software

MDPI

Article

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, Mountain View, CA 94043, USA; mahlawat@google.com (M.A.); butterdidit@google.com (P.M.); singhsusmita@google.com (S.S.); anamikamodi@google.com (A.M.)
* Correspondence: vipinsi@google.com

Abstract

Config-driven user interface (CDUI), also called server-driven user interface, moves user interface (UI) layout out of the app binary into server responses so teams can update screens without an app store release. Because one schema serves many installed app versions, edits to it can break older apps. We statically compared parsed schema declarations across four open-source runtime CDUI engines (Yandex DivKit, Cash App Redwood, EnsembleUI, and archived Spotify HubFramework) and one inactive comparator compiled at build time (Airbnb Lona). Out of 1447 commits touching schema paths, 730 modify a runtime engine's schema: 61.4% only add declarations, 38.1% remove or retype declarations, and 0.5% mark deprecations (93% parser accuracy against two-rater human labels). Standalone schema files had a higher share of additive commits than schemas written in code, though across four runtime engines this difference is not statistically significant. By Fisher's exact test across commit halves, breaking edits fell significantly only in Redwood (73.3% to 25.0%, $p = 0.012$). Finally, we compare systems across modularity, centralization, and strictness, showing when additions still break older clients (including Redwood throwing by default).

Keywords: server-driven user interface; config-driven user interface; schema evolution; schema governance; backward compatibility; software architecture; mobile application development

1. Introduction

Mobile teams want to ship changes quickly, run live user interface (UI) experiments, and keep Android and iOS in step [1,2]. Config-driven UI (CDUI), also called server-driven UI (SDUI), serves those goals by moving the layout out of the app binary [2]. The server sends a description of the screen as data, usually JavaScript Object Notation (JSON) or protocol buffers (Protobuf), and the app renders whatever it receives, so a team can change a screen without waiting for an app store review [3]. That is a move away from model–view–controller (MVC) [4], where the screen is written into the app and compiled with it, toward architectures in which screen layout and state are split between the server and the mobile app [5].

The approach is widely used in industry but has received little attention in research. Most of what is known about it sits in engineering reports, conference talks, and framework repositories rather than in peer-reviewed work [6–8]. The main cost is "Contract Fragility", the risk that a change to the schema breaks an app that is already installed. A server response can be valid under the current schema and still contain components or properties that an older app does not recognize, and that app may then fail to render the screen or

Check for updates

Academic Editor: Paulo Ferreira

Received: 28 September 2026
Revised: 3 November 2026
Accepted: 10 November 2026
Published: 15 November 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

crash. Teams also have to weigh how efficiently the payload is encoded [9] against what the client needs for accessibility and native rendering [10,11].

This paper brings the two together: a review of what industry has published about CDUI, which organizes the concepts and the history, and an analysis of the schema commit histories of four open-source runtime CDUI engines and one design tool comparator compiled at build time. To build that commit corpus, we pinned each repository at a fixed revision, collected every commit from 2015 to 2026 touching its schema files, and parsed each file before and after the commit so edits to comments, translations, or internal code were excluded.

1.1. Research Questions

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

- **RQ1 (Structure, source review):** What parts make up a CDUI system, and what distinguishes one framework from another?
- **RQ2 (History, source review):** How has the split of logic between server and client changed since 2010?
- **RQ3 (Schema change, commit analysis):** How do the schemas of open-source runtime CDUI engines change over time, and how often do those changes break older clients?
- **RQ4 (Consequences, source review):** What costs and side effects does CDUI bring for performance, accessibility, and maintenance?

1.2. Contributions

The main contributions of this paper are as follows:

1. A **descriptive framework** for comparing CDUI systems along three axes, modularity, centralization, and strictness, drawn from published engineering sources, framework documentation, and client source code (Section 4).
2. A **commit and individual change analysis** of 1447 commits touching schema paths (943 commits that modified the schema and 6866 individual schema changes, 2015 to 2026) across four open-source runtime CDUI engines and one design tool comparator compiled at build time, measuring how often changes add to, retype, or remove schema declarations (Section 6).
3. An **exploratory observation** that schemas kept in separate data files average 65.3% additive commits across the four runtime engines, compared with 53.8% for schemas written inside source code (Section 6).
4. A **review of the side effects** of CDUI: how clients cope with components they do not recognize, growth of the backend-for-frontend (BFF) layer, and how schemas carry accessibility information (Sections 6.3 and 7).

2. Related Work

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

2.1. Traditional Graphical User Interface Architectures

Research on graphical user interfaces (GUIs) has long described patterns such as MVC [4] and model–view–viewmodel (MVVM). Both assume that the layout is written into the app and compiled with it. CDUI breaks that assumption. The server decides the layout and sends it while the app is running, so the app becomes a renderer for whatever it receives rather than a program that holds its own screen logic.

https://doi.org/10.3390/software5040052

2.2. Declarative UI Frameworks

Declarative UI toolkits such as Apple's SwiftUI [12] and Google's Jetpack Compose [13] share CDUI's aim of describing what a screen should contain rather than issuing drawing commands. The difference is that these toolkits compile the description into the app, so changing a layout still needs a release, while CDUI sends the description from the server to an app that is already installed. Cash App's Redwood shows the two approaches converging: it keeps the Compose programming model, but sends the widget tree across a protocol boundary so that a screen can change without a release [14].

2.3. Cross-Platform Mobile Frameworks

Cross-platform frameworks such as Flutter and React Native [5,15] let one codebase run on both platforms, which saves building each screen twice. What they do not change is where the screen lives: it is still compiled into the app, so every layout change waits for a release. CDUI targets that delay rather than the duplicated code, and it can sit on top of native code, Flutter, or React Native. Several frameworks combine both, letting Flutter or React Native draw the layouts the server sends [16].

2.4. Micro Frontends and Web Composition

Micro frontends let separate teams deploy parts of a web page on their own schedules [17], which works because a browser will run whatever HyperText Markup Language (HTML) and JavaScript it is sent. A mobile app cannot do that freely, since app store distribution guidelines restrict what downloaded executable code may do [3]. CDUI therefore sends data rather than arbitrary native binaries, and that data can only refer to widgets already compiled into the app.

2.5. Schema Evolution

Schema evolution has long been studied for databases and semi-structured data. Curino et al. [18] analyzed how the Wikipedia database schema changed over its history, and Rahm and Bernstein [19] maintain a bibliography of schema evolution research. Qiu et al. [20] studied how database schemas and the application code that uses them evolve together, Scherzinger and Sidortschuck [21] extracted the implicit schemas of ten Java applications built on schema-free NoSQL data stores and traced how those schemas changed across project histories, and Attouche et al. [22] formalized the validation semantics of modern JSON Schema (including combinators such as `allOf` and `oneOf` and annotation keywords). Our study asks how a schema changes over time for a different kind of contract: a UI schema consumed by many deployed mobile client versions at once.

2.6. Application Programming Interface Evolution and Breaking Changes

A large body of software engineering research measures how application programming interfaces (APIs) change and how often they break their clients. Dig and Johnson [23] studied API changes in four frameworks and one library and found that over 80% of the changes that broke applications were refactorings. Raemaekers et al. [24] analyzed more than 100,000 library releases in Maven Central and found that about one third of releases introduced at least one breaking change, and that deprecation tags were often applied improperly. Brito et al. [25] asked the developers of popular Java libraries why they introduced breaking changes and found that most were motivated by new features, simpler APIs, and maintainability. Bogart et al. [26] showed that the Eclipse, CRAN, and npm ecosystems handle breaking changes very differently, largely because of different community values. In mobile development, McDonnell et al. [27] found that the Android API

changed rapidly while client applications lagged behind, with about 28% of API references in client applications outdated.

Web and microservice APIs are closer to CDUI, because the server controls the contract. Li et al. [28] analyzed the evolution of five popular web APIs, identified 16 change patterns, and noted that clients of a web API, unlike clients of a local library, often cannot keep using an old version. Schmiedmayer et al. [29] proposed an automated process that generates client libraries, for example for mobile applications, from migration guides, and evaluated it on 13 web service version increments containing 3896 changes. Recent empirical studies of microservice and web APIs likewise show how teams version endpoints and request headers and manage consumers locked onto older API versions [30,31].

CDUI differs from these settings because the schema describes the UI itself. A breaking change in a CDUI schema can make part of a screen unusable on older app versions, rather than causing a single failed request.

CDUI is also worth separating from feature flags and remote toggle services (such as LaunchDarkly or Firebase Remote Config). Empirical studies show that developers use feature flags and configuration options to switch between code paths that are already compiled into an application [32,33]. A CDUI payload instead carries the structure of the screen, and the client builds whatever it describes. A flag picks one of a fixed set of variants, while CDUI can compose new ones from the components the client already knows.

2.7. Model-Based User Interface Development

Model-based user interface development (MBUID) is the closest earlier line of research. Calvary et al.'s CAMELEON framework [34] generates interfaces for different platforms from a model, working down four levels: the user's task, an abstract interface, a concrete interface, and the final one. CDUI keeps the idea of separating what a screen should contain from how a platform draws it, but it does the work at a different moment. MBUID generated platform code before the app shipped, while CDUI sends the description to an app that is already installed, which is what lets a screen change without a new release.

We are not aware of any prior peer-reviewed study that systematically analyzes CDUI schema evolution across multiple open-source runtime CDUI engines (alongside a design tool comparator compiled at build time). This paper addresses that gap by combining a review of published industry sources with a quantitative analysis of schema commits.

3. Methodology

This study draws on three kinds of evidence: published engineering sources, the source code of the studied frameworks, and the commit histories of those frameworks. The sources and the code support RQ1, RQ2, and RQ4. The commit histories answer RQ3.

3.1. Review of Practitioner and Research Sources

Because CDUI practice is led by industry, much of what is known about it is published outside peer-reviewed venues. To describe how these systems are built and how they got there (RQ1, RQ2, RQ4), we reviewed publicly available sources: engineering blog posts and talks from organizations that operate CDUI systems, framework documentation and source repositories, and peer-reviewed work on related topics. Through October 2026, we found sources through keyword searches (using terms such as "Server-Driven UI", "Config-Driven UI", "backend-driven UI", "JSON Schema evolution", and "API breaking changes") in Google Scholar, IEEE Xplore, the ACM Digital Library, GitHub, and general web search, and by following references between sources. We gave priority to sources that describe architectural decisions, the split of logic between client and server, or contract versioning in native mobile apps, and set aside marketing material and introductory tutorials.

The review organizes what industry has published as context for the commit analysis. Every claim in Sections 4, 5, and 7 cites the source behind it, so a reader can check each one. Proprietary frameworks whose code is private, such as Zalando's Appcraft and Airbnb's Ghost Platform, are described from their published documentation.

3.2. Commit Analysis Setup

To select repositories to analyze, we searched GitHub (`topic:server-driven-ui`, `topic:sdui`, "server-driven UI", "config-driven UI", and "json-to-ui") and engineering articles through October 2026. After setting aside tutorials, sample apps, and small personal projects, we screened 15 established open-source UI and CDUI repositories (Table 1). To qualify as a runtime CDUI engine, a repository had to: (1) render UI payloads sent by a server at runtime on mobile apps; (2) keep schema or component model files separate from general app code; (3) have at least 100 commits across multiple years; and (4) provide public git history under an open-source license.

Table 1. Open-source candidate repositories screened ($N = 15$) and selection decisions.

Candidate Repository	Platform and Schema Format	Decision	Selection Rationale
`divkit/divkit`	Android, iOS, Web (JSON Schema)	**Included (Runtime CDUI)**	Active runtime engine (Yandex) with standalone `schema/` directory.
`cashapp/redwood`	Kotlin Multiplatform (KMP), Android, iOS, Web	**Included (Runtime CDUI)**	Active runtime protocol engine (Cash App) with dedicated `@Schema` modules.
`EnsembleUI/ensemble`	Flutter, iOS, Android (JSON Schema)	**Included (Runtime CDUI)**	Active runtime engine with standalone JSON Schema in `assets/schema/`.
`spotify/HubFramework`	iOS (Objective-C headers and JSON)	**Included (Runtime CDUI)**	Archived in 2018 (Spotify); complete lifecycle with public schema headers.
`Lona/Lona`	iOS, macOS, Web (`.component` JSON)	**Included (Comparator)**	Inactive since 2020 (Airbnb); design tool compiled at build time, kept for comparison.
`zup-archive/beagle-*`	Android, iOS, Web (Kotlin/Swift/TypeScript)	Excluded	Discontinued and split across archived platform repos without one shared schema history.
`flutter/packages` (`rfw`)	Flutter (text/binary widget data)	Excluded	Subpackage without separate schema files (`lib/src/flutter` widget definitions have 36 commits).
`stacdev/stac` (Mirai)	Flutter (Dart JSON serializers)	Excluded	Recent library (created July 2024) using Dart model classes without separate schema files.
`flipkart-incubator/proteus`	Android (JSON `LayoutInflater`)	Excluded	Generic Android `View` inflation library without a versioned component catalog.
`alibaba/Tangram-iOS`	iOS (Objective-C layout engine)	Excluded	Archived imperative virtual view layout library without declarative schema files.
`apache/incubator-weex`	Android, iOS (JavaScript bundle runtime)	Excluded	Archived runtime that executes compiled JavaScript bundles rather than UI schemas.

Table 1. *Cont.*

Candidate Repository	Platform and Schema Format	Decision	Selection Rationale
`phoenixframework/phoenix_live_view`	Web (server HTML diffs)	Excluded	Sends server HTML diffs over WebSockets rather than mobile UI schemas.
`hotwired/turbo`	Web, mobile WebView (HTML)	Excluded	Sends server HTML fragments to WebViews rather than native UI schemas.
`airbnb/epoxy`	Android (client view library)	Excluded	Client `RecyclerView` library; Airbnb's Ghost CDUI schema is private.
`uber/RIBs`	Android, iOS (client routing library)	Excluded	Client routing architecture; Uber's server UI schemas are private.

Bold entries in the Decision column indicate repositories included in the study; * denotes a wildcard matching platform-specific repository suffixes.

Selected Projects and Role of Lona: Four repositories met all four criteria: three active engines, Yandex DivKit [35], Cash App Redwood [14], and EnsembleUI [16], and one archived engine, Spotify HubFramework (archived in 2018) [6]. We also include Airbnb Lona (inactive since 2020) [36] as a separate comparator. Lona defines UI components and parameters in `.component` JSON files, but compiles them into static Swift, Kotlin, and React code at build time instead of sending payloads to installed apps. A breaking edit in Lona therefore requires regenerating code at build time rather than risking a failure on older apps. Throughout the paper, we report totals for the four runtime engines separately from totals across all five projects and exclude Lona from claims about live client behavior.

Why Commit Histories: We analyze full commit histories (2015 to 2026) rather than current snapshots because backward compatibility is a property of change. A snapshot shows which components and properties exist today but not how often existing declarations were removed or retyped. Full histories also cover the complete lifecycles of HubFramework and Lona alongside ongoing changes in the three active engines.

Analyzed File Paths: Table 2 lists the pinned commit hash, schema paths, and file extensions analyzed for each repository. For Redwood, we include only the UI widget and layout schema modules (plus two earlier folder names before renames) and exclude host runtime and tooling modules (`redwood-widget/`, `redwood-protocol-host/`, `redwood-tooling-schema/`, and `redwood-schema/`). For HubFramework, we include only the JSON schema and UI data model headers (in `include/` before February 2016 and `include/HubFramework/` after) plus the JSON key mappings in `sources/HUBJSONKeys.h` and `sources/HUB*JSONSchemaImplementation.m`. UIKit controllers and general `sources/HUB*Implementation.m` files are excluded. Section 6 also compares these strict paths against the broader Redwood and HubFramework software development kit (SDK) paths.

Detecting Changes in Each Format: We collected all 1447 commits from 2015 to 2026 touching the files in Table 2. Instead of filtering commits by message keywords, we parsed every touched file before and after each commit across all five repositories. Commits that changed only non-schema content (such as `translations.json`, descriptions, comments, imports, or file moves) were recorded separately as unchanged (504 commits), leaving 943 commits that changed the schema. Checking our earlier message filter showed that skipping "merge pull request" or "fix typo" messages had dropped three breaking commits in EnsembleUI and Lona while leaving parsed DivKit shares unchanged. Each format was parsed with rules built for its structure:

- **DivKit and EnsembleUI (JSON Schema):** Parse each file as JSON before and after the commit, ignoring documentation and tooling fields (`translations.json`, `description`, `$description`, `platforms`, `examples`, and `$comment`). Compare component definitions, nested `properties`, `$defs/definitions`, composition rules

(`allOf`/`anyOf`/`oneOf` and `if`/`then`/`else`), types (`type` and `$ref`), `enum` values, `required` lists, and deprecation flags.

- **Lona (`.component` JSON):** Parse each `.component` file before and after the commit, ignoring preview fields (`examples`, `devices`, and `canvases`), to compare `params` (name, type, and required status), custom `types`, `root` view layers, and `logic` blocks.
- **Redwood (Kotlin):** Exclude test files (`*Test.kt`), strip comments and `import` lines, and compare `@Widget`, `@Modifier`, `@Property`, `@Children`, `@Default`, and `@Deprecated` declarations by tag, name, Kotlin type, and default status.
- **HubFramework (Objective-C):** Strip comments, `#import` lines, forward declarations, and nullability macros, and compare `@protocol` and `@interface` blocks, `@property` and method signatures, `NS_ENUM`/`typedef` definitions, deprecation markers, and the JSON keys in `HUBJSONKeys.h` and `HUB*JSONSchemaImplementation.m`.

Table 2. Pinned repository revisions and schema file paths analyzed per framework. Paths are git pathspecs relative to the repository root, directories include their subdirectories, and git rename detection (`-M`) tracks files across directory moves.

Framework	Pinned Commit	Schema Paths/Patterns	Files
DivKit	`70273edb`	`schema/`	`.json`
Redwood	`5c49a0bc`	`redwood-layout-schema/`, `redwood-lazylayout-schema/`, `redwood-ui-basic-schema/`, `redwood-ui-core-schema/` (before renames: `redwood-treehouse-lazylayout-schema/`, `redwood-basic-schema/`)	`.kt`
Lona	`cb948109`	`*.component`	`.component`
EnsembleUI	`550a7608`	`modules/ensemble/assets/schema/` (before the monorepo move: `assets/schema/`)	`.json`
HubFramework	`4a589c15`	`include/HUB*JSON*.h`, `include/HUBSerializable.h`, `include/HUB*Model*.h`, `include/HUBComponentImageData*.h`, `include/HUBComponentTarget*.h`, `include/HUB*Identifier.h`, `include/HUBComponentCategories.h`, `include/HUBComponentType.h`, `include/HUBIcon.h` (and matching `include/HubFramework/` paths after the Feb 2016 move), `sources/HUBJSONKeys.h`, `sources/HUB*JSONSchemaImplementation.m`	`.h`, `.m`

`*` denotes a filename wildcard in git pathspecs.

Across all formats, adding a schema file counts as a node addition and deleting one counts as a removal. We removed earlier line-count fallback rules after checking all 97 commits they matched: 88 (90.7%; 81 in HubFramework, 15 in Redwood, and 1 in EnsembleUI) did not change the schema, and the other 9 are now caught directly. Reparsing the 392 commits previously left "Uncategorized" showed that 329 (83.9%) left the schema unchanged and 63 (16.1%) changed it (30 property additions, 23 type mutations, 7 removals, and 3 node additions), leaving zero uncategorized commits.

Change Definitions: All labels come from comparing parsed schema declarations before and after each commit rather than running payloads against older compiled apps. Because adding to a schema is safe at runtime only when older clients tolerate unknown elements (Section 6.3), we separate static schema edits from client runtime behavior and group changes into three classes:

- **Additive:** a *node addition* (a new component or widget) or a *property addition* (a new optional property, enum value, or relaxed `required` rule on an existing definition).
- **Breaking:** a *field or node removal* (deleting an existing component, property, method, or enum value, or making an optional property `required`) or a *type mutation* (changing the declared type, `$ref` target, or tag of an existing property or method).
- **Deprecation:** marking an existing declaration deprecated without removing or retyping it, reported separately as neither additive nor breaking.

Counting Commits and Individual Changes: Because a single commit can bundle many edits, we count changes both ways. At the *commit level*, each schema commit gets one label using a "most destructive change" order: removal, type mutation, deprecation, node addition, then property addition. At the *individual change level*, we count every added, removed, retyped, or deprecated declaration inside each commit and also report counts per 100 changed diff lines.

Classifier Validation: To test parser accuracy across all six categories, we drew a stratified random sample of 100 commits (20 per repository, random seed `20261023`). We oversampled rare and breaking categories so every repository and category pair in the 1447 commits appeared, including all 4 deprecations and 37 commits labeled breaking by the parser. Two authors (V.S. and P.M.) independently labeled each commit on GitHub within the Table 2 paths without seeing the parser output or each other's labels, marking comment, description, translation, or Swift macro edits as unchanged. Section 6.2 reports rater agreement, Cohen's κ (a 0 to 1 score for agreement beyond chance), sample accuracy against agreed human labels, and accuracy weighted back to all 1447 commits.

Metrics Extracted: For each repository we report the shares of schema commits and individual changes that are additive, breaking, or deprecations, plus shares over all raw commits, as measures of "Contract Fragility."

4. Structure of CDUI Systems (RQ1)

The sources we reviewed describe CDUI systems as built from three parts: component libraries, client parsers, and state management engines. Rather than borrowing an existing classification, we built three comparison axes from the ground up by grouping design choices that recurred across the reviewed reports and repositories: Modularity (related to interface abstraction levels in MBUID [34]), centralization, and strictness (related to how web API clients handle API changes [28]). Table 3 places seven systems on these axes. For the five open-source projects, we assigned each label by inspecting schema files and client parser code at the pinned commits in Table 2. For the two proprietary company systems (Uber Router-Interactor-Builder (RIBs)/Preferences and Airbnb Ghost), labels come from their published engineering reports. Testing whether these three axes cover a wider set of CDUI frameworks without extra dimensions is left to future work.

Table 3. The studied frameworks placed on the three axes. Rows for the five open-source frameworks were assigned from their documentation and source code; the two rows based only on documentation rest on their organizations' published descriptions and could not be checked against code.

Framework/Organization	Modularity	Centralization	Strictness
DivKit (Yandex) [35]	Atomic and Molecular	Embedded Scripting	Defensive Rendering
Redwood (Cash App) [14]	Atomic and Molecular	Embedded Scripting (Zipline)	Strict Schema Locking (default; custom handler)
Lona (Airbnb) [36]	Atomic	N/A (Design-to-Code)	N/A (Design-to-Code)
EnsembleUI [16]	Atomic and Molecular	Embedded Scripting	Defensive (props); Strict (widgets)
HubFramework (Spotify) [6]	Molecular	Action-Mapping	Defensive Rendering
Frameworks described only in documentation (proprietary, no commit analysis):			
RIBs/Preferences (Uber) [7]	Molecular	Stateless	Strict Schema Locking
Ghost Platform (Airbnb) [8]	Molecular	Action-Mapping	Defensive Rendering

4.1. Modularity

- **Atomic:** The schema exposes basic layout and view building blocks, such as `Row`, `Column`, `Button`, and `Text`, with layout attributes such as padding and alignment. This gives the server fine control at the cost of larger payloads.
- **Molecular:** The schema sends composite product components, such as a card or list row, whose internal layout is built into native client views. Frameworks exposing

both basic layout primitives and composite widgets (such as galleries or data grids) are labeled **Atomic and Molecular**.

4.2. Centralization

- **Stateless:** The payload describes the view only. What happens on a tap is written into the app.
- **Action-Mapping:** The payload pairs each user event with a named action identifier, and the app dispatches that identifier to a native handler.
- **Embedded Scripting:** The payload carries inline expressions or guest scripts that the app evaluates at runtime in a restricted language or sandbox.

The Centralization axis describes logic carried in a runtime payload, so it does not apply to Lona: its `.component` files contain logic blocks, but these are compiled into generated client code rather than interpreted from a server response. Listing 1 contrasts Action-Mapping and Embedded Scripting payloads.

Listing 1: Action-Mapping vs. Embedded Scripting Payloads

```
{
  "action_mapping_example": {
    "type": "button",
    "text": "Checkout",
    "on_click": {
      "action": "NAVIGATE",
      "target": "/cart"
    }
  },
  "embedded_scripting_example": {
    "type": "text",
    "text": "@{cart_total > 0 ? 'Checkout' : 'Empty'}",
    "visibility": "@{is_logged_in}"
  }
}
```

4.3. Strictness

We code strictness by checking how a runtime client parser handles both unknown properties on a known widget and unknown widget types (Lona is marked `N/A` because its components are compiled at build time):

- **Strict Schema Locking:** The client parser rejects the screen or throws an error when an unrecognized widget or schema element arrives (unless overridden by a custom host handler, as in Redwood). Some engines split the two levels: EnsembleUI uses defensive rendering for unknown properties on known widgets and strict schema locking for unknown widgets (Section 6.3).
- **Defensive Rendering:** Unknown optional properties are ignored, and unrecognized or invalid component nodes are omitted or replaced by a fallback view instead of aborting the screen.

5. How CDUI Developed (RQ2)

From the cited sources, we group the development of CDUI into three phases. The boundaries are approximate and reflect when each system was publicly described.

- **Phase 1: Remote Config (2010 to 2014)**
  Apps fetched flat key–value flags from a server to switch compiled features on or off or to override fixed strings [33,37], while screen assembly stayed compiled into native client code or wrapped WebViews [38].

- **Phase 2: Component-Driven Orchestration (2015 to 2019)**
  Organizations began building their own dynamic runtimes.
  - *Spotify Hub Framework (2016):* Spotify built a toolkit for consistent native iOS screens that separated the data model from the UI state [6].
  - *Uber's RIBs Architecture:* Uber's 2018 "Carbon" rewrite introduced the RIBs architecture on the client. By 2019 that structure let Uber move its Driver Preferences screens to configurations sent by the backend [7].
  - *Airbnb's Two Efforts:* Airbnb ran two separate efforts: Lona, which generates code from designs, and the Ghost Platform, a runtime for search and checkout screens [8].
- **Phase 3: Reactive and Contract-Driven CDUI (2020 to 2026)**
  Some organizations moved from loosely structured JSON payloads toward strongly typed contracts and binary protocols. For example, Faire described moving to an SDUI framework built on Protobuf, reporting a 90% cut in rendering logic and a 65% cut in overall lines of code [2].

6. How CDUI Schemas Change (RQ3)

6.1. Supporting Older App Versions (N-k)

Mobile apps update slowly, so a server usually has to serve clients that are several versions behind. We call this $N - k$ support: the current schema is version N, while some installed apps were built against version $N - k$. Figure 1 shows one way to handle it. The client sends its version in a request header, and the server compares that version with the minimum each schema component requires before deciding what to send [30,31].

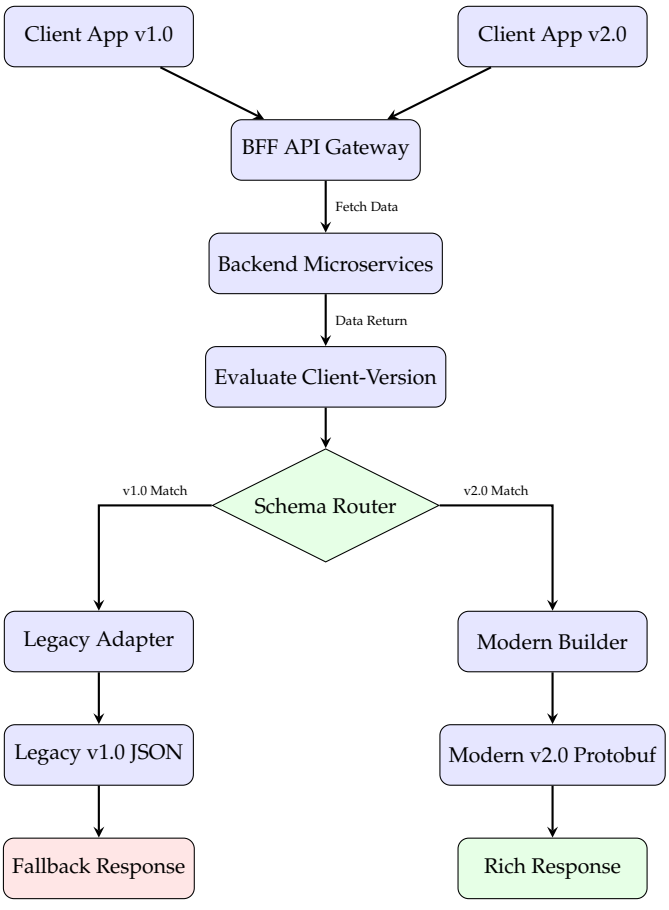


Figure 1. Architectural diagram of a CDUI BFF handling $N - k$ versioning via `Client-Version` header routing.

6.2. Results

Across the five repositories, 1447 raw commits touched the schema paths in Table 2. These include 1224 commits in the four runtime CDUI engines (DivKit 570 from 2022 to 2026, Redwood 58 from 2022 to 2025, EnsembleUI 450 from 2022 to 2024, and HubFramework 146 from 2015 to 2018) and 223 in the build-time Lona comparator (2017 to 2020). Parsing each file before and after every commit separated the dataset into 943 commits that changed the schema (730 in the four runtime engines and 213 in Lona) and 504 commits that left the schema unchanged (494 runtime and 10 Lona), with zero uncategorized commits. Most unchanged commits came from DivKit (391 of 504): 94 edited only `translations.json`, 26 changed comments, metadata, or file names without altering fields, and 271 updated the `platforms` list when an existing property rolled out to another client platform. Counting those 271 platform rollouts as additive SDK changes would raise DivKit's additive commit share from 71.5% to 88.7%.

Commit and Individual Change Counts: Table 4 gives one label per schema commit based on its most destructive edit, along with percentages out of all raw commits. Because 39.9% of runtime schema commits (291 of 730) bundle more than one kind of edit, Table 5 also counts all 5216 individual field and component changes in the four runtime engines (6866 including Lona) and reports rates per 100 changed schema lines.

Every repository has more additive commits than breaking commits: 71.5% additive in DivKit, 59.1% in EnsembleUI, 56.0% in HubFramework, 51.6% in Redwood, and 70.9% in Lona. Across the 730 runtime schema commits, 61.4% only add declarations, 38.1% remove or retype declarations, and 0.5% mark deprecations (63.5%, 36.1%, and 0.4% across all 943 schema commits when Lona is included). Counting each individual schema change in Table 5 raises the additive share to 70.3% across the four runtime engines (73.1% across all five projects), with every engine above 55% additive (55.1% to 72.8%, including 59.0% in Redwood). Dividing by diff size also explains why EnsembleUI has the largest raw change count: its single shared schema file averages 20.9 schema changes (5.7 breaking) per 100 changed lines, compared with 9.0 to 14.5 changes (3.6 to 5.9 breaking) across HubFramework, DivKit, and Redwood, and 5.1 (0.9 breaking) in Lona.

Table 4. Schema modifications per commit across 1447 raw commits (2015 to 2026). Primary percentages use each repository's schema commits (n) as the denominator, so additive, breaking, and deprecations sum to 100%. Bottom rows report verified commits with no schema change and percentages out of all raw commits (N).

Modification Type	DivKit ($n = 179$)	Redwood ($n = 31$)	EnsembleUI ($n = 411$)	HubFramework ($n = 109$)	4 Engines Total ($n = 730$)	Lona (Build Comp., $n = 213$)	All 5 ($n = 943$)
Node Additions	47 (26.3%)	10 (32.3%)	67 (16.3%)	18 (16.5%)	142 (19.5%)	77 (36.2%)	219 (23.2%)
Property Additions	81 (45.3%)	6 (19.4%)	176 (42.8%)	43 (39.4%)	306 (41.9%)	74 (34.7%)	380 (40.3%)
Total Additive	**128 (71.5%)**	**16 (51.6%)**	**243 (59.1%)**	**61 (56.0%)**	**448 (61.4%)**	**151 (70.9%)**	**599 (63.5%)**
Type Mutations	21 (11.7%)	8 (25.8%)	45 (10.9%)	21 (19.3%)	95 (13.0%)	3 (1.4%)	98 (10.4%)
Field/Node Removals	27 (15.1%)	7 (22.6%)	122 (29.7%)	27 (24.8%)	183 (25.1%)	59 (27.7%)	242 (25.7%)
Total Breaking	**48 (26.8%)**	**15 (48.4%)**	**167 (40.6%)**	**48 (44.0%)**	**278 (38.1%)**	**62 (29.1%)**	**340 (36.1%)**
Deprecations (neither)	3 (1.7%)	0 (0.0%)	1 (0.2%)	0 (0.0%)	4 (0.5%)	0 (0.0%)	4 (0.4%)
Raw Commits (N)	570	58	450	146	1224	223	1447
Unchanged (% of N)	391 (68.6%)	27 (46.6%)	39 (8.7%)	37 (25.3%)	494 (40.4%)	10 (4.5%)	504 (34.8%)
Additive % of All Raw (N)	22.5%	27.6%	54.0%	41.8%	36.6%	67.7%	41.4%
Breaking % of All Raw (N)	8.4%	25.9%	37.1%	32.9%	22.7%	27.8%	23.5%

Bold rows indicate category subtotals across additive and breaking schema modifications.

Table 5. Individual schema changes across all schema commits ($n = 5216$ runtime CDUI changes; $n = 6866$ across all five projects) and rates per 100 changed schema lines.

Change Type	DivKit ($n = 604$)	Redwood ($n = 100$)	EnsembleUI ($n = 4169$)	HubFramework ($n = 343$)	4 Engines Total ($n = 5216$)	Lona (Build Comp., $n = 1650$)	All 5 ($n = 6866$)
Node Additions	111 (18.4%)	30 (30.0%)	202 (4.8%)	28 (8.2%)	371 (7.1%)	133 (8.1%)	504 (7.3%)
Property Additions	274 (45.4%)	29 (29.0%)	2833 (68.0%)	161 (46.9%)	3297 (63.2%)	1220 (73.9%)	4517 (65.8%)
Total Additive Events	**385 (63.7%)**	**59 (59.0%)**	**3035 (72.8%)**	**189 (55.1%)**	**3668 (70.3%)**	**1353 (82.0%)**	**5021 (73.1%)**
Type Mutations	86 (14.2%)	27 (27.0%)	584 (14.0%)	61 (17.8%)	758 (14.5%)	6 (0.4%)	764 (11.1%)
Field/Node Removals	124 (20.5%)	14 (14.0%)	549 (13.2%)	93 (27.1%)	780 (15.0%)	291 (17.6%)	1071 (15.6%)
Total Breaking Events	**210 (34.8%)**	**41 (41.0%)**	**1133 (27.2%)**	**154 (44.9%)**	**1538 (29.5%)**	**297 (18.0%)**	**1835 (26.7%)**
Deprecations	9 (1.5%)	0 (0.0%)	1 (<0.1%)	0 (0.0%)	10 (0.2%)	0 (0.0%)	10 (0.1%)
Bundled Commits	48 (26.8%)	9 (29.0%)	205 (49.9%)	29 (26.6%)	291 (39.9%)	62 (29.1%)	353 (37.4%)
Changes per 100 Diff Lines	10.3	14.5	20.9	9.0	17.2	5.1	10.9
Breaking per 100 Diff Lines	3.6	5.9	5.7	4.1	5.1	0.9	2.9

Bold rows indicate category subtotals across additive and breaking schema events.

Required vs. Optional Removals, Enum Additions, and Broader SDK Paths: We also checked three edge cases across the four runtime engines. First, of the 780 individual removals, 74 delete whole components, files, or methods, 69 remove required fields or enum values, 71 turn an optional field into `required`, and 566 remove optional fields. At the commit level, 76 of the 183 removal commits delete only optional fields; even if every optional-field deletion were harmless to older clients, 27.7% of runtime schema commits (202 of 730) still contain breaking type changes, deleted components, or required-field edits. Second, 79 property additions (17 in DivKit and 62 in EnsembleUI) add a new value to an existing `enum` list. Treating enum additions as breaking for strict client parsers moves 21 commits from additive to breaking, shifting the runtime split to 58.5% additive and 41.0% breaking. Third, expanding Redwood and HubFramework beyond Table 2 to their broader SDK folders (adding `redwood-widget/`, `redwood-protocol-host/`, `redwood-tooling-schema/`, and all public Objective-C headers in `include/HubFramework/`) lowers additive commit shares to 43.0% in Redwood (43 of 100; 58.6% of individual changes) and 44.5% in HubFramework (126 of 283; 55.2% of individual changes), showing that internal host and controller code changes more often than the UI schema files themselves.

- **Schema Format Comparison:** Among the four runtime engines, both projects using standalone schema files had a higher share of additive commits than both projects writing schemas in code: 71.5% in DivKit and 59.1% in EnsembleUI versus 56.0% in HubFramework and 51.6% in Redwood (Lona recorded 70.9%, though with lower parser accuracy; Section 8.4). Averaging across the runtime repositories gives 65.3% additive commits for data files versus 53.8% for schemas in code, an 11.5 percentage point difference. Combining commits directly gives 62.9% versus 55.0% (71.7% vs. 56.0% across all 5216 runtime changes, and 67.2% vs. 53.8% by repository average or 65.0% vs. 55.0% combined when Lona is included). However, with only two runtime engines per group, a DerSimonian-Laird random effects model (which accounts for differences between repositories) estimates the gap at +10.1 points with a 95% confidence interval crossing zero (−4.6 to +24.7, $p = 0.18$; exact two-vs.-two group split $p = 0.33$). Although the same model gives +12.5 points (+2.5 to +22.5, $p = 0.01$) across individual runtime changes and +11.9 points (+0.1 to +23.8, $p = 0.05$) when Lona is added, multiple edits inside one commit are not independent and Lona is not a runtime engine. We therefore keep the four-engine commit check ($p = 0.18$) as primary and treat this gap only as an exploratory observation (Section 8.1).
- **Changes over Time:** Table 6 compares the earlier and later halves of each project's schema commits using Fisher's exact test (checking whether a difference between two proportions could happen by chance). We also computed breaking shares by

chronological quarter and a Spearman rank correlation (ρ, a -1 to $+1$ score testing whether breaking commits trend up or down over commit order). Only Redwood showed a statistically significant drop, falling from 73.3% breaking in its first eight months to 25.0% afterward (-48.3 points, Fisher's $p = 0.012$; $\rho = -0.45$, $p = 0.012$, $n = 31$; quarters 71.4%, 75.0%, 37.5%, and 12.5%). Even that drop becomes borderline ($p = 0.060$) after a Holm adjustment that accounts for testing five repositories at once. DivKit (28.1% to 25.6%, $p = 0.74$; $\rho = -0.01$), EnsembleUI (43.9% to 37.4%, $p = 0.19$; $\rho = -0.03$), and Lona (32.1% to 26.2%, $p = 0.37$; $\rho = -0.02$) stayed flat; in DivKit, unchanged commits grew from 160 to 231 in the second half as `platforms` lists expanded. HubFramework moved in the opposite direction during a late 2016 Swift API cleanup before archival, rising from 29.6% to 57.1% across quarters ($\rho = +0.20$, $p = 0.036$, $n = 109$; split-half 37.0% to 50.9%, Fisher's $p = 0.18$).

- **Deprecations and Tagged Releases:** Across all 780 runtime removal events (1071 including Lona), zero deleted declarations were marked `deprecated` in an earlier commit first. Conversely, all 10 declarations ever marked `deprecated` (9 in DivKit and 1 in EnsembleUI) remain in the schema at the pinned commit so older payloads still parse. To see how often breaking edits reached published versions, we grouped commits across all 211 official git release tags in the three runtime engines that tag releases: 180 on `main` in DivKit (`4.0.0` to `32.52.0`), 20 in Redwood (`0.1.0` to `0.19.0`), and 11 in HubFramework (`2.0.0` to `2.0.10`); EnsembleUI had only one `v0.1.0` tag before May 2024. Overall, 21.3% of all 211 releases (45 of 211: 18.9% in DivKit, 25.0% in Redwood, and 54.5% in HubFramework) and 40.5% of the 111 releases that touched the UI schema (45 of 111: 35.8%, 55.6%, and 85.7%) shipped at least one breaking schema change.

Table 6. Comparison of earlier and later commit halves for breaking schema shares across repositories, showing date boundaries, unchanged commit counts (n_{nc}), 95% Wilson confidence intervals (`[95% Int.]`, a standard uncertainty range for proportions that works well at small sample sizes), percentage point differences, and Fisher's exact p values.

Repository	Start	Split	End	Earlier Half			Later Half			Diff	Fisher
				Brk/ n	n_{nc}	Brk % [95% Int.]	Brk/n	n_{nc}	Brk % [95% Int.]	(pp)	p
DivKit	5 September 2022	29 May 2024	4 June 2026	25/89	160	28.1% [19.8, 38.2]	23/90	231	25.6% [17.7, 35.4]	−2.5	0.74
Redwood	6 October 2022	1 June 2023	21 August 2025	11/15	12	73.3% [48.0, 89.1]	4/16	15	25.0% [10.2, 49.5]	−48.3	0.012
EnsembleUI	16 March 2022	5 June 2023	29 May 2024	90/205	20	43.9% [37.3, 50.7]	77/206	19	37.4% [31.1, 44.2]	−6.5	0.19
HubFramework	16 December 2015	21 April 2016	22 March 2017	20/54	17	37.0% [25.4, 50.4]	28/55	20	50.9% [38.1, 63.6]	+13.9	0.18
Pooled 4 Engines	16 December 2015	n/a	4 June 2026	**146/363**	**209**	**40.2% [35.3, 45.3]**	**132/367**	**285**	**36.0% [31.2, 41.0]**	−4.3	**0.25**
Lona (Comp.)	25 November 2017	27 December 2018	11 April 2020	34/106	6	32.1% [24.0, 41.5]	28/107	4	26.2% [18.8, 35.2]	−5.9	0.37
Pooled All 5	16 December 2015	n/a	4 June 2026	**180/469**	**215**	**38.4% [34.1, 42.9]**	**160/474**	**289**	**33.8% [29.6, 38.1]**	−4.6	**0.15**

Bold rows indicate pooled totals across the four runtime engines and across all five repositories.

Classifier Validation Results: Across the 100 sampled commits, the two human raters agreed on 99 commits before discussing differences (99.0%; Cohen's $\kappa = 0.988$ across six categories and $\kappa = 1.000$ on whether the schema changed). Their single disagreement came from how a refactored `$defs` border definition was nested in EnsembleUI and was resolved by checking the full file on GitHub. Table 7 compares our automated parser against the final agreed human labels. Across all six categories, raw sample accuracy is 93.0% (93 of 100; 95% Wilson interval 86.3% to 96.6%; $\kappa = 0.914$), or 93.3% when each repository and category group is weighted back to its share of all 1447 commits. Grouping labels into four outcomes (additive, breaking, deprecation, and unchanged) gives 94.0% sample accuracy (94.8% weighted): precision and recall are 92.5% and 97.4% for additive commits ($n = 38$),

94.6% and 89.7% for breaking commits ($n = 39$), 100.0% and 100.0% for deprecations ($n = 4$), and 94.7% and 94.7% for unchanged commits ($n = 19$). Across the 80 commits from the four runtime engines, six-category accuracy is 96.3% (20 of 20 in DivKit, 20 of 20 in Redwood, 19 of 20 in HubFramework, and 18 of 20 in EnsembleUI, compared with 16 of 20 in Lona). The seven parser errors came from three specific edge cases: one Objective-C `NS_SWIFT_NAME` macro with nested parentheses in HubFramework, two `oneOf` wrapper cases in EnsembleUI (counting an `Action` branch as a widget and missing an outer `id` removal), and four Lona commits that wrapped layers in a new parent container, swapped a dropdown value one for one, or migrated early pre-`params` JSON keys.

Table 7. Confusion matrix across six categories with precision, recall, and F_1 scores comparing the automated parser (columns) against agreed human labels (rows) across the 100 sampled commits (20 per repository).

Human/Parser	Node Add	Prop Add	Deprec.	Type Mut	Field Rem	Unchanged	Support	Precision	Recall	F_1
Node Addition	**15**	0	0	0	1	0	16	88.2%	93.8%	90.9%
Property Addition	1	**21**	0	0	0	0	22	91.3%	95.5%	93.3%
Deprecation	0	0	**4**	0	0	0	4	100.0%	100.0%	100.0%
Type Mutation	0	1	0	**16**	0	1	18	100.0%	88.9%	94.1%
Field Removal	1	1	0	0	**19**	0	21	90.5%	90.5%	90.5%
Unchanged	0	0	0	0	1	**18**	19	94.7%	94.7%	94.7%
Total/Macro Average	**17**	**23**	**4**	**16**	**21**	**19**	**100**	**94.1%**	**93.9%**	**93.9%**

Bold diagonal entries indicate exact agreement between the automated parser and human labels; the bold bottom row reports column totals and macro-averaged scores.

6.3. How Clients Handle Unknown Components

Adding a component or property to a schema is backward compatible at runtime only if older installed clients tolerate elements they do not recognize. We inspected the client parser code of all four runtime CDUI engines (Lona is excluded because its `.component` files are compiled into static code at build time):

- **Yandex DivKit (Safe-Ignore and Item Drop):** DivKit's generated JSON parsers read only known property keys on a recognized component and ignore unknown optional keys. When a list item contains an unknown component type or an unrecognized `enum` value, the Android parsing helper catches the `ParsingException`, logs it to `ParsingErrorLogger`, and drops only that item so the rest of the screen still renders (`JsonParser.kt` and `JsonParserInternals.kt` in `div-data`) [35].
- **Cash App Redwood (Throws by Default unless Overridden):** Redwood's generated host code passes unknown widget, modifier, children slot, and property tags to a `ProtocolMismatchHandler` (`ProtocolMismatchHandler.kt` and `HostProtocolAdapter.kt`). By default, `ProtocolMismatchHandler.Throwing` throws an `IllegalArgumentException` on any unknown widget, modifier, children slot, or property. New schema additions in Redwood therefore break older clients by default and become safe only when the host app supplies a tolerant `ProtocolMismatchHandler` [14].
- **EnsembleUI (Ignores Unknown Properties, Throws on Unknown Widgets):** In EnsembleUI's Flutter runtime, `ScopeManager.setProperties` (`modules/ensemble/lib/framework/scope.dart`, lines 326 to 338) checks `InvokableController.getSettableProperties(invokable).contains(key)` and silently skips any unrecognized property sent by a newer schema. By contrast, when `ViewUtil.buildBareWidget` (`modules/ensemble/lib/framework/widget/view_util.dart`, lines 286 to 357) sees an unrecognized widget name, it throws `LanguageError("Unsupported Widget: ${model.type}")`. Optional property additions (176 com-

mits) are safe for older clients, whereas new widget additions (67 commits) fail unless gated by client version on the server [16].

- **Spotify HubFramework (Ignores Unknown Keys, Mandatory Fallback Component):** HubFramework components read properties by key and ignore unsupported dictionary entries (`HUBComponentModel.h`). When a payload names an unregistered component, the framework calls the host app's required `HUBComponentFallbackHandler` to render a fallback view instead of failing (`HUBComponentFallbackHandler.h`) [6].

Combining static schema changes with each engine's default client behavior shows the practical risk to deployed apps. DivKit and HubFramework fall back safely on additions, whereas all 16 additive Redwood commits (10 node and 6 property additions) throw by default and all 67 EnsembleUI widget addition commits throw unless gated by client version on the server. Adding those 83 commits to the 278 breaking schema commits means 361 of the 730 runtime schema commits (49.5%) can fail on an unconfigured older client, leaving 365 safe-by-default additive commits (50.0%) and 4 deprecations (0.5%).

7. What CDUI Costs: The "Contract Tax" (RQ4)

Moving to CDUI shifts work rather than removing it. The client does less layout work, while the payload, the server, and the people who maintain both take on more. We call this the Contract Tax.

7.1. Performance and "BFF Bloat"

Teams often move from JSON to Protobuf to cut payload size, with reported reductions of 60 to 80 percent [9]. Moving from Representational State Transfer (REST) over Hypertext Transfer Protocol (HTTP) 1.1 to gRPC Remote Procedure Calls (gRPC) over HTTP/2 also lowers network latency, through multiplexing and faster binary decoding [39]. Both changes shorten the time before a dynamic view becomes usable.

Centralizing UI composition and version adaptation in a BFF or API gateway layer [17,30] has its own maintenance cost, which we call BFF Bloat. When a backend must keep serving consumers locked onto older app versions over several years [30,40], every older version needs its own adaptation path, so nested conditionals accumulate in the routing layer and its cyclomatic complexity (the number of independent paths through the code [41]) grows. Each branch translates a newer molecular component, such as a rich carousel, into the atomic layout an older app can render, such as a plain list.

The following is an illustration rather than a measurement of production code. If each of C component types needs a separate branch for each of V supported client versions, the number of decision points in the routing layer grows roughly with $V \times C$, while an endpoint serving a single client version needs just one path. Listing 2 shows a simplified adapter for one component and four version ranges.

Listing 2: Monolithic Routing Adapter demonstrating BFF Bloat

```
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");
  }
}
```

7.2. Security: Logic Sandboxing

Logic delivered inside payloads raises the question of what that logic can reach on the device. Two of the studied frameworks limit it in different ways. DivKit payloads can contain expressions (for example `@{is_logged_in}`) that the client evaluates, and the schema marks which properties accept them through a `supports_expressions` flag [35]. Redwood's Treehouse runs guest code in a QuickJS engine embedded through Zipline, where guest and host code communicate through interfaces that the host explicitly bridges, and Zipline can verify signatures on downloaded code [42]. Both descriptions come from the frameworks' own code and documentation, and testing how well either mechanism resists attack is left to future work.

7.3. Accessibility in the Schema

Accessibility metadata can be part of the schema itself. DivKit's schema defines an `accessibility` object whose `type` property takes values such as `button`, `header`, and `image` (`schema/div-accessibility.json`) [35]. On Android, DivKit's client maps these types to native roles by installing an accessibility delegate that reports a matching class name (for example `android.widget.Button`) or marks the view as a heading (`DivAccessibilityBinder.kt`), and its iOS layout engine sets `UIAccessibilityTraits` from the element's accessibility description. Declaring a role, instead of overwriting the label a screen reader reads out, lets the screen reader announce what the element does, which supports Web Content Accessibility Guidelines (WCAG) conformance and mobile GUI accessibility guidelines [10,11].

8. Discussion and Implications

8.1. Schema Format and Additive Evolution

Across the four runtime engines, data file schemas average 11.5 percentage points more additive commits than code-based schemas (13.4 points when Lona is included, and 15.7 points when combined across all 5216 individual runtime schema changes). Because a grouped estimate across four engines has a wide 95% confidence interval spanning zero (-4.6 to $+24.7$ points), we treat this pattern as a hypothesis to test in a larger sample rather than a proven effect. Three reasons could explain why standalone schema files show fewer breaking edits. The first is **review visibility**: a standalone schema file produces a self-contained diff, so a reviewer sees the whole contract change in one place. A change buried in Kotlin annotations or Objective-C headers sits among implementation code, where its effect on the contract is easier to miss. Second, **tooling**: data files are easier to check automatically, for example with formal JSON Schema validators [22] or structural diff tools, so a continuous integration (CI) pipeline can block a breaking change before merge. Third, **separation**: keeping the schema in its own file makes the line between contract and implementation visible, which may encourage teams to treat a schema edit as a versioned API decision.

The two groups also differ in more than format. The runtime data file group (DivKit and EnsembleUI) consists of two actively maintained engines (plus the inactive comparator Lona when all five projects are combined), while the code-based group pairs one active engine (Redwood) with one archived engine (HubFramework). Age alone does not explain the runtime gap. Even if we set aside the older archived HubFramework (2015 to 2018, as well as inactive Lona from 2017 to 2020) and compare only the three active runtime engines started in 2022, a similar 13.7 point gap remains (65.3% in DivKit and EnsembleUI vs. 51.6% in Redwood).

8.2. Maturation and the "Exploration Phase"

A common expectation is that schemas settle down: teams change basic components freely at first, then stabilize the contract. In our UI schema data, only Redwood clearly shows that pattern: its breaking share dropped from 73.3% in its first eight months to 25.0% afterward ($p = 0.012$) once initial layout widgets settled. By contrast, breaking shares stayed flat across halves in DivKit (28.1% to 25.6%, $p = 0.74$), EnsembleUI (43.9% to 37.4%, $p = 0.19$), and Lona (32.1% to 26.2%, $p = 0.37$). HubFramework moved the other way during a late Swift API cleanup before archival ($\rho = +0.20$, $p = 0.036$; 37.0% to 50.9% across halves, Fisher's $p = 0.18$). Earlier internal history helps explain why DivKit and HubFramework show no early spike in public git logs: both were built and used inside Yandex and Spotify before open release, so their early design churn happened before the first public commit. Project age alone therefore does not guarantee fewer breaking edits, and we do not see the steady stabilization predicted by Lehman's laws of software evolution [43].

8.3. Comparison with Prior Work

Our findings relate to prior work in four ways. First, database studies examine how schemas change over a project's life [18,20,21]; we ask the same question of UI schemas, which many client versions read at once, and find that most schema commits and individual schema changes are additive. Second, breaking changes are common in library APIs: Raemaekers et al. [24] found that roughly one third of Maven Central releases break at least one API element and that deprecation tags are rarely applied systematically beforehand. Grouping our runtime CDUI commits by official git version tags gives a direct comparison by release. Across 211 tagged SDK releases in DivKit, Redwood, and HubFramework, 21.3% of all releases (45 of 211) and 40.5% of the 111 releases that touched the UI schema shipped at least one breaking change, and none of the 780 deleted runtime declarations were marked deprecated in an earlier commit first.

Third, Li et al. [28] observed that web API clients often cannot stay on an old version. CDUI is a stronger case of the same constraint, because the server sends the interface itself to every installed client version, which is why how clients handle unknown elements (Section 6.3) matters. Fourth, Meinicke et al. [32] studied feature flags as a way to vary behavior at runtime, while CDUI schemas vary structure: which UI elements exist, rather than which code paths run. That difference matters because structural changes affect what an older client can render, while flags affect only which code path runs.

8.4. Scope, Sample Size, and Generalizability

How far these results carry over to other CDUI systems depends on how they manage schemas and update clients. First, private company platforms (such as Airbnb's Ghost Platform, Uber, and Faire [2,7,8]) often use central schema registries, dedicated reviewers, and mandatory CI checks, which may keep breaking commits below our 38.1% open-source average. Second, binary Protobuf schemas [2,9] match fields by fixed tag numbers and `reserved` deleted tags rather than JSON names: renaming a field is safe on the wire, whereas changing a type or reusing an old tag breaks older clients. Third, web server component and HTML streaming frameworks (such as React Server Components, Phoenix LiveView, and Hotwire Turbo) load matching client code on page reload, avoiding the lag across old app versions ($N - k$) on Android and iOS.

Two limitations of our dataset also matter. First, because most commercial CDUI schemas are private and ten public candidates did not meet our selection rules (Table 1), our sample covers four runtime engines (two per format group) plus one build-time comparator. Even with 1447 commits and 6866 schema changes, having two engines per

group leaves wide uncertainty (−4.6 to +24.7 points on the 11.5 point format gap), and one project's workflow (such as EnsembleUI's single schema file or Redwood's first eight months) can shift combined totals. Second, we checked how parser errors affect our totals using the error rates from our 100 human-labeled commits (97.5% grouped accuracy on the 80 runtime commits and 94.0% overall in Table 7). Adjusting for those error rates shifts the additive commit share across the four runtime engines by −1.2 points, from 61.4% (38.1% breaking) to 60.2% (39.3% breaking), and across all five projects from 63.5% to 58.7%, leaving our main RQ3 finding unchanged. Most of the five-project shift comes from Lona, where the parser matched human labels on 16 of 20 sampled commits (four of our seven sample errors). Adjusting Lona alone lowers its estimated additive share from 70.9% to 58.6% (63.2% when weighted by category share), so totals that include Lona should be read with extra caution.

8.5. Implications for Practitioners

Our results suggest five things engineering teams can do when they adopt or maintain a CDUI system:

1. **Make schema changes easy to review and to check.** In our sample, schemas kept in standalone files show a higher share of additive changes. Either format can support a clear review path and automatic checking, and teams can track their own ratio of additive to breaking changes over time.
2. **Check compatibility automatically in CI.** Breaking changes account for 38.1% of schema commits across the four runtime engines, so a merge pipeline that validates backward compatibility, in the way API linters such as Spectral do, can catch a contract violation before it reaches production clients.
3. **Track the cost of version routing in the BFF.** Version branches multiply with the number of supported versions and component types, so teams can watch the complexity of their routing adapters and set a support window for old client versions. Finding the point at which refactoring pays off is open work.
4. **Decide how clients handle unknown components.** Adding to the schema is safe at runtime only when older clients tolerate elements they do not recognize. The studied runtime engines differ here: DivKit drops list items that fail to parse, HubFramework substitutes a fallback component, EnsembleUI ignores unknown properties on known widgets but throws an error on unknown widgets, and Redwood throws an exception by default unless the host application supplies a custom handler. Teams should choose and test how clients handle both unknown properties and unknown widgets against payloads from a newer schema.
5. **Put deprecation and migration policies in place deliberately.** Breaking UI schema edits fell significantly over time only in Redwood and stayed flat or rose in the other repositories, and none of the 780 removed runtime declarations went through a prior deprecation commit before deletion (while all 10 declarations marked deprecated remained in the schema). Teams cannot rely on project age to reduce breaking edits automatically and should set explicit deprecation windows before removing fields.

9. Threats to Validity

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

- **Internal Validity:** Two things limit what the review can support. The industry sources behind RQ1, RQ2, and RQ4 are mainly engineering blogs, talks, and framework documentation, which report successes more often than abandoned attempts. We recorded our search terms, databases, and dates (Section 3.1), but deciding which

individual sources to keep remained a judgement call without step-by-step hit counts, so the review reflects what we found and read; we therefore use it as context and base our measurements on the commit data. In the format comparison, the two groups also differ in project purpose, team practices, and release policy; separating those from format would need a larger study that holds them constant.

- **External Validity:** What large companies such as Uber and Airbnb do may not carry over to smaller teams without a platform group of their own. Our commit analysis covers four open-source runtime CDUI engines and one design tool comparator compiled at build time, three schema formats, and a decade of history, which is broader than a study of one framework, but frameworks whose code is private may evolve their schemas differently.
- **Construct Validity:** Although our parsers read full JSON trees, Kotlin declarations, and Objective-C headers rather than matching raw diff lines, static schema diffs still have edge cases. Adding a new value to an `enum` list is counted as a property addition in the schema even though an older client with a strict enum parser can reject payloads carrying that new value. Removing an optional field is likewise counted as a removal even when older clients do not require it; Section 6.2 reports how much the totals shift under both alternatives. Assigning one label per commit under a most destructive change rule also makes projects that bundle large edits look more breaking per commit, which is why we report individual schema change counts alongside labels per commit. Finally, both validation raters are co-authors and one (V.S.) wrote the parser; to limit bias, both raters labeled all 100 commits independently from GitHub diffs using written rules before viewing the parser's output.
- **Practical Relevance:** A breaking change in the schema does not necessarily cause a crash or a rendering failure for an end user, and our evidence stops at the schema. We leave tool building, evaluation with practitioners, and measurement of outcomes such as crash rates or maintenance effort to future work.

10. Conclusions and Future Research

CDUI has grown from a way to change strings remotely into an architecture where the schema acts as a long-lived contract between the server and many installed app versions. Across 1447 commits touching schema paths in five repositories (after setting aside 504 commits that left the schema unchanged), 730 commits modified a runtime CDUI engine's schema: 61.4% only add declarations, 38.1% remove or retype existing declarations, and 0.5% mark deprecations. Including the build-time Lona comparator (943 schema commits) gives 63.5% additive, 36.1% breaking, and 0.4% deprecations, while counting all 5216 individual runtime schema changes raises the additive share to 70.3%. Across the four runtime engines, schemas kept in standalone data files averaged 65.3% additive commits versus 53.8% for schemas written in code, an exploratory pattern whose 95% confidence interval across the four repositories spans zero. Finally, adding to a schema is safe at runtime only when older clients tolerate elements they do not recognize. DivKit and HubFramework fall back safely on unknown components, EnsembleUI ignores unknown properties on known widgets but throws an error on unknown widgets, and Redwood throws by default unless the host app supplies its own mismatch handler.

Future Research

Four open research directions follow from the boundaries of this study. First, studies of production BFF codebases are needed to measure how version routing branches grow in practice, alongside testing our three comparison axes across a wider set of commercial and open-source CDUI systems. Second, automated CI checks that combine schema

diff checks with client parser tests could verify before deployment whether a proposed change is safe for specific older app versions, particularly as declarative client toolkits and server UI protocols converge [14]. Third, benchmarking portable WebAssembly sandboxes and serverless edge runtimes [44] for CDUI could clarify whether layout assembly and embedded widget logic can run safely without custom expression languages. Fourth, as large language models are used to generate UI configurations from user prompts [45], automated schema and accessibility checks [11,22] will be needed to validate generated payloads before they reach older clients.

Author Contributions: Conceptualization, V.S. and M.A.; methodology, V.S.; software, V.S.; validation, V.S. and P.M.; formal analysis, V.S. and S.S.; investigation, V.S., M.A. and P.M.; data curation, V.S.; writing—original draft preparation, V.S.; writing—review and editing, V.S., M.A., P.M., S.S. and A.M.; visualization, V.S. and A.M.; supervision, V.S.; project administration, V.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The analysis scripts, the classification results for each commit in all five frameworks, and the manual validation sample with its labeling guide are openly available in Zenodo at https://doi.org/10.5281/zenodo.23903746. The underlying source data are the public commit histories of the five analyzed repositories, which are cited in the manuscript.

Conflicts 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.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
BFF	Backend-for-Frontend
CDUI	Config-Driven User Interface
CI	Continuous Integration
gRPC	gRPC Remote Procedure Calls
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KMP	Kotlin Multiplatform
MBUID	Model-Based User Interface Development
MVC	Model–View–Controller
MVVM	Model–View–ViewModel
Protobuf	Protocol Buffers
REST	Representational State Transfer
RIBs	Router-Interactor-Builder
RQ	Research Question
SDK	Software Development Kit
SDUI	Server-Driven User Interface
UI	User Interface
WCAG	Web Content Accessibility Guidelines

References

1. Quin, F.; Weyns, D.; Galster, M.; Silva, C.C. A/B testing: A systematic literature review. *J. Syst. Softw.* **2024**, *211*, 112011. https://doi.org/10.1016/j.jss.2024.112011.
2. Bao, P. Transitioning to Server-Driven UI. The Craft, Faire Engineering Blog. Available online: https://craft.faire.com/transitioning-to-server-driven-ui-a76b216ed408 (accessed on 30 October 2026).
3. Zhu, W.; Proksch, S.; German, D.M.; Godfrey, M.W.; Li, L.; McIntosh, S. What is an app store? The software engineering perspective. *Empir. Softw. Eng.* **2024**, *29*, 35. https://doi.org/10.1007/s10664-023-10362-3.
4. Krasner, G.; Pope, S. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object-Oriented Program.* **1988**, *1*, 26–49.
5. Jošt, G.; Taneski, V. State-of-the-art cross-platform mobile application development frameworks: A comparative study of market and developer trends. *Informatics* **2025**, *12*, 45. https://doi.org/10.3390/informatics12020045.
6. Spotify. Spotify Hub Framework. Available online: https://github.com/spotify/HubFramework (accessed on 30 October 2026).
7. Lei, B.; Dupree, B.; Nguyen, Q. Architecting a Safe, Scalable, and Server-Driven Platform for Driver Preferences with RIBs. Uber Engineering Blog. Available online: https://www.uber.com/blog/carbon-driver-app-preferences-ribs/ (accessed on 30 October 2026).
8. Brooks, R. A Deep Dive into Airbnb's Server-Driven UI System. Airbnb Engineering Blog. Available online: https://medium.com/airbnb-engineering/a-deep-dive-into-airbnbs-server-driven-ui-system-842244c5f5 (accessed on 30 October 2026).
9. Shatnawi, A.; Bahri, A.; Niang, B.T.; Verhaeghe, B. Enhancing data serialization efficiency in REST services: Migrating from JSON to protocol buffers. In Proceedings of the 20th International Conference on Software Technologies (ICSOFT), Bilbao, Spain, 10–12 June 2025; pp. 193–200. https://doi.org/10.5220/0013459500003964.
10. Di Gregorio, M.; Di Nucci, D.; Palomba, F.; Vitiello, G. The making of accessible Android applications: An empirical study on the state of the practice. *Empir. Softw. Eng.* **2022**, *27*, 145. https://doi.org/10.1007/s10664-022-10182-x.
11. Zhang, M.; Liu, H.; Chen, C.; Gao, G.; Li, H.; Zhao, J. AccessFixer: Enhancing GUI accessibility for low vision users with R-GCN model. *IEEE Trans. Softw. Eng.* **2024**, *50*, 173–189. https://doi.org/10.1109/TSE.2023.3337421.
12. Apple Inc. SwiftUI: Declarative App Development. Available online: https://developer.apple.com/documentation/swiftui (accessed on 30 October 2026).
13. Google. Jetpack Compose. Available online: https://developer.android.com/develop/ui/compose (accessed on 30 October 2026).
14. Cash App. Cash App Redwood: Multiplatform Reactive UI. Available online: https://github.com/cashapp/redwood (accessed on 30 October 2026).
15. Biørn-Hansen, A.; Grønli, T.-M.; Ghinea, G.; Alouneh, S. An empirical study of cross-platform mobile development in industry. *Wirel. Commun. Mob. Comput.* **2019**, *2019*, 5743892. https://doi.org/10.1155/2019/5743892.
16. EnsembleUI: Build Native Apps with JSON. Available online: https://github.com/EnsembleUI/ensemble (accessed on 30 October 2026).
17. Peltonen, S.; Mezzalira, L.; Taibi, D. Motivations, benefits, and issues for adopting Micro-Frontends: A multivocal literature review. *Inf. Softw. Technol.* **2021**, *136*, 106571. https://doi.org/10.1016/j.infsof.2021.106571.
18. Curino, C.; Moon, H.J.; Tanca, A.; Zaniolo, C. Schema evolution in Wikipedia: Toward a web information system benchmark. In Proceedings of the International Conference on Enterprise Information Systems (ICEIS), Barcelona, Spain, 12–16 June 2008; pp. 323–332. https://doi.org/10.5220/0001713003230332.
19. Rahm, E.; Bernstein, P.A. An online bibliography on schema evolution. *ACM SIGMOD Rec.* **2006**, *35*, 30–31. https://doi.org/10.1145/1228268.1228273.
20. Qiu, D.; Li, B.; Su, Z. An empirical analysis of the co-evolution of schema and code in database applications. In Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE), Saint Petersburg, Russia, 18–26 August 2013; pp. 125–135. https://doi.org/10.1145/2491411.2491431.
21. Scherzinger, S.; Sidortschuck, S. An empirical study on the design and evolution of NoSQL database schemas. In *Conceptual Modeling. ER 2020*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2020; pp. 441–455. https://doi.org/10.1007/978-3-030-62522-1_33.
22. Attouche, L.; Baazizi, M.-A.; Colazzo, D.; Ghelli, G.; Sartiani, C.; Scherzinger, S. Validation of modern JSON Schema: Formalization and complexity. *Proc. ACM Program. Lang.* **2024**, *8*, 1451–1481. https://doi.org/10.1145/3632891.
23. Dig, D.; Johnson, R. How do APIs evolve? A story of refactoring. *J. Softw. Maint. Evol. Res. Pract.* **2006**, *18*, 83–107. https://doi.org/10.1002/smr.328.
24. Raemaekers, S.; van Deursen, A.; Visser, J. Semantic versioning and impact of breaking changes in the Maven repository. *J. Syst. Softw.* **2017**, *129*, 140–158. https://doi.org/10.1016/j.jss.2016.04.008.
25. Brito, A.; Xavier, L.; Hora, A.; Valente, M.T. Why and how Java developers break APIs. In Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 20–23 March 2018; pp. 255–265. https://doi.org/10.1109/SANER.2018.8330214.

26. Bogart, C.; Kästner, C.; Herbsleb, J.; Thung, F. How to break an API: Cost negotiation and community values in three software ecosystems. In Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Seattle, WA, USA, 13–18 November 2016; pp. 109–120. https://doi.org/10.1145/2950290.2950325.
27. McDonnell, T.; Ray, B.; Kim, M. An empirical study of API stability and adoption in the Android ecosystem. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM), Eindhoven, The Netherlands, 22–28 September 2013; pp. 70–79. https://doi.org/10.1109/ICSM.2013.18.
28. Li, J.; Xiong, Y.; Liu, X.; Zhang, L. How does web service API evolution affect clients? In Proceedings of the IEEE International Conference on Web Services (ICWS), Santa Clara, CA, USA, 28 June–3 July 2013; pp. 300–307. https://doi.org/10.1109/ICWS.2013.48.
29. Schmiedmayer, P.; Bauer, A.; Bruegge, B. Reducing the impact of breaking changes to web service clients during web API evolution. In Proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), Melbourne, Australia, 15–16 May 2023; pp. 1–11. https://doi.org/10.1109/MOBILSoft59058.2023.00008.
30. Lercher, A.; Glock, J.; Macho, C.; Pinzger, M. Microservice API evolution in practice: A study on strategies and challenges. *J. Syst. Softw.* **2024**, *215*, 112110. https://doi.org/10.1016/j.jss.2024.112110.
31. Serbout, S.; Pautasso, C. How are Web APIs versioned in practice? A large-scale empirical study. *J. Web Eng.* **2024**, *23*, 465–506. https://doi.org/10.13052/jwe1540-9589.2341.
32. Meinicke, J.; Wong, C.-P.; Vasilescu, B.; Kästner, C. Exploring differences and commonalities between feature flags and configuration options. In Proceedings of the ACM/IEEE International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Seoul, Republic of Korea, 24 June–16 July 2020; pp. 233–242. https://doi.org/10.1145/3377813.3381366.
33. Mahdavi-Hezaveh, R.; Dremann, J.; Williams, L. Software development with feature toggles: Practices used by practitioners. *Empir. Softw. Eng.* **2021**, *26*, 1. https://doi.org/10.1007/s10664-020-09901-z.
34. Calvary, G.; Coutaz, J.; Thevenin, D.; Limbourg, Q.; Bouillon, L.; Vanderdonckt, J. A unifying reference framework for multi-target user interfaces. *Interact. Comput.* **2003**, *15*, 289–308. https://doi.org/10.1016/S0953-5438(03)00010-9.
35. Yandex DivKit: An Open Framework for Server-Driven UI. Available online: https://github.com/divkit/divkit (accessed on 30 October 2026).
36. Airbnb Lona: A Tool for Defining Design Systems. Available online: https://github.com/Lona/Lona (accessed on 30 October 2026).
37. Feitelson, D.G.; Frachtenberg, E.; Beck, K.L. Development and deployment at Facebook. *IEEE Internet Comput.* **2013**, *17*, 8–17. https://doi.org/10.1109/MIC.2013.25.
38. Heitkötter, H.; Hanschke, S.; Majchrzak, T.A. Evaluating cross-platform development approaches for mobile applications. In *Web Information Systems and Technologies (WEBIST 2012)*; Lecture Notes in Business Information Processing; Springer: Berlin/Heidelberg, Germany, 2013; Volume 140; pp. 120–138. https://doi.org/10.1007/978-3-642-36608-6_8.
39. Bolanowski, M.; Żak, K.; Paszkiewicz, A.; Ganzha, M.; Paprzycki, M.; Sowiński, P.; Lacalle, I.; Palau, C.E. Efficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems. In *New Trends in Intelligent Software Methodologies, Tools and Techniques (SOMET 2022)*; Frontiers in Artificial Intelligence and Applications; IOS Press: Amsterdam, The Netherlands, 2022; Volume 355; pp. 97–108. https://doi.org/10.3233/FAIA220242.
40. Pinto-Agüero, S.; Noel, R. Microservices evolution factors: A multivocal literature review. *IEEE Access* **2025**, *13*, 88707–88730. https://doi.org/10.1109/ACCESS.2025.3570658.
41. McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320. https://doi.org/10.1109/TSE.1976.233837.
42. Cash App. Zipline: Run Kotlin/JS Libraries in Kotlin/JVM and Kotlin/Native Programs. Available online: https://github.com/cashapp/zipline (accessed on 30 October 2026).
43. Lehman, M.M. Programs, life cycles, and laws of software evolution. *Proc. IEEE* **1980**, *68*, 1060–1076. https://doi.org/10.1109/PROC.1980.11805.
44. De Palma, G.; Giallorenzo, S.; Mauro, J.; Trentin, M.; Zavattaro, G. WebAssembly at the edge: Benchmarking a serverless platform for private edge cloud systems. *IEEE Internet Comput.* **2024**, *28*, 37–44. https://doi.org/10.1109/MIC.2024.3513035.
45. Wu, J.; Schoop, E.; Leung, A.; Barik, T.; Bigham, J.P.; Nichols, J. UICoder: Finetuning large language models to generate user interface code through automated feedback. In Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Mexico City, Mexico, 16–21 June 2024; pp. 7511–7525. https://doi.org/10.18653/v1/2024.naacl-long.417.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.