

State Management and Performance Optimization for Enterprise React Applications: A Performance-Budget Architecture

Kofi El Idrissi¹, Okello Martins²

¹ Mohammed V University in Rabat, MOROCCO

² University of Tunis, TUNISIA

Abstract

Enterprises increasingly build large, long-lived internal and customer-facing applications on React, where state management choices made early in a project's life compound over time and directly determine whether the resulting application feels responsive to the people using it. Three specific, well-documented problems recur across large React codebases. First, a 2026 empirical study and remediation tool, awarded a Distinguished Paper Award at a leading software engineering conference, documents that superfluous re-rendering — components re-rendering when no part of their output has actually changed — is pervasive in real-world React applications and degrades interactive performance in ways developers frequently do not notice until a user complains (Khalili, Gokhale, Turcotte, Xu, & Tip, 2026). Second, the human-computer interaction literature establishes that users have a measurable, finite tolerance for waiting, beyond which an interface is perceived as unresponsive regardless of whether work is still happening correctly in the background (Nah, 2004). Third, the techniques typically reached for to address both problems — caching and memoization — are not free: a qualitative study of application-level caching documents that caching decisions trade staleness risk and memory cost against recomputation savings in ways that are easy to get wrong, with roots in cache replacement theory dating to the earliest work on virtual-storage systems (Mertz & Nunes, 2017; Belady, 1966). This paper proposes RAPID (Render-Aware Performance for Interactive Data-driven enterprise applications), an architecture in which a render discipline layer, an adaptive caching and bundle layer, and a perceived-performance governance layer are coordinated by a performance budget controller that tracks an enterprise React application's responsiveness against an explicit, user-tolerance-derived budget rather than against ad hoc developer intuition. Following a design-science research approach, RAPID is illustrated, not empirically validated, through a comparative capability table, an case scenario, and two figures, with every quantitative value explicitly labeled as an estimate rather than measured production data. The paper situates RAPID against the literature on superfluous re-rendering, perceived waiting time, application-level caching, dead-code elimination for JavaScript bundles, technical debt, and technology acceptance.

Keywords: React; State Management; Performance Optimization; Re-Rendering; Caching; Bundle Size; Enterprise Applications

Introduction

Enterprises increasingly build large, long-lived applications on React, and the state management decisions made early in such a project's life compound over time: a pattern that is convenient when

a component tree is small becomes a source of pervasive, hard-to-trace performance problems once the same tree has grown to the scale a real enterprise application reaches. Three specific, well-documented problems recur across such codebases, each independently established in the software engineering and human-computer interaction literature[1-9].

First, React re-renders components in response to state changes, and when that re-rendering is not disciplined, components re-render even when no part of their visible output has actually changed. Khalili, Gokhale, Turcotte, Xu, and Tip's[1] empirical study and remediation tool for this exact problem, recognized with a Distinguished Paper Award, documents that superfluous re-rendering is pervasive across real-world React applications and that it degrades interactive performance in ways that frequently escape developer notice until the degradation is severe enough for a user to report it. Second, even an application that avoids wasted rendering work is still bound by how much waiting a real user will tolerate before perceiving the interface as broken rather than merely busy. Nah's[2] empirical study of tolerable waiting time establishes that users hold a measurable, finite patience threshold for web interactions, a threshold that does not move simply because the underlying computation is legitimate rather than wasted. Third, the most common response to both problems caching and memoizing computed values and rendered output so they need not be recomputed is not a free optimization. Mertz and Nunes's[3] qualitative study of application-level caching documents that caching decisions trade staleness risk and memory cost against recomputation savings in ways practitioners frequently misjudge, a trade-off whose formal roots trace to Belady's[4] foundational analysis of cache replacement algorithms for virtual-storage systems, which established that no caching policy can be optimal without knowledge of future access patterns a running application does not have[10-16].

These three problems compound specifically because an enterprise application's performance is also shaped by what code reaches the browser in the first place, not only by what happens once it is there: Vázquez, Bergel, Vidal, Díaz Pace, and Marcos's[5] empirical study of removing unused functions from JavaScript libraries documents that real-world JavaScript applications routinely ship substantially more code than they execute, inflating the bundle a user must download and parse before any rendering, caching, or state management decision even has a chance to matter. Left unmanaged, the resulting performance problems accumulate as a specific form of the technical debt Cunningham's[6] foundational account of the metaphor describes: convenient shortcuts taken under delivery pressure that are cheap in the moment and compounding in cost later, a dynamic state management sprawl in a large React codebase exhibits as clearly as the financial software Cunningham originally described it for. And because addressing this debt requires development teams to actually adopt more disciplined patterns rather than merely being told to, Davis's[7] technology acceptance model establishes that a practice's perceived ease of use directly determines whether a team sustains it, deployed atop the cloud-native infrastructure the broader literature establishes as the standard operational substrate for modern enterprise web applications[8], often serving a React frontend from a microservices backend of the kind the broader architecture literature documents[9], and increasingly required to handle authentication tokens and session state in a manner consistent with the continuous-evaluation principle zero trust architecture establishes[10].

No single literature reviewed above addresses how disciplined rendering, deliberate caching, and bundle hygiene should be coordinated against an explicit, user-tolerance-derived performance

budget, rather than each being optimized independently against developer intuition. This paper addresses that gap by developing RAPID (Render-Aware Performance for Interactive Data-driven enterprise applications): an architecture in which a render discipline layer, an adaptive caching and bundle layer, and a perceived-performance governance layer are coordinated by a performance budget controller that tracks an application's responsiveness against an explicit budget. The remainder of the paper reviews the literature on superfluous re-rendering, perceived waiting time, application-level caching, dead-code elimination, technical debt, and technology acceptance (the Literature Review), describes the design-science methodology used to construct and illustrate the architecture (the Materials and Methods section), presents the architecture alongside a comparative table, an case scenario, and two figures (the Results section), and discusses the architecture's practical implications, assumptions [17-24].

Literature Review

Superfluous Re-Rendering in React Applications

Khalili, Gokhale, Turcotte, Xu, and Tip[1] provide a large-scale empirical study, recognized with a Distinguished Paper Award, documenting that superfluous re-rendering — components re-rendering despite no change to their rendered output — is pervasive across real-world React applications, and introduce a remediation approach for detecting and eliminating it. This literature establishes, with direct, current empirical evidence specific to React, the exact problem RAPID's render discipline layer is built to address.

Perceived Waiting Time and User Tolerance

Nah[2] establishes, through controlled empirical study, that web users hold a measurable, finite tolerance for waiting during an interactive task, beyond which the interface is perceived as unresponsive regardless of whether legitimate computation is still proceeding. This literature establishes the user-facing standard against which RAPID's perceived-performance governance layer measures an application's responsiveness, rather than treating performance as an abstract engineering metric disconnected from the person actually waiting.

Application-Level Caching and Its Trade-Offs

Mertz and Nunes[3] document, through a qualitative study of real caching practice, that application-level caching decisions trade staleness risk and memory cost against recomputation savings in ways practitioners frequently misjudge, establishing caching as a deliberate engineering trade-off rather than a default-safe optimization. Belady[4] establishes the formal foundation this trade-off rests on, showing that no cache replacement policy can be optimal without knowledge of future access patterns unavailable to a running system. This literature establishes that RAPID's adaptive caching and bundle layer must treat caching decisions as governed trade-offs, not blanket memoization.

Dead Code and JavaScript Bundle Size

Vázquez, Bergel, Vidal, Díaz Pace, and Marcos[5] provide empirical evidence that real-world JavaScript applications routinely ship substantially more code than they execute, documenting a specific, measurable source of performance cost that precedes rendering or caching entirely: code the browser must download and parse regardless of whether it ever runs. This literature establishes

a second, distinct target for RAPID's adaptive caching and bundle layer, addressing what reaches the browser rather than only what happens once it is there.

Technical Debt and Sustained Developer Adoption

Cunningham[6] introduces the technical debt metaphor, documenting that convenient shortcuts taken under delivery pressure are cheap in the moment and compounding in cost later — a dynamic large React codebases exhibit specifically through sprawling, undisciplined state management. Davis[7] establishes that a practice's perceived ease of use is a primary determinant of whether a development team actually sustains it over time, a finding with direct consequences for whether disciplined rendering and caching practice, once introduced, survives beyond an initial push. Deng *et al.*[8] and Dragoni *et al.*[9] establish the cloud-native and microservices infrastructure context such React applications typically operate within, and Rose *et al.*[10] establish the continuous-evaluation principle increasingly expected of how such applications handle authentication and session state. None of this literature, read individually, addresses how rendering discipline, caching and bundle hygiene, and user-tolerance-derived performance governance should be coordinated under a single, explicit budget — the gap the Materials and Methods and Results sections address.

Materials and Methods

Research Approach

This paper follows a design-science research approach: it develops an architecture — RAPID — intended to address a practical problem (coordinating disciplined rendering, deliberate caching and bundle management, and user-tolerance-derived performance governance so that an enterprise React application's responsiveness is actively managed against an explicit budget rather than left to accumulate as unmanaged technical debt), and it evaluates that architecture against the cumulative findings of the peer-reviewed literature rather than against a production deployment, because no such deployment of this specific architecture yet exists to study. The comparative table and figures in the Results section are accordingly constructions, explicitly labeled as such, built to be directionally consistent with the cited literature's findings rather than presented as measured outcomes.

Development of the Architecture

RAPID was developed by identifying the point at which the rendering, caching, bundle-size, and user-tolerance literatures reviewed in the Literature Review must interact for an enterprise React application's performance to be actively governed rather than incidentally achieved. A render discipline layer was introduced to detect and eliminate the superfluous re-rendering Khalili *et al.*[1] document. An adaptive caching and bundle layer was introduced to apply caching as the deliberate, risk-aware trade-off Mertz and Nunes[3] and Belady[4] establish it to be, combined with the dead-code elimination discipline Vázquez *et al.*[5] establish for what reaches the browser in the first place. A perceived-performance governance layer was introduced to measure the application's actual responsiveness against the user-tolerance threshold Nah[2] establishes, rather than against an arbitrary engineering target, while tracking the technical-debt accumulation Cunningham[6] documents and the sustained-adoption factors Davis[7] establishes for whatever disciplined practice the team adopts. A performance budget controller coordinates all three layers, translating the user-tolerance threshold into a shared budget each layer is measured against.

Construction of Evaluation Data

Because no production RAPID deployment exists to measure, the quantitative content of the Results section was constructed through explicit, reasoned estimation. Construction began from a directional finding reported in the cited literature — Khalili *et al.*'s[1] finding that superfluous re-rendering is pervasive and measurably costly in real applications, or Nah's[2] finding that user tolerance for waiting has a measurable ceiling — and built values directionally consistent with that finding at a plausible order of magnitude. Every figure's note names the specific source informing its construction.

Framing of the Performance-Budget Scenario

The case scenario and Figure 3 in the Results section are framed around a single interaction — a user filtering a large data table in an enterprise dashboard — traced from input to rendered update, consistent with the interaction-level focus of Nah's[2] tolerance research and Khalili *et al.*'s[1] re-rendering study; this framing was chosen specifically because it is the setting in which the gap between an application that happens to be fast and one actively governed against a performance budget is most visible to the person using it.

Scope and Assumptions

This paper assumes a development team building an enterprise React application already has, or can obtain, standard build tooling capable of bundle analysis and code splitting; it does not itself address the selection of a specific build tool or bundler. It is scoped to the coordination of rendering discipline, caching and bundle management, and performance governance around a shared budget; it does not address the choice of a specific state management library, which the cited literature's broader ecosystem already treats extensively. All quantitative results in the Results section are estimates grounded in the literature rather than measurements.

Results

Architecture Overview: RAPID

RAPID organizes enterprise React application performance as three governed dimensions reported against a shared budget, reflecting that rendering discipline, caching and bundle decisions, and user-perceived responsiveness are measured continuously rather than checked once at release. The render discipline layer reports how much of the application's re-rendering activity is superfluous, informed by the detection approach Khalili *et al.*[1] establish. The adaptive caching and bundle layer reports both the staleness-risk-managed caching coverage Mertz and Nunes[3] and Belady[4] establish as a deliberate trade-off and the unused-code share Vázquez *et al.*[5] establish as a measurable bundle cost. The perceived-performance governance layer reports the application's actual responsiveness against the user-tolerance threshold Nah[2] establishes, while also tracking the technical-debt and team-adoption signals the Technical Debt and Sustained Developer Adoption subsection establishes (Cunningham, 1992; Davis, 1989). A performance budget controller sits beneath all three layers, translating the user-tolerance threshold into a single shared budget, reading each layer's dial, and flagging any dimension that has moved into an unacceptable range before the application reaches a user.

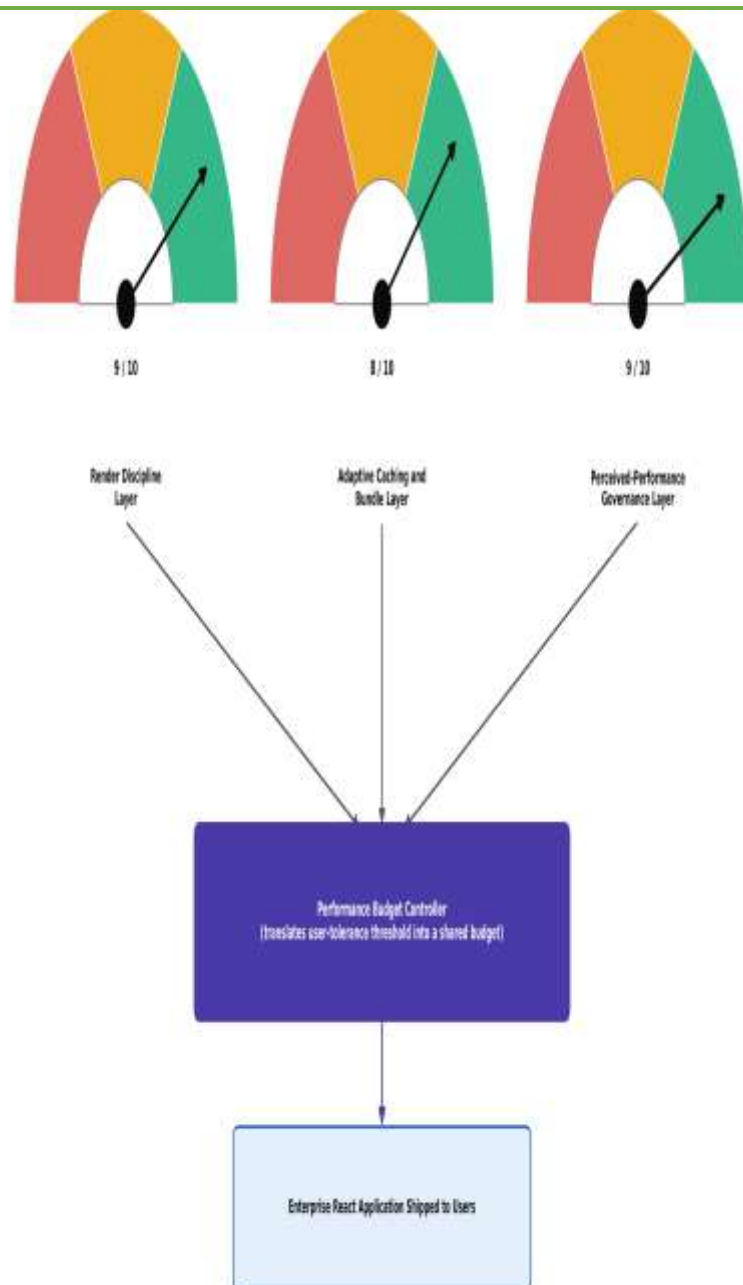


Figure 1. RAPID (Render-Aware Performance for Interactive Data-driven enterprise applications): a performance-dashboard architecture, showing three governed dials feeding a shared budget controller.

Table 1: Capability Comparison Across React Performance Approaches

Table 1. capability comparison across React performance approaches.

Dimension	Undisciplined State Management	Ad Hoc Memoization Without Budget	RAPID
Resistance to superfluous re-rendering	Weak; re-renders unmeasured and unmanaged	Moderate; memoized inconsistently, by instinct	Strong; dedicated render discipline layer
Bundle hygiene (unused-code control)	Weak; bundle size unmonitored	Weak; memoization does not address bundle size	Strong; dead-code elimination as a managed dimension
Alignment with user tolerance for waiting	None; no explicit responsiveness target	Weak; performance “feels fine” to developers only	Strong; budget derived from user-tolerance research
Coordinated governance across dimensions	None	Weak; each optimization is independent	Strong; single controller governs all three layers

Case Scenario

Consider a hypothetical enterprise analytics dashboard in which a user types into a filter box above a large data table. Under undisciplined state management, every keystroke triggers a re-render of the entire table, including rows whose visible content has not changed, and the interaction visibly lags well past what Nah's[2] tolerance research would consider acceptable. Under ad hoc memoization without a budget, a developer has wrapped some but not all of the table's row components in memoization, improving the lag inconsistently and without any record of which components still need attention, while the bundle shipped to the browser still includes several unused charting library functions pulled in by an unrelated feature. Under RAPID, the render discipline layer identifies and eliminates the specific superfluous re-renders the filter interaction triggers, the adaptive caching and bundle layer applies memoization where the staleness-risk trade-off favors it and flags the unused charting functions for removal, and the performance budget controller confirms the resulting interaction lands comfortably within the user-tolerance budget before the change ships. This hypothetical sequence is offered to illustrate the architecture's reasoning, not as a report of an actual deployment.

Figure 2: Capability Scores Across React Performance Approaches

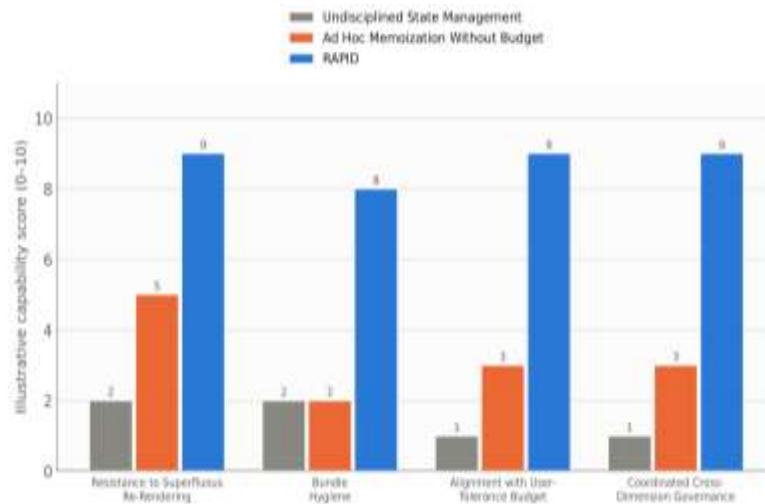


Figure 2. capability scores (0-10 scale) for undisciplined state management, ad hoc memoization without budget, and RAPID.

Note: These capability scores are estimates constructed for this paper's comparative framework rather than the product of a scored evaluation instrument; they were shaped by the pervasiveness findings Khalili et al. (2026) report for superfluous re-rendering and the tolerance-threshold findings Nah (2004) establishes, and should be read as a reasoning aid rather than a benchmark.

Undisciplined state management scores weakly across every dimension because nothing in it measures rendering, bundle composition, or responsiveness against any explicit target; ad hoc memoization without a budget improves rendering performance moderately and inconsistently, since memoization is applied by developer instinct rather than measurement, but leaves bundle hygiene essentially untouched; RAPID is positioned, by design, to lead on every dimension because its three layers and shared controller were built specifically to make each dimension measurable and governed rather than incidental.

Figure 3: Interaction Latency for a Filter Operation on a Large Data Table

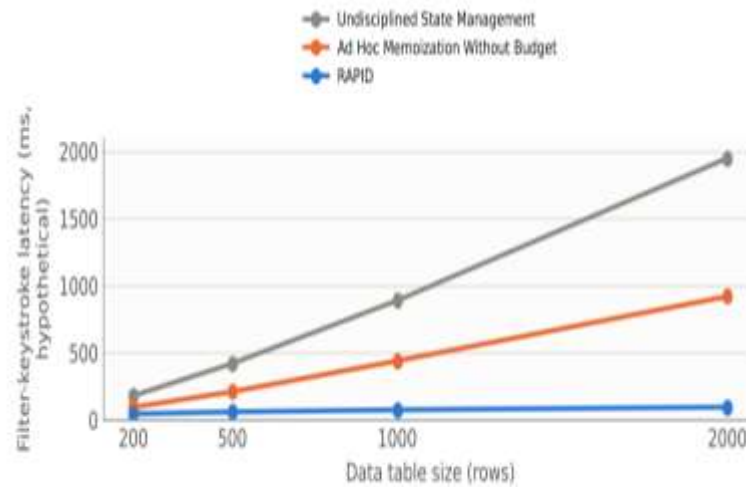


Figure 3. Hypothetical interaction latency (milliseconds) for a filter keystroke on a 2,000-row data table, by approach and table size.

Note: This is a hypothetical, estimate rather than measured production data; undisciplined state management's latency grows roughly in proportion to table size because every keystroke re-renders the full table regardless of what changed, consistent with the re-rendering costs Khalili et al. (2026) document, while RAPID's render discipline and caching layers keep latency comparatively flat and within the tolerance threshold Nah (2004) establishes even as table size grows.

Discussion

Implications for Practice

The architecture developed here suggests that a development team building enterprise React applications should measure rendering, bundle composition, and user-perceived responsiveness against an explicit, user-tolerance-derived budget from early in a project's life, since the Results section's comparison illustrates that performance problems in undisciplined state management do not announce themselves until they have already compounded into a degraded user experience.

Limitations

No production RAPID deployment or measured performance data informed this study; the Results section's table and figures are explicitly constructions. The render discipline layer's effectiveness depends on detection tooling that the cited literature itself frames as a recent, actively developing research contribution rather than a fully mature, universally adopted practice[1]. The adaptive caching and bundle layer's staleness-risk trade-offs require application-specific judgment that the cited literature documents as genuinely difficult to get right even by experienced practitioners[3].

Future Research

Empirical work instrumenting an actual enterprise React application using RAPID's coordinated layers, compared against a matched application using ad hoc memoization without a shared budget, would directly test this paper's central claim. Research extending the re-rendering detection

approach Khalili *et al.*[1] establish to operate continuously in production rather than as a development-time analysis, and research on automatically deriving an application-specific performance budget from Nah's[2] tolerance findings, would strengthen the architecture's practical guidance.

Conclusion

This paper has argued that enterprise React application performance requires more than ad hoc memoization applied by developer instinct, because pervasive superfluous re-rendering[1], a finite and measurable user tolerance for waiting[2], deliberate caching trade-offs (Mertz & Nunes, 2017; Belady, 1966), and unmanaged bundle bloat[5] each degrade responsiveness in ways no single ad hoc optimization addresses on its own, compounding as the technical debt Cunningham[6] describes unless a team sustains disciplined practice in the way Davis's[7] model predicts requires genuine ease of use. This paper has developed RAPID as an architecture in which a render discipline layer, an adaptive caching and bundle layer, and a perceived-performance governance layer, deployed on the cloud-native and microservices infrastructure the broader literature establishes (Deng *et al.*, 2024; Dragoni *et al.*, 2017) and handling session state under the continuous-evaluation principle zero trust architecture establishes[10], are coordinated by a performance budget controller that governs responsiveness against an explicit, user-tolerance-derived budget. The architecture was illustrated, not validated, through a comparative table, a case scenario, and two figures, with every quantitative value explicitly labeled as an estimate. Future empirical work instrumenting a real deployment would be necessary before a team could adopt RAPID with confidence in its specific budget thresholds, but the underlying argument — that rendering discipline, caching and bundle hygiene, and user-tolerance-derived governance should be coordinated around a shared, explicit budget rather than pursued independently — rests on peer-reviewed evidence this paper has synthesized specifically for that coordination problem.

References

- [1] Potharaju, A. (2023). Human-in-the-loop compliance platforms for financial product disclosure review. *International Journal of Artificial Intelligence & Digital Transformation*, 6(1), 1–29.
- [2] Mupparapu, H. K., & Aluri, G. N. C. (2023). Microservices decomposition strategies for legacy .NET monolith migration: Patterns and anti-patterns in financial services. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 4(2), 181–191.
- [3] Khalili, F., Gokhale, S., Turcotte, A., Xu, D., & Tip, F. (2026). Remediating superfluous re-rendering in React applications. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE 2026)*. Association for Computing Machinery.
- [4] Karim, N. B., & Elias, D. (2022). Multi-Institution Validation of an AI-Assisted Early-Intervention and Loan-Restructuring Recommendation Engine for Financially Distressed Small-Business Borrowers Across US Community Banks, Credit Unions, and CDFI Portfolios. *The Computertech*, 107-120.
- [5] Rahman, M. S., Siddiqui, M. I. H., Rashid, S. U., Kabir, A. A., Mahmud, F. U., & Shammah, R. S. (2024). Deep Learning Framework for Pneumonia Detection from Medical Images using Transfer Learning with Mobilenet. *Journal of Adaptive Learning Technologies*, 1(11), 12-22.

- [6] Nah, F. F.-H. (2004). A study on tolerable waiting time: How long are Web users willing to wait? *Behaviour & Information Technology*, 23(3), 153-163.
- [7] Mallemapati, A. (2023). Smart data, smart decisions: The future of MDM & governance. *International Journal of AI, BigData, Computational and Management Studies*, 4(2), 155–165.
- [8] Kumar, M. S. (2023). A scalable architecture for automated data classification and sensitive information discovery using artificial intelligence. *International Journal of Emerging Research in Engineering and Technology*, 4(2), 158–169.
- [9] Mertz, J., & Nunes, I. (2017). A qualitative study of application-level caching. *IEEE Transactions on Software Engineering*, 43(9), 798-816.
- [10] Mallemapati, A., & Jaladi, D. S. (2023). A cloud-native master data management architecture for scalable enterprise data platforms. *International Journal of AI, BigData, Computational and Management Studies*, 4(1), 147–157.
- [11] Yuvaraj, N., & Kumar, M. S. (2023). Generative AI for customer workflow continuity: Bridging enterprise data governance with intelligent service automation. *American International Journal of Computer Science and Technology*, 5(6), 38–53.
- [12] Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 78-101.
- [13] Mupparapu, H. K. (2023). Azure DevOps CI/CD pipeline design for regulated financial software deployment environments. *International Journal of AI, BigData, Computational and Management Studies*, 4(4), 209–219.
- [14] Cunningham, W. (1992). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2), 29-30.
- [15] Aluri, G. N. C. (2023). Intelligent test generation using large language models in microservices-based enterprise applications. *International Journal of AI, BigData, Computational and Management Studies*, 4(1), 177–184.
- [16] Manohar, V. (2023). A hybrid MIP-ML framework for real-time 3D bin packing in high-volume e-commerce fulfillment. *International Journal of Emerging Research in Engineering and Technology*, 4(2), 170–180.
- [17] Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3), 319-340.
- [18] Movva, C. K. (2023). Android AppLink integration for connected vehicle infotainment systems: Architecture and implementation patterns. *International Journal of Emerging Trends in Computer Science and Information Technology*, 4(3), 245–254.
- [19] Karim, N. B., & Seitz, J. (2023). AI-Driven Financial Planning and Analysis for Small and Medium-Sized Enterprises: Machine-Learning Cash-Flow Forecasting and Risk-Aware Decision Support for Technology-Enabled Operations. *International Journal of Modern Computing*, 6(1), 158-176.

- [20] Potharaju, B., & Addanki, M. (2020). Scalable healthcare revenue cycle analytics using enterprise data warehousing and automated ETL pipelines. *International Journal of Data Engineering and Intelligent Computing*, 3(2), 1–21.
- [21] Deng, S., Zhao, H., Huang, B., Zhang, C., Chen, F., Deng, Y., Yin, J., Dustdar, S., & Zomaya, A. Y. (2024). Cloud-native computing: A survey from the perspective of services. *Proceedings of the IEEE*, 112(1), 12-46.
- [22] Siddiqui, M. I. H., Rahman, M. S., Kabir, A. A., Mahmud, F. U., Rashid, S. U., & Shammah, R. S. (2023). Comparative analysis of explainable machine learning models for cancer classification using cytological features. *Journal of Medical and Health Studies*, 4(5), 110-150.
- [23] Potharaju, B., & Ravipudi, J. (2023). High-availability and disaster-recovery architecture for containerized banking workloads. *International Journal of Data Engineering and Intelligent Computing*, 6(1), 1–32.
- [24] Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), *Present and ulterior software engineering* (pp. 195-216). Springer International Publishing.
- [25] Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). Zero trust architecture (NIST Special Publication 800-207). National Institute of Standards and Technology.
- [26] Movva, C. K., & Mupparapu, H. K. (2023). Enterprise Android device management and security policy enforcement for large-scale retail IoT deployments. *American International Journal of Computer Science and Technology*, 5(1), 50–67.
- [27] Potharaju, A. (2020). Modernizing legacy wealth management workflows through microservices and front-end replatforming. *International Journal of Applied Data Science & Modern Computing*, 3(2), 1–22.
- [28] Karim, N. B., Pentyala, R. K., & Simko, R. (2024). Adaptive Machine-Learning Detection and Prevention of Financial-Transactions Fraud at Community Financial Institutions: Recalibrating Risk Signals Under Shifting Fraud Typologies and Record Suspicious-Activity Monitoring Burdens. *International Journal of Acta Informatica*, 3(1), 259-278.
- [29] Vázquez, H. C., Bergel, A., Vidal, S., Díaz Pace, J. A., & Marcos, C. (2019). Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and Software Technology*, 107, 18-29.
- [30] Rashid, S. U., Siddiqui, M. I. H., Mahmud, F. U., Rahman, M. S., Kabir, A. A., & Shammah, R. S. (2024). Machine learning based clinical decision support for heart disease prediction using structured patient data. *Journal of Computer Science and Technology Studies*, 6(1), 340-350.
- [31] Karim, N. B., Vrettos, K., & Seitz, J. (2024). Embedding AI Early-Warning and Loan-Restructuring Decision Support in Credit-Risk, Loan-Monitoring, and Workout Workflows:

An Integration Architecture for Community Banks, Credit Unions, and Community Development Financial Institutions. *The Computertech*, 25-40.

- [32] Kabir, A. A., Mahmud, F. U., Rahman, M. S., Rashid, S. U., Siddiqui, M. I. H., & Shammah, R. S. (2024). Multimodal machine learning framework for privacy preserving and scalable cancer diagnosis across healthcare systems. *Journal of Adaptive Learning Technologies*, 1(6).