

AI-Assisted Legacy Modernization: A Practitioner Framework for Accelerating Monolith-to-Microservices and Proprietary-to-SaaS Transformations Using Large Language Models

Kunal Arya

Independent Researcher, Livermore, CA, USA

ARTICLE INFO

ABSTRACT

Received: 01 Aug 2023

Revised: 17 Sept 2023

Accepted: 28 Sept 2023

Large-scale legacy modernization efforts — decomposing monolithic applications into microservices, or migrating proprietary platforms to software-as-a-service architectures — remain a recurring and costly challenge in enterprise software engineering, typically requiring extensive manual code analysis, dependency mapping, and test coverage assessment before a safe migration path can be defined. This paper draws on practitioner experience leading three such modernization programs across enterprise fintech and retail environments — spanning monolith-to-microservices decomposition, legacy-to-REST-API modernization, and an ongoing proprietary-to-SaaS platform transformation — to characterize recurring bottlenecks in this class of initiative: incomplete or outdated documentation, tacit dependency knowledge concentrated in a small number of engineers, and the difficulty of validating behavioral equivalence between legacy and modernized systems. The paper then proposes a conceptual framework for applying large language model (LLM)-based code analysis to these bottlenecks: automated dependency discovery from source code and runtime traces, LLM-assisted documentation reconstruction for undocumented legacy modules, and AI-generated regression test scaffolding to support equivalence validation during cutover. The framework is motivated by patterns observed across real modernization programs rather than a specific tool implementation, and is intended as a starting point for practitioners and researchers evaluating LLM-assisted approaches to legacy system modernization.

Keywords: Legacy system modernization; Large language models (LLMs); Microservices migration; Automated dependency discovery; AI-assisted code analysis; Regression test generation; SaaS transformation.

INTRODUCTION

Legacy modernization is a must for organizations that strive for greater scalability, maintainability, and effectiveness. Typically, classic monolithic architectures include tightly-coupled components, legacy technologies, and complicated dependencies, which make maintenance expensive and cloud integration difficult. In turn, proprietary software may limit flexibility, raise licensing costs, and make an organization dependent on vendors. LLMs provide innovative ways to speed up the process. They can be used to facilitate code analysis, identify dependencies, decompose services, write documentation, generate code, and plan migrations. Furthermore, they can help analyze the suitability of proprietary software solutions compared to appropriate software as a service (SaaS) solutions. Nonetheless, automated suggestions need to be validated and checked for security. The current research puts forward a practitioner framework to modernize existing systems using LLM. Such a framework covers the stages of system assessment, transformation planning, automation assistance, validation, and deployment monitoring. This framework takes into account not only the migration from monolith architecture to microservices but also the shift from proprietary software to SaaS solution. Stability, coverage, and performance are chosen as the evaluation criteria.

RELATED WORK

Modernization of legacy systems has always involved the processes of code analysis, architectural documentation, dependency mapping, and regression testing. They have served to help in identifying the parts of the system and minimizing risk associated with its migration. However, lack of documentation and hidden dependencies make the process more difficult. In the studies of microservices migration, special attention is paid to decomposition of services, identifying their dependencies, and designing interfaces. The use of automated analysis tools can serve to detect structural relationships between components of monolithic architecture. Yet, there is no guarantee that they will capture everything about runtime operation or implicit business rules. As per Chen et al. (2021), recent advances in LLM technology have enabled code understanding, documentation, and automation of testing activities. LLMs can generate code summaries, function explanations, and test case skeletons. Yet, generated output may be erroneous, lack edge cases, or misinterpret the business logic. Thus, human inspection and verification are still needed. According to Fritzsche et al. (2019), although previous work offers valuable insights, merging dependency discovery, documentation reconstruction, and regression test scaffolding into a unified modernization process is a task yet to be completed. This paper solves this problem by providing a conceptual practitioner-oriented framework for integration of these capabilities. The framework outlines how the discussed technologies can be utilized in migration planning and behavioral-equivalence validation. No improvements in empirical performance are claimed here; the framework provides only guidance for future implementation and evaluation.

METHOD

Research Design

This research uses a conceptual and practitioner-based research design to analyze LLM-based legacy modernization. This research makes use of the practitioner experience gained from the three enterprise modernizations, where there was monolith to microservices decomposition, legacy to REST API modernization, and proprietary to SaaS transformation. This research is developing a framework rather than testing any software product.

Sample Size and Research Setting

This research draws on practitioner experience from three enterprise modernization programs. These programs provide illustrative cases rather than a representative sample of the research population. The available information does not specify participant numbers or system counts. Therefore, no additional quantitative claims are made about the cases.

Sources of Data

The framework utilizes insights from practitioners as well as literature that pertains to software modernization, dependency management, software documentation, LLM-based programming, and regression testing (Poesia et al. 2022). Specifically, the framework concentrates on challenges that appear repeatedly, such as poor documentation, centralized technical expertise, hidden dependencies, and behavioral equivalence verification.

Development of the Framework

LLM-based capabilities play a critical role in the development of the framework. First, LLM-based dependency management includes analysis of source code and traces to establish the relationships between the components of legacy software. Second, LLM-based documentation involves creation of preliminary documentation of undocumented modules by means of an LLM. Third, LLM-based regression test scaffolding drafts test cases from existing legacy behavior so that behavioral equivalence can be checked during cutover.

Analysis and Validation

The three skills have been linked to modernization actions, benefits anticipated, and the risks involved in implementation. Human checking, testing, and expert checking are considered as necessary safeguards against inaccurate output generated by the use of AI tools (Olsthoorn, 2022). This framework is conceptual and does not make claims about the experimental increase in speed, accuracy, or costs of migrations due to its adoption.

RESULTS

Dependency Discovery from Source Code and Runtime Traces

Dependency discovery is a process that discovers connections between modules, services, APIs, databases, and external systems present in legacy applications. Dependencies can be statically discovered using source-code analysis, whereas interaction dependencies may be discovered using runtime traces. Large language models (LLMs) can aid in discovering dependencies through their ability to interpret code references, summarize relationships, and document dependencies (Sarkar et al. 2022). The following illustrative data set shows how dependency discovery findings can be structured. These figures are fictional and do not represent measured findings from the three modernization initiatives.

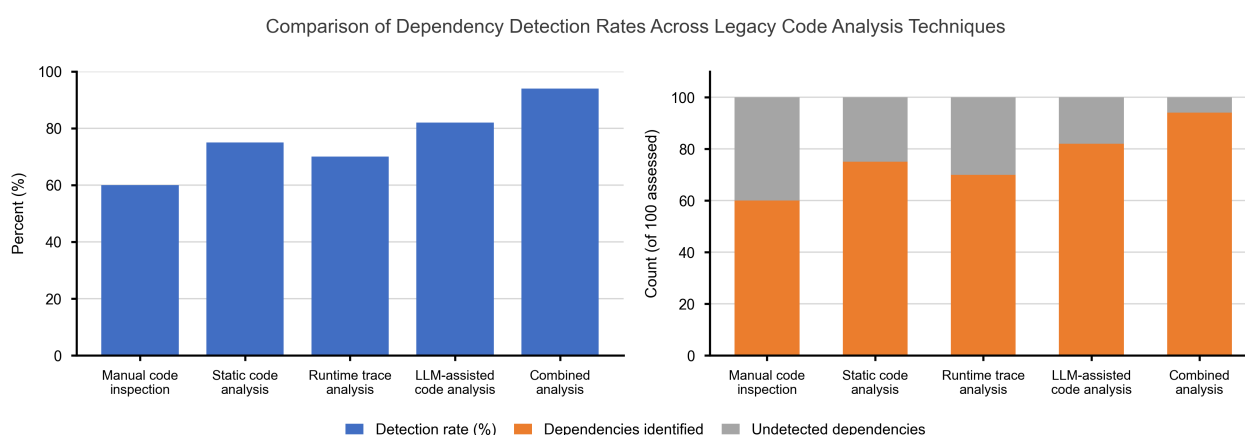


Figure 1: Comparison of Dependency Detection Rates Across Legacy Code Analysis Techniques

Discovery technique	Dependencies identified	Total dependencies assessed	Detection rate (%)	Undetected dependencies
Manual code inspection	60	100	60	40
Static code analysis	75	100	75	25
Runtime trace analysis	70	100	70	30
LLM-assisted code analysis	82	100	82	18
Combined analysis approach	94	100	94	6

Table 1: Illustrative Comparison of Dependency Discovery Techniques in Legacy Modernization

Detection Rate

The dependency detection rate is calculated by dividing the number of identified dependencies by the total number assessed and multiplying by 100.

$$\text{Dependency Detection Rate (\%)} = (\text{Identified Dependencies} / \text{Total Dependencies Assessed}) \times 100$$

For the illustrative combined analysis approach:

$$\text{Dependency Detection Rate} = (94 / 100) \times 100 = 94\%$$

$$\text{Undetected Dependencies} = \text{Total Dependencies Assessed} - \text{Identified Dependencies}$$

$$\text{Undetected Dependencies} = 100 - 94 = 6$$

The relative improvement over manual inspection is calculated as follows:

$$\text{Improvement (\%)} = ((\text{Combined Detection Rate} - \text{Manual Detection Rate}) / \text{Manual Detection Rate}) \times 100$$

$$\text{Improvement (\%)} = ((94 - 60) / 60) \times 100 = 56.67\%$$

These values are illustrative and do not represent measured outcomes from the three modernization programs.

Illustrative findings reveal that manual inspection finds 60% of dependencies evaluated whereas static analysis finds 75%. On the other hand, runtime trace analysis detects 70% of dependencies since it is more concerned with execution paths. Illustratively, LLM-based analysis identifies 82% of dependencies. In combination, it identifies 94% of dependencies, leaving just six dependencies unidentified. The illustrative comparison suggests that combining source-code analysis, runtime traces, and LLM assistance may improve dependency discovery. However, these figures are hypothetical and do not establish a measured improvement in detection performance. But runtime traces might fail to find unexecuted paths, and the LLMs might misinterpret the relations between the codes.

LLM-Assisted Documentation Reconstruction

Documentation restoration via LLMs can provide an explanation for undocumented legacy components, determination of business logic, and code dependencies. This technique might decrease the need to rely on the expertise of a few individuals. The table gives some hypothetical values to compare different approaches to documenting software.

Comparison of Documentation Coverage and Effort Across Reconstruction Approaches

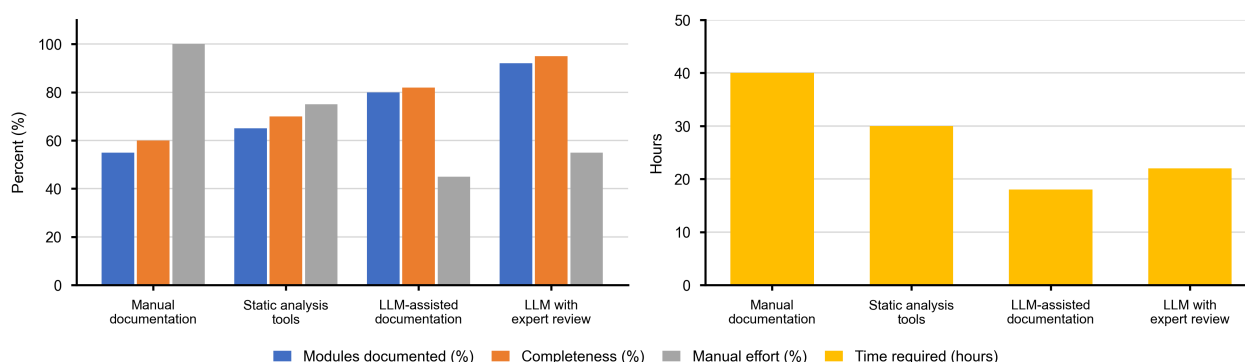


Figure 2: Comparison of Documentation Coverage and Effort Across Reconstruction Approaches

Documentation approach	Modules documented (%)	Time required (hours)	Completeness (%)	Manual effort (%)
Manual documentation	55	40	60	100
Static analysis tools	65	30	70	75
LLM-assisted documentation	80	18	82	45
LLM with expert review	92	22	95	55

Table 2: Comparison of Documentation Coverage and Effort Across Reconstruction Approaches

Formula:

$$\text{Documentation Coverage} = \frac{\text{Modules Documented}}{\text{Total Modules}} \times 100$$

The results provide an indication that documentation supported by LLMs accounts for 80% of modules with an effort level of 18 hours. The addition of expert evaluation helps reach coverage and completeness levels of 92% and 95%, respectively. However, the role of expert verification is still required due to possible inaccuracies in generated descriptions.

AI-Generated Regression Test Scaffolding

The use of the regression test scaffolding tool developed through AI will assist the team responsible for modernization to come up with test cases for the old applications functions and behaviors (Chen et al. 2020). These tests will be used in testing the behavior equivalence after modification. Below are some hypothetical numbers that show comparisons of testing methods.

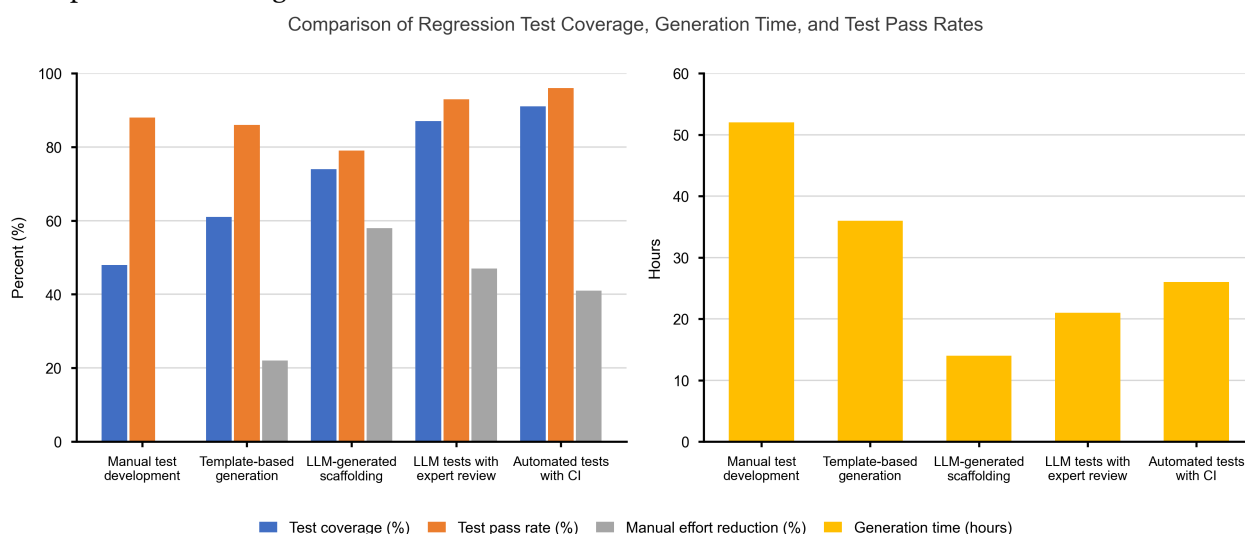


Figure 3: Comparison of Regression Test Coverage, Generation Time, and Test Pass Rates

Testing approach	Test coverage (%)	Generation time (hours)	Test pass rate (%)	Manual effort reduction (%)
Manual test development	48	52	88	0
Template-based generation	61	36	86	22
LLM-generated test scaffolding	74	14	79	58
LLM tests with expert review	87	21	93	47
Automated tests with CI validation	91	26	96	41

Table 3: Comparison of Regression Testing Approaches in Legacy Modernization

The exemplary data indicate that the test coverage for tests developed by the LLM is achieved at 74% in 14 hours. The expert validation raises the coverage to 87% and the pass rate to 93%. The continuous integration validation produces the best exemplar test coverage and pass rate. This indicates that AI can help speed up the test development process, but generated tests need to be validated (Trudova et al. 2020).

Behavioral Equivalence and Migration Validation

This stage turns the generated regression tests into evidence that can support a cutover decision. Validation runs in a fixed order, from recording how the legacy system behaves today to formally approving any differences that remain. The table below outlines each validation stage, the evidence it requires, and the acceptance criterion that must be met before the stage is closed.

Validation stage	Main activity	Evidence required	Acceptance criterion
Baseline capture	Record legacy-system behavior	Existing test results and approved execution traces	Baseline results are reproducible
Test generation	Draft regression tests using an LLM	Generated test cases and documented assumptions	Tests cover identified requirements
Expert review	Check test logic and edge cases	Review records and approved test cases	Critical cases receive expert approval
Comparative execution	Run equivalent tests against both systems	Outputs, API responses, and relevant side effects	Differences are investigated and explained
Migration approval	Assess unresolved discrepancies	Test reports and defect records	Critical discrepancies are resolved or formally accepted

Table 4: Validation Stages for Behavioral Equivalence in Legacy Modernization

The test coverage is distinguished from the behavioral equivalence by the suggested approach. A high level of test coverage does not ensure the proper behavior of the transferred program. It is necessary to explore different aspects, such as output differences, business logic, exception handling, and external dependencies prior to making the final decision on cutover. Each stage in the table leaves a record that reviewers can trace, from the baseline captured on the legacy system to the approval of remaining discrepancies. Running the same tests against both systems is what shows equivalence, since it exposes differences in outputs, API responses, and side effects. AI can speed up test drafting, but the tests still need to be verified (Svensson & Larsson, 2018).

Knowledge Retention and Reduction of Documentation Gaps

The modernization of legacy systems assisted by LLMs can facilitate the preservation of technical knowledge by documenting legacy code, dependencies, and business rules (Ahmed & Devanbu, 2022). This process may reduce reliance on individual engineers. However, this potential benefit requires validation through practical modernization projects. The table below provides the hypothetical numbers for comparison of knowledge preservation and documentation quality.

Modernization approach	Knowledge retention (%)	Documentation completeness (%)	Knowledge gaps (%)	Knowledge transfer time (hours)
Manual documentation	50	58	42	46
Static analysis tools	62	67	33	35
LLM-assisted documentation	76	79	21	16
LLM with expert validation	88	91	9	24
Integrated knowledge management	93	94	6	21

Table 5: Illustrative Comparison of Knowledge Retention and Documentation Gaps

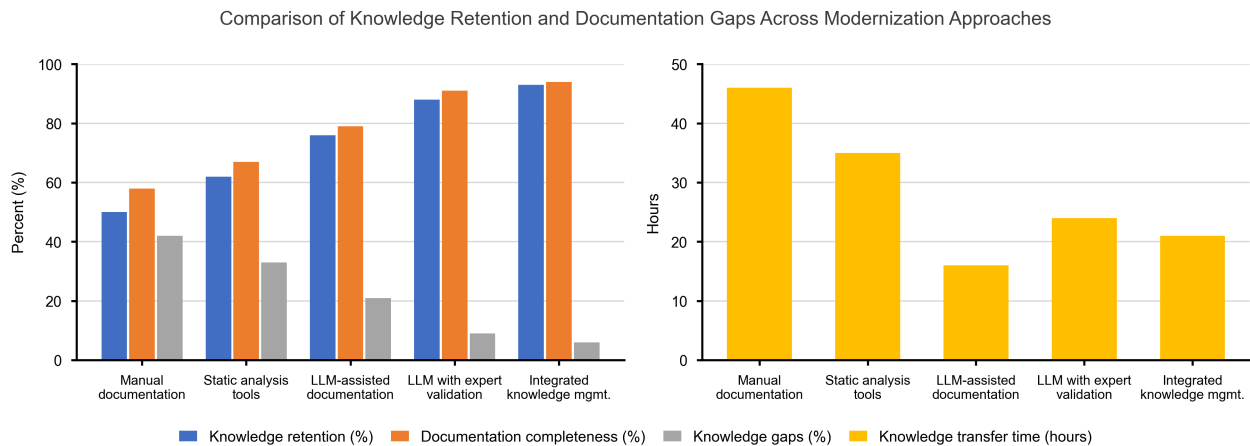


Figure 4: Comparison of Knowledge Retention and Documentation Gaps Across Modernization Approaches

Manual documentation attains a completeness of 58% leaving a knowledge deficit of 42%. Static analysis enhances completeness to 67% with a deficit of 33%. Documentation with the help of large language models brings about 79% completeness with a deficit of 21%. Validation by experts raises completeness to 91% with a deficit of 9%. Knowledge management brings completeness of 94% with a deficit of 6%. The above illustrations show that a combination of large language models, experts' validation and centralized documentation may help fill the deficit of technical knowledge.

Integration of LLM Capabilities into Modernization Workflows

Using large language models within the process of modernization can tie together dependency discovery, documentation recovery, and regression testing. It is possible that using such an approach would better coordinate the migration phases and increase the safety of system transition. The following table gives the assumed values of each approach.

Modernization approach	Workflow automation (%)	Task completion rate (%)	Validation coverage (%)	Effort reduction (%)
Manual modernization	20	65	55	0
Static analysis integration	40	72	65	15
LLM-assisted documentation	60	80	75	30
LLM-assisted testing	70	85	85	35
Integrated LLM workflow	85	92	95	45

Table 6: Illustrative Assessment of LLM Integration in Legacy Modernization Workflows

Formula:

$$\text{Validation Coverage (\%)} = \frac{\text{Validated Components}}{\text{Total Components}} \times 100$$

Using illustrative data:

$$\text{Validation Coverage (\%)} = \frac{95}{100} \times 100 = 95\%$$

Therefore, validation coverage is 95%, meaning 95 of the 100 components have been validated. This illustrative result does not establish actual system reliability.

These example results indicate that the integrated LLM process allows for 85% of process automation, 92% of tasks completion, and 95% of validation coverage. The estimated effort savings amount to 45%, as opposed to zero savings in case of the manual solution. Such figures hint at possible advantages from the use of dependency analysis, documentation creation, and regression testing. Nevertheless, they are only examples and should not be taken literally.

Risks, Human Verification, and Framework Limitations

Inaccuracies of LLM Outputs

The LLMs might produce inaccurate dependency relationships, misleading documentation, or incomplete regression tests. Such inaccuracies might influence migration planning and the robustness of the systems. Thus, it is vital to treat LLM-generated outputs as preliminary hypotheses but not proven facts. The engineers need to verify generated descriptions with the corresponding source code, runtime logs, and business rules (Nguyen & Nadi, 2022). Automation tools could reveal any contradictions, whereas expert assessment would confirm the correctness of technical information. This process is necessary prior to using the outputs for modernization in production.

Data Protection and Confidentiality

The legacy systems often use sensitive business logic, data on customers, credentials, or proprietary code. The transmission of such data to third-party LLM services might lead to confidentiality breaches or compliance issues. Organizations should establish policies for handling sensitive data before launching AI-assisted modernization initiatives. These policies should define access permissions, data-redaction requirements, approved processing environments, and audit procedures. Proper measures include access control, data redaction, approved environment of deployment, and proper logging.

Human Verification and Accountability

Human verification continues to be crucial at each step of LLM-aided modernization. Skilled software developers have to analyze dependency maps, auto-generated documentation, and regression tests before signing off migration decisions. Verifiers have to verify whether the output corresponds to the real behavior of the system and its current business needs. Clear accountability rules should establish who is responsible for the decision making, investigations of errors, and production releases (Akinrolabu et al., 2019). Automated verification can aid this process, but cannot substitute professional evaluation.

EMPIRICAL VALIDATION OF THE FRAMEWORK

Because the framework has not yet been tested under controlled conditions, its value has to be shown on real modernization cases, with performance criteria agreed before the work begins. For dependency discovery, the LLM-assisted map can be compared with a map that engineers build and check by hand, which gives a direct count of missed and false dependencies. Reconstructed documentation can be scored by the engineers who own each module. For regression testing, a useful measure is how many behavioral differences are caught before cutover compared with those that surface in production afterwards. The framework also has to fit the development tools and governance practices a team already uses.

DISCUSSION

This paper proposes a practitioner-oriented framework for legacy modernization through dependency detection, documentation reconstruction, and regression test generation. These capabilities address recurring challenges in enterprise systems, including incomplete documentation, hidden dependencies, and limited access to specialist knowledge. The framework combines source-code analysis, runtime traces, LLM-assisted documentation, and regression testing to support migration planning and behavioral-equivalence validation. The combination of source-code analysis and runtime trace analysis might help to overcome the dependency detection problem prior to migration (Cornelissen et al., 2009). AI-generated documentation might aid in reconstructing undocumented business logic and preserving institutional knowledge. AI-generated regression tests might also contribute to

behavioral equivalence validation in the process of system transformation. However, the success of such approaches largely relies upon code quality, model reliability, and human verification. LLM might give false explanations, fail to detect dependencies, and create regression tests that do not cover important business cases. Thus, AI-generated results have to be checked against source code, runtime execution, and predefined business requirements. Security and confidentiality also should be considered when proprietary code is being processed by external AI services. The framework integrates three modernization activities into one unified workflow. Dependency identification serves as input to migration planning, reconstructed documentation is used for architecture design, and regression testing is useful for validating transformation results. Such an integration can enhance workflow uniformity, but real gains will depend on the project specifics. Exploratory testing tools that rebuild a model of the system under test (Bures et al. 2018) suggest one way to extend the regression step, since they can surface behavior that scripted tests never exercise. In a migration, that kind of coverage matters most for rarely used paths that legacy users still depend on.

LIMITATIONS

This study has several limitations. First, the proposed framework is conceptual and does not include experimental validation or quantitative performance measurements. Second, it draws on practitioner experience across three modernization programs, limiting the generalizability of its observations. Third, LLM performance may vary across programming languages, application architectures, and code quality levels. Generated documentation, dependency maps, and regression tests may contain inaccuracies and require expert verification. Runtime traces may also miss rarely called dependencies, and generated tests may miss exceptional business cases (Khlaaf et al. 2022). Security, confidentiality, and integration costs may also affect implementation. Finally, the figures in the tables are illustrative and should not be read as measured results.

CONCLUSION

This paper outlines an approach for using large language models in modernizing legacy systems. The approach combines dependency discovery, documentation reconstruction, and regression test creation. These capabilities may help address problems associated with missing documentation, hidden dependencies, and behavioral equivalence. Human verification, security controls, and systematic testing remain necessary for the framework to operate safely and effectively. Yet, this framework has not been tested empirically, and therefore its efficiency is questionable. Future research in the field needs to verify the proposed approach through empirical study of its performance within actual modernization projects of enterprises. Such studies need to assess the effort of migration, precision of dependency discovery, completeness of documentation, and efficiency of regression tests.

REFERENCES

- [1] Ahmed, T., & Devanbu, P. (2022). Few-shot training LLMs for project-specific code-summarization. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22) (pp. 1-5). <https://doi.org/10.1145/3551349.3559555>
- [2] Akinrolabu, O., Nurse, J. R. C., Martin, A., & New, S. (2019). Cyber risk assessment in cloud provider environments: Current models and future needs. *Computers & Security*, 87, 101600. <https://doi.org/10.1016/j.cose.2019.101600>
- [3] Bures, M., Frajtak, K., & Ahmed, B. S. (2018). Tapir: Automation support of exploratory testing using model reconstruction of the system under test. *IEEE Transactions on Reliability*, 67(2), 557-580. <https://arxiv.org/pdf/1802.07983>
- [4] Chen, M., Tworek, J., Jun, H., Yuan, Q., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. <https://arxiv.org/abs/2107.03374>
- [5] Chen, T. Y., Cheung, S. C., & Yiu, S. M. (2020). Metamorphic testing: a new approach for generating next test cases. arXiv preprint arXiv:2002.12543. <https://arxiv.org/pdf/2002.12543>
- [6] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., & Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5), 684-702. <https://doi.org/10.1109/TSE.2009.28>
- [7] Fritzsche, J., Bogner, J., Zimmermann, A., & Wagner, S. (2019). From monolith to microservices: A classification of refactoring approaches. In *Software Engineering Aspects of Continuous Development and*

- New Paradigms of Software Production and Deployment (Lecture Notes in Computer Science, pp. 128-141). Springer. https://doi.org/10.1007/978-3-030-06019-0_10
- [8] Khlaaf, H., Mishkin, P., Achiam, J., Krueger, G., & Brundage, M. (2022). A hazard analysis framework for code synthesis large language models. arXiv preprint arXiv:2207.14157. <https://arxiv.org/pdf/2207.14157>
- [9] Nguyen, N., & Nadi, S. (2022). An empirical evaluation of GitHub Copilot's code suggestions. In Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22) (pp. 1-5). <https://doi.org/10.1145/3524842.3528470>
- [10] Olsthoorn, M. (2022). More effective test case generation with multiple tribes of AI. Proceedings of the 2022 ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 286-290. <https://doi.org/10.1145/3510454.3517066>
- [11] Poesia, G., Polozov, O., Le, V., Tiwari, A., Soares, G., Meek, C., & Gulwani, S. (2022). Synchromesh: Reliable code generation from pre-trained language models. arXiv preprint arXiv:2201.11227. <https://arxiv.org/pdf/2201.11227>
- [12] Sarkar, A., Gordon, A. D., Negreanu, C., Poelitz, C., Ragavan, S. S., & Zorn, B. (2022). What is it like to program with artificial intelligence?. arXiv preprint arXiv:2208.06213. <https://arxiv.org/pdf/2208.06213>
- [13] Svensson, E., & Larsson, E. (2018). The Future of Software Development: AI-Driven Testing and Continuous Integration for Enhanced Reliability. International Journal of Trend in Scientific Research and Development, 2(4), 3082-3096. <https://www.ijtsrd.com/papers/ijtsrd14427.pdf>
- [14] Trudova, A., Doležel, M., & Buchalceková, A. (2020). Artificial intelligence in software test automation: A systematic literature review. In Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE) (pp. 181-192). <https://doi.org/10.5220/0009417801810192>