SvedbergOpen
DISSEMINATION OF KNOWLEDGE
International Journal of Artificial Intelligence and Machine Learning
Publisher's Home Page: https://www.svedbergopen.com/
Research Paper Open Access

Ai-Assisted Legacy Front-End Modernization: Intelligent Jquery-To-React Transformation Using Automated Component Extraction And Dependency Analysis

Mohammed Sayerwala

Mphasis Corporation, USA. ORCID ID: 0009-0002-8153-6925

Abstract
Legacy enterprise web applications built with jQuery-based application architecture form a large proportion of the most ubiquitous live production systems in the financial services, health care and regulated enterprise markets. Due to the intrinsic imperative DOM manipulation, state management and event handling conventions and tight coupling of these applications, conventional enterprise-scale manual refactoring is not an option for these systems. We present an AI-assisted transformation pipeline for legacy front-end modernization, which conducts automated component extraction and dependency analysis to transform legacy jQuery web applications into React component systems. We, in particular, use static analysis and code pattern classification through machine learning to predict component boundaries in React code, create dependency graphs that illustrate inter-module dependencies, and synthesize candidate React components for inspection and modification by developers. The hybrid-coexistence layer supports incremental migration. The experiment shows that prioritizing extraction candidates based on the workflow and applying AI-assisted regression test suite generation can reduce the migration time and defects. The article concludes with an overview of the organization and governance of large-scale front-end modernization efforts.

Keywords: AI-assisted migration, jQuery to React, component extraction, dependency analysis, legacy modernization, front-end library/framework engineering, enterprise architecture.

1. Introduction

Enterprise web applications, especially those written in the early- to mid-2000s, made wide-ranging use of jQuery for their user interface. jQuery was one of the first browser bug abstraction libraries and provided a simple API to navigate the DOM tree. The jQuery interface had made it easier to write interactive content without having to also treat it as a component, but over the next 20 years, increasingly complicated features had been bolted onto the application in an architecture that was never intended to handle that level of complexity. jQuery technical debt on the front end is different from other types of stack technical debt because, rather than being inefficient code, jQuery debt is structural debt where the business logic, the user interface state and the user interface rendering of the application are so intermingled that changing any one of them will break the entire system. [1][2]

The big-bang rewrite and manually extracting components are by far the most common but have well-known limitations: the big-bang rewrite assumes the new version of the application needs to be feature-parity to the old one before switching over, resulting in a very long transition period for the organization in which both systems are maintained, leading to unwanted regressions. In particular, manually extracting every jQuery interaction would require an engineer to fully understand the implications of every one before wrapping it in a React component, which is not feasible for codebases with several hundred thousand lines of imperative JavaScript code. [3][4]

We describe an architecture and implementation of a semi-automated legacy front-end modernization pipeline that performs component boundary identification and generation, dependency graph generation, and candidate component generation via static analysis and machine-learning-based pattern detection as a human-

in-the-loop process. We discuss technical and organizational factors relevant to the construction of such an enterprise pipeline. [5]

2. Structural Limitations of jQuery-based Enterprise Frontends

2.1 Imperative DOM Coupling and State Management Fragmentation

The basic structural problem with jQuery-based enterprise frontends is that what the UI looks like and how to build it are conceptually separated in most component-based frameworks. For jQuery the natural programming model is to imperatively manipulate the DOM in response to events. In practice, the current state of a jQuery-based frontend generally matches the structure of the UI encoded in the DOM. If an user types data in a single input element of any form, the jQuery-defined event handlers on that element could make direct changes to other elements, hide sections of the form, create and populate a drop down menu, etc. This is all represented as dozens of DOM nodes in the inspector, not a single inspectable object. [6]

This fragmentation makes the behavior of complex jQuery UIs extremely difficult to reason about. When attempting to reason about what an user interface will do given specific events, an engineer must reason about every event handler that could have applied to the DOM nodes in question, the specific order in which they were fired, and which ones were dynamically bound or unbound. However, jQuery does not have an equivalent concept to React's single monolithic state object that can be inspected, serialized, and reset to a known state, nor a common way to understand the behavioral envelope of a jQuery component. To pull out a component and replace it with something else, the internals of how it works must be understood and mapped out, which is costly and laborious. [7][8]

2.2 Conventional migration methodologies are inadequate

Full rewrites are also not considered a good approach to modernizing jQuery, as they assume the development organization responsible for the system has the resources to run two versions of the system in parallel for the time the rewrite is happening. In practice, this assumption never really holds, since business needs will always force important changes to the legacy system and the new system will always be falling behind the latest feature set of the legacy system, resulting in an infinitely-extended rewrite process or a rewrite that ships incomplete. [9][10]

Manual incremental extraction avoids the parallel maintenance problem. However, it forces engineers to do the full depth, breadth, and cost of the dependency analysis of the previous section for each extract manually, and without tooling support. If the application has many thousands of jQuery interactions, this is the major cost of migration, leaving little time for development, and potentially making it too slow to keep up with changes to the legacy application. This cognitive overhead and the low execution rate are one of the main reasons why manual solutions to the problem are not suitable for enterprises [11].

3. AI-Assisted Component Extraction Framework

3.1 Pattern Recognition in the DOM and Component Boundaries

First, we identify component candidates in the legacy codebase. A component candidate is an interface with a particular structural form that we expect a React component to take: it is a bounded region of the DOM that is rendered for a well-defined set of inputs, properly descoped to a small set of user interactions, and with a small set of conceptual interfaces to the rest of the application. It is then possible to identify these code regions in a jQuery codebase by analyzing its HTML structure and event handler graph. [12]

During static analysis, jQuery components in the codebase are parsed and the event handler graph is created. The event handler graph allows the tool to map each selector to the type of event handlers that listen to it, where modifications to the DOM occur, and what modifications in the DOM then trigger which handlers. Pattern-identification with machine learning is used to identify tightly-coupled selectors and handlers that are functional units. These clusters are thus considered candidate components, and the boundary detection heuristic assigns each a confidence score that accounts for the ratio between cohesion and dependencies. Candidates with high scores go directly to generation, while those with lower scores are listed for human inspection. [13]

Table 1: Manual vs. AI-Assisted Component Extraction

Dimension	Manual Extraction	AI-Assisted Extraction
Component discovery	Engineer reads code; identifies boundaries manually	Static analysis + ML pattern recognition; auto-suggests candidates
Dependency mapping	Manual tracing across event handler graph	Automated dependency graph construction; categories classified
Coverage	Limited by engineer bandwidth; complex components deferred	Full codebase analysed in single pass; all candidates ranked
Consistency	Varies by engineer experience	Deterministic; same codebase produces same candidate list
Time to first component	Days to weeks per complex component	Hours; human review focuses on AI output rather than discovery

Table 1. Comparison of manual and AI-assisted approaches to jQuery component extraction. (Author's own analysis)

3.2 Dependency Graph Construction and Impact Analysis

Once the candidate components have been identified, a dependency graph is constructed for the set of components comprising the legacy codebase. The dependency graph describes the data, event, and shared state dependencies between these components in the legacy codebase. Thus, the relative frequency and direction of these dependency types determine the extraction order. [14]

Because of the dependency graph, we can do a pre-extraction impact assessment: when we extract a candidate component, we can inspect its dependents and the dependencies that we will need to adapt or bridge over when moving that component from the jQuery DOM into the React component tree. This results in fewer post-extraction defects because we surface special integration requirements before they result in production failures. A further hidden complexity will be revealed by the dependency graph, where components that are simple separately may be nested, giving a high cost of extraction [15].

Table 2: Dependency Graph Metrics — Categories of Inter-Component Dependencies

Dependency Category	Description	Migration Implication
Data dependency	Component A reads DOM state written by Component B	Extract B before A; or bridge via shared state object
Event dependency	Component A subscribes to events triggered by Component B	Requires event bridge during coexistence phase
Shared state dependency	A and B both read and write the same DOM nodes	High-risk; extract together or implement shared context first
Indirect dependency	A depends on C which depends on B (transitive)	Impact score increases; extraction sequence must respect chain

Table 2. Categories of inter-component dependency in jQuery codebases and their migration implications. (Author's own analysis)

4. Transformation Pipeline Architecture

4.1 Hybrid coexistence layer and incremental replacement protocol

The most important architectural design element of an incremental jQuery to React migration is building a coexistence layer for React and jQuery, allowing them to run on the same page at the same time without

interfering with one another. This coexistence layer must do the following things at the same time: render React components inside the HTML managed by jQuery, allow React components to communicate with jQuery components which have not been migrated yet, and prevent the two rendering systems from fighting with one another. [16]

The coexistence layer consists of a set of DOM nodes in the legacy HTML markup structure, anchored at stable locations where React's reconciler can treat them as its root and jQuery can manage the rest of the DOM. An event bridge, built on a small publish-subscribe algorithm with a defined message schema, connects jQuery's event handler graph to React's component tree. When the migrated component needs to respond to a jQuery-initiated event, the jQuery handler triggers an event on the event bridge, and the migrated component receives a message of the same type from the bridge, updating its state. This decouples the migrated components from jQuery's DOM state, making it possible to test them, and enabling the migration of their jQuery-dependent components. [17]

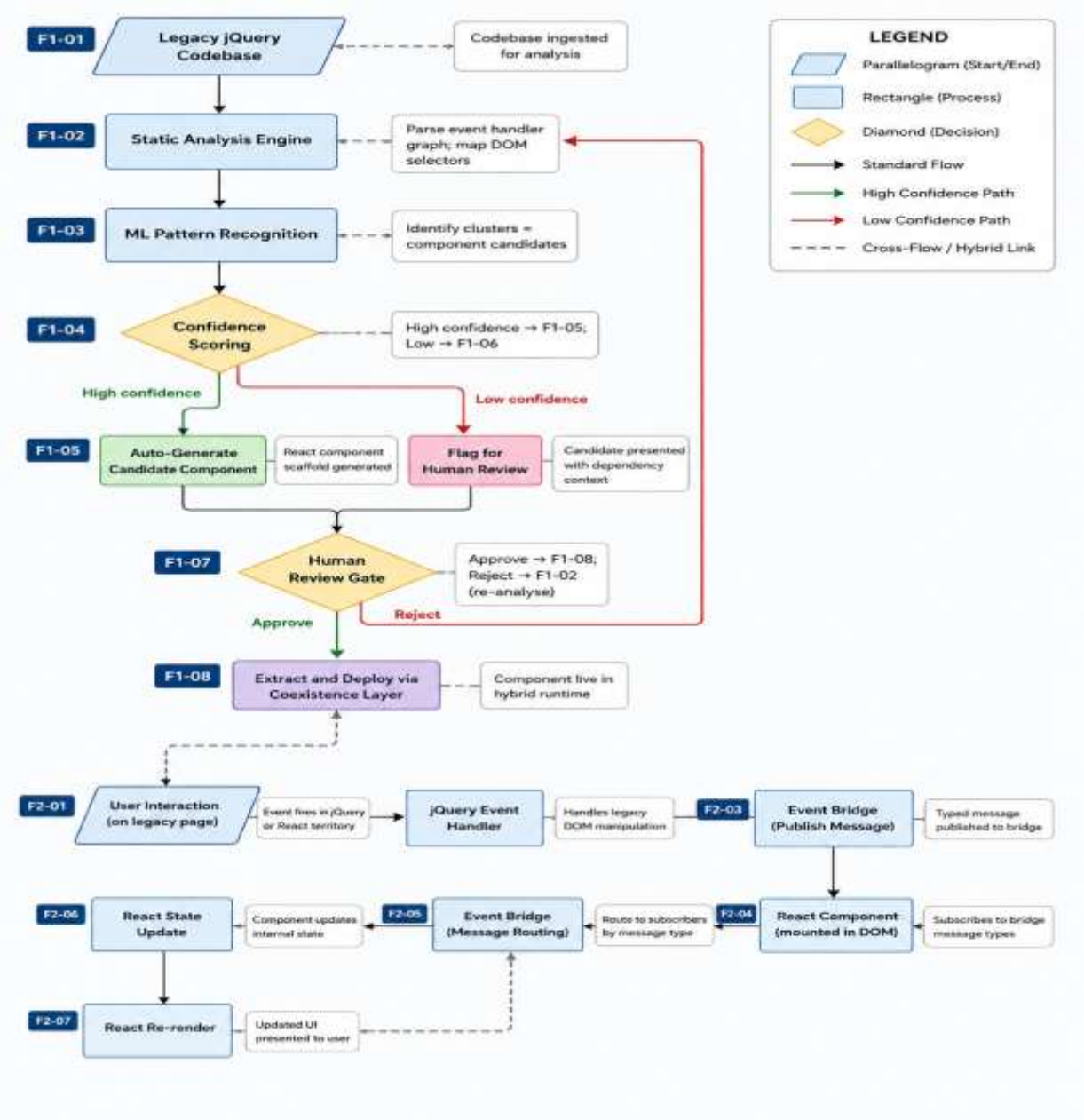

Figure 1. AI-assisted jQuery-to-React transformation workflow

[Note: AI-assisted jQuery-to-React transformation workflow illustrating automated component extraction, confidence-based review routing, hybrid coexistence deployment, and event-bridge communication between legacy jQuery handlers and React components.]

Table 3: Hybrid Coexistence Layer — Event Bridge Message Types

Message Type	Publisher	Subscriber	Purpose
FIELD_UPDATED	jQuery handler	React component	Notify React of legacy field value change
STEP_COMPLETED	React component	jQuery orchestrator	Signal step done; unlock next legacy section
VALIDATION_RESULT	Java API (via Rails)	React component	Deliver field-level validation feedback
SESSION_STATE	Rails controller	Both layers	Synchronise workflow position across technologies
HANDOFF_INITIATED	Branch system	React layer	Trigger session resumption from persisted state

Table 3. Event bridge message types enabling communication between jQuery and React layers during coexistence. (Author's own analysis)

4.2 Automatic test generation and regression testing

The biggest risk of incremental migration is behavioral regression, where the migrated React code does not behave exactly as desired or expected compared to the original interaction in jQuery. To reduce these issues, tests that capture legacy behavior well enough to distinguish correct and incorrect React code can be written. Instead, AI-assisted test generation involves instrumenting the legacy jQuery codebase so that traces (i.e. sequences of events, DOM states and server requests) can be recorded when the system is in use in production. [1][2]

This behavior is then analyzed for reusable patterns and expressed as test specifications that describe the expected input-output behavior of a particular candidate component. Once the React component is generated and deployed, the test suite validates it against the expected behavior specification based on the legacy behavior. There is no need for engineers to explicitly specify the expected behavior of each interaction. This observation has the most benefit for high-dependency and complex components that show up later in the migration, where behavioral divergence that would not otherwise be detected is likely to intrude on end-users' experiences. [3] [4]

5. Considerations for Enterprise Scale

5.1 Prioritize based on workflow for positive business impact

The dependency graph, and the confidence scores computed during the analysis phase, provide a technical priority for the extraction candidates. The migration phase also incorporates business priorities. For example, a simple component that is rarely used by clients may be deemed much less important to migrate than a complex component traversed by all clients during the new account opening process. In this way, workflow-driven prioritization ensures that migrations are performed in the most helpful order for business-critical user journeys as defined by product and operations stakeholders. [5][6]

The prioritization results in an eventual roadmap divided into phases, based on tradeoffs between the technical difficulty, possible ordering dependencies and business value of different pieces of work, with the highest value and least dependent first. These quick wins allow us to show the feasibility of the migration path and React components for other teams to build on. Medium-complexity components that are both critical to user journeys follow in subsequent phases, respecting the dependency ordering of the graph analysis. [7][8]

Table 4: Migration Phase Timeline and Risk Classification

Migration Phase	Components Targeted	Risk Level	Business Impact
Phase 1 — Quick wins	High-impact, zero external dependencies	Low	Immediate UX improvement on highest-traffic journeys
Phase 2 — Core journeys	Critical user journey components; moderate dependencies	Medium	Completes primary customer-facing workflows in React
Phase 3 — Complex integrations	Multi-dependency components; shared state patterns	High	Addresses remaining legacy interactions; decommission eligible
Phase 4 — Decommission	Legacy fallback paths; routing layer removal	Low	Full system on React; legacy system retired

Table 4. Migration phase structure balancing technical risk and business impact. (Author's own analysis)

5.2 Governance, ownership, and organizational change management

Such a large migration is not only a technical effort, but an organizational one. The jQuery code base, which is tightly coupled, presents a model of near-complete centralized ownership of code. This is in contrast to the decentralized model of feature team code ownership that is applied to the React component library. Finally, component ownership must be tracked and assigned, a shared component library with versioning and contribution guidelines must be built, and teams that own a component must have the autonomy to modify and extend that component independently of a migration team. [9][10]

One of the most important aspects of organizational change management. Engineers who have spent years working in a legacy jQuery codebase may be very reluctant to the component model, not because of any obstinacy, but simply because they don't understand the value of it or the limitations. Areas where successful teams have invested are structured training, pairing with migration experts, and executive sponsorship. [11][12]

Conclusion

AI-assisted legacy front-end modernization is an improvement on two customary approaches: big-bang rewrite (which is common in jQuery to React modernization) and unassisted manual extraction. AI-assisted modernization removes the labor overhead of unassisted approaches by automatically performing the work of dependency analysis and component boundary detection. This moves engineering work from investigation work to validation work: AI-assisted approaches put the engineering effort into reviewing candidates for component boundaries, rather than discovering them. The hybrid coexistence layer allows this work to be performed incrementally and without downtime, while the AI-generated regression test suite provides a behavioral safety net that a manual migration would lack.

Open research questions include: How well do ML-based boundary detection methods work on irregular legacy codebases? How can generated test suites cover unique program interaction paths? How do we scale the event bridge design pattern when applications generate massive amounts of events? Organizational topics (team handoffs, new governance structures, cultural change) were also important in practice but studied little in the literature of large-scale migration. Nonetheless, the AI-assisted migration framework proposed here can be of great value to enterprises with frontends, developed in the jQuery era, yet limited in terms of competitive product evolution.

References

[1] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "How do software development teams manage technical debt? An empirical study," J. Syst. Softw., vol. 120, pp. 107-122, 2016. DOI: 10.1016/j.jss.2016.05.018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412121630053X

[2] T. Besker, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," J. Syst. Softw., vol. 135, pp. 1-16, 2018. DOI: 10.1016/j.jss.2017.09.025. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0164121217302121
[3] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? Manage it? Ignore it? Software practitioners and technical debt," in Proc. 10th Joint Meeting ESEC/FSE, 2015, pp. 50-60. DOI: 10.1145/2786805.2786848. [Online]. Available: https://dl.acm.org/doi/10.1145/2786805.2786848
[4] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: An investigation of how developers spend their time," in Proc. 23rd IEEE Int. Conf. Program Comprehension (ICPC), 2015, pp. 25-35. [Online]. Available: https://robertominelli.com/assets/downloads/publications/Mine2015b.pdf
[5] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production and test code," in Proc. 1st Int. Conf. Software Testing, Verification and Validation (ICST), 2008, pp. 220-229. DOI: 10.1109/ICST.2008.47.
[6] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," IEEE Trans. Softw. Eng., vol. 32, no. 11, pp. 896-909, 2006. DOI: 10.1109/TSE.2006.112. [Online]. Available: https://ieeexplore.ieee.org/document/4015512
[7] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," IEEE Trans. Softw. Eng., vol. 41, no. 5, pp. 462-489, 2015. [Online]. Available: https://fpalomba.github.io/pdf/Journals/J1.pdf
[8] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in Proc. 35th Int. Conf. Software Engineering (ICSE), 2013, pp. 712-721. DOI: 10.1109/ICSE.2013.6606617. [Online]. Available: https://dl.acm.org/doi/10.5555/2486788.2486882
[9] R. Koschke, "Survey of research on software clones," in Dagstuhl Seminar Proc. 06301: Duplication, Redundancy, and Similarity in Software, 2007, pp. 1-24. [Online]. Available: https://drops.dagstuhl.de/storage/16dagstuhl-seminar-proceedings/dsp-vol06301/DagSemProc.06301.13/DagSemProc.06301.13.pdf
[10] M. Weiser, "Program slicing," IEEE Trans. Softw. Eng., vol. SE-10, no. 4, pp. 352-357, 1984. DOI: 10.1109/TSE.1984.5010248. [Online]. Available: https://dl.acm.org/doi/10.1109/TSE.1984.5010248
[11] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study," in Proc. 8th Int. Conf. CLOSER, 2018, pp. 221-232. [Online]. Available: https://www.scitepress.org/Papers/2018/67983/
[12] D. Lo, N. Nagappan, and T. Zimmermann, "How practitioners perceive the relevance of software engineering research," in Proc. 10th Joint Meeting ESEC/FSE, 2015, pp. 415-425. DOI: 10.1145/2786805.2786809. [Online]. Available: https://thomas-zimmermann.com/publications/files/lo-esecfse-2015.pdf
[13] R. Verdecchia, I. Malavolta, and P. Lago, "Guidelines for adopting frontend architectures and patterns in microservices-based systems," in Proc. European Conf. Software Architecture (ECSA), 2020. [Online]. Available: https://www.researchgate.net/publication/318872312_Guidelines_for_adopting_frontend_architectures_and_patterns_in_microservices-based_systems
[14] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in Proc. 38th Int. Conf. Software Engineering (ICSE), 2016, pp. 96-107. DOI: 10.1145/2884781.2884783. [Online]. Available: https://dl.acm.org/doi/10.1145/2884781.2884783
[15] V. Lenarduzzi, N. Saarimaki, and D. Taibi, "The technical debt dataset," in Proc. 15th Int. Conf. Predictive Models and Data Analytics in Software Engineering (PROMISE), 2019, pp. 2-11. DOI: 10.1145/3345629.3345630. [Online]. Available: https://dl.acm.org/doi/10.1145/3345629.3345630
[16] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Microservices in industry: Insights into technologies, characteristics, and software quality," in Proc. IEEE ICSA-C, 2019, pp. 187-195. DOI: 10.1109/ICSA-C.2019.00041. [Online]. Available: https://ieeexplore.ieee.org/document/8712375
[17] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," IEEE Software, vol. 35, no. 3, pp. 24-35, 2018.