Human-Computer Interaction & Emerging Technologies, Vol. 195, 2025, 54–60
https://doi.org/10.54941/ahfe1006223

Accelerating Legacy Code Migration With Artificial Intelligence

Amir Schur, Max Graves, Stephanie Heckel, and David Vandine

Dark Wolf Solutions, LLC 13454 Sunrise Valley Dr, Suite 550, Herndon, VA 20171, United States

ABSTRACT

Organizations relying on critical systems with decades-old code face significant challenges, making modernization an operational imperative due to issues like operational stability, security, and a lack of updated features. This process of transforming legacy code is challenging, but is now being accelerated and augmented by Artificial Intelligence (AI) and large language models (LLMs). This research investigates the use of various LLMs for legacy code translation, aiming not for a perfect solution, but to significantly assist senior software developers by accelerating the development process, enabling rapid prototyping and initial implementation. This approach allows senior engineers to refine and productize the solutions, ensuring quality and alignment with system requirements. The initial testing strategy involved evaluating small subsets of legacy code within the Motif Framework, with the ultimate goal to demonstrate AI's role as an assistive tool for senior developers in accelerating code modernization efforts.

Keywords: Artificial intelligence, Large language models, Human systems integration, Legacy code migration

INTRODUCTION

Legacy software modernization has become increasingly common due to the critical role that long-standing systems play in organizations and the mounting challenges of maintaining them in their original form. Many of these systems support core business operations, such as finance, logistics, or healthcare, and cannot simply be replaced without substantial risk and cost. Over time, however, these systems become incompatible with modern technologies, making integration, security, and scalability difficult. As Assunção et al. (2024) note, modernization efforts are often driven by the need to reduce technical debt, enable digital transformation, and improve software maintainability—all while preserving business logic that has been refined over decades.

Furthermore, modernization is not solely a technical necessity but also a strategic one. Organizations face pressure to remain agile and competitive, which often requires re-platforming legacy applications, exposing their business logic via APIs, and often migrating them to the cloud. The increasing pace of change in software ecosystems—such as operating systems, programming languages, and compliance requirements—means that legacy

systems left untouched become liabilities. According to Bass (2022), the accumulation of "architectural and technical debt" in legacy codebases makes ongoing development costly and brittle, necessitating interventions that align with Lehman's Laws of Continuing Change and Increasing Complexity (Lehman, 1976).

While some of Lehman's Laws of Software Evolution have been debated and even challenged in the context of modern software development, a significant body of scientific literature continues to affirm their relevance. Lehman provided deep insights into how large, complex software systems change over time. These laws are particularly relevant to legacy system development, where software often persists for years or decades beyond its original design lifespan.

One of Lehman's key principles is the Law of Continuing Change, which asserts that a software system must be continually adapted, or it becomes less useful over time. In the context of legacy systems, this means that maintenance and upgrades are not optional but essential. A legacy system that remains untouched quickly drifts out of alignment with organizational needs, new hardware platforms, and integration requirements. This leads to mounting technical debt and operational inefficiencies.

Another relevant law is the Law of Increasing Complexity, which states that as a system evolves, its complexity tends to increase unless work is done to reduce it. Legacy systems often accumulate patches, undocumented fixes, and workarounds that complicate the codebase. Without systematic refactoring and architectural simplification, these systems become harder to understand, test, and modify. This complexity impedes innovation, discourages new developers, and increases the risk of failure during upgrades.

When an organization does not continuously adapt their system, even though they are functioning perfectly, there comes a time that their system becomes legacy. Then when a problem arises, suddenly there is an immediate need to upgrade the legacy system. This often becomes a significant challenge in software development projects. As there is a surge of artificial intelligence development, we want to explore how much of these capabilities can be utilized to assist a legacy modernization effort. This paper explores our internal research in exploring the use of AI in legacy code modernization.

ARTIFICIAL INTELLIGENCE FOR LEGACY CODE TRANSFORMATION

The emergence of transformer architectures has significantly advanced artificial intelligence capabilities in natural language processing, code comprehension, and code generation. Introduced by Vaswani et al. (2017), the transformer model revolutionized deep learning by enabling parallel processing and attention-based contextual understanding. This advancement allowed significantly faster processing times, which opened the door for various technological advancements. This is crucial for handling complex sequential data like human language and programming code. Various modern large language models (LLMs) built on transformer foundations now

demonstrate strong proficiency in code summarization, translation between programming languages, and even automated code refactoring.

In the context of legacy code modernization, transformers are increasingly being applied to analyze outdated systems, extract business logic, and assist in re-engineering efforts. These models can also provide semantic code search, anomaly detection, and automated documentation generation—tasks that traditionally required significant manual labor and institutional memory. This is particularly valuable in environments where original developers are no longer available, and documentation is sparse.

With the recent surge of LLM development, AI-powered code assistants are being integrated into integrated development environments to assist software developers in their daily programming tasks. A comprehensive analysis of the Visual Studio Code ecosystem, one of the most popular IDEs, by Liu et al. (2024) found 179 extensions explicitly categorized as AI coding assistants, defined by terms like "ai code," "gpt," or "chatbot" in their metadata. How much can a developer rely on AI assistance? This is what we want to start exploring and have some level of expectation on this capability boost.

Even with their impressive capabilities, LLMs are essentially sophisticated prediction machines. This inherent predictive nature means they can sometimes generate outputs that are factually incorrect or nonsensical, a phenomenon often referred to as 'hallucination' (Ji et al., 2023).

MOTIF LANGUAGE

We started exploring a simple code transformation using Motif as a legacy source language. In the late 1980s and early 1990s, Motif emerged as a dominant UNIX toolkit, winning out over alternatives like OPEN LOOK/Open Windows. Its alignment with IBM's Common User Access guidelines and visual similarity to Windows and OS/2 helped drive broad adoption users (Marshall, 1999).

Our strategy is to start with a simple Motif software program, then move onto a larger more complex application. We initially targeted a simple open source tic-tac-toe program from https://github.com/spartrekus/Motif-C-Examples.

TARGET PLATFORM

Since the target Motif programs were desktop applications, a modern technology stack comprising Electron.js and Typescript React.js was chosen as a target stack. This combination offers several advantages in replacing Motif for modern desktop applications.

Electron enables the creation of cross-platform desktop applications using standard web technologies (HTML, CSS, JavaScript), wrapped in a Chromium shell and Node.js runtime. This effectively eliminates the need for platform-specific GUI libraries like Motif or Qt, ensuring consistent behavior across different operating systems.

React.js, developed by Meta, offers a declarative, component-based model for building user interfaces. This makes it easier to manage

complex GUI states and user interactions compared to Motif's imperative and callback-heavy design. TypeScript adds static typing to JavaScript, enhancing reliability and maintainability for larger codebases. This is particularly valuable in modernizing legacy systems, as it reduces runtime errors and facilitates refactoring. Compared to C code typically used in Motif applications, TypeScript offers improved productivity, tooling, and integration with modern development environments like Visual Studio Code.

Each component of this new stack is actively maintained, widely used across the software industry, and largely known by the current development workforce. These factors make this modern stack an excellent choice for extending the life of a legacy Motif system.

TESTING STRATEGY

A testing strategy is critical in legacy code modernization. The intent of any code base or software product within an enterprise is to support valuable workstreams. Each legacy application is assumed to have some, often large, number of features which support these various workstreams. Whether one completely rewrites a software application from scratch based on SME/User input (ignoring the legacy code base) or ports the legacy code base directly, having a thorough understanding of these feature sets is of utmost importance to maintaining the legacy feature set. It is equally important to have a repeatable way to test these features. This can be facilitated through the establishment of robust test procedures, often including automated test suites to perform unit/integration testing as well as manual user acceptance testing (UAT) protocols or procedures. This knowledge of, and ability to test, the core functionality of the software helps establish a baseline. Once this baseline is established, it can be used to guide the development of new software, Transformation activities can start and then at the end another comparable testing can be performed to measure level of success.

AI TOOLS UTILIZATION

Our intent is to explore the efficacy of pair programming with various LLMs, putting the LLM in the role of a junior developer and having a human senior developer task the LLM and then perform code reviews for the code product that it produces. This more closely aligns with engineering work that is performed by most agile development teams, and is in contrast to a senior developer using an LLM to augment their own development flows directly.

Additionally, since most of the modern software engineering workforce is not readily familiar with legacy software languages or frameworks, our intent is to explore the effectiveness of LLMs in accelerating the understanding of the comprehensive feature set of legacy software.

We evaluated multiple large language model (LLM) based AI tools—including ChatGPT GPT-4o (OpenAI), Claude Sonnet 4 (Anthropic) and Gemini 2.5 Pro (Google, using a custom-built internal user interface). We also evaluate a beta project from Google that can be integrated with GitHub repositories: Jules (https://jules.google.com). Jules is Google's new

autonomous, asynchronous AI coding agent, powered by Gemini 2.5 Pro model. Similar prompts are given to each AI tool and then the results are run separately. It is noteworthy that other utilities exist that can be used similarly to how Jules is used - Claude Code Pro and Github Copilot being two examples. These have not yet been evaluated, but we plan to evaluate them in future iterations of this initiative.

To provide context for our evaluation of these AI tools, we created a dedicated NotebookLM (notebooklm.google.com) to document a concise history of each activity. As a note, Notebookllm has an option to create an audio overview (can be very useful for Podcasts).

PRELIMINARY RESULTS

Our initial experiments involved providing each LLM-based tool with similar prompts to translate small segments of Motif code into React/TypeScript equivalents. The simple case (Tic-Tac-Toe) consisted of a short series of prompts, which were kept nearly identical for each LLM in an attempt to avoid bias in the results.

First, the LLMs were prompted to read the legacy Tic Tac Toe code base from Github, summarize its core feature set, then provide us with instructions for compilation. Next, the LLMs were prompted to suggest suitable modern technology stacks with which to replace the legacy project. Finally, the LLMs were prompted to port the legacy Motif code base to our preferred technology stack. Our preliminary observations indicate varying degrees of success and distinct strengths and weaknesses across the tools.

All LLMs were able to accurately describe the core functionality of the legacy code, as well as instruct us how to compile and run the project. When prompted, they were also able to fix a minor rendering issue in the original project's UI, including an explanation of why their suggested fix would address the issue. They were also able to suggest a list of suitable modern replacement technology stacks. On the other hand, all LLMs struggled to suggest a properly structured start to the new project. Each LLM presented solutions which were either outdated or not fully functional without telling them exact instructions, and had to be provided exact instructions to start the new project.

ChatGPT: Demonstrated a strong ability to understand the overall structure of the Motif code and generate React components. However, it sometimes struggled with accurately reproducing the specific UI behavior and event handling logic of the original Motif application. We noted several instances where ChatGPT invented components or properties that didn't exist in the target React/TypeScript framework, resulting in code that would not compile or function correctly without significant manual intervention.

Claude: Exhibited a more conservative approach to code generation, often providing more verbose but generally more accurate translations. While Claude produced less novel or creative solutions compared to ChatGPT, it tended to avoid "hallucinations" and adhere more closely to established React/TypeScript patterns. However, it sometimes lacked the ability to

optimize the translated code for performance or readability, resulting in code that was functional but less maintainable.

Gemini (Custom UI): Showed promise in understanding the intent of the translation task and generating code that aligned with modern React best practices. The custom UI allowed for iterative refinement of the prompts and easier debugging of the generated code. However, the model sometimes struggled with understanding the specific constraints and limitations of the target environment, leading to code that was not fully compatible with our chosen tooling and libraries.

Jules (Google): As an asynchronous AI coding agent, Jules could propose changes independently. Initial findings indicated that it was able to translate sections of code correctly, but would struggle and get stuck in endless loops during testing. Further analysis is needed to fully validate results. It is noteworthy that Jules is in Beta. It is also noteworthy that the use of Jules presents a different development paradigm than how we were using the other LLMs, since working with it more closely aligns with the asynchronous nature of agile development versus going prompt-by-prompt with the others.

Overall, our preliminary findings suggest that while LLM-based tools can significantly accelerate the initial stages of legacy code translation, they are not yet capable of fully automating the process. This is true even for the case of the very trivial Tic-Tac-Toe program. Though, for this simple case, it is noteworthy that all results were obtained without the developer needing to read or comprehend the legacy language.

Our results point toward manual review, testing, and refinement of the generated code being essential to ensure accuracy, reliability, and maintainability. The tools appear to be most effective when used in a collaborative workflow, where experienced developers can leverage their expertise to guide the LLM and correct its mistakes, or as an augmentation of their own skillset to expedite reading documentation while writing code themselves.

Further research and experimentation are needed to optimize the use of these tools and unlock their full potential for legacy code modernization. A more rigorous testing and evaluation matrix will be incorporated as the next step, using a much more complex Motif code base.

FUTURE WORK

This initial investigation into the use of AI for legacy code modernization has highlighted both the promise and the limitations of current LLM-based tools. While the ability to rapidly generate code snippets and suggest architectural transformations is compelling, challenges remain in ensuring the accuracy, reliability, and maintainability of the resulting code.

The challenge of legacy code transformation is not going away. In fact, as mission-critical systems age and the pool of developers familiar with legacy languages continues to shrink, the risk and cost of inaction only grow. Recognizing that LLMs are unlikely to fully automate legacy code modernization in the near future, we will focus on developing tools and workflows that facilitate effective human-AI collaboration. This includes

designing user interfaces that allow developers to easily review, modify, and validate AI-generated code, as well as developing methods for capturing and incorporating developer feedback into the LLM's training process.

REFERENCES

Assunção, W. K. G., Marchezan, L., Egyed, A., & Ramler, R. (2024). Contemporary software modernization: Perspectives and challenges to deal with legacy systems. arXiv. https://arxiv.org/abs/2407.04017

Bass, J. M. (2022). Technical debt, software evolution and legacy. In Agile software engineering skills (pp. 291–296). Springer, Cham. https://doi.org/10.1007/978-3-031-05469-3_20

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y.,... & Bang, Y. (2023). Survey of hallucination in natural language generation. arXiv. https://arxiv.org/abs/2202.03629

Lehman, M. M., & Belady, L. A. (1976). A model of large program development. IBM Systems Journal, 15(3), 225–252.

Liu, Y., Tantithamthavorn, C., & Li, L. (2024). Protect Your Secrets: Understanding and Measuring Data Exposure in VSCode Extensions. https://arxiv.org/abs/2412.00707

Marshall, D. (1999). The Road to X/Motif [Lecture notes]. Cardiff University. Retrieved June 16, 2025, from https://users.cs.cf.ac.uk/Dave.~Marshall/X_lecture/X_book_caller/node1.html.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N.,... & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30. https://doi.org/10.48550/arXiv.1706.03762

Zhou, Y., Guo, D., Liu, S., Ou, Y., Zhang, M., & Feng, X. (2022). CodeT5+: Open code large language models for code understanding and generation. arXiv. https://arxiv.org/abs/2305.07922