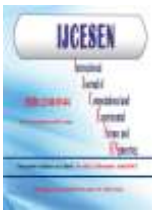
International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 11-No.4 (2025) pp. 9810-9818
http://www.ijcesen.com



Copyright © IJCESEN

ISSN: 2149-9144

Research Article

Architecting Scalable Front-End Systems for Enterprise E-Commerce Promotions: Micro-Frontend Patterns, React-Based Modernization, and Event-Driven UI Architectural Practices

Preejith Ponneth*

Independent E-Commerce Platform Architect, USA
* **Corresponding Author Email:** ponnethpreejith@gmail.com - **ORCID:** 0000-0002-9947-8880

Article Info:

DOI: 10.22399/ijcesen.4536
Received : 28 October 2025
Revised : 05 December 2025
Accepted : 11 December 2025

Keywords

Micro-Frontend Architecture,
React Component Systems,
Event-Driven UI,
Front-End Scalability,
Module Federation,
Legacy UI Modernization,
Enterprise Platform Engineering

Abstract:

Modern e-commerce platforms demand sophisticated front-end architectures capable of delivering real-time promotional experiences across diverse customer touchpoints. Legacy monolithic front-end systems create deployment bottlenecks, team coordination overhead, and limited scalability during high-traffic promotional events. Customers expect instant price updates, personalized offer displays, and consistent promotional interfaces whether browsing web applications, mobile platforms, or in-store kiosks. Traditional single-page application architectures embed promotional UI logic within tightly coupled component hierarchies, constraining innovation velocity and independent feature deployment. Micro-frontend architectures address these limitations by decomposing promotional user interfaces into independently deployable modules aligned with business capabilities. Module federation enables runtime composition of promotional components developed by autonomous teams using diverse technology stacks. React-based component architectures provide declarative patterns for complex promotional state management and dynamic UI rendering. Event-driven front-end patterns facilitate real-time promotional updates through WebSocket connections and server-sent events, maintaining UI consistency across distributed system boundaries. Legacy UI modernization strategies employing strangler patterns enable gradual migration from monolithic front-end applications to distributed micro-frontend ecosystems. Performance optimization techniques including code splitting, lazy loading, and intelligent bundling strategies ensure promotional interfaces maintain responsiveness under heavy concurrent usage. Future advancement in front-end promotional systems will progressively integrate edge computing for personalized rendering and machine learning models for predictive UI optimization based on user behavioral patterns.

1. Introduction

E-commerce promotional interfaces serve as primary touchpoints shaping customer purchase decisions across digital channels. Modern shoppers expect promotional content to render instantly, update dynamically, and maintain visual consistency across web browsers, mobile applications, and progressive web experiences. Meeting these expectations requires front-end architectures capable of managing complex promotional state, rendering personalized offers efficiently, and coordinating real-time updates across distributed UI components. Enterprise promotional platforms must handle rapid campaign activations, dynamic pricing displays, and eligibility-based content rendering while maintaining interface responsiveness under substantial concurrent user loads.

Legacy front-end promotional architectures encounter fundamental challenges in independent deployment and selective scalability. Traditional monolithic single-page applications embed promotional UI logic within tightly coupled component trees, creating dependencies that prevent isolated feature updates and targeted performance optimization. The architectural rigidity of monolithic front-end platforms constrains promotional innovation, as modifications to offer display components or eligibility indicators necessitate full application rebuilds accompanied by associated deployment risks [1]. Micro-frontend

architectures address these limitations by decomposing promotional interfaces into autonomous modules that integrate through well-defined contracts. Module boundaries align with business capabilities such as offer display, pricing presentation, and campaign banners, enabling teams to evolve promotional features independently without cross-module coordination overhead [1].

The transition to micro-frontend promotional systems introduces architectural complexity requiring careful design considerations. Module granularity decisions determine operational efficiency and system maintainability. Excessively fine-grained modules increase network requests and complicate shared state management across component boundaries. Conversely, coarse-grained modules retain monolithic characteristics that limit independent scalability and deployment flexibility [1]. Front-end architects must balance module autonomy against coordination requirements, particularly when promotional displays span multiple business contexts such as inventory availability indicators, customer loyalty status badges, and payment method eligibility notices.

React has emerged as the dominant framework for enterprise promotional interface development, providing component-based architecture patterns that align naturally with micro-frontend decomposition strategies [2]. Component composition models enable promotional UI elements to be developed, tested, and deployed independently while maintaining integration coherence. Virtual DOM reconciliation algorithms optimize rendering performance for dynamic promotional content that updates frequently during active campaigns. Hook-based state management patterns provide declarative approaches to complex promotional logic, including eligibility calculations, discount computations, and personalized offer selections [2]. The React ecosystem offers mature tooling for code splitting, lazy loading, and bundle optimization essential for promotional interface performance.

Event-driven front-end architectures provide communication patterns for real-time promotional updates across distributed UI components. WebSocket connections enable bidirectional communication channels that push promotional changes to client interfaces without polling overhead. Server-sent events deliver unidirectional promotional updates for scenarios requiring lower implementation complexity. Custom event systems facilitate loose coupling between promotional modules, allowing independent evolution while maintaining coordinated user experiences [3]. The combination of micro-frontend architecture principles, React component patterns, and event-driven communication establishes technical foundations for promotional platforms that balance performance requirements with operational flexibility and innovation velocity.

2. Architectural Transition from Monolithic to Micro-Frontend Promotional Systems

Legacy promotional interfaces embed offer rendering, pricing display, and eligibility indication within monolithic single-page applications. This centralization creates deployment bottlenecks that constrain promotional agility and interface responsiveness. Monolithic designs couple promotional components with core e-commerce interface functions through shared state management and tightly integrated routing configurations. State structures in legacy systems often intermingle promotional data with shopping cart information, user session details, and navigation context. This state entanglement complicates selective updates to promotional capabilities without risking unintended impacts on adjacent interface functions.

Migration from monolithic promotional interfaces requires systematic identification of module boundaries based on rendering dependencies and business capabilities. Component-centric analysis examines UI hierarchies to identify cohesive clusters of promotional elements that support independent module operations [4]. Offer display components, pricing badge elements, and campaign banner modules represent distinct UI domains suitable for module encapsulation. Analyzing component communication patterns reveals which promotional elements exhibit tight coupling versus those that operate independently. Modules emerge where component cohesion is high internally but coupling across domains remains minimal [4]. This identification process prevents arbitrary module decomposition that would create excessive cross-module communication and distributed state complexity.

Micro-frontend promotional architectures decompose monolithic responsibilities into specialized modules addressing distinct functional domains. Offer display modules render promotional content using dedicated state containers maintaining campaign definitions and personalization rules. Separate pricing presentation modules maintain discount display logic, tiered pricing indicators, and promotional countdown timers in isolated state structures. Module boundaries align with business capabilities rather than technical layer separation. Campaign banner modules handle promotional visual presentations including hero images and animated offer

announcements. Customer loyalty modules maintain status displays independent from discount indication logic [4].

Micro-frontend design for promotional systems must address practical operational concerns beyond theoretical architectural purity. Module granularity decisions balance autonomy benefits against increased system complexity and operational overhead [5]. Promotional offer display represents suitable micro-frontend granularity due to distinct rendering characteristics and independent evolution requirements. Overly fine-grained decomposition leads to module proliferation that complicates deployment orchestration and distributed debugging. State management across module boundaries introduces consistency challenges when promotional displays span multiple discount types and eligibility indicators. Eventually consistent UI patterns accommodate many promotional scenarios where slight delays in inventory updates or budget indicators are acceptable [5].

Module federation establishes runtime integration between promotional micro-frontends, enabling dynamic composition without build-time dependencies. Webpack Module Federation allows promotional modules to expose components for consumption by host applications while maintaining independent deployment lifecycles [6]. Remote entry configurations specify module locations and expose interfaces, enabling runtime resolution of promotional components. Shared dependency configurations prevent duplicate library loading while maintaining version compatibility across independently deployed modules. Federation patterns support both eager and lazy loading strategies, optimizing initial page load performance while ensuring promotional components are available when needed [6]. Container applications orchestrate promotional module composition, providing layout frameworks and routing integration while delegating promotional rendering to federated modules.

3. React-Based Component Architecture for Promotional Systems

React component patterns provide architectural foundations for scalable promotional interface development. Component composition models enable promotional UI elements to be structured as hierarchical trees with well-defined data flow patterns. Unidirectional data flow ensures promotional state changes propagate predictably through component hierarchies, simplifying debugging and enabling efficient rendering optimization [2]. Presentational components handle promotional visual rendering without business logic coupling, while container components manage promotional state and coordinate with backend services. This separation enables promotional visual designs to evolve independently from data integration logic.

Hook-based state management patterns address complex promotional scenarios requiring coordinated state across multiple component subtrees. The useState hook manages local promotional state for individual component instances, suitable for isolated UI interactions such as offer expansion toggles and promotional detail modals. The useReducer hook provides structured state management for complex promotional logic involving multiple interdependent state transitions [7]. Promotional eligibility determinations, discount calculations, and personalized offer selections benefit from reducer patterns that centralize state transition logic. Custom hooks encapsulate reusable promotional logic, enabling consistent behavior across promotional components while maintaining implementation flexibility.

Context-based state distribution enables promotional data sharing across component hierarchies without prop drilling overhead. React Context provides promotional state containers that inject data at arbitrary component tree depths [7]. Promotional campaign context supplies active offer definitions to deeply nested display components without intermediate component awareness. Customer context provides personalization data enabling promotional components to render targeted offers based on user attributes. Context composition patterns layer multiple promotional contexts, enabling selective subscription to relevant state domains while avoiding unnecessary re-renders from unrelated state changes.

Performance optimization for promotional React components requires careful attention to rendering efficiency and memory management. Memoization patterns prevent unnecessary promotional component re-renders when parent components update with unchanged promotional props [8]. React.memo wraps promotional presentational components, implementing shallow prop comparison to skip renders when promotional data remains stable. useMemo caches expensive promotional calculations such as discount computations and eligibility determinations, recomputing only when dependency values change. useCallback stabilizes promotional event handler references, preventing child component re-renders triggered by handler recreation during parent renders [8].

Code splitting strategies decompose promotional interface bundles into independently loadable chunks. React.lazy enables dynamic import of

promotional components, deferring bundle loading until components enter the render path [8]. Suspense boundaries provide loading state management during promotional component resolution, displaying placeholder content while promotional modules load. Route-based splitting separates promotional page bundles from core application code, reducing initial load sizes. Component-based splitting isolates heavyweight promotional features such as interactive configurators and rich media galleries into separate chunks loaded on demand. Prefetching strategies anticipate promotional component needs based on user navigation patterns, loading promotional bundles before explicit user requests.

4. Event-Driven Front-End Architecture for Real-Time Promotions

Real-time promotional experiences require front-end architectures capable of receiving and processing dynamic updates without full page refreshes. Event-driven patterns decouple promotional UI components from update sources, enabling flexible integration with diverse backend notification mechanisms. WebSocket connections establish persistent bidirectional channels between promotional interfaces and backend systems, enabling immediate push delivery of promotional changes [3]. Price updates, inventory availability changes, and campaign activations propagate instantly to connected clients, maintaining promotional display accuracy without polling overhead.

Server-sent events provide lightweight unidirectional streaming for promotional update scenarios. SSE connections deliver promotional notifications through standard HTTP, simplifying infrastructure requirements compared to WebSocket implementations [9]. Promotional countdown timers, flash sale announcements, and limited inventory warnings stream efficiently through SSE channels. Automatic reconnection handling ensures promotional update continuity during transient network interruptions. Event ID tracking enables clients to resume promotional streams from last received positions, preventing missed updates during reconnection windows [9].

Custom event systems facilitate communication between promotional micro-frontend modules without direct coupling. Browser CustomEvent APIs enable promotional modules to publish domain events consumed by interested modules [3]. Offer selection events notify pricing modules to update discount displays. Cart addition events trigger promotional recommendation modules to refresh cross-sell suggestions. Event bus abstractions provide centralized publish-subscribe infrastructure, managing event routing and subscriber lifecycle. Typed event contracts ensure promotional modules communicate through well-defined interfaces, enabling independent evolution while maintaining integration correctness.

State synchronization patterns maintain consistency between promotional UI state and backend promotional systems. Optimistic updates render promotional changes immediately upon user interaction, improving perceived responsiveness while backend confirmations propagate [10]. Rollback mechanisms revert optimistic promotional state when backend validation fails, providing accurate feedback without blocking initial interactions. Event sourcing patterns reconstruct promotional UI state from event streams, enabling consistent state recovery after connection interruptions. Conflict resolution strategies handle concurrent promotional updates, applying deterministic merge rules when simultaneous changes affect overlapping promotional state [10].

Real-time promotional architectures require robust error handling and graceful degradation strategies. Connection monitoring detects promotional update channel failures, triggering reconnection attempts with exponential backoff. Fallback polling mechanisms activate when streaming connections remain unavailable, ensuring promotional updates continue through alternative channels. Stale data indicators inform users when promotional information may be outdated due to connection issues. Circuit breaker patterns prevent cascading failures when promotional update services experience sustained unavailability, isolating failures while maintaining core interface functionality [9].

5. Legacy UI Modernization and Migration Strategies

Migration from legacy promotional interfaces requires systematic approaches that maintain business continuity while enabling architectural transformation. Legacy promotional front-ends accumulate technical debt over extended operational lifespans, embedding business logic within framework-specific patterns that resist incremental modification. jQuery-based promotional widgets, AngularJS promotional modules, and custom framework implementations represent common legacy patterns requiring modernization. Research on front-end migration reveals gradual replacement strategies as essential for managing transformation risk while maintaining promotional capability continuity [11].

Strangler pattern implementations gradually replace legacy promotional components with modern micro-frontend modules through incremental capability migration. The pattern establishes routing mechanisms that direct users selectively between legacy interfaces and newly developed React components [11]. Migration strategies emphasize gradual replacement rather than complete interface rewrites that introduce excessive business risk. Component identification during migration analyzes rendering boundaries, state dependencies, and business capabilities to determine optimal granularity for modernization. Initial migration phases target promotional components with minimal dependencies on legacy framework infrastructure, enabling isolated modernization while maintaining integrated promotional workflows.

Wrapper components enable legacy promotional widgets to coexist with modern React modules during transition periods. React wrapper components encapsulate legacy promotional elements, providing lifecycle management and event bridging [12]. Legacy promotional widgets render within React component trees while modernization progresses on adjacent components. Event translation layers convert legacy promotional events to React-compatible patterns, enabling communication between legacy and modern components. State synchronization mechanisms maintain consistency between legacy promotional state and React state containers, preventing divergence during extended coexistence periods [12].

Build system modernization establishes tooling foundations for micro-frontend promotional architectures. Webpack configurations enable code splitting, tree shaking, and bundle optimization essential for promotional interface performance [6]. Module federation plugins configure promotional component exposure and consumption patterns for distributed module composition. Babel configurations ensure JavaScript compatibility across target browser environments while enabling modern language features in promotional component development. Asset pipeline configurations handle promotional media optimization, including image compression, responsive image generation, and lazy loading instrumentation.

Continuous integration pipelines automate promotional module validation and deployment. Automated testing validates promotional component rendering, interaction handling, and state management correctness. Visual regression testing detects unintended promotional display changes through screenshot comparison [13]. Performance budgets enforce promotional bundle size limits, preventing gradual performance degradation through accumulated dependencies. Deployment automation enables independent promotional module releases, supporting rapid iteration on campaign interfaces without coordinating full platform deployments. Feature flag integration enables progressive promotional feature rollouts, limiting exposure while validating new promotional components under production conditions [13].

6. Performance Engineering for Promotional Front-End Systems

Achieving optimal promotional interface performance requires engineering practices that balance rendering efficiency with feature richness. Promotional interfaces must render complex offer displays, animated banners, and dynamic pricing information while maintaining responsiveness under heavy concurrent usage. Performance optimization strategies differ fundamentally between initial page load optimization and runtime interaction performance. Initial load optimization focuses on bundle size reduction and critical rendering path optimization. Runtime optimization addresses rendering efficiency, memory management, and interaction responsiveness [14].

Bundle optimization strategies reduce promotional interface payload sizes through multiple complementary techniques. Tree shaking eliminates unused promotional code from production bundles, removing development utilities and unused component variants [6]. Code splitting separates promotional features into independently loadable chunks, enabling progressive loading based on user navigation. Dynamic imports defer promotional component loading until explicit user interaction or viewport visibility triggers loading. Compression algorithms reduce promotional bundle transfer sizes, with Brotli compression providing superior ratios for text-based JavaScript bundles [14].

Critical rendering path optimization ensures promotional content displays rapidly during initial page loads. Server-side rendering pre-generates promotional HTML on server infrastructure, enabling immediate content display before JavaScript bundle execution [15]. Hydration processes attach React event handlers to server-rendered promotional markup, enabling interactivity without full client-side re-rendering. Streaming SSR delivers promotional content progressively as server rendering completes, reducing time to first meaningful paint. Static generation pre-renders promotional pages at build

time for content that changes infrequently, eliminating server rendering latency entirely [15].

Runtime rendering optimization maintains promotional interface responsiveness during user interactions. Virtual list implementations render only visible promotional items, enabling efficient handling of large offer catalogs without rendering all items simultaneously [14]. Intersection Observer APIs detect promotional component visibility, triggering lazy loading and animation activation only when elements enter viewport. RequestAnimationFrame scheduling aligns promotional animations with browser paint cycles, ensuring smooth visual transitions. Web Worker offloading moves expensive promotional calculations to background threads, preventing main thread blocking during complex eligibility determinations [14].

Memory management practices prevent promotional interface degradation during extended user sessions. Component cleanup ensures promotional event listeners and subscription references are properly released during unmounting. Weak reference patterns enable promotional caches to release memory under pressure without explicit invalidation. Memory profiling identifies promotional components retaining excessive references, enabling targeted optimization. Virtualization strategies limit simultaneous promotional component instances, bounding memory consumption regardless of catalog size [15].

7. Future Directions in Intelligent Front-End Promotional Engineering

Advanced promotional front-end systems will progressively incorporate intelligent capabilities that optimize user experiences through predictive and adaptive techniques. Edge computing architectures enable promotional personalization at network edge locations, reducing latency for geographically distributed users [16]. Edge-rendered promotional content incorporates user context without round-trips to centralized servers, enabling instant personalized offer displays. Edge functions execute promotional eligibility logic at CDN nodes, combining cached promotional assets with dynamic personalization. Distributed edge state enables promotional interfaces to access user preferences and behavioral data from nearest edge locations [16].

Machine learning integration enables promotional interfaces to optimize rendering strategies based on predicted user behavior. Predictive prefetching anticipates promotional content needs based on navigation pattern analysis, loading offers before explicit user requests [17]. Behavioral models identify user segments with distinct promotional interaction patterns, enabling interface optimization for different engagement styles. Attention prediction models estimate promotional element visibility likelihood, prioritizing rendering resources for high-attention areas. Reinforcement learning approaches optimize promotional layout decisions through continuous experimentation, balancing exploration of novel arrangements against exploitation of proven effective patterns [17].

Personalization engines will deliver increasingly individualized promotional experiences through real-time adaptation. Client-side inference enables promotional personalization models to execute directly in browser environments, eliminating server round-trips for personalization decisions [16]. WebAssembly enables high-performance model execution for complex promotional recommendation algorithms. Federated learning approaches enable promotional models to improve from user interactions while maintaining privacy through on-device training. Contextual adaptation incorporates environmental signals including time, location, and device characteristics into promotional rendering decisions [17].

Component-level analytics will provide granular visibility into promotional interface effectiveness. Interaction tracking captures promotional component engagement patterns, revealing which offer presentations drive conversions. Attention metrics measure promotional element visibility duration and scroll depth, indicating content engagement quality. A/B testing frameworks enable promotional component experimentation, comparing variant effectiveness through controlled exposure [17]. Automated optimization systems will progressively adjust promotional interfaces based on measured performance, reducing manual optimization effort while improving promotional effectiveness.

Table 1. *Module Decomposition in Promotional Front-End Architectures: Functional Components and Responsibilities [4, 5, 6].*

Component	Primary Function	State Managed	Deployment Trigger
Offer Display Module	Render promotional content	Campaign rules, personalization	Campaign launches

Pricing Presentation	Display discounts and tiers	Discount structures, timers	Pricing updates
Campaign Banners	Handle visual presentations	Banner assets, animations	Marketing requests
Loyalty Status Module	Maintain customer indicators	Loyalty data, tier badges	Program changes
Shell Application	Orchestrate module composition	Routing, layout context	Platform releases

Table 2. *React Component Patterns for Promotional Systems: Architecture and Optimization Techniques [2, 7, 8].*

Pattern	Implementation	Primary Benefit	Application Context
Custom Hooks	Encapsulated logic functions	Reusable promotional behavior	Eligibility checks, calculations
Context Composition	Layered context providers	Selective state subscription	Campaign data, personalization
Component Memoization	React.memo, useMemo	Prevents unnecessary re-renders	Stable promotional displays
Code Splitting	React.lazy, dynamic imports	Reduced initial bundle size	Heavy promotional features
Suspense Boundaries	Fallback UI during loading	Graceful loading states	Async promotional content

Table 3. *Event-Driven Communication Patterns for Real-Time Promotional Interfaces [3, 9, 10].*

Mechanism	Communication Pattern	Primary Benefit	Use Case
WebSocket	Bidirectional streaming	Instant push delivery	Price updates, inventory
Server-Sent Events	Unidirectional streaming	Simplified infrastructure	Timers, announcements
Custom Events	Module-to-module	Loose coupling	Cross-module coordination
Optimistic Updates	Immediate rendering	Perceived responsiveness	User interactions
Event Sourcing	State reconstruction	Connection recovery	State synchronization

Table 4. *Legacy Modernization Strategies: Migration Patterns and Enabling Capabilities [6, 11, 12, 13].*

Strategy	Implementation Approach	Key Benefit	Requirement
Strangler Pattern	Gradual component replacement	Minimizes business risk	Routing infrastructure
Wrapper Components	Legacy encapsulation	Coexistence enablement	Lifecycle bridging
Module Federation	Runtime composition	Independent deployment	Webpack 5+
Visual Regression	Screenshot comparison	Prevents display regressions	Testing infrastructure
Feature Flags	Progressive rollout	Controlled exposure	Flag management system

8. Conclusions

Architectural evolution from monolithic to micro-frontend promotional systems represents a transformative advancement in enterprise e-commerce interface engineering. Decomposition of promotional interfaces into specialized modules enables organizations to deploy and scale UI capabilities precisely where user demands concentrate. Independent module deployment maintains development velocity during promotional campaign launches without coordinating full platform releases. React-based component architectures provide declarative patterns for complex promotional state management, enabling efficient rendering of dynamic offer displays and personalized content. Hook-based patterns and context composition address sophisticated promotional scenarios while maintaining code maintainability and testing accessibility.

Event-driven front-end patterns deliver real-time promotional experiences essential for modern e-commerce engagement. WebSocket and server-sent event integrations enable instant propagation of pricing updates, inventory changes, and campaign activations to connected interfaces. Custom event systems facilitate loose coupling between promotional modules, enabling independent evolution while maintaining coordinated user experiences. Optimistic update patterns and event

sourcing provide responsive interactions with robust consistency guarantees.

Legacy modernization pathways using strangler patterns offer pragmatic transitions from monolithic front-end applications to distributed micro-frontend ecosystems. Incremental migration reduces business risk while enabling continuous capability enhancement throughout transformation periods. Wrapper components enable legacy promotional widgets to coexist with modern React modules during extended transition phases. Module federation configurations establish runtime composition patterns that support independent promotional module deployment and dynamic integration.

Performance engineering practices ensure promotional interfaces maintain responsiveness under demanding usage conditions. Bundle optimization through tree shaking, code splitting, and compression reduces payload sizes while preserving feature richness. Server-side rendering and streaming techniques accelerate initial promotional content display. Runtime optimization through virtual lists, intersection observers, and web worker offloading maintains interaction responsiveness during complex promotional scenarios.

Integration of intelligent capabilities marks the significant frontier in promotional front-end evolution beyond static rendering approaches. Edge computing enables personalized promotional experiences with minimal latency regardless of user geography. Machine learning models will progressively optimize promotional interfaces through predictive prefetching, attention modeling, and automated layout optimization. Organizations implementing advanced promotional front-end architectures position themselves competitively through superior user experiences and optimized conversion outcomes. The convergence of micro-frontend engineering with intelligent personalization capabilities creates unprecedented opportunities for promotional innovation in digital commerce environments.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

[1] Pooyan Jamshidi et al., "Microservices: The Journey So Far and Challenges Ahead," IEEE Software, 2018. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8354433

[2] Olaf Zimmermann, "Microservices Tenets: Agile Approach to Service Development and Deployment,". [Online]. Available: https://www.ost.ch/fileadmin/dateiliste/3_forschung_dienstleistung/institute/ifs/cloud-application-lab/msa-pospaperzio4summersoc2016v15nc.pdf

[3] Yamina Romani et al., "Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification," [Online]. Available: https://hal.science/hal-04101242/document

[4] Mike Amundsen, "Microservices in Practice, Part 1 Reality Check and Service Design," IEEE Software, 2017. [Online]. Available: https://d1wqtxts1xzle7.cloudfront.net/86218092/MS.2017.2420220520-1-ajzyiq-libre.pdf?1653072988=&response-content-disposition=inline%3B+filename%3DMicroservices_in_Practice_Part_1_Reality.pdf&Expires=1763530184&Signature=fCST0s1IM7R2IRDbKGyZHjLsWMQhLJg~NLMh0W4Sw84g4zepxczx9QS-dtSpTTSqElWXOdvHU3zNP5kErQGtw-EU6xILgdxMowjIFtwZEt9R~~YODacXEMrE1IyDQXQj2Ah81Ea6sQ4JluPZb02lBjReukcruZJitGNaKFe2hp96SCIjO9gOsPDznmAMt9fTQ57~1RKEglhIsnMkIY5EINf23cRAEVezrkY3kQiGqlHsgiK0x7nFkV0JAaRA3SAqqQyDj8lKI9HGG9~7OL4q7teE~sUSRXJeOSgtKO7bDGAZpGL27S6x1y0cAsX-yOcve7jTZhoffGxQhMwvTmMQzg__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

[5] Mario Villamizar et al., "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Applications in the Cloud," IEEE, 2015. [Online]. Available: https://www.researchgate.net/profile/Mario-Villamizar/publication/304317852_Evaluating_the_monolithic_and_the_microservice_architecture_pattern_to_deploy_web_applications_in_the_cloud/links/5b3ad04ca6fdcc8506ea541b/Evaluating-the-monolithic-and-the-microservice-architecture-pattern-to-deploy-web-applications-in-the-cloud.pdf

[6] Claus Pahl and Pooyan Jamshidi, "Microservices: A Systematic Mapping Study," In Proceedings of the

6th International Conference on Cloud Computing and Services Science, 2016. [Online]. Available: https://www.scitepress.org/PublishedPapers/2016/57855/57855.pdf

[7] Paolo Di Francesco et al., "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," IEEE International Conference on Software Architecture, 2017. [Online]. Available: https://research.vu.nl/ws/portalfiles/portal/309050963/ICSA_2017_Research_on_Architecting_Microservices_Trends_Focus_and_Potential_for_Industrial_Adoption.pdf

[8] Nane Kratzke and Peter-Christian Quint, "Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study," Journal of Systems and Software, 2017. [Online]. Available: https://www.researchgate.net/profile/Nane-Kratzke/publication/312045183

[9] Emrul Hasan et al., "Review-based Recommender Systems: A Survey of Approaches, Challenges and Future Perspectives," ACM Computing Surveys, 2025. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3742421

[10] Shuai Li et al., "Collaborative Filtering Bandits," arXiv, 2016. [Online]. Available: https://arxiv.org/pdf/1502.03473