SwiftUI, UIKit, WKWebView, and Hybrid Game Runtimes

Modern iOS Architecture: Deconstructing the $3B MemeArcade

Hassan Uriostegui

Waken AI Labs

Modern iOS Architecture: Deconstructing the $3B MemeArcade

Modular Applications with SPM, SwiftUI and Hybrid Web

Hassan Uriostegui

Modern iOS Architecture: Deconstructing the $3B MemeArcade

Modular Applications with SPM, SwiftUI and Hybrid Web

Hassan Uriostegui

Published by Lulu.com

Editorial brand: Waken AI Labs

First edition · 2026

ISBN 978-1-105-01722-3

The ISBN barcode appears only on the print cover, in the Lulu-provided barcode area. Lulu.com is the ISBN imprint; Waken AI Labs is the editorial brand.

To Toufeeq Hussain, a dear friend I met more than a decade ago during our time at Viddy: thank you for your friendship, generosity, and steady belief in the possibilities that emerge when creative people build together. Our conversations, shared work, and years of encouragement have stayed with me far beyond any one project.

And to Vatsal Bhardwaj: thank you for welcoming me into Jabali AI and for the opportunity to learn alongside you at the frontier of AI and games. Your vision for expanding who can imagine, create, and publish interactive worlds has been deeply inspiring.

To you both, thank you for the trust, the collaboration, and the lessons that helped shape my understanding of AI gaming. This book carries forward a small part of that journey.

Contents

Generated from the rendered canonical manuscript.

About the Author

Hassan Uriostegui is a Mexican-American technologist and author whose work spans mobile video, visual effects, AI-human interaction, and practical software systems. Through Waken AI Labs, he explores human-centered technology and the architecture that lets creative products remain understandable as they grow.

Acknowledgements

The companion repositories are open-source learning material. The book text, diagrams, illustrations, and interior title artwork remain all-rights-reserved. Thanks to every engineer, creator, and collaborator who helped turn production questions into shareable architectural lessons.

A Note on ClineFlow

This book was produced with a journaled engineering workflow. The writing, evidence, source locks, build artifacts, validation results, and approval gates are recorded in ordinary project files so a future edition can be understood rather than merely regenerated. The journal is not part of the MemeArcade application and it is not a claim about the application's private implementation. It is the publication's own development memory.

ClineFlow is the optional tool used to keep that memory legible across long-running work. It is not required to read this book, explore the public companion repositories, or build an iOS application. Its practical idea is smaller: record the reason for a decision, the evidence that supported it, the command that checked it, and the next action while the context is still available.

For this project, that habit creates a clear boundary between several kinds of information:

- Public sources and their resolved revisions.
- Generalized architectural observations that may be published.
- Private product material that remains outside the manuscript.
- Generated print artifacts and the inputs that produced them.
- Human decisions that automation cannot make on behalf of an author or publisher.

The result is intentionally portable. A reader can use ClineFlow, another agent, or plain Markdown and Git; the useful practice is owning the record in the project itself. For a book about boundaries, that is more than process. It is a way to keep research claims, implementation examples, and publishing choices separate enough to review honestly.

If you adopt a similar workflow, begin with one small task. Write the intended outcome, one constraint, the evidence you plan to inspect, and the validation you expect to run. When the work ends, write what changed and what remains unapproved. Do not put secrets, private endpoints, user data, or unreviewed captures in a journal. A durable record should make collaboration safer, not make sensitive material easier to spread.

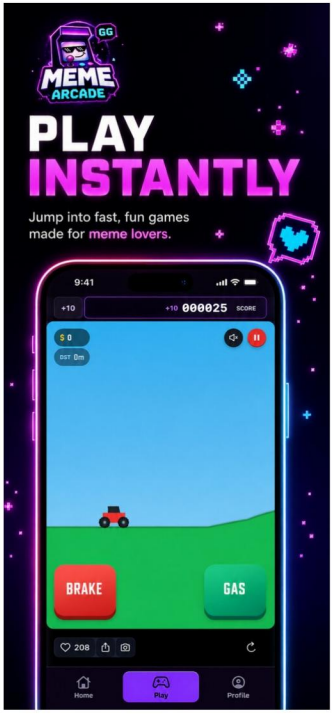
The companion repositories in this book remain their own direct public sources. The ClineFlow record simply explains how this edition connected them to the manuscript without turning the private MemeArcade case study into a source dump. That distinction lets the book teach reusable reasoning while respecting the limits of the application it studies.

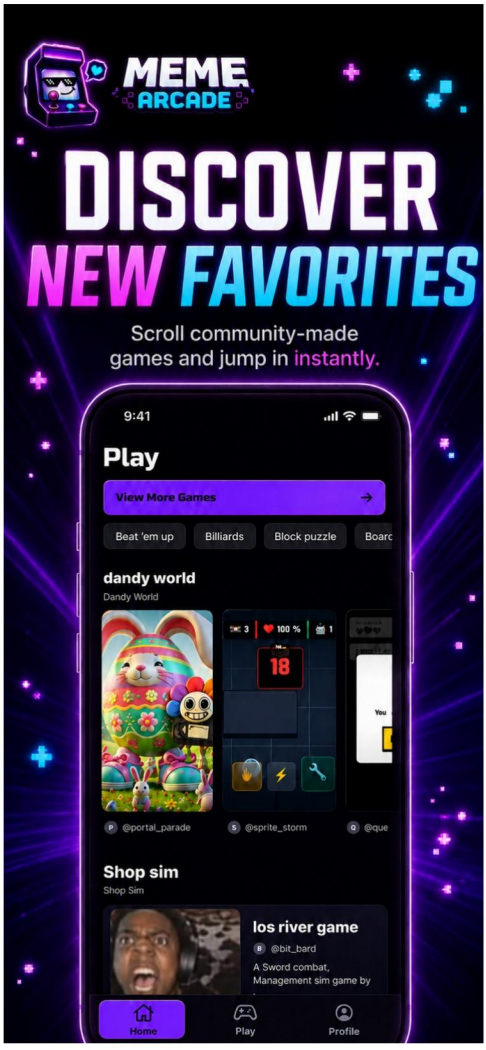
MemeArcade: AI Gaming Case Study

MemeArcade, the App, is the technical case study in this book: a native iOS shell for a scrollable, short-form interface that discovers and opens online games. This is an approved visual record of its public experience only.

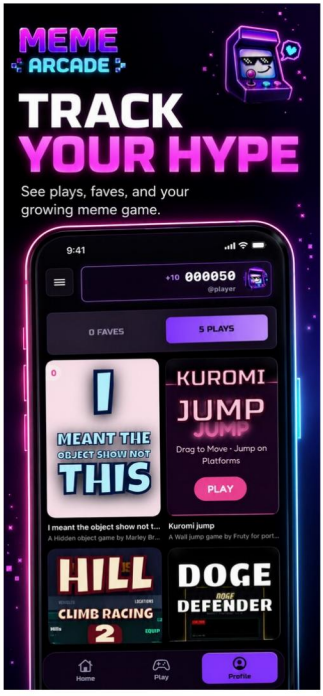
The publication approval covers only visible interface description; no private source, service, catalog rule, metric, or product logic is disclosed.

MemeArcade, the App, is the iOS case study in this book. The $3B framing refers to the broader AI-gaming market, not to the app's valuation, revenue, or financing.




MEME
ARCADE
DISCOVER
NEW FAVORITES
Scroll community-made
games and jump in instantly.
9:41
Play
View More Games
Beat 'em up
Billiards
Block puzzle
dandy world
Dandy World
@portal_parade
@sprite_storm
Shop sim
Shop Sim
los river game
@bit_bard
A Sword combat,
Management sim game by
Home
Play
Profile

Download MemeArcade



MemeArcade, the App

Scan to open the public App Store listing, or visit:

https://apps.apple.com/us/app/meme-arcade/id6 801 929 719

The $3B MemeArcade: What We Can—and Cannot—Deconstruct

An arcade cabinet has always been a compact agreement between many systems. A player sees a screen, a joystick, and a coin slot; behind that simple surface are hardware constraints, a game loop, distribution, operations, and the rules that make a session feel immediate. AI-native games and interactive apps have inherited the same problem at a much larger boundary. Their experience may be generated, refreshed remotely, or shaped by an online model, but a person still encounters it through a particular device, operating system, network, and moment of attention.

This book is about the iOS architecture that makes that encounter dependable. Its case study is **MemeArcade, the App**: an iOS product whose private implementation is not published here. Its title, **The $3B MemeArcade**, is deliberately broader. It names the emerging AI-gaming market in which products such as MemeArcade are being built. It is never a claim that MemeArcade, the App, has a $3 billion valuation, has raised that amount, or generates that amount of revenue.

The rounded figure has a bounded source. Mordor Intelligence projects a 2026 global market of USD 3.05 billion for generative AI in game environment and narrative design. Research and Markets estimates USD 3.4 billion for the broader AI-in-games market in the same year. These are different studies with different scopes; they corroborate that the category is multi-billion-dollar, but they must not be added together. In this book, **MemeArcade Market** means that category context; **MemeArcade, the App** means the technical product we study. Mordor Intelligence Research and Markets

That distinction matters because precise language is an architectural habit. Engineers learn early not to confuse a view with its model, a cache with its source of truth, or an observed request with a server guarantee. The same discipline applies to an

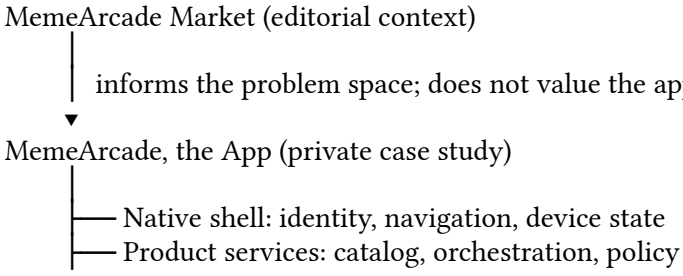
editorial case study. We will say what is public, what is generalized from an approved private observation, and what remains outside the book.

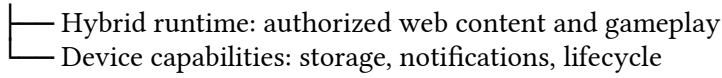
A market signal, not a product valuation

The AI-gaming category is no longer only a prediction. In June 2026, Axios reported that Sekai raised a $20 million Series A around text-prompt mini-app creation. In May 2026, Fortune reported $56 million in new funding for Astrocade, an AI-created game platform. These reports are useful as dated evidence that investors and builders are pursuing new forms of interactive creation. They do not prove that every AI-gaming product has the same architecture, audience, economics, or safety model. Axios, June 1, 2026 Fortune, May 5, 2026

Gizmo is another contextual signal. Business Insider reported in March 2026 that the team behind the AI mini-game app joined Meta's Superintelligence Labs; the report says the deal's financial terms were not disclosed. We use that story only to understand the competition for people, tools, and distribution in consumer AI. It is not evidence about MemeArcade, the App, and it does not authorize speculation about another company's implementation. Business Insider, March 5, 2026

The useful engineering question is therefore not, "Which company will win?" It is: **what must an iOS application own when playable, generative, and remote experiences arrive through a device?** The answer is not "everything." A hybrid product is healthier when it gives each runtime a legible job.





The lines in that diagram are responsibilities, not a claim about unpublished module names or private endpoints. They are the working vocabulary for the rest of this book.

The question beneath the technology

SwiftUI, Combine, Swift Package Manager, async/await, WebKit, and local notifications can look like separate topics. In a production app they converge on one question: **which component is allowed to make which decision, and for how long?**

Take a user who opens a game from a vertical feed. The native application may decide that a cell is the current candidate for activation. It may retain navigation ownership while the web runtime owns the game loop. A request service may fetch a catalog while a durable store remembers a harmless preference needed to restore the session. A notification scheduler may create a local reminder, but it cannot invent authorization that the user has not granted. Each boundary exists because every system fails differently.

This is why "native versus web" is a poor framing. The durable framing is **native orchestration and web participation**. The native shell is where device-bound responsibility belongs: lifecycle, permission, navigation, state restoration, accessibility, privacy, and the policy that determines whether a remote destination may load. The web runtime can be excellent at shipping playable content, iterating rapidly, and carrying interaction logic. Neither runtime should quietly impersonate the other.

For a junior developer, this is a practical rule: let the code that owns a fact be the code that changes it. If iOS owns which tab is selected, a web view should not silently replace that decision. If a remote game owns its rendering loop, the native cell should not attempt to recreate it. For a senior engineer, the same rule becomes an operational concern: trace ownership so that a timeout,

cancellation, process termination, or permission denial has a predictable recovery path.

The public components, and the private case study

The book does not ask readers to trust a private source dump. Instead, it moves between inspectable public examples and carefully bounded observations from MemeArcade, the App.

The public examples are deliberately smaller than the product:

- The SwiftUI and Combine article introduces reactive state and view updates.
- ios-framework gives us a concrete Swift Package Manager and tandem-app boundary.
- ios-storage provides a public persistence discussion, including state that survives a process lifetime.
- receipe-app is the small async/await laboratory: a request, a decode, a loading state, an error state, and a cancellation question.
- rezona-api is a public client-observation exercise, useful for reasoning about request models without inventing backend behavior.
- GamePlayer is the primary visible example of native paging paired with web gameplay.
- Pushscheduler isolates local notification scheduling as device infrastructure rather than a server feature.

MemeArcade, the App, supplies the composition problem: how such concerns meet in one product. We describe that composition through type-level responsibilities, diagrams, lifecycle narratives, and trade-offs. We do not publish full private files, credentials, internal endpoints, raw payloads, proprietary schemas, or unapproved business logic. A private code excerpt, production log, or product-specific diagram requires explicit human approval before it can appear in this manuscript.

This boundary is not a disclaimer appended after the interesting part. It creates better teaching material. A reader who learns to ask "what capability is crossing this boundary?" can apply that question to any project. A reader who memorizes a private class name cannot.

From Ultrakam to a hybrid application

My own path into this question began in a much more visibly native era. In "The Time Apple Featured My App at WWDC14", I describe Ultrakam and the experience of building for the iPhone platform at a time when a focused native application could carry nearly all of its behavior in one runtime.

That history does not make an argument from nostalgia. It makes the contrast clearer. The device is still where a user grants permission, receives interruption, restores a task, and judges whether an experience feels coherent. What changed is the number of collaborators behind one screen: package boundaries, concurrent tasks, remote content, a WebView process, caches, and optional re-engagement systems. The modern architecture challenge is to keep those collaborators from turning one user action into an untraceable chain of guesses.

The chapters ahead keep returning to a simple proposition:

> Native did not disappear. Native moved up the stack.

In a hybrid iOS application, native code may do less rendering than it once did, but it frequently carries more responsibility. It is the place where product intent meets operating-system reality.

What the Rezona chapter does—and does not do

One companion repository deserves special care. The network-observation chapter uses the public rezona-api repository to teach an authorized client-side method: capture traffic only when you have permission, remove credentials and personal data, model

what the client can actually observe, and label unknowns as unknowns.

That is not an investigation of Rezona as a company, and it is not an accusation of irregularity. A request seen from a client can establish that the client made a request and received a response. It cannot establish an unseen database schema, business practice, security posture, or internal decision. This distinction is especially important when using MITM tooling such as Proxyman. The tool can make a boundary visible; it does not grant authority to cross it.

The chapter will use a conservative vocabulary:

We can say	We cannot say without separate evidence
"The authorized client sent this redacted request shape."	"The backend stores data in this schema."
"The client receives this response field."	"The company uses this field for this business purpose."
"The client retries under this observed condition."	"The service guarantees this reliability behavior."

That restraint is not merely legal caution. It is sound systems engineering. Good debugging begins by separating observation from inference.

A working lens for the chapters ahead

Architecture is most useful when it turns a vague failure into a smaller question. "The game did not load" is not yet a question a team can answer. Was the selected item unavailable? Did the catalog request fail? Was an unsupported origin rejected by the navigation policy? Did the WebView process terminate? Did the game load but fail to return a native action? Was a notification scheduled for a state the user had already left? Those are distinct failures with different owners.

The book will repeatedly use four lenses to make that separation visible:

Lens	Question	Typical iOS responsibility	Typical remote responsibility
Ownership	Who may change this fact?	Navigation, persisted preferences, permission state	Play session and remote interaction state
Lifetime	How long should it survive?	View, scene, app, or local store lifetime	Response, web process, or server-session lifetime
Trust	What must be verified before use?	URL/origin policy, deep-link routing, notification input	Content delivery and remote authorization
Observation	What evidence can we record safely?	Timings, lifecycle events, redacted failures	Publicly documented interface behavior

Consider a simple choice: should a game begin loading when its cell becomes visible? A junior implementation might start immediately because visibility appears to equal intent. The lifecycle chapter will show why that assumption is too coarse. A cell may be briefly visible during a fast scroll; its task may need cancellation; its web process can consume memory even after attention has moved elsewhere. The actual design is a policy: identify a candidate, decide when it becomes primary, instrument the outcome, and release or reuse resources according to a measured budget.

The same progression applies to persistence. “Save state” is too broad to guide a production decision. A developer must ask which state is safe to retain on the device, when a write occurs, what happens if the app is killed during that write, and whether restoration will recreate a safe and coherent user path. The DataStore chapter uses a public example to teach those questions. It does not publish MemeArcade’s private schema.

This is also the reason this manuscript places trade-offs beside implementation detail. A framework pattern is not architecture merely because it compiles. It becomes architecture when its ownership, lifetime, trust, and observation costs are explicit. The public repositories give readers a chance to inspect compact answers; MemeArcade gives us the larger composition problem.

Reader activity: establish the evidence boundary

Before moving into code, open the original WWDC14 / Ultrakam account. Write two short columns in a notebook:

Publicly supported context	Not established by the article
The author’s historical experience shipping an iOS application	MemeArcade’s current internal architecture
A useful contrast between a native-first product and a modern hybrid product	Any market valuation or financing for MemeArcade, the App
Why device-level craft continues to matter	Any assertion about a third party’s backend or business practices

The expected observation is that author context is not production evidence. Carry that distinction into every reader activity. It is the discipline that lets this book make a real architectural argument without treating private implementation as public material.

How to read the book

Each chapter follows the same rhythm:

1. **Concept.** A junior-friendly model establishes the responsibility being discussed.
2. **Public implementation.** You open the original repository or article directly and inspect one bounded pattern.
3. **MemeArcade implementation.** A generalized diagram or lifecycle explains how the same responsibility composes inside the case study.
4. **Production trade-offs.** We examine the cost of the decision under cancellation, memory pressure, permissions, failure, and future change.
5. **Reader activity.** A QR code and companion link take you to the original public source—not to a duplicate web copy of this manuscript.

The GitHub Pages site is intentionally only a reader bridge. It gives you an activity and the expected observation, then sends you to the original public repository or article. The source material remains where it can be inspected in its own context; this book remains the architectural synthesis.

The first substantive chapters begin with state and modularity because an application cannot safely cross a hybrid boundary if it does not know who owns its state. From there we move through persistence, concurrency, network observation, WebView trust, GamePlayer, observability, local notifications, a complete session, and finally the reusable seams that survive the case study.

The goal is not to make every app look like MemeArcade. The goal is to make the next application easier to reason about: a system where every runtime has a job, every boundary has a policy, and every important user path can be explained from launch to return.

Explore the original public source

QR code for the original public source

Source: https://uriostegui.medium.com/the-time-apple-featured-my-app-at-wwdc14-a42dc4cd19bb

Try: Read the author-context essay and identify the product constraints that shaped its release.

Expected: A short note distinguishing author history from claims about MemeArcade, the App.

The Native Shell: Modular SwiftUI, Combine, and Product State

The first architectural decision in a hybrid application is not whether to use SwiftUI, UIKit, Combine, or WebKit. It is whether the product has a clear place to decide what is true at the device boundary. That place is the native shell.

The native shell is the part of an application that remains responsible when another runtime is loading, suspended, terminated, or simply wrong for a particular task. It owns the app-level route, the selected product surface, permissions, accessibility, restoration, and the policy through which a remote experience may participate. It does not need to own every pixel or game loop. Its job is orchestration: preserve a coherent application while the parts beneath it change.

In MemeArcade, the App, this idea is more useful than a framework slogan. A player may move through a native feed, activate a web-hosted game, return through a native action, receive a local reminder, and restore a safe product state later. If each runtime maintains an incompatible story about that sequence, the application feels brittle even when every individual screen looks correct.

The public SwiftUI and Combine tutorial is the compact starting point. Its important lesson is not a particular sample application's shape. It is the direction of change: observable state publishes an update; the view reads that state; a user action becomes an intent rather than a direct mutation scattered through the interface. In a larger hybrid product, that direction gives the native shell a way to remain understandable.

Start with a product state, not a screen tree

Many applications begin as a screen tree because that is how a user experiences them: launch screen, tab bar, feed, player, profile, settings. A screen tree is valuable, but it is not sufficient

architecture. It tells us where a view is visible, not who is allowed to make it visible or which state must survive when the view disappears.

Begin instead with a small product-state model. The names below are illustrative, not a reproduction of private MemeArcade types:

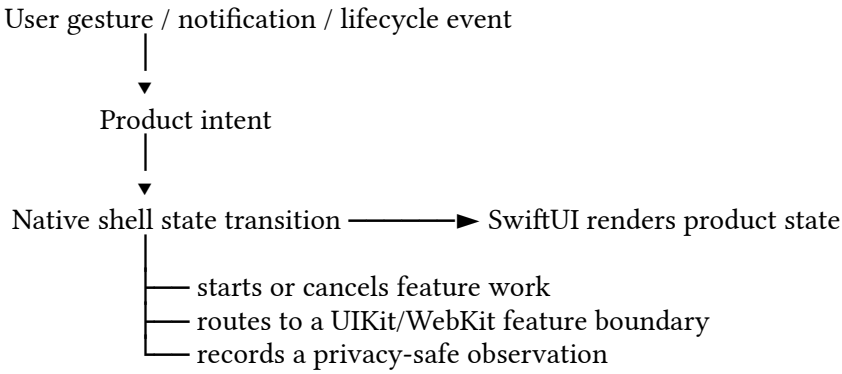
```
enum ProductRoute: Equatable {
    case home
    case player(GameReference)
    case settings
}

enum SessionStatus: Equatable {
    case restoring
    case ready
    case loading(GameReference)
    case active(GameReference)
    case recoverableFailure(UserFacingFailure)
}

@MainActor
final class AppModel: ObservableObject {
    @Published private(set) var route: ProductRoute = .home
    @Published private(set) var session: SessionStatus = .restoring

    func handle(_ intent: ProductIntent) { /* state transition */ }
}
```

The point is not to put all application code in AppModel. That would only create a new kind of monolith. The point is to give user-visible facts one accountable owner. A SwiftUI root can render route and session; feature services can perform work; a coordinator can translate an external event into an intent. But none of those layers should quietly mutate the visible product story behind the others' backs.



For a junior developer, this gives a simple debugging move: when a screen is wrong, locate the state that claims it should be right. For a senior developer, it defines a test seam. A route transition can be tested as a state change without booting a WebView or waiting for a network response.

Combine is a delivery mechanism, not the architecture

SwiftUI observes state through property wrappers and Combine-compatible publishers. It is tempting to call that reactivity itself an architecture. It is not. A publisher can deliver a correct value to the wrong owner just as efficiently as it delivers a correct value to the right one.

Use Combine where it fits the lifetime of a stream: a store change, a reachability signal, a notification authorization update, or an adapter around a legacy callback. Use structured concurrency where the work is a task with a beginning, cancellation point, and result. Both tools can feed the same native shell. The architectural decision is the boundary at which an event becomes a product intent.

For example, a remote catalog update might arrive through an async task. The feature service should decode and validate its domain data off the main actor when appropriate. The shell then receives a concise result: catalog ready, empty, refresh failed, or

stale data retained. SwiftUI sees a view-ready state rather than a half-decoded transport object. This avoids coupling a visible screen to incidental details such as URLSession errors or the shape of a server response.

Layer	May know about	Must not own
SwiftUI view	Renderable state and user intent	Network response parsing or WebView policy
App model / coordinator	Route, session state, feature intents	Game rendering loop or backend data store
Feature service	Domain work and cancellation	Global navigation decisions
Web runtime	Authorized playable interaction	Native tab selection, local permissions, or arbitrary routes

This table describes responsibility, not a mandatory folder structure. A small product may begin with fewer types. The value comes from maintaining the direction of authority as the product grows.

Native orchestration around a hybrid feature

Consider the moment a player taps an item in a feed. A simplistic implementation presents a WebView immediately and lets the page determine the rest. That works until ordinary product requirements arrive: back navigation, a full-screen presentation, an unavailable item, memory pressure, an unsupported navigation, a retry path, analytics with no private payloads, or a notification that wants to return to a safe destination.

The native shell should decide the product transition first. It can move from .ready to .loading(reference), present the hosting feature only when the route allows it, and move to .active(reference) after an approved readiness signal. The WebView remains a participant

with a restricted contract. It can render the game; it cannot become the unreviewed source of truth for the entire application.

This does not mean native code needs a heavy JavaScript bridge. In fact, a narrow contract is often safer. The GamePlayer chapters demonstrate the useful default: native paging and lifecycle management on one side, an isolated web experience on the other, and a deliberately limited interface between them. The exact bridge, if any, is a product decision that needs a trust model—not a convenience feature added to pass messages quickly.

The state-restoration test

An effective way to evaluate a shell is to imagine the process disappearing at the least convenient moment. The player has selected an item; the remote content has started loading; the app receives a memory warning; the operating system kills the process. What should the person see on relaunch?

The answer should not be "whatever object graph happens to survive." It should be an intentional restoration policy. The app may retain a harmless product route, a user preference, or an identifier that can be resolved again through current policy. It should not blindly restore a stale remote URL, replay an unvalidated action, or treat an old in-memory object as durable truth. Chapter 5 will make this distinction concrete with public persistence patterns.

This is another reason to make the shell state explicit. A model that can represent `restoring`, `ready`, and `recoverableFailure` tells the user what the app is doing. It also gives the team a recovery path that does not depend on the timing of a remote runtime.

Trade-offs: one model, many boundaries

Central product state has costs. A model that knows every feature will grow until it becomes a god object. The correction is not to make every feature autonomous. The correction is to separate feature state from product state.

Feature state answers questions such as “what did this feed load?” or “which stage is preparing?” Product state answers questions such as “which experience is the user in?” and “which capability is currently allowed?” A good shell composes these states instead of flattening them. The root does not decode a catalog; the catalog feature does not decide the global tab; the browser does not own restoration.

There is also a trade-off between immediacy and correctness. Directly setting a binding from a callback can feel fast during prototyping. Routing the callback through an intent may look ceremonial. But the intent boundary becomes valuable when the same event can originate from a gesture, a notification, a restored state, or a remote feature. It gives one place to validate, log, and change the rule.

Choice	Immediate benefit	Production cost	Default here
Views mutate shared state directly	Less initial code	Hidden transitions and fragile tests	Avoid for cross-feature state
One global observable object	Easy discovery	Monolithic ownership	Keep product state narrow
Feature-local models	Focused code	Requires explicit composition	Prefer for feature detail
Web runtime drives navigation	Fast prototype	Weak device policy and restoration	Keep navigation native-owned
Intent-based transitions	More types	Traceable, testable state changes	Prefer at boundaries

Generalized MemeArcade view

The private MemeArcade source is evidence only. The approved generalized view is that a SwiftUI application root coordinates product state and routes into a dedicated UIKit/WebKit pager feature; feed and catalog concerns remain separate responsibilities. This book does not publish its private module names, route identifiers, source files, endpoints, payloads, or implementation excerpts. Any more specific private claim or diagram requires explicit human approval.

That boundary is compatible with useful teaching. We can say that native orchestration makes lifecycle and trust policy explicit. We can show a type-level state model. We can compare the public SwiftUI/Combine example with the public GamePlayer code. We cannot claim that an unpublished class exists just because an analogous class would be convenient.

A small test plan for a large boundary

The shell earns its complexity when it gives the team useful tests. Start with state-transition tests that do not render a screen: a valid selection moves from `ready` to `loading`; a cancellation returns to a safe state; a rejected destination produces a recoverable user-facing outcome; restoration converts persisted intent into a current, validated route. These tests establish product behavior independently of a network or browser process.

Then add focused integration tests at the boundaries. A feature service should be able to prove that it cancels an obsolete task. A navigation policy should prove that it rejects an unapproved origin. A hosting controller should prove that leaving the player releases the active session according to its lifecycle contract. The test should name the ownership boundary, not merely assert that a button exists.

Finally, test the human path. Enable larger text, deny a permission, interrupt the app, return on a slow network, and perform a rapid selection change. A shell that is reactive only in a happy-path

demo will fail exactly where a hybrid product needs it most: at the moment several runtimes disagree about what should happen next.

The native shell is therefore a promise of graceful degradation. It cannot force a remote game to be available, and it should not pretend otherwise. It can keep navigation understandable, preserve only what is safe, present failure as a recoverable state, and make the next action obvious. Those are iOS responsibilities worth keeping at the top of the stack.

Reader activity: map the shell

Open the original SwiftUI and Combine article. Choose one value that flows from a publisher into a SwiftUI view. Then draw two arrows beside it:

1. Who is allowed to change the value?
2. Which product intent should be created if a user changes it?

The expected observation is that a reactive update and a product decision are related but not identical. Keep that distinction as we move into package seams, persistence, and asynchronous work. A native shell is not the place where every computation happens. It is the place where the product keeps a coherent promise to the person holding the device.

As a second pass, imagine the same change arriving from three places: a visible control, a restored session, and a local notification. If all three routes produce the same user-visible result, they should converge on the same intent or state transition. If they require different safety checks, make those checks explicit before the transition. This exercise often exposes accidental assumptions hidden in view callbacks—for example, that the presenting screen still exists, that a WebView is alive, or that a remote identifier is still valid.

Do not try to solve this by storing every object globally. Store the smallest validated information required to reconstruct a decision, then let the appropriate feature resolve its current detail. That is

how the shell stays both reactive and trustworthy as the number of screens, runtimes, and entry points grows.

One final practical rule: make loading visible as state, not as an absence of content. An empty screen may mean an empty catalog, an in-flight refresh, denied access, a cancelled task, or a true failure. Those conditions deserve different copy, different retry behavior, and different telemetry. When the shell represents them honestly, SwiftUI can render a helpful interface and a future engineer can understand why the product reached that point. When it hides them, the user and the debugging team are both left guessing.

The result is modest but powerful: the UI remains a truthful projection of product state, even while several asynchronous systems are doing work behind it.

That is the standard the rest of the architecture must protect in real production: clear ownership, bounded lifetimes, and a recoverable next action whenever reality changes.

Explore the original public source

QR code for the original public source

Source: https://uriostegui.medium.com/building-reactive-applications-with-swiftui-and-combine-a-tutorial-on-ios-app-simple3d-25d18eef7649

Try: Trace a reactive state change from publisher to SwiftUI view.

Expected: A state-ownership sketch.

Modularity as Architecture: Swift Package Manager and iOS Frameworks

Modularity is often introduced as tidiness: put related files together, make a framework, keep the project navigator pleasant. That is useful, but it misses the production reason to create a module. A module is a promise about **who may depend on whom**. It makes one capability available through a contract while keeping its implementation free to change behind that contract.

For a hybrid iOS application, that promise is not abstract. A player feature should not need to know how a notification schedule is persisted. A storage implementation should not choose the selected tab. A WebKit host should not become the only place where product navigation can happen. When those relationships are only conventions in one Xcode target, convenience slowly turns into coupling. Swift Package Manager gives the team a way to make the dependency direction visible and testable.

The public ios-framework repository is intentionally small enough to inspect in one sitting. It contains a top-level `Package.swift`, a `FrameworkLib` target, a test target, and a separate `DemoApp` that imports the library. Its value is not the size of its source. It is the shape of the experiment: build a package as a real product, then prove its integration from an app that uses it.

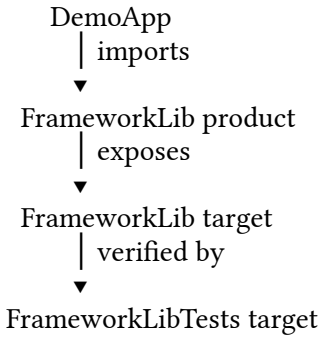
A package is a boundary, not a folder

Swift Package Manager reads `Package.swift` as a declaration of an external contract. The public example declares a library product named `FrameworkLib`, a source target with the same name, and a separate test target. It also declares the platform floor: iOS 15, macOS 12, and watchOS 8. Those lines answer real questions before anyone opens a view controller:

- What capability is being distributed?
- Which platforms are part of its compatibility promise?

- Which targets may compile against its public API?
- Where do tests live relative to the capability they protect?

The source target in the repository is deliberately minimal: it exports a public `Framework` type. The demo app imports `FrameworkLib` while retaining a tiny SwiftUI application root. This is a useful baseline because it separates two kinds of code that are often mixed together: the reusable capability and the app that demonstrates, configures, and composes that capability.



The arrow never points back upward. A reusable target may not import the demo application to discover its configuration or navigation state. Once that happens, the package is no longer a package; it is a hidden feature folder with extra build steps.

For a junior developer, this is a simple question to ask before extracting code: can another app use this feature by importing its public API without copying a screen, an app delegate, or a global singleton? For a senior engineer, the same question exposes product risk. If a capability cannot be exercised outside the only application that currently uses it, its release, testing, versioning, and migration costs are all entangled with the entire product.

The tandem-app pattern

The repository calls its separate app a “tandem app.” Apple documents the related local-package workflow: an Xcode application can depend on a package that is edited locally, so

changes in the package are reflected during app development without first publishing a release. The public README extends that idea with a configuration script that can rename the template and optionally prepare a repository for it. Apple: developing a package in tandem with an app Apple: editing a local package dependency

The tandem app is not a showcase for every possible use case. Its discipline is more useful: demonstrate only enough to prove that the package can be integrated. A demo that quietly accumulates product-specific networking, analytics, account flow, and styling is no longer a test of the package boundary. It becomes a second application that needs its own architecture.

In practice, a tandem app should answer four questions:

1. Can an app resolve and import the package using its intended integration path?
2. Can it construct the public entry point with ordinary configuration?
3. Can a visible, testable interaction prove the package is doing its job?
4. Can the package change internally without requiring the app to reach into private implementation detail?

Those questions are as relevant to an internal module as they are to an open-source SDK. The source may remain private while its boundary becomes clear.

Extract responsibilities, not screens

The wrong reason to make a package is “this folder has become large.” The better reason is “this responsibility has a coherent API, lifecycle, and test surface.” A feature can be large and still need to stay close to its host application if its behavior is inseparable from product navigation. A small service can deserve a package if it owns a stable capability such as encrypted local storage, notification scheduling, or a reusable WebView policy.

Use a responsibility map before you create targets:

Candidate capability	Useful public API	Inputs it may accept	Dependencies it must not own
Device storage	read/write validated values	explicit keys, values, migration policy	SwiftUI navigation and feed selection
Notification scheduling	schedule, cancel, refresh plans	permission status, validated destination	backend campaigns and product tab state
Hybrid player host	present an authorized game reference	approved URL/origin policy, lifecycle callbacks	app-wide account state and catalog fetching
Catalog client	fetch domain records	transport configuration, cancellation	view hierarchy and WebView lifecycle

This table does not dictate that every row must be its own package. It tests the idea. If the proposed capability requires five unrelated app models, an application coordinator, and a WebView instance merely to run a unit test, its boundary is not mature yet. If it accepts narrow domain inputs, has explicit outputs, and can be tested with substitutes for its environmental dependencies, extraction is likely to reduce rather than relocate complexity.

Dependency direction is an architectural test

Imagine a module named PlayerKit that hosts a remote game. It may depend on a small PlayerDomain model package and platform frameworks such as WebKit. The app may depend on PlayerKit and supply a route coordinator. But PlayerKit should not import AppShell simply to tell the app what to do next. Instead, it can

expose a callback, delegate, async stream, or value representing a product-neutral event. The application translates that event into its own route or analytics decision.

```
// Illustrative public boundary; not private MemeArcade code.
public protocol PlayerEventHandling: AnyObject {
    func playerDidRequestExit()
    func playerDidFail(_ failure: PlayerFailure)
}

public final class PlayerHost {
    public init(reference: GameReference,
                policy: NavigationPolicy,
                events: PlayerEventHandling) { /* ... */ }
}
```

This pattern protects both sides. The player package learns only that it has an event handler; the application decides whether “exit” means return to a feed, dismiss a modal, save a preference, or do nothing. Likewise, the app cannot reach into a web view and mutate its internal state merely because it happens to know the implementation type.

The same thinking applies to data flow. If a package exports a view model that includes transport-layer errors, database handles, and application route enums, it has leaked three boundaries through one API. A more durable contract exports domain values and failure categories that the caller can render or route according to its own product policy.

Tests are the price of a public contract

The public ios-framework package includes a FrameworkLibTests target even though the initial example has only a placeholder test. That shape matters. The moment a type becomes public, it becomes a compatibility promise. Tests should describe that promise in behavior, not in the package’s private arrangement of files.

For a storage package, test that an invalid record does not become a valid domain object. For a notification package, test that a denied permission produces a known outcome rather than silently scheduling nothing. For a player package, test the navigation policy with approved and rejected origins. For a service package, test cancellation and error mapping. These tests create a shared language for a capability and make refactoring less risky.

The tandem app complements unit tests. Unit tests prove a small contract; the app proves that Xcode can resolve the package, the host lifecycle is sound, and the public entry point is usable in a real UI. UI tests in the demo app should remain focused on that integration rather than recreate the package's entire test suite.

Versioning and the cost of extraction

Once code leaves an app target, it starts to acquire release economics. A package needs a versioning strategy, changelog discipline, compatibility policy, and an answer for local development versus tagged distribution. The ios-framework README points to a separate semantic-version tagging utility, which is a reminder that publishing is not a Package.swift feature. It is an agreement with users of the API.

Do not prematurely extract a moving implementation only because it could someday be reused. An internal package is often enough while the contract is forming. Tag and distribute it when a second consumer, a release boundary, or an independent ownership model makes compatibility meaningful. The important thing is to keep the app from depending on undeclared implementation detail; the repository location can follow later.

Choice	Benefit	Cost	Appropriate when
One app target	Fast exploration	Hidden coupling grows	A behavior is still being discovered

Choice	Benefit	Cost	Appropriate when
Internal package	Explicit seam, local development	More target and test maintenance	A capability has a stable owner
Published package	Reuse across repositories	Compatibility and release burden	Multiple consumers need a supported API
Shared source folder	Low ceremony	No meaningful contract	Rarely; only for temporary migration

A safe extraction sequence

The cleanest package boundaries are rarely designed perfectly on the first day. They are discovered through a sequence that reduces risk rather than multiplying it. Start by naming the responsibility inside the existing app target and moving only its pure models, protocols, and tests behind a local interface. This step reveals what the code was accidentally borrowing from the application: global configuration, presentation state, date providers, telemetry clients, or singleton storage.

Next, replace those hidden borrowings with explicit dependencies. A package should accept a policy or a protocol where it needs one, not import the app layer where the policy happens to be defined. Keep the app as the composition root: it selects concrete implementations and supplies configuration. The package remains responsible for the capability's behavior once those choices arrive.

Then create the package target and move the contract together with its tests. Build the host application after every small move. A failed build is useful feedback: it identifies a dependency that has not been honestly modeled yet. Resist the temptation to restore the old dependency through a back door merely to get a green build.

Finally, use a tandem app or an existing feature integration to exercise the public entry point. Ask whether an engineer unfamiliar with the package can tell how to initialize it from the public API alone. If the answer is no, improve the contract before publishing a version. Documentation and examples belong at the boundary because they are part of what users of the boundary must understand.

App target implementation
  | identify a cohesive responsibility
  ▼
Local protocol + behavior tests
  | make environmental dependencies explicit
  ▼
Package target + public API
  | prove real integration
  ▼
Tandem app / feature host
  | only then consider a versioned release
  ▼
Supported capability

This sequence makes modularity a change-management tool. It does not require a rewrite, and it does not confuse "moved code" with "stable contract." The package earns independence only after its inputs, outputs, and failure modes are explicit.

It also gives product teams a practical review question: if the app were replaced tomorrow, which part of this capability should still compile unchanged? The answer is the candidate module. Everything else is composition, policy, or a dependency that must be declared deliberately.

That clarity is the real product of a package boundary, long before reuse becomes a roadmap item.

Generalized MemeArcade view

MemeArcade, the App, is not presented as a reusable SDK and its source remains private. The approved lesson is narrower: organize

boundaries by responsibility, keep native product orchestration separate from specialized infrastructure, and make dependency direction explicit before extracting reusable components. The chapters on persistence, GamePlayer, and Pushscheduler show public examples of the kinds of capabilities that can eventually become stable seams.

We do not publish private package names, manifests, target graphs, source files, internal endpoints, or product-specific dependency rules. Any private implementation excerpt, exact module diagram, or claim about a particular private boundary requires explicit human approval. The reusable knowledge is the test: a capability should explain its inputs, outputs, lifetime, and forbidden dependencies without requiring the reader to see private source.

Reader activity: inspect a tandem boundary

Open ios-framework directly. Read Package.swift, then compare the FrameworkLib target with the DemoApp import. Draw the dependency direction and write one sentence for each side:

- The package promises _____ to an app.
- The app supplies _____ that the package must not own.

Then choose one conceptual MemeArcade capability—storage, player hosting, notification scheduling, or a catalog client—and make the same two-sentence contract. The expected observation is that a package boundary becomes useful only when it removes a dependency rather than hiding one. In the next chapter, persistence gives us a concrete device-level capability on which to apply this test.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/ios-framework

Companion article: https://uriostegui.medium.com/swiftspm-instant-spm-tandem-apps-e0b5ca3ef16c

Repository path: ios-framework/README.md

Try: Map a package boundary and the tandem demo-app contract.

Expected: A dependency-direction sketch.

Persistence as Device Infrastructure: DataStore in MemeArcade

Persistence is not a database feature added near the end of an application. It is the boundary between a fact that is true only while the process is alive and a fact the product is willing to carry into a future launch. In a hybrid iOS app, that boundary is especially consequential. A user can leave while a remote experience is loading, deny a permission after a preference was chosen, return from a notification, or reopen the app after the operating system has removed it from memory. The device needs a deliberate answer to one question: **what is safe and useful to remember?**

The public ios-storage repository, called DataStore in its README, is a compact teaching model. It connects an `ObservableObject` to a `Codable` state representation through a `DatastoreItem` protocol. On connection, the library restores stored state and observes a Combine publisher for future changes. It includes an encrypted path implemented with CryptoKit's AES-GCM APIs, an actor-based `Datastore`, a throttled save pipeline, and a background-task wrapper around archive work. Those choices make it a useful object lesson: persistence has a model contract, a concurrency contract, an encryption decision, and a recovery policy.

MemeArcade, the App, has application-state data-store integration. This chapter explains the decisions that a product must make—what may survive, how defaults recover, and why stale state must be revalidated—without exposing private schemas, keys, source, or implementation details. Any private schema, storage implementation, or excerpt requires explicit human approval.

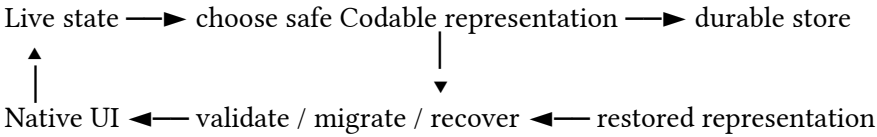
Start with the survival question

Not all state deserves persistence. It helps to sort a value by the question it answers after launch:

Kind of state	Example	Usually persist?	Reason
Durable preference	Sound setting, onboarding completion	Often	It is a user choice that remains meaningful
Safe restoration hint	Last selected tab or a stable content reference	Sometimes	It can reduce friction if revalidated
In-flight UI state	Half-open sheet, scroll velocity, loading spinner	Rarely	It belongs to the current process and scene
Remote authority	Session token, server permission, mutable catalog item	Not as app truth	It must be refreshed or validated through current policy
Sensitive material	Credentials, private keys, personal data	Only with explicit design	Encryption, access control, retention, and deletion requirements apply

The common failure is treating persistence as a snapshot of whatever objects happen to be available. That produces a haunted relaunch: an old route points to content that no longer exists, a stale response is rendered as current, or a permission-dependent feature tries to resume without checking permission. A durable store should preserve the minimum information needed to reconstruct a safe user decision, not the maximum amount of live memory.

For a player experience, a stored identifier might be a hint that the user was engaged with a particular item. On relaunch, the app still needs to fetch or validate the current catalog, apply navigation policy, and decide whether the item is available. The identifier is not an authorization token, a remote URL to load blindly, or a guarantee that the session can resume exactly where it stopped.



The important arrow on the way back is validation. Persistence is not a time machine; it is input arriving from an earlier version of the app.

The DataStore contract

The public `DatastoreItem` protocol makes the persistence decisions visible. A conforming model supplies a storage key, a `Codable` item type, a publisher, a way to read the current item, a default item, and a callback that applies a restored item. Encryption defaults to enabled through the protocol extension, although a conformer can choose otherwise.

That is a useful shape because it separates the observable application model from the serialized representation. A SwiftUI-facing model does not need to make every property public or serializable. It can produce a small state value whose compatibility can be reasoned about independently.

```
// A teaching sketch, not private MemeArcade code.
struct PersistedPlayerHint: Codable, Equatable {
    var version: Int
    var lastSelectedID: String?
    var didCompleteOnboarding: Bool
}

@MainActor
```

```
final class PlayerPreferences: ObservableObject {
    @Published private(set) var hint = PersistedPlayerHint(
        version: 1, lastSelectedID: nil, didCompleteOnboarding: false
    )

    func restore(_ saved: PersistedPlayerHint) {
        hint = migrateAndValidateLocally(saved)
    }
}
```

The version field is not decoration. A stored representation lives longer than individual releases, so a team needs a migration policy before a field changes meaning. A key such as model:v1, shown in the public README, is a simple way to separate incompatible storage generations. The library itself also prefixes item keys with a global version string. The lesson is broader than the exact mechanism: schema change needs an intentional path, not an accidental decode failure.

Restore first, then observe

The repository's connect method calls a restore-and-observe path. It restores a model, then subscribes to its storage publisher; subsequent values are throttled before archival. This sequence prevents one common error: observing the model first and immediately saving its default state over an existing record before restoration has completed.

The exact timing policy belongs to the product. A preference model can often be restored before a window becomes interactive. A feature state may need to wait until a user has authenticated or until a catalog is available. The native shell should represent this visibly: restoring, ready, or recoverable failure. A blank view does not tell a person whether data is loading, absent, corrupt, or unavailable by policy.

Throttling is also a product decision. The public implementation uses Combine's throttle on a background queue before it starts an archive task. That is sensible for a value that changes quickly, such

as a slider or an observable model with multiple updates in a row. But it creates a durability window: the last change may not have reached disk if the app is terminated before the throttle fires. A product that needs a strong "save now" guarantee needs an explicit flush or checkpoint operation, and it must decide what to do if that operation fails.

Save strategy	Benefit	Cost	Good fit
Save every change	Simple mental model	I/O churn, power cost	Rare, tiny and infrequent values
Throttle changes	Bounded write rate	Last change may be pending	Preferences and noncritical UI state
Explicit checkpoint	Known durability moment	More API and failure handling	Drafts, user confirmations, handoff points
Background-only save	Less foreground work	OS may end work early	Supplement, never the only guarantee

Encryption is a design, not a checkbox

The public DataStore code creates or retrieves a symmetric key through Keychain APIs and uses CryptoKit AES-GCM for the encrypted storage path. AES-GCM provides authenticated encryption: a successful open operation verifies that ciphertext has not been altered under that key. That is a meaningful protection for an appropriate local record, but it does not settle every security question.

Before persisting sensitive data, decide whether the data should be on the device at all; which Keychain accessibility class and access controls are appropriate; whether backups are acceptable; how a user can delete data; and what the app should do after a key, record, or schema failure. Do not describe a store as "secure"

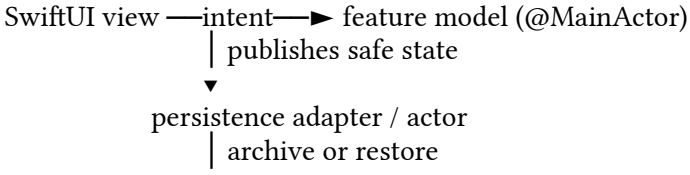
merely because it calls encryption APIs. Security comes from the whole lifecycle: minimization, key handling, access policy, error behavior, observability, and a tested recovery path.

The public implementation is useful precisely because it makes a trade-off visible. In its restore path, any failure in the encrypted load/decrypt/decode sequence falls back to the model's default. That keeps the app from being trapped by a corrupt or missing record. It can also hide the distinction between a brand-new store, an incompatible migration, damaged data, and a lost key. A production product should decide which of those conditions are safe to treat as a reset and which require a user-visible recovery state or privacy-safe diagnostic event.

Concurrency and the main-actor handoff

The Datastore type is declared as an actor, which is a clear attempt to serialize its internal mutable state. The public worker decodes and reads the store through that actor, then dispatches setStorageItem back to the main queue. This reflects a familiar SwiftUI constraint: a model observed by the UI should be mutated on the appropriate UI isolation context, while disk and crypto work should not block a render path.

The detail to preserve is not "always call DispatchQueue.main.async." Modern Swift code should prefer actor isolation that makes intent visible, such as a @MainActor UI model. The deeper rule is: one component owns serialized storage operations, and a separate UI-owned component applies a validated result. Avoid letting arbitrary views write storage directly. A view should express a user intent; the feature model should decide whether that intent changes state; the persistence adapter should then observe or receive the permitted durable representation.



This arrangement also makes cancellation and testing clearer. A storage adapter can be tested with an in-memory substitute. A feature model can be tested by injecting a store that returns a known value or throws a known error. The UI test need only prove that the resulting state is presented honestly.

Migration, deletion, and the right to start clean

Every persisted record eventually meets change. A field is renamed, a default becomes unsafe, an identifier is no longer meaningful, or a product changes what it is willing to keep. Planning for this does not require an elaborate database framework. It requires a versioned representation and a small set of explicit outcomes.

One approach is to decode the oldest compatible value, then migrate it in memory to the current domain state. Another is to use a new key or namespace for a breaking format and discard only the obsolete record. Both can be valid. What matters is that a future app build can tell the difference between “this record is from an old version,” “this record is damaged,” “the user requested deletion,” and “there was never a record.” If all four become a generic default without telemetry or product reasoning, a team loses the ability to diagnose real migration problems.

Deletion deserves equal attention. A sign-out, privacy request, account switch, or reset action should say exactly which device-bound records it removes and which records remain. Do not leave a local restoration hint attached to a person after an account transition merely because it was convenient to cache. Do not erase an independent accessibility preference just because an unrelated feature resets. Data minimization is not only about collecting less; it is about retaining and deleting with precise ownership.

For the hybrid player, a clean start is often the safest fallback. If a persisted selection cannot be reconciled with current catalog policy, return to an ordinary native surface and tell the user what

happened if context requires it. Do not force a replay of a remote session, preserve a stale browser process, or construct an unvalidated URL from a stored value. The product can restore continuity without pretending continuity is certainty.

Recovery condition	Safe default	Possible richer behavior
No record	Construct a known default	Offer onboarding or a first-run explanation
Old compatible record	Migrate to current model	Record a privacy-safe migration outcome
Damaged or undecryptable record	Reset only that record	Present recovery help if user work could be lost
User reset or sign-out	Delete owned records	Preserve independent device preferences where appropriate
Stale remote reference	Return to a native safe route	Re-resolve through current catalog and policy

The table is intentionally about outcomes rather than storage APIs. A durable store is successful when a person can recover from the past without the app mistaking historical data for present truth.

Generalized MemeArcade view

The approved generalized MemeArcade observation is that the application integrates state persistence at the device boundary. The publication-safe lesson is to persist only product state that remains meaningful after relaunch, restore it through a validated model, and keep private schemas private. The application root can coordinate restoration while individual features retain ownership

of their detail; a remote gameplay runtime cannot make itself durable merely by leaving a WebView alive.

This book does not publish MemeArcade's stored keys, record shapes, encryption strategy, migration history, account data, game identifiers, remote URLs, or private source. Any claim beyond this responsibility-level description requires explicit human approval. The public DataStore example is a teaching artifact, not proof that the production app uses the same data model or crypto configuration.

Reader activity: classify a persisted value

Open ios-storage directly. Find the public example's State, storageKey, storagePublisher, getStorageItemDefault, and setStorageItem methods. Then choose one value from a hypothetical game app—last selected tab, a user preference, a deep-link target, or a session token—and answer:

1. Is this value safe to keep after the process ends?
2. Is it a durable user choice, a restoration hint, or remote authority that must be refreshed?
3. What should happen if decoding, decrypting, or migration fails?

The expected observation is that persistence is a recovery contract. The data structure is only the beginning; the real architecture is the policy that decides what a future launch is allowed to believe.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/ios-storage

Companion article:
https://medium.com/@uriostegui/datastorage-encrypted-data-persistence-for-swiftui-f0187f9f2e40

Repository path: ios-storage/README.md

Try: Trace observable state through encrypted persistence.

Expected: A state restoration boundary.

Async/Await: From Recipe App to Production Orchestration

async and await make asynchronous code easier to read, but they do not automatically make it correct. A network request can still outlive the screen that asked for it. A response can still arrive after the user selected something else. A cache can still return stale data. A decoding failure can still become a blank screen because nobody decided how to represent it. Production concurrency begins when work has an owner, a lifetime, and a user-visible outcome.

The public receipe-app is a useful small laboratory. Its RecipesViewModel is @MainActor and publishes recipes, a selected recipe, an API error/message, and loading state. Its APIService is an actor that accepts a URLSession and a mock mode, exposes a high-level fetchRecipes() method, uses a generic get function for Decodable & Sendable responses, and maps decode/server failures into its own error cases. Its image cache is another actor: it returns a cached image when available or downloads one, then schedules a disk write in a detached background task. The repository's tests cover mocked valid, empty, and malformed recipe responses, plus cache behavior.

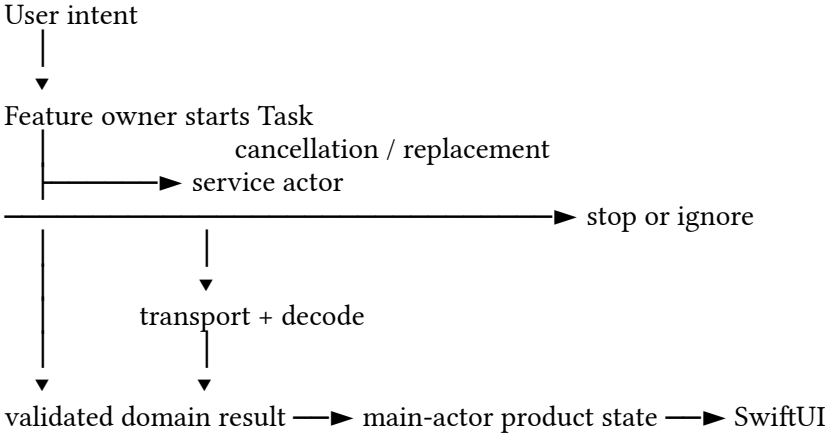
That is enough to teach the central production question: **when an asynchronous result arrives, is its owner still interested in it?** MemeArcade, the App, applies the same discipline to catalog/feed work and product services. The exact service types, endpoints, models, payloads, and implementation remain private. The reusable lesson is that network results must become validated product state through an explicit owner—not through whichever callback happens to finish last.

A task needs an owner

Every asynchronous operation should answer three questions before it starts:

1. Which feature or coordinator requested this work?
2. Under what condition should the work be cancelled or ignored?
3. How will success, empty data, cancellation, and failure appear in product state?

The owner may be a view model, an application coordinator, a refresh controller, or a domain service. It should not be a transient SwiftUI view body. Views can appear repeatedly; recomputation is not a durable task lifetime. A view expresses an intent such as "refresh" or "select this item." The owner starts the task, stores it if needed, and decides whether its result still applies.



The diagram separates a task from a result. Cancellation is cooperative: asking a task to cancel does not prove no work will finish. Therefore a feature owner should also identify a result. If a second refresh replaces the first, the first result should be ignored even if the underlying transport finishes after cancellation. A generation token, current request identifier, or identity comparison can make that rule explicit.

```
// Teaching sketch; not private MemeArcade code.
@MainActor
final class CatalogModel: ObservableObject {
    @Published private(set) var state: LoadState<[GameSummary]> =
```

```
.idle
    private var task: Task<Void, Never>?
    private var generation = 0

    func refresh() {
        task?.cancel()
        generation += 1
        let requestGeneration = generation
        state = .loading

        task = Task {
            do {
                let items = try await service.fetchCatalog()
                guard !Task.isCancelled, requestGeneration == generation
else { return }
                state = .loaded(items)
            } catch is CancellationError {
                // A replaced request is not a user-facing failure.
            } catch {
                guard requestGeneration == generation else { return }
                state = .failed(map(error))
            }
        }
    }
}
```

The exact LoadState design may vary, but the rule should not: do not let obsolete work overwrite newer user intent.

Isolate mutable services, not user decisions

The Recipe App's APIService actor holds its session and mocking configuration. Its generic fetch function decodes off the main path and converts errors into a smaller service failure vocabulary. This is an important split. A service owns mutable transport-related concerns; a @MainActor view model owns UI state. Neither needs to know every detail of the other.

Actors are not merely a replacement for serial queues. They express that only one task at a time may touch an actor's isolated state. Use them for shared caches, request coordination, token refresh, or mutable service configuration. Do not use an actor as a dumping ground for UI decisions. A catalog service can say "these records decoded" or "the request failed"; it should not decide whether the app dismisses a player or selects a tab.

The same separation protects a hybrid application. The service that obtains a catalog is not the same owner as the WebView that renders a selected game. The product shell decides whether a validated catalog item becomes a route. The player feature decides whether its own activation task is still relevant. The web runtime cannot elevate a transport response into product authority by itself.

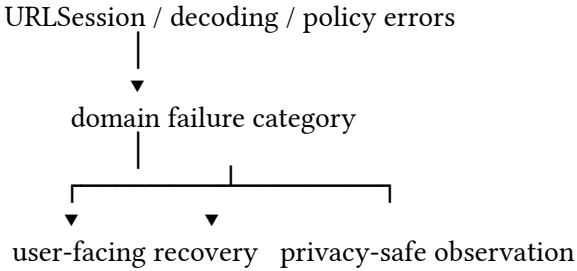
Responsibility	Good owner	Must not decide
Fetch and decode a resource	Service actor	App navigation or visible copy
Render loading/error state	Main-actor feature model	Transport implementation details
Choose whether a result still applies	Feature owner/coordinator	Server behavior it cannot observe
Cache bytes or images	Cache actor	Current product route
Validate remote data for a route	Product/feature policy	Private backend intent

Failure is a state, not a log line

The public recipe service distinguishes decoding failures, server failures, and an unknown URL construction failure. Its tests deliberately exercise valid, empty, and malformed responses. That is better than relying only on a happy-path live endpoint because it forces the application to say what each condition means.

An empty catalog is not necessarily an error. A malformed item may be skippable if the rest of the response remains useful, or it may indicate the response cannot be trusted. An offline failure may allow cached content. A policy rejection may require a hard stop. The mapping belongs near the product boundary, where a team can decide what the user sees and what privacy-safe evidence is recorded.

Avoid presenting raw NSError descriptions as product copy. They are unstable, often unhelpful, and can expose implementation context. Map low-level failures into a small set of user-facing states: retryable network problem, unavailable content, invalid response, permission required, or something went wrong. Preserve technical detail only in redacted logs where it is useful and authorized.



This keeps the UI honest without making it a diagnostic console.

Cache is an optimization, not product truth

The Recipe App's ImageCache uses an actor, a file-system directory, a URL-derived file name, and a detached background write. It returns an image immediately after a network load rather than waiting for the disk write. Its own README identifies the trade-off: the file system is the cache's source of truth, and a stronger production design could add an in-memory index, reconciliation, a size budget, and eviction policy.

That is an excellent teaching moment. A cache exists to improve latency or reduce work. It must be disposable. If a cache becomes

the only source from which an application can reconstruct critical product state, then it has become a database without the migration, consistency, or ownership design discussed in the previous chapter.

For a feed, cache catalog images or safe public assets when it helps the user experience. Bound the space, observe hit/miss behavior without recording sensitive URL data, and tolerate cache loss. Do not cache a remote permission, a privileged endpoint, or an unvalidated session as though it were durable authority. A cache miss should make the product slower, not incorrect.

Cache choice	Benefit	Risk to manage
In-memory cache	Fast reuse	Memory pressure and eviction
Disk cache	Survives process lifetime	Storage growth and stale files
Background write	Fast first render	Write may not finish before termination
URL-based key	Simple lookup	Normalize and avoid leaking sensitive identity
Persisted index	Better eviction/reconciliation	Becomes another durable schema

Concurrency across the native and web boundary

A common hybrid failure starts with a simple race. A player cell becomes visible, starts preparing remote content, then loses primary status during a fast scroll. If the activation task is not owned by the cell/controller lifecycle, its result can arrive late and attach a WebView to the wrong visual state. If a catalog refresh changes the item, a previous request may still supply stale metadata. If the app backgrounds, work may be cancelled or the process may disappear entirely.

Treat these as normal states, not exceptional accidents. Give each activation a reference and a lifecycle. Cancel tasks when their owner goes away. Check identity before applying completion. Re-enter the main actor only to update UI-owned state. Let the native shell decide a safe fallback when work cannot complete. Chapters 8 and 9 will apply this model to GamePlayer's native paging and WebView economics.

The performance trade-off is subtle. Starting tasks early can improve perceived responsiveness, but it also increases wasted work, memory, network activity, and the number of completions that must be discarded safely. Starting only after a user settles on a stage conserves resources but may delay activation. There is no universal answer; measure on target hardware and make cancellation correctness the prerequisite for any optimization such as prewarming.

Test the lifecycle, not just the response

The Recipe App's mocked responses make a crucial kind of test cheap: the application can see valid, empty, and malformed inputs without relying on a live service. Extend that idea to lifecycle. A production test plan should be able to create a slow response, start a replacement request, cancel the first owner, and prove that the older completion never changes visible state. It should test an empty result separately from a decode failure. It should confirm that a cache miss and a cache write failure do not turn into the same user-facing condition.

This is where dependency injection becomes practical rather than ceremonial. Inject a session, clock, cache, or service protocol only where a test needs control. The public Recipe App uses a session and mock mode in its service initializer. A larger application may inject a catalog client or an image loader protocol. The test does not need to know how the real endpoint works; it needs a controllable answer at the boundary.

```
// Illustrative test intent, not a copy of a private test suite.
func testReplacementRefreshIgnoresOlderCompletion() async {
```

```
    let client = ControlledCatalogClient()
    let model = await CatalogModel(service: client)

    await model.refresh()        // request A
    await model.refresh()        // request B replaces A
    await client.completeA(with: .oldItems)
    await client.completeB(with: .newItems)

    await expect(model.state).toEqual(.loaded(.newItems))
}
```

The shape matters more than the helper names. The test proves that the latest user intent wins. It should also prove the opposite when appropriate: a task that remains relevant should deliver a result even if the view temporarily rerenders. Do not equate every SwiftUI lifecycle event with cancellation. Tie task ownership to a documented feature or coordinator lifetime.

Observability closes the loop. Record a privacy-safe event when a request begins, completes, is cancelled, or maps to a user-visible failure category. Avoid logging full request bodies, user identifiers, or raw remote URLs unless there is a reviewed reason to do so. The goal is to distinguish “the service was slow” from “the user replaced the request” from “the response could not be accepted.” Those distinctions will matter when a hybrid session crosses native and web runtimes.

Async code becomes tractable when tests and telemetry share its vocabulary: owner, operation identity, cancellation, result category, and visible outcome. Without those terms, a task that “sometimes disappears” stays a mystery. With them, it becomes a state transition that can be reproduced and improved.

This vocabulary also makes reviews shorter and more precise. Instead of asking whether a new Task “looks safe,” ask what owns it, when it is cancelled, whether a late completion can still apply, and how the user recovers. Those questions work equally well for a one-screen recipe demo, a feed refresh, or a remote game

activation. They are the connective tissue between Swift concurrency syntax and reliable product behavior.

They keep concurrency review grounded in customer-visible consequences instead of framework folklore or accidental timing assumptions.

Generalized MemeArcade view

The approved generalized MemeArcade view is that catalog/feed and product service work use owned, cancellable asynchronous operations whose results are validated before they become visible state. This book does not publish private service names, endpoints, request shapes, payloads, cache keys, data models, task graphs, logging records, or performance measurements. Any specific implementation excerpt or operational claim requires explicit human approval.

The public Recipe App is not a miniature copy of MemeArcade. It is a smaller executable example of the same discipline: isolate mutable work, represent loading and failure honestly, test response variants, and avoid letting a late result override current intent.

Reader activity: follow one request and cancel it

Open receipe-app directly. Trace RecipesViewModel to APIService.fetchRecipes(), then into the generic request/decode path. Identify where mocked success, empty, and malformed input enter the service. Next, inspect ImageCache and write down what happens when a disk write has not completed before the next lookup.

Finally, add a cancellation point to the trace: imagine the recipe list is dismissed while its request is outstanding. Which owner should cancel or ignore the result, and which state should the user see if the feature is shown again? The expected observation is that async/await makes the path readable, but ownership and validation make it production-safe.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/receipe-app

Repository path: receipe-app/README.md

Try: Trace one request, cancellation, and failure state.

Expected: An async ownership sketch.

Observing the Network Boundary with Proxyman and MITM

Network inspection is a powerful debugging method because it turns a vague complaint—"the app did something strange"—into a sequence of observable events. It is also a method that can expose credentials, personal data, proprietary content, and state-changing controls. The correct starting point is therefore not a proxy configuration. It is authorization.

This chapter treats Proxyman and similar HTTPS-interception tools as instruments for observing traffic you are authorized to inspect: your own application, a local test environment, a permitted account, or a public research corpus whose method and limitations are documented. They are not tools for bypassing access controls, collecting another person's session, or converting client-visible behavior into claims about an unseen backend.

The public rezona-api repository is an unusually explicit example of the distinction. It preserves an evidence trail for observed API responses and documents a collector that checkpoints raw captures locally, aggregates public metadata, validates selection/order/identity, and keeps raw captures out of Git. Its README labels historical platform claims as attributed rather than independently verified; it treats observed IDs as a lower bound, describes an overlap calculation as exploratory rather than an inventory estimate, and limits state-changing or account-coupled routes to authorized research. That restraint is the lesson—not an allegation about Rezona as a company.

MemeArcade, the App, is a private technical case study. This book does not publish its network traffic, private endpoints, credentials, cookies, headers, payloads, or backend inferences. Any private production capture or excerpt requires explicit human approval. The method below is reusable precisely because it teaches a boundary rather than a target.

Observation is not inference

When a client sends a request and receives a response, the client can establish a limited set of facts: the request was attempted from that client context; a particular response was returned at that time; a visible field had a particular value; and a retry, redirect, or failure was observable. It cannot establish how a provider stores data, which internal service made a decision, whether an unseen endpoint exists, or why a business chose a behavior.

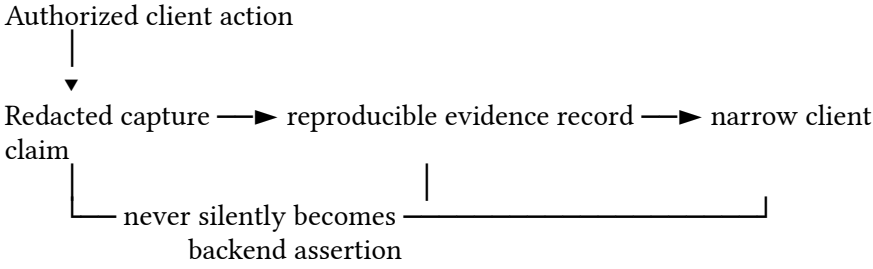
Keep an evidence ladder beside every capture:

Level	What the record supports	What it does not support
Direct observation	A redacted request/response behavior from an authorized client	Backend architecture or business intent
Reproducible collection	A lower bound within the documented query and time window	A complete platform census
Client artifact	A visible parameter, route, or data shape	A server guarantee or implementation
Inference	A clearly labeled hypothesis to test	A factual conclusion without independent evidence

This language makes teams better debuggers. It also protects readers from a common error in API analysis: reporting an implementation guess with the confidence of a measurement.

For example, a paginated search can show that the inspected client received a bounded page of results for a chosen query. If many queries reach a page cap, that tells us the recorded window is incomplete. It does not tell us how many total records the provider

has. The public Rezona repository documents exactly this kind of limitation: its saved windows supply reproducible observations, but capped and dependent query choices prevent them from proving a full inventory.



Set the scope before opening a proxy

Before starting Proxyman, write down the target, account, environment, purpose, and retention rule. If you cannot state why you are allowed to inspect a request, do not inspect it. For a product team, the safe default is a dedicated development or test account, a non-production environment, a narrow reproduction case, and a clear deletion plan for captures.

The scope document should answer:

1. **Authorization:** Who owns the client, account, or environment, and what permission applies?
2. **Purpose:** Which user-visible defect, contract, or performance question is being investigated?
3. **Data minimization:** Which headers, bodies, identifiers, and images must be redacted or excluded?
4. **Retention:** Where may an encrypted local capture live, how long, and who may access it?
5. **Publication:** What derived facts may appear in a ticket, repository, or book, and who approves them?

HTTPS interception requires a locally trusted certificate. Treat that certificate as a controlled diagnostic tool, not a permanent device

configuration. Use a test device or simulator where possible, remove the trust configuration when the task ends, and never ask a customer or untrained colleague to install a certificate merely to satisfy an investigation. Certificate pinning and other transport protections may intentionally prevent interception; do not weaken a production security control without explicit authorization and a controlled engineering plan.

Capture a minimum viable trace

The first capture should be small. Reproduce one action once: open a known screen, trigger one refresh, select one item, or perform one permitted test operation. Filter the proxy to the relevant host and time window. Save only what is necessary to explain the behavior.

Then normalize the observation into a client model. Do not paste a raw capture into a design document. A useful model removes secrets and irrelevant values while preserving structure:

```
Operation: catalog search (authorized test environment)
Input:    query, page token, bounded page size
Output:   records[], next-page indicator, response status
Observed: retry after transient transport failure
Unknown:  server ranking, storage, identity rules, total inventory
```

This form teaches a team to separate the request contract it needs from the backend story it does not know. It is also safer to review. No bearer token, full cookie, personal identifier, signed URL, or raw payload needs to appear in the model.

The next step is repetition. Change one input at a time. Does an empty query behave differently from an absent query? Is pagination a cursor or page number? Does a malformed client request receive a stable validation response? Which visible field identifies a record across two responses? Write the answer as "observed under these conditions," not "the API always." External services evolve; a capture records a moment.

From a captured request to a client abstraction

An observation becomes architecture when it gives the application a safer interface. Suppose an authorized inspection shows a catalog operation returning a list, a next-page hint, and an error status. The application should not distribute raw proxy-derived dictionaries through its UI. It should create a narrow client abstraction that owns transport, decoding, validation, and error mapping.

```
// Teaching sketch; no private endpoint or payload is implied.
protocol CatalogClient: Sendable {
  func search(_ query: CatalogQuery) async throws -> CatalogPage
}

struct CatalogPage: Sendable, Equatable {
  var items: [CatalogItem]
  var continuation: CatalogContinuation?
}
```

This boundary has several advantages. Tests can provide a fixed CatalogPage without a proxy or network. The product layer sees validated domain values rather than headers and JSON. A transport change can stay in the client adapter. And a public book can teach the contract without distributing an internal URL, proprietary query syntax, or sensitive response fields.

The abstraction should not erase relevant failure distinctions. A CatalogClient might expose an unavailable service, an invalid response, a policy rejection, or cancellation as distinct categories. The main-actor feature model then decides whether it can keep cached content, offer retry, or return the user to a safe route. This continues the ownership model from Chapter 6.

Responsible collection and reproducibility

The Rezona repository's collection method contains several practices worth carrying forward: bounded concurrency, retries with limits, checkpoints after saved responses, deduplicated identity handling, a manifest/checksum for the published

aggregate, and tests that validate the collector and aggregation logic. Those choices do not grant permission to collect arbitrary data. They make an authorized, bounded collection auditable.

Reproducibility has an ethical dimension. If a public result depends on raw captures that cannot be shared because they include credentials or sensitive data, publish a description of the selection method, redaction policy, aggregate outputs, and limits—not the secrets. The repository's approach of keeping raw search/detail payloads local and Git-ignored while publishing a deduplicated metadata corpus and provenance shape is a useful pattern. It allows someone to examine what the public artifact claims without treating raw access as a publishing requirement.

Be explicit about a lower bound. A collector that examines 100 results for each selected search term can accurately say how many unique records it observed in those windows. It cannot claim it discovered every record matching every possible term. This is a general rule for network research: the measurement frame is part of the result.

Practice	Why it helps	Boundary to preserve
Bounded concurrency/retries	Limits load and makes failures visible	Do not turn retries into uncontrolled scraping
Checkpoints	Preserves a reproducible sequence	Store them only where access is authorized
Deduplication	Avoids counting the same record twice	Define identity and version semantics clearly
Manifest/checksum	Connects published artifact to a build	Does not prove source completeness

Practice	Why it helps	Boundary to preserve
Redacted provenance	Explains selection	Must not disclose secrets or personal data

Proxyman as a product-debugging tool

A product engineer often needs less than a research collector. Proxyman can answer practical questions quickly: did the app request a resource once or repeatedly; did a redirect change the host; which status class followed a user action; was a cache header present; did a response arrive after the feature that initiated it had been cancelled? Use it alongside device logs and lifecycle traces, not as the sole source of truth.

For a hybrid app, correlate a redacted request with a native event such as “catalog refresh began,” “candidate became primary,” or “player left foreground.” Do not log arbitrary content or user identity merely to make correlation easy. A generated operation identifier can link the native event to a permitted client trace without copying sensitive payloads into analytics.

If the observation exposes a vulnerability, stop expanding the capture. Record the minimum necessary information through the appropriate responsible-disclosure or internal security channel. Do not turn a bug report into a public reproduction guide that enables abuse. The same applies to unexpected credentials or third-party personal data: cease collection, protect the material, and escalate through the owner who can determine retention and disclosure.

A review checklist before sharing evidence

Before a network observation leaves the device on which it was made, review it as though it were production data—because it may be. Remove authorization headers, cookies, access tokens, signatures, personal identifiers, device identifiers, full URLs that include secrets, images, and bodies whose contents are not

necessary for the claim. Replace values with typed placeholders such as <redacted-token> or <authorized-test-account> rather than inventing realistic-looking data.

Then test the wording of the claim. A strong sentence names the frame: “In a permitted test session on this date, the client received a page-shaped response after this redacted input.” A weak sentence turns that into “the service has this database” or “the company does this for every user.” The first can be reproduced and revised; the second overstates what the artifact knows.

Finally, consider the audience. A local engineering ticket may need more detail than a public teaching artifact. A book needs less than both. Publish the smallest representation that supports the lesson: a diagram, a type-level contract, a count with its method, or a redacted failure taxonomy. Keep sensitive operational material in the reviewed system that is authorized to hold it, not in a repository or QR destination.

This review is not bureaucracy around useful work. It is how a team preserves the trust that lets it keep observing and improving complex systems.

It also improves engineering velocity. A redacted, versioned request model can be read by an iOS developer, a backend owner, a security reviewer, and a product manager without asking each person to interpret a packet trace. When behavior changes, update the model and its evidence boundary. The discussion stays about a contract and a user outcome, rather than becoming a debate over whose tool captured the most bytes. That is the difference between inspection as a one-off trick and inspection as a maintainable engineering practice.

It gives future readers a durable method they can apply responsibly in their own authorized environments, regardless of the service behind the client.

Generalized MemeArcade view

The approved MemeArcade lesson is that remote content and product services require narrow, validated client contracts, while the native application retains trust and routing policy. No private network capture, API route, header, credential, cookie, request body, response body, endpoint, backend claim, or operational metric is approved for publication. Any private production excerpt requires explicit human approval.

The public Rezona repository is treated as an authorized, documented observation case study. It does not establish irregularity, wrongdoing, actual inventory, provider workflow, costs, or internal architecture for Rezona or any other company. Its value here is methodological: define the evidence frame, preserve provenance, redact aggressively, and report the limits with the same care as the findings.

Reader activity: write an evidence-safe request model

Open rezona-api directly. Read its disclosure, method, and API-reconstruction sections before looking at its aggregate data. Choose a documented read-only operation and create a four-line model: input, output, observed behavior, and unknowns. Do not copy a token, cookie, raw capture, personal record, or state-changing route.

Then label each sentence in your model as **observed**, **reproducible**, or **inferred**. The expected observation is that a client can support a useful engineering contract while leaving many backend facts unknown. That discipline makes proxy tooling valuable in a production iOS workflow without turning it into a license to overclaim.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/rezona-api

Repository path: rezona-api/README.md

Try: Separate observable client behavior from backend inference.

Expected: A redacted request model.

Two Runtimes, One Application: WebView as a Trust Boundary

WKWebView is not a decorative view. It is a second runtime inside an iOS application: it loads remote documents, executes JavaScript, manages browser storage, follows navigations, decodes media, and can be terminated or disrupted independently of the native process. A hybrid product becomes reliable only when it treats that runtime as a trust boundary.

The boundary is not an argument against web content. A remote game can iterate quickly, bring its own rendering and interaction model, and make a catalog of experiences practical without rebuilding every game as native UI. The native app contributes a different kind of value: it owns the device-level policy around the game. It decides when a page is active, which navigation is permitted, whether data is retained, how a failure appears, and which native capabilities a remote page may request.

The public GamePlayer repository makes those choices inspectable. Its MAWebViewFactory creates each view with a non-persistent WKWebsiteDataStore, JavaScript and inline/autoplay media enabled for gameplay, and a navigation delegate. MAWebNavigationPolicy allows only HTTPS URLs with a host; tests reject HTTP, file:, data:, javascript:, custom schemes, and invalid URLs. The player cell applies the policy to both navigation actions and responses, loads only when a page becomes primary, and exposes native action controls without installing a JavaScript bridge or injecting native script. Its README also records the critical trade-off: allowing any HTTPS host is flexible for CDNs and cross-domain assets, but a curated catalog should move toward an explicit host allowlist.

MemeArcade, the App, is described only through this generalized boundary. Its private source, URLs, policies, bridge decisions, and remote content are not published. Any private implementation excerpt or diagram beyond approved responsibility-level material requires explicit human approval.

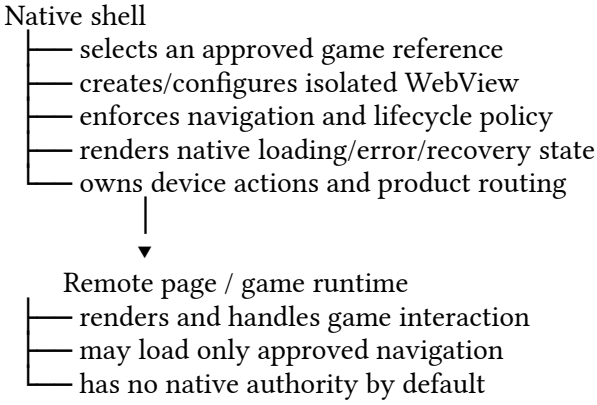
Name the two runtimes

The easiest mistake is to call the WebView "part of the app" and stop there. It is part of the user experience, but it does not automatically receive the app's authority. The native runtime and web runtime should have distinct capability sets.

Capability	Native shell	Remote web content	Default posture
App navigation and presentation	Owns	Requests only through narrow contract	Native-owned
Game rendering and input	Hosts/ contains	Owns inside authorized page	Web-owned
Device permissions	Requests and explains	No implicit authority	Native-owned
Cookies/web storage	Configures retention	Uses available web context	Minimize/ ephemeral when appropriate
Deep links and external URLs	Validates/ routs	Cannot silently open arbitrary schemes	Native policy
Product state restoration	Validates durable hints	Cannot assume a web process survives	Native-owned
Native actions	Implements	No access by default	Add only reviewed capabilities

This division is a product decision. A game may legitimately need JavaScript, audio, touch input, and cross-origin assets. It does not

therefore need arbitrary native routing, a way to invoke app features, or persistent cookies across every player session. The capability should be granted because a documented user experience requires it, not because a bridge is easy to add.



Navigation policy is a gate, not a URL check in one place

GamePlayer's public policy is intentionally simple: require a nonempty HTTPS host. It applies the decision before a navigation starts and again when a navigation response arrives. Checking both matters because a request can redirect, a document can navigate, and an initially valid-looking action can lead somewhere a product did not intend to host.

The rule rejects several dangerous or inappropriate categories by default:

- `http` lacks the transport security requirement of HTTPS.
- `file` would point into a local-file context rather than remote approved content.
- `data` and `javascript` are executable/document schemes that should not be treated like a game URL.
- custom schemes can jump to a different app or capability and need their own explicit routing policy.

- missing or malformed URLs should fail closed, not be coerced into a request.

This is not a claim that HTTPS alone makes content trustworthy. HTTPS protects transport to the requested host; it does not evaluate a page's behavior, its third-party dependencies, or the appropriateness of every redirect. A mature catalog adds a source-of-truth rule: validated catalog data, a controlled provider allowlist, signed configuration, or another product-specific approval path.

Policy level	Example rule	Benefit	Remaining risk
Scheme-only	HTTPS required	Blocks obvious unsafe/local schemes	Any HTTPS host may still be reached
Host allowlist	Hosts must match curated providers	Narrows remote surface	CDN/asset migration needs maintenance
Path/origin rules	Specific origins and paths allowed	Strongest predictable contract	Can be brittle if content evolves
User-mediated external route	Open external destination only after clear action	Preserves app context	Requires a deliberate UX path

Choose the narrowest rule that supports the product. Do not silently widen it after a game fails to load. First determine whether the game is intended to use a new host, whether that host is under an acceptable provider relationship, and whether the user experience needs a visible external transition rather than an in-place navigation.

Ephemeral data is a product choice

GamePlayer configures .nonPersistent() website data storage. In practical terms, the WebView's cookies and web storage do not accumulate as a long-lived shared browser profile across player sessions. That is a sensible default for a feed of independent remote games: a reused cell should not silently carry browsing identity or storage from one game into another.

Ephemeral storage has costs. A game may reload more often, lose an expected web-session preference, or require a new sign-in flow if the product legitimately supports web accounts. Persistent storage can improve continuity, but it creates a stronger data-retention obligation: define which origin shares the store, what it may retain, how it is cleared, and what a user expects after sign-out or account switch. There is no "free" default. There is only a storage policy that the product can explain.

The same caution applies to cache configuration. GamePlayer's player cell uses a reload-ignoring-local-cache request when it activates an item, while the app separately uses an avatar cache with URL, MIME, and size validation. These are different artifacts with different failure modes. A cache for a public image is not permission to persist a remote game session. Keep the purpose and retention of each store separate.

JavaScript and bridges: grant the minimum capability

Interactive games commonly require JavaScript and media playback. The public factory enables both because disabling them would make many playable web experiences nonfunctional. That grant is bounded by the navigation and data-store rules around it. It does not imply that native code must inject script, expose WKScriptMessageHandler, or provide an arbitrary command channel to the page.

GamePlayer's public security model chooses no JavaScript bridge and no native script injection. This is a strong default for a generic remote-game host because every bridge message becomes an input-validation and versioning problem. If a product needs a bridge, define it as a small protocol rather than a bag of string commands:

```
Web event:    { version, type, opaqueSessionReference }
Native checks: origin, schema, version, rate limit, active-session identity
Native result: a product-neutral event, never direct arbitrary method
execution
```

The native side should reject unknown types, impossible state transitions, malformed values, messages from a non-primary session, and events that do not originate from the expected content context. It should not accept a web-supplied URL and load it, treat a web-supplied account identifier as authenticated truth, or allow a page to invoke privileged device actions. Keep bridge semantics observable with redacted event categories, and maintain compatibility intentionally when either runtime changes.

The best bridge is often no bridge. If a native overlay can own save, share, more, or exit actions—as GamePlayer's public cell demonstrates—then the web page does not need a route into those device capabilities. A native action handler can receive the already validated game reference and decide what the product is willing to do.

Lifecycle is part of trust

A WebView may finish loading after its cell is reused, fail provisionally before a response exists, terminate under memory pressure, or become hidden after a paging transition. These are not just performance events. They are moments when stale content can be displayed in the wrong product context unless lifecycle ownership is explicit.

GamePlayer assigns the active page as the interactive primary page and lazy-loads it. Reused cells reset their interface and replace their

web view before rendering a different item. The implementation handles finished and failed navigations with native loading/error UI; later chapters examine its memory and activation economics in more detail. The trust lesson here is that a page's authority should end with the session that created it. Reuse must not accidentally preserve a previous game's document, callbacks, or visual state.

On any failure, favor a native, understandable recovery path. The native container can show that a game could not load, offer retry if the reference remains valid, or return the user to a feed. It should not silently retry an unbounded number of times, carry an old WebView into a different item, or treat an invisible page as safe to keep doing work.

A capability review for a hybrid feature

Before shipping a WebView host, review the feature as a table of grants:

Question	Example decision
Which origins can the initial document and redirects reach?	HTTPS plus curated allowlist where feasible
Which storage survives the session?	Non-persistent by default; documented exception only
Is JavaScript needed?	Enable only if interactive content requires it
Is a bridge needed?	No by default; versioned, validated protocol if justified
How do external links behave?	Explicit native route or user-mediated handoff
What happens on failure/termination?	Native error state, bounded retry, safe return
What does the app log?	Redacted lifecycle/failure categories, never raw sensitive content

The goal is not to prohibit rich web content. It is to ensure the product can answer what it has granted and why. A capability that nobody owns becomes an incident later.

Keep authority on the side that can enforce it

Some responsibilities are tempting to delegate to the web page because the page is already displaying the game. Resist that shortcut when the responsibility depends on iOS policy or user consent. The remote page should not decide that a notification may be scheduled, that a native account is authenticated, that a deep link is safe, or that a file can be opened. It can request a narrowly defined experience; the native host validates the request against current application state and system permission.

Likewise, do not make the WebView a hidden session manager. A page can hold its own ephemeral game state while it is alive, but the application should not assume that state survives cell reuse, backgrounding, process termination, or a content update. If a user-facing recovery flow needs durable context, store the smallest native-owned, validated hint described in Chapter 5 and resolve it again under present policy.

This discipline reduces accidental privilege escalation. A compromised or simply buggy remote page has less ability to reach device resources, and a native product change has less chance of breaking the game by changing an undocumented bridge behavior. The boundary is also an organizational advantage: web and iOS teams can evolve independently when the contract is explicit, versioned, and narrow.

When a new capability is proposed, write a short decision record: user outcome, remote input shape, native validation, granted device action, failure behavior, logging/redaction policy, owner, and expiry/review date. If that record feels disproportionate, the capability may not justify a bridge at all. Simple native overlays and ordinary browser navigation are often safer than a permanent cross-runtime API.

The constraint is productive: it forces the experience to be explainable, testable, revocable, carefully observable, and safe under a changing remote runtime.

Generalized MemeArcade view

The approved MemeArcade view is a native product shell that hosts remote content through an explicit WebView boundary: native code retains navigation, lifecycle, device policy, and product routing; remote content owns its authorized interactive surface. The public GamePlayer repository demonstrates one possible implementation with HTTPS-only navigation, non-persistent web data, no JavaScript bridge, and lazy activation. It is not proof that MemeArcade uses the same exact configuration.

No private MemeArcade hostnames, origins, WebView configuration, bridge protocol, headers, cookies, content, scripts, routes, source, security findings, or performance measurements are approved for publication. Any detail beyond the generalized responsibility map requires explicit human approval.

Reader activity: make a capability table

Open GamePlayer directly. Inspect MAWebNavigationPolicy, MAWebViewFactory, the navigation tests, and the security-model section of its README. Create a two-column table for a hypothetical web game: “capability required for the game” and “native capability it must not receive by default.”

Then decide whether HTTPS-only policy is enough for that game or whether a host allowlist is required. State the operational cost of your choice. The expected observation is that a WebView is safest when every granted capability is deliberate, bounded, and owned by the native application.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/GamePlayer

Repository path:
GamePlayer/GamePlayer/MAWebNavigationPolicy.swift

Try: Build a WebView capability table and define a navigation trust policy.

Expected: A bounded trust model with explicit native ownership.

Native Scrolling, Web Gameplay

The first instinct for a feed of playable web games is often to put a WKWebView in every card and let the browser decide what happens. That creates a confusing product: a vertical drag can mean “move to the next game” or “scroll inside this page,” loading work starts for content the reader will never see, and a browser process quietly becomes the owner of the app’s most visible interaction.

The better question is not whether WebKit is powerful enough. It is: **which runtime owns each interaction?** In this chapter, native iOS owns the feed, page selection, lifecycle, native affordances, and the security boundary. The remote game owns only its own HTML, JavaScript, and touch interaction after it becomes active.

The engineering problem

A short-form game feed has two independent jobs. The feed must react to a confident vertical flick, settle on one item, update native chrome, and survive cell reuse. The active game must receive direct touches and load remote assets without making the rest of the application feel like a browser. If those jobs are assigned to the same scrolling surface, the team spends its time resolving gesture exceptions instead of defining a product model.

GamePlayer makes the split concrete. Its native pager owns the collection view, drag threshold, page settling, and refresh gesture. A reusable player cell owns one isolated web-game session and its loading or error presentation. A small SwiftUI bridge lets a SwiftUI application host the UIKit engine rather than asking SwiftUI to reproduce a performance-sensitive, full-screen paging primitive. This is not a vote against SwiftUI. It is a decision to use each framework at the boundary where it is strongest.

A model a junior engineer can use

Think of the screen as three nested surfaces:

1. **The native shell** knows the app's navigation, safe areas, lifecycle, accessibility, loading state, and actions such as share or save.
2. **The native pager** knows which item is primary and when the user has expressed an intention to change it.
3. **The web stage** knows how to run one game once native code has deliberately granted it the active stage.

Only one page should be primary. That does not merely mean "the one most visible on screen." It means one page has permission to receive game interaction and begin web work. Candidate pages may be prepared as native cells, but they should not all behave as live browser sessions. This turns an ambiguous scrolling problem into an explicit state transition:

candidate → primary → active web session → leaving → reused/cleared

The state machine matters because collection-view reuse is not an implementation detail. A cell that once represented game A can later represent game B. Its web state, callbacks, progress indicator, and native footer must be reset before B is displayed. Otherwise a reader can see stale loading state, old artwork, or—worse—a previously loaded game in the wrong product context.

Minimal implementation boundary

The following pseudocode is intentionally general. It describes ownership rather than a private implementation:

```
final class GameStageCell: UICollectionViewCell {
  func configure(candidate: Game) {
    clearSession()
    renderNativeMetadata(for: candidate)
  }
```

```
    func becomePrimary() {
        startWebSessionIfNeeded()
    }

    override func prepareForReuse() {
        super.prepareForReuse()
        clearSession()
    }
}
```

The pager, not the web view, decides when becomePrimary() is called. The web view, not the pager, receives game input once active. This small rule prevents a great many accidental couplings: a remote page cannot hijack vertical navigation, and a paging update does not require the page to expose its internal scroll position.

Lifecycle is product architecture

WKWebView is expensive enough that a feed must make its lifetime visible in the architecture. Creating a browser for every catalog item makes resource use proportional to the catalog. Reuse makes it proportional to the visible viewport. Lazy activation makes network work proportional to reader intent.

That is why GamePlayer starts a game only for the primary page. It also uses ephemeral web storage, so browser data does not silently accumulate through a long game session. The policy is useful beyond games: any hybrid list with remote, heavy, or untrusted content benefits from a bounded number of active sessions and from a clear disposal point.

Preloading is not a default. A team can measure the perceived latency of the next game, memory pressure on its target devices, WebKit process churn, and the effect of a nearby preload. Only then can it decide whether a candidate should prepare more than native metadata. A blanket “previous/current/next” strategy may

improve one demo while causing termination or jank in a real catalog. Measurement decides; architecture preserves the option.

Gesture ownership: make the handoff visible

The most visible interaction in a game feed is a vertical movement, but that movement is ambiguous. Inside an active game it may mean steer, jump, drag an object, or scroll a game-owned interface. Outside the game it may mean move to the next stage. Letting two recognizers negotiate this implicitly produces the worst kind of bug: it feels intermittent because the winner depends on a particular page's HTML, touch target, and timing.

GamePlayer resolves the ambiguity with native ownership of the pager. Its collection view does not use the ordinary free-scrolling behavior; it installs a pan gesture and calculates a target index from distance and velocity thresholds. The public controller uses a completion-progress threshold and a velocity threshold, then snaps to a bounded page index. It also reserves a pull-to-refresh interaction at the first page. These implementation details matter less than the product rule: a native policy decides whether the reader meant to navigate, and only the selected stage becomes interactive.

There are other valid designs. A feed might provide an explicit next/previous affordance, reserve a native footer gesture region, require a deliberate swipe from an edge, or use an onboarding overlay to teach the handoff. Choose the model that makes the game controls reliable and is understandable with assistive technologies. What does not scale is asking each remote game to cooperate with an undocumented native scroll convention.

Interaction	Owner	Why
Vertical page transition	Native pager	It changes product selection and lifecycle
Direct game touch	Active WebView	It belongs to the playable surface

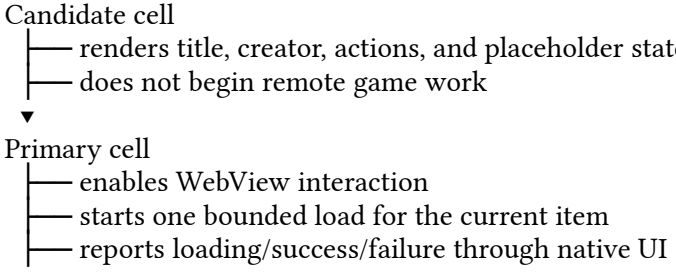
Interaction	Owner	Why
Like/share/more actions	Native overlay	It invokes product policy and device features
Pull-to-refresh at feed start	Native pager	It refreshes catalog-level state
Loading/error/retry	Native stage container	It remains available when remote content fails

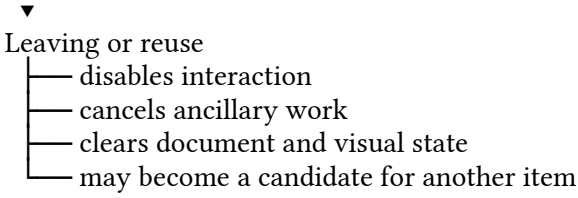
This allocation supports accessibility as well as performance. The native shell can expose a labeled page position, provide an accessible action to advance, retain focus around a loading transition, and present an error that VoiceOver users can discover. A remote game can remain independently accessible within its own content. Neither layer should have to infer the other's semantic structure.

The candidate-to-primary transition

The public pager stores a current index, updates the collection view's offset, and informs visible cells when the active page changes. The player cell receives a playback state; when it is primary it becomes visible and interactive, and only then runs loadIfNecessary(). During reuse, it cancels the avatar request, clears its item, stops loading, replaces the document content, and resets its native interface.

That sequence is a compact lifecycle contract:





The important invariant is identity. A completion for game A must not update a cell that now represents game B. In the public component, the cell's configure path resets before assigning a new item and tracks whether the current item has already begun loading. A more elaborate player may use session IDs, task handles, or a dedicated state machine. The architecture is the same: every callback must be tied to the active item and ignored once ownership changes.

Native chrome is not cosmetic

The stage cell includes native metadata and action controls—creator, title, description, and like/share/more actions—above the remote game. This has a structural advantage: product actions are not dependent on a page implementing the right JavaScript command. The native app can apply account rules, share-sheet policy, accessibility labels, analytics redaction, and error behavior in one place. The game remains playable without becoming a source of authority for the surrounding product.

For a larger application, actions should leave the cell as typed product-neutral events. The cell can say "the active item received a share intent"; an application coordinator decides whether the user is eligible, which share representation is valid, and which native route to present. This avoids coupling a reusable feed component to account state, backend mutations, or a global navigation controller.

```
// Illustrative contract; not a private implementation.
enum StageAction { case like, share, more }

struct StageEvent: Sendable {
  var reference: GameReference
```

```
    var action: StageAction
}

// The pager emits StageEvent; the application owns its product consequence.
```

The distinction becomes essential if the same player is used in a signed-out preview, a full app session, or a moderation review. The visual affordance can be the same while the product's response differs by policy.

Test the seams, not just the swipe

GamePlayer includes focused catalog and navigation tests alongside UI tests. That is the right direction. A pager is difficult to validate only with screenshots because the interesting failures occur across state transitions: a short drag should settle back; a fast flick should move one page; an attempt to advance at the end should remain bounded; a refresh should not leave an invisible spinner; an unsafe URL should be rejected before it becomes a web session.

Test at three levels:

1. **Pure policy tests** for URL validation, index calculation, and catalog record validation.
2. **Component tests** for cell reuse, active/inactive state, loading/error presentation, and callback identity.
3. **Journey tests** for a real launch, a page change, a failed game, an accessibility action, and a return to a stable native surface.

The tests should deliberately use fixture URLs and approved content. GamePlayer's README says to redistribute game catalog data only with permission from platforms and creators. This book follows the same boundary: it explains the architecture but does not establish redistribution rights for any game catalog.

The WebView is a trust boundary

Remote games are not native views wearing a different renderer. They are untrusted web content with their own redirects, asset requests, storage behavior, and failure modes. GamePlayer's public policy is intentionally narrow: HTTPS navigation is allowed; http, file, data, javascript, custom schemes, and invalid URLs are cancelled; each player uses a non-persistent data store; there is no JavaScript bridge and no native script injection.

This model makes the default capability small. If a product later needs a native-to-web message bridge, that bridge should be designed as an API with a schema, allowlisted commands, origin checks, validation, observability, and a revocation strategy. It should never be introduced as a convenience callback. Likewise, allowing any HTTPS host is a practical public-component default, not the end state for a curated production catalog. A known provider set should lead to an explicit host allowlist.

From public component to product integration

MemeArcade has the same responsibility categories inside a larger product: a SwiftUI application root and product state, a dedicated UIKit/WebKit vertical-pager module, catalog/feed services, persistence, native overlay surfaces, and fullscreen session coordination. The book does not reproduce its source or its catalog; any private excerpt requires explicit human approval. The useful architectural observation is that the player is not the application. It is a module with inputs from the product and callbacks back to the product.

That distinction protects both reuse and privacy. A portable player receives a candidate model, a lifecycle decision, a navigation policy, and typed callbacks for native actions. Product code supplies feed choices, account state, experimentation, copy, and navigation destinations. The module does not reach into a global app state to discover an endpoint or decide what a notification should say. When a team can draw that seam, it can test the pager with fixture

games, replace a catalog source, and debug a web failure without turning every investigation into a full-app investigation.

Senior trade-offs

The right implementation is rarely “all UIKit” or “all SwiftUI.” UIKit is a strong fit for a mature collection-view paging engine and reuse semantics. SwiftUI is a strong fit for application composition and isolated native surfaces that benefit from declarative state. WebKit is the required runtime for a remote browser game. The production decision is to minimize translation points: one owner for feed motion, one owner for web lifetime, and one owner for product navigation.

Observe those boundaries with events that answer operational questions: when a candidate became primary, when a web session began and ended, navigation-policy rejections, time to interactive, memory warnings, reuse counts, and whether a native action came from the active page. Avoid logging game URLs, private payloads, or user content unless there is an approved privacy policy. Good observability makes the architecture debuggable without widening the data surface.

Companion activity

Open the GamePlayer activity linked at the end of this chapter. Trace the pager, stage cell, web-view factory, and navigation policy. Draw a lifecycle diagram with the states candidate, primary, active, leaving, and reused. Then write one sentence for every transition: who initiates it, what resource starts or stops, and what native state remains authoritative.

Takeaway

Native code is not a thin wrapper around a game website. It is the orchestrator of reader intent, device lifecycle, performance budget, and trust policy. The web stage can stay expressive precisely because it is not also responsible for the application around it.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/GamePlayer

Repository path: GamePlayer/README.md

Try: Trace pager ownership, cell reuse, and WebView isolation.

Expected: A lifecycle sketch with explicit ownership.

GamePlayer II: Activation, Lifecycle, and WebView Economics

The most expensive mistake in a hybrid game feed is to confuse visibility with entitlement. A cell can exist without being selected. A selected item can be moving during a drag without being settled. A settled page can be loading without being interactive. A WebView can have begun work even though the user has already moved on. These are different product states, and collapsing them into a Boolean such as `isVisible` creates wasted work, stale callbacks, and memory pressure that is difficult to explain.

GamePlayer provides a compact public example. Its collection-view reuse bounds the number of player cells to the viewport rather than the catalog size. Its pager tracks a current index, decides page movement from native drag thresholds, applies playback state to visible cells, and makes the current page primary. A stage cell only loads its WebView when it becomes primary; it clears web state and cancels ancillary work on reuse. The README says plainly that adjacent-page preloading may improve perceived latency but can increase `WKWebView` memory pressure, so it must be profiled on target hardware before being introduced.

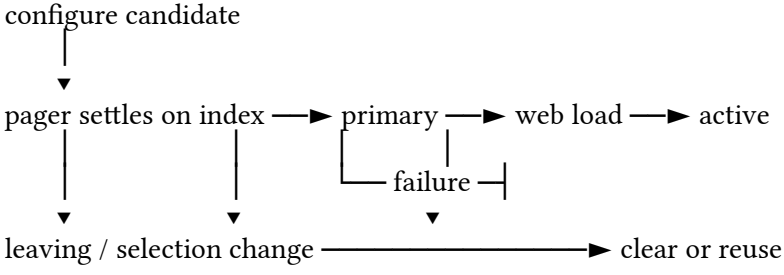
This chapter turns those facts into a production method. The goal is not to ban prewarming or promise a fixed memory number. It is to give every WebView session an owner, a lifecycle, a budget, and evidence for any optimization. MemeArcade, the App, is discussed only through the generalized principle of measurement-led hybrid session management. No private metrics, source, catalog, or runtime behavior is published.

Five states instead of one vague “loaded” flag

A useful state model begins before a web request exists:

State	Native meaning	Web work allowed?	Exit condition
Candidate	A cell represents a possible item	No	It becomes primary or is reused
Primary	Native pager has selected this item	Start bounded session	Load succeeds, fails, or selection changes
Active	Authorized page is ready for interaction	Yes, while current	User leaves, failure, memory event, reuse
Leaving	Selection or lifecycle no longer grants interaction	Cancel/stop/ clear as policy requires	Cleared or retained under measured policy
Reused	Cell will represent a different item	No	Configure with a new candidate

The names can differ, but the transition rules should be visible. In GamePlayer, a cell's playback state is .inactive or .primary; the primary state unhides and enables the WebView and calls the lazy load helper. prepareForReuse cancels the avatar request, clears the item, stops loading, replaces document content, and resets the visible interface. This keeps a completed load for a previous item from becoming an accidental starting point for the next item.



The state machine is a cost-control mechanism. It ensures that network, JavaScript, media, and rendering work follows deliberate user intent rather than the number of records in a catalog.

A WebView budget is not merely memory

WKWebView work has several costs at once: native view allocation, WebKit process activity, network requests, JavaScript execution, image/video decoding, GPU surfaces, cookies or web storage, and the cognitive cost of a page that may need recovery. A team that counts only resident memory may miss the reason a feed feels slow; a team that measures only first-load time may miss the reason the process terminates after several transitions.

Define a budget in dimensions, then decide what can trade against what:

Dimension	Question	Example signal
Activation latency	How long from settled selection to usable interaction?	Native timing span around load/finish
Scroll responsiveness	Does native paging remain smooth while a game runs?	Frame hitches, drag-to-settle delay
Memory pressure	What happens after repeated stage transitions?	Warnings, WebContent termination, resident trend
Network waste	How many loads finish after the user leaves?	Cancelled/obsolete activation count
Reuse integrity	Does an old session ever render in a new cell?	Identity mismatch assertions/tests
Failure recovery	Can a user return or retry safely?	Error-to-recovery completion rate

The exact metrics and collection system depend on the product and privacy policy. The important point is to distinguish an operation from an outcome. “Preloaded a page” is not success if it hurts responsiveness or loads content the reader never sees. “No preload” is not success if target devices show unacceptable activation latency and an adjacent session can be retained safely.

Measure the baseline first

Start with the simplest defensible policy: reuse cells, create isolated sessions as needed, and lazy-load only the primary item. GamePlayer already embodies this baseline. Then instrument a small, representative test matrix: low-memory supported device, modern device, slow and fast network, a lightweight game, a heavy game, long vertical browsing, background/foreground transition, and a load failure.

For each scenario, record a before/after trace around native events rather than exposing remote content. A privacy-safe timeline might include candidateConfigured, primarySelected, loadStarted, navigationFinished, navigationFailed, leftPrimary, reused, and memoryWarning. Associate the events with a generated session token, not a raw game URL or user identifier. This gives the team enough evidence to ask whether an optimization improved a real user outcome.

```
0 ms     primarySelected
8 ms     loadStarted
620 ms   navigationFinished
635 ms   interactionEnabled
3,300 ms leftPrimary
3,320 ms sessionCleared
```

The numbers above are a shape, not a benchmark. Never copy them into a performance target. Establish targets from the actual devices and content that the product supports.

Prewarming is a hypothesis

Prewarming has several possible meanings, which should not be conflated:

- **Native preconfiguration:** create the next cell's native metadata and layout without WebView network work.
- **WebView creation:** allocate a browser instance before it is selected.
- **Document load:** begin fetching/initializing the next game's page.
- **Interactive readiness:** load enough that the next page can receive input immediately.

Each level increases potential responsiveness and cost. Native preconfiguration is usually inexpensive. WebView creation can consume meaningful resources. Document loading may create network work that becomes obsolete. Interactive readiness can be the most expensive because it exercises exactly the JavaScript/media path the user may never choose.

GamePlayer's public README takes the responsible position: profile before adjacent-page preloading. A production experiment should choose one candidate policy, such as retaining only the immediate next primary candidate for a limited time, and compare it with the baseline. Define a rollback condition before running it: sustained memory warnings, increased termination/reload rate, degraded paging, unacceptable network waste, or no meaningful improvement in activation latency.

Policy	Potential gain	Cost	Evidence required
Primary only	Lowest resource use	Cold activation	Baseline user latency/recovery data

Policy	Potential gain	Cost	Evidence required
Native candidate only	Faster layout/chrome	Little web benefit	Frame/paging comparison
Adjacent WebView created	Some setup saved	Higher memory footprint	Target-device memory and termination data
Adjacent document loaded	Faster next transition	Network/CPU waste	Latency gain versus obsolete-load count
Broad catalog prewarm	Demo may feel instant	Unbounded browser cost	Usually reject without extraordinary evidence

The point is not to make the feed as eager as possible. It is to make its resource policy explainable.

Lifecycle interruptions are normal

Mobile processes do not behave like a desktop browser tab. The app can enter the background, receive a memory warning, lose network, change orientation, or have its WebContent process terminated. The user can scroll rapidly between items, invoke a native action while a game is loading, or revisit a page whose cell has already been reused. Treat each condition as an expected transition with a native-owned response.

When a page leaves primary state, first remove its ability to receive input. Then decide whether to stop loading, clear the document, retain a short-lived session, or capture a safe restoration hint. The answer depends on measured cost and product value. What should not happen is an invisible, unbounded web session continuing because nothing told it that ownership ended.

GamePlayer's public cell uses an ephemeral data store and clears web content on reuse. This favors isolation and bounded lifetime over guaranteed in-progress game continuity. That is a legitimate early policy. If a product later needs resume behavior, it needs a deliberate contract: what minimum state can the game supply, how is it validated, where is it stored, how long does it remain valid, and what happens when the remote content has changed? A reused WebView is not durable product state.

Instrument the state transitions, not the reader

The observability goal is to understand the session economy without creating a surveillance economy. Instrument state changes and coarse durations. Avoid raw URLs, page text, touch events, user-generated game content, cookies, account identifiers, or full request payloads. Use sampling and retention limits appropriate to the product. Make it possible to answer "why did load time increase?" without retaining what a particular person played.

Useful event categories include:

Event	Purpose
primary_selected	Establish a native activation attempt
web_load_started / web_load_finished	Measure bounded activation duration
web_load_failed	Classify recovery path without raw payloads
session_left_primary	Identify obsolete work / lifecycle end
cell_reused	Confirm bounded resource behavior
navigation_rejected	Surface trust-policy pressure
memory_warning / web_process_terminated	Correlate lifecycle stress with policy

These events must have a defined owner and consent/privacy review. They are not permission to collect every WebKit delegate callback. The best diagnostic signal is the smallest one that distinguishes the decision the team needs to make.

Turn measurements into a decision

An instrumentation plan is incomplete until it says who will read the result and what decision each result can change. A useful weekly review might compare the primary-only baseline with one limited prewarm cohort. It should ask whether the measured improvement is large enough for a person to notice, whether it holds across supported devices, and whether the resource cost appears in the same sessions that benefit.

Avoid average-only reasoning. Averages can hide the device class that experiences repeated WebContent termination or the network condition where speculative loads become waste. Review distributions and tails: how often does activation take too long; how often does a primary stage fail; what is the highest observed simultaneous session count; how often does a candidate preload become obsolete before it is selected? The product may accept a modest median improvement only if it does not make the slowest or lowest-memory experience worse.

Create a decision record for every policy change:

Decision record field	Example question
Hypothesis	Will one adjacent WebView reduce primary activation latency?
Scope	Which devices, content classes, and network conditions are included?
Primary metric	Time from settled selection to interactive state

Decision record field	Example question
Guardrails	Memory warnings, termination rate, paging responsiveness, obsolete loads
Result	Did the change help enough, consistently enough?
Action	Keep, narrow, revise, or roll back

This prevents an optimization from becoming permanent simply because it once made a demo feel faster. It also makes a rollback non-dramatic. If a new game type or iOS release changes the resource profile, the team can return to the baseline policy with an explanation rather than debate an undocumented performance superstition.

The same discipline applies to visible polish. A delayed action overlay, a pull-to-refresh indicator, or a native transition timer has a lifecycle cost. GamePlayer's public pager notifies visible cells when page movement begins and when a page settles. Any timer or overlay attached to those events must cancel on drag, deactivation, and reuse. Treat UI timing as a resource with an owner, not as a fire-and-forget animation. This is particularly important in a collection view, where a reused cell can otherwise reveal an action intended for a previous item.

In short, lifecycle economics means making both performance and interaction behavior reversible. The app can experiment because it knows how to observe, bound, and stop the work it starts.

That is what lets a hybrid feed remain fast for a reader without silently consuming a device's finite browser, network, battery, memory, and attention budgets.

Generalized MemeArcade view

The approved MemeArcade lesson is that a hybrid session should move through explicit candidate, primary, active, leaving, and

reuse/clear responsibilities, with optimization guided by target-device measurement. This is a design principle, not a claim that private code uses GamePlayer's exact states, counters, thresholds, WebView policy, or metrics.

No private MemeArcade lifecycle trace, memory measurement, performance target, game catalog, URL, source, session identifier, analytics event, WebView configuration, or prewarming behavior is approved for publication. Any exact implementation or performance claim requires explicit human approval.

Reader activity: design a measurement plan

Open GamePlayer directly. Inspect its lazy primary-page loading, cell reuse, WebView factory, and README performance guidance. Draw its candidate-to-reuse lifecycle, then write one metric for each of these questions: activation latency, memory pressure, obsolete work, reuse integrity, and navigation rejection.

Finally, propose one adjacent-page prewarm experiment with a success threshold and a rollback condition. The expected observation is that prewarming is not an architectural feature to announce. It is a reversible hypothesis that must earn its place through measurement on real devices.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/GamePlayer

Repository path: GamePlayer/README.md

Try: Measure activation and memory trade-offs before prewarming.

Expected: A measurement plan.

Seeing Across Native and Web

A hybrid failure is rarely located where the user experiences it. A person sees “the game did not open.” The cause may be a stale catalog item, a rejected navigation, a slow network, a malformed response, an inactive cell, a WebContent process termination, a cancelled task, or a native route that was never applied. Without a shared vocabulary, each team investigates only its own runtime and the incident becomes a collection of partial stories.

Observability is the architecture that connects those stories without exporting private game data, user content, or raw browser traffic. It begins with an event grammar: an activation was requested, a candidate became primary, a session started, a policy allowed or rejected a navigation, a page finished or failed, a native action returned, and the session left or was reused. Each event answers a question about a state transition. None needs to contain a full URL, page body, cookie, account identifier, or touch trace.

The public components provide small examples. GamePlayer’s pager already has native lifecycle boundaries—current-index changes, page-settle notifications, active/inactive cell state, navigation decisions, and native loading/error presentation. Recipe App’s main-actor view model has explicit loading and API-failure state, while its logging service exposes distinct debug, info, warning, and error levels. These are ingredients, not a complete production telemetry system. The lesson is to turn them into a coherent cross-runtime trace with privacy and retention limits.

MemeArcade, the App, is discussed only in that generalized form. No private production log, trace, event schema, URL, game payload, user data, metric, source, or implementation excerpt is approved for publication. Any such material requires explicit human approval.

Make events describe transitions

An event name should describe what changed, not what a developer happened to log while debugging. primary_selected is more useful than cell 12 appeared; navigation_rejected is more useful than WK error; catalog_decode_failed is more useful than a generic error. The event should be stable enough that an iOS engineer, web engineer, support lead, and product analyst use the same word for the same boundary.

Start with a small session model:

native_route_requested
  → catalog_item_validated
  → candidate_configured
  → primary_selected
  → web_load_started
  → web_load_finished | web_load_failed | navigation_rejected
  → native_action_requested | session_left_primary
  → session_cleared | cell_reused

This is not a required sequence for every application. A cached item may skip a network stage; a reader may leave before a document starts; a failure may take the user to a native recovery screen. The value is that any path can be described without guesswork.

Event	Native owner	What it establishes	What it must not include
candidate_configured	Pager/cell	A specific lifecycle slot has an item	Raw game URL or full catalog record
primary_selected	Pager	Native intent settled on a stage	Touch stream or user content
web_load_started	Player host	An authorized session began	Cookies, headers, request body

Event	Native owner	What it establishes	What it must not include
navigation_rejected	Policy	A navigation failed validation	Full rejected URL unless securely reviewed
web_load_failed	Player host	A bounded page load did not complete	Unredacted error object/payload
session_left_primary	Pager	The stage lost interaction authority	Behavioral profile of the user
cell_reused	Cell	Resources should no longer describe prior item	Prior content or session data

For a junior engineer, this table turns logging from print statements into an ownership exercise. For a senior engineer, it creates a vocabulary for dashboards, alert thresholds, tests, and incident reports.

Correlate without identifying the person

The native and web portions of one activation need a correlation key. A generated, short-lived session token can serve that purpose. It is created when native code selects a primary item and retired when the session leaves or is reused. It is not an account ID, a permanent game ID, or a hash of a URL. Its only job is to connect events that already belong to the same bounded native lifecycle.

```
session token S-123
  native: primary_selected
  native: web_load_started
  native: navigation_rejected
```

```
native: session_left_primary
native: session_cleared
```

If a reviewed bridge exists, the native host may attach the token to a small protocol event after validating origin and session identity. Do not put it into arbitrary page JavaScript, a query string, or browser storage by default. The less data crosses the runtime boundary, the easier it is to reason about its privacy, expiry, and revocation.

Correlation also has a time boundary. A session token should not survive product relaunch merely to make analytics convenient. If durable analysis needs a broader cohort, use a separate consented and privacy-reviewed mechanism. The player session trace is for diagnosing a short interaction, not reconstructing a person's history.

Build a failure taxonomy before you need one

Recipe App distinguishes service-side decoding and server failures, and its view model makes loading/error state visible. GamePlayer distinguishes a failed navigation from a finished one and exposes a native error label. Combine those ideas into a taxonomy that describes recovery, not just cause:

Category	Example trigger	Native response	Typical next action
Input invalid	Catalog record fails local validation	Do not create session	Refresh catalog or skip item
Policy rejected	URL/origin/scheme outside current policy	Stop before load	Return to safe native state

Category	Example trigger	Native response	Typical next action
Transport unavailable	Offline, timeout, transient failure	Preserve clear error state	Retry if still primary
Response invalid	Decode/ content contract failure	Reject unsafe result	Retry later or report provider issue
Web runtime failed	Provisional/ load/process failure	Hide stale stage, show native recovery	Reload or leave stage
Obsolete completion	User selected another stage	Ignore result	No user-visible error
Native action failed	Share/route/ permission denied	Explain boundary	Offer permitted alternative

The category is more useful than an error string. It tells the UI what to show, tells support what to ask, and tells engineering where to start. Preserve low-level diagnostic material only in an appropriate reviewed environment; do not send it indiscriminately to analytics or display it verbatim to a user.

Logs, metrics, and traces have different jobs

These terms are often mixed together. Separate them:

- **Logs** are discrete records for a developer or incident investigation. They should be structured, rate-limited, and redacted.
- **Metrics** are aggregate counts or durations: activation percentile, rejection rate, error category count, or successful recovery rate.

- **Traces** connect a bounded sequence of operations through a session/operation token.

Use logs when someone needs context, metrics when a team needs a trend, and traces when a team needs to reconstruct a single lifecycle. Do not turn every trace into a permanent log or every log line into a metric label. High-cardinality fields such as URLs, account IDs, error text, or content names can create privacy risk and operationally expensive dashboards.

The public Recipe App's logging examples are useful for teaching level selection, but a production hybrid application should audit what each message contains. Logging a full remote URL or file path may be inappropriate even if it helped during local development. Prefer categories and bounded details: `image_cache_miss`, `catalog_decode_failure`, `stage_load_duration_ms`, `navigation_rejected_scheme`. Use developer-only diagnostics under a controlled configuration when deeper detail is genuinely needed.

Instrument the handoffs

The most valuable events occur when ownership changes. GamePlayer's native pager settles a page and updates which visible cell is primary. The stage cell starts a web load, then displays either success or a native error. The product can capture coarse timing around those handoffs:

```
T0 primary_selected
T1 web_load_started
T2 web_load_finished | web_load_failed
T3 first_native_action | session_left_primary
T4 cell_reused
```

From that sequence, a team can derive useful metrics without knowing what content the reader played:

- Activation duration: `T2 - T0`.
- Time spent before leaving: `T3 - T2` when a session becomes active.

- Obsolete work rate: a load outcome after session_left_primary.
- Recovery rate: a failed session followed by a successful retry or safe return.
- Policy pressure: rejected navigations divided by attempted active sessions.

Each metric needs a product decision attached to it. A rising policy-rejection rate might mean a provider introduced a new trusted CDN, or it might reveal malformed catalog input. A slower activation percentile might justify a limited prewarm experiment only after validating device and content cohorts. Metrics reveal a question; they do not decide policy alone.

Privacy is a feature requirement

"We will redact later" is not an observability strategy. Decide data classes before writing events. Categorize fields as forbidden, permitted aggregate, short-lived diagnostic, or explicitly approved sensitive data. Review the schema when adding a bridge, a new provider, a share action, or an account feature.

Data class	Default	Examples
Forbidden in routine telemetry	Never emit	Cookies, authorization headers, full payloads, page text, credentials
Avoid/high-cardinality	Do not use as labels	Raw URL, account ID, game title, free-form error
Permitted coarse signal	Aggregate or short-lived	Failure category, duration bucket, session lifecycle state

Data class	Default	Examples
Reviewed diagnostic material	Controlled access and retention	Redacted trace necessary for an authorized incident

Set retention deliberately. A real-time operational counter may need only aggregates; a sampled debugging trace may have a short expiry; a security incident may have a separate handling procedure. Give someone ownership of deletion and access. Observability that cannot explain its retention eventually becomes another unbounded store of product data.

Diagnose an incident from the outside in

When a support report says “a game was blank,” resist opening every log source at once. Start with the user-visible transition: did the native pager select the intended stage; did the player present loading; did the page fail before a response, fail after a response, or never receive permission to navigate; did the user leave before completion; did a safe recovery route appear? The event grammar turns this into a sequence of answerable questions.

1. Locate the bounded native session or reproduction window.
2. Determine the last confirmed transition, not the last noisy log line.
3. Classify the failure using the taxonomy.
4. Inspect only the authorized detail necessary to distinguish the likely owner.
5. Choose a recovery, product fix, provider investigation, or policy adjustment.
6. Add a test or metric only if it will make the next occurrence cheaper to diagnose.

This workflow prevents an incident from becoming a request for more data by default. Often the missing piece is a state transition, not a raw payload. If a navigation was rejected, the important facts may be the policy category and the active-session state—not the

entire rejected address. If a load was obsolete, the key fact is that the pager had already selected another item—not every byte of the first page.

Use a small incident record that separates evidence from conclusion:

Field	Example form
User-visible outcome	“Stage showed a native retry state after selection.”
Observed transitions	primary_selected → web_load_started → web_load_failed
Confirmed boundary	Native player host observed failure callback
Unknowns	Remote service cause, content behavior, backend state
Recovery	Retry remains available while the item is current
Follow-up	Add a fixture that triggers the same failure category

The “unknowns” row is essential. It protects a team from turning a diagnosis into a claim about an unobserved backend or remote page. It also directs work toward the next testable question.

Observability should improve the product as well as the on-call experience. If a repeated category cannot be explained to a user or recovered from in the UI, the missing work may be a product state —not a dashboard. A failure taxonomy is successful when it produces clearer retry affordances, safer route fallbacks, and fewer ambiguous blank screens.

Review the vocabulary after every meaningful product change. A new bridge message, account state, provider, or notification entry point adds a possible lifecycle branch. If the branch cannot be named, owned, and observed in a redacted form, it is not ready to

be treated as routine production behavior. This discipline keeps the trace small enough to maintain, test, review, audit, and evolve safely under ongoing change, yet broad enough to explain the journeys the product actually offers.

Generalized MemeArcade view

The approved MemeArcade lesson is a shared, privacy-aware native/web event vocabulary that can diagnose a bounded reader session from selection through recovery or release. Native code owns the session identity, route and policy transitions; the remote runtime is observed through its authorized, minimal integration points. This is not an assertion about private telemetry vendors, event names, dashboards, logs, payloads, URLs, or metrics.

No private production log, trace, analytics record, source, user data, game content, endpoint, identifier, security finding, or performance number is approved for publication. Any specific production evidence requires explicit human approval and a privacy review.

Reader activity: write a failure taxonomy

Open GamePlayer directly, then compare its pager lifecycle, navigation policy, and load/error state with Recipe App's loading and service-failure states. Write six event names for one session and put every possible failure into one recovery category from the table above.

For each event, remove every field that is not required to understand the state transition. The expected observation is that a useful hybrid trace can explain what happened without retaining who the reader was, what they typed, or the full content that a remote page rendered.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/GamePlayer

Repository path:
GamePlayer/GamePlayer/VerticalPager/MAStagePlayerCell.swift

Try: Build a privacy-safe lifecycle trace and failure taxonomy.

Expected: A bounded event vocabulary with recovery categories.

Pushscheduler: Local Notifications as Edge Infrastructure

Notifications are often discussed as a growth channel, which makes it easy to skip the more important architectural question: **where does the event originate, and who is allowed to decide it is worth interrupting someone?** If the device already knows the content and timing of a reminder, a local notification can be the simplest, most private implementation. If a server-side event, another person, or cross-device state must trigger the interruption, local scheduling is the wrong tool and remote push is required.

The public Pushscheduler repository is an experimental SwiftUI lab for this device-owned half of the problem. It requests and inspects notification permission, persists notification plans in UserDefaults, replaces/refreshes/clears a namespaced family of requests, limits its scheduled plan to 32 pending requests, shows foreground presentation, validates local deep-link payloads, and surfaces a selected route rather than opening it automatically. The implementation splits responsibility across an alert orchestrator, schedule vault, notification navigator, alert conductor, and studio UI.

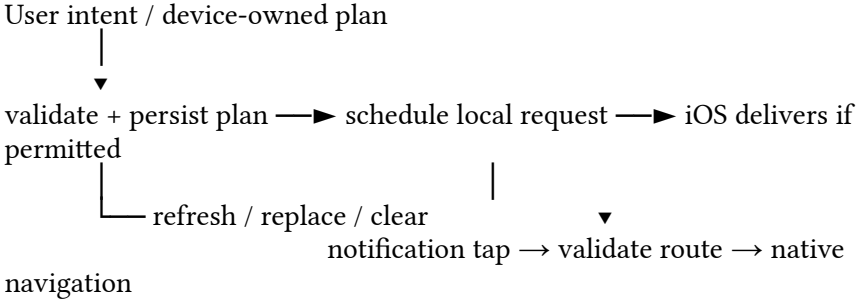
That decomposition matters for MemeArcade, the App. A product may want to re-engage a reader after a device-known moment: a locally scheduled follow-up, a reminder they asked for, or a safe "ready" state produced while the app is still responsible for the timing. The product must not pretend that a local notification is an APNs message, a guarantee of delivery, or evidence that a remote game/server reached a condition. This chapter explains the boundary through public code and generalized architecture only. No private notification copy, payload, route, plan, source, device data, or analytics is published.

Local versus remote is an origin decision

Start with the trigger, not the notification UI.

Trigger origin	Appropriate mechanism	Why
Device knows a chosen reminder time	Local notification	No server decision is needed
App creates a local follow-up plan	Local notification	Plan can persist and refresh on the device
Server observes a completed job	Remote push or in-app refresh	Device cannot reliably invent this external event
Another user sends a message	Remote push	Trigger exists outside one device
Account state changes elsewhere	Remote push / sync	Requires cross-device authority
User is active in foreground	In-app UI, optionally notification presentation	Do not interrupt blindly

Local notifications continue after the app process exits once the system has accepted a request, but "continues" is not the same as "guarantees a business outcome." A user can deny or later disable permission, change device settings, remove the app, clear a notification, or tap after the underlying product state has changed. The app must treat a tap as a fresh input requiring validation, not as a command to restore an old route.



Permission is a state machine

Asking for permission is not a one-time button action. It is a state machine with product consequences. The public MAAlertOrchestrator exposes both requestPermission() and currentPermissionStatus(), while the conductor/UI can represent that status and offer a Settings handoff. That is a better model than assuming a permission prompt means the user has opted in forever.

For every entry point, decide what happens when permission is undetermined, denied, authorized, provisional, or otherwise limited by the system. Do not block the primary game/feed experience behind a reminder permission. Explain the value at the moment it is relevant; if the user declines, preserve the core experience and offer a future Settings path only when it is useful.

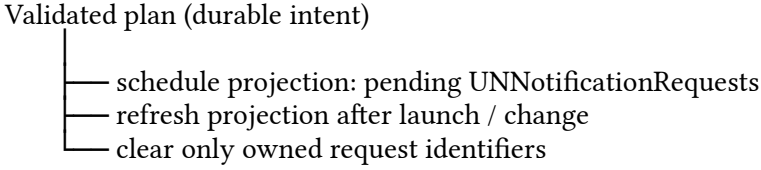
Permission state	Product behavior
Not determined	Explain a specific benefit, then request only after user intent
Authorized	Schedule only validated, user-appropriate plans
Denied	Do not retry prompts; offer a non-blocking Settings route when relevant
Changed in Settings	Re-read status before scheduling/refreshing
Foreground	Decide whether to show an in-app/foreground presentation deliberately

The important ownership rule is that the notification layer reports permission state; the product decides whether and how to request it. A scheduler should not manufacture persuasive copy, change app navigation, or quietly turn every user action into a prompt.

Persist the plan, not a pile of requests

Pushscheduler stores a notification plan, not merely an unmanaged list of pending requests. Its orchestrator constructs a validated plan, saves it through MAScheduleVault, clears only its own scheduled request family, then refreshes pending items. During refresh it reads the current pending requests, filters namespaced identifiers, calculates remaining capacity under a 32-request cap, and schedules only upcoming plan items that are not already pending.

This is a useful device-infrastructure pattern. The plan is the source of intent; system notification requests are a projection that may need reconstruction after app relaunch, replacement, or capacity change. Without a plan, an app can only inspect the current pending queue and guess whether it still represents the user's choice.



The same distinction from Chapter 5 applies. A persisted plan is historical input. Validate its schema, identifiers, time values, and copy before using it; recover safely from corrupt data; make a reset explicit. Do not store credentials, unrestricted remote URLs, private game payloads, or product state that must be refreshed from a server. The public vault's recovery behavior is a useful model: a corrupted plan should not prevent the application from starting.

Namespaced identifiers and bounded schedules

Notification centers are shared at the app level. Removing all pending notifications because one feature changed is a classic ownership bug. Pushscheduler avoids it by using MA-namespaced identifier families and filtering scheduled versus "game ready" requests. The pattern is portable: each feature owns identifiers it

can recognize, replace, and clear without disturbing another device feature.

The 32-request cap in the public lab is a conservative application policy, not a universal platform limit to copy blindly. It forces the scheduler to make capacity visible: create a small window of upcoming reminders, refresh it when appropriate, and reject/handle a plan that cannot be represented safely. A product should set its own budget after considering platform limits, UX, and the risk of filling a queue with stale interruptions.

Scheduling decision	Why it matters
Identifier family	Enables precise replacement and cleanup
Bounded copy/IDs	Avoids malformed or oversized payloads
Capacity policy	Stops unbounded queues and makes refresh necessary
Time validation	Prevents impossible past/invalid schedules
Replace rather than append	Keeps latest user intent authoritative
Clear delivered and pending owned requests	Prevents stale re-entry paths

The system should be able to answer: which plan created this request, whether that plan is still current, and what action clears it.

A notification tap is untrusted input

It is tempting to treat a local notification as trusted because the app created it. But it may have been scheduled by an older app version, persist after a state reset, or contain a malformed/corrupt payload. A tapped notification is an entry point into the application and deserves the same routing discipline as a deep link.

Pushscheduler's MANotificationNavigator accepts only a local pushscheduler:// scheme, validates payloads, bounds input length, maps a recognized destination, and deliberately surfaces rather than auto-opens the selected route. That final choice is important. The navigator tells the product "a route was requested"; the product checks current state and chooses whether it is valid to navigate now.

```
// Teaching sketch; not private MemeArcade code.
func handleNotificationTap(_ payload: NotificationPayload) {
    guard let route = navigator.validatedRoute(from: payload) else {
        return showSafeDefault()
    }
    coordinator.consider(route, under: currentProductState)
}
```

The consider step can reject an expired reference, require the user to select an account, refresh a catalog, or route to a stable native surface. Never load a web URL supplied by a notification payload without the same origin and catalog validation described in Chapters 6 and 7.

Foreground presentation and respectful interruption

When the app is foregrounded, a notification may be redundant or disruptive. The public lab includes foreground toasts and notification callbacks, making the presentation a deliberate UI choice instead of an accidental duplicate interruption. A production product should decide whether the foreground experience needs a toast, a badge, a quiet state update, or no presentation at all.

Respect is an implementation requirement. Schedule only events that have a clear user benefit, provide a clear way to change/disable the plan, avoid repeated reminders that no longer match the product state, and never use local scheduling to imitate a remote social or server event. If a game-ready message is genuinely

local, say what local condition made it ready. If readiness belongs to a service, wait for the service-owned trigger.

Measure the scheduler's health with privacy-safe aggregates: permission status category, plan creation/replacement/clear result, number of owned pending requests, validation rejection category, and route outcome. Do not report notification body text, full deep links, game content, or account identifiers as routine telemetry. A high tap rate does not make an interruption appropriate; product judgment remains necessary.

Local scheduling is not APNs

APNs remote notifications involve server credentials, device-token lifecycle, provider delivery, payload construction, and a remote trigger. Pushscheduler intentionally includes none of those concerns. This is a feature of the teaching example: it lets a reader understand the local edge of notification architecture without claiming that a serverless plan can solve cross-device or externally-triggered product needs.

When a product grows into remote push, retain the same internal boundaries: permission/status, validated payload, bounded route, foreground presentation, metrics, and safe navigation. The transport changes; the device-side trust model does not disappear. A remote payload should be treated as at least as untrusted as a local persisted plan.

Test schedules as time-dependent state

Notification code is easy to demo and easy to leave untested because time, permission, and system delivery are involved. Pushscheduler makes several seams injectable—notification center, plan store, timing policy, and calendar—which allows its tests to examine behavior without waiting for a real day to pass. Carry that approach into any product scheduler.

Test the planner separately from the system request projection. Given a fixed clock and a plan, verify that invalid identifiers,

malformed copy, impossible clock values, duplicate request identifiers, and capacity overflow are rejected or recovered as intended. Given an existing plan plus a simulated set of pending request identifiers, verify that refresh schedules only future owned items and does not remove unrelated notifications. Given corrupted stored data, verify that the app returns a safe empty/default state rather than failing launch.

Then test re-entry. A route produced from a valid fixture should reach the coordinator as a request, not cause navigation before current account, catalog, and permission policy have a chance to decide. A malformed route should result in a safe default. A cancelled plan should remove both pending and delivered notifications from its owned family when that is the stated product behavior.

Test level	Example assertion
Policy unit test	Identifier/copy/deep-link validator rejects malformed input
Plan test	Replacement retains only current plan intent
Projection test	Refresh respects capacity and avoids duplicate owned requests
Vault test	Corrupt serialized plan recovers safely
Route test	Tap produces a validated request, not automatic navigation
UI journey	Permission denial still leaves the core app usable

The system notification center remains the delivery authority, so tests should avoid pretending they prove every device-level delivery behavior. They do prove the application logic surrounding

delivery: what it asks for, what it schedules, what it owns, and how it recovers. That is the part of notification architecture the iOS application can actually make dependable.

Review those tests whenever notification copy, route formats, account boundaries, or scheduling rules change. A notification is an application entry point that may fire long after the original UI disappeared. The longer its lifetime, the more valuable it is to keep its plan, payload, and recovery behavior small, explicit, and independently testable.

That discipline protects both the reader's attention and the product's ability to explain why an interruption appeared, what it means now, and how it can be dismissed.

Generalized MemeArcade view

The approved MemeArcade lesson is that device-owned re-engagement may be modeled as a validated local plan, a bounded projection into system notification requests, and a native-owned route decision on re-entry. Local notifications are used only when timing/content are owned by the device; server- or cross-device-originated events require a different remote-push design.

No private MemeArcade notification text, plan, identifier, schedule, deep link, route, game reference, account state, source, payload, analytics, remote-push configuration, token, or server behavior is approved for publication. Any product-specific notification evidence requires explicit human approval.

Reader activity: trace a local plan

Open Pushscheduler directly. Trace one plan from permission status through validation, vault persistence, namespaced pending request, foreground/tap handling, and route selection. Identify which state is durable intent and which is merely the current system projection.

Then design one reminder for a hypothetical game app. State why its trigger is device-owned, what clears it, how a denied permission changes the experience, and how a tapped route is validated before navigation. The expected observation is that local notifications are small, powerful pieces of edge infrastructure only when their origin, ownership, and re-entry path are explicit.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/Pushscheduler

Repository path: Pushscheduler/README.md

Try: Identify permission, persistence, schedule, and routing boundaries.

Expected: A local-notification state flow.

Deconstructing MemeArcade: One Complete Session

Architecture becomes real when a person follows one path through the product. A diagram can say that SwiftUI owns app state, a package owns persistence, a client owns a request, a pager owns selection, a WebView owns gameplay, and a scheduler owns a local reminder. A session proves whether those responsibilities compose without asking one layer to impersonate another.

This chapter follows a **generalized** MemeArcade session from launch to possible re-entry. It is deliberately not a replay of private source or an assertion about a specific catalog record, endpoint, game, account, notification, or production metric. Each transition is built from the public patterns already studied: SwiftUI/Combine state, DataStore-style restoration, async service ownership, validated network contracts, GamePlayer paging and WebView policy, privacy-aware events, and Pushscheduler's device-owned plan/routing boundary.

The purpose is not to make every app follow the same path. It is to give a team a method for tracing one: identify the owner, validate the input, name the state transition, record only the necessary evidence, and preserve a safe recovery path.

The session at a glance

Launch
→ restore safe native state
→ establish product route
→ obtain/validate catalog
→ configure candidate stage
→ native pager selects primary item
→ create constrained web session
→ render game or native recovery
→ handle native action
→ optionally create device-owned reminder plan

→ later notification/deep-link input
→ validate current route and re-enter safely

At no point does “remote content loaded” mean “the application has surrendered control.” At no point does “a persisted value exists” mean “it is current authority.” At no point does “a notification was tapped” mean “navigate without checking current state.” The session works because each event re-enters the product through a native-owned decision.

1. Launch: restore intent, not a fossilized object graph

The first responsibility belongs to the native shell. It constructs a product model that can represent restoration as a visible state rather than assuming all remembered data is immediately safe. Chapter 5’s DataStore model gives the right mental shape: restore a small `Codable` representation, migrate it if needed, and apply a default or recovery policy if the record cannot be used.

For a generalized MemeArcade session, the restored value might be a harmless preference, onboarding completion, or a stable selection hint. It must not be treated as a still-valid remote page, an authenticated session, an arbitrary web URL, or a complete in-progress game state. The shell starts in a state such as `restoring`, then resolves into `ready`, `needsOnboarding`, or a safe fallback.

Restored input	Native check	Possible result
App preference	Decode/migrate/ value bounds	Apply setting
Last surface hint	Does current product policy allow it?	Restore native surface or default home
Content reference	Is it available in current catalog/policy?	Re-resolve or discard

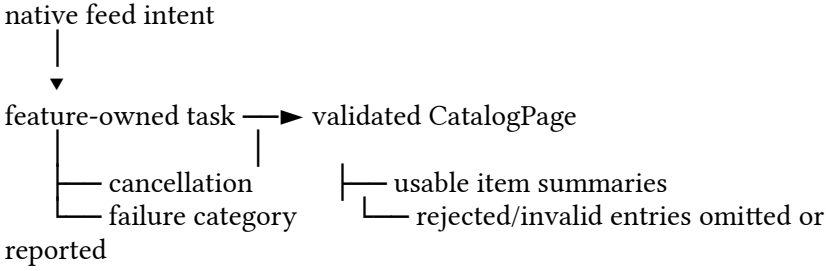
Restored input	Native check	Possible result
Notification plan	Is schema/time/permission still valid?	Refresh, retain, or clear
Unknown/corrupt record	Recovery policy	Safe default plus redacted diagnostic

The product should never display a blank screen while this happens. “Restoring your session” can be a legitimate native state. It tells a user why remote content is not yet visible and gives the application time to make current decisions.

2. Catalog: turn transport into validated product data

Once the native shell is ready, a catalog/feed service may begin owned asynchronous work. Recipe App’s public example establishes the essential split: a @MainActor feature model owns loading/error state; an actor-based service owns transport/decoding; a response becomes a domain value only after validation. The same structure applies to a game catalog without claiming the same source or endpoint.

The service task has an owner and an operation identity. If the user refreshes, leaves the surface, or a newer request supersedes it, obsolete work is cancelled or ignored. The result is classified—not thrown directly into a view as an arbitrary transport error.



The catalog should supply only the fields the player needs: a product-valid reference, approved metadata, and any policy input required by the host. It should not let a raw remote response decide native routing or WebView configuration. If a response is malformed, the feature can retain an existing safe list, show a retry state, or render an empty state. The choice is product policy, not an accident of JSON decoding.

3. Candidate: native UI before browser work

The catalog produces candidates, not an army of browser sessions. GamePlayer's public pager configures a reusable stage cell with native metadata and keeps it inactive until the native page selection settles. This is an important separation: the feed can show title, attribution, accessible controls, and placeholders without fetching every game's remote document.

Candidate configuration has a bounded contract:

1. Clear the cell's previous identity, callbacks, and visual state.
2. Validate the candidate's reference against current native policy.
3. Render native metadata and loading-neutral state.
4. Do not grant game interaction or start the remote load yet.

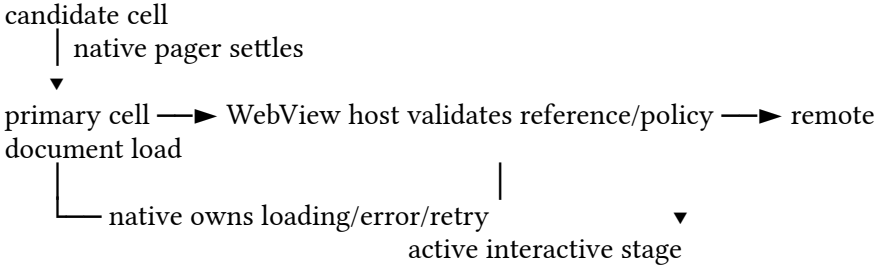
If a candidate is removed, becomes invalid, or is replaced by a catalog refresh, no browser cleanup is needed because no session was granted. That is one reason candidate and primary should remain different states.

4. Primary selection: native intent becomes a session grant

The native pager owns the decisive interaction. In GamePlayer, UIKit determines drag threshold, velocity, target index, settling, and which visible cell receives .primary. A production app may use a different gesture policy, but the transition should remain native-

owned because it changes product selection and resource allocation.

When a candidate becomes primary, the player host creates the smallest authorized session: a constrained WebView with the navigation/storage policy described in Chapter 8. The host records primary_selected and web_load_started as bounded lifecycle transitions. It does not expose a private URL in routine telemetry or let the page become a navigation authority.



If the user swipes away before the load completes, the product transitions to leaving. A late callback must not update the now-inactive or reused cell. The task/session identity rule from Chapter 6 applies equally to browser delegates.

5. Web session: participate without becoming the product

The remote page owns its interactive game surface after the host grants it primary status. Native code continues to own navigation policy, WebView lifetime, device permissions, product actions, and error recovery. A valid page load does not grant a JavaScript bridge, arbitrary deep-link access, persistent account state, or ability to select the app's next route.

The public GamePlayer default is intentionally conservative: HTTPS-with-host navigation, a non-persistent website data store, no JavaScript bridge, and native action controls. A product can make different choices only by documenting their capability, validation, owner, and revocation behavior. In the generalized

MemeArcade session, the relevant fact is not which exact policy is used; it is that one exists and is enforced on both the initial reference and subsequent navigation.

When the session fails, a native layer tells the truth: the stage did not load, retry may be available, and the feed remains an escape path. When it succeeds, a native overlay can still provide share, save, more, or exit actions without asking the game page to command the app.

6. Native action: translate an intent at the boundary

Suppose a user taps a native action while a game is active. The player emits a typed, product-neutral event associated with the active validated reference. The application coordinator decides its meaning: share a safe representation, open a native sheet, record an approved action, or decline the action under current account/policy state.

This separation is what makes the player reusable. It does not need to import account services, notification scheduling, or app navigation. It provides an intent; composition code supplies the product consequence.

Event source	What crosses boundary	What remains native-owned
Pager	Selected valid reference	Route and session policy
WebView	Load lifecycle outcome	Error/retry UI and telemetry policy
Native action control	Typed action + active reference	Account, share, navigation decision
Notification tap	Validated route request	Current-state re-entry decision

The table also prevents accidental coupling. A native share result should not quietly change a WebView document; a game completion signal should not automatically schedule a notification; a scheduled reminder should not reopen stale web state.

7. Optional re-engagement: create a local plan only when the device owns it

Some product moments justify an optional reminder. If the timing and content are entirely device-owned—a reader asks for a follow-up, sets a reminder, or starts a local countdown—the app can construct a validated notification plan. Pushscheduler shows the pattern: save durable intent, project a bounded set of namespaced notification requests into the system center, refresh/replace them as needed, and clear only the family the feature owns.

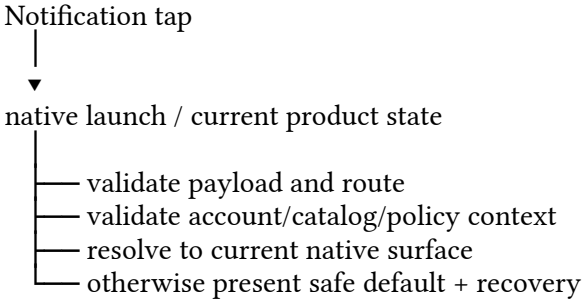
Do not schedule a "game ready" notification merely because a remote game was loaded or because a server may later do something. That would confuse source of truth. Local scheduling is appropriate only when the device can make the promise. Otherwise the app needs a remote-trigger design and must treat its payload as another untrusted external input.

The local plan itself carries no authority to reopen a game. Its payload becomes a requested native route later, subject to the same validation as a deep link: current account/product state, current catalog/policy, and safe fallback.

8. Re-entry: validate the present, not the past

Hours later, a person may tap the notification. The original application process may be gone; the catalog may have changed; permission may have been revoked; the previous game reference may no longer be valid. The app launches into the same restoration flow as any other launch, reads the notification request, validates its local scheme/fields, and surfaces a route request to the coordinator.

The coordinator can then decide whether it can restore a native surface, refresh content, show a clear unavailable state, or take the user home. It must not load an embedded URL directly from the payload or assume an old WebView session can be recreated safely.



This is what “one complete session” means in a mobile system. The session is not a single linear process kept alive forever. It is a series of validated handoffs across lifetimes.

Observe the path, protect the reader

The session can be diagnosed with the minimal event vocabulary from Chapter 11: restoration began/ended, catalog result category, candidate configured, primary selected, web load outcome, native action result, local plan result, route request validated/rejected, and session released. Correlate only within a bounded lifecycle using an ephemeral operation/session token. Do not retain page content, raw URLs, headers, notification body text, account identifiers, or game interactions simply because they would make a trace richer.

The same events make tests possible. A fixture catalog can yield a valid candidate, invalid candidate, empty result, or failed load. A fixture notification can yield a valid route request or a malformed one. The test should prove the safe transition at every fork:

- corrupt persisted state → safe native default;
- obsolete catalog task → no stale UI overwrite;
- invalid candidate → no WebView session;
- policy rejection → native recovery state;

- leaving primary stage → no late callback applied;
- malformed notification payload → no automatic navigation;
- valid device-owned plan → only owned requests replaced/cleared.

A session review is an architecture review

Use this trace during design review before a feature ships. Pick the most important reader path, then ask five questions at every boundary: Who owns this fact? What validates the input? How long may this state live? What cancels or replaces the work? What does the user see if this step fails? If a boundary has no answer, it is not a minor implementation omission; it is an unresolved product decision.

The review is also a way to prevent accidental scope expansion. A new server-triggered event belongs in the remote-push architecture, not in a local scheduler because it is convenient. A new web capability belongs in a documented bridge review, not in an arbitrary JavaScript message. A new persisted field needs migration and deletion behavior, not just `Codable` conformance. One complete session makes those hidden dependencies visible while they are still inexpensive to correct.

Generalized MemeArcade view

The approved MemeArcade view is an architecture composed from native product state, validated device persistence, owned asynchronous catalog work, a native pager, a constrained remote stage, typed native actions, optional device-owned re-engagement, and validated re-entry. It is a responsibility trace, not a publication of private product flow.

No private source, module name, route, catalog entry, game, URL, endpoint, payload, account state, notification, session trace, analytics event, security policy, metric, or implementation detail is

approved for publication. Any such evidence requires explicit human approval.

Reader activity: draw the responsibility trace

Open GamePlayer and Pushscheduler directly. Draw the eight transitions in this chapter and assign one owner to each: native shell, persistence adapter, service actor, pager, player host, remote stage, notification scheduler, or route coordinator.

For every arrow, write one validation or cancellation rule. The expected observation is that the whole product can be explained as a chain of owned decisions—not as a screen sequence or an invisible collection of callbacks.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/GamePlayer

Repository path: GamePlayer/README.md

Try: Trace a complete native-to-web-to-native session and assign one owner to every handoff.

Expected: A responsibility trace with validation and cancellation rules.

From MemeArcade to Reusable Architecture

The goal of a case study is not to copy its product. It is to identify the contracts that remain useful when the catalog, brand, business model, remote provider, and user journey all change. In a hybrid iOS application, those contracts are rarely whole screens. They are responsibility boundaries: a product state transition, a durable-state adapter, a cancellable client, a WebView policy, a pager lifecycle, a notification plan, or a privacy-aware event grammar.

MemeArcade, the App, is private and product-specific. Its value as a case study is precisely that it forces a distinction between what should be extracted and what must remain contextual. A reusable component is not "the MemeArcade player with different colors." It is a smaller capability with explicit inputs, outputs, lifetimes, trust rules, tests, and forbidden dependencies.

The public repositories demonstrate different pieces of that extraction discipline: ios-framework makes a package and tandem app visible; ios-storage isolates durable device state; receipe-app isolates concurrent transport/cache work; GamePlayer isolates a native pager and constrained web session; Pushscheduler isolates local notification planning/routing; and rezona-api demonstrates evidence boundaries around client observations. This final chapter turns those pieces into an extraction method.

Begin with the contract that already exists

Teams often begin extraction by moving files into a folder called Core or by creating a package because a target feels large. That changes source layout without necessarily changing coupling. Start instead with a contract that the rest of the app already needs:

- "Given a validated game reference and navigation policy, host one bounded remote stage."

- “Given a device-owned plan, create, refresh, and clear only the local notification requests this feature owns.”
- “Given a Codable state representation, restore it safely and observe future durable changes.”
- “Given a catalog query, return validated domain values or a known failure category.”

Each sentence identifies an input, a responsibility, and a result. It should also identify what the capability does **not** own. A player host does not own the global route; a scheduler does not decide marketing policy; storage does not decide which screen appears; a client does not expose raw transport detail to a SwiftUI view.

```
Reusable capability
  inputs: validated domain values + explicit policy/dependencies
  owns:   one coherent behavior and its local lifecycle
  outputs: domain result / typed event / failure category
  forbids: global navigation, private product data, hidden singletons,
unrelated UI
```

This contract-first test prevents a common failure: extracting a module that still imports half of the application and therefore cannot be used, tested, or released independently.

A reusable architecture map

The following map groups the book’s capabilities by their natural responsibility. It is a proposal for reasoning, not a claim about MemeArcade’s private package graph.

Capability	Stable contract	Possible public teaching source
Product shell	intents → route/session state	SwiftUI/Combine article
Durable state	restore/save validated representation	ios-storage

Capability	Stable contract	Possible public teaching source
Catalog client	query ⟶ domain page/failure	receipe-app, rezona-api method
Player host	reference + policy ⟶ typed lifecycle/action events	GamePlayer
Web policy	URL/response ⟶ allow or reject	GamePlayer
Local scheduler	validated plan ⟶ owned requests/route request	Pushscheduler
Observability adapter	state transition ⟶ redacted event	GamePlayer + Recipe App

Notice that none of the reusable contracts says "knows MemeArcade." The product composes them. That is the architectural center of gravity: reusable parts move downward into policy-constrained capabilities; product decisions remain at the composition root.

Extract in dependency order

Not every boundary should become a package immediately. Start with pure domain models and policy protocols because they have few dependencies. Then extract infrastructure adapters whose inputs can be explicit. Extract UI hosts only after their lifecycle and callback contracts are stable. Keep product composition in the app target until a second consumer or release boundary makes independent distribution valuable.

Domain values / protocols
↓
Pure validation and policy
↓

Infrastructure adapters (storage, client, scheduler)
↓
Feature host (pager, WebView container)
↓
Application composition (route, account, product choices)

The arrows point downward: higher layers may depend on lower-layer contracts; lower layers should not import the application to discover what to do. ios-framework's tandem-app pattern gives a practical test. If a demo app can import a package and exercise its public entry point without copying global product state, the boundary is becoming real. If the package needs a global app singleton, an app route enum, or private service configuration just to compile, continue refining its inputs.

For example, a PlayerHost can accept a GameReference, a navigation-policy protocol, and an event sink. It should not import AccountManager, HomeTab, or a proprietary catalog client. The app supplies those concerns when it composes the host. That makes the host testable with fixtures and lets a second product use it under a different routing policy.

Types are better seams than callbacks without contracts

A reusable component needs a language for crossing its boundary. Prefer small domain types and protocols over unstructured dictionaries, unbounded notification-center messages, or strings that mean different things in different features.

```
// Teaching sketch; not a private API.
public struct StageReference: Sendable, Equatable {
    public let opaqueID: String
}

public enum StageEvent: Sendable, Equatable {
    case loadStarted
    case loadFailed(StageFailure)
    case requestedAction(StageAction)
```

```
    case leftPrimary
}

public protocol StageEventSink: AnyObject {
    func receive(_ event: StageEvent, for reference: StageReference)
}
```

The example does not need to expose a URL, raw browser message, account identity, or full catalog record. It gives the host enough information to perform its job while reserving product interpretation for the app. That is a privacy boundary as well as a modularity boundary.

Use the same discipline for notification routing. A scheduler can emit a validated RouteRequest, not open a URL. A store can return a RestorationResult, not mutate the app's navigation state. A client can return a CatalogPage, not a transport JSON object. Each typed result gives tests a stable surface and keeps product data from leaking into components that do not need it.

Decide what must remain product-specific

Extraction is successful when it leaves important product decisions where they belong. The following areas should resist generic SDK treatment until there is a true shared contract and explicit approval:

- Brand voice, notification copy, artwork, onboarding, and accessibility language.
- Catalog ranking, moderation, content/provider agreements, and entitlement rules.
- Account state, user-generated content, private endpoints, server payloads, credentials, and analytics identifiers.
- Host allowlists, security exceptions, bridge capability decisions, and incident data.
- Growth experiments, pricing, marketing logic, and any claim about market/business performance.

These are not "messy details" to hide inside a component. They are the product. A generic library that quietly includes them becomes difficult to audit and dangerous to reuse. Keep them in an application policy layer with clear owners and appropriate access controls.

Extract when	Keep product-specific when
Input/output and lifecycle can be stated without brand context	Behavior depends on accounts, contracts, ranking, or editorial choice
A second consumer could use the capability safely	Reuse would expose private data or policy
Tests can use public fixtures	Tests require production endpoints/content
Failure/recovery behavior is generic	Failure needs product-specific communication or compliance
Versioning has a clear compatibility promise	The contract is still changing with the product

Versioning is a promise to consumers

When a component has more than one consumer, a version is not a tag decoration. It is a statement about compatibility. ios-framework's README points from local tandem development toward tagged package distribution, and that transition should happen only when the public surface is ready to support it.

Before publishing a reusable module, establish:

1. A supported platform/toolchain baseline.
2. A public API that does not expose private product models.
3. Fixtures and tests that prove the contract without live services.
4. A changelog and deprecation approach.
5. Clear ownership for security updates, policy changes, and release decisions.

Do not promise semantic-version stability for a feature still being redesigned every week. An internal package with a local tandem app may be the correct intermediate step. It gives the team a real boundary and tests without forcing a public distribution obligation too early.

Reuse the quality gates too

The most valuable extraction is often a quality gate rather than a runtime component. Every module can carry the same habits:

- Explicit source/evidence boundaries and no private material in public fixtures.
- Deterministic formatting, linting, unit tests, and focused UI/journey tests.
- Contract tests for validation, cancellation, route handling, and recovery.
- Privacy review for logs, metrics, persistent state, and bridge payloads.
- A reader/demo app that proves integration without becoming another product.

GamePlayer and Pushscheduler make this concrete by pairing focused source modules with quality scripts and test targets. rezona-api does it differently: manifests, checksums, collection journals, and a declared limit on what the aggregate proves. The common idea is evidence. A reusable component should make its promises testable and its unknowns visible.

Run the reuse test before publishing a package

Before a boundary earns a package name, run a short design review with a deliberately boring question: could another application use this without learning anything it should not know? The point is not to predict every future consumer. It is to find accidental dependence on the current product while the change is still inexpensive.

Start with a fixture-only integration. A small demo target should be able to construct the public types, supply a policy implementation, trigger the meaningful lifecycle transitions, and inspect typed results without an account, a production service, or a live remote page. This is the practical lesson of a tandem app: the library gets a consumer that is close enough to reveal API friction but separated enough to reveal hidden coupling. An integration that requires special environment variables, copied app models, an authenticated session, or a global route coordinator has not crossed a healthy boundary.

Then inspect the failure paths. A client package needs a cancellation result and a recoverable failure category; it should not turn an error into a brand-specific alert. A player host needs a rejection or load-failure event; it should not decide whether the app retries, offers support, or changes tabs. A notification package needs to report an invalid plan or capacity conflict; it should not choose the campaign copy. The application is the right place to translate a generic outcome into a product decision.

Finally, run an ownership review. For every stored value, event field, URL-like input, log field, and callback, ask who is allowed to create it, who can observe it, how long it survives, and what happens when it is invalid. If the answer names a user account, a provider agreement, private content, or a business experiment, the information normally belongs above the reusable boundary. This review makes reuse safer because it treats privacy and security as interface design rather than cleanup work after a package has spread.

The result does not need to be a public open-source library. A private internal package can still be successful when its contract is explicit, its fixtures are synthetic, its dependencies are narrow, and its owner can update it deliberately. The distinction is important: reusable means a capability can move without dragging product context along; publishable means a separate legal, security, support, and maintenance decision has also been made.

An extraction proposal for the next product

If a new iOS product needs a hybrid feed, do not start by importing a monolithic "arcade framework." Start by choosing the smallest capabilities it actually needs:

1. Define domain references and failure categories without URLs or account data.
2. Build a native product shell that owns routes and feature intent.
3. Add a validated catalog/client adapter with cancellation and fixtures.
4. Add a constrained player host with explicit WebView policy and typed events.
5. Add device persistence only for safe, durable values.
6. Add a local scheduler only for device-owned triggers.
7. Add redacted event instrumentation at each ownership handoff.
8. Extract a package only after a capability has a stable, tested contract and a real consumer.

This sequence produces an architecture that can grow in the same direction as the product: more content, more routes, more providers, or more devices without multiplying global state and undocumented cross-runtime messages.

Generalized MemeArcade view

The approved MemeArcade conclusion is that reusable architecture is the extraction of stable responsibility contracts—state, persistence, client, player host, WebView policy, scheduling, and observability—while product-specific data, content, routes, business rules, and private implementation remain in the application boundary. The private product is not a reusable SDK and is not published through this book.

No private package graph, source, module name, API, endpoint, payload, catalog, account data, notification copy, metric, security

policy, or business logic is approved for publication. Any private excerpt or exact implementation claim requires explicit human approval.

Reader activity: propose one honest extraction

Open ios-framework, GamePlayer, and Pushscheduler directly. Choose one capability from this chapter and write a one-page extraction proposal: public inputs, outputs, lifecycle, forbidden dependencies, fixtures, tests, and the product policy that stays outside.

The expected observation is that a reusable component becomes smaller as its contract becomes clearer. If the proposal needs private account models, endpoints, content, or global navigation to work, it is not ready to leave the product boundary.

Explore the original public source

QR code for the original public source

Source: https://github.com/hassanvfx/ios-framework

Repository path: ios-framework/README.md

Try: Propose an extraction with explicit inputs, outputs, lifecycle, forbidden dependencies, fixtures, and product policy that stays outside.

Expected: A reusable contract with an explicit private and product-specific boundary.

Appendix A: Architecture Atlas

This atlas is a responsibility map for the book's case-study vocabulary. It is deliberately not a private MemeArcade diagram or package graph. The names below describe roles that recur across the public companion projects and the generalized architecture discussion. They let a reader reason about ownership without receiving private source, endpoints, catalog data, or product policy.

The composition view

User intent
→ native product shell
→ route and session policy
→ feature capability
↘ durable state
↘ cancellable client/catalog boundary
↘ hybrid player and WebView policy
↘ local notification scheduler
→ typed result, recovery state, and redacted observation

The native shell owns the application's visible state and its user-facing decisions. A capability owns one narrow behavior. Infrastructure adapters turn durable storage, transport, WebKit, or notification-system APIs into narrow contracts. This direction matters: the storage adapter cannot select a tab; a pager cannot decide a campaign; a remote page cannot own account or route policy.

Area	Owns	Does not own	Useful test seam
Product shell	Intent, route, screen state, recovery presentation	Raw WebKit/URLS ession details	State reducer or view-model fixture

Area	Owns	Does not own	Useful test seam
Domain/ policy	Valid references, allowed transitions, failure categories	Brand copy and private credentials	Pure unit tests
Storage adapter	Restore/save validated device state	Navigation or account decisions	In-memory/ temporary store
Client boundary	Request lifetime, decoding, mapping to domain result	Backend inference or raw JSON in views	Stub transport and cancellation test
Player host	Native pager lifecycle, one bounded remote stage	Catalog ranking and global route ownership	Synthetic stage reference/event sink
Web policy	Navigation/ response acceptance and capability boundary	Private host lists or exceptions	Allow/reject table
Scheduler	Device-owned notification-plan projection	Server/APNs behavior or marketing policy	Pending-request fixture
Observation adapter	Redacted transition/failure event	User content, tokens, full URLs, raw payloads	Event schema snapshot

Runtime flow

A generalized session is not an exact claim about the private app. It is a safe sequence of handoffs that readers can use to review any hybrid iOS feature:

1. The native shell accepts a user intent and reads only validated restoration state.
2. A feature starts cancellable catalog or service work under an owner with a known lifetime.
3. The result becomes a domain reference, not an unbounded backend object.
4. A native pager may mark a cell as a candidate, but only the primary cell is entitled to activate a remote stage.
5. The WebView policy evaluates navigation and response behavior; remote content remains external input.
6. A native action is returned as a typed request, then the product shell decides whether it is allowed.
7. A device-owned schedule, if any, is validated and projected into local notification requests.
8. Re-entry repeats validation rather than trusting stale routes or payloads.

At every arrow, record a small transition event without user content. The same map supports debugging: if a session fails, first ask which owner had the value, what validation ran, and which recovery behavior was available.

Persistence and notification maps

Durable storage has a different clock from interface state. A screen can disappear while a save is pending; a process can be interrupted between a write and the next launch. Store only values that the device can safely own, restore into a validated representation, then let product state decide what to display. Encryption, key handling, migration, retention, and reset behavior must be intentional rather than accidental consequences of a convenience API.

Local notifications follow a similar projection pattern:

validated device plan → named local requests → system delivery → validated route request → native product decision

The plan is not the same thing as system requests, and receipt is not permission to navigate automatically. This distinction prevents duplicates, stale delivery, and untrusted payloads from acquiring more authority than they deserve.

Hybrid trust model

The hybrid boundary separates two runtimes. Native code owns device policy, product navigation, local state, and capability grants. Remote content owns only its authorized presentation and interaction surface. HTTPS is a transport baseline, not a sufficient trust decision. A policy should make schemes, hosts, response behavior, storage mode, JavaScript allowances, bridge capability, and lifecycle reset behavior explicit. It should be reviewed whenever a capability expands.

The public GamePlayer example makes this shape inspectable: native paging and primary selection remain native, while one stage hosts a constrained remote session. The book does not publish private origins, bridge contracts, headers, cookies, or remote-game behavior.

How to use the atlas

Use the tables as a design-review checklist. For a proposed feature, name its owner, lifetime, input validation, output type, persistence rule, trust boundary, observation fields, and safe failure state. If a component needs an app singleton, a private model, or a live account simply to exercise its contract, move that product concern back to the composition root. A clean map is not a promise that every part is a Swift package; it is evidence that responsibilities can be tested and changed without hidden authority.

Architecture review worksheet

Use this worksheet before accepting a new dependency or remote capability. It is intentionally short enough to complete during planning, then revise during implementation.

Prompt	A useful answer looks like
What event begins the work?	A user intent or validated restoration, not an incidental view redraw
Who owns cancellation?	One named feature/session owner with a known end-of-life event
What crosses the seam?	Small domain types and typed failures, never raw credentials/payloads
What is durable?	A device-safe representation with a migration/rejection path
What is remote?	A constrained input surface governed by explicit navigation/capability policy
What can be observed?	Redacted lifecycle outcomes with a retention and access rule
What is the safe failure?	A visible native recovery state that does not invent data or authority

If any row requires a private implementation detail to be intelligible, document the category rather than publishing the detail. The purpose of an atlas is not to expose more architecture; it is to make the architecture that can be responsibly discussed coherent.

Change-impact pass

When a capability changes, trace its impact through the map before merging it. A new durable field can affect migration, encryption, restore validation, UI state, logs, and notification routes. A new WebView capability can affect allowlists, process lifetime, accessibility, incident response, and the private data a remote page might receive. A new client result can affect cancellation, cache invalidation, error presentation, and reader-facing claims. The map does not prescribe a single implementation, but it makes downstream owners visible early enough to involve them.

This pass is also the appropriate place to separate a general teaching diagram from a private design. The appendix may state that a product composition root supplies policy to an adapter. It may not publish a real private target graph, internal service name, endpoint, host list, message format, or production measurement without explicit human approval. Treat that approval as a release gate, not a writing convenience. A diagram becomes more useful when its edges are labeled with responsibility, validation, and lifetime rather than private identifiers that other readers cannot safely reuse.

Appendix B: Companion Repository Activities

The companion is a reader bridge, not an online duplicate of this book. The canonical prose lives in the print corpus. Each chapter-end panel and GitHub Pages activity opens the original public repository or article directly, then provides a bounded task and an expected observation. This design keeps the public source authoritative for its own code and keeps the book from becoming a stale fork of a tutorial.

How the bridge works

One committed manifest records, for every activity, the chapter, original source URL, optional article, repository/path, task, expected observation, evidence sheet, and QR identity. Generators use that one record to create the small GitHub Pages activity page and its printed QR panel. A broken link is therefore a build failure, not a correction that must be made independently in the site and the manuscript.

The QR code encodes the direct source URL. It does not encode a private application address, a tracking identity, or a page that mirrors source content. If the reader has the print book without a current site build, the code still points at the original material.

Chapter area	Direct public source	Reader task	Expected result
Author context	WWDC14/ Ultrakam article	Separate author history from app claims	A scope note
Native shell	SwiftUI/ Combine article	Trace a publisher-to-view transition	A state-ownership sketch

Chapter area	Direct public source	Reader task	Expected result
Modularity	ios-framework and SPM article	Map library, test, and tandem-app direction	A dependency sketch
Persistence	ios-storage and DataStore article	Trace restore and observation boundaries	A restoration boundary
Async work	receipe-app	Follow request, cancellation, and visible failure state	An async ownership sketch
Network observation	rezona-api	Separate observed behavior from inference	A redacted request model
Web/player	GamePlayer	Examine policy, pager, reuse, lifecycle, and event boundaries	A bounded lifecycle diagram
Notifications	Pushscheduler	Identify plan, permission, request, and route ownership	A local-notification state flow
Extraction	ios-framework	Write a contract/fixture/forbidden-dependency proposal	A reusable boundary

A reader's evidence notebook

For every activity, write five short fields: the source revision or publication date; the file or section inspected; an observation; the claim the observation can support; and a limit on what it cannot support. This notebook prevents an attractive README sentence or an observed response from becoming a claim about an entire production system.

For source code, distinguish a public teaching implementation from a production prescription. Inspect its test target, configuration, error paths, and documented limitations before deciding a pattern is reusable. For an article, distinguish explanatory intent from a verified measurement or market claim. For a network observation, preserve only authorized, minimized, redacted evidence and never infer backend behavior that cannot be observed.

Activity safety rules

Do not use the activities to probe systems without authorization. Do not submit credentials, copy cookies/tokens, publish payloads, redistribute game content, or treat a source repository as permission to access a live service. The Rezona material is an explicit boundary exercise: it teaches attributable evidence and client modeling, not a verdict about a company or backend.

GamePlayer activities may reference a catalog only after redistribution rights are confirmed. Until then, use local/synthetic references or inspect the source structure. The App Store card on the landing page is merely a public download bridge for MemeArcade, the App; it is not evidence for the book's market framing.

Maintaining stable activities

When a source moves or a repository changes shape, update the manifest and evidence sheet together, regenerate QR assets/pages, validate every direct URL, and record the result in the ClineFlow journal. Never replace a missing source with copied course prose. If

an activity can no longer be supported by a public source, retire or rewrite its claim before publishing a new print run.

The durable promise is simple: the book explains the architecture, the original public projects remain the executable references, and every bridge tells the reader exactly what to inspect and what conclusion it can honestly support.

Bridge maintenance checklist

Before an edition or deployment, verify that every activity uses a direct https source URL; repository name/path are descriptive rather than a substitute for the URL; an evidence sheet exists; a task asks the reader to inspect a bounded behavior; and the expected observation is a conclusion a public source can support. Regenerate the QR assets from the manifest and scan a sample from a rendered proof on a real phone. Then build the site in production mode and click the CTA with a keyboard as well as a pointer.

When a public article becomes unavailable, do not use an archive or copied text as a quiet replacement. Mark the activity for editorial review, identify a current public source or remove the claim, update the chapter's prose if its evidence changes, and preserve the decision in the journal. This small discipline makes the bridge a durable reference tool instead of a fragile marketing layer.

What an activity does not prove

An activity is a guided inspection, not certification of a reader's implementation. A GamePlayer exercise can demonstrate the public repository's pager structure and documented policy choices; it cannot prove the behavior, revenue, security posture, or content agreements of a different production app. A storage exercise can show a teaching implementation's persistence boundary; it cannot establish that encryption alone solves retention or account-policy questions. A network exercise can support a redacted observation of an authorized client; it cannot establish backend facts. Keeping these limits adjacent to the task helps readers develop evidence

habits instead of transferring confidence from one context to another.

The same rule protects the private case study. The bridge must never become a route to private code, internal builds, credentials, endpoints, or unapproved product documents. Any activity that would require private implementation detail needs explicit human approval and normally should be redesigned around a public teaching source. The book's value comes from the architecture lens and the inspectable companions, not from pretending a private app is a downloadable course artifact.

Appendix C: Network Observation and Security Boundaries

Network tooling is useful only when paired with consent, scope, and restraint. This appendix provides an ethical method for observing traffic that a reader is explicitly authorized to inspect. It does not authorize interception of third-party services, extraction of credentials, circumvention of platform controls, or publication of private endpoints and payloads. The Rezona public repository is used as a study in evidence discipline, not as an accusation about a company or its infrastructure.

Authorization comes first

Before configuring a proxy, record who owns the device and account, which environment may be observed, what purpose the capture serves, which people may see it, how long it will exist, and how it will be deleted. Prefer a sandbox, a test account, a mock service, or a public teaching project. If authorization is unclear, stop. Technical access is not permission.

Allowed learning outcome	Not an allowed outcome
Model a request shape with redacted, authorized evidence	Publish a production URL, token, cookie, or full payload
Confirm that a client handles timeout/cancellation/failure	Infer unobserved backend algorithms or data stores
Test a development proxy in a controlled environment	Bypass certificate controls or inspect another person's traffic
Improve a client contract and report a responsible issue	Name a company as irregular based on incomplete observation

Minimize what is collected

Start with the smallest question: for example, "Does this development client map a non-success response to a recoverable

state?" Capture only enough metadata to answer it. Never retain credentials, authorization headers, cookies, device identifiers, personal data, payment data, full URLs with sensitive query strings, or user-generated content. Replace values with type-preserving labels such as <redacted-token> or <opaque-id>; remove records immediately if they are not needed for the stated purpose.

An evidence note should contain an authorization statement, the date, tool and version, controlled environment, reproduction steps, redaction method, a narrow observation, and a limit. A good limit might read: "This shows how the inspected client behaved in this controlled run; it does not establish server implementation, other-user behavior, or production policy."

Model the client boundary

The safest useful artifact is a typed client model, not a raw capture dump. Describe a request in terms of method category, input validation, response category, cancellation, retry ownership, and mapping to a domain result. Keep the transport object inside the adapter; return a small domain value or known failure category to the feature. A SwiftUI view should not receive raw headers or JSON merely because the client had to process them.

```
validated input → owned request task → response classification
    → domain result / recoverable failure → native presentation
```

This same boundary improves hybrid apps. A WebView navigation action is external input; a host policy decides whether it is accepted. A deep link in a local-notification payload is external input; a native route parser validates it. Similar shapes make security review repeatable across network, web, and notification systems.

Proxies and MITM in development

Use tools such as Proxyman only in a development or authorized test context. A proxy certificate changes the trust environment, so document its installation and remove it when the test is complete.

Do not normalize disabled certificate validation or broad exceptions into production behavior. Test the failure case as well: an application should make a safe recovery decision when a request cannot be trusted or completed.

An observation is not a vulnerability report by itself. If you identify a plausible security concern in an authorized system, preserve the minimum evidence necessary, avoid public disclosure, and use the owner's responsible-reporting process. The book publishes no live incident details, private hostnames, or exploit instructions.

Observed, inferred, and unknown

Label every research statement. **Observed** means the authorized run or public source showed the behavior. **Attributed** means a named public source made the statement. **Inferred** means an interpretation that must remain conditional. **Unknown** means no evidence supports a conclusion. This vocabulary is especially important when a UI, API, or remote game looks simple: a client-visible response cannot prove the server's storage, ranking, moderation, financial, or security logic.

The goal is not to make inspection less rigorous. It is to make the result more trustworthy. A narrow redacted observation, paired with its limits and an improved client contract, is more valuable than an impressive-looking capture that cannot be ethically shared or reproduced.

Incident and disclosure path

If authorized testing reveals behavior that could materially affect users, do not turn the observation into a chapter anecdote or a social post. Preserve a minimal, redacted record; verify the scope without increasing collection; identify the owner; and use the organization's documented security or support channel. Include the environment, date, reproduction conditions, expected/safe behavior, observed category, and the evidence limit. Do not attach

tokens, certificates, unredacted captures, or instructions that would enable misuse.

For a book revision, the editorial question is narrower: does the public material teach a durable engineering practice without exposing a real system? If yes, use a synthetic diagram or a generalized type-level example. If no, omit it. Any private incident, endpoint, security finding, capture, or implementation reference requires explicit human approval before it can appear in research or prose, and approval is not assumed simply because an author had authorized access. A responsible technical book makes its security boundary visible precisely by refusing to make sensitive material its proof.

Appendix D: Production Trade-offs

Architecture is a sequence of choices made under product, safety, and operating constraints. The comparisons here are decision aids, not universal rankings. Start by naming the user outcome, the owner, the lifetime, the evidence required, and the failure behavior. Then choose the smallest mechanism that makes those properties clear.

Interface ownership

Decision	Prefer this when	Watch for	Review question
SwiftUI feature shell	Product state, navigation, composition, and accessibility semantics are primary	A view that silently owns service lifetimes	Which object owns the state transition?
UIKit container	Paging/ scrolling/ reuse behavior needs explicit lifecycle control	Duplicate route/product ownership	Can the container emit typed events to SwiftUI?
Native presentation	Device policy, account context, or accessibility must be authoritative	Rebuilding remote content unnecessarily	What cannot safely be delegated?

Decision	Prefer this when	Watch for	Review question
Web presentation	Content is provider-owned and constrained by an explicit policy	Treating remote navigation as trusted state	What origin/capability/lifetime is granted?

SwiftUI and UIKit are not opposing teams. The useful seam is often a SwiftUI product shell that hosts a focused UIKit/WebKit capability. That arrangement makes it easier to isolate reuse and WebView lifecycle while preserving native ownership of routes, account decisions, overlays, and recovery communication.

Loading and lifecycle

Choice	Benefit	Cost	Evidence required before adoption
Lazy primary-only loading	Lower steady resource use and clearer ownership	First activation may be slower	Baseline activation timing and failure rate
Adjacent preparation	May improve expected-next transition	More memory/network/process pressure	Target-device measurements and rollback guardrail
Broad prewarming	Can hide one latency symptom	Expensive, stateful, hard to cancel	A demonstrated user outcome that smaller fixes cannot achieve

Choice	Benefit	Cost	Evidence required before adoption
Reuse/reset	Bounded allocation and predictable cells	May discard in-progress remote state	Explicit user expectation and recovery behavior

Visibility is not entitlement. A cell becoming visible does not automatically deserve a network task, WebView, or analytics session. Candidate, primary, active, leaving, and reused states provide a vocabulary for deciding which resources may exist. Measure on representative hardware, include interruption/memory-pressure paths, and keep a rollback path before treating a performance experiment as architecture.

Data and concurrency

Choice	Good fit	Failure to prevent
Actor/service boundary	Shared transport/cache work with concurrent callers	Mutable cache races and unowned tasks
@MainActor view model	Visible state updates and cancellation on screen lifetime	Background mutation of UI state
Durable encrypted store	Device-owned values that survive relaunch	Saving secrets or unvalidated state by convenience
Ephemeral WebView store	Remote surfaces where cookie/history persistence is unnecessary	Cross-session state leakage

Do not cache merely because a response is expensive. Define freshness, capacity, invalidation, cancellation, and what data is acceptable to retain. A restore is an untrusted input from an earlier version of the application; it should be decoded, migrated or rejected, and mapped to a safe visible state.

Notifications and re-engagement

Choice	Use when	Keep separate from
Local notification	The device can schedule a known reminder without a server	APNs token/remote-campaign assumptions
Remote notification	A service needs to decide delivery from server-side state	Automatic route authority on receipt
Durable plan	The feature needs reconciliation across launch	System pending requests, which are a projection
Typed route request	A tap should be evaluated by native product policy	Direct navigation from opaque payload text

Request permission in context, show the denied state gracefully, namespace only the requests a feature owns, and validate a route again at tap time. Notification copy, targeting, timing, and growth policy are product-specific; a scheduler component should never smuggle them into a generic API.

Observability and publishing

Metrics answer whether something changed at aggregate scale; logs help reconstruct a bounded event; traces join a controlled lifecycle. None should routinely include raw URLs, tokens, cookies, payloads, account identifiers, or user content. Start with event names, outcome categories, duration buckets, and a short-lived

non-user correlation strategy. Establish retention, access, and deletion rules before adding data.

For the book, the comparable trade-off is convenience versus provenance. A second web copy of the manuscript would be convenient but diverges. The reader bridge instead points to original public sources and records task/evidence/QR data in one manifest. The print pipeline favors deterministic generated artifacts, rendered-pagination TOC entries, and an isolated LibreOffice-headless final export over manual page numbers or an untracked hand-edited PDF.

Appendix E: ClineFlow Research Journal

The journal is the execution record for this interior-only publication. It answers a practical question that polished books often leave invisible: what evidence, decisions, tools, and approvals produced this exact manuscript and PDF? It is not a diary of every keystroke. It is a compact, reviewable record of material choices.

What belongs in the journal

Every chapter or production milestone records the date, scope, source evidence, resolved public revisions, claims allowed by that evidence, excluded/private material, artifacts created, validation commands/results, human approvals needed, and next action. Material changes also receive a shorter entry in knowledge/log.md, making the project history easy to scan without replacing the fuller record.

Record	Purpose	Example question it answers
Source lock	Pin a public repository URL and commit	Which GamePlayer revision supported this wording?
Evidence sheet	Separate observation, claim, limit, and approval	Does this statement describe evidence or inference?
Decision record	Preserve a consequential editorial/technical choice	Why do QR codes open originals rather than a local course page?
Artifact record	Identify generated DOCX, PDF, QR, or manifest	Which inputs created the review master?

Record	Purpose	Example question it answers
Validation record	Capture command and result	Did source links and the Docusaurus build pass?
Approval gate	Name a decision requiring a human	Has private-code wording or final PDF art been approved?

The chapter-close protocol

Before drafting, create or update an evidence sheet. Read the locked public source, identify only claims it can support, and write the private boundary in plain language. Draft from the canonical Markdown file, not from a duplicate web page or generated print copy. Add a direct reader activity only when a stable public source can support it.

At the chapter close, check the editorial contract: junior explanation, a small model/diagram, public implementation, generalized MemeArcade view, production trade-offs, reader task, sources, and an explicit limit. Update the journal and knowledge log, run ./validate-okf plus the relevant book/bridge checks, then record their results. A passing structural command is evidence of structure; it is not editorial approval or a factual-market review.

Approval gates are deliberate

Automation cannot decide whether market prose is fair, a diagram is legible, a private reference is sufficiently generalized, or a print page feels right. This project requires a human decision before publication for market claims, any private-code excerpt, manuscript text, interior artwork, final visual PDF review, Lulu configuration, and the physical proof.

Use clear statuses: **draft**means written but not approved; **evidence checked** means sources/limits were reviewed;

approved for beta means it may enter a review PDF; **approved for release** means the authorized reviewer accepted it for the named edition. Do not convert an absent approval into a checked box because a build passed.

Reproducibility and restraint

The journal also protects readers and collaborators. Full third-party article archives do not belong in Git; retain citation metadata and short permitted notes. Public repositories may be cached in ignored storage for inspection, but the committed lock preserves their source URL and commit. The private MemeArcade checkout is evidence only. Its code, secrets, internal endpoints, payloads, product data, and unapproved implementation details never become journal attachments or manuscript examples.

When a source is corrected, a link disappears, or an assumption changes, add a dated correction rather than silently rewriting history. Update the source lock/evidence sheet, revise only the canonical manuscript, regenerate dependent assets, rerun validation, and record the result. This is how a future edition can explain not just what it says, but why it was allowed to say it.

Appendix F: Reproducible Interior Publishing Protocol

This repository produces the Lulu US Trade 6×9 **interior only**. It prints the assigned ISBN in the copyright page, but does not generate an exterior cover, spine, barcode placement, or Lulu wrap template. Lulu.com is the ISBN imprint and Waken AI Labs is the editorial brand. The approved visual is an interior title plate. Keeping that boundary explicit prevents the page-count-dependent cover work from being mistaken for an interior deliverable.

One canonical corpus, several renditions

Published prose exists once in Markdown under book/chapters/ plus approved front matter and appendices. The website is not a second course: it is a bridge that directs readers to original public repositories and articles. The print system makes a temporary rendition of the corpus with generated reader panels/QR codes, then converts it through the versioned lulu-us-trade-6x9-no-bleed-v2 DOCX template. Its 1.10-inch body bottom margin is reserved above the physical folio; table rows are non-splitting, so they move to the following page rather than cross that footer band. Corrections always begin in canonical Markdown, never in the generated DOCX, QR panel, or PDF.

The public production record for this edition lives at hassanvfx/meme-arcade-book. It contains the canonical publishing corpus, source locks, validation scripts, and reproducibility records; it is not a replacement for the original companion repositories and articles linked by the reader bridge.

canonical Markdown + front matter + reader manifest
→ print rendition with generated source panels
→ versioned 6×9 DOCX template
→ LibreOffice headless PDF (authoritative)
→ normalization/flattening and Lulu audit
→ publication manifest with hashes, TOC, pages, validation

LibreOffice headless is the authoritative renderer for this project, using an isolated temporary profile for every conversion. This follows the reproducible ai-on-mac master-PDF pattern: the DOCX is rendered once through a deterministic command-line route, then committed assembly and PDF-audit scripts produce a traceable candidate. The renderer does not replace human judgment; page-by-page review and a physical proof remain mandatory.

Assembly order and contents

The interior order follows the established ai-on-mac cadence: title plate, copyright, dedication, generated contents, About the Author, acknowledgements, courtesy page, ClineFlow preamble, chapters, then appendices. Chapter openers retain their dedicated layout and chapter ends receive generated direct-source reader panels.

The table of contents is generated only after rendered pagination is available. A script reads rendered page positions, writes entries, re-renders, and verifies the final positions. Never type printed page numbers into source Markdown. A last-minute paragraph, font substitution, or title adjustment can move every later page.

Automated beta gate

Before a beta proof, run source-lock verification, Markdown/front-matter checks, chapter/evidence audits, direct reader-bridge/QR checks, the Docusaurus production build, DOCX/PDF construction, and preflight. The release manifest records manuscript/template/artwork hashes, source lock data, generated TOC entries, page count, validation output, and the master-PDF hash. It is a ledger of inputs and results, not a substitute for visual review.

Lulu interior preflight

The final PDF must have 6×9-inch portrait single pages, no encryption, embedded fonts, safe margins/gutter, and appropriately prepared placed images. Transparency is flattened where the workflow requires it; placed interior imagery is audited against the

300 ppi target. The no-bleed template is intentional. A future full-bleed interior requires a different Lulu-aware template and a changed preflight, not merely a larger image.

Review every rendered page at print-relevant size: title plate, copyright/dedication cadence, TOC numbers, headings, running heads, folios, tables, diagrams, QR readability, widows/orphans, image sharpness, and blank-page behavior. A command can confirm dimensions and embedded resources; only a human can confirm that the book reads and looks finished.

Release gates

The workflow stops for human approval at six points: market wording; any private-code reference; chapter/appendix manuscript text; interior artwork; final PDF visual review and Lulu settings; and the physical proof. Record each decision in the ClineFlow journal. Once a proof reveals an issue, make the correction in canonical Markdown or versioned assets, regenerate every dependent rendition, repeat preflight, and write a new manifest. Do not patch the final PDF by hand.

The handoff is therefore a reproducible, validated interior master and its manifest. The author separately creates the exterior wrap using Lulu's current template and the final confirmed page count.

Beta-to-master checklist

1. Freeze a named beta from canonical Markdown and record its source/template/art hashes.
2. Generate reader panels and QR codes from the direct-source manifest; do not hand-edit them.
3. Render the DOCX through the authoritative LibreOffice-headless path to discover layout issues and produce the candidate master.
4. Derive the contents from the rendered pagination and rebuild until its entries agree with the final rendered pages.

5. Export the candidate master from LibreOffice headless with its isolated profile, then run exact page-size, single-page, font, image, transparency, margin/gutter, and encryption checks.
6. Render every page to images for a human page-by-page review. Check the title plate, opening cadence, tables, code, diagrams, QR contrast, and blank pages as carefully as prose.
7. Write a release manifest only after all automated gates pass, then obtain the required editorial/PDF/Lulu approvals and order a physical proof.

The proof is a test of the actual printing system, not a ceremonial final step. If it reveals clipped headings, a margin issue, soft image, unreadable QR, an awkward blank, or a pagination change, correct the source and rebuild the entire chain. The reproducibility record is valuable precisely because it makes that iteration controlled rather than mysterious.

Page-by-page visual review rubric

A preflight tool can prove page dimensions, but it cannot tell whether a human reader can comfortably use the book. Render the candidate PDF to page images and review it in sequence. Begin with the structural pages: the title plate should look intentional at 6×9, copyright and dedication should have breathing room, the contents must agree with printed folios, and About/Acknowledgements/ClineFlow material should follow the planned cadence. Check each chapter opener for the blue title/rule/folio treatment inherited from the established production pattern. Check each chapter ending for a reader panel that clearly names the original public source and has enough QR contrast and physical size to scan.

For body pages, inspect the first and last line of each page, heading isolation, widow/orphan behavior, table wrapping, code samples, diagrams, URLs, and footnote/citation legibility. A forced page break can be correct when it protects a chapter opener, but a blank

page that results from accidental overflow needs a source-level correction. Review at normal reading scale as well as zoomed scale; a page may be technically complete yet look crowded, asymmetrical, or visually abrupt at print size.

Image checks are editorial and technical. Confirm every placed image is intentional, attributed or approved, sharp enough for the final scale, and inside safe margins. Confirm that decorative art is limited to approved interior usage. This project contains no external front-cover proof: the turtle plate is an interior page, while the author separately owns the exterior wrap, spine, barcode, and Lulu cover-template work. Do not let a PDF thumbnail or a site image create the misleading impression that a cover has been delivered here.

Run a second pass specifically for accessibility and reader handoff. Large titles need sufficient contrast; tables must retain their column meanings when they wrap; code must not rely exclusively on color; QR panels must include a printed direct URL or source name so a reader is not blocked by a camera failure. Links on GitHub Pages should be keyboard reachable, use descriptive labels, and open original sources rather than a manuscript mirror. The app-download card must continue to state that MemeArcade, the App is the technical case study and that the $3B label is broader market framing, never an app valuation.

Finally compare the rendered master with the release manifest. The corpus hash, DOCX-template hash, interior artwork hashes, page count, final TOC entries, validation results, renderer declaration, and PDF hash should all match the candidate under review. If the document changes after review, it is a new candidate: regenerate the manifest, rerun the audit, and review the changed pages plus any pages whose pagination moved. This may seem strict, but it is the difference between a reproducible interior and a file that merely happened to print once.

Edition control

Give every review artifact an edition label and a clear status: draft, beta, release-candidate, or master. A filename alone is not enough; record the status, date, source revision, exporter, and manifest hash in the journal. A beta may be shared for manuscript and layout comments, but it is not upload-ready. A release candidate is technically complete but still awaits visual/Lulu approval. A master is the exact approved interior whose manifest and PDF hash are handed to the author for the separate cover-wrap workflow.

Keep corrections narrow and traceable. If a reviewer changes a market sentence, revise its evidence sheet and citation record. If a page break changes, regenerate the TOC and inspect later folios. If a source bridge moves, update the manifest, QR asset, activity page, and chapter link together. If interior art changes, update its hash and repeat image review. These rules avoid a common publishing failure: a correct source tree paired with a stale PDF, or a correct PDF paired with a stale manifest.

Do not overwrite the previous approved master. Retain its manifest and review notes so that an edition can be reconstructed, compared, or rolled forward with intent. The process is deliberately conservative because print interiors have a long tail: readers, reviewers, and Lulu may interact with a file long after the repository has moved on.

One final rule keeps the system honest: a successful command is a recorded fact about that command, not an approval proxy. The source audit, site build, QR generator, DOCX conversion, PDF inspection, and manifest all reduce preventable errors. None can decide that a market sentence is fair, a private boundary is adequately generalized, a page is attractive, or a physical proof is acceptable. Those are human decisions, named and dated in the journal before the interior is called final.

That separation of automated evidence from human judgment is the publishing system's final control: reproducible mechanics serve editorial accountability, rather than replacing it.

It also keeps later editions understandable, auditable, and safe to revise.

That durable trail lets an editor verify exactly what was printed, why it was accepted, and which source changes require renewed review.