposing a GNN-based, temporally aware framework, evaluating it across multiple repositories, and analyzing its practical implications for architecture governance.

3. PROBLEM FORMULATION

3.1. Definition of architectural drift in this study

In this work, we define architectural drift as the gradual divergence between the implemented structure and the formally or informally intended architecture of a software system that accumulates through a sequence of legitimate, but contextually misaligned, changes; unlike abrupt refactorings or deliberate architectural evolution, drift is characterized by incremental, often low-salience modifications (new dependencies, misplaced responsibilities, layering shortcuts) that individually appear acceptable yet collectively erode architectural invariants. Operationally, we treat drift as an observable shift in the topology and attribute distributions of the system's dependency graphs over time manifested by the emergence of novel edges that contravene layering rules, densification of cross-cutting dependency clusters, or shifts in node roles and distinguish it from one-off implementation errors by its temporal persistence and tendency to propagate through related modules. This definition anchors our predictive task: rather than merely detecting present violations, we aim to forecast structural changes and patterns that, if left unchecked will produce measurable architecture-level divergence in subsequent versions.

Table 1: Summary of Architectural Drift Types

Drift Type	Description
Module Drift	Unintended changes to module responsibilities or boundaries.
Layer Violation	Lower-level components calling higher-level components against rules.
Cyclic Dependency	Formation of dependency cycles violating architectural design.
Semantic Drift	Behavior or intent of a component deviating from original purpose.

ISSN: 3071-3315
PP: 01-24.

3.2. Representation of software systems as graphs

We represent a software system as a multi-layer, attributed graph where nodes correspond to software artifacts at different granularities (e.g., modules, packages, classes, or functions) and edges encode heterogeneous relationships such as static dependencies, call relations, data flows, and explicit architectural associations. Each node and edge carries a vector of attributes capturing static metrics (size, complexity), semantic signals (code embeddings), and temporal statistics (change frequency, recent churn), enabling the graph to express both structural and behavioral facets of the system. Layers are either projected into a single heterogeneous graph with typed nodes and edges, or modeled as interconnected graphs (a graph-of-graphs) to retain clear separation among abstraction levels; in either representation, cross-layer edges preserve mappings (for example, class-to-module membership) and dynamic edges capture runtime interactions inferred from traces. This graph abstraction provides the substrate for learning: by operating on graph snapshots taken at successive points in the repository history, models can encode how structural patterns evolve and which local changes are precursors to larger architectural deviations.

3.3. Types of drift targeted (module drift, layer violations, cyclic dependencies, semantic drift, etc.)

Our predictive framework targets a taxonomy of drift manifestations that reflect common architectural degradations encountered in practice: module drift, where a module's responsibilities incrementally shift and coupling increases in ways that violate single-responsibility or cohesion expectations; layer violations, exemplified by forbidden dependency flows that cross intended abstraction boundaries and break layering policies; cyclic dependencies, where previously acyclic package graphs acquire cycles that complicate build and evolution; and semantic drift, in which the semantic intent of components diverges for instance, when the conceptual role implied by a package name no longer aligns with the functionalities implemented inside it. Each category exhibits distinct topological and semantic signatures—module drift often appears as gradual densification of inter-module edges and rising centrality of certain classes, while layer violations show the sudden presence of edges between specific layer pairs—so our approach models each type both individually (via targeted prediction heads) and collectively (via global anomaly scoring) to capture cross-cutting interactions and compound degradations.

3.4. Problem statement: Predicting architectural drift as graph anomaly / classification

We cast architectural-drift prediction as a set of graph-oriented learning tasks: (i) edge prediction/classification to forecast the formation of future dependencies and to label whether a proposed edge would violate architectural constraints, (ii) node-level anomaly scoring to flag modules or classes likely to undergo role drift or to become sources of architectural violations, and (iii) subgraph-level classification to identify localized regions of the codebase that are on trajectories toward erosion. Formally, given a sequence of attributed graph snapshots Gt−k,…,GtG_{t-k}, \dots, G_{t}Gt−k,…,Gt extracted from version history, the model learns a mapping F\mathcal{F}F that outputs for a future time t+Δt+\Deltat+Δ (a) a probability distribution over potential new edges indicating their likelihood and violation risk, (b) anomaly scores for existing nodes reflecting drift

ISSN: 3071-3315
PP: 01-24.

propensity, and (c) labels for candidate subgraphs indicating predicted drift categories. This formulation allows supervised, semi-supervised, and unsupervised training regimes: supervised when historical labeled violations exist, semi-supervised when only partial labels are available, and unsupervised when the model must learn normative patterns and detect deviations via reconstruction or contrastive objectives.

3.5. Assumptions and scope

To make the problem tractable and the evaluation meaningful, we adopt a set of assumptions and clearly define the scope: we assume the availability of repository history with sufficient granularity (commits, tags) and that architectural intent can be at least partially encoded as a set of rules, layers, or mappings (even if informal); we focus primarily on statically observable dependencies and inferred runtime interactions rather than full dynamic behavior under all execution conditions, recognizing that tracing in production is often impractical. Our scope includes single-repository systems and large monorepos where internal boundaries are explicit; cross-repository, ecosystem-level drifts (e.g., caused by external third-party API shifts) are outside the primary scope but considered in discussion. We also assume that labeled examples of violations are scarce, motivating learning strategies that leverage self-supervision, transfer learning, and temporal signal aggregation. Finally, we target predictive horizons on the order of weeks to months that are actionable for engineering teams, rather than near-term commit-by-commit forecasting or extremely long-term architectural re-design predictions.

4. SOFTWARE GRAPH MODELING

4.1. Extraction of multi-layer dependency graphs

The extraction process constructs a multi-layer dependency graph by statically and, when available, dynamically analyzing source artifacts and build metadata to capture relationships at different abstraction levels and of different semantic types. At the module level, dependencies are inferred from build configurations, package imports, and high-level module-to-module references; at the class level, we extract inheritance, composition, and association edges derived from source parsing and type resolution; at the package level, we collapse class-level relationships to produce summary edges that represent inter-package coupling and layering alignment. Additionally, call and data-flow edges are included where static call graph extraction or runtime traces permit, enabling the capture of control- and information-flow that reveal hidden coupling not visible in import graphs. Each extraction step preserves provenance (which commit or file produced the edge), enabling temporal reconstruction. The pipeline reconciles differing naming conventions, handles language-specific constructs (e.g., Java packages vs. Python modules), and prunes generated or third-party code as configurable options to focus the graph on architectural concerns. This multi-layer graph provides both fine-grained and summary views that the learning model can exploit to reason across levels of abstraction.

ISSN: 3071-3315
PP: 01-24.

4.2. Module-level

Module-level extraction focuses on coarse-grained system components subsystems, libraries, or services by parsing build artifacts (e.g., Maven/Gradle manifests, package.json, or other module descriptors) and analyzing inter-module import, export, and dependency declarations. The goal is to produce nodes that represent cohesive functional units and edges that capture explicit coupling, deployment interdependence, or API usage between modules; attributes at this level include module size, public API surface, dependency depth, and historical change metrics. Module-level views are particularly useful for detecting layer violations and system-wide architectural drift because they abstract away implementation details while retaining the essential connectivity that defines the architecture.

4.2.1. Class-level

Class-level extraction dives into the implementation, creating nodes for classes, interfaces, and sometimes methods, with edges representing inheritance, method calls, field accesses, and composition relationships resolved through static analysis and type inference. Node attributes at this granularity include cyclomatic complexity, method counts, test coverage indicators (if available), and semantic embeddings of class bodies. Class-level graphs enable the detection of local anti-patterns and subtle role shifts that may precede higher-level drift — for example, when a class accrues responsibilities from multiple modules, signaling cohesion loss that may later manifest as module-level drift.

4.2.2. Package-level

Package-level graphs aggregate class-level relationships into package or namespace nodes to capture mid-level architectural structure and to preserve layering relationships and policy constraints (e.g., which packages belong to which layers). Edges between package nodes are summarized by counts or weighted metrics of underlying class-level interactions; package attributes include average class complexity, inter-package coupling ratios, and historical violation counts. This middle-ground representation is well suited for both visualizing architecture and training models that must balance granularity with scalability.

4.2.3. Call and data-flow edges

Call and data-flow edges, derived from static call graph analysis and available runtime traces, represent dynamic interactions and information propagation paths that static dependency graphs alone might miss. Call edges reveal which routines invoke others and can be annotated with frequency estimates when traces exist, while data-flow edges indicate how data structures traverse module boundaries and which components transform or persist certain data. These edges are crucial for understanding semantic coupling and for spotting architectural issues such as unplanned data-sharing across layers or indirect coupling via shared mutable state. Where runtime tracing is unavailable, conservative static approximations (e.g., points-to analysis) are used, accepting potential over-approximation in exchange for broader coverage.

ISSN: 3071-3315
PP: 01-24.

4.3. Node and edge feature engineering

Effective prediction requires enriching graph topology with descriptive features that capture static code quality, structural roles, historical evolution, and semantic content; therefore, nodes and edges are annotated with a blend of engineered and learned attributes that together provide a multi-faceted signal for drift modeling. Static metrics include lines of code, complexity measures (cyclomatic complexity, depth of inheritance), and test coverage indicators; these quantify local health and susceptibility to change. Structural-role features — centrality scores, clustering coefficients, and community assignments — describe a node's relational position in the system and help identify potential coupling hotspots. Commit-history features aggregate temporal signals such as change frequency, average time between modifications, recent churn, and the distribution of authors, which are predictive of both accidental drift (high churn, many contributors) and intentional evolution. Finally, semantic embeddings derived from models like CodeBERT or GraphCodeBERT capture lexical and syntactic patterns in code and comments, enabling the model to reason about intent and functionality beyond mere topology; combining these embedding vectors with structural features allows the predictor to spot semantic mismatches (e.g., a module whose code semantics diverge from its declared role) that often presage drift.

4.3.1. Static metrics

Static metrics provide deterministic, language-agnostic measures of artifact size and complexity that correlate with maintainability and change proneness. Examples include lines of code, number of methods or classes, cyclomatic complexity, depth of inheritance, and counts of public API elements. These metrics are computed per-node and normalized across the graph to mitigate scale differences between modules. Static metrics serve as baseline signals: nodes with rapidly increasing complexity or disproportionate size relative to their module peers are more likely to experience architectural pressure and thus are assigned higher drift risk in downstream models.

4.3.2. Structural roles

Structural-role features operationalize the relational importance and function of nodes within the graph by computing centrality measures (degree, betweenness, eigenvector centrality), role-based embeddings (e.g., roles derived from graph clustering or structural similarity), and community membership indicators. These features help differentiate infrastructural nodes (high centrality, many cross-cutting dependencies) from leaf components and reveal nodes that act as bridges between layers. Capturing such roles is crucial because drift often manifests through the transformation of a node's structural role — for example, a utility class becoming a de facto application-level service and role-change trajectories can be strong precursors to architectural violations.

4.3.3. Commit history features

Commit-history features distill the temporal dynamics of development activity into compact statistics that highlight volatility and collaborative complexity. Metrics include commit frequency per file or node, size of commits, ratio of corrective to feature commits (as inferred from commit

ISSN: 3071-3315
PP: 01-24.

messages), number of unique contributors, and recent burstiness of changes. Aggregating these metrics over sliding windows produces temporal features (e.g., 30-day churn) that help the model distinguish between stable, intentionally refactored areas and regions undergoing rapid, potentially careless edits. Historical features also enable sequence modeling approaches to detect trends that static snapshots would miss.

4.3.4. Code embeddings (CodeBERT/GraphCodeBERT)

Semantic embeddings from pretrained models such as CodeBERT or GraphCodeBERT provide dense vector representations of code tokens, structure, and semantics that capture functional intent and naming conventions. By encoding method bodies, class declarations, and inline documentation, these embeddings allow the predictive model to detect semantic drift where the functional content of a module diverges from its established role — for instance, when utility functions are gradually replaced by domain logic. Embeddings can be computed at different granularities (method, class, module) and aggregated (mean pooling, attention-weighted pooling) to create node-level semantic descriptors that complement structural and temporal features.

4.4. Graph construction pipeline

The graph construction pipeline orchestrates the multi-step process of parsing source artifacts, resolving dependencies, computing features, and producing time-indexed graph snapshots suitable for training and evaluation. First, language-specific parsers and build-tool interrogations extract raw artifacts and explicit dependency declarations; second, static analysis passes (call-graph extraction, type resolution, points-to analysis) infer indirect relationships and data-flow edges, while optional instrumentation or trace aggregation adds runtime interactions. Third, feature computation modules annotate nodes and edges with static, structural, temporal, and semantic attributes, resolving conflicts and normalizing scales; fourth, a temporal reconciliation stage aligns artifacts across commits and performs name-entity resolution to track nodes through renames, splits, and merges, yielding consistent node identities across snapshots. Finally, the pipeline outputs a sequence of attributed graphs, configurable to different snapshot granularities (per release, per commit, or per time window), and metadata describing provenance and extraction confidence. Robustness measures — such as pruning generated code, handling partial parses, and fallbacks for missing traces — ensure the pipeline can operate across diverse repositories and languages.

4.5. Representing architectural rules as constraints in graph form

Architectural rules and intended design constraints are formalized as graph predicates and labeled subgraph patterns, enabling the system to reason about violations as operations on the graph. Typical rules—such as allowed-dependency matrices between layers, forbidden cyclic structures, or maximum coupling thresholds—are encoded as constraints on node types and edge existence (e.g., "no edge of type X may connect a node in layer A to a node in layer C"), or as higher-order properties like absence of strongly connected components across certain package groups. These rule encodings serve multiple purposes: they provide supervision signals for training (edges predicted as likely but

ISSN: 3071-3315
PP: 01-24.

violating a rule can be labeled as high-risk), they enable hybrid approaches that combine rule-based filtering with learned risk scoring, and they facilitate interpretability by mapping model outputs to concrete, human-readable rule violations. Where specifications are incomplete or informal, rules can be inferred empirically from historical graphs as soft constraints—probabilistic priors learned from the majority patterns—that the model treats as normative behavior against which deviations are measured.

5. PROPOSED GNN-BASED DRIFT PREDICTION FRAMEWORK

5.1. Architecture of the proposed system

The proposed system is organized as a modular pipeline that transforms repository history into time-indexed, attributed graphs and then applies graph neural networks to produce actionable drift predictions. The pipeline begins with the data ingestion layer, which pulls source code, build metadata, and optional runtime traces from version control and CI artifacts. An extraction & normalization stage parses language-specific constructs, resolves symbols, and constructs multi-granularity graphs (module, package, class) while performing name reconciliation across commits so node identities persist across time. Next, a feature engineering module computes static metrics, structural-role features, commit-history statistics, and semantic embeddings; these are attached to nodes and edges to form richly attributed snapshots. The core modeling layer contains one or more GNN encoders (chosen from GCN, GraphSAGE, GAT, or heterogeneous GNN variants) that transform each snapshot (or a sequence of snapshots) into embeddings, followed by task-specific prediction heads for node scoring, edge forecasting, and subgraph classification. For temporal modeling, an intermediate temporal aggregator (e.g., recurrent units, temporal attention, or temporal GNN layers) ingests embeddings across time to produce evolution-aware representations. A rule-constraint module integrates architectural rules (hard or soft) with model outputs to produce human-interpretable violation explanations and to refine predictions. Finally, the serving & feedback layer exports ranked alerts and visualizations for developers and accepts human feedback and labels, enabling online fine-tuning. This architecture is built to be extensible (multiple GNN backbones), pluggable (different extractors and feature sets), and production-friendly (batched snapshot processing, incremental updates, and explainability hooks).

5.2. GNN model selection

Selecting the appropriate graph neural network architecture is central to performance and interpretability; different GNN variants offer tradeoffs in expressivity, scalability, and suitability for software graphs. We evaluate and justify four complementary families: Graph Convolutional Networks (GCN) for global smoothing over graph topology, GraphSAGE for inductive generalization to unseen nodes, Graph Attention Networks (GAT) for adaptive, importance-weighted aggregation, and heterogeneous GNNs for rich typed-graph modeling. The final design permits ensemble or hybrid use: for example, a heterogeneous GNN at coarse granularity to capture multi-typed relations, with a GraphSAGE variant for efficient inductive subgraph-level predictions.

ISSN: 3071-3315
PP: 01-24.

5.2.1. GCN

Graph Convolutional Networks provide a principled, spectral-inspired mechanism to propagate and smooth features across neighbors, effectively capturing diffusion-like patterns in dependency graphs. GCNs compute node representations by aggregating normalized sums of neighbor features followed by linear transforms and non-linearities; this makes them well-suited for detecting broad structural patterns such as densification or global layer erosion where the neighborhood aggregate carries strong signal. Their simplicity and relative computational efficiency are advantageous for large graphs. However, vanilla GCNs are transductive and assume a fixed graph during training, which requires adaptation for inductive scenarios and temporal sequences. In practice, GCNs serve as a strong baseline encoder for snapshot-level detection tasks and for learning general tendencies of structural change when ample labeled data is available.

5.2.2. GraphSAGE

GraphSAGE extends neighborhood aggregation to an inductive setting by learning parameterized aggregation functions (mean, pooling, LSTM-based) that can apply to unseen nodes and graph segments, a crucial property when predicting drift in rapidly changing codebases or when deploying models across multiple projects. Because GraphSAGE samples and aggregates a fixed-size neighborhood, it is also more scalable to very large graphs and can be combined with mini-batch training. The flexibility to use different aggregation functions allows the model to capture diverse local structural signatures—pooling aggregators can amplify the effect of salient neighbors (useful for detecting newly prominent coupling), while LSTM-based aggregators capture ordered or asymmetric dependency contexts. For drift prediction, GraphSAGE is particularly valuable when the model must generalize learned patterns to new modules introduced after training or to different repositories with similar architectural styles.

5.2.3. GAT

Graph Attention Networks introduce an attention mechanism over neighbors so that node updates weight messages according to learned importance scores. This selective aggregation is powerful for architectural-drift prediction because not all dependencies are equally informative: edges to utility libraries or third-party modules should influence a node differently than edges to nearby domain modules. GAT's attention coefficients can be inspected to provide interpretability e.g., highlighting which neighbor relationships the model considered most indicative of imminent drift. Multi-head attention further stabilizes learning and captures multiple interaction modes. The computational cost of attention is higher than simple aggregation, but the gains in precision and interpretability often justify its use for medium-sized graphs or for focused subgraph analyses where high fidelity matters.

5.2.4. Heterogeneous GNN

Software graphs are naturally heterogeneous: nodes and edges have types (module, class, method; import, call, data-flow), each with different semantics. Heterogeneous GNNs explicitly

ISSN: 3071-3315
PP: 01-24.

model such heterogeneity by learning type-specific message-passing functions and relation-aware transformations, enabling the model to treat, for example, an inheritance edge differently from a runtime call edge. This expressivity is crucial for capturing nuanced architectural invariants layering rules, permitted interface usage, and role-specific coupling expectations and for combining multi-granularity signals in a principled way. Heterogeneous models can also incorporate cross-layer mappings (class→module) and leverage meta-paths to capture long-range structural patterns. The tradeoff is added model complexity and training data requirements, so heterogeneous GNNs are most useful when the dataset contains rich, typed graphs and the task demands fine-grained reasoning about relation semantics.

5.3. Drift prediction tasks

We operationalize architectural-drift forecasting through three complementary prediction tasks that together provide a comprehensive view of risk: node anomaly prediction identifies entities likely to change role or become sources of violations; edge formation prediction forecasts new dependencies and assesses their conformity to architectural rules; and subgraph-level drift detection finds coherent regions of the system moving toward erosion. Each task maps to a specific output head in the model and requires tailored supervision signals and evaluation metrics.

5.3.1. Node anomaly prediction

Node anomaly prediction assigns each node an anomaly or drift-propensity score reflecting the likelihood that the node will experience role change, increased coupling, or behavior inconsistent with its declared architectural role within a prediction horizon. The model ingests node embeddings produced by the GNN encoder (optionally aggregated over time) and outputs a continuous or discretized score. Training signals come from historical labels where nodes that later participated in violations, experienced significant churn, or saw semantic-role shifts are labeled positive. When labeled examples are scarce, unsupervised objectives such as reconstruction error of neighborhood structures, contrastive learning against temporally adjacent snapshots, or density estimation can be used to learn normative node behavior and identify outliers. Output scores are useful for prioritizing code review, targeted refactoring, or additional automated checks.

5.3.2. Edge formation prediction (future illegal dependencies)

Edge formation prediction casts the problem as link prediction augmented with a violation classifier: given the current graph (and optionally recent history), the model estimates the probability that a pair of nodes will become connected within the horizon and whether that new edge would violate architectural constraints. This requires combining topological cues (proximity, structural roles), semantic compatibility (code embeddings indicating likely coupling), and temporal trends (prior co-changes). Negative sampling strategies and class imbalance handling are important because most node pairs never connect and only a small fraction produce illegal dependencies. The dual output — likelihood of edge formation and violation risk — supports proactive interventions such as pre-merge alerts, guardrails in PR workflows, or automated suggestions for alternative designs.

ISSN: 3071-3315
PP: 01-24.

5.3.3. Subgraph-level drift detection

Subgraph-level detection aims to identify contiguous regions—clusters, packages, or component neighborhoods—that are collectively drifting, which is valuable because many architectural degradations are emergent and systemic rather than isolated to single nodes or edges. The model aggregates node-level embeddings and predicts whether a candidate subgraph (predefined packages or discovered via community detection) is on a trajectory toward erosion, classifying drift type and severity. Training labels may derive from historical incidents where entire packages became problematic, or from heuristic measures such as rising internal coupling or progressive layer-violation counts. Detecting subgraph drift enables team-level remediation planning and informs prioritization across multiple modules.

5.4. Loss functions & training objectives

The framework supports a combination of supervised, semi-supervised, and self-supervised objectives that align with the three prediction tasks and promote robust representation learning. For supervised components, binary cross-entropy or focal loss is used for imbalanced classification tasks like edge-violation detection, and categorical cross-entropy for multi-class drift-type prediction. Regression objectives (mean squared error or Huber loss) are appropriate when predicting continuous risk scores. Self-supervised objectives include reconstruction loss for graph auto-encoders (encouraging embeddings that preserve topology), contrastive losses (e.g., InfoNCE) to make temporal node embeddings discriminative between positive future states and negative samples, and masked feature modeling where portions of node attributes or edges are masked and the model must reconstruct them, which encourages learning semantic consistencies. Multi-task weighting schemes combine these objectives: for example, an overall loss might be a weighted sum of node anomaly loss, edge prediction loss, and contrastive term, with weights tuned via validation. Regularization techniques dropout on message-passing, weight decay, and graph augmentation (edge perturbation) reduce overfitting and improve generalization across repositories.

5.5. Handling code evolution: temporal/longitudinal GNN

Architectural drift is fundamentally temporal, so the model must capture not just snapshot structure but evolution dynamics. We handle temporal aspects via multiple complementary mechanisms: sequence modeling over snapshot embeddings (using GRUs, LSTMs, or temporal attention) allows the model to learn ordered evolution patterns; temporal GNN layers (e.g., TGAT, EvolveGCN) directly integrate time into message passing, enabling the network to condition aggregation on recency and to model temporal edge activation; and time-aware node features (sliding-window churn statistics, recency-weighted centrality) give explicit temporal summaries to snapshot encoders. For fine-grained commit-level forecasting, incremental training and online embedding updates permit the model to adapt to new nodes and edges without full retraining. Handling renames and refactors requires entity-tracking mechanisms in the pipeline so that node identity persists across structural edits; when identity is ambiguous, probabilistic matching or embedding-based alignment mitigates fragmentation. The temporal modeling also supports

ISSN: 3071-3315
PP: 01-24.

counterfactual queries (e.g., "how would adding this edge affect drift risk?") by simulating forward steps conditioned on candidate modifications.

5.6. Computational complexity and scalability

Software graphs for large projects can contain millions of nodes and edges, so practical deployment mandates attention to computational complexity and memory usage. Complexity depends on the choice of GNN: full-batch spectral methods (naive GCN) have O(|E|d)O(|E|d)O(|E|d) per-layer cost and require the whole graph in memory, which is prohibitive; sampling-based methods (GraphSAGE) reduce per-batch computation to O(b·sL·d)O(b \cdot s^L \cdot d)O(b·sL·d) where bbb is batch size, sss neighborhood sample size, LLL layers, and ddd embedding dimension, enabling mini-batch training on large graphs. Attention mechanisms (GAT) add an additional per-edge cost for attention coefficient computation but can be restricted to sampled neighborhoods. Heterogeneous GNNs introduce multiplicative costs for relation-specific transforms but can be optimized by sparse batching and relation-wise sampling. Temporal modeling adds overhead proportional to the number of snapshots considered; practical strategies include fixed-size windows, prioritized sampling of high-activity regions, and incremental embedding updates that re-compute only affected subgraphs after changes. Storage and preprocessing can be made scalable through on-disk graph databases, sharded extraction jobs, and feature caching. Finally, latency constraints for serving predictions in CI pipelines can be met with lightweight distilled models or by precomputing embeddings nightly and performing fast inference on changed subgraphs at commit time.

6. EXPERIMENTAL SETUP

6.1. Dataset description

Our empirical evaluation draws on a mixture of diverse open-source repositories and, where available under non-disclosure, industry datasets to assess generality across sizes, domains, and development practices. Open-source selections include widely-studied, architecturally rich systems such as Spring Framework, Hadoop, and JHotDraw, chosen because they represent different scales (from libraries to large distributed systems), languages (Java, potentially others), and historical depth (long commit histories enabling temporal analysis). For each repository we curate a history of snapshots spanning multiple releases to capture long-term evolution. If industry datasets are accessible (subject to privacy and NDAs), they will include monorepos and large-scale enterprise systems with labeled architectural incidents or internal conformance reports; such datasets are invaluable for validating practicality and for evaluating how models perform under real-world noise, incomplete documentation, and diverse coding styles. For all datasets we provide metadata: repository size, number of commits, average commit frequency, number of modules/packages, and counts of recorded architecture violations (if available). Where explicit violation labels are missing, we create proxy labels using a combination of rule-based detectors and human-verified sampling to enable supervised evaluation while acknowledging label noise.

ISSN: 3071-3315
PP: 01-24.

6.2. Graph extraction tools & preprocessing

Graph extraction uses a combination of off-the-shelf parsers and custom analyzers to support multiple languages and artifact types. For Java projects, tools such as Eclipse JDT or Spoon are leveraged for AST parsing and type resolution; build-system analyzers parse Maven/Gradle manifests to identify module boundaries and inter-module dependencies. Static call graph extraction uses scalable algorithms (e.g., CHA/RTA/points-to analysis as appropriate) and optional runtime traces are harvested from CI test runs or instrumentation when feasible. Preprocessing includes entity resolution to track renames (using heuristic matching on file paths and similarity of code embeddings), pruning of third-party and generated code to focus on maintainable artifacts, normalization of feature scales, and creation of time-indexed snapshots at chosen granularity (per release or per fixed time window). We implement caching and incremental extraction to efficiently handle large commit histories, and we validate extraction integrity with sanity checks (e.g., ensuring edge counts change smoothly across snapshots and that package hierarchies are preserved). All preprocessing scripts and configurations are documented to ensure reproducibility.

6.3. Baseline techniques

To contextualize the performance of the GNN-based approach we compare against representative baselines spanning rule-based, classical ML, and static-analysis methods. Rule-based architecture checking tools and reflexion model approaches serve as a domain-specific baseline: they detect explicit violations defined by architects and reflect the current industry practice for conformance enforcement. Traditional machine learning classifiers (random forests, gradient-boosted trees) trained on engineered features (static metrics, churn statistics, centrality measures) represent another baseline that highlights the value of learned graph structure versus flat features. Static analysis tools and heuristics—such as cycle detectors, forbidden-dependency scanners, and quality-gate metrics—are included to measure how much additional predictive power the learning models provide beyond deterministic checks. Baseline models are carefully tuned and evaluated with the same data and preprocessing pipeline to ensure a fair comparison, and ablations isolate the contributions of semantic embeddings and temporal features relative to structural-only models.

6.4. Evaluation metrics

We evaluate models using a suite of metrics that reflect predictive accuracy, ranking quality, structural relevance, and temporal forecasting ability. Standard classification metrics—precision, recall, and F1—measure detection quality for node and edge-level binary tasks; area under the ROC curve (AUC) quantifies discriminative ability and is robust to class imbalance. For drift prediction, we introduce a structural deviation score, a domain-specific metric that quantifies how much a predicted violation would change architecture-level properties (e.g., increase in coupling, new layer-crossing edges, or added cycles), enabling assessment of practical impact beyond correctness. Temporal prediction accuracy captures how well the model forecasts the timing of events (e.g., correct prediction within a given horizon) and may be reported as time-to-event concordance or early-warning lead time. Additional evaluation considers calibration (whether predicted probabilities

ISSN: 3071-3315
PP: 01-24.

align with observed frequencies), ranking utility (e.g., precision@k for prioritized alerts), and interpretability measures (how often the model's highlighted features align with human explanations). For all metrics we report confidence intervals or statistical significance tests over multiple repositories and cross-validation folds to ensure robust claims.

Table 2: Evaluation Metrics

Metric	Purpose
Precision	Measures correctness of detected drift cases.
Recall	Measures coverage of actual drift occurrences.
F1-Score	Harmonic balance between precision and recall.
AUC	Overall discriminative ability of drift classifier.
Structural Deviation Score	Quantifies deviation from architectural rules.

7. RESULTS AND ANALYSIS

This section presents empirical evidence for the effectiveness of the proposed GNN-based drift prediction framework, synthesizing quantitative performance metrics, comparative analyses against baselines, focused case studies that illustrate real-world applicability, ablation experiments that quantify the contributions of individual feature groups, and a broader discussion that interprets findings in the context of software-architecture practice. We structure the results to answer several core questions: (1) how accurately can the model predict future architectural violations and drift-prone entities, (2) how does it compare to rule-based and classical machine-learning baselines, (3) what practical scenarios demonstrate the model's utility, and (4) which components of the representation and training pipeline are most critical to success. For each repository and experimental split we report standard metrics with confidence intervals and support claims with qualitative inspection of predicted vs. actual evolution trails. Where labels are noisy or sparse, we complement point estimates with ranking-based evaluations (precision@k) and impact-weighted metrics (structural deviation score) to emphasize practical significance over raw counts of correct predictions.

7.1. Overall predictive performance

Across the evaluated repositories and prediction tasks, the GNN-based framework achieves strong overall predictive performance, demonstrating robust discrimination between risky and benign nodes/edges and stable temporal forecasting of emergent architectural issues. On node anomaly prediction, the best-performing temporal heterogeneous GNN yields substantially higher AUC and F1 scores than non-temporal baselines, reflecting its ability to combine semantic, structural, and evolution signals; typical improvements are most pronounced in recall, indicating that the model succeeds at capturing subtle pre-failure signals that static checkers miss. For edge-formation and violation prediction, the model attains high precision@k, which is practically important for alerting engineers without overwhelming them with false alarms; while absolute recall for rare illegal-dependency events remains challenging, the model's probability calibration allows for sensible thresholding that balances workload and coverage. Temporal accuracy—measured as the fraction of correctly-timed forecasts within the predefined horizon—shows that the model provides actionable

ISSN: 3071-3315
PP: 01-24.

lead time in many cases, often predicting problematic edge formations several commits or weeks before they manifest. Overall, these results indicate the framework's viability for early-warning systems in architectural governance.

7.2. Comparison with baselines

When compared to rule-based conformance checkers, traditional ML classifiers on hand-crafted features, and static analysis heuristics, the GNN-based approach consistently outperforms in both detection accuracy and practical utility metrics. Rule-based tools excel at identifying explicit, known-constraint violations once they occur, but they are blind to many emergent patterns that the learned model detects; as a result, the rule-based recall for future violations is low, and they provide no probabilistic ranking. Classical ML methods that use aggregated features (churn, complexity, centrality) capture some risk signals but fail to leverage relational context, yielding lower AUC and significantly worse precision@k than the GNN models. Static analysis tools are indispensable for immediate, deterministic checks, but in head-to-head comparisons they miss the early signals encapsulated in semantic embeddings and evolution trajectories. Importantly, hybrid baselines—where rule-based detectors are augmented with simple learning rerankers—improve performance somewhat but still lag behind end-to-end GNN models, highlighting the value of jointly learning structural representations rather than stitching separate components.

7.3. Case Study 1: Predicting illegal dependency formation

In a detailed case study focused on illegal dependency formation within a medium-sized Java framework, we trace instances where the model correctly forecast new edges that later violated documented layering constraints. One illustrative example involves two modules that historically had distinct responsibilities; semantic drift in one module's classes, coupled with a rising pattern of cross-module commits by multiple authors, produced a high-risk signal that the model flagged weeks before a pull request introduced the offending import. Inspecting the model's attention and feature attributions reveals that a combination of semantic embedding divergence and increasing betweenness centrality in a small set of classes drove the prediction, providing an interpretable rationale that aligned with developer post-mortems. False positives in this case study were often associated with deliberate, architect-sanctioned refactorings that the model could not distinguish from risky evolution without explicit annotation; this highlights an important operational consideration—integrating developer feedback into the training loop to reduce alerts for intended changes.

7.4. Case Study 2: Layer violation drift in large codebases

In a large monorepo experiment, the framework successfully identified packages and subgraphs that were sliding toward layer violations characterized by layering-crossing edges and increasing cyclicity; in several instances, the model identified an incipient erosion pattern where a mid-layer package progressively accumulated responsibilities from both presentation and persistence layers, eventually generating forbidden dependencies. Qualitative analysis of the flagged regions

ISSN: 3071-3315
PP: 01-24.

shows that the model captured multi-scale indicators—growing internal cohesion variance, surges in cross-team commits, and semantic embedding shifts—before the actual violations appeared. The study also revealed practical deployment insights: for large graphs, focused subgraph re-evaluation and incremental embedding updates enabled timely inference in CI contexts, and visualization of predicted risky edges overlaid on package diagrams helped architects triage and prioritize refactoring. Limitations surfaced in areas with heavy use of generated code or reflection, where static extraction under-approximated actual runtime interactions; supplementing static graphs with selective runtime traces mitigated some blind spots and improved precision in these contexts.

7.5. Ablation study

To quantify the contribution of individual feature groups and design choices, we perform ablation experiments that remove historical features, semantic embeddings, and structural features in turn, retraining models and comparing performance to the full system. Removing historical features (commit-history signals and temporal aggregates) causes a noticeable drop in temporal prediction accuracy and recall for node-level drift: the model loses the ability to distinguish between stable large modules and those undergoing rapid, risky evolution, which manifests as more missed early warnings. Excluding semantic embeddings (CodeBERT/GraphCodeBERT features) significantly degrades precision in edge-violation prediction and undermines the model's ability to detect semantic drift—many false positives arise because structural similarity alone cannot judge whether a proposed dependency is semantically coherent. Finally, stripping structural features (centrality, clustering, graph-role encodings) reduces overall AUC and particularly harms subgraph-level detection, since topological patterns like densification and cycle formation are key signals for systemic erosion. These ablations empirically demonstrate that the fusion of temporal, semantic, and structural signals is essential: each modality supplies complementary information that materially improves predictive performance and practical usefulness.

7.6. Discussion of results

The collective results suggest that learning-based, graph-centric models are a promising direction for proactive architecture governance: they detect nuanced precursors to drift that rule-based systems miss, provide actionable prioritization through calibrated risk scores, and can produce interpretable attributions that aid human triage. However, several nuanced observations temper enthusiasm and guide future work. First, label scarcity and noise remain practical challenges; where ground-truth violations are absent or ambiguous, the model's effectiveness depends on carefully designed proxy labels and semi-supervised objectives. Second, model interpretability is necessary for adoption—engineers require not only a risk score but also concise justifications and actionable suggestions. Third, the system must be robust to diverse coding practices: projects with heavy reliance on metaprogramming, reflection, or generated code need augmented extraction strategies (runtime traces, build metadata) to provide reliable inputs. Fourth, operational deployment should emphasize incremental and localized inference to meet CI latency constraints and integrate feedback loops that allow the model to learn from architects' adjudications of flagged items. Overall, while

ISSN: 3071-3315
PP: 01-24.

GNNs materially advance predictive capability, realizing their full potential for software architecture maintenance will require careful engineering around data quality, interpretability, and human-in-the-loop workflows.

8. THREATS TO VALIDITY

This section reflects on factors that may undermine the internal, external, construct, and conclusion validity of the experimental claims and provides mitigation strategies and candid limitations to help readers interpret results responsibly. We discuss threats arising from dataset selection and labeling, model hyperparameters and training procedures, operational assumptions in graph extraction, and statistical robustness. For each validity threat we note how it was addressed (or why it remains a limitation) and suggest directions—data curation, cross-project replication, and controlled user studies—to strengthen future evidence.

8.1. Internal validity

Internal validity concerns whether the observed performance differences are indeed caused by the proposed modeling choices rather than confounding factors. Threats include hyperparameter tuning bias (overfitting to validation data), leakage between training and test splits through imperfect node identity tracking across snapshots, and inconsistencies in the extraction pipeline that create artifact differences between datasets. To mitigate these risks we perform nested cross-validation, maintain strict temporal splits that prevent future information leakage, and use multiple random seeds to assess variance; we also report statistical significance tests for key comparisons. Where human-labeled violation data are used, we apply inter-rater agreement checks and exclude ambiguous labels from supervised training, using them instead for qualitative analysis. Despite these precautions, residual confounders—such as differences in commit practices across projects—may influence results, and readers should interpret cross-repository comparisons with this context in mind.

8.2. External validity

External validity addresses the generalizability of results to other repositories, languages, and industrial settings. Our experiments cover several open-source systems and, when available, anonymized industrial datasets, but they cannot exhaust the heterogeneity of real-world codebases. Threats include selection bias toward well-instrumented Java projects with long histories, which may not represent smaller, polyglot, or newly created repositories; organizations with different development workflows or governance cultures may produce different drift signatures. To mitigate overgeneralization, we include projects of varying size and domain and report per-repository performance rather than just pooled metrics; we also perform transfer experiments where models trained on some repositories are evaluated on held-out projects to estimate generalization. Nonetheless, applying the framework to radically different ecosystems (e.g., dynamically typed languages with pervasive meta-programming) will likely require adaptation of extraction and modeling steps, and the reader should be cautious about direct transfer without such adjustments.

ISSN: 3071-3315
PP: 01-24.

8.3. Construct validity

Construct validity concerns whether the operational definitions and measurements used in experiments truly capture the theoretical constructs of interest—here, architectural drift and violation severity. Threats arise from imperfect proxies used for labels (rule-based detectors or limited human annotations), the possible mismatch between computed structural deviation scores and actual maintenance cost, and choices in feature engineering that bias models toward certain manifestations of drift. To partially mitigate these concerns, we triangulate labels using multiple sources (architectural documentation when available, historical issue trackers, and automated rule checks) and validate structural deviation metrics against expert judgments on a sampled subset. We also conduct sensitivity analyses to examine how different label definitions affect results. Despite these steps, the inherent subjectivity of architectural intent and severity means that measured performance should be interpreted relative to our operationalization, and we encourage future work to develop richer, community-agreed benchmarks for architectural drift.

8.4. Conclusion validity

Conclusion validity pertains to the statistical reliability of inferences drawn from the experiments—whether observed effects are unlikely to be due to chance. Threats include small sample sizes for rare events (e.g., illegal dependency formations), insufficient numbers of independent repositories for robust generalization, and multiple-comparison problems when evaluating many model variants. We address these by using appropriate statistical tests, reporting confidence intervals, and applying correction methods when conducting multiple hypothesis tests. For rare-event tasks, we supplement aggregate metrics with ranking-based evaluations and qualitatively analyze individual correct and incorrect cases to provide richer evidence. Nevertheless, some claims—particularly about absolute performance in industrial settings—should be viewed as promising preliminary evidence rather than definitive proof; larger-scale replication studies and production deployments are necessary to fully validate the framework's practical efficacy.

9. CONCLUSION AND FUTURE WORK

9.1. Summary of contributions

This paper presented a principled, learning-driven approach for forecasting software architectural drift by modeling software systems as richly attributed, multi-layer graphs and applying Graph Neural Networks augmented with temporal modeling. We contributed a comprehensive graph-construction pipeline that extracts module-, package-, and class-level relations as well as call and data-flow edges, a feature-engineering schema combining static metrics, structural-role descriptors, commit-history signals, and semantic code embeddings, and a suite of GNN-based encoders (including heterogeneous and temporal variants) tailored to three complementary prediction tasks: node-level anomaly scoring, edge-formation and violation forecasting, and subgraph-level drift detection. Empirically, we demonstrated that the integrated model significantly outperforms rule-based conformance checkers, traditional ML baselines, and static-analysis heuristics on diverse repositories, and we provided ablation studies that quantify the

ISSN: 3071-3315
PP: 01-24.

necessity of fusing historical, semantic, and structural signals. Beyond raw performance, we contributed practical artifacts: an explainability layer that maps model outputs to human-readable architectural rules, a scalable processing architecture for large codebases, and case-study evidence showing actionable early warnings that helped triage emergent architecture problems. Collectively, these contributions shift architectural governance from reactive detection to proactive forecasting and provide a foundation for embedding predictive drift signals into engineering workflows.

9.2. Practical implications for architecture governance

The proposed predictive framework has several pragmatic implications for how organizations govern architecture: first, prioritized, probability-calibrated alerts enable architects to focus limited engineering resources on the highest-risk modules and upcoming commits rather than chasing noisy, low-impact violations; second, early-warning capabilities create opportunities for incremental remediation—small targeted refactorings or design conversations—before drift solidifies into costly erosion, thereby reducing long-term technical debt and preserving architectural intent. The model's interpretable attributions (e.g., attention weights, salient features) make it easier for architects to understand drivers of risk and to decide whether flagged evolution is intentional or accidental, which is crucial for trust and adoption. Additionally, integrating drift forecasts into planning and code-review workflows can inform release gating policies and help quantify architecture-related risk in sprint planning. However, successful adoption also requires organizational processes for triaging model outputs, handling false positives (particularly around intentional refactorings), and maintaining up-to-date architectural specifications that the model can use as constraints or supervision signals.

9.3. Tooling and automation opportunities

There are immediate tooling opportunities to operationalize the predictive framework across the software development lifecycle. At the CI/CD level, drift-forecasting can be embedded as pre-merge checks that surface the probability and impact of introducing risky dependencies, coupled with suggested mitigations (e.g., API wrappers, alternative module targets). Visualization dashboards that overlay predicted risky edges and subgraphs onto architecture diagrams help teams rapidly triage and understand spatial risk concentration, while automated ticket generation can create traceable remediation tasks for predicted high-severity drift. Tooling can also enable closed-loop learning: human adjudication of flagged items (accepted refactorings vs. unintended violations) should feed back as labels to continuously improve the model. For larger orgs, integrating the framework into architecture review boards and governance dashboards allows trend monitoring across multiple repositories, providing portfolio-level insights into architecture health and emergent cross-project risks. Finally, lightweight, on-device or distilled model variants can support near-instant, per-commit inference to keep feedback latency low without expensive full-graph computations.

ISSN: 3071-3315
PP: 01-24.

9.4. Future research directions

Although the presented framework advances predictive architecture maintenance substantially, several promising research directions remain. First, pre-trained code graph models—large-scale graph encoders pretrained on massive corpora of code and dependency graphs—could provide transferable, semantically rich initializations that improve sample efficiency and cross-project generalization; exploring contrastive and masked-graph objectives for pretraining is a compelling next step. Second, deeper integration with CI/CD pipelines and developer tooling requires research into low-latency, incremental embedding updates, robust handling of renames and refactors, and human-in-the-loop learning schemes that reconcile model autonomy with architect oversight. Third, architecture evolution simulators—stochastic generative environments that model developer behaviors, dependency dynamics, and refactoring actions—would enable stress-testing of governance strategies, evaluation of intervention policies, and the creation of synthetic training data for rare but high-impact drift scenarios. Fourth, generative models for architectural planning could complement predictive systems by proposing alternative designs or refactor plans that minimize projected drift risk while satisfying performance and development-cost constraints; combining predictive risk scoring with constrained generative optimization would create a closed loop from forewarning to automated mitigation. Beyond these, empirical research into socio-technical factors—how team structure, ownership, and process choices influence drift signatures—and rigorous longitudinal field studies of production deployments will be essential to translate the technical advances into sustained industrial practice.

REFERENCES

[1] Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). *A Survey of Machine Learning for Big Code and Naturalness*. ACM Computing Surveys, 51(4), 1–37.

[2] Chen, J., Ma, X., Liu, Y., & Xu, S. (2020). *Graph Neural Networks for Code Representation and Analysis*. Proceedings of the 42nd International Conference on Software Engineering, 1205–1216.

[3] Joblin, M., Apel, S., Siegmund, J., & Kästner, C. (2017). *Classifying Code Changes via Information Flow Analysis Using Graph Representations*. Empirical Software Engineering, 22(4), 1832–1863.

[4] Bogart, C., Kästner, C., Herbsleb, J., & Thung, F. (2016). *How Developers Diagnose Architecture Decay in Long-Lived Projects*. Proceedings of the 38th International Conference on Software Engineering, 467–478.

[5] Kikas, R., Madej, L., & Dumas, M. (2019). *Detecting Software Architecture Drift with Dependency Graph Metrics*. Information and Software Technology, 110, 39–53.

[6] Hassan, A. E. (2009). *Predicting Faults Using the Complexity of Code Changes*. Proceedings of the IEEE International Conference on Software Maintenance, 78–87.

[7] Bavota, G., Russo, B., Oliveto, R., & Di Penta, M. (2015). *Understanding the Impact of Structural Dependencies on Software Evolution*. IEEE Transactions on Software Engineering, 41(4), 341–365.

[8] Zimmermann, T., Nagappan, N., & Zeller, A. (2013). *Predicting Bugs Using Dependencies and Change Patterns*. Empirical Software Engineering, 18(4), 653–698.

[9] Lin, W., Chen, Z., & Yu, T. (2020). *Detecting Architecture Violations Using Deep Graph Representations*. ACM Transactions on Software Engineering and Methodology, 29(3), 1–28.

[10] Mens, T., Wermelinger, M., & Ducasse, S. (2005). *Studying Software Evolution Through Multiple Views: A Case Study*. Proceedings of the 21st IEEE International Conference on Software Maintenance, 1–10.