Release Testing on Pawn to Don

QA Case Study · Funcell Games

Owning full release-candidate validation on a shipped mobile RPG, and tracing a bug with nothing in the logs back to its actual cause.

Role	QA / Release Testing (embedded in development)
Studio	Funcell Games
Title tested	Pawn to Don — Unity, mobile
Platform	Android
Outcome	Shipped — 1M+ downloads, 4.4★ (21.2K ratings)
Timeline	Oct 2024 – Nov 2025

Overview

Pawn to Don is a 30-day branching narrative crime RPG: the player has 30 in-game days and a $100,000 target, with choices splitting between a legal grind and criminal work. That branching structure is exactly what makes a game hard to test properly. There's no single "the game," there are dozens of paths through it, and a bug three choices deep down one branch can be invisible on every other path.

I was embedded in development at Funcell rather than sitting in a separate QA function, which meant I owned release-candidate validation end to end: performance, functional, and regression passes, analytics and content-delivery verification, and go/no-go sign-off before every Play Store submission. This case study covers that process, plus one investigation that's a better demonstration of how I actually work than any checklist description would be.

The release validation process

Every release candidate got a full validation pass before it went to the Play Store: roughly 45 minutes to 1 hour 45 minutes per pass, run across 3 devices spanning low-end to high-end.

Device and platform checks

Startup time across devices, frame rate consistency against a 30fps floor, and screen orientation (flipping landscape/portrait to catch UI breaking, a common Android failure mode).

UI and settings

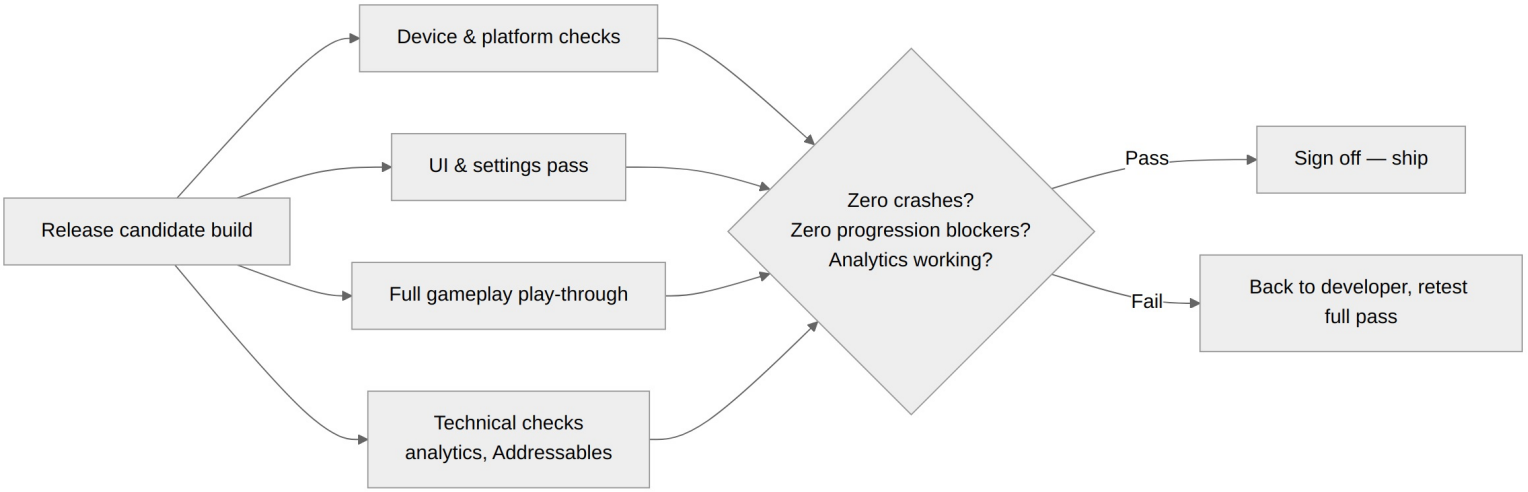
Every element correctly positioned, visible, and interactive across the main menu and settings screens; volume sliders, mute, and adjustable settings actually persisting between sessions.

Gameplay

A full play-through logging any gameplay bugs, plus level loading/unloading, rendering artifacts, and visual glitches.

Technical

Analytics events firing correctly, and Addressables/downloadable content loading properly. That last one had bitten the project before, so it stayed on the checklist permanently.



Release validation flow: every candidate build runs through four checks before a go/no-go decision.

Sign-off bar: zero crashes, zero progression blockers, working analytics events. Minor cosmetic issues could ship as known issues, logged for the next build rather than blocking release. If a build failed, it went back to the developer and I'd retest with a full pass, not a spot-check.

> **Why this mattered for efficiency:** A full replay of a 30-day branching story to verify one fix is expensive. I built a library of save files at key story points, so regression testing meant jumping straight to the affected section instead of replaying hours of content to reach it. That's the same

> underlying need that led to the checkpoint tool described in the development-side case study — the tester and the tool-builder were solving the same problem from two directions.

Bug tracking and regression

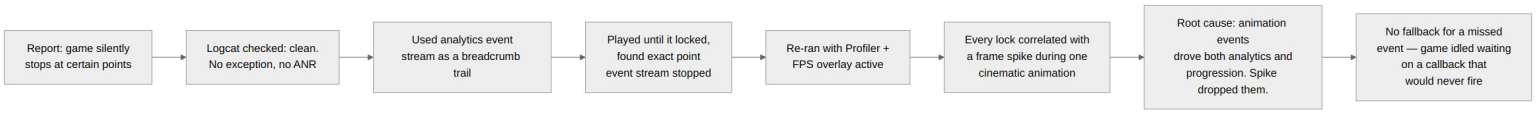
Bugs were tracked in Jira, with severity and priority logged separately, since a bug that's severe (breaks the game badly) isn't always urgent (blocking the current milestone), and treating them as the same field loses information. Every report included repro steps, build number, platform, and a save file or capture where possible.

We triaged as a group with the developers roughly twice a week, re-prioritizing based on what was actually blocking. Tickets moved through In Progress, Blocked, Testing, and Complete — anything landing in Testing came back to me to verify against the next build before it closed.

Each new build got a full start-to-finish pass looking for crashes and progression blockers first, then a targeted pass down the priority list to verify fixes in order before touching lower-severity items.

Investigation: the bug with nothing in the logs

The report was simple and unhelpful: the game would sometimes just stop. Not a crash, not a black screen — it rendered fine, accepted input, and did nothing. No exception. No ANR. Logcat was clean, which is what made it interesting: nothing had technically failed.



Investigation trail: from a silent report to an identified root cause.

What I did have was the analytics events, since the developers fired those through the same event system driving game progression. I used them as a breadcrumb trail: play until it locked, then find exactly where the event stream stopped. That gave me a precise point in the sequence instead of "somewhere in the cutscene."

From there I re-ran it with the Profiler and the FPS overlay up. Every lock correlated with a frame spike at the same moment: one of the heavier cinematic animations. Dialogue was synced to that animation, and animation events were driving both analytics and the actual state transitions. When the spike hit, events got dropped. Because those events were the only thing telling the game to advance, a missed one left it waiting on a callback that was never coming. Nothing had errored. It was idling in a state with nowhere to go.

Optimizing the animation removed the spike and fixed the immediate symptom. But I flagged that this wasn't really a fix, it removed the trigger, not the fragility. The critical path still advanced only on animation events with no fallback, and any device slow enough to spike would hit the same failure. We'd only validated on three devices, and the install base was going to be much wider than that.

> **Why I'm flagging this as the case study's centerpiece:** finding the bug is table stakes for QA. Recognizing that the fix addressed a symptom rather than the underlying fragility, and being able to say precisely why, is the difference between "found a bug" and understanding a system well enough to know when a fix is incomplete.

Backend and content verification

Verified analytics events end to end: confirmed they left the client via logcat, then confirmed in the backend that they landed with correct user data and values. Also tested Addressables and DLC delivery (server-hosted content) across the device range, including load correctness after download.

Tools

Jira · Unity Profiler (draw calls, CPU frame time, memory) · Android logcat · in-build FPS overlay · save-file checkpoint library for regression · stopwatch/log-timestamp load-time measurement across devices

Outcome

The title shipped and has since crossed 1M+ downloads at a 4.4★ average across 21.2K reviews. None of that is proof any single bug mattered, but it's the context the testing happened in: a live audience at scale, which is exactly where an unresolved soft-lock or a missed regression would have surfaced fast.