GS STUDIO™

TECHNICAL POST-MORTEM · APRIL 2026

Five Years on ATONE

A Technical Post-Mortem of a Paused MMORPG

For almost six years, a significant portion of our studio's engineering effort was going into a project called ATONE — a next-generation action-adventure MMORPG that fused the social systems of classic MMOs with Souls-like combat. It was ambitious in a way that, in hindsight, was probably irresponsible. We shipped it internally through multiple vertical slices, built production infrastructure to support it, and made technical bets I still think were right. **It was cancelled before release.**

STUDIO	ENGINE	STATUS
GS Studio™	**Unreal Engine 5**	**On hold**

§ 00 · POST-MORTEM

Introduction

This is the honest version of what happened. Not a case study. Not a marketing piece. If you're building something in this space — publisher, studio lead, tech director weighing your stack — there's value in hearing the uncut version of what works and what breaks when you try to run Souls-like combat on top of a persistent, scalable MMO architecture in UE5.

I'll be concrete about what we built, specific about the technical reasoning, and honest about where we got it wrong.

§ 01 · POST-MORTEM

The Project

ATONE was designed as a genre fusion. The pitch: take the social density and persistent world of a classic MMO — guilds, shared zones, long-horizon progression, raids and world bosses — and layer on the moment-to-moment combat feel of a Souls game. Deliberate animations, committed attacks, stamina management, iframes on dash, precise hitboxes. Not the auto-targeted, ability-rotation combat of most MMORPGs, but something that could legitimately stand next to Dark Souls or Where Winds Meet on combat feel.

The team was **31 engineers, 12 designers, 49 artists** at peak working out of our office in Cyprus and remotely all around the world. The tech stack settled on **UE5**, with a custom dedicated-server backend designed to support **70 concurrent players per layer**.

The project was put on hold 3 months ago when our client's funding picture shifted and they elected to pause and re-evaluate how to deliver the game more capital-efficiently. **The engineering was on track. The vertical slices were landing. The tech bets were proving out.** What changed was external to the production — a budget decision on the client's side, not a failure in the work. Large MMO builds are expensive, and the cost structure of a solo, long-horizon internal build is a hard problem in the current market. I'll come back to that, because it's one of the more important lessons in this whole story and has nothing to do with Unreal.

Writing this now, 3 months after we wound the project down, the team has dispersed to other work inside the studio. The project is now the history. The lessons are not.

> The project is now the history. The lessons are not.

§ 02 · POST-MORTEM

The Ambition: Why This Fusion Is Hard

Before the technical detail, it's worth being precise about what we were asking the tech to do. The difficulty of ATONE is invisible unless you've tried it.

Classic MMO architecture optimizes for many players sharing a world. State is authoritative on the server. Client-side prediction is limited because in a spell-and-cooldown combat model, small latency costs are hidden by animation windup and cast times. You can tolerate a **150ms** round trip and the player won't feel it. Furthermore, classic games all feature a targeting system which simplifies their architecture and technology.

As for Souls-like games, they require non-targeting and collision calculations precisely for this reason. The fun is in the frame-accuracy of a dash, the commit window of light and heavy attacks combo system. That entire loop falls apart close to roughly **100ms** of perceived input lag. You cannot hide it with animation — the animation is the thing.

Putting these two together means you're asking your networking stack to deliver MMO-grade persistence and player density — many people in a shared zone, full state replicated — with Third-Person Action responsiveness. The standard solutions for each pull in opposite directions.

That framing shaped every technical bet we made. Most of them were bets on this being the right fusion to make, and bets on UE5 being far enough along to support it.

§ 03 · POST-MORTEM

Key Technical Bets

Unreal Engine 5

The first decision was engine. Custom engine was never seriously on the table — we're not id Software and this was not a tech demo. We initially used Unreal Engine 4 but switched to Unreal Engine 5 as soon as it became stable. During the transition we considered the challenges and incorporated solutions into the project architecture. This enabled us to upgrade the engine easily and regularly without any pain.

The reasoning: UE5 was where multiplayer-at-scale was headed. IRIS, Mover, Nanite, Lumen, World Partition — every one of those features was either replacing something we'd fight with in UE4 or unlocking something we couldn't do at all. Building a five-year MMO on UE4 meant shipping against an engine that would be a generation behind by launch. Building on UE5 meant shipping into whatever UE5 became.

However the goal was to launch the game using the most current, yet stable, version of the engine. The industry prioritizes cutting-edge technology over outdated or legacy solutions at the release date. While this ambition sounds positive, dealing with “taming the dragon” — new, experimental technologies — is inherently complex and unstable. Nevertheless, we were investing our expertise and experience in utilizing these new tools because players, and consequently marketing, will value the innovation. Crucially, we must have learnt from and avoided the mistakes seen in previously released titles.

The downside — which we underestimated — is that **“experimental” in an Epic release note is load-bearing.** IRIS, Mover, and several other features we depended on were marked experimental through much of our development window. That has real costs, and I’ll come back to them.

IRIS for Replication

IRIS is UE5’s new replication system, and for a 70-player layer it was the right bet. The old Replication Graph model works, but tuning relevance and priority for high-density combat with per-frame precision is painful. IRIS gives you a filter-and-prioritization model that’s actually designed for the problem — dirty-state tracking, per-connection filters, priority-based delta updates.

The bandwidth math for 70 players in Souls-like combat isn’t trivial. Every character broadcasts position, orientation, animation state, stamina, status effects, buff stacks, and combat flags at high tick rates **60Hz**. Without aggressive filtering by spatial relevance, interest management, and state-delta compression, you saturate a consumer uplink quickly. IRIS lets you express those filters declaratively and keeps the hot path in native code.

Nanite for the World

Nanite was a pure win. For an MMO with a large traversable world and a small environment-art team, virtualized geometry collapses the LOD problem and lets artists author high-density environments without baking normals into every tree. We leaned on it heavily for environment art, and the pipeline savings were real.

Mover for Character Movement

Mover is UE5’s attempted replacement for the venerable CharacterMovementComponent (CMC). CMC is a roughly fifteen-year-old, ten-thousand-line monolith that handles movement for the majority of UE games, and it is both an engineering marvel and an engineering tragedy. Anyone who has shipped a game with tight movement feel has spent nights in it.

For a Souls-like, CMC’s opinions about acceleration, root motion, and prediction are a fight. Mover is modular, data-driven, and more amenable to the kind of movement we needed: root-motion-heavy, animation-driven, precise stop-on-frame behavior, tight integration with combat state.

We bet on Mover early. **That bet paid off in the vertical slice and hurt us when Mover’s API changed between UE5 minor versions.** More on this later.

GAS for Abilities, Attributes, and Effects

Every UE multiplayer team eventually has the GAS conversation. Epic's Gameplay Ability System is the de facto framework for abilities, attributes, and gameplay effects in serious UE projects, and the decision is rarely whether to acknowledge it — it's how deep to go.

We went all in, and used GAS largely as it ships. That wasn't a given five years ago. Earlier versions of GAS had enough rough edges — prediction corner cases, replication cost under load, general opacity — that serious teams often ended up either heavily extending it or building parallel systems. By UE5, the framework had matured enough that stock GAS covered the combat, stats, and effects surface we needed without meaningful re-architecture. Abilities, attributes with modifiers, gameplay effects, tags, and the standard prediction model all held up in production.

Where we extended it was prediction, but only at the edges. GAS's built-in prediction handles the base cases — movement, interactions, simple ability activations — correctly. For combat mechanics with Souls-like timing sensitivity, we layered our own predictors inside the GAS ecosystem rather than replacing the framework's prediction model. The distinction matters: **we weren't fighting GAS, we were extending it where our specific combat timing required more control than the default path provided.** That's a very different engineering posture than forking GAS or routing around it, and it's the posture I'd recommend to any team making the same call today.

For attributes and stats, we used AttributeSets and GameplayEffects as designed. Character stats, stamina and health, weapon modifiers, status effect stacking rules — all of it ran through the standard framework. Running a parallel stats system alongside GAS is a trap several teams have fallen into; the replication and modification logic GAS gives you for free is worth more than the flexibility of rolling your own.

The one place GAS cost showed up meaningfully was AI. For players, the per-character GAS tick cost was absorbed comfortably by the server budget — 70 concurrent players with full ability and effect state wasn't a problem. For AI, it was. Regular mobs carrying full GAS state, each running their own ability graphs and effect ticks on every server update, didn't scale.

The optimization we built was to group regular mobs into a single global tick, processing their individual sub-ticks within that unified server update. Because mob behavior isn't highly unique or complex — most of them share a small set of ability patterns, attribute modifiers, and effect timings — we could generalize their tick path and amortize the cost across the group instead of paying it per instance. Boss AI and anything with genuinely unique behavior stayed on the per-character tick model; the optimization applied to the long tail of ordinary mobs that make up the bulk of any populated zone.

The practical effect was significant — the bottleneck shifted off GAS tick cost entirely — at the new density, navmesh queries and perception updates became the next limiting factor rather than per-mob ability and effect evaluation. More importantly, it meant we didn't have to fight the GAS architecture to get AI density — we kept everything inside the framework and restructured the tick topology around it.

The broader lesson on GAS: by UE5 it's production-grade out of the box for projects of serious scope, and **most teams' instinct to "just write it ourselves" is wrong.** Extend at the edges

where your genre demands it. Plan for AI-tick cost specifically at density. Default to using GAS's primitives rather than paralleling them.

Custom Backend Over Middleware

We considered the usual middleware: SpatialOS (what's left of it), Photon, Pragma, AccelByte. We ended up building a custom dedicated-server backend.

The argument against middleware, for this specific project: every service we evaluated forced tradeoffs on the combat loop. Interest-management models that assumed lower tick rates. Authority models that didn't cleanly support the hybrid we wanted — server-authoritative for high-stakes state, client-authoritative with rollback for cosmetic state. Persistence models that didn't map cleanly to a world-shard MMO topology.

The argument for building custom was control over the latency budget end-to-end, the ability to co-design server logic with the UE5 client, and avoiding per-CCU licensing at scale.

Was this the right call? For this specific combat requirement, yes. **It's also the single largest ongoing cost line on a project like this, and one of the things that any team attempting it should cost out with open eyes.** I'll come back to the economics of custom infrastructure in the lessons section.

§ 04 · POST-MORTEM

What We Built That Was Novel

Setting aside the obvious — a full UE5 MMORPG client and a custom backend service, neither of which is trivial — four pieces of work are worth describing in detail because they're portable and we're proud of them.

The 70-Player Dedicated-Layer

The layer-server design looks simple on the surface: authoritative state, IRIS replication out to clients, persistence to a shared database layer between layers. The hard parts are in the details.

Tick budget is the binding constraint. At a 60Hz server tick with 70 characters carrying active combat state, AI around them, spatial queries for perception and aggro, hit registration against precise capsules, and replication work, there's very little per-tick headroom. We spent significant effort on a spatial index octree for interest management and broad-phase collision, and on moving AI perception to an async update model where not every tick updates every AI.

Server-side validation is implemented using Rollback. The process involves the client sending its input or movement delta to the server. The server then validates this input, processes the movement, and transmits the resulting position back to the client. Meanwhile, the client has been moving based on a prediction and subsequently adjusts its own position to match the authoritative position received from the server. Combat hits are server-resolved, with client-side immediate feedback on local attacks reconciled on server confirmation. This is not novel in the

abstract — it's what every competitive multiplayer game does — but doing it cleanly with Souls-like animation-driven combat, where the "correct" hit frame is tightly coupled to the animation, takes work. We built a small animation-synchronized hit-registration layer that mapped hitboxes to anim notify windows with latency-aware reconciliation.

Blendshape-Based Hand Animation

Most character animation is bone-driven. Fingers have bones. Gestures are bone poses. For a game with heavy weapon interaction, grip precision, and fine-grained object handling, we moved hand animation onto a blendshape-based system.

The case for blendshapes over bones on hands: bone-rigged fingers are notoriously expensive to author per-weapon, per-animation. Every new weapon type needs custom finger curls. Blendshapes let you author a small set of canonical grips — hammer grip, pinch, point, open palm — and blend between them procedurally, driven by the weapon's grip metadata. The same animation works across weapon types with a parameter change.

The tradeoff is memory and deformation quality. Blendshapes are per-vertex deltas and cost GPU memory per character. For a crowded 70-player layer with unique characters, that memory bill adds up, and we spent effort on blendshape compression and LOD'd blendshape tiers — fewer active blendshapes at distance.

The artistic payoff is real: hand interactions feel authored in a way most MMOs don't bother with, and the authoring cost at scale is genuinely lower than bone-per-weapon once the canonical grip set is defined. **It's a technique I'd recommend to any team doing heavy weapon or object interaction, with the caveat that the memory budget has to be designed for it from day one.**

GPU Skinning

Standard UE GPU skinning exists — most skeletal meshes skin on GPU already. What we built was a more aggressive version that moved additional animation work from the CPU animation graph to GPU compute shaders.

The specific bottleneck: in a crowded zone, the CPU cost of running the animation graph for every visible character — pose evaluation, blend trees, IK, state-machine updates — dominated the frame on our target platforms (PS and consoles). GPU skinning alone doesn't fix this; the CPU still does the graph evaluation and hands a pose to the GPU.

What we did was move chunks of the pose-evaluation pipeline — specifically blend math on pre-baked poses, and a class of IK — into compute shaders, and batch the skinning dispatch per skeletal-mesh type. The practical effect: 70 visible characters doing similar things (walking, running, idling, common combat animations) got amortized GPU work instead of 70 separate CPU graph evaluations.

The limit of the technique is that you can't move arbitrary anim-graph logic to GPU. Procedural control, event-driven state machines, gameplay-driven blends don't belong in compute shaders. So this was a targeted optimization for the common case, not a wholesale redesign of Unreal's animation system. Our 70-character scene was previously eating **8–10ms** and finally fit in **~2.5ms**, roughly **70–75% CPU anim-tick reduction**. That represented around **80% of**

visible crowd.

POI Sharing

Most multiplayer games ping. You press a button, spit a temporary marker into the world at a location, your teammates see it, it fades out. That's not what we wanted.

An MMO's social contract around shared world information is richer. Players leave notes at dungeon entrances. Guilds mark farming routes. Raid leaders annotate boss arenas. A temporary ping system doesn't carry that weight.

The POI system we built is persistent, permissioned, and semantic. A POI is a typed object with a location, author, scope (private / party / guild / world), TTL, and payload — text, icon, or structured data like "quest hand-in here" or "resource node, respawns 14:00 UTC." POIs render client-side with a visibility model that respects scope. They persist to the backend and survive logout. They can be subscribed to — your guild's POIs are pushed to you on login, not re-discovered.

The technically interesting part was designing the POI channel so it didn't become another replication firehose. POIs replicate through a separate channel from combat state, with different staleness tolerance, and we built a subscription model — client subscribes to POI channels relevant to it — rather than broadcasting all POIs to all clients in a zone. POIs ride a separate reliable, ordered, event-driven channel — TCP-backed — because a POI never needs to arrive in 16ms, but it absolutely must arrive and must not arrive out of order.

Rather than one flat "POI channel," we ended up with channels keyed by scope: one per guild the player belongs to, one per active party, one per zone, plus a private channel for self. On login/zone transition/guild-join, the client subscribes to the relevant set. On unsubscribe (logout, leaving zone, leaving guild), the server tears down the fanout. **This is the crucial design decision — scope is the subscription axis, not proximity — because scope is what governs visibility in the first place, and it maps cleanly onto pub/sub primitives.**

This is one of those systems that seems minor until you play with it, at which point the standard 3D ping feels thin by comparison.

§ 05 · POST-MORTEM

What We Learned About MMO Production at This Scale

This is the section that matters most for anyone reading with a live project.

> **Experimental engine features are a schedule risk, not a feature risk.**

Betting on IRIS, Mover, and other UE5 features in flight meant every Epic release dropped API changes on us. **A minor version bump that should have been a day's integration was sometimes a two-week reorganization.** On a five-year project this compounds. Starting today, we'd wait one more UE5 minor on the load-bearing experimental features, or we'd fork and freeze them.

> **Multiple novel systems in one project compound schedule risk.**

We had four genuinely novel systems in production — the custom backend, blendshape hands, GPU-assisted animation, the POI system — plus the MMORPG itself. Each paid off technically, and we'd build them all again on a project designed to justify them. On a cost-constrained build, the calculus is different: every novel system is additional schedule variance, and schedule variance is what turns budgeted projects into over-budgeted ones. **Knowing which systems are load-bearing for the pitch and which are "nice to have" is a capital-planning exercise as much as a technical one.**

> **Dedicated-server economics on a long-horizon MMO are the hardest part of the whole enterprise.**

This is the lesson I'd most want the industry to internalize. The custom backend gave us the combat latency we needed. It also meant that the cost of running the project scaled with concurrent players in a way middleware does not. That math is tolerable at scale and painful below it, which creates a hard tension: a new IP has to budget for a scale of success it hasn't achieved yet. The reason MMO funding gets re-evaluated mid-flight — and the reason we're writing this post — is that the economics of building and operating a persistent multiplayer world at this density are genuinely hard, independent of how good the team or the tech is. **Anyone sizing an MMO build should start with this problem, not end with it.**

> **Content production and engineering are different disciplines with different ramp curves.**

Building engine capability to support a world scales differently than building the content to fill it. We had the systems to support more content than we were provisioned to author at a sustained cadence. This isn't a UE5 problem or a backend problem — it's a team-shape and capital-allocation problem, and it's one of the places where external co-development can meaningfully reshape the cost structure, since content production can be scaled up and down more elastically than core engineering.

§ 06 · POST-MORTEM

Why This Matters for Studios Building Multiplayer Games Today

A few things I'd offer to anyone currently standing where we stood in year one, or anyone reviewing a pitch that looks like where we stood in year one.

If you're building a multiplayer game in UE5 that needs more than roughly **32 concurrent players** in a layer with real combat feel: you are on a short list of teams in the world solving this problem. The standard UE5 multiplayer stack gets you to small-squad shooters and co-op ARPGs with minimal friction. Past that, you're doing original engineering. IRIS is the right bet. Mover is the right bet if you have the engineering capacity to track its API. Custom backend is a real question whose answer depends on your latency budget and your CCU projection, not your preference.

If you're a publisher or studio lead evaluating a pitch that promises MMO density and action-combat fidelity: ask how they plan to pay the tick budget at their target player count, ask which UE5 features they depend on and what version they're tracking, and ask how many novel systems are on their critical path. **Two is a lot. Four is meaningful only if the team has shipped them before.**

If you're a publisher whose internal MMO build is running expensive — and most of them are — the answer is rarely "cut scope." Scope cuts on a partially-built MMO destroy more value than they save. The more productive conversation is usually about team structure. **A mature co-development partner that has already absorbed the cost of specializing in UE5-multiplayer-at-density is structurally cheaper to engage than standing up or maintaining an internal team with the same specialization, because you're not paying for the ramp.** This isn't a theoretical point — it's increasingly how large multiplayer projects in the West are actually getting built.

If you're a team considering novel animation systems like blendshape hands or GPU-side pose work: these are real techniques, not research. They ship. They have memory and authoring costs that compound with roster size. Budget for them at the content-pipeline level, not just engineering.

The engineers who built ATONE are still at GS Studio™, and the technical capability we assembled — UE5 multiplayer at density, custom dedicated-server backend, production-grade IRIS and Mover experience, novel animation tooling — is now among the deeper concentrations of this specific skill set worldwide. That's partly why this post exists. If you're a studio or publisher building something in this neighborhood, we're an obvious conversation.

> Most MMOs die before release. Very few teams write about it in technical detail.

But even if you never talk to us, the honest version of this story is more useful to the industry than another post-launch sizzle reel. Most MMOs die before release. Very few teams write about it in technical detail. If this saves one studio from re-learning what we learned about

experimental UE5 features, or the compounding cost of stacking novel systems, or the specific difficulty of putting Souls combat on top of persistent zones, **it has paid for itself.**

ATONE is on hold.

The engineering shipped.

If you're working on something in this space and want to talk to the people who built this: you can DM us here or drop a message to ***hello@gs-studio.eu***