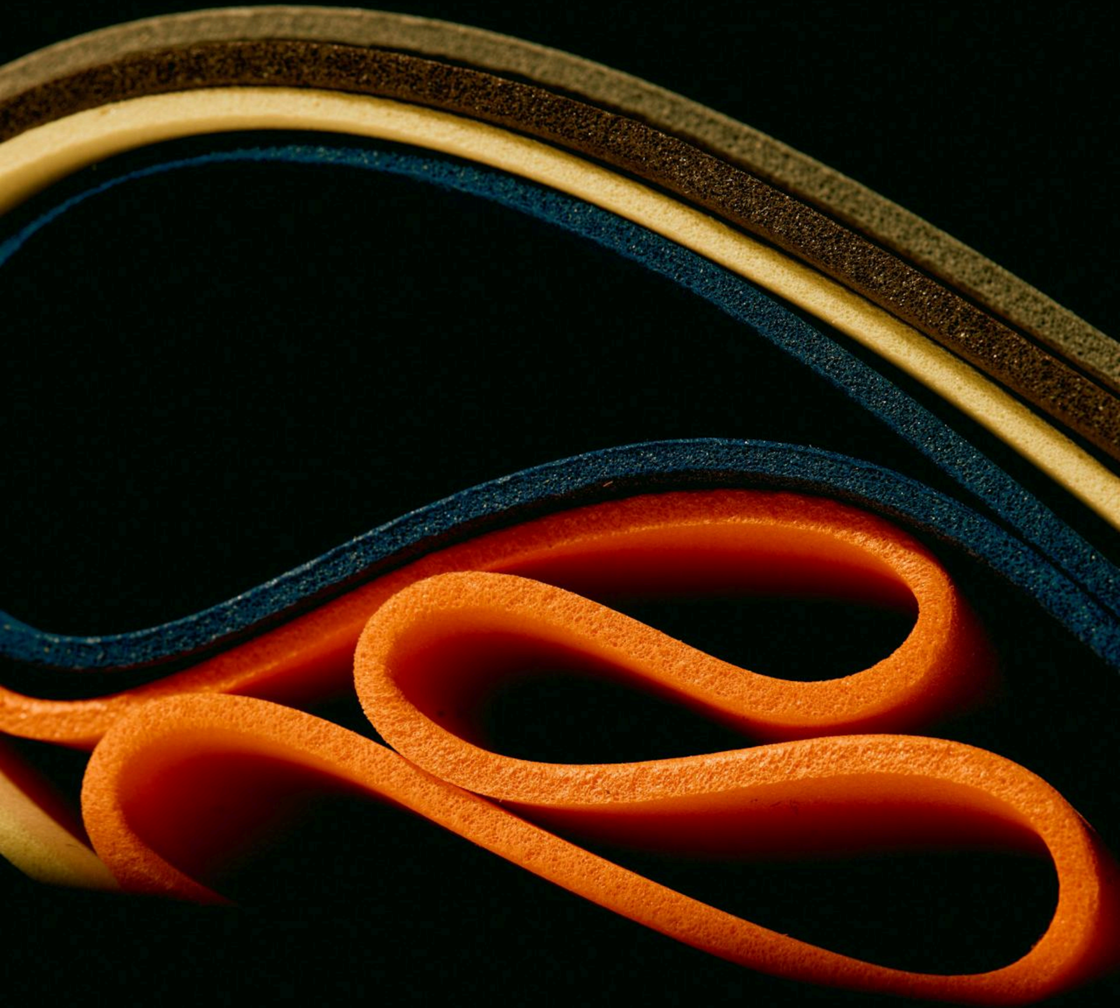
WHITE PAPER

# Your game engine is not your LiveOps engine

How to decouple your LiveOps pipeline from your game engine



Reaktor

# Index

About the author
P — 03

Introduction
P — 04

The hybrid UI model
P — 06

Architecture deep-dive:
building the invisible interface
P — 09

Platform-specific realities
P — 12

Use cases
P — 14

Addressing the objections
P — 16

The agility gap self-assessment
P — 18

About Reaktor
P — 20

Jussi Enroos is a Lead Developer and Gaming Technology Advisor at Reaktor. He works on ambitious projects to help Reaktor's Gaming and Entertainment clients improve their technology capabilities and value streams.

**Jussi Enroos**
Lead Developer,
Reaktor Gaming

# Introduction

## One pipeline – two competing demands

Game engines need stability. Stability requires long QA cycles and deliberate release schedules. LiveOps content needs speed and the ability to react to what's happening right now. Most studios have some degree of decoupling through remote configs and asset bundles, but UI layouts and richer content experiences are still baked into the client. LiveOps content that could ship today often waits for the next client release, and engine updates get slowed down by content changes hitching a ride on the same build.

## What a shared pipeline costs you

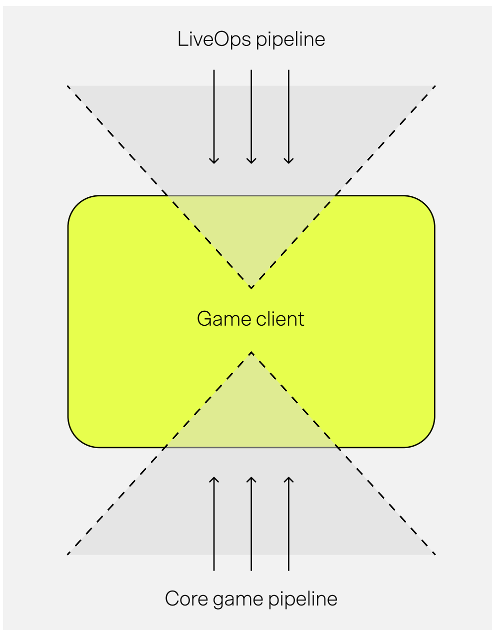
The degree of friction varies by studio, but the patterns are familiar. Seasonal content gets locked in weeks early because it needs to make a submission build. A/B testing a shop layout becomes an engineering project instead of a configuration change. UI fixes, even a broken support URL, can end up waiting for the next client patch. And engine-side UI specialists, who are expensive and hard to hire, spend a chunk of their time on layout and localization work that doesn't require engine expertise.

These effects compound. Over time, they shape what your LiveOps team is willing to attempt.

## The shift in player expectations

Player expectations around the meta-game have been set by titles that update their shops, events, and seasonal content continuously. When your competitor's battle pass refreshes weekly and your seasonal event has shown the same layout since launch, players notice, even if they can't articulate why one game feels more alive than the other. The rest of this paper looks at how web technologies can help close that gap.

# The hybrid UI model




## The core idea

Keep core gameplay native. Move your game's volatile, frequently updated layers (shops, events, seasonal content, social features, analytics dashboards, compliance text) into a web-based layer deployed independently from the game client.

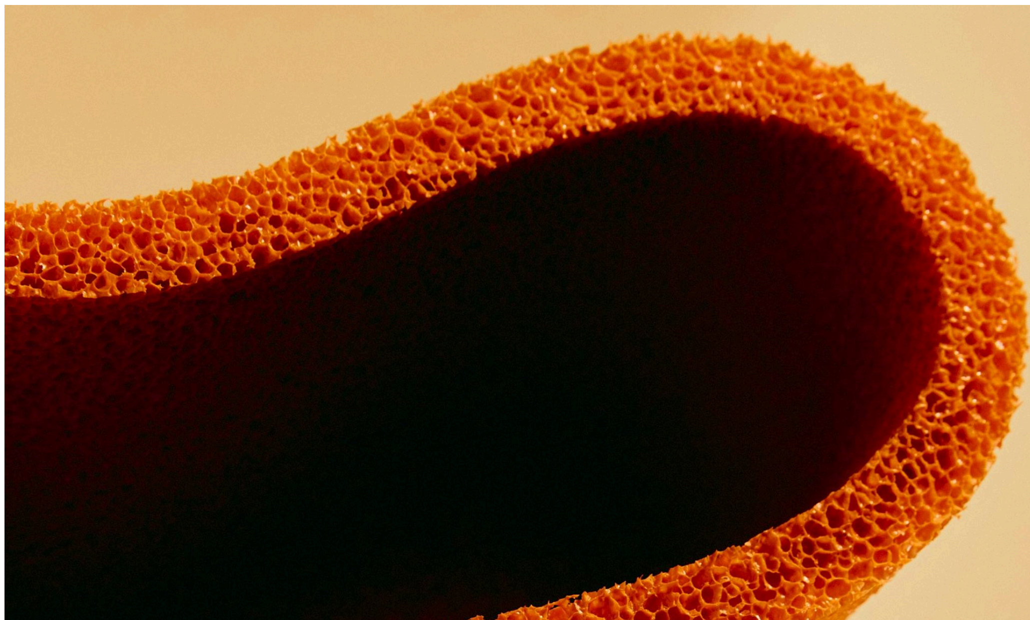
This is a hybrid UI architecture: two pipelines, one native interface between the layers, and a player who never notices the seam.

This paper focuses on using platform-provided default webviews rather than purpose-built rendering engines like Coherent. Engines like Coherent let you build full game GUI with web technologies, meeting player expectations on fluency and responsiveness. Default webviews are available at no extra licensing cost and work well for the meta-game layers discussed here.

## The line of feasibility in 2026

For years, the industry consensus was straightforward: use webviews for the EULA and the news feed, use the game engine for everything else. That was a reasonable safety measure. Webviews were slow, memory-intensive, and lacked the responsiveness players expected from anything they actually interacted with.

The technical landscape has shifted. Hardware-accelerated rendering is ubiquitous, and high-performance JS engines have matured. Here's what web technologies can now handle inside a game:



**Localized narrative content** — interactive visual novels or season recaps that you can update with new story beats without a client patch

**Dynamic storefronts** — personalized, A/B-testable shops that change by the hour

**Data-heavy dashboards** — real-time telemetry, player statistics, live-market auction houses pulling from complex APIs

**Animated meta-games** — complex crafting systems, interactive battle pass maps with 60fps transitions and particle effects via Canvas or WebGL

What stays native: core gameplay rendering, physics, real-time multiplayer, anything where frame-level precision matters.

This isn't experimental. League of Legends has run its entire client UI (lobby, shop, champion select, crafting) on Chromium Embedded Framework (CEF) for years, and Guild Wars 2 uses Chromium for its Trading Post and in-game content. Separately, engines like Coherent Gameface power the full UI of titles including PUBG, World of Tanks, and Alan Wake 2, but that's a different integration path from default webviews.

## What this unlocks

Three things become possible:

**Ship content without client builds.** Web content updates without separate asset downloads, remote catalogs, addressables, or client patches. It's a website: you can update it at a minute's notice. Your content team reacts to player sentiment, designs new events with custom UI functionality outside the client update cycle, A/B tests at shorter cadence, adds lore as it's finished, and pulls content that isn't working based on live analytics.

**Hire from a broader talent pool.** High-level Unity or Unreal UI specialists are rare and often buried in engine-specific technical debt. A web-based pipeline opens the door to senior frontend engineers. Modern tools like React, Svelte, or Three.js let you build complex, responsive interfaces faster than native engine UI, and the AI tooling trained on web code is significantly more mature.

**One implementation across platforms.** With caveats covered in the platform-specific section, you can now create deployable game UI-level visuals on all platforms with a single technology stack.

| Feature | Engine layer (native) | LiveOps layer (web) |
|---|---|---|
| Update frequency | Every two to three months (stable) | Daily/hourly (fluid) |
| Deployment | Platform store (App Store, Steam) | Server push (CDN) |
| Talent pool | C++, C#, graphics engineers | React, Vue, TS, UI/UX designers |
| Risk profile | High (requires full QA/cert) | Low (scoped to the web container) |

# Architecture deep-dive: building the invisible interface

The technical challenge is creating a window that is high-performance, secure, and context-aware. This requires more than embedding a URL. It requires a robust communication layer.

## The glass floor principle

The player should never feel like they've left the game engine. Everything in this section serves that goal.

## The two-way data interface

To keep the source of truth secure while maintaining a responsive UI, the architecture needs a native-to-JS interface.

**Engine → Web (push):** The game client pushes real-time data to the web view. Player inventory, currency, and progression state are all available to the JavaScript layer automatically. The web UI reflects the game state without constantly polling an API.

**Web → Engine (callback):** When a player interacts with a web-rendered button, the JS sends a message back to the C++/C# layer. The engine validates the action, executes the logic, and tells the web view to update the visual state. The engine is the source of truth. The web layer is presentation only.

## Reaktor it.

Ask us about designing and building hybrid UI architectures, from client-WebView integration to the team model that keeps your engine and LiveOps pipelines moving independently.

## Eliminating visual jank

Nothing breaks immersion faster than a loading spinner or a white flash while a page initializes. To eliminate these, use either pre-warming (render in memory before the player needs it) or fast-loading static pages with aggressive caching, and keep the view hidden until minimum content has rendered so the player never sees the layout process.

**Pre-warming.** Initialize the WebView and render the page off-screen during the game's boot sequence or a loading screen. When the player clicks "Shop," the view is already rendered and ready to flip into the foreground. The tradeoff is that the webview consumes resources even when not visible. This works best for menus, where more resources are available than during gameplay.

**Managed loading.** Use lightweight content and don't show the view before the page has fully loaded. Handle the wait gracefully with an in-game animation or an overlay if loading takes too long.

**Asset passthrough.** Rather than having the WebView re-download the same textures or fonts the game already has in memory, pass local file references or use specialized virtual protocols. This reduces bandwidth and ensures visual parity between native and web elements, but increases coupling between the web content and the game implementation.

## Input handling and spatial navigation

In a native game, input is precise. In a web view, it can feel floaty.

**Input masking.** The native engine handles raw input first, then passes specific events (button clicks, D-pad navigation) to the web layer. This makes feedback feel nearly indistinguishable from native game UI. Mouse and touch inputs often feel clean and snappy without masking.

**Spatial navigation.** For console and controller support, don't rely on standard HTML focus. Implement a spatial navigation system that tells the web view exactly which element to highlight when the player moves the joystick. Standard HTML autofocus doesn't understand gameplay logic. You need explicit focus management that maps controller inputs to logical UI jumps.

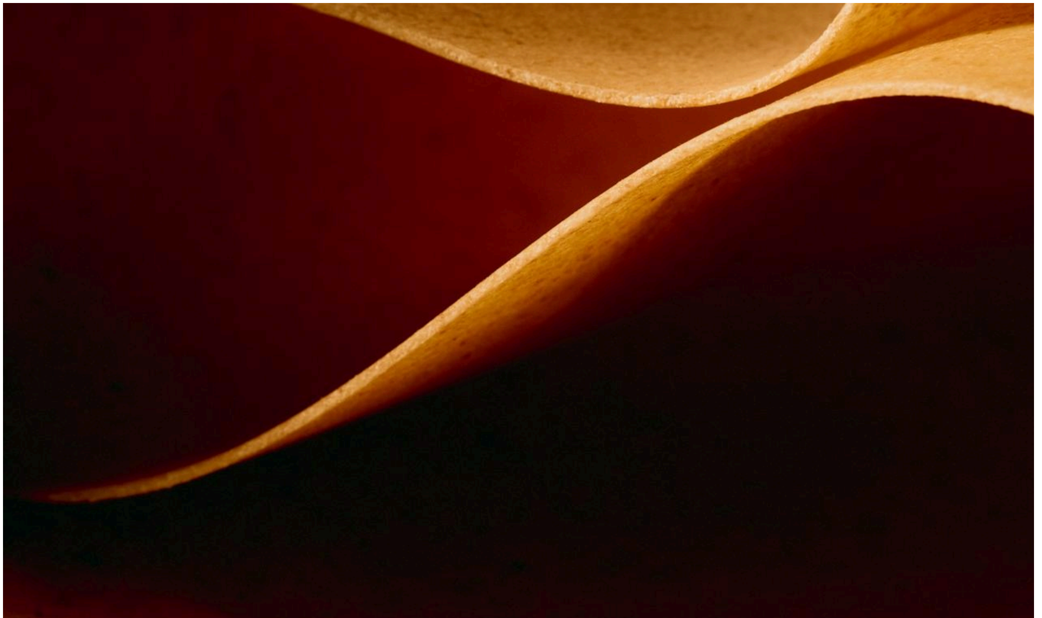
## Managing dependencies between teams

How tightly to couple the engine and web teams depends on the use case.

If the web content is deeply involved in game logic and the teams work together (or are the same team), more dependencies are natural. Features that update with every client release fit this model.

If the teams are separate and the content is less involved with game mechanics, a lightweight integration layer makes more sense. Keep native interface changes minimal so web content development can move independently from client development.

The dependency model should be a deliberate design decision, not something that emerges by accident.



# Platform-specific realities

Reaktor it.

Ask us about shipping WebView content on console, mobile, and PC, including certification compliance, memory budgeting, and cross-platform performance tuning.

## Console constraints: ephemeral storage and memory limits

Developers often rely on localStorage or IndexedDB for web state. On PlayStation and Xbox, these environments are often ephemeral. If the user closes the game or the OS reclaims memory, that storage gets wiped.

The solution: use the game client as the source of truth. Don't store critical data in the WebView. Use the bridge to pass data back to the native layer, which handles save-game or cloud sync. The web layer remains stateless, acting only as the presentation window.

## Console certification and security

Consoles have strict Technical Requirement Checklists (TRCs). If a WebView allows a user to navigate to an external URL or type in a website, the game can be rejected during certification. Build a hardened sandbox: implement a strict domain whitelist, disable all external navigation via the bridge, and ensure the WebView only communicates with trusted servers. This satisfies platform holders and prevents XSS vulnerabilities.

## Mobile considerations

WebView behavior differs between iOS and Android. Lower-end Android hardware and Switch require special attention to maintain the smoothness players expect. Performance profiling and memory budgeting are non-negotiable on these platforms.

## Memory budgeting

Every megabyte of VRAM matters on consoles, and a Chromium instance can easily consume 1 GB if left unchecked. Target a strictly managed ~200 MB memory footprint for the web layer. Optimize web assets: prefer SVG over PNG, use minimal JS frameworks, and cache aggressively.

## Handling offline and degraded states

Some games need to consider what happens when the player is offline or your LiveOps server is down.

Build a local-first cache. If the WebView can't reach the latest version, fall back to a locally stored static version that informs the player they're offline and provides whatever features work without a connection. Never show a 404 or a broken page.

# Use cases

## Dynamic storefronts and personalized offers

In-game economies are living systems. They need seasonal themes, flash sales, and personalized offers based on player behavior.

Designing a Lunar New Year shop skin in HTML/CSS takes hours, not days. You can A/B test two different layouts, one video-heavy, one stat-focused, and measure conversion in real time. The shop feels fresh every time the player logs in, without a single client update.

## Interactive patch notes

Standard patch notes are wall-of-text that players skip. Move them to a web view, and they become a media experience: embedded video clips showing before-and-after ability changes, interactive sliders for weapon stat adjustments, and 3D model viewers via Three.js for newly released skins.

Higher engagement, better community sentiment, and players who actually read the patch notes.

## Player-facing analytics and season recaps

Players love data about themselves. "Spotify Wrapped for games," season recaps, stat dashboards, and shareable year-in-review content have become a retention tool.

Web is the natural medium for this: heatmaps, charts, interactive data visualizations using libraries like D3.js or Recharts. These are significantly faster to build with web tools than in-engine. The output is shareable content that calls lapsed players back.

## Social and community features

Building guild management, live chat with rich-text support, or a message board system in a native engine is painful.

Web frameworks already solve these problems. Integrate video players, live streams, guild boards, and friend lists with party-invite hooks. The result is a community experience that feels like a modern social platform, not a legacy menu system.

## Compliance and legal content

Privacy policies, Terms of Service, and regional legal disclosures change constantly. This is the most established webview use case, and for good reason.

Serve legal content via a web layer. When your legal team updates a clause, every player sees it instantly. No client patch or cert cycle needed.

# Addressing the objections

## "WebViews look cheap and feel slow"

The stigma comes from 2015-era wrapper apps. Modern hybrid architecture with proper bridge design, pre-warming, and asset optimization delivers native-feeling performance. The distinction matters: the core gameplay remains native, the meta-game is web-driven. ArenaNet documented measurable performance improvements after moving Guild Wars 2's Trading Post to CEF.

## "Our engine team can handle all UI"

They can. But should they? The question isn't capability, it's resource allocation. If your senior C++/C# engineers are spending more than 10% of their time on UI positioning, font-tweaking, or localization layout, that's expensive time spent on problems web technologies solved a decade ago.

## "This adds complexity to our stack"

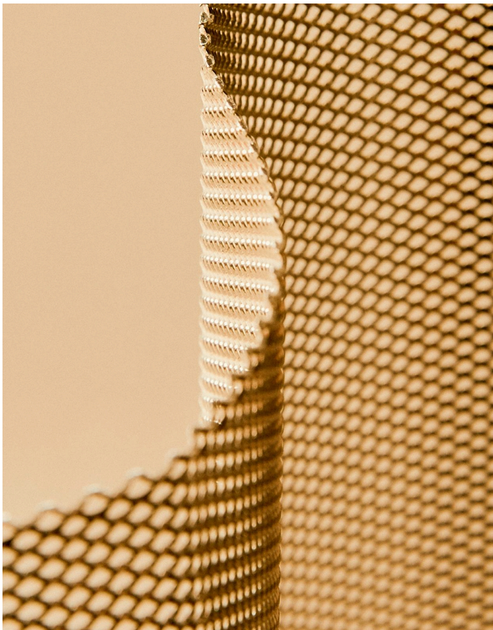
It adds a layer, but it removes dependencies. The net effect, when architected properly, is less cross-team blocking, faster iteration, and lower deployment risk for content changes. The bridge is an investment in decoupling, not additional complexity for its own sake.

## Reaktor it.

Ask us about evaluating whether a hybrid UI model fits your game, your team structure, and your platform requirements. We'll help you find the right starting point.

## "Didn't Riot move away from CEF?"

League of Legends used WebView technology extensively. Riot ran into issues integrating the out-of-game web client with the in-game engine. In their case, the game engine and WebView content became so tightly coupled that they were interfering with each other's performance. They also have the resources and capability to create custom content pipelines and custom UI systems that match the flexibility WebViews provide. If you have those resources and can't keep web content separate from in-game content, that might be the right path for your studio too. For most teams, WebViews remain the more practical choice for dynamic content.



# The agility gap self-assessment

## The diagnostic questions

| Question | Yes | No |
|---|---|---|
| 1. Are minor UI fixes or promotional banners waiting three to five days for App Store or console certification? | ☐ | ☐ |
| 2. Are your senior C++/C# engineers spending more than 10% of their time on UI positioning, font-tweaking, or localization layouts? | ☐ | ☐ |
| 3. Do you avoid running mid-week events because the cost of a client update outweighs the potential revenue? Are seasonal assets frozen weeks before an event because they must be bundled into the binary? | ☐ | ☐ |
| 4. Is your UI team constantly blocked by the engine team's release schedule? | ☐ | ☐ |
| 5. Is your in-game shop static, with the same offers to every player regardless of playstyle or region? | ☐ | ☐ |
| 6. Is it impossible to show different shop UI or customized offers to different player segments without hardcoding logic into the binary? | ☐ | ☐ |
| 7. Does a text typo or a broken support URL require a multi-gigabyte client patch? Do localization teams have to wait for a full build cycle to see their translations in context? | ☐ | ☐ |
| 8. Is running an A/B test on a UI layout a major engineering task rather than a marketing toggle? | ☐ | ☐ |
| 9. Does integrating a dynamic community feature, a live stream embed, or a social feed feel like a months-long engineering project? | ☐ | ☐ |

## Interpreting your results

If you answered "yes" to two or more, your studio likely has pipeline friction. You're spending expensive engineering time on tasks that a modern web stack handles with less effort and lower risk.

Patterns to watch for:

- **Mostly 1, 3, 7:** content velocity problem, your release cycle is throttling your content team
- **Mostly 2, 4:** talent bottleneck, specialized engine engineers are doing generalist UI work
- **Mostly 5, 6, 8:** monetization rigidity, you can't personalize or test your revenue-generating surfaces
- **Mostly 9:** integration overhead, features that should be straightforward become major projects

## Where to start

Moving to a hybrid model doesn't happen overnight, and it shouldn't be a rip-and-replace operation. The most successful implementations start small, with interactive patch notes or a dynamic shop layer, to prove the bridge architecture before expanding to complex social or guild systems.

The goal state: your engine team is focused on immersion, your LiveOps team is focused on engagement, and neither is waiting on the other.

# About Reaktor

Growing a good game into a successful one takes more than exceptional game design. Players expect a seamless experience from start to finish – whether it's logging in, saving progress, making purchases, or chatting with friends. Every interaction matters.

Reaktor helps gaming companies level up with scalable, cutting-edge technology, player-centered UI/UX design, and strong community and brand-building expertise. Our senior experts have worked with gaming companies for years and know what it takes to impress your players with exceptionally well-executed solutions.

Teaming up with Reaktor allows both the game studio and the gaming community to focus on the one thing that matters the most – the game itself.

Reaktor. Go beyond the game.

Discover more: reaktor.com/gaming