A Data-Driven Framework for Game Development *

Luke Del Giudice (wcc5ub) *University of Virginia*

I have been co-developing a first-person shooter video game with my brother since the beginning of college, almost four years ago. The game, called Astro, can be summarized as "Call of Duty in zero gravity." Multiplayer shooters are typically produced by large studios, and balancing such games (tuning weapons, movement, and other systems in a high-dimensional design space) usually requires entire teams dedicated to the task. Since Astro's development team consists only of my brother and me, I have not been able to invest significant time into systematic balancing or data-driven decision making. This changed recently with a redesign of Astro's networking architecture to be fully deterministic and driven entirely by player inputs, which I can store efficiently. As a result, entire matches can now be resimulated and reviewed without storing the full game state, unlike traditional networking solutions. This capability has enabled several development improvements, including the ability to collect data easily after games by resimulating matches and extracting all relevant information from the reconstructed game state. In this project, I collected and analyzed data from the most recent Astro playtest to provide insights for future balancing and development, ensuring that upcoming changes reflect actual gameplay rather than our limited personal experience or the subjective impressions captured in post-playtest surveys.

Introduction

Astro's networking solution, *NetTick*, is a fully deterministic lockstep networking system, which runs the entire game simulation detached from the presentation (graphics, sound, UI, etc).

Definition

Deterministic Lockstep: Networking Model where only inputs are sent across the network, and the game is simulated completely deterministically so that the results on all clients is exactly the same, bit-for-bit, without state replication.

The result, as mentioned, is that any game can be replayed by simply replaying the sequence of inputs, which just requires capturing the input stream of every player. An input snapshot is 41 bytes, however, a compressed sequence can get down to approximately 3 bytes per input. Therefore, serializing an entire game (approximately 10 to 15 minutes with 8 players) is just 200 to 400 KB. This makes it possible to upload every game to an AWS S3 bucket. Later, when I want to run analyses, I attach a custom measurement script to the game simulation and spin up a cluster of EC2 instances. Each instance pulls replay files from the S3 bucket, re-simulates the corresponding games, and uses the script to collect and aggregate the desired data in the background. This process of re-simulation took approximately one and a half hours for the analysis outline in this report on 113 games played by 119 different playtesters. I have attached the C++ script *ReplayDriver_GameStats.cpp* that performed data aggregation in the submitted repo, and if it's needed for confirmation I can send the executable that actually performs the simulation like usual after attaching *UReplayDriver_GameStats* as the replay driver and writes the results to disk. However, while this replay and initial aggregation pipeline is relevant to a computing course, its remaining implementation details are largely domain-specific (custom JSON serialization and interactions with *NetTick* specific architecture, which is proprietary and outside the scope of this work), and the overall AWS workflow closely mirrors the structure of the subsequent analysis stage. The more interesting component is the offline analysis of

*Formatting based on svmille guide (http://github.com/svmiller).

the aggregated data, which is where the statistical techniques come into play. Like the replay step, this analysis is embarrassingly parallel, so the multiprocessing design is straightforward; most of the effort went into specifying and implementing the logic for the different proposed studies.

These studies address a range of key questions that I developed to align with our current development priorities. This report outlines the overall code structure of the repo, explains the three data structures involved, discusses key challenges and requirements of processing the data, and finally describes the proposed studies along with their implementations and results.

Approximate Timeline

I initially spent time reading balancing papers to understand how larger studios and academics approached data-driven balancing. While this did inspire some of the studies I created, most of the papers were academic and relied on synthetic data, which is entirely inappropriate for balancing a high-skill 3D shooter. After this research phase and extensive discussion with my brother, I began writing up the actual studies we wanted to pursue. Finally, I implemented the studies that were deemed highest priority and not focused on long-term longitudinal results.

Overview of the Pipeline

The data flows through two distinct phases: extraction (per match) and analysis (per study). The *master_analysis.py* script acts as the central hub, passing data objects between the **etl** layer and the **analysis** layer.

To be more specific: the script *game_etl_core.py* is where all of the logic processing game state lives, *game_etl.py* acts as an interface between this game state logic and the *master_analysis.py* script, all of the study files are in the *studies/* directory, and the results of running *master_analysis.py* are stored in the *results/* folder (results are further organized into the subfolders *graphs/*, *models/*, *spatial/*, and *data/*).

Multiprocessing and Optimization

Given that this analysis is going to be used for future (hopefully) larger playtests, I decided to utilize Python's *concurrent.futures.ProcessPoolExecutor* to implement multiprocessing (this was chosen over multithreading due to Python's GIL limiting true parallelism for CPU-bound tasks).

(1) Data Parallelism

In the first phase, the workload is embarrassingly parallel. Each match (consisting of one JSON file and two CSVs) is independent of every other match.

- I first detect the number of available logical cores.
- Then I map the list of 113 match file groups across these cores.
- Then each worker process instantiates its own *GameProcessor* and executes the etl logic for all studies.
- Lastly, the resulting data dictionaries are serialized (pickled) and returned to the main process.

(2) Task Parallelism

In the second phase, I pivoted to task-based parallelism. Once all data is aggregated into memory, the independent studies are submitted as separate tasks.

- To prevent pickling errors common with Python modules, a dynamic import wrapper (*run_study_wrapper*) is used. The master script passes the string name of the module to the worker.
- The worker dynamically imports the module and executes the *run* function.
- This ensures that CPU-intensive tasks, such as s8 or s9, do not block the generation of simpler studies (ie: s0).

(3) Performance Benchmarks

Table 1: Performance Impact of Parallel Processing (24 CPU cores)

Metric	SingleCore	MultiCore	ImprovementFactor
Extraction Time	47s	11s	**4.3X**
Analysis Time	15s	15s	**1X** (overhead)
Total Runtime	1m 2s	27s	**2.3X**

Data Sources

(1) Match JSON

This is the complete record of all relevant events that happen in a match. While any given game has tens of thousands of ticks, I can ignore this complexity by mainly interacting with the relevant ticks as determined by the previous re-simulation step.

Key Content:

- `GameId`, `GameMode`, `Map`, `DurationSec`, `winner` / `winning team` - basic metadata
- `DeviceProfiles` – array of hardware and graphics settings for every player in the match (GPU, CPU, resolution, FPS cap, etc)
- `Players` - per-player summary statistics (Kills, Deaths, AvgSpeed, etc)
- `Items` – item usage statistics (PlayTime, Kills, Deaths, etc for each item)
- `events` – chronological event stream (the following list is not comprehensive)
  - `join`, `leave`, `spawn`, `set_team`, `set_loadout`, `equip` - non-gameplay events
  - `fire`, `throw`, `swing`, `stab`, `burst` - various fight related events
  - `damage` - whenever a player takes any damage
  - `elim` - whenever a player dies
  - `dash`, `kick`, `surface_lock`, `push_off` - movement events

(2) Player Update CSV

High-frequency telemetry snapshot of every player's state for when required information is disconnected from events.

Key Content:

- `Stamp` – timestamp in the same units as the JSON events (1 stamp = $\frac{1}{40}$ second)
- `PlayerId` – the same ID that appears in the JSON events

- `Health`, `Location.X`, `Location.Y`, `Location.Z`
- `Velocity.X`, `Velocity.Y`, `Velocity.Z`
- and many other metrics relating to player state (`Ammo`, `Rotation.Roll`, etc)

(3) Performance CSV

Client performance collected for every player in the match for tracking optimization.

Key Content:

- `Frame Avg` - frame time in milliseconds
- `Packet Latency` – round-trip ping in milliseconds
- `GPU Bound`, `GT Bound`, `NTT Bound` - percentage of frames that were bottlenecked by GPU, game thread, or *NetTick* thread
- and many other metrics relating to player performance or hardware (`RAM MB`, `GPU Avg`, etc)

Methods and Implementations

Here are the specific details of the data processing and studies mentioned, and a high-level overview of their implementations. The "P" prefix means the section references processing, while "S" indicates a corresponding study that matches the file prefixes in the repo.

(P1) Tracking User State

The match data processed is entirely event driven as previously described. These events contain fields that further identify what's happening in the event beyond what type it is. For example, an `equip` event will also have the fields `stamp`, `player`, and `item`. When this event is examined it shows that the specified player switched to that item at the given time. However, a few complications arise from this examination. First, the `player` field, which gives a player id integer, is not unique to a single player. For example, a player could join and get assigned that id, but if they disconnect mid-match then this id is free to be assigned to a new player joining later. This is obviously a problem for analysis that focuses on each [player's abilities][(10) Skill Estimation], so I need to keep track of who's who using the recoverable Steam usernames.

To solve this, the *game_etl.py* script reconstructs the timeline in the **_reconstruct_gamestate** function where the mapping between an id and player is a function of time. Every time a `Join` event is detected, a new session segment for that specific id is created linking it to the username. This segment remains open until a corresponding `leave` event is found or the match ends. When a specific data point needs to be processed, I simply resolve the given id at the specified timestamp with the **resolve_user** function. This same logic is also used for tracking the loadout state of a player. The script builds a parallel timeline of `equip` events. To determine what item was used in a fight, I query this timeline with the **get_weapon_at_time** function for the last known item change prior to the fight's start time.

(P2) Fight Detection

Raw game data is a continuous stream of discrete events: a shot fired here, damage taken there, a random kick. The concept of a "Fight" does not exist in the raw data; it is a logic concept that must be derived for the purposes of analyzing thousands of fights systematically.

The challenge here was to create a definition that makes sense for fights. This is the final definition of a "Fight" I used:

- Any time a player damages another (this is an event), a potential fight is started between just these two players (all fights under this definition are one versus one).
- I look back 3 seconds to find the first "fight action" (`fire`, `kick`, `throw`, etc) since Astro uses a projectile system (not hitscan) and the initial damage is rarely the result of an action occurring at the same timestamp as the `damage` event. This effectively rewinds the start time of the fight to include the initial shots that might have missed, providing a more accurate measurement of the engagement's actual duration.
- From here, a specified timer (configured as 5 seconds) begins counting down. If no other "fight action" occurs by the two players fighting, then the fight is considered over and it's logged as a non-fatal fight. Otherwise, if one of the players dies, then I immediately identify all active fights involving that victim. These fights are marked as fatal, and the survivor in each pairing is credited with a win (but that win is scaled based on how much damage they contributed).

This process also builds on the tracking of user state from the previous data processing; for example, verification needs to be done to make sure both players are still in the game and any id still belongs to an original fight initiator.

(S0/S1) Metadata and Descriptive Statistics

Before performing advanced modeling, I established a comprehensive baseline of the dataset by aggregating high-level metadata and fundamental player/item statistics across all 113 processed matches.

Game mode and map frequencies were computed directly from the `GameMode` and `Map` fields in the match JSON files. These reflect player preferences, as the game allows voting between matches.

Player-level statistics (total kills, deaths, playtime, and wins) were aggregated using the identity resolution system described in Section 2. This system maps transient numeric player ids to persistent usernames across rejoins and multiple matches, enabling the construction of lifetime profiles for each participant.

Item usage statistics were derived from the event stream:

- Weapon swaps were counted via `equip` events.
- Time held was calculated by differencing consecutive equip timestamps per player, with the final segment capped at match end time.
- First-pick rates were determined by tracking the first observed loadout per player per match.

These metrics provide an unbiased view of item popularity and player behavior before effectiveness analysis.

(S2) Health Regeneration Dynamics

The game features passive health regeneration: a fixed delay after taking damage, followed by periodic healing ticks. To quantify how often players enter fights below full health, which directly lowers effective TTK, I modeled the long-term health distribution using a Markov chain Monte Carlo simulation.

From the fight detection system, I extracted two empirical distributions: * Total damage taken per fight (representing health lost in a single engagement). * Idle time between fights (time available for regeneration).

I then simulated thousands of fight–rest cycles using the exact in-game regeneration rules (5 second initial delay, $+1$ health every 2 seconds). For each simulated fight, I sampled damage and idle time independently from the observed distributions, applied regeneration logic, and recorded the resulting starting health for the next fight.

The stationary distribution of this Markov chain represents the equilibrium probability of beginning a fight at any health value from 0 to 10. This provides a data-driven estimate of average combat health, and can even be used to test new regeneration rules since those values are parameters in the code.

(S3) Item Win Rate

Raw win rates per item are heavily confounded by context (distance, health, player skill) and kill stealing. To isolate true item effectiveness, I fitted a weighted binomial generalized linear model (logistic regression) predicting fight outcome (win/loss) from the primary item used, controlling for starting distance and health.

To solve kill-stealing and multi-weapon fights, I implemented a damage-proportional fractional credit system:

- If a player dealt all 10 damage needed to kill, their item receives full weight (1.0).
- If they dealt only 3 damage, their item receives weight 0.3.
- The losing player's items are weighted by their own damage contribution.

This ensures cheap finishers (ie: switching to Sword at the last moment after enemy is already low health) are down-weighted, while sustained performers are properly credited. The model outputs odds ratios relative to a baseline item (Astro Rifle), interpretable as: "holding the Spark increases win probability by 42% versus holding the Astro Rifle, all else equal."

A mixed-effects version was initially planned to control for individual player skill but was abandoned due to library limitations with frequency weights. Damage attribution was prioritized over skill adjustment.

(S4) Spatial Death Density

There was no way to get an understanding of where fights were taking place on the map from some 2D visual, as evidence by the 2D player death heatmaps generated in this section using kernal density estimation. To address this, I developed a 3D spatial analysis pipeline to visualize high-density death clusters directly within the game's native engine (Unreal Engine 5).

The spatial extraction process aggregated x, y, z coordinates for all `elim` events. Because raw point clouds of thousands of deaths are visually noisy and difficult to interpret, I applied a two-stage clustering algorithm to identify hotspots:

- **Micro-Clustering**: I first applied MiniBatchKMeans to group local neighbors into small micro-clusters (5 deaths per group)
- **Macro-Merging**: I then fed these centroids into an AgglomerativeClustering algorithm with a distance threshold of 3 meters. This recursively merged adjacent micro-clusters until spatially distinct blobs of activity remained.

This hybrid approach reduced thousands of raw data points into a manageable set of weighted centroids, where the weight corresponds to the total death count in that specific area.

To visualize these 3D centroids, I engineered a custom interoperability pipeline between Python, Blender, and Unreal Engine:

- **Geometry Generation (Blender)**: the script *death_cluster_importer.py* ingested the processed CSV data into Blender. Using Geometry Nodes, it instanced spherical meshes at each centroid coordinate, scaling the sphere radius logarithmically based on the death count to distinguish high kill areas.
- **Data Baking**: a critical challenge was preserving the death count data during the export to a static mesh. The *death_cluster_exporter.py* script solved this by converting the geometry instances into actual polygons and mathematically mapping the death count to a dynamic blue to red color gradient. This gradient was baked directly into the mesh's vertex colors, ensuring the data persisted without needing complex textures.
- **Engine Integration (Unreal)**: the resulting FBX was imported into Unreal Engine. By applying a material that drives Emissive Color using Vertex Color data, I projected the historical death density directly onto the level geometry.

This study will allow my brother and I to fly through the map and instantly identify choke points, spawn traps, and unfair lines of sight that are invisible in 2D analysis.

(S5) Sequential Pattern Mining

Given the movement-heavy nature of the game, I investigated whether specific sequences of locomotion actions predict success or failure.

For every elimination, I extracted the 10 second movement event stream (ie: `dash`, `kick`, `surface_lock`") for both killer and victim. These were converted into 2-grams, such as `dash` to `kick`. Frequencies were computed separately for successful and unsuccessful outcomes. By comparing these frequencies, I can identify tactical patterns; for example, discovering that a `dash` to `kick` sequence appears 30% more often in successful engagements than in fatal ones would help confirm that learning base movement is improving player's performance.

(S6) Functional Velocity Analysis

Beyond the discrete actions themselves, the actual shape of a player's movement speed before a kill is also predictive. Does accelerating immediately before a fight improve survival, or is sustained speed more important? To test this, I used pointwise logistic regression on functional velocity curves.

All curves were time-aligned to $t = 0$ (moment of kill/death) and left-padded with NaN where data was unavailable (pre-spawn). At each time point, a separate logistic regression predicted outcome (kill vs death) from instantaneous speed.

The resulting coefficient curve shows when speed is most predictive; for example, a strong positive peak at approximately -2.8 seconds would indicate that high velocity specifically during the approach phase is a powerful predictor of winning the engagement.

(S7) Survival Analysis

Theoretical TTK ignores accuracy, movement, and positioning in favor of a simple fomula ($\frac{\text{dps}}{\text{health}}$). I computed real behavioral TTK using the Kaplan–Meier estimator on fight durations from the detection system discussed previously.

Two curves were produced:

- All engagements (including timeouts); this measures average fight length.
- Fatal fights only; this is the true empirical TTK distribution

The median of the fatal fights curve represents the time at which 50% of lethal fights conclude, which is the effective TTK.

(S8) Skill Estimation

During my research for this project, I focused in on more proprietary systems being used to rank players beyond ELO. During that search I found TrueSkill, the Bayesian ranking system developed for Halo initially. While more advanced approaches have since been created, I settled for the original simple framework to score each player from the playtest's overall skill.

Per-match performance scores were computed as:

$\text{Score} = \text{Kills} - \text{Deaths} + 5(I_{\text{Won}})$

Players were ranked within each match by this score and fed into TrueSkill, which maintains a belief distribution (mean μ, uncertainty σ) per player and updates it based on whether the match results were surprising (ie: a low-rated player outscoring a high-rated player). This converges on a statistically robust skill rating for every unique user, with the oppurtunity for future modifications to the performance score formula as new game modes are developed.

This study, in particular, will extend well to future playtests where returning playtesters' would keep their ranks. This could also be used to test if a new mechanic is difficult for the previously skilled players to adjust to.

(S9) Playstyle Archetypes

Next, I sought to categorize players into "styles" (ie: "Passive Gunner", "Speed Demon") without manually defining the thresholds for those categories. Internally, my brother and I always believed that playtesters could be categorized into very specific archetypes; this study was designed with the goal of collecting evidence of this. To test the hypothesis that players naturally segregate into distinct playstyles, I applied K-means clustering ($k = 3$) to four behavioral features:

- Time spent above high-speed threshold
- Dashes per minute
- Average absolute roll angle
- Mean fight initiation distance

Features were standardized before clustering to ensure that units (ie: degrees vs meters) did not bias the distance calculations. The algorithm grouped players into 3 distinct clusters based on the similarity of their feature vectors. This allows me to mathematically define playstyles based on the data rather than human intuition.

(S10) Performance Aggregation

This final study focuses on the client performance CSVs previously dissuced.

Network performance significantly impacts gameplay fairness. Averages can be misleading because lag spikes are rare but impactful. Therefore, while I did aggregate all of the performance files for each player to allow for any metric to be analyzed, I decided modeling the distribution of packet latency would be the most helpful first analysis.

So I start by collecting raw packet latency samples from every player across every match, filtering out invalid zero-values. Latency data is typically right-skewed (mostly low, with a long tail of lag spikes). A Normal distribution is a poor fit for this. Instead, I decided to fit a Gamma Distribution to the data using

maximum likelihood estimation. By solving for the Gamma parameters (α, β, location), I generated a probability density function. This allows me to calculate precise probabilities for "bad experience" thresholds, such as determining exactly what percentage of gameplay time occurs above 100 ms latency.

Results

Since I wanted to keep this report from being too long, I'll only show/discuss a few of the results. If you want to see any other studies, the results are in the repo.

The study that took the most time, but had the most satisfying result was s4. Figure 1 is a screenshot inside the Outpost map level in UE5 showing the clusters of death locations marked by size and color.

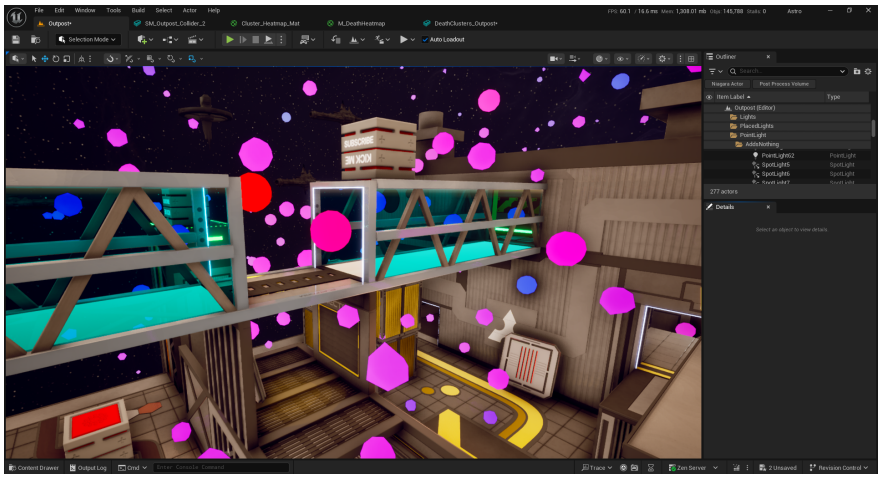

Figure 1: Unreal Engine Editor Screenshot

"I have already identified weak map areas, such as the Outpost basement, where fights occurred too infrequently and with too little positional variety. This will directly motivate changes to these map layouts to keep all areas viable.

Another important result was from s7, which summarized the K/D as seen in Figure 2.

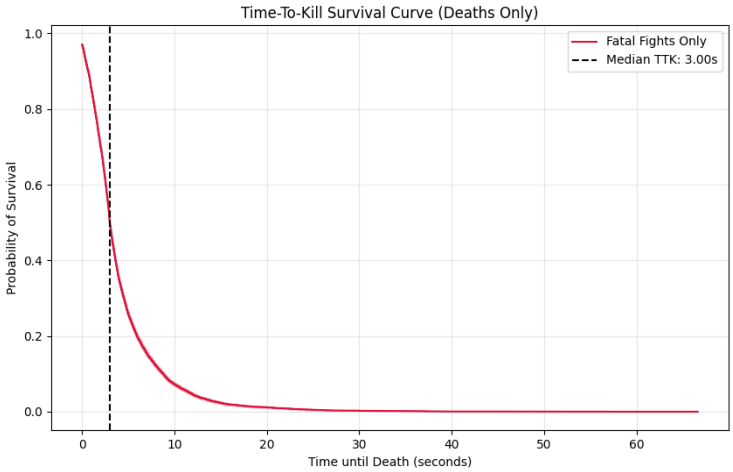

Figure 2: s7 TTK Graph

This establishes that the current effective K/D of Astro is three seconds (time at which 50% of lethal fights conclude). As we continue to update gameplay, we can monitor if our changes are actually effecting TTK. The more important observation from this graph, though, is how quickly the probability of survival drops off after just a few seconds. A major focus of development has been increasing the strategy and opportunities to extend fights; for example, the kick mechanic grants momentary invulnerability to help players escape direct combat. It's clear that these prolonged fights are more rare than hoped, and we'll be monitoring the skewness in this graph to hopefully increase not just the median, but also the probability of sustaining active fights over longer periods where actions are continually performed, countered, and so on.

The last result I'll focusing on was the odds probability graph created for the lethal items (non-lethal items like the grapple had to be excluded for this study).

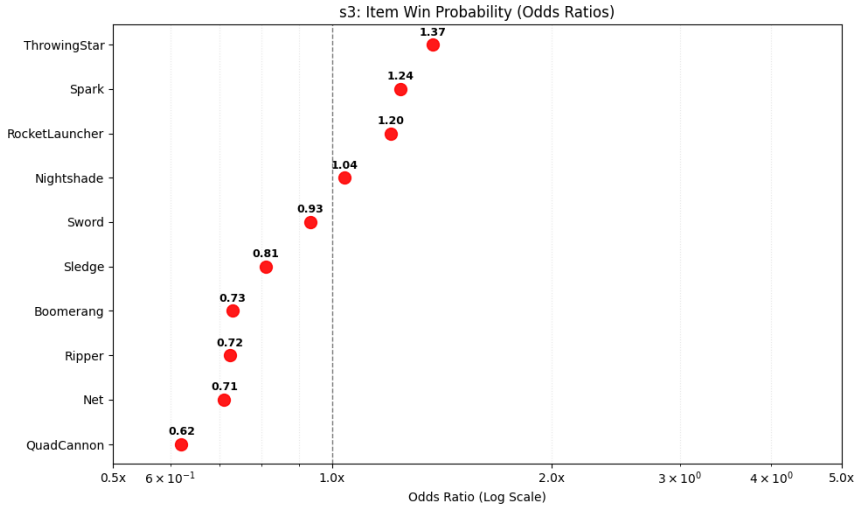

Figure 3: s7 TTK Graph

This was the most important result to obtain, as unbalanced items have consistently been the bane of playtests. Until now, we've only been able to balance based on direct feedback from playtesters and our own intuition. Now, we can actively begin the process of nerfing the Throwing Star and buffing the Quad Cannon.

Conclusions

Overall, this project has been a successful introduction to gameplay data analysis for Astro. As development continues, I plan to refine the pipeline to capture more insights and connect data across different sessions. A key focus will be linking returning players to create longitudinal studies, which will help control for the different skill levels of our playtesters. To support this, I intend to implement a persistent database. Ideally, the final result will be a completely hands-off system that runs automatically in the background after every playtest.