International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 5, Issue 4, PP 261-272, 2024
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I4P127

Large Language Model-Based Program Comprehension for Enterprise Software Repositories: A Context-Aware Framework for Code Summarization, Dependency Analysis, and Impact Prediction

Dr. Karthik .S[1], Dr. Adithya Balasubramanyam[2], Dr. Roopa Ravish[3], Dr. Surabhi Narayan[4]
[1,2,3,4]Professor, PES University CSE, Bengaluru, Karnataka, India.

Abstract: Enterprise software repositories contain complex, long-lived, and continuously evolving code assets whose behavior is distributed across services, frameworks, configuration files, database schemas, event streams, and deployment scripts. Conventional program comprehension tools provide useful static analysis, search, and dependency visualization capabilities, but they often fail to explain why a code element exists, how a change propagates across enterprise boundaries, and which downstream artifacts are likely to be affected by a modification. Large language models have recently demonstrated strong ability in source code understanding, documentation generation, and natural-language reasoning over program artifacts. However, direct application of generic LLMs to enterprise repositories is insufficient because enterprise comprehension requires contextual grounding, dependency-aware retrieval, architectural constraints, security awareness, and traceable impact prediction. This paper proposes RepoComprehend-LLM, a context-aware framework for LLM-based program comprehension in enterprise software repositories. The framework integrates repository mining, static dependency extraction, semantic code representation, retrieval-augmented generation, graph-based impact propagation, and confidence-calibrated explanation generation. Unlike isolated method-level summarization, the proposed framework generates hierarchical summaries at method, class, service, API, and business-capability levels; constructs multi-layer dependency graphs across code, configuration, database, event, and deployment artifacts; and predicts change impact using structural, semantic, historical, and developer-activity signals. The paper presents the architecture, methodology, scoring functions, evaluation protocol, implementation considerations, and enterprise governance controls for deploying such a framework in regulated software engineering environments. The central contribution is a research-oriented, implementation-ready approach that treats program comprehension as a context-aware, evidence-grounded, and continuously updated intelligence layer for software maintenance, modernization, and release risk assessment.

Keywords: Large Language Models, Program Comprehension, Software Repository Mining, Code Summarization, Dependency Analysis, Change Impact Prediction, Retrieval-Augmented Generation, Enterprise Software Engineering, Software Maintenance, Technical Debt.

1. Introduction

Enterprise software systems are no longer organized as isolated applications with simple source-code boundaries. They are distributed socio-technical ecosystems composed of microservices, shared libraries, rules engines, message brokers, API gateways, configuration repositories, data pipelines, cloud infrastructure, and continuous delivery workflows. In such environments, program comprehension is not limited to understanding what a function does. It requires understanding the role of a code element within a broader business process, the upstream and downstream systems that depend on it, the non-functional constraints that govern it, and the release consequences of modifying it. Traditional static analysis, documentation search, and architecture diagrams provide partial support, but they often become stale, fragmented, or too low-level for enterprise developers who must make change decisions under delivery pressure.

Large language models have created a new opportunity for program comprehension because they can translate code into natural language, infer intent from identifiers and comments, and support interactive explanation. Foundational code models such as CodeBERT showed that bimodal training over programming language and natural language can support code search and documentation-generation tasks [1]. This capability is directly relevant to enterprise comprehension because many maintenance questions are expressed in natural language, such as “which services will be affected if this field changes?” or “why does this validation rule exist?” However, enterprise repositories introduce context-length, grounding, privacy, and dependency-tracing challenges that are not solved by simple prompt-based code summarization.

The need for context-aware program intelligence is aligned with broader work on AI-assisted lifecycle governance and software quality management. Sivva, Thalakanti, Bandari, and Yettapu describe the integration of machine learning defect prediction and automated testing into agile lifecycle governance, emphasizing the need for architecture-centered intelligence rather than isolated analytics [2]. That perspective is important for repository comprehension because summaries and impact predictions become valuable only when they influence real engineering decisions, including code review, release planning, regression-test selection, modernization prioritization, and operational risk reduction.

This paper addresses the following research problem: How can large language models be integrated with repository mining and dependency analysis to provide reliable, context-aware program comprehension for enterprise software repositories? The problem is difficult for four reasons. First, source code is not self-contained; meaning is often distributed across annotations, configuration files, API specifications, database schemas, feature flags, and deployment manifests. Second, dependencies are multi-modal; a method may depend on a class, a REST endpoint, a Kafka topic, a database column, a cloud secret, or a rule expression. Third, enterprise repositories evolve through developer activity, pull requests, incident fixes, and release branches, so historical change signals are essential. Fourth, LLM explanations can be fluent but incorrect unless grounded in retrievable evidence and constrained by graph-derived facts.

The proposed framework, RepoComprehend-LLM, combines three complementary intelligence layers. The first layer performs context-aware code summarization, generating explanations at multiple abstraction levels and linking each statement to retrieved evidence. The second layer performs multi-layer dependency analysis, constructing a typed software knowledge graph from static analysis, build files, API contracts, database definitions, deployment manifests, and runtime traces where available. The third layer performs impact prediction, estimating the probability that a proposed change will affect downstream code, tests, services, data objects, or business capabilities. The framework does not treat the LLM as an autonomous authority. Instead, the LLM is used as a reasoning and language-generation component governed by retrieved context, dependency evidence, confidence scoring, and human-in-the-loop review.

Graph-aware code representation research supports the claim that program comprehension requires structural information beyond token sequences. GraphCodeBERT incorporated data-flow information into pre-training to better capture semantic relationships in code [3]. RepoComprehend-LLM extends this principle to enterprise repositories by representing dependencies across multiple artifact types, not only variables and methods. The framework therefore treats source code as one component of a repository-scale knowledge graph.

The remainder of the paper is organized as follows. Section II reviews related work. Section III defines the research problem and design objectives. Section IV presents the proposed architecture. Section V describes the methodology for repository indexing, context retrieval, summarization, dependency extraction, and impact scoring. Section VI defines the evaluation protocol. Section VII discusses analytical results and expected enterprise outcomes. Section VIII presents implementation and governance considerations. Section IX discusses threats to validity. Section X concludes the paper.

2. Background and Related Work

Program comprehension has traditionally been studied through static analysis, information retrieval, software visualization, traceability recovery, and change impact analysis. These approaches help developers identify relevant code fragments, understand dependencies, and estimate the effects of modifications. In enterprise repositories, however, the scale and heterogeneity of artifacts create a gap between what traditional tools expose and what developers need to know. A call graph may show that method A calls method B, but it may not explain that the call protects a regulatory rule, updates a billing state, or prevents an exception in a downstream integration.

The rise of intelligent CI/CD and risk-aware deployment frameworks has increased the demand for automated repository understanding. Thalakanti and Goud Bandari propose machine learning-driven risk detection for banking CI/CD environments and emphasize that deployment decisions should consider risk signals rather than rely only on binary build outcomes [4]. This insight is relevant because program comprehension is not merely descriptive. A high-quality comprehension system should help predict which changes are risky, which tests should be prioritized, and which services require additional validation before release.

Pre-trained code models provide a technical foundation for natural-language interaction with source code. CodeT5 introduced an identifier-aware encoder-decoder architecture that supports code understanding and generation tasks by explicitly leveraging developer-assigned identifiers [5]. This matters for enterprise repositories because identifiers often encode domain concepts, such as “claim,” “order,” “invoice,” “member,” “coverage,” or “rebate.” However, identifiers alone are not sufficient. Enterprise code may contain legacy naming conventions, abbreviated fields, copied patterns, or domain-specific terms whose meanings require project-level context.

In regulated industries, program comprehension must also account for compliance, auditability, and security. Gudi's work on AI-driven fax-to-digital prescription automation demonstrates the importance of combining OCR, machine learning, and microservices in healthcare operations where data correctness and traceability are essential [6]. Similar constraints apply to LLM-based repository comprehension. If an LLM summarizes a validation rule or predicts the impact of a field change, the system must provide evidence and confidence rather than unsupported natural-language output.

Unified code representation models have further expanded the ability to support multiple code intelligence tasks. UniXcoder introduced cross-modal pre-training for code representation and used attention mechanisms to support understanding and generation within a unified framework [7]. RepoComprehend-LLM adopts a similar multi-task philosophy but places it inside an enterprise pipeline where summarization, dependency extraction, impact prediction, and developer feedback are continuously connected.

Dependency modeling has long been central to failure propagation and software maintenance. Mutyam's graph-based modeling of service dependencies for predicting failure propagation in distributed systems highlights the importance of representing services as connected structures rather than isolated components [8]. RepoComprehend-LLM applies this idea to source repositories by constructing a dependency graph that includes code, configuration, APIs, events, database artifacts, and tests. The graph is not merely used for visualization; it becomes an input to retrieval, summarization, and impact scoring.

Benchmarks such as CodeXGLUE have helped standardize evaluation for code understanding and generation tasks [9]. Nevertheless, many benchmark tasks remain function-level or file-level, while enterprise program comprehension is repository-level. A model that performs well on code summarization may still fail to infer cross-file dependency behavior or understand the operational consequences of a configuration change. This gap motivates repository-level evaluation metrics that combine factuality, context coverage, dependency accuracy, and change-impact usefulness.

Prior applied software lifecycle research has emphasized decision intelligence for project governance. Gunda, Yettapu, Bodakunti, and Bikki proposed a decision intelligence methodology for AI-driven agile lifecycle governance and architecture-centered project management [10]. RepoComprehend-LLM builds on this governance-oriented view by transforming repository artifacts into actionable intelligence for development teams, architects, release managers, and quality engineers.

Large-scale code-search corpora have also shaped program comprehension research. CodeSearchNet introduced a multilingual corpus and semantic code search challenge that supports natural-language retrieval over code [11]. Semantic code search is essential for RepoComprehend-LLM because developers often begin comprehension with natural questions. However, the proposed framework extends retrieval beyond lexical or embedding similarity by incorporating dependency distance, ownership history, change co-occurrence, and architectural layer constraints.

Automated testing research remains closely connected to program comprehension. Gudi's comparative analysis of automated testing frameworks in Java enterprise systems highlights that testing strategy is a core element of reliability engineering [12]. In the proposed framework, dependency-aware impact prediction is linked to test recommendation. When a method, API, or configuration artifact changes, the system ranks tests not only by file proximity but also by graph connectivity and historical failure association.

Recent repository-level benchmarks show that multi-file code intelligence remains challenging. RepoBench was introduced to evaluate repository-level code auto-completion and retrieval in Python and Java, reflecting the need to move beyond single-file contexts [13]. RepoComprehend-LLM similarly assumes that repository-level comprehension requires explicit context selection. Instead of feeding an entire repository into an LLM, the framework retrieves compact, typed, and ranked evidence bundles.

Privacy-preserving learning is also relevant to enterprise adoption. Thalakanti, Goud Bandari, and Sivva discuss federated learning for fraud detection across financial institutions, emphasizing operational governance and privacy-preserving collaboration [14]. Enterprise program comprehension systems must follow comparable constraints. Source code, architecture, credentials, and business rules are sensitive assets; therefore, repository embeddings, prompts, and generated summaries must be governed through access control, local deployment options, and audit logging.

The CodePlan framework introduced repository-level coding with planning, combining dependency analysis, change-impact analysis, and LLM calls for multi-step repository modifications [15]. While CodePlan focuses on repository-level coding tasks, RepoComprehend-LLM focuses on comprehension and risk prediction before code is changed. The two directions are complementary: high-quality comprehension can improve automated planning, and planning feedback can refine future impact models.

Secure microservice architectures provide another important enterprise context. Gudi's work on HIPAA-compliant prescription processing using AWS and OpenShift illustrates the role of cloud-native design, compliance boundaries, and microservice governance in healthcare software [16]. RepoComprehend-LLM is designed for similar settings where source repositories contain not only application logic but also deployment templates, security policies, and infrastructure configuration that affect compliance and runtime behavior.

Change impact analysis is a mature area of software engineering, but LLM-based systems require new integration patterns. Lehnert's work on software change impact analysis surveyed approaches for identifying consequences of modifications across evolving systems [17]. The proposed framework does not replace traditional impact analysis. Instead, it combines structural impact analysis with semantic explanation, historical learning, and natural-language developer interaction.

Explainability is essential when AI is used in decision support. Bandari, Sivva, and Thalakanti present explainable AI for regulatory-grade fraud detection with auditable decision pathways [18]. The same principle applies to software repository comprehension. An impact prediction without an evidence path is not sufficient for enterprise release decisions. RepoComprehend-LLM therefore reports affected artifacts, graph paths, retrieved snippets, confidence scores, and uncertainty reasons.

Open code LLMs such as CodeT5+ further demonstrate the expansion of code understanding and generation models [19]. However, model capability alone does not guarantee enterprise trust. The proposed framework treats the LLM as one part of a larger system that includes retrieval, graph constraints, policy filters, human validation, and continuous monitoring.

Gunda's discussion of the expanding role of machine learning models in software development anticipates a shift from manual developer assistance toward embedded intelligence throughout the engineering lifecycle [20]. RepoComprehend-LLM operationalizes this shift by embedding program comprehension into pull requests, code review workflows, release planning, and modernization analysis.

Earlier neural code representation methods also inform the proposed approach. Code2vec showed that path-based representations can capture semantic properties of code by using abstract syntax tree paths [21]. Although modern LLMs operate at larger scale, this earlier work remains relevant because it demonstrates that source-code structure contributes meaningful signals beyond raw tokens. RepoComprehend-LLM uses this idea through graph-derived context features.

Enterprise repositories often include performance-sensitive and operationally critical workflows. Gudi's work on predictive analytics and Redis-backed caching for specialty medication fulfillment and inventory management illustrates how enterprise software behavior is shaped by data latency, caching, and operational constraints [22]. For this reason, impact prediction in RepoComprehend-LLM includes runtime-adjacent artifacts such as cache keys, message topics, scheduled jobs, and database access patterns when these artifacts are available.

The broader "big code" literature shows that software repositories contain statistical regularities that can be learned by machine learning systems. Allamanis, Barr, Devanbu, and Sutton surveyed machine learning over large code corpora and argued that source code exhibits naturalness that can be exploited for software engineering tasks [23]. RepoComprehend-LLM extends this idea from corpus-level learning to enterprise-specific context modeling.

Sivva's end-to-end AI-based systems engineering paradigm links lifecycle governance, predictive quality assurance, automation economics, and cybersecurity intelligence [24]. This integrated view is useful because program comprehension should not be evaluated only by summary quality. It should also be judged by whether it improves defect prevention, impact awareness, delivery safety, and engineering productivity.

Classical code summarization research remains valuable. Sridhara and colleagues showed that method summaries can be generated by identifying important statements and translating them into natural language [25]. LLMs improve the fluency and flexibility of such summaries, but the fundamental challenge remains: summaries must select the right facts, preserve semantic correctness, and avoid irrelevant detail.

Gunda's comparative analysis of machine learning models for software defect prediction reinforces the importance of empirical model evaluation in software engineering [26]. RepoComprehend-LLM adopts a similar evaluation discipline by defining measurable outputs for summarization fidelity, dependency extraction accuracy, and impact prediction performance rather than relying only on subjective developer impressions.

Concept location research is also relevant because developers often search for where a business concept is implemented. Marcus, Sergeyev, Rajlich, and Maletic proposed information-retrieval methods for locating concepts in source code [27].

RepoComprehend-LLM generalizes concept location into interactive comprehension, where concept queries return summaries, dependency paths, affected tests, and change-risk explanations.

Balerao's converged AI architecture for innovation, software lifecycle optimization, and cybersecurity risk mitigation emphasizes that AI systems in software engineering should connect productivity with risk management [28]. This paper follows that principle by combining developer-facing explanations with governance-grade controls for confidence, auditability, and access management.

Recent work on context-aware code summary generation shows that project context improves the usefulness of method-level summaries [29]. RepoComprehend-LLM adopts this context-first assumption but expands it to repository-wide dependency and impact analysis. A method summary should explain not only what the method computes but also why it exists in the project and what may break if it changes.

Cloud observability and root-cause analysis research also informs enterprise repository comprehension. Manga, Sivva, and Manga propose a unified architecture for AI-enhanced observability and automated root-cause analysis in cloud systems [30]. RepoComprehend-LLM can use observability signals as optional evidence, allowing impact prediction to incorporate production incidents, error logs, and service-level dependencies.

Gunda's work on fault prediction using Random Forest, Logistic Regression, and K-nearest neighbors illustrates that software reliability analysis benefits from comparing multiple model families [31]. The proposed framework similarly separates structural baselines, semantic baselines, graph-based models, and LLM-augmented models in the evaluation protocol.

Finally, recent impact-analysis work such as ATHENA combines dependence graph information with conceptual coupling to improve impact understanding [32]. RepoComprehend-LLM follows this hybrid direction by combining graph paths with semantic similarity and historical change signals.

3. Research Problem and Design Objectives

The central research problem is to design a program comprehension framework that can answer enterprise maintenance questions with contextual accuracy, dependency awareness, and actionable impact prediction. The framework must address a set of practical constraints that are often overlooked in narrow code summarization research.

First, enterprise repositories are heterogeneous. A single feature may involve Java or Python code, TypeScript front-end components, YAML deployment files, SQL scripts, API specifications, Terraform modules, message schemas, and test fixtures. A comprehension system that indexes only source files will miss important sources of meaning. RepoComprehend-LLM therefore models repository artifacts as typed nodes in a software knowledge graph.

Second, enterprise repositories are historically shaped. Code meaning is partly encoded in commit messages, pull-request discussions, incident tickets, release notes, and developer ownership patterns. The same method may appear simple in isolation but may have been changed repeatedly due to production defects. The proposed framework includes repository mining features such as change frequency, co-change relationships, author concentration, defect-linked commits, and release proximity.

Third, enterprise program comprehension must support different users. A developer may need a line-level explanation. An architect may need service-level dependency analysis. A quality engineer may need impacted test recommendations. A product owner may need business-capability summaries. RepoComprehend-LLM therefore generates hierarchical summaries and adapts the explanation level to the user role while preserving evidence links.

Fourth, LLM output must be controlled. Generic LLMs may hallucinate nonexistent dependencies, overstate confidence, or generate plausible but incorrect business explanations. The proposed framework mitigates this risk through retrieval-augmented generation, graph-constrained prompting, evidence attribution, uncertainty labeling, and optional local model deployment for sensitive repositories.

The design objectives are as follows. The first objective is context completeness, meaning that the system must retrieve not only lexical matches but also semantically and structurally related artifacts. The second objective is dependency faithfulness, meaning that generated explanations must align with extracted dependency paths. The third objective is impact usefulness, meaning that predicted impacts must help developers choose review scope, test scope, and release safeguards. The fourth objective is enterprise governability, meaning that the system must preserve confidentiality, traceability, and human accountability.

4. Proposed Framework: RepoComprehend-LLM

RepoComprehend-LLM is organized into six modules: repository ingestion, artifact normalization, software knowledge graph construction, contextual retrieval, LLM reasoning and summarization, and impact prediction.

4.1. Repository Ingestion Layer

The ingestion layer collects source files, build descriptors, dependency manifests, API schemas, database migration scripts, configuration files, deployment manifests, test files, commit metadata, pull-request metadata, and optional runtime observability traces. The ingestion process is incremental. Instead of reprocessing the entire repository on every commit, the system computes changed artifacts, affected dependency neighborhoods, and updated embeddings.The ingestion layer also performs artifact classification. Files are labeled as source code, unit test, integration test, configuration, infrastructure, database, API contract, message schema, documentation, or build artifact. This classification allows the retrieval module to balance context. For example, when summarizing a REST controller, the retriever should include route annotations, request and response models, service-layer calls, tests, API documentation, and relevant configuration.

4.2. Artifact Normalization Layer

The artifact normalization layer parses files into structured representations. For source code, it extracts functions, classes, interfaces, imports, annotations, comments, method signatures, call sites, data-flow hints, and exception paths. For configuration, it extracts keys, environment variables, endpoints, feature flags, secrets references, and service bindings. For database artifacts, it extracts tables, columns, stored procedures, constraints, and migration dependencies. For API contracts, it extracts routes, request models, response models, status codes, and authentication schemes.

Normalization is essential because LLMs should not receive raw repository content without structure. A typed representation allows the system to generate compact evidence packets. For example, a Java method may be represented by its signature, enclosing class, called methods, domain identifiers, selected body statements, related tests, recent commits, and dependency graph neighborhood.

4.3. Software Knowledge Graph

The software knowledge graph is the core structural component of the framework. Nodes represent artifacts such as methods, classes, modules, services, APIs, topics, tables, columns, configuration keys, tests, deployment units, and business capabilities. Edges represent relationships such as calls, imports, implements, extends, reads, writes, publishes, subscribes, validates, deploys-to, tests, co-changes-with, owned-by, and fails-with.

The graph supports both deterministic and probabilistic edges. Deterministic edges are extracted through parsing or static analysis. Probabilistic edges are inferred through semantic similarity, historical co-change, naming patterns, or LLM-assisted candidate extraction. Each edge stores provenance, confidence, timestamp, and extraction method. This allows the system to distinguish a parser-confirmed method call from a semantically inferred business relationship.

4.4. Contextual Retrieval

The retrieval module selects evidence for a comprehension query. A query may be developer-initiated, such as “summarize this service,” or event-triggered, such as a pull request that changes a field. Retrieval uses a hybrid scoring function:

$$\text{ContextScore}(a, q) = \alpha\ \text{SemanticSim}(a, q) + \beta\ \text{GraphProximity}(a, q) + \gamma\ \text{ChangeCoupling}(a, q) + \delta\ \text{ArtifactPriority}(a) + \varepsilon\ \text{Recency}(a)$$

Here, SemanticSim captures embedding similarity between the query and artifact. GraphProximity measures distance in the software knowledge graph. ChangeCoupling measures historical co-change frequency. ArtifactPriority increases the score of tests, API contracts, schemas, and deployment files when relevant. Recency accounts for recent commits or incidents. The weights are configurable by task. Summarization may emphasize semantic similarity, while impact prediction may emphasize graph proximity and change coupling.

4.5. LLM Reasoning and Summarization

The LLM module receives a structured evidence packet rather than the whole repository. The packet includes the target artifact, selected dependency paths, related tests, domain terms, recent change history, and confidence constraints. The model is instructed to generate evidence-grounded outputs and to avoid unsupported claims. Output is organized into several formats:

- Method summary: explains behavior, inputs, outputs, side effects, and exceptions.
- Class or module summary: explains responsibility, collaborators, and lifecycle role.
- Service summary: explains APIs, data dependencies, event interactions, and deployment context.
- Business capability summary: maps technical artifacts to domain functions.
- Change explanation: explains what a proposed change modifies and why it matters.
- Impact narrative: explains likely affected artifacts and recommended validation steps.

The framework separates natural-language fluency from factual confidence. A generated summary includes evidence references and uncertainty tags. If the retrieved context is insufficient, the output explicitly states that the explanation is partial.

4.6. Impact Prediction Module

Impact prediction estimates the likelihood that a change to artifact x affects artifact y. The model uses four feature families: structural, semantic, historical, and operational. Structural features include graph path length, edge types, fan-in, fan-out, and dependency centrality. Semantic features include embedding similarity between artifacts, shared domain terms, and LLM-extracted intent labels. Historical features include co-change frequency, common authorship, previous defects, and release timing. Operational features include runtime dependency, incident association, alert linkage, and deployment coupling when available.

The impact score is defined as:

$$\text{Impact}(x, y) = \sigma(w_s S_{xy} + w_g G_{xy} + w_h H_{xy} + w_o O_{xy} + b)$$

Where S_{xy} denotes semantic relatedness, G_{xy} denotes graph connectivity, H_{xy} denotes historical coupling, O_{xy} denotes operational coupling, and σ is a logistic transformation. The system ranks impacted artifacts and groups them by type: source files, tests, APIs, schemas, services, configurations, and business capabilities.

5. Methodology

The proposed methodology includes five phases: repository preparation, graph construction, summarization generation, impact prediction, and developer feedback integration.

5.1. Repository Preparation

Repositories are cloned or connected through enterprise source-control systems. The system identifies branches, modules, build files, programming languages, and repository ownership metadata. Sensitive files are detected and excluded or masked according to policy. Secrets, credentials, and tokens are never included in LLM prompts. For highly regulated environments, embeddings and LLM inference can be executed within the enterprise network.

The preparation phase also creates a domain vocabulary. This vocabulary is extracted from identifiers, comments, API names, database terms, commit messages, and documentation. Domain terms are clustered to identify synonyms and abbreviations. For example, “cust,” “customer,” and “member” may be related in one repository but not another. This vocabulary improves summarization because the LLM receives repository-specific terminology instead of relying only on general programming knowledge.

5.2. Graph Construction

The graph construction process combines language-specific parsing and repository-wide mining. Static parsers extract functions, classes, imports, method calls, inheritance relationships, annotations, and package structures. Build-tool analyzers extract dependencies from Maven, Gradle, npm, pip, Docker, or other ecosystem files. API analyzers extract route definitions and OpenAPI contracts. Database analyzers extract schema relationships and migration history. Event analyzers extract message producers and consumers when topic names and schema references are available.

Historical edges are computed from commit and pull-request data. If two files frequently change together, a co-change edge is added. If a test file frequently changes with a production file, a test-coupling edge is added. If a production incident references a component, an incident edge may be added. These edges are time-weighted so that recent relationships can be emphasized without discarding older architectural dependencies.

5.3. Context-Aware Summarization

Summarization begins with artifact selection. The user may select a method, class, package, service, or pull request. The retrieval module builds a context packet containing the target artifact, direct dependencies, selected transitive dependencies, related tests, configuration references, historical notes, and domain vocabulary. The LLM then generates a structured summary using a constrained prompt.

A method-level summary includes purpose, control flow, important conditions, data access, exceptions, side effects, and related tests. A service-level summary includes exposed APIs, internal modules, external dependencies, data stores, events, deployment context, and business function. A pull-request summary includes changed artifacts, likely motivation inferred from commits, affected dependency neighborhoods, risk areas, and validation recommendations.

The summarization process is intentionally hierarchical. Method summaries are aggregated into class summaries. Class summaries are aggregated into module summaries. Module summaries are aggregated into service summaries. This prevents

the framework from forcing the LLM to infer a service-level explanation from raw code alone. Instead, service-level explanation is generated from lower-level summaries and graph evidence.

5.4. Dependency Analysis

Dependency analysis uses typed graph traversal. For a target artifact, the system retrieves incoming and outgoing edges, groups them by type, and computes centrality and dependency depth. Dependencies are classified into compile-time, runtime, data, configuration, event, deployment, and test dependencies. This classification is important because not all dependencies imply the same kind of impact. A compile-time dependency may cause build failure, while a data dependency may cause silent downstream inconsistency.

The dependency analyzer also detects architecture boundary violations. For example, if a controller directly accesses a repository layer or if a service bypasses a domain abstraction, the system flags the relationship. LLM-generated explanations can then include architecture-aware observations, but only when the graph provides evidence.

5.5. Impact Prediction

Impact prediction is triggered by a change event or by a developer query. The system identifies changed artifacts, expands graph neighborhoods, computes impact scores, and ranks likely affected items. The ranking includes explanations. For example, a database column change may affect an ORM entity, validation rule, API response model, reporting query, integration test, and downstream data pipeline.

The model can be trained using historical pull requests as weak supervision. If files changed together in the same pull request or defect fix, they are treated as positive impact examples. Negative examples are sampled from unrelated artifacts with graph distance and module constraints. Training data is imperfect because co-change does not always imply true dependency, but it provides useful enterprise-specific signals. Human feedback from developers can refine labels over time.

5.6. Developer Feedback Loop

Enterprise deployment requires continuous feedback. Developers can mark summaries as accurate, incomplete, misleading, or stale. They can confirm or reject predicted impacts. They can link missed artifacts to a change. This feedback updates retrieval weights, graph edges, and model calibration. Feedback is also used to identify documentation gaps. If developers repeatedly correct a summary for the same component, the system flags that component as requiring documentation improvement or architecture review.

6. Evaluation Protocol

A rigorous evaluation protocol is necessary because LLM-generated program comprehension can appear useful even when it is incomplete or inaccurate. The proposed evaluation includes three tracks: summarization evaluation, dependency evaluation, and impact prediction evaluation.

6.1. Summarization Evaluation

Summarization quality is evaluated using expert review and automated checks. Expert reviewers assess factual correctness, completeness, conciseness, domain relevance, and actionability. Automated checks verify whether the summary mentions unsupported dependencies or omits critical retrieved facts. Metrics include factual precision, factual recall, unsupported-claim rate, evidence coverage, and reviewer usefulness score.

A strong evaluation should compare four baselines. The first baseline is raw code without generated summary. The second baseline is comment extraction. The third baseline is generic LLM summarization using only the target file. The fourth baseline is retrieval-augmented summarization without dependency graph features. RepoComprehend-LLM should be evaluated against these baselines to determine whether context and graph evidence improve comprehension quality.

6.2. Dependency Evaluation

Dependency extraction is evaluated against parser-confirmed dependencies, build dependencies, manually validated service dependencies, and known API or database relationships. Metrics include precision, recall, edge-type accuracy, and path explanation correctness. Because some enterprise dependencies are implicit, manual validation is important. For example, a Kafka topic dependency may be represented only as a string constant, and a database relationship may be implicit in query text.

Dependency explanations are evaluated separately from dependency detection. A system may correctly identify that service A depends on service B but incorrectly explain the business reason. Therefore, reviewers should assess whether the explanation is supported by retrieved code, configuration, or documentation.

6.3. Impact Prediction Evaluation

Impact prediction is evaluated using historical pull requests, defect fixes, and release incidents. For each historical change, the system receives the initial changed artifact and predicts additional impacted artifacts. Metrics include top-k recall, mean reciprocal rank, precision at k, normalized discounted cumulative gain, and missed-critical-artifact rate. Test recommendation can be evaluated using historical test failures or changed test files.

The evaluation should also include developer-in-the-loop studies. Developers are asked to perform comprehension and impact-analysis tasks with and without the framework. Measures include time to answer, correctness of identified dependencies, confidence, perceived usefulness, and number of missed artifacts. Such studies are necessary because the ultimate purpose of program comprehension is to improve human decision-making.

6.4. Governance Evaluation

Governance evaluation measures whether the system follows enterprise constraints. Metrics include sensitive-data leakage rate, unsupported-claim rate, prompt policy compliance, evidence traceability, access-control correctness, and audit-log completeness. In regulated environments, these metrics are not secondary; they determine whether the tool can be deployed.

7. Analytical Results and Expected Outcomes

Since this paper proposes a framework and evaluation protocol rather than reporting a completed controlled experiment on a private enterprise repository, the results are presented as analytical expectations grounded in the architecture. The framework is expected to improve program comprehension in four measurable ways.

First, context-aware summarization should produce more accurate and useful summaries than target-only LLM summarization. Target-only summarization may describe local control flow but miss service role, configuration behavior, and downstream impact. By including dependency paths, related tests, domain terms, and historical changes, RepoComprehend-LLM should reduce unsupported claims and improve evidence coverage.

Second, multi-layer dependency analysis should improve impact visibility. Traditional call graphs may detect code-level dependencies but miss configuration, database, event, and deployment relationships. Enterprise changes often fail because these non-code relationships are overlooked. A typed software knowledge graph can expose these relationships and support more complete impact prediction.

Third, historical repository mining should improve ranking. Graph connectivity alone may over-rank structurally close but rarely co-changing artifacts. Historical co-change alone may over-rank files that changed together due to temporary release activities. Combining graph, semantic, historical, and operational features should provide a more balanced impact score.

Fourth, evidence-grounded LLM explanations should improve developer trust. Developers are unlikely to trust black-box impact predictions without reasons. RepoComprehend-LLM provides graph paths, retrieved snippets, and uncertainty notes. This enables developers to validate recommendations quickly and reject incorrect predictions when necessary.

The expected enterprise outcomes include faster onboarding, improved code review, better regression-test selection, reduced change-related defects, more accurate modernization planning, and improved documentation quality. For legacy modernization, the framework can identify high-centrality components, poorly documented modules, unstable dependency clusters, and services with high change-risk concentration. For release planning, it can help teams identify changes requiring extended testing or architecture review.

8. Practical Enterprise Implementation

A practical implementation of RepoComprehend-LLM should begin with a limited pilot rather than immediate organization-wide deployment. A suitable pilot repository should contain multiple services, meaningful tests, active pull-request history, and known maintenance pain points. The initial deployment should focus on read-only program comprehension and pull-request impact summaries. Automated code modification should be excluded until comprehension quality is validated.

The recommended architecture uses an internal indexing service, a graph database, a vector database, a policy engine, and an LLM inference layer. The indexing service processes repository events and updates graph and embedding stores. The graph database stores typed artifact relationships. The vector database supports semantic retrieval. The policy engine enforces access control, masking, and prompt restrictions. The LLM inference layer may use an enterprise-hosted model or a controlled external model depending on data sensitivity.

Integration points include integrated development environments, pull-request platforms, CI/CD systems, architecture dashboards, and incident-management tools. In an IDE, the system can provide method and class explanations. In pull requests,

it can generate change summaries and predicted impacts. In CI/CD, it can recommend tests or flag high-risk changes. In architecture dashboards, it can show dependency clusters and modernization candidates. In incident tools, it can map runtime failures back to likely code and configuration artifacts.

A key implementation principle is progressive trust. Early outputs should be labeled as advisory. Developers should be able to inspect evidence and provide corrections. Over time, confidence thresholds can be tuned. High-confidence outputs may become part of standard code-review templates, while low-confidence outputs remain suggestions.

Security controls are essential. The system should avoid sending secrets, credentials, personal data, or proprietary business rules to unauthorized inference endpoints. It should log prompts and outputs for audit when policy allows. It should support role-based access so that users see only repositories and artifacts they are authorized to access. Generated summaries should inherit the access classification of their source artifacts.

9. Challenges and Threats to Validity

The first challenge is repository heterogeneity. Enterprise systems use multiple languages, frameworks, and build tools. A parser or dependency extractor that works well for Java may not work for JavaScript, Python, SQL, or YAML. The framework must therefore be modular and extensible.

The second challenge is implicit dependency detection. Some dependencies are encoded as strings, naming conventions, external configuration, or runtime behavior. Static analysis may miss these relationships, while LLMs may infer them incorrectly. Combining static analysis with historical mining and developer feedback reduces but does not eliminate this risk.

The third challenge is summary hallucination. Even with retrieval, an LLM may generate unsupported claims. Evidence grounding, constrained prompts, and unsupported-claim detection are necessary safeguards. However, no safeguard can guarantee perfect factuality.

The fourth challenge is evaluation difficulty. Enterprise impact ground truth is noisy. Files changed together in a pull request are not always truly dependent, and truly impacted files may not be changed because a developer missed them. Evaluation should therefore combine historical data with expert validation.

The fifth challenge is organizational adoption. Developers may reject the system if it adds noise to code review or produces generic summaries. Successful adoption requires careful UX design, fast response time, visible evidence, and feedback mechanisms.

The sixth challenge is privacy and compliance. Source repositories may contain sensitive intellectual property, regulated business logic, or security-relevant configuration. Organizations must choose deployment patterns that match their risk tolerance.

10. Future Research Directions

Future research should investigate adaptive retrieval strategies that learn which context types are most useful for each repository and task. For example, service summarization may benefit from API contracts and deployment files, while defect impact prediction may benefit from historical test failures and incident traces.

Another direction is causal impact modeling. Current impact prediction often relies on correlation through graph links and co-change history. Future systems should distinguish causal dependencies from coincidental co-change patterns. This may require combining static analysis, runtime tracing, controlled change experiments, and developer feedback.

A third direction is benchmark development for enterprise-scale comprehension. Existing benchmarks are valuable but often focus on open-source repositories and specific tasks. New benchmarks should include multi-language repositories, configuration dependencies, service boundaries, database schemas, API contracts, and realistic change-impact labels.

A fourth direction is human-AI collaboration. Program comprehension is not only a technical task; it is a cognitive workflow. Future studies should measure how developers use LLM-based explanations, when they trust them, when they ignore them, and how such tools affect onboarding, review quality, and maintenance outcomes.

A fifth direction is compliance-aware code intelligence. Regulated industries need repository comprehension systems that can explain not only technical dependencies but also policy, privacy, and audit implications. This requires integrating security classification, compliance metadata, and risk controls into the software knowledge graph.

11. Conclusion

This paper proposed RepoComprehend-LLM, a context-aware framework for large language model-based program comprehension in enterprise software repositories. The framework integrates repository mining, artifact normalization, software knowledge graph construction, retrieval-augmented generation, hierarchical code summarization, dependency analysis, and impact prediction. Its central argument is that enterprise program comprehension cannot be solved by generic method-level LLM summarization alone. It requires repository-level context, typed dependency evidence, historical change signals, governance controls, and human validation. The proposed framework contributes a structured architecture for transforming enterprise repositories into an intelligence layer that supports developers, architects, quality engineers, and release teams. It generates evidence-grounded summaries, identifies multi-layer dependencies, predicts likely change impacts, and supports test and review prioritization. By combining LLM reasoning with graph-constrained evidence and repository mining, RepoComprehend-LLM offers a practical research direction for improving software maintenance, modernization, reliability, and release safety in complex enterprise environments.

References

1. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online, 2020, pp. 1536–1547, https://doi.org/10.18653/v1/2020.findings-emnlp.139.
2. Sivva, S. D., Thalakanti, R. R., Bandari, S. S. G., & Yettapu, S. D. R. (2023). 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*, *4*(4), 167-172. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I4P118
3. D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. K. Deng, C. Fu, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *International Conference on Learning Representations*, 2021, https://openreview.net/forum?id=jLoC4ez43PZ.
4. Thalakanti, R. R., & Goud Bandari, S. S. (2024). 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*, *5*(4), 168-175. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I4P118
5. Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708, https://doi.org/10.18653/v1/2021.emnlp-main.685.
6. Gudi, S. R. (2024). 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*, *5*(1), 111-116. https://doi.org/10.63282/3050-922X.IJERET-V5I1P113
7. D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified Cross-Modal Pre-training for Code Representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, Dublin, Ireland, 2022, pp. 7212–7225, https://doi.org/10.18653/v1/2022.acl-long.499.
8. Mutyam, N. (2024). Graph-based modeling of service dependencies for predicting failure propagation in distributed systems. *International Journal of Multidisciplinary Evolutionary Research*, *5*(1), 113–116. https://doi.org/10.54660/IJMER.2024.5.1.113-116
9. S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *Proceedings of Machine Learning and Systems Datasets and Benchmarks Track*, 2021, https://arxiv.org/abs/2102.04664.
10. 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;4(1):102-8. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I1P112
11. H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," arXiv preprint arXiv:1909.09436, 2019, https://arxiv.org/abs/1909.09436.
12. Gudi, S. R. (2023). Enhancing Reliability in Java Enterprise Systems through Comparative Analysis of Automated Testing Frameworks. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(2), 151-160. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I2P115
13. T. Liu, C. Xu, and J. McAuley, "RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems," arXiv preprint arXiv:2306.03091, 2023, https://arxiv.org/abs/2306.03091.
14. Thalakanti, R. R., Goud Bandari, S. S., & Sivva, S. D. . (2024). Federated Learning for Privacy Preserving Fraud Detection across Financial Institutions: Architecture Protocols and Operational Governance. *International Journal of Emerging Research in Engineering and Technology*, *5*(2), 108-114. https://doi.org/10.63282/3050-922X.IJERET-V5I2P111

15. R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "CodePlan: Repository-level Coding using LLMs and Planning," arXiv preprint arXiv:2309.12499, 2023, https://arxiv.org/abs/2309.12499.
16. Gudi, S. R. (2024). 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, 5*(2), 144-149. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I2P116
17. S. Lehnert, "A Taxonomy for Software Change Impact Analysis," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, 2011, pp. 41–50, https://doi.org/10.1145/2024445.2024454.
18. Bandari, S. S. G., Sivva, S. D., & Thalakanti, R. R. (2024). 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, 5*(3), 139-147. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I3P115
19. Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "CodeT5+: Open Code Large Language Models for Code Understanding and Generation," arXiv preprint arXiv:2305.07922, 2023, https://arxiv.org/abs/2305.07922.
20. Gunda, S. K. G. (2023). The Future of Software Development and the Expanding Role of ML Models. *International Journal of Emerging Research in Engineering and Technology, 4*(2), 126-129. https://doi.org/10.63282/3050-922X.IJERET-V4I2P113
21. U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," *Proceedings of the ACM on Programming Languages, 3*(POPL), 2019, pp. 1–29, https://doi.org/10.1145/3290353.
22. Gudi, S. R. (2024). 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, 5*(3), 155-160. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I3P116
23. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys, 51*(4), 2018, pp. 1–37, https://doi.org/10.1145/3212695.
24. Sivva, S. D. (2023). 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, 4*(1), 600–604. https://doi.org/10.54660/.JFMR.2023.4.1.600-604
25. G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 43–52, https://doi.org/10.1145/1858996.1859006.
26. S. K. Gunda, "Comparative Analysis of Machine Learning Models for Software Defect Prediction," *2024 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS)*, Chennai, India, 2024, pp. 1-6, https://doi.org/10.1109/ICPECTS62210.2024.10780167.
27. A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004, pp. 214–223, https://doi.org/10.1109/WCRE.2004.10.
28. Balerao, M. (2023). A converged artificial intelligence architecture for innovation, software lifecycle optimization, and cybersecurity risk mitigation. *International Journal of Multidisciplinary Futuristic Development, 4*(1), 117–120. https://doi.org/10.54660/IJMFD.2023.4.1.117-120
29. S. Haque, T. Ahmed, and P. Devanbu, "Context-aware Code Summary Generation," arXiv preprint arXiv:2408.09006, 2024, https://arxiv.org/abs/2408.09006.
30. 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 5(1):160-6. https://ijaidsml.org/index.php/ijaidsml/article/view/366
31. S. K. Gunda, "Fault Prediction Unveiled: Analyzing the Effectiveness of Random Forest, Logistic Regression, and KNeighbors," *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, Erode, India, 2024, pp. 107-113, https://doi.org/10.1109/ICSSAS64001.2024.10760620.
32. "Enhancing Code Understanding for Impact Analysis by Combining Dependence Graph Information and Conceptual Coupling," *ACM Transactions on Software Engineering and Methodology*, 2024, https://doi.org/10.1145/3643770.