ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

Beyond Code Translation: A Systematic Review of Repository-Level Legacy Modernization

Syed Musadiq Illahi[1]; Sankalp Shankar[2]; Surabhi Singh[3]; Pravith S. Upadhyaya[4]; Abhishek Kumar[5]

[1]Department of AI&ML Acharya Institute of Technology Bangalore
[2]Department of AI&ML Acharya Institute of Technology Bangalore
[3]Department of AI&ML Acharya Institute of Technology Bangalore
[4]Department of AI&ML Acharya Institute of Technology Bangalore
[5]Department of AI&ML Acharya Institute of Technology Bangalore

Publication Date: 2026/11/12

Abstract:

- ***Background:***

Legacy code modernization is no longer about just translating one file into another language. Real software repositories contain dependencies between files and external services, hidden assumptions, and tests that must continue to work after migration. These repository-level relationships make modernization much harder of a task than simple code translation.

- ***Objective:***

This paper reviews graph-aware, recursive, and neuro-symbolic approaches for legacy modernization and examines how well they perform when applied to real-world software repositories.

- ***Methods:***

We reviewed English-language studies indexed in IEEE Xplore, the ACM Digital Library, and arXiv from January 2020 to April 2026, focusing on repositorylevel code translation, refactoring, and validation.

- ***Results:***

There are three recurring patterns. High-level repository understanding is enhanced by graph-based representations. LLMs can operate beyond a single context window thanks to recursive orchestration and when executability and contract preservation truly matter, pure generative techniques are outperformed by validation-heavy pipelines such as AlphaTrans and TRAM. Benchmark gains are not the same as deployment-ready because often benchmark suites underrepresent runtime variability, validation is costly, and graph extraction is frequently incomplete, failing to account for undocumented assumptions.

- ***Conclusions:***

No single approach solves everything. The conclusion is that all three layers—graph structure, orchestration technique, and deterministic checking— must be combined.

Keywords: *Legacy Modernization, Neuro-Symbolic AI, Graph Neural Networks, Recursive Language Models, Code Translation, Software Engineering.*

How to Cite: Syed Musadiq Illahi; Sankalp Shankar; Surabhi Singh; Pravith S. Upadhyaya; Abhishek Kumar (2026) Beyond Code Translation: A Systematic Review of Repository-Level Legacy Modernization. *International Journal of Innovative Science and Research Technology,* 11(10), 3211-3220. https://doi.org/10.38124/ijisrt/26oct385

I. INTRODUCTION

- *Motivation*

Critical systems are powered by legacy code, and maintaining behavior that is dispersed over several files, build configurations, tests, and undocumented conventions is more difficult to migrate than syntax. Banking, healthcare, government, and industrial applications continue to make extensive use of legacy software. Organizations cannot afford to lose the crucial business logic included in these systems.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

Because a system's behavior is dispersed over several files, interfaces, tests, and configurations, modernization is challenging. Small functions can be effectively translated by AI models such as translating only one single page from a given language to another language, but repository-level translations are still difficult since numerous dependencies need to be properly maintained as discussed [1]–[3].

The limitations of AI become clear when it is used on real software repositories instead of benchmark examples. Textbased prompting often fails to capture these relationships [4]– [6]. This is why recent work has moved toward graph-aware retrieval, repository-level planning, recursive orchestration, and neuro-symbolic validation—each addressing a part of the problem that the previous approach left open.

Early neural code models—CodeBERT, GraphCodeBERT, PLBART, CodeT5, UniXcoder—established that large-scale pre-training improves code search, summarization, and generation [7]–[11]. Generative systems like Codex, InCoder, StarCoder, AlphaCode, and Code Llama then showed that scaling further supports synthesis, repair, and infilling [12]–[16]. The catch is that nearly all of these were evaluated on function-level tasks—CodeXGLUE, HumanEval-style generation, issue resolution—not on full legacy modernization pipelines [12], [17], [18]. That gap is what this review is concerned with.

➢ *Scope and Research Questions*

This review focuses on three important technologies used in software modernization: repository graphs, staged orchestration, and deterministic verification. Repository graphs help represent relationships between files and functions. Staged orchestration breaks large modernization tasks into smaller steps. Deterministic verification checks whether the generated code is actually correct. We use these concepts to study how modern repository-level migration systems works. These are the questions guiding our analysis:

- What are the limitations of traditional Generative AI models when working with large repositories?
- How do graph aware representations of the repositories help the process and in which areas do they fall short?
- How do the neuro-symbolic framework ensure that the migrated software maintains both correct structure and correct behavior after modernization?
- How repository-level modernization systems should be evaluated?

Reliable evaluation should include compilation success, test execution, dependency preservation, runtime behavior, architectural design preservation, and detailed failure analysis. We also propose a taxonomy that classifies modernization methods based on repository representation, orchestration strategy, and validation depth.

➢ *Systematic Review Methodology*

To collect the studies for this review, we searched IEEE Xplore, the ACM Digital Library, and arXiv. The search process follows a PRISMA-style workflow adapted for software engineering evidence synthesis [19]. We used search terms related to repository-level translation, graph-based methods, neuro-symbolic validation, and legacy modernization. When a relevant paper cited another important study, we reviewed that work as well.

Our review focused mainly on research that evaluated complete software repositories. Most of the selected papers were peer-reviewed, although recent preprints were also included when they were directly related to repository-level modernization. We did not include blogs, tutorials, product pages, or studies that only demonstrated results on small code fragments. The final review is more qualitative rather than statistical. The selected papers differ in programming languages, repository sizes, evaluation methods, and experimental settings, so direct numerical comparison would not have been meaningful. Instead, we compared the studies based on their methods, evaluation strategy, strengths, and reported limitations. The final analysis was qualitative instead of statistical. The selected studies used various programming languages, repository sizes, and evaluation methods. Because of these differences, direct statistical comparison was difficult. Therefore, we compared the studies based on common themes, methods, strengths, and limitations.

II. THEORETICAL FOUNDATIONS: SOFTWARE AS A MATHEMATICAL GRAPH

Early modernization systems mostly treated code migration like a text conversion task. They focused on changing old code into new code without properly understanding the relationships inside the repository. However, software projects include a lot of interconnected files, functions, and dependencies [1], [2]. As a result, graph representations rather than plain text are used in current research. Graphs make it easier for models to comprehend the repository's structure.

➢ *Code Graphs and Structural Dependencies*

A software repository can be represented as a directed graph. In this graph, nodes represent files, classes, or functions, while edges represent relationships between them. These relationships may include function calls, inheritance, or data dependencies. AI systems may better comprehend repository structure and make judgments during modernization with the aid of such graph representations.

The paper discusses four commonly used graph structures. Abstract Syntax Trees (AST) captures the program syntax, Control Flow Graphs (CFG) shows the flow of execution of the code inside a program, Call Graphs show how all the functions call each other and Code Property Graphs (CPG) combine syntax, control flow, and dependency information into a single structure. In practice, modernization pipelines combine several of these rather than relying on any single graph [5], [6].

Graph-based software analysis is actually not a new concept. Program dependence graphs, control dependence graphs, and interprocedural slicing established how control

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

and data dependencies can be tracked across procedures [20]–[22]. Class hierarchy analysis and fast type analysis, two static call-graph techniques, demonstrated how object-oriented repositories may be modeled before execution. IFDS-style graph reachabil-ity made interprocedural dataflow practical at scale [23]–[25]. These representations were later made useful for vulnerability research and huge codebases by Soot and code property graphs [26], [27]. These concepts are used by newer systems, which instead focus on context selection, edit propagation, and validation rather than optimization or auditing.

Deterministic graph extraction matters because it creates a very stable representation of the repository before the code generation even begins. This helps in the system's understanding of repository structure and dependencies. After analyzing the repository, techniques like AlphaTrans and RepoGraph carry out translation or retrieval operations [5], [6]. As a result, there is less chance of inconsistent code modifications.

➢ *Graph Foundation Models and Attention Mechanisms*

When it comes to using graphs with language models, researchers have taken two different approaches. The first approach keeps the language model as it is and adds a graphbased retrieval system around it. The second approach changes the model itself so that it pays attention to graph structure while processing code [5], [28]. Both approaches have their own advantages and the choice between them affects both the cost of the system and where it tends to go wrong.

GOFA is an example of the first kind. It adds graph layers into an existing language model without changing the model's core weights [29]. CGM is closer to what modernization actually needs. It pushes repository graph structure directly into how the model attends to information and also uses graph-aware retrieval on top of that [28]. Both systems improve how well the model understands repository structure, but they both share the same weakness that is, if the graph they are built on is incomplete or wrong, the model's understanding will also be wrong. Thus, the bottleneck here is the creation of the graph and the amount of information we can add in it.

Table 1 Systematic Review Protocol Used in This Study

Aspect	Decision	Rationale
Time range	January 2020 to April 2026	Captures the LLM era while still covering earlier neural translation work
Databases	IEEE Xplore, ACM Digital Library, arXiv	Balances peer-reviewed software engineering papers with recent preprints in a fast-moving topic
Inclusion criteria Exclusion criteria Selection pipeline Synthesis target	English full text; repository-level translation, refactoring, or modernization; explicit graph, planning, recursive, or validation component Blogs, Medium posts, product pages, tutorial-style material, and snippet-only studies without repository context Identification → title/abstract screening → full-text eligibility review → thematic synthesis Method structure, evaluation design, and deployment trade-offs	Keeps the review focused on methods that operate beyond isolated code snippets Reduces weak evidence and removes sources with limited methodological detail Makes the review process explicit and reproducible Supports the taxonomy and the critical comparison developed later in the paper

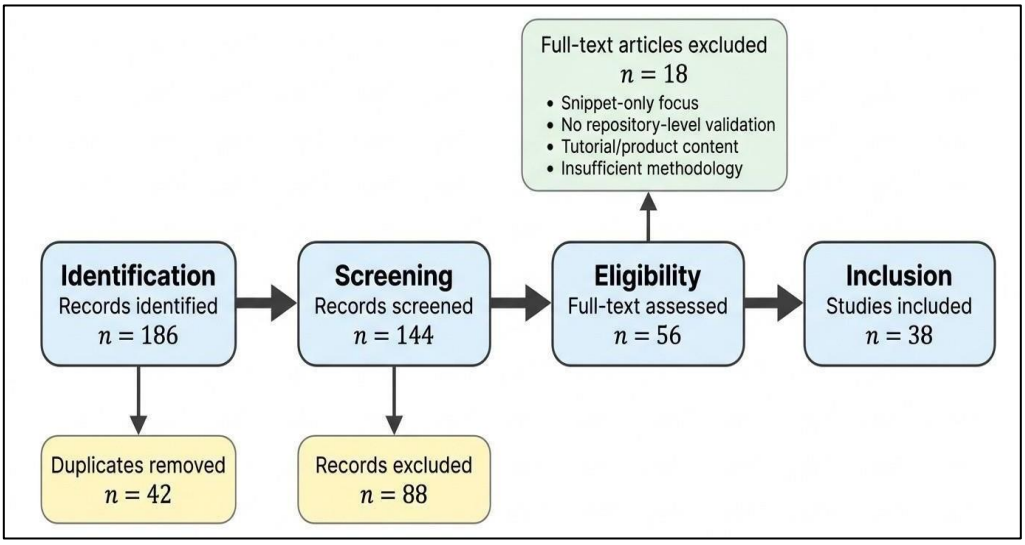


Fig 1 PRISMA-Style Study Selection Flow for the Literature Review. Of the 186 Records Identified, 38 Studies Met the Inclusion Criteria After Duplicate Removal, Screening, and Full-Text Assessment.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

CGM resolved 43% of issues on SWE-bench Lite using graph-aware retrieval [28]. The result suggests that repository graphs improve context selection. Their performance, however, still depends on how complete and accurate the underlying graph is.

Table 2 presents the taxonomy used throughout this review. The methods are grouped by the kind of repository representation they use, how updates are coordinated across files, and the level of validation applied during migration. Looking at the methods through these categories makes the differences between them easier to see.

➢ *Implications of Taxonomy*

Different projects need different modernization strategies. For smaller tasks where quick iteration matters more than perfection and the mistakes don't cost more, text-based and retrieval-heavy approaches are often enough. They work well when developers can review each generated changes. Therefore, they are intended for developer-assisted workflows, not automated production-grade migration pipelines.

How much verification you need depends on what a bad migration costs you. If rollback is limited or the software is safety-critical, the extra compute and cost for validationcentric neuro-symbolic pipelines is worth paying for.

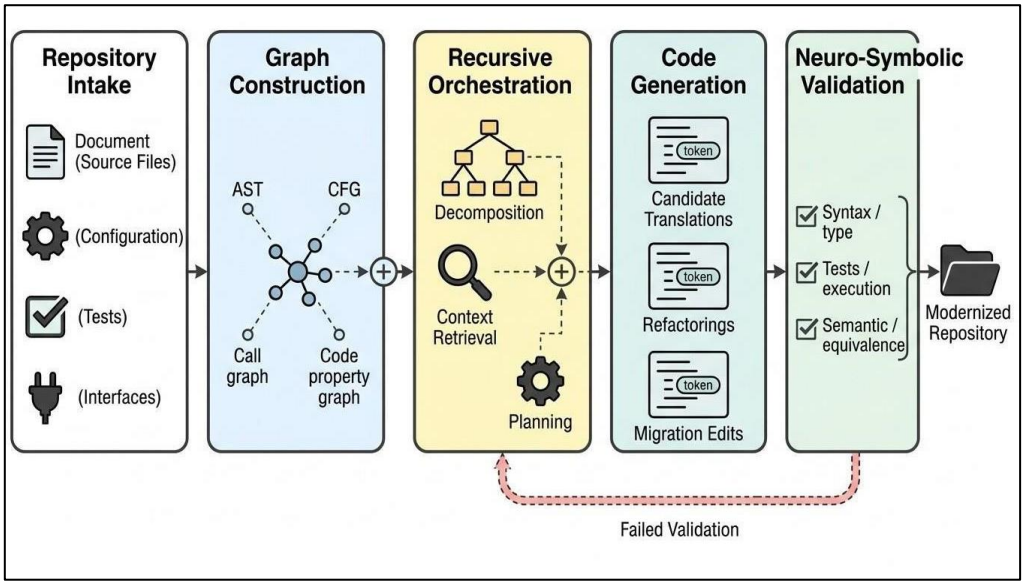


Fig 2 Conceptual pipeline illustrating the common stages found in repository-level modernization systems. A repository graph is built first, recursive orchestration decomposes the scope, code is generated, and validation checks types, interfaces, builds, and tests. Failed checks route back to graph updates and replanning.

Table 2 Proposed Taxonomy of Repository-Level Modernization Methods

Method family	**Representative systems**	**Repository representation**	**Primary validation style**	**Typical risk in real systems**
Text-centric	Direct repository	Flat text windows and	Compilation or tests	Cross-file inconsistencies and
prompting	prompting	ad hoc retrieval	after generation	poor context selection
Graph-guided	RepoGraph	External repository	Task-level execution	Missed runtime dependencies
retrieval		graph used for	or benchmark	if the graph is incomplete
		navigation and retrieval	harnesses	
Graph-integrated	CGM,	Graph structure fused	Benchmark execution	Expensive model adaptation
models	GOFA-inspired	into model attention or	with graph-aware	and graph-construction
	graph-language designs	hidden layers	context	overhead
Planning-driven	CodePlan	Incrementally updated	Consistency checks	Stale impact analysis can
editing		dependency graph plus	across derived edits	propagate wrong edits across
		edit obligations		files
Validation-centric	AlphaTrans, TRAM,	Program decomposition	Multi-stage syntactic,	Higher latency, stronger tooling
translation	MatchFixAgent	plus type and test	semantic, or	requirements, and partial

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

		artifacts	in-isolation validation	runtime coverage
Externalized	Recursive Language	Repository state kept	Tool-mediated	Orchestration bugs, unstable
recursive	Models	outside the base	decomposition and	decomposition, and
orchestration		context window	selective verification	environment dependence

But most migrations aren't that. When the goal is just recovering enough repository context to make good decisions, not proving strict behavioral equivalence, graph-guided retrieval or recursive orchestration gets you there for a lot less overhead. They fall short of full neuro-symbolic validation, but they still beat prompt-only approaches, because they give the model better structured context and a reasoning trail instead of just hoping it guesses right.

III. DEPENDENCY GRAPH CONSTRUCTION AND ANALYSIS IN PRIOR WORK

Most repository-level systems as we studied in our literature review, don't generate code first and reason afterward. They build a representation of the repository, then translate against it, this way an edit has something concrete to check against.

➢ *Deterministic Extraction Versus Neural Inference*

Here, static analysis is performed in both industry and academic systems, with the first approach being deterministic extraction and the second, neural inference. In IBM's quality evaluation pipeline for COBOL-to Java transformation, repository analysis is seen as a prerequisite to quality evaluation of generated code [30]. Similarly, AlphaTrans breaks up projects into pieces with program analysis prior to their translation and validation [6]. The principle is dependency extraction should be carried out prior to generation and not post failure.

Some newer systems take it one step further and combine static structure with proof of execution. A good example is SemanticForge, which integrates static and dynamic knowledge into a single graph, allowing for constraints to be imposed both at compile and run time, during code generation [31]. Useful when there are undocumented state changes and/or runtime coupling that are not captured in static graphs.

Table 3 Dependency Graph Update Mechanisms During Refactoring

Change Type	Update Mechanism	Propagation Action
Method Modification	Analyze new method signature; update parameter nodes and return type edges	Redirect Calls/CalledBy edges to maintain inheritance integrity
Method Deletion	Remove node for Method M and sever all incident edges	Flag all upstream callers for required refactoring
Field Modification Class Declaration	Add new node and edges representing updated field declaration Instantiate new class node; map object instantiation edges via constructors	Trace data flow to ensure downstream operations handle new type bounds Update dependency registry to reflect new modular boundaries

➢ *Change May-Impact Analysis and Graph Maintenance*

It is only the beginning of the construction of the graph. For repository level editing, such edit to the graph should also happen as the code is edited. Otherwise, later edits are based on an outdated context.

In CodePlan, this is done through "Change-impact reasoning". Instead of doing each edit individually, it generates subsequent edits based on the dependency graph, and subsequently uses the resulting obligations to maintain consistency in the repository [4]. This is essential for modernization, as it is possible that one interface change can lead to a chain of required changes.

A general idea is that you need to know how far a ripple effect extends to ensure that no interfaces are broken and no functions are left uncalled. In the literature, the dependency graph should be maintained as a true and up-to-date record of the event and not as a snapshot at the beginning of migration.

A. Topological Orchestration VIA Recursive Language Models

Dependency graphs help with structure awareness. They don't fix context scaling, still a repository is bigger than what fits in a single prompt of a modern day LLM.

Recursive Language Models (RLMs) solves this bottleneck by treating long context as an external environment instead of forcing it through one forward pass [32].

➢ *The Mechanics of Recursive Inference*

An RLM does not just consume a whole repository, but he/she reads it programatically. The model can examine files and break down the task into smaller components and make recursive calls of itself on those parts [32].

- Programmatic exploration: Examine the repositoryand determine target migration site.
- Task decomposition: decomposition of the migration problem into smaller, coherent units.
- Recursive solving: The system fragments an intimidating problem and tries to solve it in smaller and smaller parts. It delegates to smaller pieces to sub-tasks or revisits to

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

reconsider difficult subproblems, while maintaining control on the focus and the manageability.
- Assembly and verification: after those small pieces are solved it is all put together to form the main project. Last but not least, it performs specific tests to ensure that the newly added code integrates and operates well with the rest of the system.

➢ *Computational Advantages*

The true benefit here doesn't simply lie in its ability to deal with huge scale, it's about being completely in control. Recursive Language Models (RLMs) enable a system to intelligently determine when to read, summarize, delegate, or verify information. The original RLM paper points out that this approach can also control inputs up to two orders of magnitude higher than conventional context windows and does not incur a significant increase in costs relative to conventional longcontext approaches [32].

This is a lifesaver when you have to deal with legacy code migration. This is helpful if a project involves hundreds of different files, but you don't need all of them at a given point in the project.

There's a catch, though. The big drawback is that the context shift becomes a true systems engineering problem instead of a task of prompting. In the end your success depends on the quality of your tools, how easily you can decompose the codebase, and how easily you can reproduce all of that orchestration.

IV. NEURO-SYMBOLIC VALIDATION AND GLOBAL CONTRACT PRESERVATION

Translating syntax is the easy part. Keeping interface arity, data types, side effects, execution assumptions, etc. intact across an entire repository is the main problem.

This is where neuro-symbolic pipelines become useful. They combine generative models with deterministic checks so that the final decision is not left to a probabilistic decoder alone [6], [33], [34].

Table 4 Context-Management Strategies for Repository-Scale Modernization

Strategy	Strength	Where it helps	Typical failure mode
Direct long-context prompting	Minimum system engineering	Small repositories or local edits	Context amalgamation and weak cross-file consistency
Retrieval-augmented prompting	Effective targeted context	Repeated lookups over known files or APIs	Absent dependencies where retrieval keys are poor
Recursive orchestration (RLM-style)	Direct decomposition and extensible context handling	Large repositories with many loose paired subproblems	Tooling complexity, orchestration overhead, and reproducibility issues

➢ *Dual-State Architecture for Code Translation*

In these pipelines, generation and validation are separated. The model suggests code, and symbolic tooling determines if the suggested code is acceptable. The separation is the key difference from the conventional LLM generation approach.

The validation layer usually verifies 3 things: type compatibility (is the translated code type-compatible with the data types used by the calls?); interface compatibility (are the callers, callees, and data types compatible within the files?); and executable or symbolic validation (is the behavior still the same as the source code?).

➢ *Prominent Neuro-Symbolic Code Migration Frameworks*

The most obvious example to the repository level in this space is AlphaTrans. It is a new feature that can be used to back-translate software in reverse call order: from bottom to top, up the call graph [6].

They do not choose the order randomly. Mitigates the risk of generating upper layer modules against unstable or absent translated interfaces. AlphaTrans gets 99.1% syntactic correctness on 10 projects containing 6,899 translated fragments. However, the same paper reveals the reason that this number is deceptive: functional validation is much more difficult, and syntactic correctness doesn't necessarily mean behavior equivalence [6].

➢ *Iterative Synchronization Mechanisms*

These systems are iterative by design. A translated fragment changes the repository state, so later fragments must be checked against the updated interfaces and tests. That slows the pipeline—but it is often the price of consistency. The IBM WCA4Z pipeline makes the same point from an industrial angle: scalable modernization requires continuous quality checks, not a single pass/fail decision at the end [30].

A. Evaluation Frameworks and Multidimensional Benchmarking

Repository-level modernization cannot be evaluated with text-overlap metrics. Executability, functional equivalence, and structural consistency matter far more than lexical similarity [1], [2], [35].

➢ *Execution-Based and Repository-Level Benchmarks*

Evaluating repository-scale modernization needs test-driven, black-box environments that replicate real-world build complexity.

RepoTransBench is a key benchmark because it measures repository-level translation under executable evaluation. It includes 1,897 real-world samples across 13 language pairs and shows that full-repository translation is still fragile, especially when strict type recovery is required [2].

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

SWE-Refactor focuses on behavior-preserving refactoring. Its 1,099 validated instances show that compound, multi-file refactorings remain difficult even for strong models, which is important because modernization often contains refactoring and translation at the same time [3].

RepoMod-Bench extends this picture to code modernization with hidden, implementation-agnostic test suites. When scaling is done, it shows a significant drop in pass rates with respect to the growth of repositories. These is where support for strong decomposition and validation comes into picture [35].

➢ *Multidimensional Quality Metrics*

Three measures are used across all benchmarks:

- Build success: whether the translated repository compiles, links, and starts correctly.
- Functional equivalence: whether the translated repository matches the source repository in terms of behavior under the provided test cases.
- Structural fidelity: whether interfaces, dependency relations of one function with other, and modular boundaries remain intact after translation or refactoring.

Table 6 should be read as directional rather than comparing directly because the studies measure different endpoints. Still, we started noticing a pattern. Success rates starts to decline when evaluation starts requiring repository executability or behavioral preservation across the repository, unless the system makes planning and validation explicit.

B. Critical Comparison of Method Families

Benchmark scores are useful but don't explain much about production failures. We found these four failure modes come up again and again.

Table 5 Representative Validation-Oriented Modernization Frameworks

Framework	Core Integration Mechanism	Primary Innovation	Quality Assurance
AlphaTrans	Program analysis plus LLM translation	Reverse-call-order translation over repository fragments	Multi-level validation; strong syntactic accuracy, partial runtime validation
TRAM	Retrieval plus in-isolation mock validation	Resolves type mappings before fragment-level validation	Reduces dependence on whole-repository execution during validation
MatchFixAgent	Various agents included for validation and repairs pipeline	Separates equivalence analysis, testing, repair, and verdict generation	Produces high-coverage verdicts and repairs inequivalent translations
IBM WCA4Z evaluation pipeline	Analytic checkers with LLM-as-a-judge evaluation	Scalable quality evaluation for COBOL-to-Java transformation	Continuous reporting instead of a single end-of-run verdict
SemanticForge	Dual static-dynamic knowledge graph with constraint-guided search	Aligns graph queries and generation with executable constraints	Attempts stronger semantic control during generation itself

Table 6 Indicative Quantitative Comparison Across Representative Repository-Level Methods

Method	Benchmark / setting	Reported primary result	Main limitation
CodePlan	Repository-level package migration and temporal edits	5/6 repositories passed validity checks in the reported study	Small evaluation set; success rate depends on accurate and impactful analysis and planning state [4]
RepoGraph	SWE-bench and CrossCodeEval integration	Strong gains over four open-source baselines, yielding a new open-source state of the art in the paper's setup	Improvement depends on graphic completeness and benchmark-specific tooling [5]
CGM	SWE-bench Lite	43.0% resolution rate with graph-integrated context	Still below dependable enterprise-grade reliability; graphic features construction adds overhead [28]
AlphaTrans	10 Java-to-Python repositories	99.1% syntactic correctness and 25.8% runtime/functional validation success	Functional equivalence remains much harder than syntax preservation [6]
RepoTransBench baselines	1,897 repository-level translation samples	Performance degrades sharply on harder language pairs and type-sensitive tasks	Highlights initial difficulties more than a single winning method [2]
RepoMod-Bench	Hidden implementation-agnostic tests	Pass rates drop as repositories grow in size and coupling	Scaling makes fragile context selection and validation coverage visible [35]

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

➢ *Why Text-Only Systems Fail*

If the model cannot identify base level dependencies there is certainty that only prompt text is will fail. When the model misses one caller, one generated file, or one build rule, it is enough to make an otherwise plausible patch unusable. Both RepoTransBench and RepoMod-Bench shows that repository growth worsen it, it doesn't gets better [2], [35].

➢ *Trade-offs in Graph-Aware Methods*

Graph-aware methods can highly improve context selection of the existing code, but their performance depends highly on how accurately the repository graph represents the codebase. CGM and RepoGraph benefit from explicit repository struc-ture [5], [28]. When runtime behavior, reflection, configuration files, or generated code are not represented, the resulting re-trieval process can be misleading despite appearing confident.

In practice, graph-aware retrieval often reduces navigationrelated mistakes but continues to struggle with environment and configuration issues.

➢ *Trade-offs in Planning and Validation Pipelines*

Planning-based systems hold up well when a task creates a chain of dependent edits [4]. Their weakness is maintenance cost: the repository model must stay current as edits accu-mulate. Validation-heavy systems offers strong solution [6], [33], [34], but they need parsers, type recovery, mock envi-ronments, tests, and repair loops which makes them slower, but also much closer to what enterprise modernization actually requires.

➢ *Recursive Orchestration as a Systems Trade-off*

Although Recursive Orchestration transfers complexity from the model to the surrounding system, RLM-style orches-tration solves a real context problem [32]. The method breaks down labor into digestible chunks, which makes it effective for huge repositories. The catch is that actual deployments need to control reproducibility, recursion depth, sandbox state and tool execution. Recursive orchestration raises engineering pressure while decreasing prompt pressure.

Table 7 Critical Comparison of Method Families for Real-System Modernization

Method family	What it does well	Why it still fails in real systems	Best-fit use case
Text-only prompting Graph-guided retrieval Planning-driven editing	Effective local edits and simple translation work Better repository navigation and cross-file context selection Multiplies dependent edits in systematic manner	Misses hidden dependencies, environmental assumptions, and variance in file ripple effects Fails when extracted graphs omit dynamic behavior or stale repository state Derived edits can amplify early mistakes if impact analysis is incomplete	Small repositories or enclosed scoped edits Mid-scale repositories with reliable static features Refactoring-heavy modernization with explicit dependency maps
Validation-centric neuro-symbolic pipelines	Strong guardrails for syntactic and correct interpretation	High rate of gaps in between, higher tooling burden, and limited coverage when tests are inefficient	Safety-critical or business-critical migrations
Recursive orchestration	Enlarged reasoning across large repositories without one huge prompt	Dependency on efficient tools, good decomposition, and careful systems engineering	Large heterogeneous codebases with iterative modernization workflows

V. CURRENT LIMITATIONS AND FUTURE RESEARCH DIRECTIONS

The literature demonstrates significant advancements. However, a number of constraints prevent the field from achieving true production reliability.

➢ *Computational Overhead of Continuous Symbolic Verification*

The cost of validation is high. Overhead is increased by static analysis, graph maintenance, mock creation, and ex-ecutable validation, and this expense increases as reposito-ries expand. Stronger accuracy guarantees necessitate more robust infrastructure, as demonstrated by SemanticForge and AlphaTrans [6], [31]. The unresolved issue is how to allocate that verification budget solely in situations where it genuinely improves results.

➢ *Dynamic and Low-Resource Language Support*

The quality of graph varies by language. Java, C++, Go and other compiled, well typed ecosystems are good candidates for static analysis. For dynamic languages and specialized heritage languages with limited or insufficient tooling, it is substantially weaker. Type-sensitive migrations are nevertheless significantly less reliable than simpler translation directions, as RepoTransBench makes evident [2]. Current graph-aware techniques do not really fill this need.

➢ *Trajectory Toward Selective Autonomy*

Selective autonomy—a pipeline that selects the appropriate tool at each stage rather than a single model handling everything—is the most likely course of action. Graph-guided retrieval, planner-generated edit obligations, and targeted validation loops will probably be combined in future systems. Only when the surrounding tooling is sufficiently developed to be auditable and reproducible will recursive orchestration become more widespread [31], [32]. That remains an open engineering challenge at this time.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

VI. CONCLUSION

When systems see repositories as lengthy strings rather than structured software artifacts, legacy modernization fails. Although this framing seems simple, the majority of text-only techniques continue to make the same error.

A multifaceted understanding of the issue is supported by the evaluated studies. Models can find and maintain structure with the use of graphs. They can operate at repository scale without blowing up a single context window due to recur-sive orchestration. Neuro-symbolic validation distinguishes between code that behaves correctly and code that is reasonable. A single layer is insufficient on its own, and any article asserting otherwise should be viewed with suspicion.

The useful takeaway is that an approach that raises one benchmark score may not be prepared for business migration. The better articles in this field are those that demonstrate how modifications spread across files, reveal their failure mechanisms, and validate behavior using a more rigorous method than text similarity. Future work should be held to that standard.

REFERENCES

[1.] X. Chen, J. Xue, X. Xie, C. Liang, and X. Ju, "A systematic literature review on neural code translation," *arXiv preprint arXiv:2505.07425*, 2025.

[2.] Y. Pan, J. Zhao, C. Wang, and Q. Liu, "Repotransbench: A realworld benchmark for repository-level code translation," *arXiv preprint arXiv:2412.17744*, 2024.

[3.] Y. Xu, J. Yang, and T.-H. Chen, "SWE-refactor: A repository-level benchmark for real-world LLM-based code refactoring," *arXiv preprint arXiv:2602.03712*, 2026.

[4.] R. Bairi, A. Sonwane, A. Kanade, S. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "Codeplan: Repository-level coding using LLMs and planning," in *International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=d0A2pc2kFp

[5.] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu, "Repograph: Enhancing AI software engineering with repository-level code graph," in *International Conference on Learning Representations (ICLR)*, 2025. [Online]. Available: https: //openreview.net/forum?id=dw9VUsSHGB

[6.] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand, "Repository-level compositional code translation and validation," *arXiv preprint arXiv:2410.24117*, 2024.

[7.] 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*, 2020, pp. 1536–1547.

[8.] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pretraining code representations with data flow," in *International Conference on Learning Representations*, 2021.

[9.] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "PLBART: Unified pre-training for program understanding and generation," in *North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2113–2125.

[10.] Y. Wang, W. Wang, S. Joty, and S. C. 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.

[11.] D. Guo, S. Lu, N. Duan, L. Shou, M. Gong, and D. Jiang, "UniXcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[12.] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, G. Mishra, I. Sutskever *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[13.] D. Fried, A. Aghajanyan, J. Lin, S. I. Zhu, U. Alon, W. Choi, M. Chen, X. Hao, W. Y. Wang, K. Cho *et al.*, "InCoder: A generative model for code infilling and synthesis," *International Conference on Learning Representations*, 2023.

[14.] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: May the source be with you!" *Transactions on Machine Learning Research*, 2023.

[15.] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competitionlevel code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[16.] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Wang, P. G. Sessa, M. Caccia, Y. Bitton, M. Eliav, M. Khalifa, J. M. Rehg *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[17.] 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, and N. Duan, "CodeXGLUE: A machine

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26oct385

learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

[18.] C. E. Jimenez, J. Yang, A. Wettig, K. Lieret, S. Yao, K. Pei, O. Press, and J. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" *International Conference on Learning Representations*, 2024.

[19.] M. J. Page, J. E. McKenzie, P. M. Bossuyt, I. Boutron, T. C. Hoffmann, C. D. Mulrow, L. Shamseer, J. M. Tetzlaff, E. A. Akl, S. E. Brennan, R. Chou, J. Glanville, J. M. Grimshaw, A. Hrobjartsson, M. M. Lalu, T. Li, E. W. Loder, E. Mayo-Wilson, S. McDonald, L. A. McGuinness, J. A. Stewart, J. Thomas, A. C. Tricco, V. A. Welch, P. Whiting, and D. Moher, "The PRISMA 2020 statement: An updated guideline for reporting systematic reviews," *BMJ*, vol. 372, p. n71, 2021.

[20.] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program de-pendence graph and its use in optimization," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1984, pp. 298–305.

[21.] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.

[22.] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[23.] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, 1995, pp. 77– 101.

[24.] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1996, pp. 324–341.

[25.] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 49–61, 1995.

[26.] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot – a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999, p. 13.

[27.] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[28.] H. Tao, Y. Zhang, Z. Tang, H. Peng, X. Zhu, B. Liu, Y. Yang, Z. Zhang, Z. Xu, H. Zhang, L. Zhu, R. Wang, H. Yu, J. Li, and P. Di, "Code graph model (CGM): A graph-integrated large language model for repositorylevel software engineering tasks," *arXiv preprint arXiv:2505.16901*, 2025.

[29.] L. Kong, J. Feng, H. Liu, C. Huang, J. Huang, Y. Chen, and M. Zhang, "GOFA: A generative one-for-all model for joint graph language modeling," *arXiv preprint arXiv:2407.09709*, 2024.

[30.] S. Froimovich, R. Gal, W. Ibraheem, and A. Ziv, "Quality evaluation of COBOL to Java code transformation," *arXiv preprint arXiv:2507.23356*, 2025.

[31.] W. Zhang, C. Zhang, Z. Luo, J. Ma, W. Yuan, C. Gu, and C. Feng, "Semanticforge: Repository-level code generation through semantic knowledge graphs and constraint satisfaction," *arXiv preprint arXiv:2511.07584*, 2025.

[32.] A. Zhang, S. Karp, N. Setlur, H. Wang, N. Malkin, T. Hashimoto, and C. Tomlin, "Recursive language models," *arXiv preprint arXiv:2512.24601*, 2025.

[33.] K. Ke, A. R. Ibrahimzada, R. Pan, S. Sinha, and R. Jabbarvand, "Advancing automated in-isolation validation in repository-level code translation," *arXiv preprint arXiv:2511.21878*, 2025.

[34.] A. R. Ibrahimzada, B. Paulsen, R. Jabbarvand, J. Dodds, and D. Kroening, "Matchfixagent: Language-agnostic autonomous repository-level code translation validation and repair," *arXiv preprint arXiv:2509.16187*, 2025.

[35.] X. Li, N. Ben-Israel, Y. Raz, B. Ahmed, D. Serebro, and A. Raux, "Repomod-bench: A benchmark for code repository modernization via implementation-agnostic testing," *arXiv preprint arXiv:2602.22518*, 2026.