International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I2P106
Eureka Vision Publication | Volume 7, Issue 2, 38-45, 2026

Original Article

Shared Without Coupling: A Cross-Framework State Contract Model for Multi-Product E-Commerce Funnels in Angular and React

Althaf Khan Pattan
Independent Researcher Exton, Pennsylvania, USA.

Received On: 17/02/2026 ***Revised On: 23/03/2026*** ***Accepted On: 31/03/2026*** ***Published On: 08/04/2026***

Abstract - Managing application state across independently deployed micro-frontends built on different frameworks is one of the harder unsolved problems in large-scale front-end engineering. Most production platforms serving multiple product lines end up with Angular teams using NgRx, React teams using Redux or Zustand, and no clear agreement on how those stores should share data without quietly depending on each other's internals. The result is typically a collection of ad hoc solutions: window globals, duplicated API calls, and undocumented custom event conventions that hold together until something changes and then break in ways that are hard to trace.This paper proposes the Cross-Framework State Contract Model (CFSCM), a structured approach to defining what state gets shared across framework boundaries in a micro-frontend e-commerce platform, how that state flows, and what guarantees each fragment can rely on. The model separates state belonging to a single team's fragment from state that must be visible platform-wide, expressing both as explicit typed contracts rather than informal conventions. A neutral event bus sits between frameworks and enforces those contracts at runtime, so Angular and React fragments never reach into each other's stores.The model is evaluated against a simulated four-fragment platform running Angular NgRx for account flows and React Redux or Zustand for catalog, configuration, and checkout flows. Simulated results show state synchronization latency under 12ms at the 95th percentile, inter-fragment coupling dropping by 62% compared to a window global baseline, and developer onboarding time reducing by an estimated 43% when contracts are documented and enforced. All figures come from simulation and require production validation.

Keywords - Cross-Framework State Management, Micro-Frontends, Angular Ngrx, React Redux, State Contracts, E-Commerce, Event Bus, Zustand, Multi-Product Platform.

1. Introduction

Large e-commerce platforms rarely sell a single product. A typical platform might offer broadband subscriptions, mobile plans, streaming services, and device bundles, each with its own pricing logic, eligibility rules, checkout flow, and regulatory requirements. When these product lines are built by separate engineering teams, the front-end architecture tends to follow the same organizational split. Team A owns the Angular application handling account management. Team B owns the React application handling product discovery and configuration. Team C owns another React application handling checkout and payment. Each team picks the state management library that fits their stack and for a while that division works fine.

The problem surfaces when these independently built fragments need to share state. A user who selects a mobile plan in Team B's fragment arrives at Team C's checkout carrying device preferences, plan tier selection, and promotional eligibility flags. Those values need to follow the user across framework boundaries accurately, without either team needing to know how the other stores them internally. In practice, this cross-fragment state handoff is where things break down. Teams reach for whatever is expedient: a value on the window object, a URL parameter, a duplicated API call, or a custom event with an undocumented payload shape. Each of these works until something changes, the payload shape evolves, an event name gets renamed, or a new engineer joins who does not know the convention exists.

What is missing is not a better state management library. NgRx, Redux, and Zustand are all mature and well suited to their respective frameworks [1][2][3]. What is missing is a governed layer above the stores: a shared agreement about what state crosses fragment boundaries, in what format, through what channel, and with what lifecycle guarantees. That coordination layer is what this paper formalizes as a state contract a typed, versioned agreement that governs what crosses framework boundaries, in what shape, and with what lifecycle guarantees.

The Cross-Framework State Contract Model (CFSCM) has three main parts. First, a state boundary taxonomy classifying all platform state into one of three categories: fragment-private state that never leaves the owning fragment, platform-shared state that all fragments can read, and handoff state that passes from one fragment to the next at navigation boundaries. Second, a contract specification format making the shape, ownership, version, and lifecycle of shared state explicit and machine-readable. Third, a runtime event bus enforcing contract boundaries between

Angular NgRx and React Redux or Zustand fragments without either framework reaching into the other's store.

The main contributions of this paper are: (1) a three-tier state boundary taxonomy tailored to multi-product e-commerce funnels; (2) the CFSCM contract specification covering state shape, ownership, versioning, and lifecycle; (3) a framework-agnostic event bus connecting NgRx Effects with Redux middleware and Zustand subscribers without tight coupling; (4) concrete implementation patterns for the four most common shared state scenarios; and (5) simulated evaluation results measuring synchronization latency, inter-fragment coupling reduction, and estimated developer onboarding impact.

2. Background and Related Work

2.1. State Management in SPA Architectures

Redux, introduced alongside React in 2015, established the dominant client-side state pattern for single-page applications: a single immutable store, unidirectional data flow, and pure reducer functions [1]. NgRx brought the same Redux philosophy to Angular but built it on RxJS, making state changes observable streams that components and effects can subscribe to [2]. Zustand takes a lighter approach, removing the boilerplate of actions and reducers in favor of a simpler store factory while keeping centralized state and predictable updates [3]. Each library solves state management well within its own framework and codebase. The difficulty begins when a platform spans multiple codebases built on different frameworks, because cross-framework state sharing was never a design goal of any of them.

2.2. State Management in Micro-Frontend Systems

Micro-frontend architecture brings service decomposition thinking to the presentation layer [4]. Geers [5] provides a comprehensive treatment of micro-frontend patterns and recommends minimizing shared state in favor of URL-encoded state and browser custom events. Jackson [6] notes that frequent state sharing between micro-frontends often signals that decomposition boundaries were drawn incorrectly. The micro-frontends.org specification [11] goes further, advising teams to avoid shared state or global variables entirely. These recommendations work well when fragments are truly independent. They become impractical in multi-product e-commerce, where fragments represent sequential stages of a connected purchasing funnel and state must move between them at each stage boundary.

2.3. Cross-Framework Interoperability

Web Components offer a browser-native way to wrap framework-specific components in a framework-agnostic shell [7] and have appeared as an integration mechanism in some micro-frontend platforms. They address component-level interoperability, not state-level interoperability. The Custom Events API provides a lightweight publish-subscribe mechanism that any framework can use [8] but enforces no schema on event payloads, leaving teams to detect structural drift manually. Nx supports shared libraries across Angular and React workspaces [9] and improves build-time type safety through shared TypeScript definitions [12], but does not address runtime state synchronization or contract enforcement.

2.4. Research Gap

Existing literature addresses state management within individual frameworks well [13][14] and discusses micro-frontend communication patterns at a high level. The specific problem of structured, contractual state sharing across different framework stores in a multi-product purchasing funnel, particularly in SPA architectures where server-side log analytics are unavailable, has not been addressed with enough specificity to guide practitioners. The CFSCM targets that gap by proposing a model sitting above individual stores that governs how state crosses between them in a typed, versioned, and lifecycle-managed way.

3. Problem Formulation

3.1. The Multi-Product State Problem

Consider a platform with four independently deployed fragments: an Account fragment built in Angular with NgRx, a Product Catalog fragment built in React with Redux, a Configuration fragment built in React with Zustand, and a Checkout fragment built in React with Redux. A typical purchase journey moves a user through all four in sequence. The user authenticates in the Account fragment, selects a product in the Catalog fragment, configures options in the Configuration fragment, and completes payment in the Checkout fragment. At each boundary, state must transfer between fragments, crossing at least one framework divide. Without a defined mechanism for that transfer, teams build their own solutions, and those informal mechanisms accumulate technical debt that compounds over time [15].

3.2. Categories of Shared State

Not all state needs to cross fragment boundaries, and getting that classification right matters before anything else. Fragment-private state lives entirely within a single fragment accordion expansion state, form validation messages, loading indicators. The owning fragment is the only one that ever reads or writes it, and that should stay true regardless of what the rest of the platform does. Platform-shared state is different: it is written by one fragment but read by many. Authentication status, session tokens, user identity, feature flags, active promotional campaigns — these need to reach any fragment regardless of when it mounts, so they require a persistent and reliable distribution mechanism. Handoff state is the third category and the one that causes the most production problems in practice. It moves from one fragment to the next at a specific funnel boundary, has one producer and one consumer, and should disappear immediately after delivery. The reason the cleanup matters is practical: a user who reverses and re-selects should arrive at the next stage with fresh state, not whatever was left over from their previous pass through the funnel.

3.3. Failure Modes without Contracts

Three failure patterns appear repeatedly in platforms managing cross-fragment state without explicit contracts. Silent payload drift happens when a producing fragment

changes the structure of a shared state object and consuming fragments receive unexpected shapes with no warning at build time or runtime. This failure does not throw an error immediately; it produces subtly wrong behavior downstream, often in a fragment that had nothing to do with the change. Write ownership conflicts occur when multiple fragments update what they each believe is shared state, producing conflicting values with no defined resolution. Stale handoff accumulation happens when handoff state is not cleaned up after delivery, so values from a previous product selection appear in a later traversal of the same funnel and incorrectly pre-populate fields.

3.4. Design Criteria

Four requirements shape a workable solution. First, every piece of shared state needs a declared owner, a declared scope, and a defined set of consumers — nothing should cross fragment boundaries without someone being accountable for it. Second, structural changes to shared state should fail at build time, not silently at runtime when a consuming fragment receives a shape it no longer understands. Third, no fragment should need to know which framework produced a value or how it is stored on the other side; the transport layer is the only part of the system that should care about that. Finally, shared state needs a clearly defined creation point, a validity window, and a cleanup trigger that fires reliably in both forward and backward navigation the stale handoff problem from Section III-C is almost always a lifecycle failure, not a data problem.

4. The Cross-Framework State Contract Model

4.1. Architecture Overview

The CFSCM operates as a coordination layer between individual framework stores and the broader platform. Diagram 1 shows the full architecture. Each fragment keeps its own internal store unchanged, using whichever library fits its framework. The CFSCM does not replace or wrap those stores. Instead, it defines what state leaves each store, the format it takes in transit, and how it arrives at consuming stores. The four components are: the Contract Registry, a shared TypeScript file declaring all platform-shared and handoff contracts with their shape, version, owner, and permitted consumers; the Event Bus, a thin runtime module accepting state publications from any fragment, validating them, and delivering validated payloads to subscribers; the NgRx Bridge, an NgRx Effect publishing relevant store changes to the event bus; and the Redux Bridge and Zustand Bridge, which perform the same role for React fragments through middleware and store subscribers respectively.

Table 1. CFSCM Architecture - Framework Stores, Bridges, and Event Bus

Angular Fragment	CFSCM Layer	React Fragments
NgRx Store Actions - Reducers - Effects Selectors	Contract Registry Typed state contracts Version + ownership	Redux Store / Zustand Actions - Reducers - Subscribers Selectors / hooks
▼	▼	▼
NgRx Bridge Effect listens for state Calls bus.publish()	Event Bus Validates against contract Dispatches CustomEvent	Redux / Zustand Bridge Middleware / subscriber Calls bus.publish()
▼	▼	▼
Subscribes to bus Receives validated events Updates NgRx slice	Contract Enforcement Schema validation Lifecycle management	Subscribes to bus Receives validated events Updates Redux / Zustand slice
▼	▼	▼

Table 1. The CFSCM coordination layer sits between Angular NgRx and React Redux or Zustand stores. Framework-specific bridges translate internal store changes into typed event bus publications. The event bus validates each publication against the Contract Registry before delivery. No fragment holds a direct reference to another framework's store or internal state structures.

4.2. The State Boundary Taxonomy

The three state categories from Section III map directly onto CFSCM components. Diagram 2 shows the taxonomy with ownership, scope, lifecycle, examples, and event bus role for each category. Fragment-private state never touches the event bus and is never declared in the Contract Registry. Platform-shared state is published by its owning fragment whenever it changes; the bus retains the most recent validated publication for each persistent contract so that late-mounting fragments receive current state on subscription without waiting for a re-broadcast. Handoff state is published at the navigation moment and consumed by the receiving fragment on mount; the bus clears it after first delivery to prevent stale values from appearing in later navigations.

Table 2. State Boundary Taxonomy

State Boundary Taxonomy - Three Categories		
Fragment-Private State	**Platform-Shared State**	**Handoff State**
Owner Fragment team only	Owner Shell or designated fragment	Owner Producing fragment

Scope Never leaves the fragment	Scope All fragments can read	Scope One fragment to the next only
Lifecycle Lives with component	Lifecycle Persistent until revoked	Lifecycle Transient, consumed once
Examples UI open/close state Form validation errors Loading indicators	Examples Auth status Session token Feature flags Active promotions	Examples Selected SKU Configured bundle Eligibility result Cart deltas
Event Bus Not involved	Event Bus Published on change All subscribers notified	Event Bus Published at navigation Consumed once then cleared

Table 2. The three-tier state boundary taxonomy showing ownership, scope, lifecycle, examples, and event bus role for each category. Fragment-private state stays within the owning fragment. Platform-shared state flows persistently to all fragments. Handoff state flows one-to-one between specific fragments as a transient contract.

4.3. Contract Specification Format

Each shared state item is described by a contract object in the Contract Registry. A contract specifies: a unique contract ID used as the event bus channel name; a semantic version string allowing consumers to detect breaking changes; the owning fragment identifier; a list of permitted consumer fragment identifiers; the TypeScript interface describing the state shape; and a lifecycle policy of either persistent or transient. The Contract Registry lives in a shared library imported by all fragments at build time and serves as the single source of truth for all cross-fragment state. When a contract changes in a breaking way, the version string increments and any consumer holding an outdated contract reference produces a build-time type error, making the change detectable before it reaches production.

4.4. The Event Bus

The event bus is a compact, framework-agnostic JavaScript module wrapping the browser's CustomEvent API with contract validation. Its implementation targets under 2KB minified, a design constraint set during planning rather than a measured production figure. When a fragment publishes state, the bus verifies the channel name against the Contract Registry, confirms the publishing fragment is the declared owner, and checks the payload shape using a lightweight runtime schema derived from the TypeScript contract. If all checks pass, the bus dispatches a typed CustomEvent on the window. If any check fails, the bus logs a structured error and drops the publication. For transient handoff contracts, the bus marks state consumed after first delivery and ignores any subsequent deliveries for that navigation instance.

4.5. Framework Bridges

The NgRx Bridge is an NgRx Effect selecting relevant store slices and calling the event bus publish method when they change. It runs inside the Angular fragment's existing Effects pipeline with no modifications to reducers, actions, or selectors. The Redox Bridge is a middleware function intercepting relevant dispatched actions and publishing post-reducer state to the event bus. The Stand Bridge is a store subscriber watching relevant slices and publishing changes. In every case, the bridge is the only CFSCM-aware code in the fragment. The store itself stays idiomatic to its framework and completely unmodified. A team adopting the CFSCM writes one bridge per fragment and continues working within their framework's native patterns.

5. Implementation Patterns

5.1. Shared Session State

Authentication status and user identity are the most universally needed platform state. The Account fragment, built in Angular NgRx, owns the session contract. When a user authenticates successfully, the NgRx Bridge publishes a session event containing a user identifier, session token, expiry timestamp, and list of active entitlements. Every other fragment subscribes to the session channel on mount and stores a local copy for its own rendering. When the session expires or the user signs out, the Account fragment publishes a revocation event and all consumers clear their local copy. Because the session contract is persistent, the bus delivers the most recent validated session state to any new subscriber immediately on subscription. A fragment mounting mid-session does not wait for a re-broadcast.

5.2. Funnel-Stage Handoff State

When a user completes product selection in the Catalog fragment and navigates forward, the Catalog fragment publishes a product selection handoff event containing the chosen SKU, selected tier, and any bundle components. The Configuration fragment consumes this on mount, uses the data to pre-populate its configuration form, and the bus clears the handoff. If the user returns and changes their selection, a fresh handoff event is published and the previous one is never seen by Configuration. When the user proceeds from Configuration to Checkout, the Configuration fragment publishes a second handoff event containing the finalized bundle, computed price, and eligibility determination. The Checkout fragment uses this to render a complete order summary on first render, removing a separate API round-trip and reducing latency at the most conversion-critical step in the funnel.

5.3. Cart State

Cart state sits between platform-shared and handoff state. The shell needs an up-to-date item count for its mini-cart component, and the Checkout fragment needs full cart

contents, but cart state is not consumed and cleared at a single boundary. The CFSCM handles this through a persistent contract owned by the Catalog fragment. Cart updates are published as typed delta events: item added, item removed, quantity changed. Consumers apply each delta in sequence to build a locally computed cart. A full cart snapshot is published at session start and on demand for any fragment that mounts after cart activity has already begun, ensuring late-mounting fragments receive correct state rather than an empty initial view.

5.4. Rollback and Recovery

When the user navigates backward in the funnel, the platform must clear any handoff state produced by the abandoned path before the user re-enters it and makes a new selection. The shell owns a navigation contract. When it detects backward navigation, it publishes a navigation event specifying the origin and destination fragment identifiers. Any fragment holding transient handoff state tied to the abandoned path responds by publishing a revocation event for that contract, clearing the stale data. Every transient contract in the registry declares which navigation events trigger its revocation, making backward navigation behavior explicit, discoverable, and independently testable. Diagram 3 shows the state flow across all four fragments, organized by state category.

Table 3. Funnel State Flow across Four Fragments

	Account Angular NgRx	**Product Catalog React Redux**	**Configuration React Zustand**	**Checkout React Redux**
Platform-Shared State - Published by Account fragment, received by all fragments via event bus				
Publishes	Session token User ID Entitlements Feature flags			
Receives		Session token User ID Entitlements Feature flags	Session token User ID Entitlements Feature flags	Session token User ID Entitlements Feature flags
Handoff State - Transient, one-to-one at each funnel boundary, cleared after delivery				
Catalog handoff		Publishes: Selected SKU Product tier Bundle items	Receives and uses to pre-populate configuration form	
Config handoff			Publishes: Bundle config Computed price Eligibility result	Receives and renders full order summary on first load
Cart State - Persistent delta contract, owned by Catalog, subscribed by shell and Checkout				
Cart updates		Publishes deltas: Item added Item removed Qty changed		Subscribes to deltas, applies each in order to build full cart

Table 3. State flow organized by category across the four-fragment funnel. Platform-shared state originates from the Account fragment and reaches all others via the event bus. Handoff state moves one-to-one at each funnel boundary and is cleared after delivery. Cart state flows as persistent deltas from the Catalog fragment to the shell and Checkout.

6. Experimental Evaluation

6.1. Simulation Setup

Note (Simulation): All results in this section are derived from simulation. These figures illustrate expected model behavior and are a starting point for understanding tradeoffs. Controlled A/B testing on live traffic is required before drawing conclusions about real-world impact.

A four-fragment platform was simulated matching the architecture in Section IV: an Angular NgRx Account fragment, a React Redux Catalog fragment, a React Zustand

Configuration fragment, and a React Redux Checkout fragment. Five thousand synthetic user sessions covered three purchase journey types: a direct single-product purchase, a multi-fragment bundle purchase requiring configuration, and a session with one backward navigation and re-selection. Two conditions were measured: a baseline using shared window globals and informal custom events with no schema enforcement, and the CFSCM with the Contract Registry and a validated event bus. Inter-fragment coupling was measured by counting direct code references from one fragment's source into another fragment's internal state structures, store selectors, or action creators, specifically the number of places where one team's codebase takes an unmanaged dependency on another team's internals.

6.2. Results Summary

The most practically meaningful result is the coupling reduction. Direct cross-fragment code references dropped from 34 in the baseline to 13 under the CFSCM, a 62% reduction. The baseline's 34 references included direct NgRx selector imports from the Angular fragment used inside React components, Redux action creators referenced across team boundaries, and untyped window property reads with no schema contract. The 13 remaining references under the CFSCM are all to shared TypeScript interfaces in the Contract Registry, which is the governed form of cross-fragment dependency. When any fragment changes its internal state structure under the CFSCM, only explicitly registered consumers are affected rather than any fragment that happened to read from the window object.

State synchronization latency was 4ms at the median and 11ms at the 95th percentile, with approximately 1.5ms from the contract validation step. For state transfers occurring at navigation boundaries, when the browser is already occupied loading the next fragment's bundle, this overhead is not perceptible. Developer onboarding time was estimated at 3.5 days in the baseline and 2.0 days with the CFSCM, a 43% improvement modeled on the reduction in undocumented information a new engineer must discover before safely adding a new fragment.

Table 4. CFSCM Evaluation Results vs. Window Global Baseline (Simulated)

Metric	Baseline (window globals)	With CFSCM	Change	Notes
Sync latency median	N/A (synchronous)	4ms	-	Event bus and validation overhead
Sync latency p95	N/A (synchronous)	11ms	-	Includes schema check (~1.5ms)
Cross-fragment refs	34 direct references	13 contract refs	-62%	Registry dependencies only
Contract violations	Undetected silently	Logged and dropped	Detectable	Runtime enforcement added
Onboarding estimate	~3.5 days	~2.0 days	-43%	Task model, not measured

7. Challenges and Limitations

7.1. Contract Version Discipline

The contract versioning mechanism works when teams follow it. Under delivery pressure, discipline can slip. A team might update a shared state shape without incrementing the version, and the build-time type error that should catch this will not fire if the Contract Registry update and the consuming fragment's code update land in the same deploy. A CI step comparing contract versions against a published baseline would close this gap. This tooling is not yet part of the reference implementation and represents the most concrete near-term need.

7.2. Event Bus as a Shared Runtime Dependency

The event bus is a runtime dependency shared by all fragments. If it fails to load, due to a CDN error, a script exception during initialization, or a network timeout, all cross-fragment state synchronization stops. Fragments still render using their own internal state but will not receive platform-shared updates. Each fragment needs a graceful degradation path: a defined timeout after which it falls back to a neutral or loading state, or triggers a lightweight session check of its own, rather than rendering with stale or empty shared data.

7.3. Schema Validation Scope

The runtime schema validation step adds approximately 1.5ms per event in simulation. For high-frequency state updates such as publishing on every keystroke in a search field, this overhead would accumulate. The CFSCM is designed for state that changes at navigation boundaries, not continuous reactive state. High-frequency UI state should stay fragment-private. Routing it through the event bus would misuse the architecture and invalidate the latency characteristics on which these estimates are based.

7.4. Simulation vs. Production Validity

All results in Section VI come from synthetic simulation. Real platforms introduce variability that parametric simulation does not fully capture: CDN latency variation across regions, underpowered devices where JavaScript takes materially longer, race conditions between fragment mount order and event bus initialization, and memory pressure from multiple framework runtimes running simultaneously. The coupling and onboarding estimates depend on assumptions about codebase structure and team workflow that will differ across organizations. Controlled A/B experimentation on live traffic is needed before these results can be treated as reliable production estimates.

8. Discussion and Future Work

8.1. Implications of Explicit State Contracts

The CFSCM makes state contracts a first-class engineering artifact rather than an informal team convention. The practical implication is that state sharing becomes something teams design deliberately before them build,

rather than something that accumulates as features are added. That shift addresses the organizational root cause of the failure modes in Section III-C: teams break state contracts not because they are careless but because those contracts were never written down in a form that could be checked or enforced.

8.2. Contract Compliance and Observability

A concrete extension is automated compliance checking in CI. The Contract Registry currently provides structural type safety at build time but cannot detect semantic violations. A fragment could publish to the correct channel with the correct shape but with values that violate an implied semantic constraint. Consumer-driven contract testing [10], adapted from microservices practice using tools such as Pact, could extend coverage to behavioral contracts. Contract observability is a related need: understanding which contracts are active, how frequently each channel is used, whether publications are being dropped due to validation failures, and what the runtime latency distribution looks like would give platform teams visibility into inter-fragment state health without touching individual fragments.

8.3. Generalization to Other Frameworks

The CFSCM was designed for Angular and React because those two frameworks appear together most frequently in enterprise multi-product platforms. The event bus is framework-agnostic, so extending the model to platforms including Vue, Svelte, or vanilla Web Components requires only writing the corresponding bridge. The Contract Registry and event bus remain unchanged. Documenting those bridges is a near-term contribution that would widen the model's applicability significantly.

9. Conclusion

Cross-framework state management in multi-product micro-frontend platforms is a real and recurring engineering problem that existing literature does not address with enough specificity for practitioners to act on. Teams facing this problem tend to solve it locally, building informal conventions that create fragile cross-fragment dependencies: hard to see in code review, hard to test in isolation, and hard to diagnose when they silently break in production. The Cross-Framework State Contract Model is a structured alternative to those informal conventions. The taxonomy from Section III gives teams a shared vocabulary before they start building — a simple way to decide upfront whether a given state item ever needs to leave the fragment that owns it. Once that is settled, the contract format makes the ownership, shape, version, and lifecycle of shared state explicit and checkable. The event bus then enforces those contracts at runtime, with no framework needing to reach into another’s internals. And because the NgRx, Redux, and Zustand bridges are the only CFSCM-aware code in each fragment, existing stores stay completely untouched.

The simulated results — a 62% drop in inter-fragment coupling, synchronization latency under 12ms at the 95th percentile, and roughly a 43% reduction in estimated onboarding time — need to be treated as directional rather than definitive until they can be validated against live traffic. What the numbers do suggest is that the cost of adopting the model is low relative to the coupling it removes. A team could adopt only the taxonomy and the contract format, skip the event bus entirely, and still get most of the organizational benefit. The components are designed to be independently useful. State management at scale is not only a technical problem. It is a coordination problem. Systems break because teams do not agree on what is shared, who owns it, and how it moves. The CFSCM makes those agreements explicit and enforceable.

Acknowledgment

The author thanks the front-end engineering and distributed systems research communities whose published work on micro-frontend architecture, event-driven state patterns, and consumer-driven contract testing provided the conceptual foundation this paper builds on.

References

[1] D. Abramov and A. Clark, "Redux: A Predictable State Container for JavaScript Applications," GitHub, 2015. [Online]. Available: https://redux.js.org

[2] NgRx Contributors, "NgRx: Reactive State for Angular," GitHub, 2023. [Online]. Available: https://ngrx.io

[3] P. Hnilica, "Zustand: Bear necessities for state management in React," GitHub, 2021. [Online]. Available: https://github.com/pmndrs/zustand

[4] ThoughtWorks, "Micro Frontends," Technology Radar, Nov. 2016. [Online]. Available: https://www.thoughtworks.com/radar/techniques/micro-frontends

[5] M. Geers, Micro Frontends in Action. Shelter Island, NY, USA: Manning Publications, 2020.

[6] C. Jackson, "Micro Frontends," martinfowler.com, Jun. 2019. [Online]. Available: https://martinfowler.com/articles/micro-frontends.html

[7] W3C, "Web Components Specification," W3C Working Draft, 2023. [Online]. Available: https://www.w3.org/TR/components-intro/

[8] MDN Web Docs, "CustomEvent API," Mozilla Developer Network, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent

[9] Nx Contributors, "Nx: Smart Monorepos - Fast CI," 2024. [Online]. Available: https://nx.dev

[10] M. Fowler, "Consumer-Driven Contracts," martinfowler.com, Jun. 2011. [Online]. Available: https://martinfowler.com/articles/consumerDrivenContracts.html

[11] Micro Frontends, "Micro Frontend Principles," micro-frontends.org, 2023. [Online]. Available: https://micro-frontends.org

[12] A. Leitner, S. Schulte, and U. Zdun, "Micro Frontend Architecture: A Systematic Mapping Study," in Proc. IEEE International Conference on Software Architecture (ICSA), 2021, pp. 112-122.

[13] D. Abramov, "You Might Not Need Redux," Medium, 2016. [Online]. Available: https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

[14] NgRx Team, "NgRx: When Should I Use NgRx?" NgRx Documentation, 2023. [Online]. Available: https://ngrx.io/guide/store/why

[15] M. Fowler, "Technical Debt," martinfowler.com, 2019. [Online]. Available: https://martinfowler.com/bliki/TechnicalDebt.html