American Impact Review

Review Article

From UIKit to SwiftUI: a reproducible protocol and a priori typology for open-source iOS application migration

Tomasz Kubiak*

Independent Researcher, Kraków, Poland

ORCID: 0009-0006-1245-8838

* Corresponding author: tomasz.m.kubiak@119mail.com

OPEN ACCESS

Citation:
Tomasz Kubiak (2026). From UIKit to SwiftUI: a reproducible protocol and a priori typology for open-source iOS application migration. *Am. Impact Rev.* 10.66308/air.e2026068

Received: May 7, 2026
Accepted: June 14, 2026
Published: June 18, 2026

DOI:
10.66308/air.e2026068

ISSN: 3071-124X

Abstract

The coexistence of UIKit and SwiftUI has made iOS user-interface migration technically incremental but empirically difficult to classify. Prior repository research identifies several broad adoption trajectories, yet it does not distinguish replacement of existing functionality from greenfield addition, capability-specific adoption, stable coexistence, or UIKit fallback inside a SwiftUI-first application. This structured analytical review develops a reproducible protocol and an a priori, mechanism-oriented typology for a longitudinal multiple-case analysis of five production open-source iOS applications. The evidence base combines peer-reviewed software-engineering research, public Git histories and project documents, and official Apple interoperability documentation. The protocol defines a migration episode through integration direction, functional granularity, trigger, fate of UIKit artifacts, compatibility constraints, and associated test changes. It specifies monthly repository snapshots, SwiftSyntax and XML extraction, manual dual coding, agreement assessment, within-case timelines, cross-case synthesis, and sensitivity checks. The resulting analytical framework comprises six distinguishable configurations and a conditional matrix linking observable constraints to interoperability seams, candidate strategies, and scope-specific completion criteria. The framework treats SwiftUI growth without corresponding UIKit decline as adoption rather than replacement and uses a SwiftUI-first case with isolated UIKit bridges as a negative control. Because repository-wide extraction has not yet been executed, the article reports no project counts, effect estimates, or causal claims. Its contribution is therefore methodological and conceptual: it converts a broad technology transition into falsifiable classifications that can be evaluated on frozen, openly accessible repositories.

Keywords: SwiftUI, UIKit, software migration, mining software repositories, iOS development, framework interoperability, longitudinal case study, open-source software

Highlights

- A reproducible protocol separates UIKit-to-SwiftUI migration from adoption.
- Six configurations distinguish replacement, addition, coexistence, and fallback.
- Episodes are classified by direction, scope, trigger, UIKit fate, and constraints.
- A conditional matrix links constraints to interoperability seams and exit criteria.
- A SwiftUI-first negative control prevents bridges from being classified as migration.

1. Introduction

SwiftUI did not replace UIKit through a single platform event. Apple introduced a declarative user-interface framework while retaining the imperative framework, its controls, and explicit mechanisms for embedding one technology inside the other. The resulting transition is observable in repositories as a mixture of new SwiftUI code, removal or retention of UIKit artifacts, bridging controllers, representable wrappers, and changing module boundaries. A current repository can therefore contain both frameworks for several fundamentally different reasons. It may be midway through replacement, may reserve SwiftUI for new functionality, may isolate a SwiftUI module behind a UIKit route, or may be SwiftUI-first while retaining a UIKit capability for which no equivalent has been adopted. Treating all of these states as a single category of "migration" obscures the mechanism that produced them.

The closest large empirical study, by Baresi et al., examined the evolution of iOS development technologies in 140 open-source applications and identified four broad SwiftUI trajectories: exclusive use, complete transition, hybrid integration, and modest addition [1]. That work establishes heterogeneity rather than uniform replacement, and its authors explicitly leave project rationales and migration mechanisms for qualitative investigation. A separate historical analysis of programming-language use in the iOS ecosystem likewise shows why a single snapshot is inadequate for reconstructing technological change [2]. Neither contribution, however, operationalizes the boundary between migration and adoption at the level of a feature, flow, module, or runtime integration seam.

This gap matters because adjacent evidence cannot safely substitute for migration evidence. Static quality indicators in iOS applications may describe code properties, but they do not establish that a UI-framework change improved or degraded quality [3]. Small comparative implementations can clarify the declarative and imperative programming models of SwiftUI and UIKit, yet they provide limited evidence about the evolution of long-lived production codebases [4]. A credible migration study must therefore reconstruct what changed, where it changed, what was removed or retained in the same functional scope, and which rationale was actually documented. It must also preserve the distinction between temporal association and causation.

The object of this study is the evolution of the UI layer in production open-source iOS applications after the public introduction of SwiftUI. Its subject is the sequence and form of UIKit-to-SwiftUI change: integration direction, functional granularity, migration trigger, fate of UIKit artifacts, minimum supported iOS version, API constraints, and associated testing or architectural documentation. The aim is to formulate a reproducible protocol and a mechanism-oriented classification that can support a subsequent longitudinal multiple-case analysis without presupposing that one strategy is universally superior.

The main research question is: **Which reproducible UIKit-to-SwiftUI migration strategies can be distinguished in open-source iOS applications, and under what observable technical conditions do their applicability boundaries differ?** Four subsidiary questions refine it:

1. At what functional granularity is SwiftUI introduced, and through which runtime integration seam?
2. What longitudinal artifact pattern distinguishes replacement from addition or bounded coexistence?
3. Which public project documents associate deployment-target, API, navigation, state-management, or modularity constraints with a particular boundary?
4. What evidence is required to distinguish migration of existing functionality from SwiftUI-first development with an isolated UIKit fallback?

The contribution is deliberately bounded. First, the study specifies an a priori typology in which direction, granularity, trigger, and UIKit fate jointly determine classification; an import statement alone does not. Second,

it turns interoperability mechanisms into a conditional decision matrix with entry conditions and completion criteria. Third, it defines a repository-mining procedure that combines syntax-aware extraction with manual validation and public decision documents. Finally, it introduces a negative-control role for a SwiftUI-first application containing UIKit bridges, preventing interoperability from being mistaken for migration. These are protocol and framework outputs, not completed repository-wide empirical findings. The article reports no uncalculated counts, productivity effects, defect outcomes, or claims about the App Store population.

2. Literature Review

2.1. iOS technology evolution and the limits of framework comparison

Research on iOS evolution provides two complementary starting points. Baresi et al. trace project histories and show that SwiftUI adoption follows several trajectories rather than one replacement curve [1]. Domínguez-Álvarez et al. focus on programming-language usage, but their longitudinal perspective reinforces a general methodological point: platform evolution is a sequence of repository states, not a property of the latest revision [2]. Together, these studies justify temporal reconstruction while leaving open the finer-grained question of how UI-framework boundaries move inside an application.

Two other strands are informative mainly because they mark the limits of inference. Habchi et al. demonstrate that static analysis can compare code smells in mobile ecosystems [3]. Their measures, however, concern code characteristics and precede the SwiftUI transition; they cannot be used as outcome evidence for migration strategy. Wiertel and Skublewska-Paszkowska compare UIKit and SwiftUI through relatively small applications and identify differences between imperative and declarative construction [4]. Such comparisons help define the technological contrast, but their external validity does not extend automatically to multi-year product repositories with mixed architectures, deployment constraints, and accumulated navigation or state-management conventions. The present study consequently treats framework properties as context, not as proof that migration yields a particular quality or productivity effect.

2.2. Migration as a heterogeneous, scoped process

Empirical migration research outside iOS offers a stronger vocabulary for mechanism. Islam et al. show that library migration comprises heterogeneous change types and cannot be reduced to a simple one-to-one API mapping [5]. He et al. similarly extract migration events and rationales from Java projects, while noting that explicit motivations are not available for every event [6]. These findings support two design choices here: candidate events must be reconstructed from coordinated changes, and an undocumented rationale must remain undocumented rather than be inferred from repository age or code style.

Architectural migration studies add the idea of scope. Ayas et al. describe migration toward microservices as a process involving technical and systemic changes rather than a single conversion step [7]. The domain is different, so no microservice result is transferred directly to UI frameworks. The useful analogy is narrower: migration may proceed through bounded units and may leave durable interfaces between old and new structures. Balalaie et al. organize migration patterns through context, problem, solution, and consequences [8]. That pattern form is appropriate for the present typology because it forces each strategy to state the conditions under which it is recognizable and the criterion by which its intended scope can be considered complete.

These migration studies also expose a terminological problem. “Adoption” describes the appearance or expansion of a technology; “replacement” requires evidence that an existing implementation declines or disappears within a defined scope. “Coexistence” describes a state but does not by itself reveal whether that state is temporary, capability-driven, or architecturally bounded. “Fallback” reverses the historical assumption:

UIKit may be embedded in a codebase whose baseline was already SwiftUI. A useful typology must therefore classify episodes, not repositories as indivisible objects, and must record the pre-state and post-state of the same functional unit.

2.3. Repository mining, corpus validity, and documentary interpretation

Mining GitHub introduces threats that are especially consequential for migration studies. Kalliamvakou et al. show that repository metadata and visible activity can misrepresent how projects are developed [9]; their extended study further documents incomplete histories and other hazards of interpreting GitHub as a transparent record of engineering work [10]. Munaiah et al. address the related problem of distinguishing engineered software projects from tutorials, personal experiments, and other repositories [11]. These findings justify the exclusion of demos and libraries, the use of production-distribution evidence, the recording of repository moves, and the refusal to treat stars or forks as quality indicators.

The literature-selection procedure follows established software-engineering guidance. Petersen et al. require transparent search, screening, and classification for systematic mapping [12], while Wohlin shows how backward and forward snowballing can complement an initial set of direct sources [13]. The present corpus is not claimed as an exhaustive systematic review; it is a purpose-built methodological corpus. Nevertheless, its lines of evidence are separated into iOS evolution, migration taxonomies, repository validity, qualitative synthesis, and official interoperability mechanisms so that each source has a defined analytical function.

Temporal sequence also creates a risk of causal overstatement. Tufano et al. examine when code smells are introduced, but the more general lesson for this study is that proximity between a change and a subsequent state does not establish that one caused the other [14]. Public explanations require structured treatment rather than selective quotation. Cruzes and Dybå describe a staged thematic synthesis for software-engineering evidence [15], and Krippendorff provides the basis for a codebook whose categories are explicit, reviewable, and, where possible, mutually distinguishable [16]. Accordingly, project documents will be coded as evidence of a publicly recorded constraint or rationale, not as a complete account of organizational decision-making.

2.4. Interoperability as mechanism, not effectiveness evidence

Apple's SwiftUI documentation defines the declarative framework and its composition model [17]. More important for migration research, the interoperability documentation specifies how UIKit views and controllers can be exposed to SwiftUI through representable protocols [18]. The WWDC session "Use SwiftUI with UIKit" documents the opposite direction through UIHostingController and component-level use through UIHostingConfiguration, while also emphasizing explicit data-flow boundaries [19]. These materials establish that several seams are technically supported. They do not establish their cost, maintainability, or superiority in production.

Project documentation can add case-specific constraints. The Firefox iOS wiki explicitly links selective SwiftUI use to its supported iOS baseline and to the availability of APIs in newer system versions [20]. This is stronger than inferring motivation from commit timing because the project states the constraint. It is still one project's documented policy, not a universal causal law. The distinction illustrates the central evidence gap: broad adoption trajectories are known, and interoperability mechanisms are documented, but the literature has not yet connected episode sequence, seam, scope, and constraint in a reproducible classification. The protocol below is designed to make that connection testable.

3. Materials and Methods

3.1. Study type and evidence architecture

This is a structured analytical review and methodological protocol for a mixed-method longitudinal multiple-case study. It uses only openly accessible materials and reports the analytical outputs that can be established before repository-wide extraction. The future empirical phase will combine mining software repositories, structured analysis of public project documents, and cross-case synthesis. Quantitative traces answer what changed and when; documents record which constraint or rationale was publicly stated; comparative synthesis determines whether a configuration recurs across cases. The design permits analytical generalization to technically similar settings, but not statistical estimation for all iOS applications.

The evidence architecture contains four corpora. The scientific-methodological corpus comprises peer-reviewed work on iOS evolution, software migration, repository validity, and qualitative synthesis [1 - 16]. The longitudinal Git corpus contains four migration cases and one negative control. The documentary corpus comprises repository-owned wiki pages, README or AGENTS files, issues, pull-request descriptions and reviews, and commit messages that state a concrete decision or constraint. The interoperability corpus contains official Apple documentation [17 - 19]. A curated directory of open-source iOS applications is used only for candidate discovery, not as evidence that a project migrated [21].

3.2. Case selection and frozen corpus

Cases were selected purposively to maximize contrast in the broad trajectories identified in prior work while retaining public, auditable histories. Inclusion requires a production application, an accessible Git history and license, an observable UIKit baseline followed by runtime SwiftUI use for migration cases, at least 24 months of history or a completed transition, and enough project context to connect code changes with public documents. Tutorials, sample applications, libraries, forks without independent histories, snapshot-only repositories, and projects in which SwiftUI appears only in previews or tests are excluded. A SwiftUI-first case is retained solely as a negative control because its UIKit bridge tests the boundary of the migration definition.

Table 1. Frozen case corpus and analytical role. The table establishes the comparative logic of the sample; it does not report migration outcomes.

Case	Frozen public repository state	Analytical role and inclusion rationale
SimpleLogin	simple-login/Simple-Login-iOS; HEAD 4b50dda9f74f14b6b2dfa5caa6a373 3187e61a76	Candidate complete-replacement trajectory reported in prior research; production application with a UIKit baseline [1, 23].
WooCommerce	woocommerce/woocommerce-ios; HEAD 3dcd6350da6fd411d4b813f3610eca 38d146e226	Candidate hybrid and module-by-module trajectory; public issues and module documentation expose navigation and boundary decisions [22].
DuckDuckGo	archived duckduckgo/iOS, HEAD 7b3f6010d27a3a69fe92a2ad698543 f2e67c8900; successor duckduckgo /apple-browsers, HEAD 8d81e8c892 6f59bb6a7e06444cab49be91c5f0eb	Candidate modest or capability-specific addition; documented repository move permits a continuity sensitivity check [1, 24].
Firefox iOS	mozilla-mobile/firefox-ios; HEAD 0b209058dfe94f60b51a9cb0ae7240 6359b4af1c	Candidate bounded coexistence with an explicit public policy concerning deployment target and API availability [20].
E-Rezept	gematik/E-Rezept-App-iOS; HEAD 3f158fa05141ef2eef197599cfab71 ad429cec4b	SwiftUI-first negative control with isolated UIKit fallback; tests whether interoperability is wrongly classified as migration [1, 25].

Earlier repository history may be used to establish each UIKit baseline, while the SwiftUI analytical window begins with its public introduction on 3 June 2019. Each remote URL, default branch, license, frozen SHA, and documented repository move will be preserved in the replication metadata. Source code will not be republished contrary to project licenses; the replication package will contain hashes, scripts, exclusion rules, and derived event records.

3.3. Unit of analysis and operational definitions

The primary unit is a **migration episode**: a connected sequence of commits or pull requests in which SwiftUI replaces existing UIKit or storyboard functionality, or alters an established UI-layer boundary. Nested units are the commit or pull request, UI file, functional component, screen, flow, module, application lifecycle, and monthly repository snapshot. This nesting prevents a project-wide label from erasing different strategies used in different areas of one application.

An episode is distinguished from an **addition**, defined as a new SwiftUI function without observable removal or replacement of a UIKit equivalent in the same functional scope. An **integration seam** is the runtime boundary through which the frameworks coexist: UIHostingController, UIHostingConfiguration, UIViewRepresentable, UIViewControllerRepresentable, a coordinator or router boundary, or a shared model layer. **Completion** refers to the disappearance of substantial UIKit UI artifacts from the declared migration scope, not the literal absence of every UIKit reference in the repository. **Pace** refers to descriptive change in UI artifacts over fixed snapshots; it is not developer velocity or labor productivity.

Each candidate episode will be coded along seven dimensions: integration direction (SwiftUI-in-UIKit, UIKit-in-SwiftUI, or bidirectional), functional granularity, UIKit fate within scope, trigger, compatibility constraint, associated test change, and strategy. Trigger values are replacement, new feature, platform-only capability, architecture, and undocumented. "Undocumented" is not interpreted as "no rationale." Compatibility is

coded only when a deployment-target constraint, API gap, or absence of such a constraint is supported by public evidence.

3.4. Repository extraction and event detection

Each repository will be mirrored and analyzed on its default branch. Pods, Carthage artifacts, Swift Package Manager build directories, generated code, fixtures, and vendored code will be removed according to a published YAML rule set. The final commit of each calendar month will form the regular series; tagged releases and all candidate migration commits will be added as event-sensitive observations. Missing periods will remain missing rather than be interpolated.

Swift files will be parsed with SwiftSyntax. Storyboard and XIB files will be parsed as XML. The extraction will record framework imports, definitions conforming to View, subclasses of UIView or UIViewController, storyboard and XIB artifacts, interoperability constructs, module or feature path, test files, deployment target, and line additions or deletions. Regular expressions may generate candidate locations, but final classification will rely on syntax, target membership, surrounding context, and manual review. This restriction addresses the risk that imports occur only in previews, tests, dead code, generated files, or files that do not participate in the application target.

Candidate events will be generated from five signals: first runtime SwiftUI use; addition of a hosting or representable seam; concurrent deletion of UIKit or storyboard artifacts and addition of SwiftUI artifacts in the same feature path; migration language such as “replace,” “rewrite,” or “migrate” in a linked change; and a related issue or pull request. A candidate becomes a validated episode only after the pre-state, change, post-state, scope, and runtime boundary can be reconstructed. This rule operationalizes the difference between a technology appearing in a repository and an existing UI responsibility moving across frameworks.

3.5. Documentary coding and reliability

The documentary corpus is restricted to project-owned wiki pages, repository documentation, issues, pull requests, reviews, and commit messages containing a concrete decision or constraint. User comments without code linkage, general feature requests, marketing texts, and third-party tutorials are excluded. The unit is a meaning-bearing decision fragment. Categories include trigger, granularity, direction, seam, state or data flow, navigation, compatibility, testing, rollback, outcome, rationale confidence, and link to a code event.

Documents will be linked to a commit or pull request and to a window of 90 days on either side. A document outside that window can describe current policy but cannot be used as the motive for an earlier event. Two coders will independently review all candidate episodes. Cohen’s kappa will be calculated on 20% of the corpus; a value below 0.70 will trigger codebook revision and recoding. Because no coding run has yet occurred, no agreement coefficient is reported here. The procedure, threshold, and consequence are specified in advance to prevent post hoc adjustment.

3.6. Longitudinal measures and cross-case synthesis

For each snapshot, the study will calculate counts and normalized shares of UIKit, storyboard/XIB, SwiftUI, and mixed-framework UI artifacts. Event-aligned series will compare the direction of change before and after validated episodes. File counts and lines of code will be retained as separate proxies; neither is interpreted as functional weight or effort. Within each case, the analysis will produce a causally neutral sequence: observed state, event, subsequent observed state, and any documented constraint.

Cross-case synthesis will cluster episodes by direction, granularity, UIKit fate, and trigger. A recurring pattern

will be recognized when it occurs in at least two cases or when a single case is supported by an independent empirical study or an official interoperability mechanism. Unique configurations remain case-specific or provisional. Classification will be tested under four sensitivity conditions: files versus lines of code, inclusion versus exclusion of previews and tests, monthly versus quarterly snapshots, and stitched versus separate DuckDuckGo repositories. These checks evaluate the stability of classification, not the causal effect of migration.

3.7. Validity boundaries and permissible inference

The protocol supports statements such as "in the analyzed cases," "the observed trajectory is consistent with," or "the project documents an associated constraint." It does not support claims that SwiftUI improves performance, maintainability, quality, or team velocity; that coexistence reduces risk; or that deployment targets cause a strategy. No regression, hypothesis test, defect model, user study, interview, experiment, or productivity analysis is claimed. If the number of independent episodes is small, quantitative reporting will remain descriptive and will not be upgraded to inferential analysis.

4. Results: Analytical Outputs of the Protocol

The outputs in this section are results of conceptual operationalization and evidence synthesis. They are not results of the pending repository-wide extraction. Their purpose is to make the empirical phase falsifiable: every class has defining observations, a boundary, and a condition under which it must be rejected or relabeled.

4.1. Result 1: a mechanism-oriented strategy typology

The first result is an a priori typology of six configurations. It refines broad repository trajectories by requiring evidence about direction, scope, trigger, and the fate of UIKit within that scope. The analytical function of Table 2 is to prevent a classification based on the mere presence or growth of SwiftUI. Five rows are candidate migration or adoption strategies; the sixth is a contrast class that must not be counted as UIKit-to-SwiftUI migration.

Table 2. Mechanism-oriented typology of UIKit-to-SwiftUI configurations. Classification requires the joint configuration of seam, scope, trigger, and UIKit fate; labels are provisional until the frozen repositories are coded.

Configuration	Defining mechanism	Required longitudinal evidence	Boundary or falsifier
Replacement wave	Existing UIKit screens or flows are progressively rebuilt in SwiftUI through one or more hosting boundaries.	SwiftUI increases while UIKit/storyboard artifacts decline or are removed in the same declared scope.	If UIKit remains stable and SwiftUI serves only new features, relabel as addition or coexistence.
Feature-level strangler	A bounded screen or flow is routed to SwiftUI while adjacent UIKit responsibilities remain.	Repeated scope-local replacement episodes and an explicit route or hosting seam.	A single new SwiftUI screen without a replaced predecessor is greenfield addition, not strangling.
Modular SwiftUI island	A module develops predominantly in SwiftUI behind a stable interface to a UIKit application.	Module-local SwiftUI concentration, stable external seam, and no requirement for project-wide UIKit decline.	If the module was SwiftUI from inception and replaces nothing, classify it as modular adoption rather than migration.
Capability enclave	SwiftUI is introduced for a platform capability or extension without displacing the main UIKit UI.	Capability-linked addition, scope isolation, and stable UIKit outside the enclave.	It is not a replacement strategy unless an existing UIKit implementation in the same capability is removed.
Bounded coexistence	UIKit and SwiftUI retain explicit, durable responsibilities because of documented constraints or architectural boundaries.	A persistent division of responsibility plus public evidence or repeated repository states consistent with that boundary.	Incomplete migration cannot be ruled out when no boundary or constraint is documented.
SwiftUI-first fallback bridge	The baseline is SwiftUI; an isolated UIKit view or controller supplies a missing or retained capability through a representable bridge.	Low UIKit presence from inception and no prior UIKit implementation of the application's UI scope.	This is a negative-control configuration, not UIKit-to-SwiftUI migration.

This typology sharpens four distinctions. First, replacement is assessed within a functional scope rather than by project-wide framework absence. Second, a hosting seam does not determine strategy by itself: the same mechanism can support a temporary replacement boundary or a stable module interface. Third, capability-driven addition is separated from voluntary modernization. Fourth, UIKit-in-SwiftUI is not evidence of reverse or incomplete migration when SwiftUI was the baseline. The classification is therefore mechanistic rather than quantitative: artifact counts are necessary for trajectory evidence but insufficient for meaning.

4.2. Result 2: conditional mapping from constraint to seam and strategy

Official interoperability documentation establishes which seams are technically available [18, 19], while the public Firefox policy and selected repository records identify case-specific constraints [20, 22 - 25]. Their synthesis yields a conditional matrix rather than a ranking. The analytical function of Table 3 is to connect an observable boundary to a candidate strategy and a scope-specific exit or stop criterion. "Confidence" refers to the present source basis, not to a statistical estimate.

Table 3. Conditional mapping between observable project constraints, interoperability seams, and candidate strategies. The matrix summarizes source-supported propositions and does not identify a universally preferred migration path.

Observable condition or boundary	Seam and candidate strategy	Exit or stop criterion	Present evidence confidence
Existing UIKit screen or flow has a separable route and a known predecessor.	UIHostingController; feature-level strangler or replacement wave.	Within-scope UIKit implementation is removed or deliberately retained with a documented boundary; replacement tests cover the migrated responsibility.	Moderate: official seam plus prior trajectory and public case cues [1, 19, 22, 23].
A collection or table cell can adopt SwiftUI content while the container remains UIKit.	Hosting configuration; component-level migration or addition.	Cell responsibility is stable and state flow is explicit; no project-wide completion claim is made.	Mechanism-supported; prevalence untested [19].
A feature is organized as an independently routed module.	Coordinator/router boundary; modular SwiftUI island.	Module-level entry, ownership, tests, and rollback boundary are explicit; UIKit outside the module is out of scope.	Case-specific to moderate [22].
A platform capability or extension favors SwiftUI without replacing the main UI.	Extension boundary; capability enclave.	Capability remains isolated; success is not defined by decline of unrelated UIKit artifacts.	Case-specific, consistent with the DuckDuckGo trajectory reported in prior work [1, 24].
Minimum supported iOS version limits APIs used by current SwiftUI architecture.	Explicit framework boundary; bounded coexistence.	Reassess when the deployment target or API availability changes; until then, do not label stable UIKit responsibilities as failed migration.	Case-specific but explicitly documented by Firefox iOS [20].
A SwiftUI-first application requires a UIKit-only or retained system capability.	Representable protocols; fallback bridge.	Remove the bridge only when an adequate SwiftUI path exists and behavioral equivalence is verified.	Official mechanism plus negative-control case [18, 25].

The matrix imposes two discipline rules. A strategy is not selected solely because a seam is available, and an exit criterion must use the same scope as the entry condition. For example, a modular island can be complete even if the host application remains largely UIKit. Conversely, a project-level replacement claim cannot be supported by the completion of one SwiftUI flow. The matrix also exposes evidence asymmetry: deployment-target constraints are explicit in Firefox documentation, whereas rationales for historical episodes may be absent elsewhere. Such asymmetry must remain visible through confidence labels.

4.3. Result 3: diagnostic trajectory signatures

The third output defines the longitudinal signatures that the empirical phase must test. A replacement signature requires an increase in SwiftUI artifacts accompanied by decline or removal of UIKit or storyboard artifacts in the same functional scope. An addition signature permits SwiftUI growth with a stable or growing UIKit baseline. A bounded-coexistence signature combines a persistent mixed state with a documented or repeatedly observable division of responsibility. A SwiftUI-first fallback signature begins with low, isolated UIKit use and lacks an earlier UIKit implementation of the application scope.

These signatures are interpretive rules, not measured curves. No numeric trajectory is shown because monthly SwiftSyntax and XML extraction has not been executed. Once calculated, event-aligned plots must display raw normalized shares, validated episode markers, repository-specific observation windows, and separate lines for UIKit/storyboard and SwiftUI artifacts. Smoothing is not permitted unless reported as a secondary transformation, and missing months must not be interpolated. A classification that changes under reasonable file-versus-LOC or monthly-versus-quarterly sensitivity checks will be reported as unstable.

4.4. Result 4: completion is scope-specific and falsifiable

The synthesis produces a final planning principle: migration completion must be defined at the same level as the strategy. A replacement wave can use decline or removal of UIKit responsibilities within the nominated screens or flows; a feature strangler terminates when the predecessor responsibility has moved and the routing boundary no longer carries transitional duplication; a modular island uses module ownership, interface stability, testing, and rollback criteria; a capability enclave has no obligation to reduce UIKit elsewhere; bounded coexistence is reassessed when the documented constraint changes; and a fallback bridge ends only when the capability can be supplied without UIKit and behavioral equivalence is demonstrated.

This formulation makes the decision matrix falsifiable. If the repository shows SwiftUI growth but no same-scope predecessor, a replacement label fails. If a claimed bounded architecture has no stable division or public constraint, the analysis must acknowledge that incomplete migration cannot be ruled out. If UIKit exists from the first observable SwiftUI-first baseline only as a bridge, a historical migration claim fails. The framework therefore produces not a maturity score but a set of classification tests.

5. Discussion

5.1. Relation to prior iOS adoption research

The protocol extends, rather than contradicts, the four trajectories reported by Baresi et al. [1]. Their categories describe repository-level patterns of technology presence over time. The present framework asks what mechanism can produce a similar pattern. A hybrid curve, for example, may reflect a feature strangler, a modular island, a capability enclave, or bounded coexistence. These are not synonymous: they have different scopes, triggers, seams, and completion criteria. Likewise, “modest addition” becomes analytically stronger when the study can distinguish platform-capability adoption from an early stage of replacement.

The negative control also clarifies the boundary of the earlier categories. A SwiftUI-first application with a small UIKit wrapper can look “hybrid” at a snapshot. Historical reconstruction reveals that no UIKit application layer was replaced. Calling this configuration migration would conflate interoperability with technological transition. This distinction is theoretically useful because it makes the pre-state constitutive of the phenomenon: migration is a change in responsibility, not a count of co-occurring imports.

5.2. Migration theory at the UI-framework boundary

The framework is consistent with library-migration research that treats migration as a set of heterogeneous changes rather than a mechanical mapping [5, 6]. UI frameworks strengthen this requirement because the migrated object may be a view, a controller, a navigation flow, a module, or the application lifecycle. The same API bridge can serve different strategies depending on the scope and intended duration. By combining context, problem, solution, and consequence in the pattern tradition [8], the typology avoids turning names such as “strangler” or “island” into labels detached from observable criteria.

The analogy to architectural migration remains limited. Microservice studies emphasize multi-level change and long-running transition [7], but their organizational findings are not treated as evidence about iOS teams. What transfers is the analytical principle that durable coexistence can be an architecture rather than an unfinished stage. Even this interpretation requires caution. Public repositories rarely expose the complete roadmap; where a boundary is undocumented, the study can describe persistent coexistence but cannot always decide whether it is intentional or merely unresolved.

5.3. Practical implications as conditional propositions

For practitioners, the typology redirects planning from “UIKit or SwiftUI?” toward three prior questions: what responsibility is moving, what seam will carry the transition, and what observation would show that the nominated scope is complete. A team maintaining a UIKit application with a separable flow can evaluate a feature-level strangler without committing to project-wide replacement. A separately owned product module may justify a SwiftUI island if routing, state transfer, testing, and rollback remain explicit. A platform-specific extension can adopt SwiftUI without being counted as progress toward replacing the main UIKit UI. Conversely, a SwiftUI-first application can retain an isolated UIKit bridge without being described as a failed migration.

The deployment target deserves separate treatment because it is observable and time-dependent. Firefox iOS documents a relationship between its supported baseline, newer SwiftUI APIs, and selective use [20]. The appropriate inference is not that deployment targets universally cause bounded coexistence. It is that a stated compatibility constraint can explain why a stable boundary is rational in one case and can establish a date for reassessment. The decision matrix therefore maps conditions to plausible strategies; it does not calculate return on investment or rank frameworks.

Testing is included as an episode attribute but not as an outcome variable. A change that migrates a flow and updates relevant tests provides stronger evidence that responsibility moved than a source-file count alone. It does not prove that the strategy reduced defects. Similar restraint applies to file and line counts: they support temporal classification, not estimates of effort, functional weight, or maintainability. This distinction is essential if the framework is to remain useful without overstating what public repositories reveal.

5.4. Threats to validity

Construct validity

Framework imports and type declarations are imperfect proxies for runtime responsibility. Mixed imports, previews, generated files, test utilities, and target membership can create false events. Syntax-aware parsing, XML analysis, manual validation, and scope coding reduce this threat but cannot eliminate ambiguity in dynamic composition. File counts and lines of code also fail to capture functional weight; the protocol keeps them separate and treats classification as unstable when reasonable proxies disagree.

Internal validity

The design is observational. A migration event followed by a change in code structure does not establish that the framework caused that change. Public issues and pull requests may record only the rationale considered suitable for publication, and the absence of documentation does not imply the absence of a motive. Documents outside the event window are therefore used as current policy rather than retroactive explanation. No claim about quality, productivity, performance, or developer experience is made.

Conclusion validity

The number of independent validated episodes may be too small for inferential statistics, and episodes within one repository are not necessarily independent. The protocol consequently defaults to descriptive profiles and cross-case pattern logic. Inter-coder agreement will be reported only after coding; the prespecified threshold is a quality-control rule, not an anticipated result. A pattern supported by one case remains provisional unless an independent study or official mechanism provides additional support.

External validity

Open-source production applications are not a random sample of the App Store. They may differ from closed projects in architecture, governance, deployment targets, and the visibility of decision records. The purposive cases are chosen for contrast, not national or market representation. Findings can therefore be transferred analytically to projects with similar conditions, but their prevalence cannot be generalized statistically. The DuckDuckGo repository move adds a history-stitching threat that will be tested by analyzing the repositories jointly and separately.

Protocol status

The most important limitation is that the frozen repositories have not yet undergone the specified full extraction and coding procedure. Tables 2 and 3 are operational outputs derived from the checked source package, not validated frequency results. A completed empirical article must replace protocol language with provenance-linked episode counts, case timelines, agreement statistics, and sensitivity outcomes. Until then, claims that any configuration is common, effective, or observed a specified number of times would be unsupported.

6. Conclusion

UIKit-to-SwiftUI change cannot be classified reliably from framework coexistence at a single repository revision. The scientific problem is to identify when an existing UI responsibility moves, at what scope, through which seam, under what publicly observable constraint, and with what fate for the predecessor implementation. This article addresses that problem by defining a reproducible longitudinal protocol and an a priori mechanism-oriented typology for five open-source iOS cases.

The principal contribution is a set of falsifiable distinctions. Replacement requires same-scope UIKit decline or removal; SwiftUI growth without such decline is adoption; persistent mixed use can be treated as bounded coexistence only when its boundary is observable and incomplete migration cannot be silently excluded; and UIKit fallback in a SwiftUI-first application is interoperability rather than migration. The conditional matrix links these classifications to hosting, configuration, representable, routing, and module seams while preserving scope-specific exit criteria.

The framework is intended to discipline the empirical phase, not to predetermine it. Its conclusions are limited to methodological and conceptual claims because no repository-wide counts or causal outcomes have been calculated. The next step is to execute the frozen extraction protocol, publish episode-level provenance and sensitivity results, and test whether the proposed configurations recur beyond the selected open-source cases. Validation in closed industrial projects would then be needed to assess whether the same boundaries hold when internal roadmaps, effort data, and product outcomes are available.

Declarations

Author contributions. T.K.: Conceptualization, Methodology, Investigation, Formal analysis, Writing original draft, Writing review & editing, and Visualization. The author has read and approved the final version of the manuscript.

Funding. This research was independently conducted and received no external funding.

Ethics statement. Ethical approval is not applicable to the protocol as specified because it uses publicly accessible software repositories, technical documentation, and published literature and involves no human participants or private data.

Conflict of Interest. The author declares that there are no conflicts of interest related to the preparation and publication of this article.

References

1. Baresi L., Di Penta M., Quattrocchi G., Tamburri D. A. How Have iOS Development Technologies Changed over Time? A Study in Open-Source // Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems. 2024. P. 33 - 42. DOI: 10.1145/3647632.3647988.

2. Domínguez-Álvarez D., Gorla A., Caballero J. On the Usage of Programming Languages in the iOS Ecosystem // 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation. 2022. P. 176 - 180. DOI: 10.1109/SCAM55253.2022.00026.

3. Habchi S., Hecht G., Rouvoy R., Moha N. Code Smells in iOS Apps: How Do They Compare to Android? // 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems. 2017. P. 110 - 121. DOI: 10.1109/MOBILESoft.2017.11.

4. Wiertel P., Skublewska-Paszkowska M. Comparative Analysis of UIKit and SwiftUI Frameworks in iOS System // Journal of Computer Sciences Institute. 2021. Vol. 20. P. 170 - 174. DOI: 10.35784/jcsi.2662.

5. Islam M., Jha A. K., Akhmetov I., Nadi S. Characterizing Python Library Migrations // Proceedings of the ACM on Software Engineering. 2024. Vol. 1, FSE. P. 92 - 114. DOI: 10.1145/3643731.

6. He H., He R., Gu H., Zhou M. A Large-Scale Empirical Study on Java Library Migrations: Prevalence, Trends, and Rationales // Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021. DOI: 10.1145/3468264.3468571.

7. Ayas H. M., Leitner P., Hebig R. An Empirical Study of the Systemic and Technical Migration towards Microservices // Empirical Software Engineering. 2023. Vol. 28. Art. 85. DOI: 10.1007/s10664-023-10308-9.

8. Balalaie A., Heydarnoori A., Jamshidi P., Tamburri D. A., Lynn T. Microservices Migration Patterns // Software: Practice and Experience. 2018. Vol. 48, no. 11. P. 2019 - 2042. DOI: 10.1002/spe.2608.

9. Kalliamvakou E., Gousios G., Blincoe K., Singer L., German D. M., Damian D. The Promises and Perils of Mining GitHub // Proceedings of the 11th Working Conference on Mining Software Repositories. 2014. P. 92 - 101. DOI: 10.1145/2597073.2597074.

10. Kalliamvakou E., Gousios G., Blincoe K., Singer L., German D. M., Damian D. An In-Depth Study of the Promises and Perils of Mining GitHub // Empirical Software Engineering. 2016. Vol. 21, no. 5. P. 2035 - 2071. DOI: 10.1007/s10664-015-9393-5.

11. Munaiah N., Kroh S., Cabrey C., Nagappan M. Curating GitHub for Engineered Software Projects // Empirical Software Engineering. 2017. Vol. 22. P. 3219 - 3253. DOI: 10.1007/s10664-017-9512-6.

12. Petersen K., Vakkalanka S., Kuzniarz L. Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update // Information and Software Technology. 2015. Vol. 64. P. 1 - 18. DOI: 10.1016/j.infsof.2015.03.007.

13. Wohlin C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering // Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. 2014. Art. 38. DOI: 10.1145/2601248.2601268.

14. Tufano M., Palomba F., Bavota G., Oliveto R., Di Penta M., De Lucia A., et al. When and Why Your Code Starts to Smell Bad // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. 2015. P. 403 - 414. DOI: 10.1109/ICSE.2015.59.

15. Cruzes D. S., Dybå T. Recommended Steps for Thematic Synthesis in Software Engineering // 2011 International Symposium on Empirical Software Engineering and Measurement. 2011. P. 275 - 284. DOI: 10.1109/ESEM.2011.36.

16. Krippendorff K. Content Analysis: An Introduction to Its Methodology. 4th ed. Thousand Oaks: Sage, 2018.

17. Apple. Introducing SwiftUI [Electronic resource]. URL: https://developer.apple.com/tutorials/swiftui (accessed: 03.05.2026).

18. Apple. Interfacing with UIKit [Electronic resource]. URL: https://developer.apple.com/tutorials/swiftui/interfacing-with-uikit (accessed: 03.05.2026).

19. Apple. Use SwiftUI with UIKit. WWDC22, Session 10072 [Electronic resource]. URL: https://developer.apple.com/videos/play/wwdc2022/10072/ (accessed: 03.05.2026).

20. Mozilla Mobile. How and When to Use SwiftUI // Firefox iOS Wiki [Electronic resource]. URL: https://github.com/mozilla-mobile/firefox-ios/wiki/How-and-When-to-use-SwiftUI (accessed: 03.05.2026).

21. dkhamsing. Collaborative List of Open-Source iOS Apps [Electronic resource]. URL: https://github.com/dkhamsing/open-source-ios-apps (accessed: 03.05.2026).

22. WooCommerce. WooCommerce iOS App Repository [Electronic resource]. URL: https://github.com/woocommerce/woocommerce-ios (accessed: 03.05.2026).

23. SimpleLogin. Simple-Login-iOS Repository [Electronic resource]. URL: https://github.com/simple-login/Simple-Login-iOS (accessed: 03.05.2026).

24. DuckDuckGo. iOS Repository (Archived) [Electronic resource]. URL: https://github.com/duckduckgo/iOS; Apple Browsers Repository. URL: https://github.com/duckduckgo/apple-browsers (accessed: 03.05.2026).

25. gematik. E-Rezept-App-iOS Repository [Electronic resource]. URL: https://github.com/gematik/E-Rezept-App-iOS (accessed: 03.05.2026).