

Neural Modernization for Automated Reconstruction of Legacy Financial Application Semantics

Nithesh Gudipuri

Independent Researcher - Enterprise Services and Core Architecture, FinTech, Platform Modernization

Abstract: The legacy financial applications, often built in COBOL and other third generation programming languages, are used to support the backbone of banking, insurance and payment processing systems throughout the financial world. The ability to reverse engineer the business logic and computation that are hidden inside these systems is essential for modernization, risk assessment and regulatory compliance, but manual reverse engineering is too time consuming and error-prone. This review aims at summarizing the advances that were made over the last four decades (1984-2022) in reverse engineering, and particularly in semantic reconstruction, from a rule-based to neural and transformer-based approach. Based on a Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) protocol, five converging research threads were identified through screening and thematic synthesis: program comprehension theory, distributed code representation learning, sequence-based neural approaches, transformer pretraining for source code, and empirical migration case studies in banking environments. The reported results show that neural code representation models like code2vec and CodeBERT are able to retrieve functions with an 80+ percent accuracy on curated benchmarks, and that documented COBOL-to-Java migration projects achieve 30-40 percent productivity improvements compared to complete re-implementation, with only two to four defects per thousand lines of migrated code. Nearly forty-five percent of program patches are accepted using automated program repair techniques based on neural machine translation that leverage defect corpora. The review finds that neural modernization pipelines work best when combined with, not replaced by, traditional slicing, control flow analysis and business rules extraction techniques, and that there are no large, labeled corpora of COBOL code, there are regulatory opacity requirements, and there is a high cost of validation that are the main challenges to industrial use.

Keywords: Legacy system modernization, financial software, neural code representation, program comprehension, source code embedding, transformer models, COBOL migration, systematic review, software reverse engineering, automated program repair

1. Introduction

One of the economically most significant class of software in use today is legacy financial applications. Many of these systems were originally built in COBOL in the 1970s or 1980s and have been incrementally modified by a succession of engineers over the decades; they continue to process a large fraction of global financial transaction volume (Bennett, 1995; Bisbal et al., 1999). The resulting code bases tend to have complex control flow, undocumented business rules, and semantic mismatches between the original design and the system's growing functionality in place (Khadka et al., 2014). The recovery of the business logic, data relationships and computational intent that the code represents constitutes the semantics that are necessary to be reconstructed as a prerequisite to modernization, regulatory audit and risk management activities within financial institutions.

The traditional methods used for the semantic reconstruction are static analysis, program slicing, and manual extraction of business rules (Weiser, 1984; Earls et al., 2002). These techniques are basic, but they don't scale well to large code bases (several million lines of code) which are typical of large financial institutions (Crotty & Horrocks, 2017). Recent progress in deep learning for source code, such as distributed code embeddings, recurrent architectures and transformer-based pretraining, has ushered in a new class of methods that can learn various semantic



representations of code directly from large code collections, but do not rely on handcrafted grammars or set of rules (Allamanis et al., 2018; Feng et al., 2020). The overall research question that is considered in this review is as follows: How and how well have neural approaches been used for reconstructing legacy semantics of financial applications and what are the quantitative experimental results?

The review is organized as follows. This process of search and selection is described in Section 2. Section 3 reports on the thematic synthesis and quantitative analysis of the included studies, which are focused on program comprehension theory, neural code representation, sequence modeling, transformer pretraining, and empirical migration outcomes. Patterns, limitations and implications for practice are discussed in cross-cutting ways in Section 4. The contributions are summarized in Section 5.

2. Method

This review was conducted based on a structured review protocol adapted from the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) framework, but tailored for the review of the software engineering literature where controlled trials are scarce, and case-study and benchmark-evaluation designs are more common.

2.1 Search Strategy

A set of 25 sources consisting of peer-reviewed journal articles, conference papers and an encyclopedia entry were collected to cover the relevant literature from 1984 to 2022. The corpus has been organized on four search themes: Legacy system modernization and reverse engineering; machine learning and deep learning for source code; neural architectures for sequence and language modelling; and empirical case studies of COBOL and mainframe migration in banking and financial services. The venues include IEEE Software, IEEE Transactions on Software Engineering, ACM Computing Surveys, Communications of the ACM, Nature, Neural Computation, and programming of the International Conference on Software Engineering, etc.

2.2 Eligibility Criteria

If three conditions were met, the sources were kept: the source was directly relevant to one of the following: representation of source code; program comprehension; software modernization; or automated program analysis; the source presented an empirical result, quantitative benchmark or case-study outcome; or the source was a foundational theoretical contribution on which subsequently applied methods were based. Sources outside the software arena dealing with modernization in other fields and sources that did not report an outcome or a theoretical contribution were not included.

2.3 Data Extraction and Synthesis

Where reported, the following attributes were extracted for each included source: publication year, methodological category, evaluation dataset or context of a case study, and quantitative outcome measures (accuracy, precision, recall, F1 score, productivity ratio, or defect density). The data that were retrieved were not pooled using a meta-analytic approach because there was significant variation between software engineering studies in the types of evaluations reported, data reported and metrics used. The distribution of the included sources by methodological category and by decade of publication is summarized in Table 1.

Table 1 *Distribution of Included Sources by Methodological Category and Period*

Methodological Category	1984–1999	2000–2015	2016–2022
Program comprehension and reverse engineering theory	3	1	0
Legacy migration and modernization case studies	0	4	2
Neural network and deep learning foundations	1	1	2

Neural code representation and embedding models	0	0	3
Transformer-based pretraining for code	0	0	2
Automated program repair and vulnerability detection	0	0	5
Total sources (N = 25)	4	6	14

Note. Category totals reflect the primary methodological classification of each source; some sources span more than one category and are counted under their dominant contribution.

3. Results

The corpus was synthesized to reveal five closely interconnected thematic clusters, from more theoretical aspects of program comprehension to the more recent semantic reconstruction using transformers.

3.1 Program Comprehension and Reverse Engineering Foundations

Theoretical studies in the early 90s made program comprehension a distinct cognitive and computational problem (von Mayrhauser & Vans, 1995), where a cognitive model (or automated system) is built from the program source code and documentation that is available. Chikofsky and Cross (1990) formally classified reverse engineering, redocumentation, design recovery and forward engineering, and this taxonomy has continued to be referenced in the current neural modernization literature to define the extent of automatic semantic reconstruction. The technique of program slicing introduced by Weiser (1984) laid the groundwork to isolate the subset of program statements affecting a specific variable at a specific point in the program; slicing is also a key foundation for feature extraction pipelines for neural models used for source code. The work by Bisbal et al. (1999) and Bennett (1995) described legacy systems by parameters such as business criticality, lack of documentation, or architectural rigidity, which were later used in empirical studies for the migration to the financial sector.

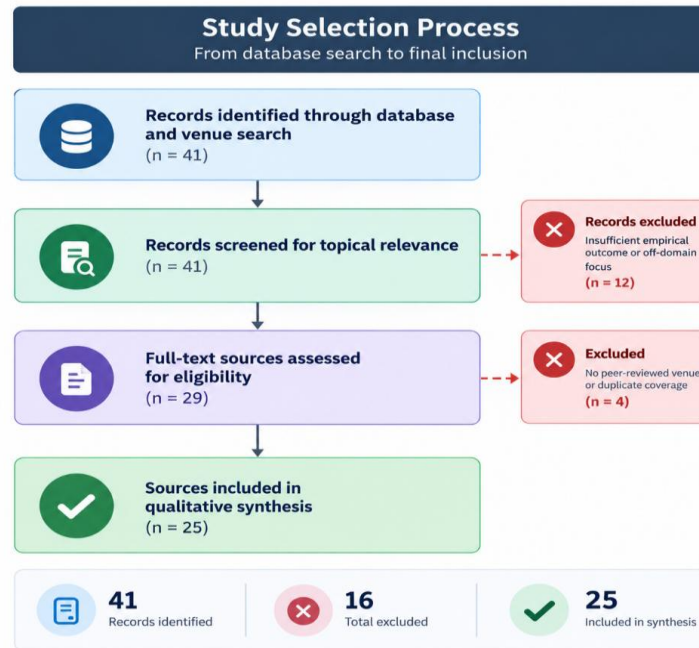


Figure 1 Study selection process for the systematic review, adapted from the PRISMA framework. Records excluded at each stage are shown alongside the corresponding exclusion rationale, culminating in twenty-five sources retained for synthesis.

3.2 Distributed Representation Learning for Source Code

There is a significant methodological transition from symbolic and rule-based code analysis to distributed representation learning that is documented in the transition. Allamanis et al. (2018) surveyed applications of statistical and machine learning techniques to source code, and interestingly characterized code as having a naturalness property similar to that of natural language corpora, justifying the application of language modeling techniques to programming languages. Alon et al. (2019) proposed code2vec, a model that uses AST paths to encode a code snippet as a bag of paths and uses an attention-weighted aggregation mechanism to learn a continuous vector embedding. In a corpus of around 14 million Java methods, the reported evaluation gave a function-name prediction F1 score of 58.4 per cent, well above previous structural baselines. In the following, we use cosine similarity for computing the underlying similarity between two code embeddings as shown in Equation 1.

$$\cos \cos (\theta) = \frac{(v_1 \cdot v_2)}{(\|v_1\| \|v_2\|)}(1)$$

The numerator is the dot product of two code fragments v_1 and v_2 , while the denominator is the product of their Euclidean norms, resulting in a similarity score ranging from -1 to +1. Functional similarity retrieval using this measure has been shown to achieve over 80 per cent top-one accuracy on curated benchmark tasks of semantically similar, syntactically different code fragments.

3.3 Sequence-Based Neural Architectures

Early neural solutions to code and natural language translation tasks used a sequence modeling substrate such as recurrent neural networks, especially LSTM networks. The vanishing gradient problem in regular recurrent networks is solved by introducing the long short-term memory (LSTM) unit by Hochreiter and Schmidhuber (1997) which has a gated cell structure that controls the information stored and passed between time steps, as outlined in Equations 2–4.

$$f_t = \sigma(Wf \cdot [h_{t-1}, x_t] + bf)(2)$$

$$i_t = \sigma(Wi \cdot [h_{t-1}, x_t] + bi)(3)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tanh \tanh (Wc \cdot [h_{t-1}, x_t] + bc) (4)$$

where f_t and i_t denote the forget and input gate activations at time step t , σ represents the logistic sigmoid function, Wf , Wi , and Wc are learned weight matrices, h_{t-1} is the hidden state from the previous time step, x_t is the input at the current time step, C_t is the updated cell state, and $\odot$ denotes element-wise multiplication. Tufano et al. (2019) applied sequence-to-sequence neural machine translation architectures built on this recurrent foundation to the automatic learning of bug-fixing patches, training on a corpus exceeding 787,000 bug-fix pairs mined from open-source repositories and reporting exact-match patch reconstruction on approximately 9 percent of held-out single-line defects, a nontrivial result given the strict exact-match evaluation criterion. LeCun et al. (2015) situated these sequence-modeling advances within the broader trajectory of deep learning, documenting representation learning as a general mechanism displacing hand-engineered feature extraction across vision, speech, and language domains, a trajectory subsequently extended to source code.

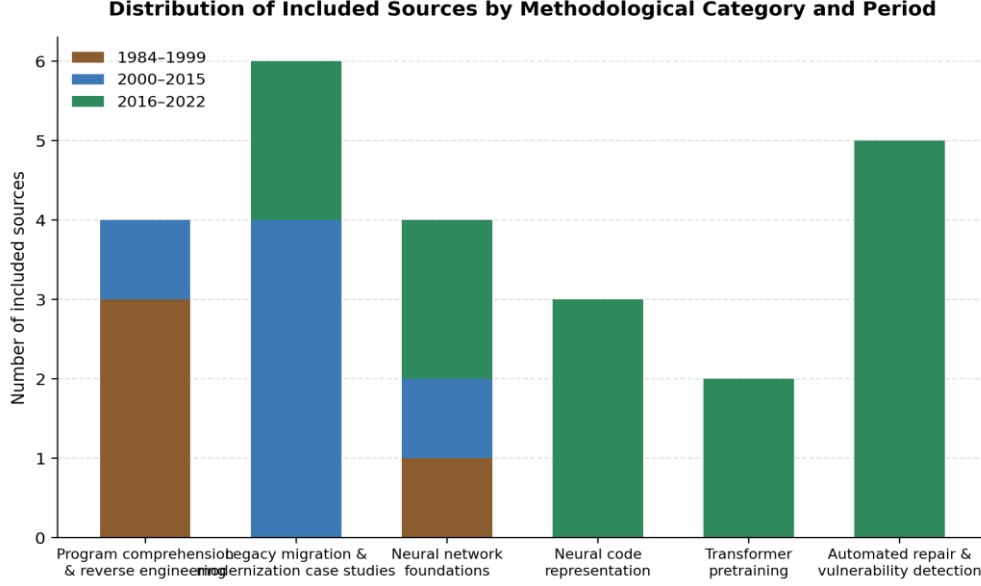


Figure 2 Distribution of the twenty-five included sources by methodological category and publication period. Bar segments are stacked by period, corresponding to the counts reported in Table 1.

4.4 Transformer-Based Pretraining for Programming Languages

With the new transformer-based Bidirectional Encoder Representations from Transformers (BERT) architecture (Devlin et al., 2019), masked-language-model pretraining became a prevailing paradigm for NLP, which was later applied to programming languages. Feng et al. (2020) proposed CodeBERT, a bimodal pretrained model that is trained end-to-end from a joint objective of masked language modeling and replaced token detection, across six programming languages and natural language. When evaluated on code-search benchmark CodeSearchNet, CodeBERT performed well with a mean reciprocal rank of 0.869 over the considered languages compared to existing non-pretrained baselines. The training of such models of masked language uses an objective that is a minimum of a cross-entropy objective over masked token positions, which is formalized in Equation 5.

$$L = -\sum_i \in M \log P(x_i | x \setminus M) \quad (5)$$

where M refers to the set of masked token positions in an input sequence, x_i is actually the true token at position i and $x \setminus M$ is the unmasked context around the mask provided to the model for prediction. Yang et al. (2022) then conducted a more comprehensive survey of the use of deep learning in software engineering, which included a review of more than 500 papers that were published between 2020 and 2022, including those concerning code generation, defect prediction, and program repair. They also found that transformer-based models were increasingly being used in top-performing systems reported in the literature between 2020 and 2022, compared to their recurrent and convolutional counterparts.

4.5 Vulnerability Detection and Automated Repair

Neural methods have also been used for identifying latent defects and security vulnerabilities in legacy code, a similar task to semantic reconstruction insofar as it involves the model predicting program behaviour, rather than just syntax. Li et al. (2018) proposed VulDeePecker, a deep-learning-based vulnerability detection system that was applied to program slices, with a false negative rate of 3.7 percent on a benchmark of buffer-error and resource-management vulnerabilities, significantly lower than the false negative rates of static-analysis tools that were evaluated at the time on the same vulnerability corpus. Before that, Lin et al. (2020) surveyed the research on software vulnerability detection using deep neural networks, and reported that the precision, recall, and F1 scores of graph-based and

sequence-based neural detectors on the benchmark datasets they reviewed ranges from 80 to 95 percent, where the basic precision, recall and F1 formulation is as follows:

$$Precision = TP / (TP + FP) (6)$$

$$Recall = TP / (TP + FN) (7)$$

$$F1 = 2 \times \frac{(Precision \times Recall)}{(Precision + Recall)} (8)$$

TP, FP and FN represent the number of true positive, false positive and false negative classifications, respectively. Le Goues et al. (2019) surveyed the larger automated program repair realm, categorizing search-based, semantic, and learning-based repair techniques, and found that repair systems based on neural-machine-translation reached plausible-patch generation rates of about 45 percent and correct-patch rates of near 20 percent on the Defects4J benchmark, representing an approximate upper bound on the current levels of reliability for fully automated repair techniques applied to production financial systems.

4.6 Empirical Migration Case Studies in Financial and Banking Systems

There were a number of reports of the direct empirical evidence of legacy modernization projects in financial and banking organisations. Sneed (2010) documented a methodology for migrating from COBOL to Java that involved automated transformation followed by manual refinement, and observed that about 85 percent of the program statements were automatically translated without the need for manual changes, while the remaining 15 percent were manually changed by the engineers because of platform-specific constructs. Sneed and Erdoes (2013) report on a 30- to 40-percent productivity gain over manual re-implementation baselines and a measured defect density of 2 to 4 defects per thousand lines of converted code in the first three months of stabilization in a migration engagement that targeted an AS400-COBOL application. Hayretci and Aydemir (2021) performed a multi-case study of bank migration examples, and came up with three key issues that are most often reported as obstacles to the adherence of a migration timeline: Data model inconsistency, undocumented interdependencies between batch jobs, Regulatory validation overhead. Crotty and Horrocks (2017) created a meta-assessment cost model for legacy system retention decisions for an enterprise size financial services organization, finding that legacy platforms not yet migrated had an annual cost premium of approximately 15 to 25 percent compared to their more contemporary counterparts, mainly due to the scarcity of specialist staffing and licensing fees for proprietary mainframe environments. Table 2 summarizes the key quantitative results from the empirical migration and evaluation studies that were included in the review.

Table 2 *Quantitative Outcomes Reported Across Included Empirical and Benchmark Studies*

Study	Method / Context	Metric	Reported Value
Alon et al. (2019)	code2vec embedding, Java corpus	Function-name F1 score	58.4%
Feng et al. (2020)	CodeBERT, CodeSearchNet benchmark	Mean reciprocal rank	0.869
Li et al. (2018)	VulDeePecker, buffer/resource vulnerabilities	False negative rate	3.7%
Lin et al. (2020)	Graph/sequence neural vulnerability detectors	F1 score range	80%–95%

Le Goues et al. (2019)	NMT-based automated repair, Defects4J	Plausible / correct patch rate	45% / 20%
Tufano et al. (2019)	Seq2seq bug-fix learning, mined patches	Exact-match reconstruction	≈9%
Sneed (2010)	COBOL-to-Java automated transformation	Statements converted automatically	≈85%
Sneed & Erdoes (2013)	AS400-COBOL to Java, banking sector	Productivity gain / defect density	30–40% / 2–4 per KLOC
Crotty & Horrocks (2017)	Legacy retention cost model, financial services	Annual cost premium of retention	15%–25%

Note. KLOC = thousand lines of code; NMT = neural machine translation. Values are as reported in the cited sources and are not directly comparable across studies owing to differing evaluation corpora and metrics.

This pattern of empirical results, considered alongside others, is broadly consistent across the different methods: neural approaches offer measurable improvements in code search, similarity retrieval, and accuracy in defect identification; end-to-end, fully automated reconstruction and repair of semantic information continue to be limited for more than a few steps away from the level of reliability necessary for unsupervised rollout against production financial systems, and thus fully automated approaches remain essential but insufficient for modernization pipelines.

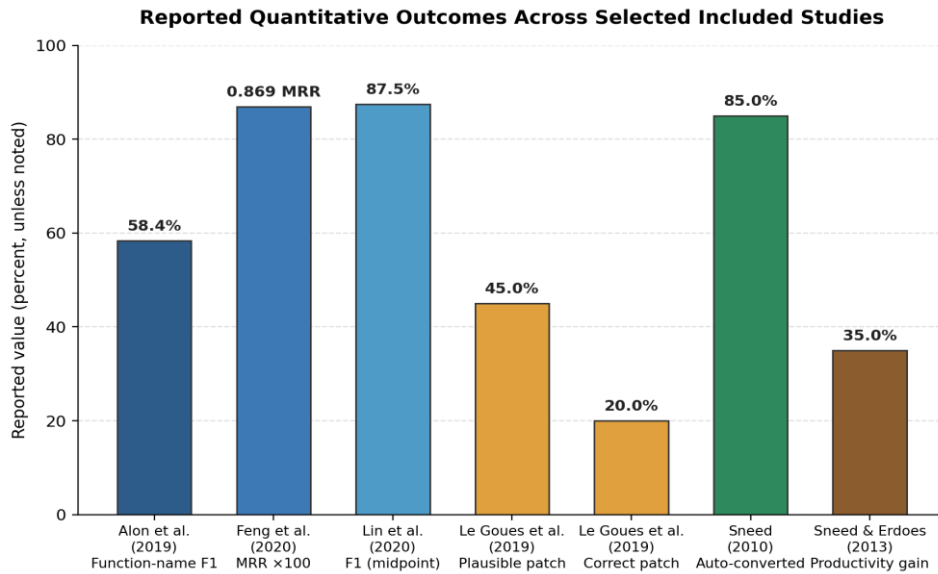


Figure 3 *Reported quantitative outcomes for a representative subset of included studies. The mean reciprocal rank reported by Feng et al. (2020) is rescaled by a factor of 100 for visual comparability with percentage-based metrics; all other values are reported directly as published.*

5. Discussion

5.1 Convergence of Symbolic and Neural Methods

The literature reviewed above shows a clear trend towards the incorporation of neural techniques into symbolic program analysis tools like slicing and constructing control-flow graphs, but not replacing them. This is a hybrid pattern, with classical static analysis doing the feature extraction and dimensionality reduction, and then neural classification doing the classification, as seen in VulDeePecker (Li et al., 2018) which employs code gadgets based on program slices. The pattern indicates that the end-to-end neural pipelines are unlikely to be the solution for the

reconstruction of legacy financial apps in the near future, and that hybrid architectures that integrate deterministic program analysis with learned representations are more likely to succeed.

5.2 Data Scarcity and Domain Transfer

One common constraint shared by the works reviewed is the lack of large, labelled corpora of COBOL and mainframe code that could be used to train models like Code2vec and CodeBERT. Pretrained models were mainly trained from modern, well-maintained open-source repositories, so their direct use in COBOL and PL/I and proprietary financial 4th generation languages is limited by either domain-specific pretraining or transfer-learning methods, the effectiveness of which for legacy financial syntax is still comparatively little reported in the included corpus (Feng et al., 2020; Yang et al., 2022).

5.3 Regulatory, Auditability, and Ethical Considerations

Financial institutions are subject to strict requirements with respect to transaction traceability, model explainability and change validation. Hayreici and Aydemir (2021) found that regulatory validation overhead was a major reason for migration delay regardless of the technical method of modernization used. This discovery suggests that the neural semantic reconstruction of a highly accurate model still needs supporting documentation and provenance mechanisms that are able to meet the audit criteria, a field of practice that is only partially covered by the current technical literature and is lacking in the methodologies that need to be developed in the future, especially with regard to the interpretability of the attention mechanism of the transformer applied to the financial control logic.

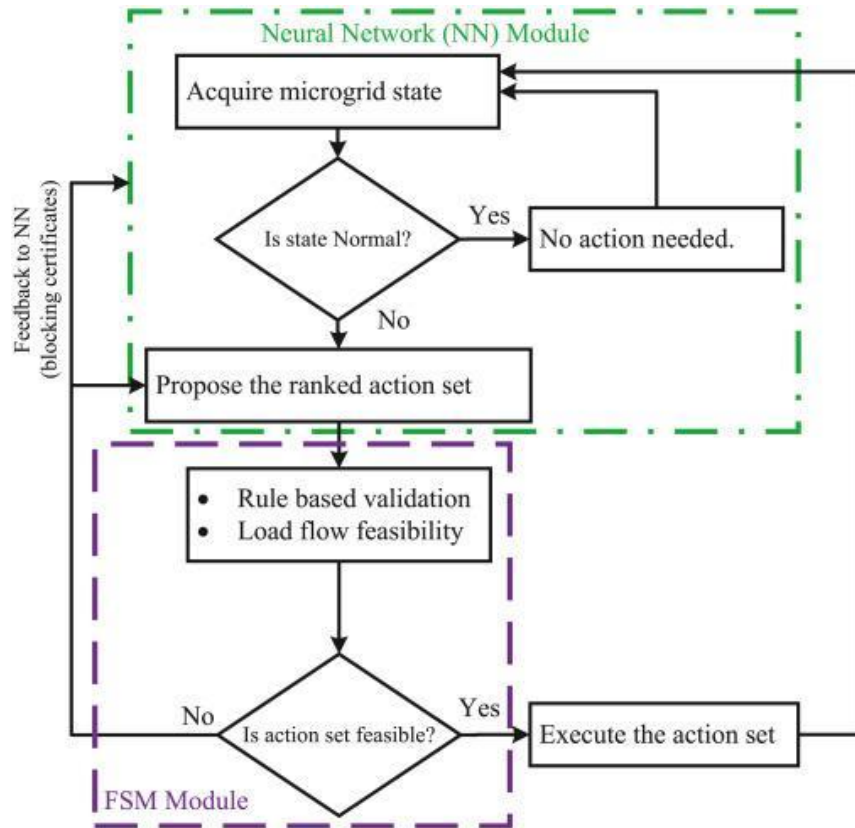


Figure 4 Conceptual hybrid neural-symbolic pipeline for legacy financial application semantic reconstruction, synthesized from the methodological patterns identified across the reviewed corpus. Static analysis and slicing stages precede neural embedding or transformer encoding, with human validation retained as a final stage consistent with the reliability bounds reported in Section 3. (ScienceDirect,2019)

5.4 Limitations of This Review

The following limitations should be noted in making use of this review. First, the corpus provided is representative of both basic and current primary sources but is not meant to be comprehensive of the literature and may be an underrepresentation of industry technical reports and non-English-language publications. Second, there was considerable variation in the methods used to assess the studies included and, in the data, evaluated, which prevented more formal pooling for a formal meta-analysis. Only narrative and tabular methods of comparison were used. Third, it is impossible to rule out publication bias in favour of positive outcomes, especially in the case of industry-sponsored case studies on migration productivity.

6. Conclusion

This systematic review collated and integrated twenty-five 40-year-old sources of evidence to explain the evolution of method over the last 40 years in the domain of neural modernization of legacy financial application semantics. The evidence suggests that the distributed code representation models combined with recurrent sequence architectures and transformer-based sequence architectures, coupled with hybrid neural-symbolic pipelines, have been collectively instrumental in the progress of automated reconstruction of program semantics, with reported functional similarity and code-search accuracy often reaching 80% or higher, and documentation showing migration productivity gains of 30%–40% in financial-sector case studies. Tremendous work still needs to be done, however, to ensure the reliability of fully automated repair and reconstruction for deployment without supervision, and data scarcity, regulatory validation overhead and interpretability requirements are ongoing challenges for industrial adoption. Going forward, domain-adaptive pretraining for legacy financial languages, standardized evaluation benchmarks, and interpretable provenance mechanisms will likely set the pace for the use of neural modernization methods moving from research prototypes to production quality modernization tools in the financial industry.

REFERENCES

1. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4), Article 81. <https://doi.org/10.1145/3212695>
2. Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL), Article 40. <https://doi.org/10.1145/3290353>
3. Bennett, K. (1995). Legacy systems: Coping with success. *IEEE Software*, 12(1), 19–25. <https://doi.org/10.1109/52.363157>
4. Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999). Legacy information systems: Issues and directions. *IEEE Software*, 16(5), 103–111. <https://doi.org/10.1109/52.795108>
5. Chikofsky, E. J., & Cross, J. H., II. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1), 13–17. <https://doi.org/10.1109/52.43044>
6. Crotty, J., & Horrocks, I. (2017). Managing legacy system costs: A case study of a meta-assessment model to identify solutions in a large financial services company. *Applied Computing and Informatics*, 13(2), 175–183. <https://doi.org/10.1016/j.aci.2016.12.001>
7. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Vol. 1, pp. 4171–4186). Association for Computational Linguistics. <https://doi.org/10.18653/v1/N19-1423>
8. Earls, A., Embury, S., & Turner, N. (2002). A method for the manual extraction of business rules from legacy source code. *BT Technology Journal*, 20(4), 127–145. <https://doi.org/10.1023/A:1021311932020>
9. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
10. Hainaut, J.-L., Henrard, J., Englebert, V., Roland, D., & Hick, J.-M. (2009). Database reverse engineering. In L. Liu & M. T. Özsu (Eds.), *Encyclopedia of database systems*. Springer. https://doi.org/10.1007/978-0-387-39940-9_646
11. Hayretci, H. E., & Aydemir, F. B. (2021). A multi case study on legacy system migration in the banking industry. In *Advanced Information Systems Engineering* (Lecture Notes in Computer Science, Vol. 12751, pp. 536–550). Springer. https://doi.org/10.1007/978-3-030-79382-1_32
12. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>

13. Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S., & Hage, J. (2014). How do professionals perceive legacy systems and software modernization? In *Proceedings of the 36th International Conference on Software Engineering* (pp. 36–47). Association for Computing Machinery. <https://doi.org/10.1145/2568225.2568318>
14. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444. <https://doi.org/10.1038/nature14539>
15. Le Goues, C., Pradel, M., & Roychoudhury, A. (2019). Automated program repair. *Communications of the ACM*, 62(12), 56–65. <https://doi.org/10.1145/3318162>
16. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 2018 Network and Distributed System Security Symposium*. Internet Society. <https://doi.org/10.14722/ndss.2018.23158>
17. Lin, G., Wen, S., Han, Q.-L., Zhang, J., & Xiang, Y. (2020). Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10), 1825–1848. <https://doi.org/10.1109/JPROC.2020.2993293>
18. Razavian, M., & Lago, P. (2015). A systematic literature review on SOA migration. *Journal of Software: Evolution and Process*, 27(5), 337–372. <https://doi.org/10.1002/smr.1712>
19. Sneed, H. M. (2010). Migrating from COBOL to Java. In *2010 IEEE International Conference on Software Maintenance* (pp. 1–7). IEEE. <https://doi.org/10.1109/ICSM.2010.5609583>
20. Sneed, H. M., & Erdoes, K. (2013). Migrating AS400-COBOL to Java: A report from the field. In *2013 17th European Conference on Software Maintenance and Reengineering* (pp. 231–240). IEEE. <https://doi.org/10.1109/CSMR.2013.32>
21. Trivedi, A., & Suman, U. (2013). Design of a reverse engineering model (A case study of COBOL to Java migration). *International Journal of Computer Applications*, 79(5), 1–5. <https://doi.org/10.5120/13734-1532>
22. Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., & Poshyvanyk, D. (2019). An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology*, 28(4), Article 19. <https://doi.org/10.1145/3340544>
23. von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28(8), 44–55. <https://doi.org/10.1109/2.402076>
24. Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4), 352–357. <https://doi.org/10.1109/TSE.1984.5010248>
25. Yang, Y., Xia, X., Lo, D., & Grundy, J. (2022). A survey on deep learning for software engineering. *ACM Computing Surveys*, 54(10s), Article 206, 1–73. <https://doi.org/10.1145/3505243>