CASE STUDY · PROOF OF PLAY

Architecting the combat engine of a live auto-battler.

How I designed the systems that make a deterministic RPG feel fast, reactive, and worth replaying, and built the tooling to balance it with data instead of guesswork.

Role: Special Projects Lead | 559 PRs / 522 merged | ~7 months

Live: shibastorygo.com ↗

The challenge

The mandate was speed. We were building a live game the **AI-forward** way, prototyping and shipping new mechanics faster than a traditional studio works. The only way to move at that pace without the game collapsing into one-off spaghetti was to make the systems **composable**. So the real problem was never any single feature. It was building an engine and a content vocabulary that let us invent fun, novel mechanics *rapidly*, author them as data, and keep shipping them at that speed.

On top of that, the combat is **deterministic** by design: client and server run bit-identical, seeded simulations with no hidden dice, validated by per-command hashing. Every battle has to *read* as snappy, reactive, and strategically deep while actually being a replayable simulation that two machines can verify down to the last hit. Fast to author, fun to play, and provably consistent, all at once.

I owned that combat engine and designed most of the systems players experience inside it: the turn order, the buffs and debuffs, the way data reaches into the middle of a fight to bend the rules, the elemental damage model, the roguelike build meta, the raids, the mounts and gear, and the simulation suite the design team uses to balance all of it.

My role

I joined Proof of Play as **Special Projects Lead** and became the in-house owner of the combat engine and a large part of the game's systems design. I work across the whole stack: I architect the engine primitives, design the content and mechanics on top of them, and ship. In roughly seven months I authored **559 pull requests (522 merged), peaking at 168 merges in a single month**, spanning deep engine work, live-ops content, and the internal tooling below.

Product sense, amplified by AI

I approach systems as a product and UX designer, not only an engineer. My background is behavioral design and consumer products people come back to, the same behavioral-incentive thinking behind user-facing work like Sworkit, and I brought that instinct to a deterministic combat engine. The systems are deep, but the question was always how a fight *feels* and why a player runs it again.

AI is the multiplier on that judgment. Strategic enablement with AI, building the harnesses, directing a set of agents, and automating the unglamorous work, is what let one person prototype, design, and ship engine systems at a pace that normally takes a team, and keep the game fun while moving that fast. Product sense decides what is worth building; AI leverage is how it ships rapidly.

That combination is the capability I actually bring: a product designer's eye for what makes something fun, an engineer's hand for building the primitives, and AI leverage to do both at speed. It is what made me one of the most productive designers on the project.

What I designed and built

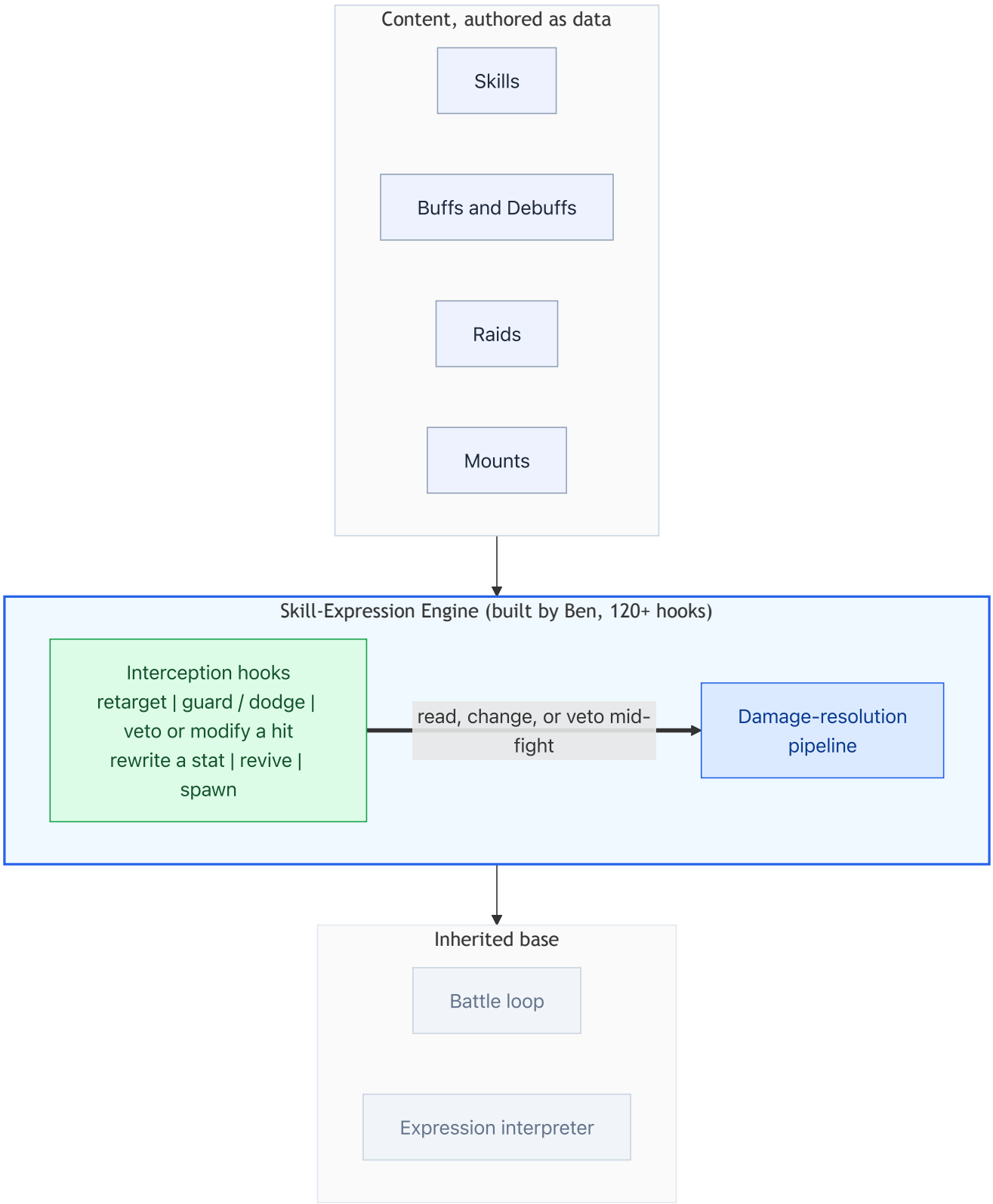
A combat engine you author as data, not code

The core of my work is an **expression system**: skills, buffs, and enemy behaviors are written as small scripts that fire on combat lifecycle events and call named primitives. Instead of hard-coding each new mechanic in C#, designers (and I) compose them from a vocabulary of hooks. I expanded that engine and built the large majority of the authoring surface designers work in: **92 of the engine's 121 skill functions and most of its lifecycle events**, the surface every new mechanic in the game is authored against. I shipped **over 500 PRs in roughly seven months**.

The part I am proudest of is the **interception model**. I added the hooks that let a piece of data reach into the middle of damage resolution and read, change, or veto what happens:

- A guardian can pull an incoming hit off a fragile ally, absorb it, and counterattack.
- A last-stand skill can survive a killing blow (revive on “would die”).
- A healer can convert overheal into bonus attack by rewriting a stat change before it lands.
- A summoned creature can act at the exact seam between its summoner's turns.

None of those required an engine change. They are authored as data on top of the interception hooks, which is what lets the game keep shipping genuinely new mechanics at content speed.


Content, authored as data
Skills
Buffs and Debuffs
Raids
Mounts
Skill-Expression Engine (built by Ben, 120+ hooks)
Interception hooks
retarget | guard / dodge |
veto or modify a hit
rewrite a stat | revive |
spawn
read, change, or veto mid-fight
Damage-resolution pipeline
Inherited base
Battle loop
Expression interpreter

Read top to bottom: the content players see is authored as data on the expression engine I built, which sits on a small inherited base. The green interception hooks reach into the resolution pipeline to read, change, or veto a fight, which is what makes new mechanics composable instead of one-off code.

A speed-based turn order that makes initiative a real strategy

I replaced a fixed turn order with a **speed-driven, mixed-initiative system**. Every actor's Speed stat decides when it acts, with owner and side inheritance for pets and mounts, slow mechanics, and a live mid-round re-sort when a speed buff lands.

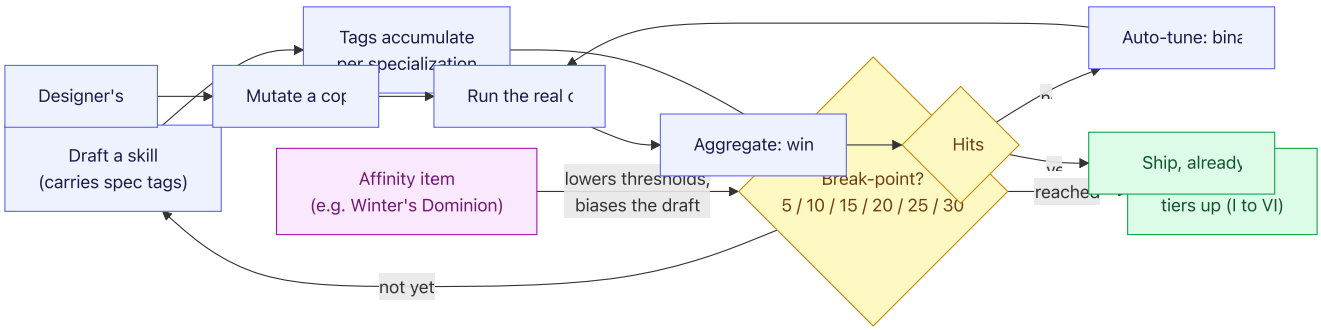
For players this turns Speed into a genuine build axis. In a raid you face a real decision: stack speed and burst the enemy before it swings, or stack attack and trade blows. It is the difference between watching a static queue and outmaneuvering a fight. I shipped it behind a feature flag with a clean fallback, so it went live invisibly and only changed behavior once content started using non-zero speeds.

An expression-based buff system

I re-architected buffs and debuffs so that a buff is no longer hard-coded behavior. It is a bundle of expression scripts that fire on lifecycle hooks (applied, removed, end of round, before a stat changes, and so on). Poison, stun-immunity, healing transfusion, evasion swarms: all of them are now data, run through the same compiled-expression engine as skills. This is the reason the game can keep introducing novel status effects without an engine release each time.

The Expertise system: a build identity every run

The roguelike heart of the game is the **8-Expertise system**, my flagship design contribution. As you draft skills during an adventure, each carries specialization tags. Accumulate enough tags and you cross a **break-point** that awards the next tier of that specialization's mythic skill. Eight archetypes (Pyromancer, Cryomancer, Stormlord, Bladedancer, Berserker, Warden, Lifebinder, Soulreaper) give every run a build identity to commit to and a mythic payoff to chase. It is the meta-game that makes a deterministic auto-battler worth running again.



One clean loop: draft skills, fill a specialization's tag meter, cross a break-point, tier up its mythic skill. Affinity items bend the thresholds. The whole meta is authored as data.

On top of it I built **Specialization Affinity**, a reusable system so a single equipped item can reshape the draft meta for one archetype, all as data with no new code per item. An affinity can lower a specialization's break-point thresholds so its mythic skills arrive sooner, bias the skill draft toward it, and keep it "warm" in the synergy window. That gives designers a clean lever ("this ring makes you a Cryomancer faster") and players a build-defining chase item. The Lifebinder track is a good example of the system in motion: a mythic skill where overheal converts into bonus attack, paired with an accessory that heals every round, which at full health becomes pure overheal that feeds the skill. A self-reinforcing archetype, built entirely in data.

Raids: strategic depth on top of the engine

Raids are the endgame: multi-enemy, multi-wave, mechanic-dense fights designed to make veteran players solve a puzzle of counters and reads. A few I designed:

- **Whiteout.** The environment erodes your defenses, then forces strategy choices with real combat trade-offs (dig through the ice and your melee drops, push through the storm and your ranged drops) before the fights even start. Enemies include debuff-immune gargoyles and mages that counterspell and deflect.
- **The Hollow Warden.** A beast that summons and resurrects thralls mid-fight using the deterministic summon system, with turn-order interleaving so a fast summon can act at the seam before its summoner swings.
- **The Crystal Hornet swarm.** An out-DPS pressure fight where one hornet becomes four by the end of a round, each new one more dangerous but more fragile, opened by a strategic attack-versus-speed choice.

The wave system, the formations, the guard and counter pipeline, and the summon mechanic underneath all of these are engine primitives I built and then designed content against.

A real elemental damage model

I led the overhaul from a flat “main attack plus skill” model to a true **Physical / Magical / True** damage axis layered over a five-element framework (Fire, Water, Nature, Dark, Light) with per-element resistances. It introduced genuine counter-play: magic attacks are harder to dodge, so mages hard-counter dodge-stacked teams; dodged area attacks still leak damage; and resistance above 100% flips damage into healing, which enables “immune and absorbing” boss designs. I rolled it out across several releases behind a live killswitch so it could be tuned safely on real traffic.

Reactive adventures and temporary items

I re-architected the adventure engine from pre-rolling a run's events up front to a **reveal-time resolver** that picks each day's event just-in-time against live run state. That made conditional storytelling a first-class authoring concept: events that only appear if you have been stealthy, or are below half health, or made a particular earlier choice. I also built an **ephemeral item system**, a single primitive that grants an item temporarily (usable for one adventure, never owned, never sellable). It powered try-before-you-buy, temporary quest weapons, and a tension mechanic where you sacrifice a temporary rune to break a set bonus.

Breadth: mounts, gear, luck, and loadouts

Beyond the headline systems, I designed and balanced most of the game's collectible surface: all of the **mounts** and their mechanics (including multi-mount deployment and a granular per-level progression model), all of the **gear sets**, a **luck and pity system** that quietly improves a player's odds after a losing streak so a randomized game never feels punishing, and a **loadout system** that lets players store and swap full gear and mount configurations in one tap.

BalanceLab: turning game balance into a search problem

A deterministic combat engine has a quiet superpower. Because the simulation is exactly reproducible, you can run it offline thousands of times and *predict* balance instead of guessing at it. I built **BalanceLab** to exploit that, and it changed how the whole team tunes the game. Instead of a designer hand-playing a few matches and forming a hunch, BalanceLab explores the real permutation space (parties, content, gear, and seeds) at a scale no human playtest can reach, and answers balance questions with evidence.

BalanceLab is a designer-facing what-if simulator. A designer asks a balance question in plain language, the tool mutates a copy of the game data or a real player's roster, runs the actual combat simulation across many seeds, and returns win rates, survival curves, and side-by-side deltas. No live data is touched.

The BalanceLab loop: a plain-language balance question becomes a mutated copy, runs through the real simulation across many seeds, and either ships or auto-tunes the lever toward the target band.

What makes it powerful:

- **Plain-language intent.** “Is Raid 25 beatable by a mid-game party?” or “run this elder party through it.” It fans out across seeds and reports the real clear rate.
- **Target-constraint solving.** “Tune this so only the top guilds, under five percent of parties, can beat it.” BalanceLab binary-searches the relevant number until it hits the target band automatically.
- **Real-player replay.** It can pull live guild rosters and replay them against new content, so balance decisions are tested against how people actually build, not against theory.
- **Lever sensitivity.** It runs a baseline plus one pass per variable and reports which lever moves the outcome, in which direction, and how strongly, so a designer knows that HP barely matters here but attack speed is the real driver.
- **Parallel sweeps.** It runs many experiments at once, so a full balance pass that used to mean days of manual playtesting becomes an overnight batch.

The result is a balancing process that runs on evidence and **cuts QA load dramatically**: a balance pass that used to mean days of manual playtesting becomes an overnight batch, and content is validated by thousands of simulated runs before it ever reaches a human tester. The team tunes with confidence instead of vibes. The simulation toolchain is essentially all mine.

Why this matters beyond one game

The deeper point is not the game, it is the capability. Most teams balance complex systems by intuition and slow manual testing, which does not scale and quietly burns enormous time. BalanceLab replaces that with a simulation loop: model the system, mutate it cheaply, run it at scale, and let the data point to the answer. That pattern generalizes far past combat tuning. Any organization with a system it can model (pricing and promotions, logistics, onboarding funnels, an in-game economy, a recommendation

policy) can trade guesswork for simulation and target-constrained search. Standing this up from scratch, and getting an organization to actually trust and use it, is a repeatable strategic capability I bring, not a one-off script.

How I think about design

The throughline across everything I build is **designing for engagement and moments of joy so people come back**. A few examples:

- **Make hidden state visible.** AFK rewards accrue and cap, but players had no sense of how much had banked. I built a timer that fills, turns red at the cap, and keeps showing time past full, so players feel the value building and come back to claim it. Engagement went up because the loop became legible.
- **Design the feel, not just the math.** A deterministic fight can look sluggish if actions resolve one at a time. I designed coordinated attack cascades so same-speed heroes and pets fire in tandem, which makes a fully scripted battle read as fast and alive.
- **Protect the emotional arc.** In a heavily randomized game, a string of bad luck is discouraging. My pity system quietly improves your odds after losses, keeping the tension while clipping the worst streaks.
- **Interrupt, don't punish.** When players were making a costly mistake, I shipped a smart confirmation that only fires when the action genuinely makes no sense, and a reversible path so they could self-correct. Guide the player, keep their freedom.

What players say

The systems above are not just shipped, they are validated by the people playing. Shiba Story Go holds a **4.72-star average across 167 App Store ratings**, and reviewers keep calling out exactly what I built: the game design, the *depth* that makes a run worth repeating, the battles, and the mounts.

> *"The polish and game design for this game are really best in class."*
>
> ★★★★★ App Store review

> *"You can play it fast, but there is enough depth to keep you entertained for months."*
>
> ★★★★★ Google Play review

> *“I love the battles.”*
>
> ★★★★★ App Store review

> *“Love the characters, the mounts, the art, even the sound effects.”*
>
> ★★★★★ App Store review

> *“It’s so much better than others of its type.”*
>
> ★★★★★ App Store review

By the numbers

4.72★
App Store rating across 167 reviews

559 / 522
PRs authored / merged in ~7 months

168
Merges in the peak month

92 / 121
Skill functions authored, the engine's authoring surface

114k+
Test executions, zero failures, via a probabilistic-assertion harness

8
Expertise archetypes in the roguelike meta

days→hrs
BalanceLab turns a manual balance pass into an overnight batch

Flagship systems designed end to end: the **8-Expertise meta**, **Specialization Affinity**, the **elemental damage model**, the **reveal-time adventure engine**, the **ephemeral item system**, all **mounts and gear sets**, **luck and pity**, and **loadouts**. Hardened with statistical testing: a custom probabilistic-assertion harness, data-integrity validators, and CI gates.

> *Shiba Story Go is a live title at Proof of Play. The systems above are in production. Happy to walk through any of them, with code, in a conversation.*

Ben Young · Special Projects Lead, Proof of Play

benyoung.ai · **o.benjamin.young@gmail.com**