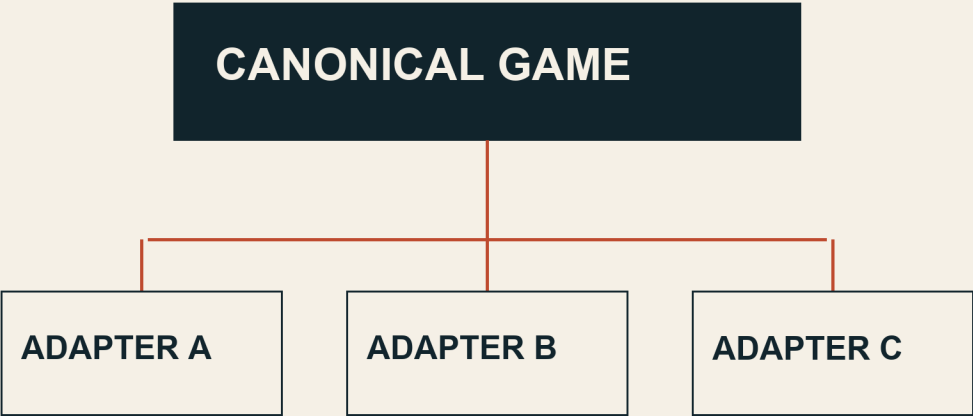


Browser Game Release Engineering

Ship One Canonical Game
Across Portals with Adapters,
QA, and Release Evidence



Browser Game Release Engineering

FREE SAMPLE — opening and Chapters 1–2. Ship One Canonical Game Across Portals with Adapters, QA, and Release Evidence

Xiaoqing Wang • Edition 1.0 • October 1, 2026

Copyright © 2026 Xiaoqing Wang. All rights reserved. AI-assisted development is disclosed in the edition note. Provider requirements are dated and qualified. Examples are learning material, not production certification.

Contents

A release manual for one game and many hosts	4
1. Keep the product canonical and the packages distinct	5
2. Model capabilities and failures explicitly	8
What the full manual adds	11

A release manual for one game and many hosts

Your game works in the development tab. One portal needs a lifecycle callback, another provides storage through its SDK, and a third wraps the game in an iframe that changes audio behavior. You copy the build, patch the button, and upload again. A month later, five "versions of the same game" have different save behavior and nobody can identify the bytes players actually received.

This opening is an illustrative composite, not a claim about a particular historical release. The problem is real enough to design against: distributing one browser game should not require maintaining a separate product codebase for every host.

This manual helps indie developers ship one browser game across multiple HTML5 portals without turning every platform integration into a separate codebase and release process. It assumes you already have a playable game and can build JavaScript or TypeScript projects. It is not an engine tutorial or a portal-by-portal upload directory.

The causal spine is one canonical product, an explicit platform boundary, isolated adapters, qualified packages, hosted verification, telemetry, and retained release evidence. Chapters 1-3 define the boundary. Chapters 4-7 qualify what players experience. Chapters 8-10 carry that proof through submission and updates.

First-party notes draw on retained browser-game code and release reports. They identify whether evidence was code, local QA, a hosted test, or a dated publication receipt. A report that a build was ready is not evidence that it was submitted. A Sandbox publication is not proof of main-catalog acceptance. A media clock advancing is not a recording of speaker output. These distinctions are part of release engineering.

All provider requirements described as current were checked against official sources on October 1, 2026. The portal cards are volatile. The architecture and evidence discipline are designed to survive their replacement. Scenarios labeled illustrative are teaching exercises, not invented case studies disguised as experience.

1. Keep the product canonical and the packages distinct

"Same game" needs a definition

A canonical game is the source of product truth: rules, content, progression, canonical state, interaction intent, and product version. A platform package is a composition of that game with a profile, adapter, host configuration, and build policy.

One source does not imply one binary. A portal may require a different SDK, asset base, packaging shape, or authentication boundary. Distinct packages are healthy when their differences come from explicit composition. Separate game implementations are expensive when they arise from scattered conditionals and hand-edited export directories.

Use this architecture as a reasoning aid:

```
canonical rules + content + save schema
      |
    shared platform boundary
      |
selected adapter + platform profile
      |
immutable qualified package
      |
hosted QA -> distribution -> telemetry
      |
    retained release evidence
```

A gameplay bug belongs in the canonical game. A portal callback mapping belongs in its adapter. A portal's required archive shape belongs in packaging. The distinction matters because the remediation should change the smallest truthful boundary.

The expensive failure chain

Illustrative chain: a portal rejects a game because Continue freezes after an ad request. The developer patches the exported JavaScript. The portal accepts the patched file. The source still contains the old behavior. The next canonical build overwrites the fix. Meanwhile, a second portal gets the old save schema because its copy was never updated.

The first failure was a callback. The maintenance failure was the unowned patch. The release failure was the missing identity between source, package, and hosted game. Solving only the callback leaves the other two intact.

Record a product version and a package identity separately. Product version tells a reader or player what experience changed. Package identity records which adapter/profile/toolchain/configuration composed it. The archive checksum identifies the actual bytes. None substitutes for the others.

A first-party composition seam

In an audited implementation, the bundler resolves one `@platform-adapter` import at build time. The shared bridge consumes that import; puzzle code does not choose SDKs [B1]. This is observed code, not a claim that every resulting portal package was accepted.

The lesson is portable: create one narrow composition seam, then prove the unselected SDKs are absent. A runtime `if (portal === ...)` can still work, but it often bundles every integration and makes static review harder. Compile-time selection is particularly useful when portals prohibit competing branding, ads, or SDKs.

Responsibility matrix

Concern	Canonical/shared owner	Adapter/profile owner
Rules and win state	Game	No independent rule changes
Save format and migrations	Game save service	Storage transport and host scope
Pause and input reasons	Shared lifecycle controller	Host/ad callbacks
Ad placement intention	Product policy	Supported request and result mapping
Reward grant	Game consequence with identity	Evidence that the contract's reward condition occurred
Audio preference	Shared audio service	Host mute and ad playback signals
Asset resolution	Shared loader	Build base and hosting package
Telemetry meaning	Game measurement contract	Destination, transport, and capability

A no-ad portal profile does not need a new game fork. It needs a policy for a missing capability. A host without durable storage needs an honest persistence mode. A vendor SDK should not redefine your game's meaning of completion.

The canonical-game inventory

Before integrating another portal, name the authoritative source for rules, content, save schema, product version, localization, art/audio licenses, and telemetry events. Identify any existing hand-edited output as historical evidence. Reconstruct its source or retain it as an exception; do not silently treat it as canonical.

Then list every platform conditional in gameplay code. Move only those that express host behavior. Keep legitimate product decisions visible. If a portal requires a product adaptation that changes fairness or progression, document it

as an explicit approved variant, with separate QA. "Adapter" is not permission to hide a second game.

First implementation action: choose one existing portal package and trace its source, profile, adapter, archive checksum, and hosted state. The first broken link tells you where to repair the release process.

2. Model capabilities and failures explicitly

A Boolean support flag is too weak

A platform may support rewarded ads while having no ad available in this session. Its SDK may load while host authorization fails. Storage may be supported but temporarily unavailable. A local fallback may allow play without establishing persistent progress.

Separate declared support, initialized readiness, and per-operation outcome. This small conceptual interface avoids leaking SDK types into product code:

```
type Outcome = "ok" | "unsupported" | "unavailable" | "failed";
type AdResult = {
  outcome: Outcome;
  rewardEarned: boolean;
};
interface PlatformPort {
  init(): Promise<Outcome>;
  readonly capabilities: {
    rewardedAds: boolean;
    commercialAds: boolean;
    storage: "none" | "local" | "host";
  };
  requestAd(kind: "rewarded" | "commercial"): Promise<AdResult>;
  readSave(key: string): Promise<unknown>;
  writeSave(key: string, value: unknown): Promise<Outcome>;
}
```

This is an interface sketch. A real storage result should distinguish absent, failed, malformed, and loaded data; a real ad interface also needs lifecycle callbacks and request correlation. The point is that a promise resolving must not mean that an economic or persistence outcome succeeded.

Choose fallback policy at the product boundary

Missing capability	Product decision	What to disclose or prove
Commercial ad unavailable	Continue through the natural break	Navigation does not depend on ad fill
Rewarded ad unavailable	Hide, disable, or offer a separately named fallback	Do not claim the player watched an ad
Host storage unavailable	Continue in an allowed local/session mode or hold safely	Visible save status; no false cloud-save claim
SDK initialization fails	Apply profile-specific degraded mode	Required integrations may block submission
Custom telemetry unavailable	Keep play working; mark measurement limits	Do not invent a funnel from absent events

A free hint fallback can be a legitimate product choice, but it is not an earned advertising reward. Give it a separate consequence identity and user message. Otherwise analytics will report a reward conversion when no ad was shown.

Initialize once, degrade deliberately

Make initialization idempotent inside your adapter. Concurrent callers should receive one initialization result. Register lifecycle listeners once, remove them on disposal, and bound your waiting policy. A loader that adds a new script tag on every retry can create duplicate event handlers even if only one visible game exists.

The platform profile decides whether initialization failure blocks release or allows play. A generic web profile can operate without a portal SDK. A portal that requires integration cannot be certified simply because a fallback kept the game alive.

Playgama’s official Bridge project documents initialization before use and a shared interface for several platform capabilities [D4]. That can reduce integration work. It does not eliminate the need to test each host context. An abstraction’s safe local defaults can conceal missing hosted behavior if you mistake them for production success.

Store the adapter contract card

For each profile, record required SDK/version or loading strategy, initialization evidence, capabilities, callback ordering/correlation, lifecycle semantics, storage scope, ad result rules, packaging shape, and current official sources. Separate requirements from recommendations.

Give the card a checked date and bind QA to its version. If a documentation change alters reward completion or storage behavior, you know which adapter tests and book passages need review. A living support matrix without dates is a source of false confidence.

Audit for accidental authority

Search for SDK globals inside game components, direct `localStorage` calls outside the save service, telemetry sends inside reducers, and unconditional resume/unmute calls after ads. These are likely places where a host detail has gained product authority.

A first-party platform-status document from September 14 later disagreed with current source about adapter availability [B2]. The useful lesson is not that documentation is worthless. It is that "supported" must name a snapshot and a proof level: registered in code, locally tested, hosted tested, submitted, or accepted. A single green badge cannot carry all five.

What the full manual adds

The full edition continues into lifecycle ownership, ad callback failures, audio and loading recovery, storage hydration, responsive input QA, immutable packaging, rejection diagnosis, telemetry denominators, and new-platform onboarding. Hosted first-party failures and the complete release artifacts remain in the paid edition.

Sample source notes

D4: [Official Playgama Bridge initialization and capabilities](#). B1: audited bundler/shared-bridge source establishes one selected adapter, not portal acceptance. B2: a dated status document differed from later adapter registration; code existence does not establish hosted support.

This sample contains the opening and the first two complete chapters, selected to demonstrate the manual's reasoning before purchase. The sample is independently usable; the paid edition extends this reasoning into release and recovery artifacts.