Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Modernizing a Legacy Banking Platform: How a Strangler Fig Approach Enabled Zero-Downtime Migration from jQuery Monoliths to a Scalable React Component Architecture

Mohammed Sayerwala
Mphasis Corporation, USA

ARTICLE INFO

ABSTRACT

Introduction: Legacy jQuery-based monolithic banking platforms are a major challenge for digital transformation at financial services companies because of tightly coupled front-end architectures, scalability issues, limited mobile performance, the need to fix technical debt, and a general loss of performance. These challenges make it difficult to deliver modern banking capabilities like real-time identity verification, digital funding services, omnichannel customer onboarding, and zero-downtime modernization in a regulated environment.

Objectives: Following the Strangler Fig pattern, this paper describes the step-wise transition of legacy banking account origination platforms into a user-friendly and scalable React component architecture. It also highlights the importance of component-driven design, distributed onboarding workflows and session recovery in providing constantly available service with improved customer experience and operational resilience.

Methods: We propose an architecture developed on the principles of the enterprise software modernization and the Strangler Fig migration pattern. The architecture uses the principles of incremental traffic routing, React component libraries, asynchronous REST-based service integration, client-side validation in real-time, distributed multi-applicant workflow orchestration, state synchronization, and persistent session recovery. By analyzing migration phases, comparing models, examining workflow states, and considering operational recovery scenarios, it is determined that zero-downtime transformation is possible within regulated banking contexts.

Results: The evolution approach allows for incremental replacement of legacy modules in production services, as well as incremental rollback of the migration at any time. The React-based component architecture provides for more modular, reusable, responsive, and interoperable APIs than customary jQuery-based implementations have been able to provide. Distributed workflow orchestration allows multi-applicant account origination in parallel, while persistent session management allows workflows to continue during device loss and via branch assistance, reducing operational risk and customer onboarding effort while gradual legacy system migration becomes possible.

Conclusions: In this paper, combined the Strangler Fig migration pattern with modern React component architecture to create a zero-downtime migration approach to modernizing legacy banking platforms. This approach employs incremental migration, omnichannel component design, workflow synchronization of components, and session resilience, that enables scalable, streamlined, digital transformation while achieving business continuity, regulatory compliance, and an improved end-user experience. The framework is a reusable architectural blueprint for enterprise modernization efforts in regulated financial systems.

Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Keywords: Strangler Fig Pattern, React, Jquery, Legacy Modernization, Banking, Account Origination Systems, Zero Downtime, Component Architecture.

INTRODUCTION

Financial firms running legacy web applications built with jQuery and other JavaScript frameworks that peaked in the jQuery era face a double challenge of modernizing their architecture by addressing the structural debt such monolithic front-ends accrue. They become increasingly difficult to maintain and adapt to the services expected today by customers, such as real time identity verification, instant funding, or mobile first account opening. The operational dimension is sometimes less appreciated but no less important: the systems that need modernization are the same systems handling customer's transactions every day, and any approach to modernization that entails service interruption incurs costs in the form of customer experience degradation, regulatory scrutiny and operational risk that the financial institution cannot tolerate. [1][2]

The Strangler Fig pattern is a reaction to this limitation. Instead of replacing the legacy system in one cutover, Strangler Fig pattern pushes the new system to surround the existing one, by routing some particular types of requests to the new system and others to the existing implementation. Thus, over time, the new system assumes an increasing percentage of the application's total workload while the old system assumes a progressively smaller percentage of the workload, until the old system is eventually retired with no downtime to the customer. [3][4]

This article discusses how the Strangler Fig pattern was used to migrate from a jQuery-based banking platform into a component-based architecture, what technical migration patterns were used to modernize the platform, and what operational migration patterns were used to support and maintain the business process workflow and regulatory compliance during the migration. [5]

THE STRANGLER FIG PATTERN IN ENTERPRISE MODERNIZATION

2.1 Incremental Replacement Strategy And Traffic Routing

Strangler Fig begins with a routing layer that sits in front of the client and legacy application. The routing layer inspects each request and makes a routing decision. It sends requests to any features that have already been moved to the new React application, and to the legacy application for any features that are still there. This is achieved by the use of the routing layer, which allows new and old systems to run in production simultaneously, serving requests to a common client base, without the client being aware of which system is serving a request. [6][7]

Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

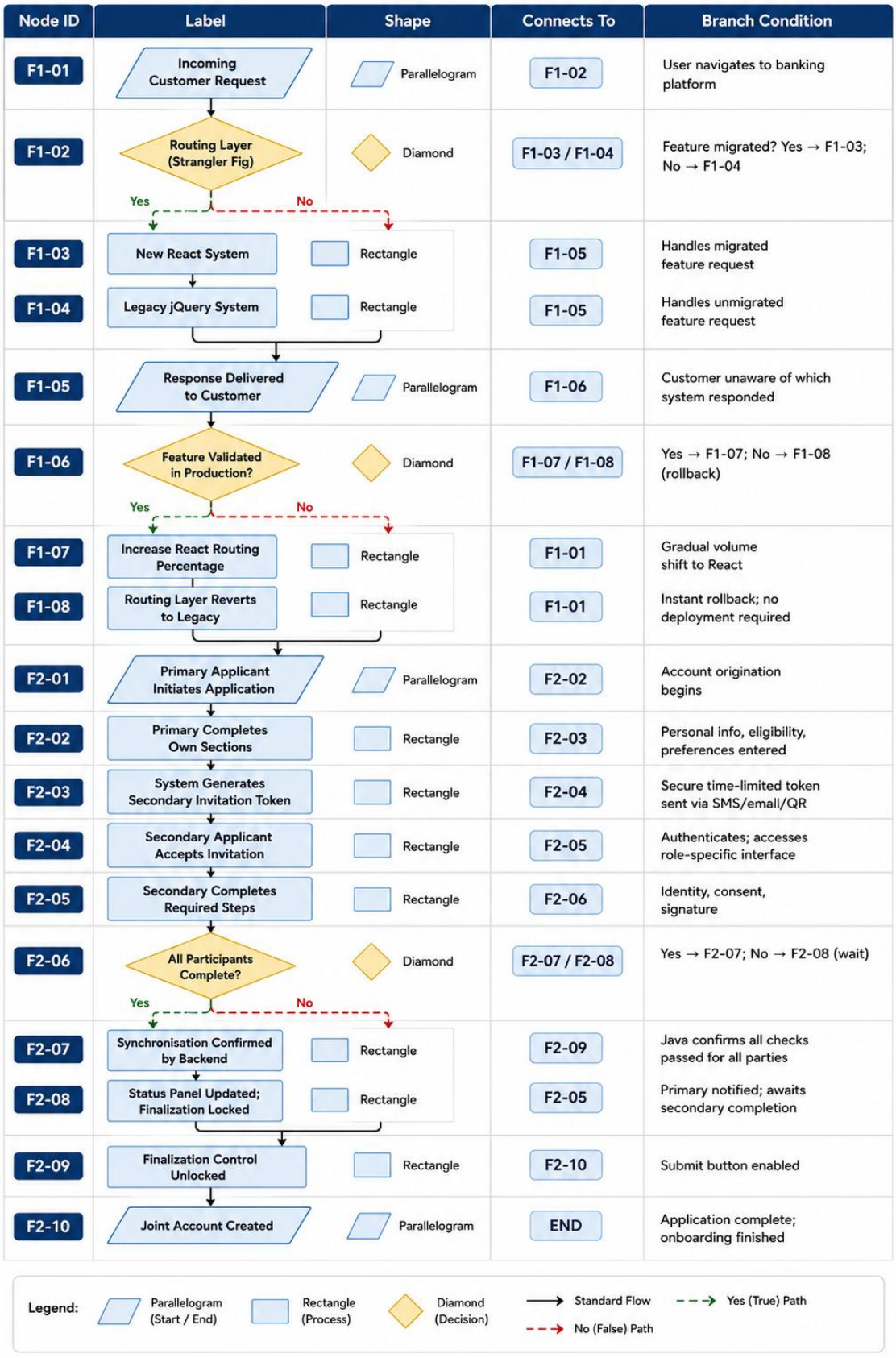

Figure 1. Feature Migration And Onboarding Process Flow

Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

[Note: **Figure 1.** Strangler Fig migration workflow illustrating routing-based incremental replacement, rollback-enabled traffic shifting, and distributed multi-applicant onboarding synchronization in a React-modernized banking platform.]

The migration phase plan attempts to balance technology dependencies with business value. The best candidates for the early migration phases are features that are well separated in the legacy system, valuable to customers, and do not have direct dependences on legacy system components or services that are not changing. For example, in a banking account origination context, this is often the first several screens of a new account application - the most high-traffic, isolated, and high-impact portions of the legacy system. [8]

Table 1: Strangler Fig Phase Plan

Migration Phase	Legacy Module	React Replacement	Routing Condition	Validation Criteria
Phase 1	Account application entry (steps 1-3)	React account initiation component	New session requests only	Completion rate parity with legacy; zero data loss events
Phase 2	Identity verification flow	React identity component + third-party API	All individual application traffic	Verification pass rate; API error handling confirmed
Phase 3	Funding selection and bank link	React funding component	All funded application traffic	Funding success rate; account activation integrity
Phase 4	Joint/multi-applicant flows	React distributed workflow	All multi-applicant traffic	State synchronisation confirmed; all participants completing
Phase 5	Branch representative workflows	React session recovery + handoff	All branch-assisted sessions	Handoff success rate; session state integrity validated

Table 1. Migration phase structure showing legacy modules, React replacements, routing conditions, and validation criteria. (Author's own analysis)

2.2 Risk Mitigation and Parallel Validation in Financial Environments

One direct benefit of the Strangler Fig pattern is risk reduction. In regulated financial environments, rather than having a big-bang cutover (and risk failing the entire business due to exposing a newly deployed system to production load with no tested rollback path), the Strangler Fig pattern allows a migrated component to prove its production-readiness before any meaningful volume is routed to it. The rewritten React component for the account application entry flow can be tested with a small fraction of the real customer traffic (typically, using a feature flag for gradual rollout), while the legacy system handles the bulk of the traffic. [9]

Furthermore, rollback is a continuous property of the Strangler Fig migration. At any point in the migration, the routing layer can switch any component back to the legacy system without a need to push new code, or to restart the service. This rollback capability remains available during the migration until the engineering team can revert the production behavior to the legacy behavior in case an incident occurs. In regulated industries, where the time allowed to respond to an incident is a contractual or regulatory requirement, this may be a compliance requirement. [10]

Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

3.1 Using Jquery Scripts To Create Reusable UI Component Libraries

The most important architecture change supporting this modernization is replacing the imperative jQuery DOM manipulation code with declarative React components, organized into a shared component library. This changes the unit of work in the front-end from an imperative approach of writing scripts that manipulate specific DOM nodes in response to specific events to a declarative approach where components produce a defined interface in response to a defined state, and communicate with the rest of the application through a well-defined interface. [11]

If a component library is responding to multiple viewport sizes and there are definitions for how it changes for a desktop view versus a mobile view in both layout and type or interaction, then the component library can be built once to support the web and the mobile web. The component library uses a design system to give visual and interaction consistency across the entire account origination experience and avoid inconsistencies that can often build up in jQuery codebases as multiple software engineers create similar interactions. [12]

Table 2: jQuery Monolith vs. React Component Architecture

Dimension	jQuery Monolith	React Component Architecture
State management	Implicit; distributed across DOM nodes	Explicit; single state object per component
Rendering model	Imperative DOM manipulation	Declarative; UI is function of state
Mobile responsiveness	Often absent; fixed-width layouts	Built-in; responsive via CSS flexbox/grid
Reusability	Low; tightly coupled to page context	High; components deploy across products unchanged
Testability	Requires browser; brittle DOM assertions	Unit-testable in isolation; no browser dependency
Team scalability	Centralized; shared code creates merge conflicts	Distributed; teams own bounded component domains
Integration with modern APIs	Synchronous page reloads; limited async	Async-first; real-time validation via REST APIs

Table 2. Architectural comparison of jQuery monolith and React component architecture across key engineering dimensions. (Author's own analysis)

3.2 Asynchronous Service Integration and Real-time Data Validation

The most fundamental change to improve the customer experience was removing full page reloads as the primary mechanism for posting data to the server and displaying validation errors. In the legacy jQuery based system it was necessary to make a full round trip to the server to post a part of a form and show server-side validation errors, which would cause the customer to lose sight of their current state for a few seconds and potentially lose data in case of a network failure. The replacement does this in React fashion, asynchronously posting the form data as a REST API call, returning structured validation information, and refreshing the page to update only the affected parts of the DOM. This achieves the goal of not losing current state of the application while being able to provide validation feedback within the API call latency. [13]

Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

The React component architecture allows for a feature called real-time validation, where the application can validate customer input on the client side at a field level and not wait until the entire form is submitted. For example, asynchronous calls to the server can be debounced when the user updates a field, allowing for validation of format restrictions, checking for duplicate information in a backend database, or exposing eligibility requirements applicable to certain fields. This prevents multiple submission-time validation failures, so that the customer does not have to fill out an entire multi-field form and be returned with error messages. [14]

MULTI-USER WORKFLOWS AND STATE SYNCHRONIZATION

4.1 Distributed Onboarding For Joint Financial Accounts

A process for opening a joint bank account must accommodate two or more applicants completing a single application process. Legacy processes, either enforcing multiple applicants' physical presence or requiring multiple applications to be completed by each applicant in turn, tend to have higher abandonment rates than a single application. This, in turn, tends to reduce customer adoption of joint bank accounts. [15]

This React-enabled modernization addresses this (and other) issues by providing a distributed workflow so that multiple applicants are able to complete the application on their chosen device, at their chosen pace, and with the minimum involvement of other applicants. It is initiated when the principal applicant completes the personal details section. If any action is required from the secondary applicant, a time-limited invite token is generated and sent to them via email, SMS or QR code. The secondary applicant gets their own interface that only contains their parts of the application, created on the same React component library as the main applicant interface, but filtered based on the workflow state and the definition of the role of the secondary applicant. [16]

4.2 State Synchronization And Process Integrity Mechanisms

Given a distributed computation, the workflow integrity problem is defined as follows. How can the backend of the protocol prevent the joint account application from being finalized until all parties in the workflow have finished their respective sub-workflows, regardless of the order in which the sub-workflows are finished? The synchronization mechanism at the workflow orchestration level tracks the finish of each party, and prevents the finalize from succeeding until all have finished. [17]

The React front-end will display the statuses of all participants to the main applicant in real time, by either polling or push from the orchestration layer, showing what actions are still required of each participant. This reduces the support load of joint applications. For example, if customers can see that their co-applicant has yet to complete identity verification, they can forward their inquiry to their co-applicant, and are not required to call support to inquire about the status of the application. The guarantee makes it impossible to skip finalization by dropping the pending message after the orchestration layer has completed for all participants.

Table 3: Multi-Applicant Workflow State Machine

Participant	Step Status	Sync State	System Response
Primary applicant	All sections submitted; awaiting secondary	Waiting	Lock finalization; display secondary status panel
Secondary applicant	Invitation accepted; identity pending	In progress	Maintain lock; update primary's status display
Secondary applicant	Identity verified; consent signed	Complete	Notify orchestration layer; advance sync state

Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Participant	Step Status	Sync State	System Response
Both participants	All checks confirmed by backend	Fully synchronised	Enable finalization control; proceed to account creation
Secondary applicant	Identity verification failed	Exception	Branch: offer re-invitation or convert to individual product
Any participant	Session expired before completion	Interrupted	Persist state; generate resumption token; notify primary

Table 3. Participant completion states and corresponding system responses in distributed joint account workflows. (Author's own analysis)

WORKFLOW CONTINUITY AND FAILOVER MECHANISMS

5.1 Device Failure And Session Recovery In Branch Environments

With branch assisted account origination, device failure or session interruption are no longer an issue. After all, if a consumer applies for an account in a branch, the application on the device can run out of battery, lose connectivity or need to be handed off to another device, once the branch visit is over. In all these cases on the legacy system there would have been a need for the representative to start from the beginning of the process, which was associated with customer dissatisfaction and high representative costs. [1][2]

Server-side session serialization also reduces this failure mode. By treating the state of the workflow as a persistent data entity and writing it to permanent storage with the customer session id tag at workflow milestones (end of a completed section, identity verification and selection of a funding source), the session state is always available to the user at their request on any device the customer or representative authenticates on, even if the active device crashes or the user disconnects. In these cases, the React application will load the session workflow from the last step saved to storage, filling in the completed steps with valid data, and exposing the remaining steps. [3][4]

Table 4: Session Recovery Scenarios

Trigger	Recovery Mechanism	Data Integrity Outcome
Browser crash (customer device)	Session serialized at last step; token-based resumption	No data loss; resumes at last validated step
Network interruption mid-form	Optimistic UI with server-side state sync on reconnection	Partial inputs preserved; validation re-runs on reconnect
Device handoff (customer to branch)	Secure short-lived token; representative authenticates and retrieves state	Full session transferred; no re-entry of submitted data
Branch device failure	State persisted centrally; any branch terminal can resume	Session recoverable from any authenticated enterprise device
Participant timeout (joint workflow)	State preserved; re-invitation issued; primary notified	Joint transaction preserved; no restart required

Table 4. Session interruption triggers, recovery mechanisms, and data integrity outcomes. (Author's own analysis)

Journal of Information Systems Engineering and Management
2026, 11(3s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

5.2 Bypassing Legacy Systems and Ensuring Data Consistency

One of the biggest operational risks with a partially migrated system is the temptation (and sometimes necessity) to cause customers, whom the new system cannot complete a step for, to fall back to the legacy application. If this becomes a common occurrence, it effectively negates the migration by keeping the legacy system's user population alive and hence, the decommissioning timeline. More critically, the pass-through of session state data from the new system to the legacy fallback may involve data integrity issues: due to minor differences in data schemas or data validation/business logic between the new and the fallback systems, records opened in one system but completed in the other may not be consistent. [5][6]

Session recovery as described above allows the elimination of legacy fallback, by providing a within-system recovery path for device failure, handoff between branches, and takeover of a representative. When all of these legacies can be dealt with within the new system, the incentive for maintaining active legacy fallback is reduced, and the routing layer can be modified to route around legacy paths as they become redundant. In addition, the clean decommission process allows the migration to eventually reach a fully modern architecture, rather than a hybrid of modern and legacy systems that are always in a decommission-pending state. [7][8]

CONCLUSION

The migration of a customary jQuery banking application to React components is another example of the Strangler Fig pattern in practice. With this approach, a zero-downtime application modernization at enterprise scale becomes possible with an architectural approach that also meets special enterprise operational requirements for regulated financial applications. The routing layer for incremental feature-by-feature replacement provides an added benefit of the ability to continuously roll back features throughout the migration. To support joint account origination scenarios, retrofitted jQuery code was superseded with a React component library for omnichannel consistency, real-time validation, and distributed workflows.

This distributed onboarding and state synchronization pattern provides a better answer to multi-party workflows compared to legacy systems, driving higher adoption rates and exposing more customers to the joint financial products and capabilities. Session recovery eliminates the restart-from-the-beginning failure mode of a customary legacy branch environment for the customer and the banker. Such architectural decisions can be classified as business-driven technical standards that constitute an archetype for the modernization of enterprise banking platforms favoring a balance between operational continuity and technical advancement by treating the decommissioning of the legacy platform as the target state to be achieved along a journey, rather than as a standalone risk event.

REFERENCES

[1] Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," IEEE Software, vol. 33, no. 3, pp. 42-52, 2016. DOI: 10.1109/MS.2016.64. [Online]. Available: https://ieeexplore.ieee.org/document/7436659

[2] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in Proc. DEVOPS 2018, Springer LNCS 11350, 2019, pp. 128-141. DOI: 10.1007/978-3-030-06019-0_10. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-06019-0_10

[3] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in Proc. IEEE SCC, 2016, pp. 813-818. DOI: 10.1109/SCC.2016.113. [Online]. Available: https://ieeexplore.ieee.org/document/7557535

[4] N. Dragoni et al., "Microservices: Yesterday, today, and tomorrow," in Present and Ulterior Software Engineering, Springer, 2017, pp. 195-216. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12

[5] Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in Proc. 6th Int. Conf. Cloud Computing and Services Science (CLOSER), 2016, pp. 137-146. [Online]. Available:

https://dl.acm.org/doi/10.5220/0005785501370146

[6] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," ACM SIGAPP Appl. Comput. Rev., vol. 17, no. 4, pp. 29-45, 2018. DOI: 10.1145/3183628.3183631. [Online]. Available: https://dl.acm.org/doi/10.1145/3183628.3183631

[7] Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," IEEE Software, vol. 35, no. 3, pp. 56-62, 2018. DOI: 10.1109/MS.2018.2141031.

[8] M. Waseem, P. Liang, M. Shahin, A. Ahmad, and A. R. Nasab, "On the nature of issues in five open source microservices systems," in Proc. 25th Int. Conf. EASE, 2021, pp. 201-210. DOI: 10.1145/3463274.3463337. [Online]. Available: https://dl.acm.org/doi/10.1145/3463274.3463337

[9] 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

[10] 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.

[11] 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] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in Proc. ASPLOS, 2019, pp. 3-18. DOI: 10.1145/3297858.3304013. [Online]. Available: https://dl.acm.org/doi/10.1145/3297858.3304013

[13] L. Zhu, L. Bass, and G. Champlin-Scharff, "DevOps and its practices," IEEE Software, vol. 33, no. 3, pp. 32-34, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7458765

[14] R. Heinrich et al., "Performance engineering for microservices: Research challenges and directions," in Proc. 8th ACM/SPEC ICPE Companion, 2017, pp. 223-226. DOI: 10.1145/3053600.3053653. [Online]. Available: https://dl.acm.org/doi/10.1145/3053600.3053653

[15] 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

[16] C. Esposito, A. Castiglione, and K.-K. R. Choo, "Challenges in delivering software in the cloud as microservices," IEEE Cloud Computing, vol. 3, no. 5, pp. 10-14, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7742281

[17] 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