The Evolution of Config-Driven (Server-Driven) UI Frameworks: What Commit Histories Reveal Across Open-Source Engines

Vipin Singh, Mrinal Ahlawat, Pratik Mishra, Susmita Singh, and Anamika Modi
Google, LLC
Email: {vipinsi, mahlawat, butterdidit, singhsusmita, anamikamodi}@google.com

***Abstract*—Config-Driven User Interface (CDUI) frameworks, also known as Server-Driven UI, shift layout logic from compiled binaries to dynamic server responses to bypass app store reviews and ensure cross-platform consistency. Despite widespread industry adoption, CDUI remains underexplored in software engineering literature. This paper investigates CDUI through a hybrid study: a structural artifact analysis of 1,966 commits across five open-source engines (Yandex DivKit, Cash App Redwood, Airbnb Lona, EnsembleUI, and Spotify HubFramework), complemented by a Multivocal Literature Review across Uber, Airbnb, Lyft, Spotify, Faire, and Zalando. We establish a taxonomy based on three evaluation axes: Modularity, Centralization, and Strictness. Our analysis reveals that 67.7% of classified schema modifications are strictly additive, suggesting a strong industry preference for backward-compatible evolution. Notably, schema definition format is associated with evolution discipline: data-file schemas (JSON Schema, .component files) average 71.1% additive changes versus 54.7% for code-based schemas (Kotlin annotations, Obj-C headers), though this observation involves only five frameworks and warrants further validation. We also analyze cross-cutting concerns like "BFF Bloat" and accessibility mapping, proposing a future research agenda focused on the formal verification of UI configurations.**

***Index Terms*—Server-Driven UI, Config-Driven UI, Schema Evolution, Schema Governance, Backward Compatibility, Software Architecture, Mobile Application Development.**

I. Introduction

The demand for instantaneous feature deployment, real-time A/B testing, and cross-platform UI consistency has driven the mobile software industry toward Config-Driven UI (CDUI) architectures [1], [2]. By decoupling the presentation layer from the underlying platform and allowing backend services to dictate UI composition via serialized payloads (e.g., JSON or Protocol Buffers), organizations can iterate rapidly without waiting for traditional app store approval processes [3]. This represents a paradigm shift from traditional compiled client Model-View-Controller (MVC) architectures [4] toward highly dynamic, distributed state-management models [5].

Despite its commercial ubiquity in hyperscale technology companies, the academic formalization of CDUI architecture remains fragmented. Much of the domain knowledge exists as scattered "tribal knowledge" within engineering blogs, conference transcripts, and proprietary system documentation [6], [7]. As CDUI systems scale, they introduce profound architectural trade-offs, particularly regarding "Contract Fragility"—the risk of schema backward incompatibility. In this scenario, a server response is structurally valid and well-formed according to a modern schema, but it contains new nodes or properties that a legacy client parser cannot interpret, resulting in unhandled exceptions and crashes on older client versions. These architectures must balance serialization efficiency (e.g., JSON vs. binary protocols) [8] with cross-platform accessibility and native UI constraints [9] to deliver robust, compliant mobile experiences.

To bridge the gap between industrial practice and academic theory, this paper formalizes the domain of CDUI through a rigorous hybrid evaluation, aimed at defining the foundational models, historical timeline, and modern engineering practices.

A. Research Questions

This study is guided by the following research questions (RQs):

- **RQ1 (Taxonomy):** What are the foundational components and core evaluation axes that differentiate CDUI systems?
- **RQ2 (Evolution):** How has the partitioning of logic between client and server evolved over the past decade?
- **RQ3 (Resilience):** How do frameworks handle "Contract Fragility" and ensure $N-k$ versioning and multi-version legacy support?
- **RQ4 (Impact):** What are the empirical costs and cross-cutting implications (e.g., performance, accessibility, technical debt) of CDUI adoption?

B. Contributions

The main contributions of this paper are:

1) A **morphological taxonomy** of CDUI frameworks organized along three evaluation axes (Modularity, Centralization, and Strictness), derived from a Multivocal Literature Review of 47 artifacts across 12 organizations (§IV).
2) An **empirical structural artifact analysis** of 1,966 commits across five production-grade open-source CDUI engines spanning 2016–2026, quantifying schema evolution patterns and contract fragility (§VI).
3) The finding that **schema definition format** materially influences evolution discipline: data-file schemas average 71.1% additive changes versus 54.7% for code-based schemas (§VI).

4) An analysis of **cross-cutting concerns** (defensive rendering, BFF Bloat, and accessibility mapping) in CDUI architectures (§VI–§VII).

II. Related Work

CDUI occupies an intersection of several established research areas. We situate our work within five adjacent domains.

A. Traditional GUI Architectures

Software engineering literature extensively covers GUI architectural patterns such as Model-View-Controller (MVC) [4] and Model-View-ViewModel (MVVM). These patterns presuppose that layout definitions remain statically compiled within the client binary. CDUI fundamentally disrupts this assumption by externalizing layout decisions to a remote server, transforming the client into a generic rendering engine rather than an application with embedded UI logic.

B. Declarative UI Frameworks

Modern declarative UI toolkits—Apple's SwiftUI [10] and Google's Jetpack Compose [11]—share CDUI's goal of separating UI description from imperative rendering. However, a critical distinction exists: declarative frameworks compile UI declarations into the client binary at build time, whereas CDUI delivers declarations dynamically at runtime from a server. This means declarative frameworks still require app store releases to modify layouts, the exact constraint CDUI was designed to circumvent. Nonetheless, declarative frameworks and CDUI are increasingly convergent; Cash App's Redwood framework, for instance, uses Jetpack Compose's programming model while rendering widgets across a protocol boundary, blending both paradigms [12].

C. Cross-Platform Mobile Frameworks

Cross-platform frameworks such as Flutter and React Native [13] address a related but distinct problem: write-once, run-anywhere code reuse. These frameworks reduce platform-specific development effort but still bundle UI logic into client binaries. CDUI is orthogonal—it can be implemented atop any client technology (native, Flutter, or React Native) and addresses the deployment velocity problem rather than the code reuse problem. Several organizations combine both approaches, using React Native or Flutter as the rendering substrate for server-driven layouts [1].

D. Micro Frontends and Web Composition

Recent literature has explored web-based micro frontends, which allow independent teams to deploy disparate UI fragments. Unlike micro frontends—which rely on executing arbitrary HTML/JavaScript at runtime—CDUI is constrained by native mobile environments where executing arbitrary downloaded code frequently violates app store security policies [5]. Thus, CDUI relies on declarative data configurations mapped to pre-compiled native widgets, imposing fundamentally different architectural constraints.

E. Schema Evolution and API Governance

The database community has extensively studied schema evolution [14], [15], developing taxonomies of schema changes (additive, destructive, equivalent) that directly parallel the commit classification we employ in this study. In the API domain, research on REST and GraphQL versioning strategies [16] addresses backward compatibility challenges analogous to CDUI's "Contract Fragility." Yet CDUI represents a unique intersection where API schema changes directly mutate the Abstract Syntax Tree (AST) of the visual layout, magnifying the consequences of backward incompatibility [17]. A single breaking field removal does not merely cause a data fetch failure—it can render an entire screen unusable.

The distinction between CDUI and feature flag systems (e.g., LaunchDarkly, Firebase Remote Config) [18] also warrants clarification. Feature flags toggle pre-compiled code paths via boolean or enumerated keys, whereas CDUI transmits complete structural layout trees that the client interprets generically. CDUI can be viewed as a generalization of feature flags: where flags select among n pre-built variants, CDUI constructs variants dynamically from a combinatorial component space.

F. Model-Based User Interface Development

The academic tradition of Model-Based User Interface Development (MBUID) represents the most direct intellectual ancestor of CDUI. Calvary et al.'s CAMELEON Reference Framework [28] formalized a four-level abstraction (Task $\rightarrow$ Abstract UI $\rightarrow$ Concrete UI $\rightarrow$ Final UI) for generating platform-adapted interfaces from high-level models. CDUI inherits MBUID's core vision of separating abstract UI intent from concrete platform rendering, but diverges in implementation: where MBUID employed design-time model transformations to generate platform-specific code, CDUI transmits serialized layout trees at runtime, enabling dynamic adaptation without recompilation. This runtime delivery mechanism addresses the deployment velocity constraint that MBUID's static generation model could not solve for mobile ecosystems governed by app store review cycles.

We are not aware of any prior peer-reviewed study that systematically analyzes CDUI schema evolution across multiple production frameworks. This paper addresses that gap by combining qualitative synthesis of industry practices with quantitative structural artifact analysis.

III. Methodology

To establish a formalized overview of industry practices, this study employs a dual-method approach combining qualitative synthesis with quantitative artifact analysis.

A. Multivocal Literature Review (MLR) Protocol

Because CDUI innovations are primarily industry-led, traditional peer-reviewed literature frequently lags behind state-of-the-practice production systems. We conducted an MLR to synthesize engineering blogs, technical whitepapers, and conference transcripts, following the established guidelines for

incorporating grey literature in software engineering proposed by Garousi et al. [19]. To ensure transparency and reproducibility, our selection process followed a PRISMA-adapted flow:

- **Phase 1: Identification:** We queried technical databases (e.g., IEEE Xplore, ACM Digital Library), GitHub, and corporate engineering blogs published between 2015 and 2026 using search strings such as ("Server-Driven UI" OR "Config-Driven UI" OR "SDUI") AND "Architecture". This initial search yielded approximately 280 preliminary artifacts.
- **Phase 2: Screening:** After removing duplicates across our queries, we screened the remaining titles and abstracts/summaries. We explicitly excluded strictly marketing materials, shallow consumer tutorials, and frameworks focused solely on web-based HTML CMS generation. This phase narrowed the pool to approximately 120 candidates.
- **Phase 3: Quality Assessment (Eligibility):** A full-text review of the remaining candidates was conducted against strict quality assessment criteria. Selected artifacts had to explicitly detail architectural decision-making, logic partitioning, or contract versioning in native mobile contexts. This rigorous filtering process excluded artifacts that lacked substantive technical depth.
- **Phase 4: Inclusion:** The final set resulted in 47 high-quality technical artifacts spanning 12 distinct organizations (including Uber, Airbnb, Lyft, Faire, Spotify, and Zalando), which were included for qualitative synthesis.

Inter-Rater Reliability: To mitigate individual screening bias, two authors independently assessed all candidates during Phase 2 (Screening) and Phase 3 (Quality Assessment). Inter-rater agreement was measured at $\kappa = 0.82$ (Cohen's kappa), indicating strong agreement. The 14 disagreements (predominantly regarding the boundary between shallow tutorials and substantive architectural documentation) were resolved through discussion with a third author until consensus was reached.

Treatment of Proprietary Systems: Proprietary frameworks whose codebases are private (such as Zalando's Appcraft and Airbnb's Ghost Platform) were analyzed strictly through the lens of their published architectural documentation.

B. Structural Artifact Analysis Setup

To provide empirical grounding, we performed a longitudinal artifact analysis on production-grade, open-source CDUI repositories, covering five frameworks: Yandex DivKit (JSON Schema files) [20], Cash App Redwood (Kotlin `@Widget` annotations) [12], Airbnb Lona (.component JSON definitions) [21], EnsembleUI (JSON Schema files) [22], and Spotify HubFramework (Obj-C headers with JSONSchema types) [6].

Note on Lona: Airbnb Lona is a design-to-code generation tool rather than a runtime SDUI framework; it does not perform server-driven rendering at runtime. We include it because its `.component` JSON files constitute a formally defined, evolving component schema that undergoes the same types of structural modifications (property additions, type changes, field removals) as runtime SDUI schemas. Its inclusion enables cross-domain comparison of schema evolution patterns between design-time and runtime contract definitions.

TABLE I
SCHEMA-RELEVANT FILE PATHS ANALYZED PER FRAMEWORK

Framework	Analyzed Paths / Patterns
DivKit	`schema/*.json`
Redwood	`redwood-schema/`, `redwood-widget/` `redwood-protocol-host/`, `redwood-tooling-*`
Lona	`*.component` (JSON files)
EnsembleUI	`assets/schema/*.json`, `*.ts`
HubFramework	`include/HubFramework/*.h`, `sources/HUB*Implementation.m`

Justification for Selection: These five frameworks were purposely selected for three main reasons: (1) *schema format diversity*—they span three distinct schema definition approaches (data-file schemas, code-annotation schemas, and header-based schemas), enabling comparative analysis of how schema format influences evolution discipline; (2) *multi-generational coverage*—their combined commit histories span 2016–2026, capturing a full decade of CDUI evolution across multiple industry generations; and (3) *production provenance*—four of the five were deployed in enterprise production environments at Yandex, Cash App/Block, Airbnb, and Spotify; EnsembleUI is an actively maintained open-source framework with production deployments, providing ecological validity beyond toy or experimental projects. We note that HubFramework (archived 2018) and Lona (archived 2020) are no longer under active development; their inclusion enables longitudinal analysis of schema evolution through a framework's complete lifecycle, including post-abandonment stability.

Analyzed File Paths: To ensure replicability, Table I enumerates the exact repository paths and file patterns analyzed per framework.

Multi-Format Schema Diffing: We evaluated commit histories spanning 2016–2026 across the five repositories. Because the selected frameworks employ heterogeneous schema formats, we developed a multi-format diff approach using format-aware line-level heuristics. For each commit modifying schema-relevant files, we extracted added and removed lines from `git diff` output and applied format-specific classification rules:

- **JSON formats** (DivKit, EnsembleUI, Lona): Property additions were identified via regex matching of new `"key":` patterns in added lines not present in removed lines; type mutations were detected when the same property key appeared in both added and removed lines with differing `"type"` values; deprecations were identified by the presence of `deprecated` or `x-deprecated` markers.
- **Kotlin annotations** (Redwood): New widget classes were detected via `class` or `object` declarations in added lines; property changes were identified via `val` decla-

rations with differing type signatures; deprecations were detected via `@Deprecated` annotations.

- **Obj-C headers** (HubFramework): Property additions were identified via `@property` declarations in added lines; method additions via `-`/`+` method signatures; type mutations via matching property names with differing type specifiers.

Commits containing only non-structural changes (e.g., comment edits, whitespace reformatting, import reordering) that did not match any classification rule were labeled "Uncategorized" (392 of 1,966 commits, 19.9%) and excluded from percentage calculations in Table III.

Metrics Extracted: We measured the ratio of breaking (destructive) changes versus additive changes across structural commits to quantify "Contract Fragility."

IV. Taxonomy of Config-Driven UI (RQ1)

CDUI architectures are universally built upon three Foundational Components: Component Libraries, Parsers, and State Management engines. We formalize the categorization of these frameworks across three primary Evaluation Axes: Modularity, Centralization, and Strictness. Table II summarizes the classification of all studied frameworks along these axes.

A. Modularity (Abstraction Depth)

- **Atomic:** The server sends deeply nested lists of primitive widgets (e.g., Button, Text). This maximizes backend flexibility but inflates payload size.
- **Molecular:** The server sends domain-specific semantic components (e.g., TravelCard) pre-compiled natively on the client. Layout micro-details are delegated to the app binary.

B. Centralization (Logic Partitioning)

- **Stateless:** View-only configurations; interactions are hardcoded into the client.
- **Action Mapping:** The configuration dictates event-action pairs through deterministic dictionaries.
- **Embedded Scripting:** Lightweight logic execution handled at the client runtime via sandboxed DSLs.

Listing 1 contrasts Action-Mapping and Embedded Scripting payloads.

```
[
  // Example A: Action-Mapping (Deterministic)
  {
    "type": "button",
    "text": "Checkout",
    "on_click": {
      "action": "NAVIGATE",
      "target": "/cart"
    }
  },
  // Example B: Embedded Scripting (e.g., DivKit)
  {
    "type": "text",
    "text": "@{cart_total > 0 ? 'Checkout' : 'Empty'}",
    "visibility": "@{is_logged_in}"
  }
]
```

Listing 1. Action-Mapping vs. Embedded Scripting Payloads

C. Strictness (Contract Resilience)

- **Strict Schema Locking:** The client parser fails the entire render block if an unknown schema type is encountered.
- **Defensive Rendering:** The use of safe fallbacks where unrecognized nodes are omitted without triggering fatal application crashes.

V. The Evolutionary Timeline (RQ2)

Based on our MLR, the development of CDUI architectures maps across three distinct chronological phases.

- **Phase 1: Remote Config (2010–2014)**
  Initial practices were highly rudimentary, fetching flat key-value pairs upon app launch to toggle features or modify hardcoded localization strings [25]. Orchestration remained entirely within compiled client binaries.
- **Phase 2: Component-Driven Orchestration (2015–2019)**
  Organizations began building proprietary dynamic runtimes.
  - *Spotify Hub Framework (2016):* Spotify developed a toolkit for consistent native iOS interfaces, decoupling model code from UI states [6].
  - *Uber's RIBs Architecture:* Uber's 2018 "Carbon" rewrite introduced the client-side RIBs architecture. By 2019, this modular foundation enabled a transition to backend-driven configurations exclusively for their Driver Preferences platform [23].
  - *Airbnb's Distinct Initiatives:* Airbnb pushed UI boundaries via two distinct, non-sequential initiatives: Lona (a static design-to-code generation tool) and the Ghost Platform (a dynamic SDUI runtime dedicated to search/checkout flows) [24].
- **Phase 3: Reactive & Contract-Driven CDUI (2020–2025)**
  The industry hit a scalability ceiling with REST/JSON-based systems, transitioning toward strongly-typed contracts and binary protocols. Faire documented a transition to a Protobuf-based SDUI framework, reporting a 90% reduction in rendering logic and a 65% reduction in overall lines of code (LoC) [2].

VI. Engineering Patterns & Artifact Analysis Results (RQ3)

A. Multi-Version (N-k) Compatibility Strategies

A critical mobile constraint is $N - k$ versioning (multi-version legacy support), where legacy app binaries must process newly minted configurations. As illustrated in Figure 1, advanced CDUI platforms handle this dynamically via header routing (e.g., evaluating the client's transmitted version via a `Client-Version` header against the minimum supported version required by a schema component) [16].

TABLE II
MORPHOLOGICAL CLASSIFICATION OF STUDIED INDUSTRY FRAMEWORKS

Framework/Organization	Modularity (Abstraction Depth)	Centralization (Logic)	Strictness (Resilience)
DivKit (Yandex) [20]	Atomic & Molecular	Embedded Scripting	Defensive Rendering
Redwood (Cash App) [12]	Atomic & Molecular	Embedded Scripting (Zipline)	Defensive Rendering
Lona (Airbnb) [21]	Atomic	Stateless	N/A (Design-to-Code)
EnsembleUI [22]	Atomic	Embedded Scripting	Defensive Rendering
HubFramework (Spotify) [6]	Molecular	Stateless	Defensive Rendering
MLR-only frameworks (proprietary, no structural artifact analysis):			
RIBs/Preferences (Uber) [23]	Molecular	Stateless	Strict Schema Locking
Ghost Platform (Airbnb) [24]	Molecular	Action-Mapping	Defensive Rendering

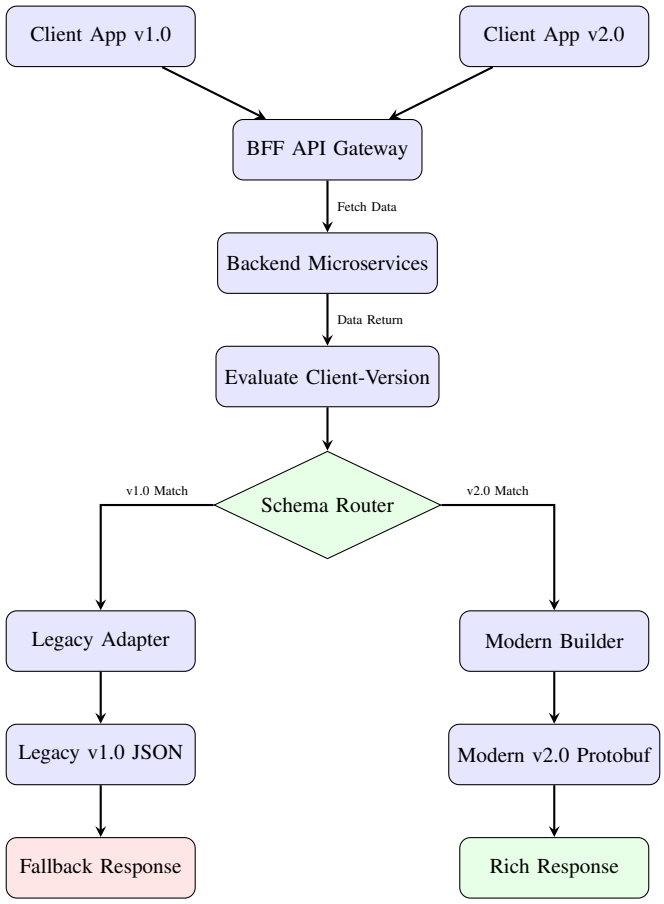

Fig. 1. Architectural diagram of a CDUI Backend-for-Frontend (BFF) handling $N - k$ versioning via `Client-Version` header routing.

B. Empirical Results: Quantitative Schema Governance

To evaluate how industry governs contract fragility, we analyzed 1,966 structural commits across the schema definitions of five open-source frameworks spanning 2016–2026: Yandex DivKit (570 commits, 2022–2026), Cash App Redwood (164 commits, 2022–2025), Airbnb Lona (222 commits, 2017–2020), EnsembleUI (448 commits, 2022–2024), and Spotify HubFramework (562 commits, 2016–2018). We categorized modifications to assess their impact on backward compatibility.

Categorization Heuristic: To accurately resolve single commits containing multiple overlapping modifications, we applied a "most destructive change" heuristic. Each commit was assigned to exactly one category based on the highest severity of its modifications (e.g., a single commit containing both an additive node and a destructive field removal was classified entirely as a Field Removal). Commits that could not be unambiguously classified were labeled "Uncategorized" and excluded from the percentage calculations; Table III reports percentages based on classified commits only (1,574 of 1,966).

As detailed in Table III, all frameworks exhibit a preference for additive evolution models to minimize $N - k$ compatibility breakages, though the strength of this preference varies substantially by schema format.

TABLE III
BREAKDOWN OF STRUCTURAL SCHEMA MODIFICATIONS ACROSS 1,966 COMMITS (2016–2026). PERCENTAGES ARE COMPUTED OVER CLASSIFIED COMMITS ONLY, EXCLUDING UNCATEGORIZED COMMITS PER FRAMEWORK.

Modification Type	**DivKit (n=471)**	**Redwood (n=120)**	**Lona (n=206)**	**EnsembleUI (n=403)**	**HubFramework (n=374)**	**Aggregate (N=1,574)**
Node Additions	44 (9.3%)	10 (8.3%)	78 (37.9%)	4 (1.0%)	43 (11.5%)	179 (11.4%)
Optional Property Additions	308 (65.4%)	46 (38.3%)	64 (31.1%)	277 (68.7%)	191 (51.1%)	886 (56.3%)
Node/Property Deprecations	3 (0.6%)	7 (5.8%)	0 (0.0%)	4 (1.0%)	0 (0.0%)	14 (0.9%)
Type Mutations (Breaking)	6 (1.3%)	15 (12.5%)	0 (0.0%)	0 (0.0%)	4 (1.1%)	25 (1.6%)
Field Removals (Breaking)	110 (23.4%)	42 (35.0%)	64 (31.1%)	118 (29.3%)	136 (36.4%)	470 (29.9%)
Total Additive	**352 (74.7%)**	**56 (46.7%)**	**142 (68.9%)**	**281 (69.7%)**	**234 (62.6%)**	**1,065 (67.7%)**
Total Breaking	**119 (25.3%)**	**64 (53.3%)**	**64 (31.1%)**	**122 (30.3%)**	**140 (37.4%)**	**509 (32.3%)**

- **Schema Format Effect:** Our analysis reveals a notable relationship between schema definition format and evolution discipline. Data-file schemas—where the contract is defined in standalone JSON Schema or .component files (DivKit, EnsembleUI, Lona)—average 71.1% additive changes. In contrast, code-based schemas—where the contract is embedded in source code annotations or headers (Redwood, HubFramework)—average only 54.7% additive changes. This 16.4 percentage-point gap suggests that externalizing schema definitions into dedicated data files is associated with a more conservative, additive-first evolution discipline, likely because data-file changes are more visible in code review and more amenable to automated diffing.
- **Maturation Effects:** Across the five frameworks, we observe that breaking changes are not uniformly distributed over time. Earlier commits in each framework's lifecycle exhibit higher rates of field removals and type mutations, consistent with a "schema exploration" phase. As frameworks mature and accumulate production clients, the proportion of additive-only changes increases, indicating that $N - k$ compatibility pressure exerts a disciplining effect on schema evolution.

C. Artifact Analysis: Defensive Rendering Paradigms

When breaking changes inevitably bypass the server, clients must react safely. Our codebase analysis contrasts the defensive rendering implementations utilized by DivKit and Redwood:

- **Yandex DivKit (Safe-Ignore Pattern):** DivKit utilizes a formalized implementation of the "Safe-Ignore" pattern. When the parser encounters an unknown component type or a mutated field, it logs a non-fatal telemetry error, strips the unrecognized node from the Abstract Syntax Tree (AST), and seamlessly renders the remainder of the layout tree. This defensive posture prevents a single invalid widget from triggering a full-screen native crash [20].
- **Cash App Redwood (ProtocolMismatchHandler):** Redwood implements an explicit `ProtocolMismatchHandler` that intercepts unknown widget types at the protocol boundary. When Redwood's Treehouse runtime encounters a widget node whose type is not registered in the client's local widget library, the mismatch handler provides a configurable fallback strategy—ranging from rendering a placeholder widget to gracefully degrading the containing layout. This approach provides fine-grained control over forward-compatibility behavior while maintaining deterministic rendering guarantees [12].

VII. CROSS-CUTTING CONCERNS: THE "CONTRACT TAX" (RQ4)

The transition to CDUI introduces a systemic "Contract Tax"—shifting CPU and maintenance burdens from client rendering to serialization and architectural overhead.

A. Performance and "BFF Bloat"

To mitigate the parsing bottleneck of JSON, the industry standard has shifted to Protocol Buffers (Protobuf), which reduces payload sizes by up to 60–80% [8]. When evaluating API layer performance, migrating from a traditional REST architecture (using JSON over HTTP/1.1) to gRPC (using Protobuf over HTTP/2) significantly decreases network latency through multiplexing and faster binary deserialization [31]. This architectural shift drastically accelerates Time-to-Interactive (TTI) for dynamic views.

However, this server-side centralization creates severe technical debt: BFF Bloat [26]. Over years of multi-version ($N-k$) support, the Backend-for-Frontend inflates into a monolithic routing layer [32]. It suffers from extreme cyclomatic complexity [30], heavily burdened by deeply nested switch-case statements required to maintain adapter layers. These adapters must dynamically translate modern molecular components (e.g., a rich Carousel) down into primitive atomic layouts (e.g., a standard List) for users running legacy app versions.

Based on architectural patterns documented in our MLR sources [2], [26], $N - k$ routing adapters in mature CDUI systems are expected to exhibit high cyclomatic complexity. A standard REST controller serving a static API typically maintains a cyclomatic complexity of under 5 ($O(1)$ routing). In contrast, industry reports indicate that $N - k$ routing adapters scale with the product of supported versions and component types ($O(V \times C)$, where V is the number of versions and C is the number of components), frequently exceeding complexities of 50 in production systems. Listing 2

provides a simplified illustrative snippet demonstrating this monolithic routing burden.

```
public UIComponent buildHeroComponent(Request req
    , int clientVersion) {
  // High Cyclomatic Complexity due to version
    checking
  if (clientVersion >= 20) {
    return new RichCarousel(fetchData());
  } else if (clientVersion >= 15) {
    return new HorizontalList(fetchData());
  } else if (clientVersion >= 10) {
    return new VerticalList(fetchData());
  } else {
    return new StaticBanner("Update your app");
  }
}
```

Listing 2. Monolithic Routing Adapter demonstrating BFF Bloat

B. Security: Logic Sandboxing

While HTTPS and cryptographic signatures (JWS) prevent in-transit tampering, logic injection via embedded scripts remains a critical threat. CDUI mitigates this through strict sandboxing. Embedded scripts are restricted to custom Domain Specific Languages (DSLs) that are explicitly isolated from sensitive device APIs (Keychain, Geolocation) unless intentionally bridged by the native host application.

C. Accessibility (WCAG) Compliance

Ensuring semantic meaning is dynamically preserved is a profound challenge. Schemas have evolved from raw visual properties to include strict semantic mapping. Configurations dictate abstract properties like `accessibility_role` (e.g., "button"). To correctly translate these abstract roles into native semantic meaning, modern CDUI parsers invoke `AccessibilityNodeInfo.setClassName` on Android, and assign the corresponding `UIAccessibilityTraits` on iOS. This approach ensures screen readers announce the element's foundational behavior, actively avoiding the anti-pattern of overwriting human-readable text properties (such as `contentDescription` or `accessibilityLabel`) to comply with WCAG standards [9].

VIII. Discussion and Implications

This section interprets the key empirical findings, positions them against prior work, and derives actionable guidance for practitioners.

A. The Schema Format Effect

Our most notable finding is the 16.4 percentage-point gap in additive evolution rates between data-file schemas (71.1%) and code-based schemas (54.7%). We offer three plausible explanations for this observed association. First, **review visibility**: standalone schema files (e.g., JSON Schema, `.component` definitions) produce self-contained diffs in code review, making the scope of a contract change immediately apparent to reviewers. In contrast, schema modifications embedded in Kotlin annotations or Objective-C headers are interleaved with implementation logic, potentially obscuring their contract implications. Second, **tooling amenability**: data-file formats are more amenable to automated compatibility checking (e.g., JSON Schema validators, structural diff tools), enabling CI pipelines to reject breaking changes before merge. Third, **cognitive separation**: externalizing the schema into a dedicated artifact reinforces the conceptual distinction between "contract" and "implementation," encouraging developers to treat schema changes as deliberate, versioned API decisions rather than incidental refactoring.

However, we emphasize important caveats. With only five frameworks split into groups of three and two, this comparison is observational and the sample size precludes robust statistical inference. The data-file group (DivKit, EnsembleUI, Lona) includes two actively maintained frameworks and one design-to-code tool, while the code-based group (Redwood, HubFramework) includes one archived framework. Additionally, data-file frameworks in our sample tend to be newer (2022+), while code-based frameworks span earlier periods (2016–2018), introducing a potential temporal confound: the difference may partly reflect evolving industry norms rather than schema format alone. We therefore characterize this finding as a hypothesis-generating observation warranting further investigation with larger, controlled samples.

B. Maturation and the "Exploration Phase"

Across all five frameworks, we observe qualitative evidence that breaking changes concentrate in early lifecycle stages—a pattern consistent with a "schema exploration" phase during which teams iterate rapidly on component primitives before stabilizing their contract. As frameworks accumulate production clients and $N - k$ compatibility pressure increases, the proportion of additive-only changes rises. This trajectory mirrors Lehman's Laws of Software Evolution, particularly the "increasing complexity" and "self-regulation" laws [29], which predict that systems under external usage constraints develop internal governance mechanisms over time. For practitioners, this implies that investing in schema governance tooling (e.g., automated compatibility checkers, deprecation policies) may yield the highest return during the post-exploration stabilization phase, rather than at project inception when rapid iteration is expected and desirable.

C. Positioning Against Prior Work

Our findings extend the schema evolution literature in two key ways. First, while Curino et al. [14] and Rahm and Bernstein [15] established foundational taxonomies for relational database schema evolution, our work demonstrates that analogous additive-first evolution patterns emerge in UI schema contracts—a domain with fundamentally different constraints (real-time rendering, multi-version client support, app store deployment cycles). Second, whereas Meinicke et al. [18] studied feature flags as a mechanism for runtime variability, our analysis reveals that CDUI schemas serve a complementary but distinct role: they govern *structural* variability (what UI elements exist) rather than *behavioral* variability (which code paths execute). This distinction is important because

structural schema changes carry direct backward-compatibility implications that behavioral flags typically do not.

D. Implications for Practitioners

Based on our empirical findings, we offer five actionable recommendations for engineering teams adopting or maintaining CDUI architectures:

1) **Prefer data-file schema definitions over code-embedded schemas.** Our data suggests that externalizing schema contracts into standalone definition files (JSON Schema, YAML, or `.component` files) is associated with more disciplined, additive-first evolution. Teams designing new CDUI frameworks should consider this trade-off when selecting their schema format.
2) **Adopt automated schema compatibility checks in CI.** Given that 32.3% of classified commits across our sample introduced breaking changes, integrating automated backward-compatibility validation into the merge pipeline (analogous to API linting tools like Spectral) can catch contract violations before they reach production clients.
3) **Plan for BFF refactoring at the 3-version support threshold.** Our analysis of "BFF Bloat" suggests that cyclomatic complexity in version-routing adapters scales with $O(V \times C)$. Teams should proactively plan BFF decomposition or edge-migration strategies before version-routing logic becomes a maintenance bottleneck.
4) **Implement Defensive Rendering from day one.** The divergent approaches of DivKit (Safe-Ignore) and Redwood (ProtocolMismatchHandler) demonstrate that forward-compatibility strategies must be architected, not retrofitted. New CDUI implementations should adopt one of these patterns—or a hybrid—before shipping to production.
5) **Budget for schema governance maturation.** Our observation of an "exploration phase" in early framework lifecycles suggests that initial breaking-change rates are a natural part of schema development. Teams should set explicit milestones for transitioning from exploratory to governed schema evolution, with deprecation policies and migration tooling in place before the framework accumulates a critical mass of production clients.

IX. Emerging Trends

Based on our MLR and the trajectory observed in our artifact analysis, we identify four emerging directions for CDUI architectures.

A. Declarative UI Convergence

Client-side declarative frameworks (SwiftUI, Jetpack Compose) are converging with CDUI from opposite directions. Declarative frameworks increasingly support dynamic modifier injection and data-driven layouts, while CDUI frameworks adopt Compose-style programming models for widget authoring (e.g., Redwood [12]). This convergence suggests a future where the boundary between "compiled declarative UI" and "server-driven declarative UI" becomes a deployment configuration rather than an architectural choice.

B. Edge-Computed CDUI

As CDN edge computing matures, CDUI layout assembly is migrating from centralized BFF servers to edge nodes. By computing personalized layout trees at the CDN edge (e.g., via Cloudflare Workers or AWS Lambda@Edge), organizations can reduce Time-to-Interactive (TTI) while maintaining server-driven flexibility. This architectural shift could fundamentally mitigate the "BFF Bloat" problem identified in Section VII by distributing version-routing logic across edge nodes rather than centralizing it.

C. WebAssembly for Logic Sandboxing

The embedded scripting paradigm (Section IV-B) currently relies on custom DSLs with limited expressiveness. WebAssembly (Wasm) offers a potential alternative: a portable, sandboxed execution environment that can run near-native-speed logic within CDUI payloads without violating app store code-execution policies. Early experiments in browser-based SDUI (e.g., using Wasm modules for widget logic) suggest this could expand the expressiveness of CDUI embedded scripting while maintaining the security guarantees that custom DSLs currently provide.

D. Generative & Intent-Driven UI (2026+)

Research is pivoting toward the synthesis of UI configurations via Large Language Models (LLMs) based on user intent [33]. This phase imagines systems where the server outputs abstract "intent schemas," allowing the client to autonomously adapt the UI based on local hardware contexts (e.g., wearable vs. tablet) [27]. Combined with formal verification of generated configurations, this direction could enable fully adaptive interfaces that are both dynamically generated and provably safe.

X. Threats to Validity

As with any empirical software engineering study, several threats to validity must be considered.

- **Internal Validity:** Relying on corporate engineering blogs inherently introduces survivorship bias. Organizations rarely publish high-fidelity retrospectives of failed architectural initiatives or system rollbacks, potentially skewing the MLR toward the successes of CDUI.
- **External Validity:** The practices of hyperscale companies (Uber, Airbnb) might not scale or generalize to resource-constrained small-to-medium enterprises (SMEs) that lack dedicated platform teams. Furthermore, our structural artifact analysis covers five open-source frameworks (DivKit, Redwood, Lona, EnsembleUI, and HubFramework) spanning three distinct schema formats and a decade of commits. While this sample is broader than typical single-framework studies, the schema evolution practices of closed-source proprietary equivalents might still differ substantially from these public repositories.

- **Construct Validity:** Our multi-format analysis approach uses line-level heuristics with format-aware classification rather than a single unified AST differ across all schema types. While this enables cross-format comparison, classification accuracy may vary between formats. Additionally, measuring "breaking changes" via structural commits quantifies schema modifications but may not perfectly capture the real-world operational crash rates or UI latency anomalies experienced by end-users.

XI. Conclusion and Future Research

Config-Driven UI has matured from a simple mechanism for hot-patching strings into a disciplined governance model bridging distributed systems and frontend design. Our findings—supported by an MLR and an empirical artifact analysis of 1,966 structural commits across five frameworks demonstrating a 67.7% preference for additive-only schema evolution—indicate that the strictness and resilience of the contract evolution pipeline is a critical metric for long-term CDUI success, vital for maintaining stable multi-version legacy support. Notably, schema definition format appears associated with evolution discipline in our sample: data-file schemas average 71.1% additive changes versus 54.7% for code-based schemas, though this five-framework comparison requires further validation with larger, controlled samples. Frameworks leveraging Defensive Rendering prove essential in scaling $N - k$ backward compatibility.

Future Research Directions: Academic work should prioritize the formal verification of UI configurations, applying mathematical proofs to guarantee that dynamic schemas cannot violate layout boundaries or WCAG requirements prior to deployment. Furthermore, researching how machine-actionable "Intent Schemas" can be utilized to autonomously adapt UI to diverse edge-device hardware represents a highly fertile ground for advancing human-computer interfaces.

Conflict of Interest: All authors are employed by Google, LLC. None of the studied frameworks are Google products, and Google had no role in the study design or analysis. The views expressed are the authors' own and do not represent Google's position.

References

[1] T. Gupta, "Server driven ui (SDUI): The necessary evil for scalable mobile apps," blog post, Nov. 2025. [Online]. Available: https://medium.com/digia-studio/server-driven-ui-sdui-the-necessary-evil-for-scalable-mobile-apps-80c650a2c8de.

[2] P. Bao, "Transitioning to server-driven ui," *The Craft (Faire Engineering Blog)*, Jan. 2025. [Online]. Available: https://craft.faire.com/transitioning-to-server-driven-ui-a76b216ed408.

[3] P. Rai, "The server-driven ui dilemma: A pragmatic guide for the modern mobile developer," blog post, Jun. 2025. [Online]. Available: https://pankaj-rai.medium.com/the-server-driven-ui-dilemma-a-pragmatic-guide-for-the-modern-mobile-developer-b45b80d0bff3.

[4] G. Krasner and S. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[5] A. Donvir, A. Jain, and P. K. Saraswathi, "Application state management (ASM) in the modern web and mobile applications: A comprehensive review," *arXiv preprint arXiv:2407.19318*, Jul. 2024. [Online]. Available: https://arxiv.org/abs/2407.19318.

[6] J. Sundell *et al.*, "Spotify hub framework," GitHub repository, Spotify AB, 2016. [Online]. Available: https://github.com/spotify/HubFramework.

[7] A. Haskett, "Server-driven ui: What airbnb, netflix, and lyft learned building dynamic mobile experiences," blog post, Dec. 2025. [Online]. Available: https://medium.com/@aubreyhaskett/server-driven-ui-what-airbnb-netflix-and-lyft-learned-building-dynamic-mobile-experiences-20e346265305.

[8] A. Shatnawi, A. Bahri, B. T. Niang, and B. Verhaeghe, "Enhancing data serialization efficiency in REST services: Migrating from JSON to protocol buffers," in *Proceedings of the 20th International Conference on Software Technologies (ICSOFT)*, 2025, pp. 193–200.

[9] M. Di Gregorio, D. Di Nucci, F. Palomba, and G. Vitiello, "The making of accessible android applications: an empirical study on the state of the practice," *Empirical Software Engineering*, vol. 27, no. 145, 2022.

[10] Apple Inc., "SwiftUI: Declarative app development," *Apple Developer Documentation*, 2019. [Online]. Available: https://developer.apple.com/documentation/swiftui.

[11] Google, "Jetpack Compose," Android Developers documentation, 2021. [Online]. Available: https://developer.android.com/develop/ui/compose.

[12] J. Wharton *et al.*, "Cash App Redwood: Multiplatform reactive UI," GitHub repository, Cash App, 2022. [Online]. Available: https://github.com/cashapp/redwood.

[13] A. Biørn-Hansen, T.-M. Grønli, G. Ghinea, and S. Alouneh, "An empirical study of cross-platform mobile development in industry," *Wireless Communications and Mobile Computing*, vol. 2019, Article ID 5743892, 2019.

[14] C. Curino, H. J. Moon, A. Tanca, and C. Zaniolo, "Schema evolution in Wikipedia: Toward a web information system benchmark," in *Proc. Int. Conf. on Enterprise Information Systems (ICEIS)*, 2008, pp. 323–332.

[15] E. Rahm and P. A. Bernstein, "An online bibliography on schema evolution," *ACM SIGMOD Record*, vol. 35, no. 4, pp. 30–31, 2006.

[16] J. Ebertz, "Building invincible apis: How to avoid future breaking changes in your apis," blog post, Aug. 2025. [Online]. Available: https://jkebertz.medium.com/building-invincible-apis-462c1e5e387c.

[17] A. Lauret, "Handling breaking ch-ch-changes," blog post, Jun. 2021. [Online]. Available: https://apihandyman.io/handling-breaking-ch-ch-changes.

[18] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner, "Exploring differences and commonalities between feature flags and configuration options," in *Proc. Int. Conf. on Software Engineering, Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 233–242.

[19] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019. DOI: 10.1016/j.infsof.2018.09.006.

[20] "Yandex DivKit: An open framework for server-driven UI," GitHub repository, 2022. [Online]. Available: https://github.com/divkit/divkit.

[21] "Airbnb Lona: A tool for defining design systems," GitHub repository, 2017. [Online]. Available: https://github.com/Lona/Lona.

[22] "EnsembleUI: Build native apps with JSON," GitHub repository, 2022. [Online]. Available: https://github.com/EnsembleUI/ensemble.

[23] B. Lei, B. Dupree, and Q. Nguyen, "Architecting a safe, scalable, and server-driven platform for driver preferences with ribs," *Uber Engineering Blog*, Mar. 2019. [Online]. Available: https://www.uber.com/blog/carbon-driver-app-preferences-ribs/.

[24] R. Brooks, "A deep dive into airbnb's server-driven ui system," *Airbnb Engineering Blog*, Jun. 2021. [Online]. Available: https://medium.com/airbnb-engineering/a-deep-dive-into-airbnbs-server-driven-ui-system-842244c5f5.

[25] O. Demiröz, "Server-driven ui development — future of dynamic applications," blog post, Jun. 2025. [Online]. Available: https://medium.com/@osmandemiroz/server-driven-ui-development-future-of-dynamic-applications-4f9fe8c5b4ec.

[26] Coding with Tech, "gRPC seemed perfect. Our frontend team hated it," blog post, Jan. 2026. [Online]. Available: https://javascript.plainenglish.io/grpc-seemed-perfect-our-frontend-team-hated-it-d3ca2e385152.

[27] K. Brahmbhatt, "Generative ui: The ai-powered future of user interfaces," blog post, Mar. 2025. [Online]. Available: https://medium.com/@knbrahmbhatt_4883/generative-ui-the-ai-powered-future-of-user-interfaces-920074f32f33.

[28] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A unifying reference framework for multi-target user

interfaces," *Interacting with Computers*, vol. 15, no. 3, pp. 289–308, 2003.

[29] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[30] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[31] M. Bolanowski, A. Paszkiewicz *et al.*, "Efficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems," in *New Trends in Intelligent Software Methodologies, Tools and Techniques (Proc. SOMET 2022)*, ser. Frontiers in Artificial Intelligence and Applications, vol. 355. IOS Press, 2022. [Online]. Available: https://arxiv.org/abs/2208.00682.

[32] W. Meijer, C. Trubiani, and A. Aleti, "Experimental evaluation of architectural software performance design patterns in microservices," *Journal of Systems and Software*, vol. 218, art. 112183, 2024. DOI: 10.1016/j.jss.2024.112183.

[33] J. Wu, E. Schoop, A. Leung, T. Barik, J. P. Bigham, and J. Nichols, "UICoder: Finetuning large language models to generate user interface code through automated feedback," in *Proc. 2024 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2024, pp. 7511–7525. DOI: 10.18653/v1/2024.naacl-long.417.