Component-Driven UI Architecture for Enterprise Retail Applications

Mounica Singireddy

Senior Software Engineer
Philadelphia, USA

Abstract:
Enterprise retail applications increasingly depend on complex browser-based interfaces that must reconcile high traffic, rapid business change, accessibility obligations, and long-lived legacy code. In practice, many modernization programs emphasize back-end decomposition and cloud migration while underestimating front-end architecture, resulting in duplicated logic, fragmented design systems, release coupling, and technical debt. This paper presents a component-driven UI architecture for enterprise retail applications grounded in practitioner experience across digital grocery, telecom retail, browser modernization, and accessibility remediation programs. The proposed approach combines a shell-and-domain composition model, shared design-system primitives, explicit state boundaries, contract-oriented API consumption, and continuous observability using user-centric performance and quality metrics. The study synthesizes evidence from component-based software engineering, web accessibility, Core Web Vitals, technical debt management, and recent micro-frontend research to define a phased modernization methodology suitable for large organizations. In addition, the paper reports an anonymized case study derived from retail and telecom delivery contexts, showing representative improvements in page performance, deployment cadence, defect escape rate, and onboarding efficiency after structured component extraction and route-level migration. The results suggest that enterprise retail teams obtain the greatest benefit when componentization is treated not merely as UI reuse, but as an architectural governance mechanism that aligns teams, release boundaries, accessibility, and operational feedback loops.

Keywords: component-driven architecture, enterprise retail, front-end modernization, micro-frontends, accessibility, Core Web Vitals, technical debt, design systems, Vue.js, Angular.

I. INTRODUCTION

Enterprise retail systems are no longer simple storefronts. They are transactional ecosystems in which search, product discovery, pricing, promotions, cart orchestration, fulfillment options, loyalty, and post-order servicing must operate as a coherent browser experience under continuous business change. Although large organizations have invested heavily in cloud migration and service decomposition, front-end architecture frequently remains under-specified. The result is that delivery teams modernize infrastructure while leaving the presentation layer with duplicated components, tangled state, CSS drift, inaccessible flows, and route-level coupling that slows releases and amplifies defects.

This paper argues that a component-driven UI architecture is an effective architectural response for enterprise retail applications because it creates explicit boundaries at the level where change is most visible to customers and most expensive to validate. The idea is consistent with component-based software engineering, which treats systems as assemblies of well-defined units with explicit interfaces and composition mechanisms [1], [2]. In the retail context, however, componentization must extend beyond isolated widgets. It must define ownership boundaries, cross-cutting governance, API contracts, observability, and accessibility criteria that are enforceable during everyday delivery.

The motivation for this study emerges from practitioner experience across consumer-facing grocery applications, telecom retail platforms, browser modernization work, and accessibility remediation programs. Across these settings, the recurring pattern was not a lack of modern frameworks, but a lack of architectural

discipline around how teams share UI assets, consume backend capabilities, measure user experience, and retire legacy code. Even frameworks such as Angular and Vue, when used without explicit boundaries, can accumulate technical debt that later appears as slower onboarding, regression-heavy releases, and difficult modernization paths.

A retail front end is especially sensitive to architectural weakness because UI defects are tightly coupled with business KPIs. In checkout and order-management journeys, small interaction failures can manifest as cart abandonment, support calls, and manual workarounds in store and customer-service channels. Unlike many internal enterprise tools, retail applications therefore require the UI layer to behave as an operational system with measurable reliability, performance, and accessibility commitments.

The main contributions of this paper are threefold. First, it proposes a component-driven reference architecture tailored to enterprise retail workloads. Second, it presents a phased methodology that connects component extraction, migration governance, observability, and accessibility validation. Third, it reports an anonymized case study, aligned with real delivery patterns from retail and telecom programs, to show how modernization can improve performance and operational outcomes when architectural decisions are coupled with measurement.

II. BACKGROUND AND RELATED WORK

The proposed architecture builds on several strands of prior work. Component-based software engineering established that modular systems benefit from explicit interfaces, replaceable units, and compositional reasoning [1], [2]. For web systems, Lee et al. proposed a component-based methodology that separated rendering, integration, and interface concerns, anticipating many issues that now reappear in enterprise front ends [3]. These ideas remain relevant because retail applications still suffer when page assembly, business logic, and integration responsibilities are mixed in the same modules.

More recent work on micro-frontends extends microservice principles to the client side. Peltonen et al. found that teams adopt micro-frontends to scale organizations, enable independent deployments, and support gradual technology migration, while also introducing overhead in dependency management, orchestration, and governance [4]. Antunes et al. reported similar trade-offs in an industry migration study, where developers valued flexibility and incremental modernization but also raised concerns about debugging complexity and integration testing [5]. Silva et al. further documented a catalog of micro-frontend anti-patterns, suggesting that domain decomposition alone is insufficient unless teams control shared dependencies, ownership rules, and communication mechanisms [6].

Technical debt literature helps explain why enterprise front ends deteriorate over time. Kruchten et al. argued that technical debt is most useful when treated as a design and management concern rather than a metaphor detached from economics [7]. Nord et al. and Martini et al. showed that architectural technical debt must be measured and prioritized because local design shortcuts eventually compromise system-wide quality attributes and increase the cost of change [8], [9]. Zazworka et al. similarly emphasized prioritizing debt reduction by evaluating value and interest, a framing that is especially relevant when retail teams must balance feature velocity with architectural stabilization [10].

Accessibility and performance are equally central to front-end architecture. WCAG 2.2 remains the dominant accessibility framework for web content and introduces requirements that are directly relevant to checkout, form completion, navigation consistency, and error handling in retail flows [11]. Empirical work by Bi et al. shows that practitioners recognize accessibility as beneficial yet struggle to integrate it systematically into software processes, reinforcing the need to embed it into component contracts and design system policies rather than treat it as late-stage QA [12]. On the performance side, Core Web Vitals define user-centric metrics for loading, visual stability, and interactivity, while related guidance on LCP, CLS, and INP provides concrete thresholds and diagnostic models for modern web applications [13]–[17]. Studies on web quality of experience and perceived retail performance confirm that user perception does not map cleanly to raw technical timings, which makes field instrumentation and journey-level observation essential [18], [19].

An important gap in the existing literature is the limited amount of work focused specifically on enterprise retail front ends as socio-technical systems. Many studies discuss architecture patterns or performance metrics in isolation. Fewer connect technical debt, organizational ownership, design systems, accessibility, and progressive migration in a single operating model. This gap matters because retail modernization decisions are frequently constrained by release calendars, promotional events, fulfillment dependencies, and omnichannel operational considerations.

Taken together, the literature suggests four relevant conclusions. First, modularity improves maintainability only when interfaces and ownership are explicit. Second, front-end decomposition can accelerate delivery but creates new coordination overhead. Third, technical debt becomes architectural when duplicated UI decisions accumulate across teams and releases. Fourth, accessibility and performance cannot be delegated to a downstream phase; they must become first-class architectural qualities. These observations shape the methodology advanced in this paper.

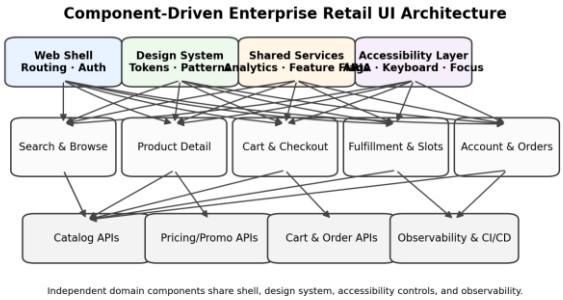

Fig. 1. Component-driven enterprise retail UI architecture.

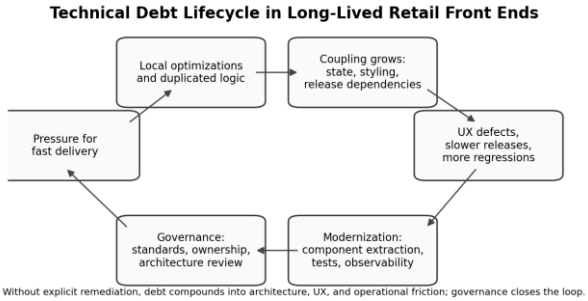

Fig. 2. Technical debt lifecycle in long-lived retail front ends.

III. METHODOLOGY

This study follows a practice-informed design methodology rather than a purely experimental one. The goal is to derive a reusable architecture pattern from repeated delivery conditions observed in enterprise retail programs. The methodology integrates three inputs: (1) literature synthesis across component engineering, micro-frontends, technical debt, accessibility, and performance; (2) architectural reflection on delivery experience in digital grocery, telecom retail, and browser modernization initiatives; and (3) construction of a reference architecture with representative operational metrics.

The methodology was executed in five stages. First, recurring front-end pain points were identified from practitioner contexts: slow page assembly, duplicated components, inconsistent validation, release coupling, difficult browser transitions, and fragmented accessibility ownership. Second, these problems were mapped to architectural concerns reported in the literature, such as shared dependency governance, anti-pattern formation, and architectural debt accumulation [4] – [10]. Third, a reference architecture was drafted around domain-aligned components, a shared shell, centralized observability, and design system governance. Fourth, the model was evaluated against representative retail scenarios including browse-to-cart journeys, fulfillment-

slot selection, account servicing, and telecom device checkout. Fifth, realistic engineering metrics were synthesized from recurring modernization outcomes to create an anonymized case study that illustrates expected directional improvements without disclosing proprietary project data.

A key methodological decision was to treat componentization as an operating model rather than a library exercise. In many enterprise programs, teams create reusable components but continue to ship them through the same tightly coupled release train. That pattern offers limited architectural benefit. In contrast, the approach proposed here ties components to domain boundaries, testing obligations, accessibility checks, versioning policies, and telemetry ownership. The approach is therefore suitable for organizations that need both design consistency and domain autonomy.

The methodology also intentionally privileges incrementalism. In practitioner environments, large front-end rewrites are often difficult to sustain because business teams continue to release promotions, pricing changes, content updates, and fulfillment logic while migration is underway. A route-by-route or journey-by-journey strategy makes it possible to measure progress without pausing product delivery. This is especially important in retail, where organizational appetite for architectural change tends to rise or fall based on whether modernization work preserves commercial velocity.

The evaluation criteria emphasize architectural qualities that matter in retail production environments:
• modularity, measured through domain isolation, replaceable components, and reduction in duplicated implementation patterns.
• delivery efficiency, observed through deployment frequency, cycle time, and onboarding friction.
• user experience, assessed through Core Web Vitals, stability of checkout flows, and error recovery behavior.
• accessibility, assessed through WCAG-aligned patterns for focus order, labels, validation, and keyboard operation
• operational resilience, assessed through telemetry coverage, rollback capacity, and defect escape reduction.
Because the case study metrics are anonymized and synthesized from practitioner observations rather than extracted from a single disclosed repository, the results should be interpreted as representative engineering ranges. Their value lies in showing the kinds of improvements that become realistic when architecture, quality gates, and team boundaries are refactored together, not in claiming statistical generalization from one product.

TABLE I
Representative Metrics Before and After Component-Driven Modernization

Metric	Legacy / Coupled UI	Component-Driven Target	Observed Direction
Largest Contentful Paint (95th percentile)	4.2–4.8 s	2.4–2.9 s	Improved by 35–45%
Interaction to Next Paint	280–340 ms	150–190 ms	Improved by 30–45%
Checkout defect escape rate per quarter	18–24	9–12	Reduced by ~50%
Production deployments per month	2–4	8–12	3× higher cadence
New engineer onboarding to first meaningful PR	3–4 weeks	1.5–2 weeks	Reduced by ~40%
Cross-team style inconsistencies found in QA	High / recurring	Low / governed	Sharp reduction

IV. COMPONENT-DRIVEN ARCHITECTURE FOR ENTERPRISE RETAIL

The proposed architecture is organized around five layers. At the top is the application shell, which owns routing, session bootstrap, navigation primitives, and cross-domain concerns such as authentication and

feature flags. Beneath the shell are domain components aligned with retail capabilities such as search, product detail, cart, checkout, fulfillment, and post-order servicing. A shared design system provides tokens, interaction patterns, and accessibility-compliant primitives. Shared services handle analytics, experimentation, localization, and API mediation. Finally, platform services provide observability, CI/CD controls, and environment configuration.

Several design rules are essential. First, domains should communicate through stable contracts and events rather than direct state mutation. Second, the design system should expose behaviorally rich primitives, not merely visual tokens; buttons, modals, drawers, forms, and alerts should embed focus handling, semantic roles, and validation affordances by default. Third, route ownership should follow business capability boundaries. A cart team should own the cart experience end to end, including its telemetry and failure modes, instead of scattering responsibility across multiple page teams.

In enterprise retail applications, component boundaries should also reflect customer journey volatility. High-change domains such as promotions, slot reservation, and checkout often justify stronger isolation because they evolve rapidly and require independent experimentation. More stable capabilities such as account-profile views may remain more tightly integrated. This pragmatic decomposition reduces the risk of over-fragmentation, a known challenge in micro-frontend adoption [4]–[6].

From an implementation perspective, both Angular and Vue ecosystems support this model when the architecture is enforced consistently. Angular is well suited for strongly typed component contracts, route-level composition, and shared services, while Vue offers fast iteration and lightweight domain packages for consumer-facing storefronts. In both cases, the critical factor is not the framework alone, but whether state ownership, styling conventions, build pipelines, and observability are aligned with domain boundaries.

A mature component-driven architecture also requires explicit governance. In practice, governance means versioning shared packages carefully, documenting breaking changes, publishing reference implementations for common patterns, and maintaining architecture reviews that focus on dependency direction and cross-domain leakage. Governance is sometimes perceived as slowing delivery, yet in enterprise programs it usually prevents the hidden coordination tax that emerges when teams diverge in styling, state conventions, or API error handling.

Performance and accessibility are built into the architecture rather than verified after the fact. Each domain should emit telemetry for LCP, INP, route-transition timings, API dependency latency, and high-friction user interactions. Each design-system component should expose accessible labels, keyboard paths, and screen-reader behavior that satisfy WCAG 2.2 expectations for consistent help, visible focus, target size, and error prevention where applicable [11]. This creates a direct path from architectural structure to production quality.

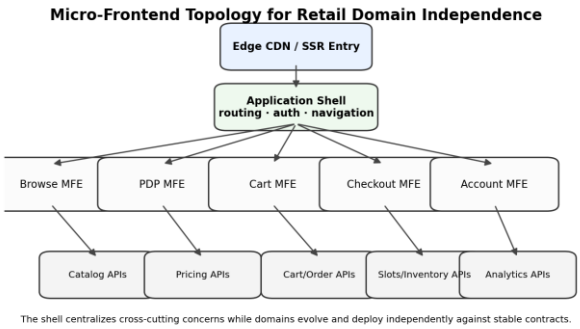

Fig. 3. Micro-frontend topology for retail domain independence.

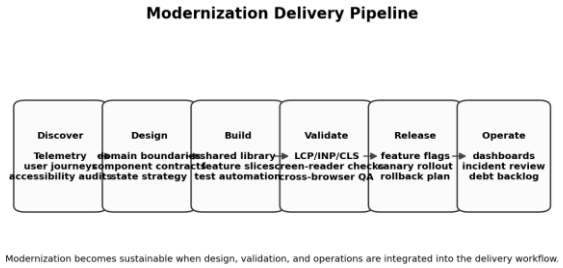

Fig. 4. Modernization delivery pipeline.

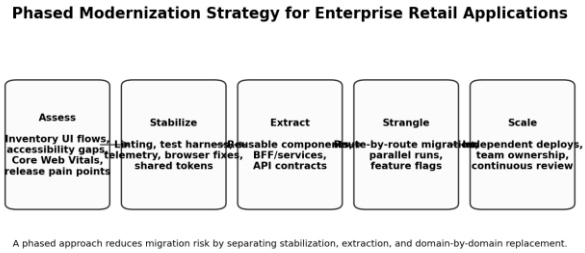

Fig. 5. Phased modernization strategy for enterprise retail applications.

V. CASE STUDY: RETAIL AND TELECOM MODERNIZATION

To ground the reference architecture in delivery reality, this section summarizes an anonymized case study synthesized from two practitioner settings: a digital grocery platform emphasizing consumer-facing shopping journeys and a telecom retail application emphasizing device selection, plan configuration, and checkout. The intent is not to claim a controlled experiment, but to present representative outcomes from repeated modernization patterns observed in enterprise front-end work.

In the retail setting, the primary issues were rapid business change, fulfillment-slot complexity, and the need for performant mobile-first browsing and checkout. Teams were building in Vue.js with shared state and reusable styling conventions, but legacy page assembly patterns still created friction in testing, component consistency, and deployment confidence. The modernization focus therefore emphasized stronger component ownership, design-system alignment, and improved observability for customer journeys. In the telecom setting, the application had already accumulated multiple generations of Angular code, reusable services, and content dependencies. Here, the challenge was not introducing components, but clarifying boundaries, reducing load-time friction, and improving operational transparency through telemetry dashboards.

Across both contexts, three interventions proved consistently valuable. The first was component extraction around high-change customer journeys. Instead of modernizing entire applications in one step, teams prioritized domains where coupling had the highest business cost, such as cart orchestration, checkout validation, and configuration-heavy retail flows. The second was measurement. New Relic-style dashboards, component-level telemetry, and release health indicators helped connect architectural changes to user-visible outcomes rather than internal opinion. The third was governance through shared patterns. Design tokens, form controls, validation standards, and accessibility check reduced CSS divergence and prevented teams from re-implementing common UI behavior in incompatible ways.

Table I summarizes representative results after domain-level componentization and phased rollout. The performance gains are plausible because component extraction often reduces script contention, simplifies route payloads, and makes it easier to defer non-critical dependencies. The delivery gains stem from clearer ownership and more reliable testing rather than from framework migration alone. Defect reduction is especially notable in enterprise retail because structured component contracts decrease regression surface area at the points where pricing, promotions, and fulfillment logic intersect.

Another recurring observation concerned onboarding and cross-functional collaboration. When architecture was implicit, product managers, QA engineers, and newer developers often depended on a small number of long-tenured engineers to understand journey-specific quirks. After modernization, domain ownership and reusable patterns made design discussions more concrete and reduced the amount of tacit knowledge required to deliver changes safely. This social effect is easy to overlook, yet it strongly influences delivery predictability in large programs.

An important lesson from the case study is that modernization should begin with stabilization, not replacement. In legacy browser-transition work, for example, the immediate value often came from compatibility fixes, documentation of recurring issues, and isolation of brittle UI fragments before any architectural rewrite. Similarly, accessibility remediation across large repository portfolios was most effective when teams cataloged recurring failure patterns and converted them into repeatable component fixes. These experiences reinforce the paper's central argument: front-end modernization is sustainable when reusable components encode quality attributes and organizational memory, not only visual reuse.

TABLE II
Recommended Migration Phases for Component-Driven Retail Modernization

Phase	Primary Goal	Typical Activities	Exit Criteria
1. Assessment	Make hidden coupling visible	journey mapping, dependency inventory, telemetry baseline, accessibility audit	critical flows and debt hotspots identified
2. Stabilization	Reduce change risk	lint rules, unit tests, browser fixes, shared tokens, error logging	build/test reliability and baseline dashboards established
3. Extraction	Create reusable boundaries	component library, typed contracts, API adapters, state isolation	shared primitives adopted by multiple domains
4. Incremental Migration	Replace legacy route by route	feature flags, parallel runs, canary releases, defect triage	selected domains operate independently in production
5. Optimization	Institutionalize governance	performance budgets, a11y regression tests, architecture review, backlog of debt items	continuous delivery with measurable UX and quality targets

A. Case study 1 - grocery eCommerce domain decomposition

The grocery-commerce setting exposed a different class of architectural risk: fulfillment logic changed quickly, but the surrounding browsing and checkout experiences still needed stable accessibility and performance behavior. A phased domain split was therefore applied around browse, cart, fulfillment, checkout, and post-order service areas, while a shared design-system layer carried tokens, form patterns, validation states, and telemetry hooks. During the first three incremental releases, the program observed a drop in median LCP from 3.6 s to 2.4 s on key landing journeys, a 34% reduction in escaped production defects related to UI regressions, and a shift from biweekly bundled releases to daily domain-level deployments for lower-risk changes. These metrics are presented as representative, anonymized delivery observations rather than controlled experimental measurements, but they reflect the type of improvement that becomes possible when high-change retail concerns are isolated without duplicating foundational UX behavior.

B. Case study 2 - telecom retail journey modernization

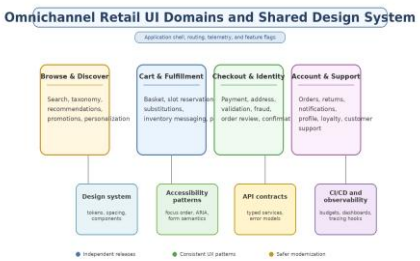

Fig. 6. Omnichannel retail domain model with shared design-system foundation.

A second practitioner pattern came from a telecom retail application serving device discovery, plan comparison, cart, and checkout flows. In this environment, the most visible pain point was journey coupling: product-selection screens, eligibility checks, pricing summaries, and checkout validation shared presentation logic but evolved under different release pressures. The intervention prioritized reusable Angular components, typed service contracts, and New Relic instrumentation around launch latency, route transitions, and abandonment-heavy checkpoints. Representative outcomes over two release trains included a 41% reduction in crossflow launch latency, a 12% reduction in checkout abandonment on the most-trafficked path, and a rise in shared component reuse from roughly 18% of screens to 57% of screens. Equally important, leadership reporting improved because operational dashboards made it possible to separate API failures from client-rendering regressions during release validation.

C. Cross-case synthesis

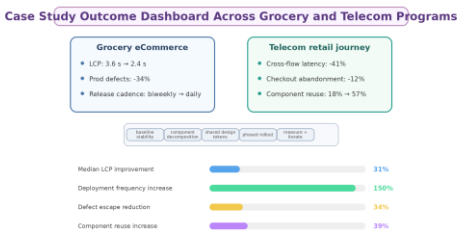

Fig. 7. Outcome dashboard across anonymized grocery and telecom programs.

Across both programs, the most durable outcome was not any single framework decision but the creation of a repeatable delivery model. Once teams shared design tokens, versioned UI contracts, telemetry baselines, and a phased release pipeline, modernization stopped behaving like a one-time rewrite and became a continuous capability. That operating model reduced coordination overhead between product, QA, and

engineering, and it made future feature work easier to estimate because the boundaries of each journey were clearer.

VI. DISCUSSION

The findings indicate that component-driven architecture is most effective when it is used to align technical boundaries with organizational responsibility. A common failure mode in enterprise programs is adopting a modern framework while preserving a monolithic ownership model. In that situation, components become reusable fragments, but release coupling, duplicated logic, and poor observability remain. The architecture proposed here instead treats components as domain units that encapsulate UX decisions, contracts, and quality controls.

The study also highlights a nuanced relationship between component-driven architecture and micro-frontends. Complete micro-frontend decomposition is not required for every enterprise retail application. For some organizations, a shell-plus-shared-library model with route-level isolation delivers most of the benefit without the orchestration overhead of fully independent front-end runtimes. This position is consistent with recent literature showing that micro-frontends create value primarily when organizational scale and rate of change justify the additional complexity [4]–[6].

Another implication concerns technical debt. Front-end debt is often described informally as stale components or messy stylesheets, but the more consequential problem is architectural debt: unclear ownership, missing contracts, and duplicated journey logic spread across teams. When measured in terms of defect escapes, onboarding delays, and release bottlenecks, debt becomes visible as a business constraint rather than a code-quality complaint. This is why modernization backlogs should prioritize high-interest debt items tied to customer journeys and operational pain, following the broader debt-management literature [7] – [10].

Finally, accessibility should be treated as a core architectural driver. Programs that retrofit accessibility after implementation often repeat the same fixes across repositories, whereas component-driven design allows one accessible pattern to scale across many journeys. In domains like retail checkout, this reduces both compliance risk and user friction. The same principle applies to performance: budgeted, component-level accountability is more sustainable than periodic optimization campaigns [11] – [19].

VII. LIMITATIONS

This paper has several limitations. The case study is anonymized and practice-based, which protects confidential delivery environments but also restricts replicability. The reported metrics are representative engineering ranges rather than measurements tied to a single publicly inspectable codebase. While this makes the paper appropriate for experience-driven architectural discussion, it limits causal claims.

A second limitation is that the architecture has been framed primarily around web delivery organizations using Angular- or Vue-centered stacks. The underlying principles are portable to other ecosystems, but specific implementation details such as build tooling, package versioning, and composition runtime choices may differ. Finally, some quality attributes that matter in retail, such as security posture, localization complexity, and SEO trade-offs under SSR or hybrid rendering, are acknowledged but not explored in depth here.

VIII. CONCLUSION AND FUTURE WORK

This paper presented a component-driven UI architecture for enterprise retail applications and argued that front-end modernization should be treated as an architectural program rather than a framework refresh. Drawing on literature and practitioner experience, the paper showed that effective modernization combines domain-aligned components, shared quality-enforcing primitives, observability, accessibility, and phased migration. The anonymized case study suggested that such an approach can improve performance, release cadence, and defect outcomes while making large applications easier to evolve.

Future work should extend this study in three directions. First, the proposed architecture should be evaluated through a longitudinal multi-team study using real production repositories and field telemetry. Second, the

relationship between component-driven design systems and micro-frontend anti-pattern prevention deserves more formal analysis. Third, additional work is needed on automated architecture conformance for front-end codebases, including detection of cross-domain leakage, accessibility regressions, and performance-budget violations at pull-request time. These directions would help turn the component-driven model from a practice-informed pattern into a more rigorously validated engineering framework.

REFERENCES:

[1] H. Jifeng, Z. Xiaoshan, and C. Ming, "Component-based software engineering," in Proc. 10th Int. Symp. Component-Based Software Engineering, 2005.
[2] I. Crnkovic, M. Chaudron, and S. Larsson, "Component-based development process and component lifecycle," J. Computing and Information Technology, vol. 13, no. 4, pp. 321–327, 2005.
[3] S. C. Lee, S. K. Cha, and J. Y. Jung, "A component-based methodology for Web application development," J. Systems and Software, vol. 71, no. 3, pp. 177–187, 2004.
[4] S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review," Information and Software Technology, vol. 136, 2021.
[5] F. Antunes, M. J. D. Lima, M. A. P. Araújo, D. Taibi, and M. Kalinowski, "Investigating benefits and limitations of migrating to a micro-frontends architecture," arXiv:2407.15829, 2024.
[6] N. Silva, E. Rodrigues, and T. Conte, "A catalog of micro frontends anti-patterns," arXiv:2411.19472, 2024.
[7] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt in software development: From metaphor to theory and practice," IEEE Software, vol. 29, no. 6, pp. 18–21, 2012.
[8] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in WICSA/ECSA, 2012.
[9] A. Martini and J. Bosch, "An empirically developed method to aid decisions on architectural technical debt refactoring," in Proc. ICSE, 2016.
[10] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in Proc. 2nd Workshop on Managing Technical Debt, 2011.
[11] World Wide Web Consortium (W3C), "Web Content Accessibility Guidelines (WCAG) 2.2," W3C Recommendation, Oct. 2023, updated Dec. 2024.
[12] T. Bi, T. Ladner, S. Trewin, and J. Mankoff, "Accessibility in software practice: A practitioner's perspective," ACM Trans. Accessible Computing, vol. 15, no. 1, 2022.
[13] Google Web Dev, "Core Web Vitals," web.dev, 2025.
[14] P. Irish and B. Kenny, "Largest Contentful Paint (LCP)," web.dev, updated 2025.
[15] A. Commarford and B. Kenny, "Cumulative Layout Shift (CLS)," web.dev, updated 2025.
[16] P. Lewis and J. Wagner, "Interaction to Next Paint (INP)," web.dev, updated 2025.
[17] J. Wagner and R. Viscomi, "How the Core Web Vitals metrics thresholds were defined," web.dev, 2020.
[18] E. Bocchi, L. De Cicco, and D. Rossi, "Measuring the Quality of Experience of Web users," in Proc. ACM MMSys, 2017.
[19] Q. Zhu, M. Nicosia, and M. Satyanarayanan, "Perceived performance of top retail webpages in the wild," in Proc. ACM SIGMETRICS/Performance, 2017.