POLITECNICO DI TORINO

**Laurea Magistrale in Ingegneria Informatica,
orientamento Grafica e Multimedia**

Politecnico
di Torino

Tesi di Laurea Magistrale

Data Collection Pipeline all'interno di un prodotto videoludico multigiocatore

Relatore

Prof. MARCO MAZZAGLIA

Candidato

RICCARDO PERELLI

Luglio 2022

"Questa è la mia storia"
-Tidus, Final Fantasy X

Abstract

Questo elaborato è parte di un progetto più grande, progettato e sviluppato da un team studentesco chiamato *Bugstards*, un videogioco multiplayer in Unreal Engine 4.
è stato sviluppato un prototipo che si ispira a vari giochi e videogiochi già sul mercato (primo tra tutti, il dodgeball). L'applicativo è un RTS-Sportivo, in cui i giocatori si affrontano sfruttando un set di abilità e caratteristiche proprie del personaggio che sceglieranno. In particolare, questa tesi si concentra sullo studio delle tecnologie e architetture legate alla pipeline di dati, analizzando alcune delle novità che attualmente sono più utilizzate in ambito di data engineering. Si è cercato di approfondire gli aspetti coinvolti nella costruzione di una pipeline di telemetria in un videogioco real-time, sfruttando paradigmi architetturali come il *lakehouse*, in voga negli ultimi tempi.
Oltre al ruolo di Data engineer, sono stati coperti altri ruoli, alcuni in condivisione con gli altri partecipanti al progetto come Game Designer, Gameplay Programmer e Project manager. è stato necessario uno studio approfondito degli aspetti tecnici di unreal engine per implementare un sistema di abilità complesso e scalabile, in prospettiva di agganciarlo al sistema di collezionamento dei dati.
Nello specifico la tesi si divide in 3 sezioni:

1. Ideazione, progettazione e analisi delle regole di gioco, un tuffo nel game design di BugBall.

2. Studio e utilizzo del Gameplay Aybility System, un plugin di Unreal che permette di creare un sistema di abilità e caratteristiche molto vasto e flessibile.

3. Analisi delle caratteristiche delle pipeline di analisi dei dati e utilizzo del paradigma lakehouse. seguita da un'implementazione di un sistema di collezionamento dei dati in game prodotti dal sistema di abilità implementato in Unrreal Engine e spiegazione delle scelte effettuate per l'implementazione dell'architettura progettata. In questa fase si sono approfondite tecnologie come MongoDB, Pyspark e Databricks.

In ogni capitolo si discuterà del perchè sono state fatte determinate scelte e si approfondiranno gli aspetti implementativi che hanno portato alla realizzazione di una pipeline funzionante, basandosi sulle necessità che un videogioco multiplayer presenta.

Ringraziamenti

Ho pensato davvero tanto a cosa scrivere in questa sezione, mi è già capitato, quasi 3 anni fa, di scrivere dei ringraziamenti per la mia tesi triennale, ma, diversamente da questo, quel progetto è stato fatto in fretta e furia, e con prospettive diverse. Mi ricordo che il giorno dopo la mia laurea triennale, sono andato a fare un servizio da cameriere, era il modo con cui pagavo i miei studi. Non ero felice, non avevo spazio per godere dei miei piccoli successi, e avevo la malsana credenza che fosse dovuto, in parte, all'ambiente che mi circondava e nella quale ero cresciuto. Ma adesso, ancora ragazzino, con una consapevolezza totalmente diversa, mi rendo conto di quanto quell'ambiente mi avesse dato la forza necessaria per inseguire i miei obiettivi e i miei sogni, di quanto quelle persone, che io credevo fossero un limite, sono state il carburante che mi spingeva ad andare oltre.
Voglio che il lettore non badi all'ordine dei ringraziamenti, tutti i menzionati hanno un posto nel mio cuore, e, seppur in modi diversi, ognuno di loro mi ha aiutato a raggiungere questo obiettivo. Ringrazio le mie sorelle e mio fratello.
Annalaura, perchè è una forza della natura, molto più forte di me; parlare con lei, e ammirare la dedizione che applica al lavoro che ama, mi ricarica, sono grato che tu sia mia sorella e che tu sia sempre disposta ad ascoltarmi.
Michela mi ha insegnato che nessuna sfida è grande abbastanza per non essere affrontata, e che grazie all'amore, si possono costruire dei bellissimi campi di fiori a partire da qualsiasi maceria, grazie per essere mia sorella.
Ad Alessandro, il nostro piccolo fratello, dico grazie per continuare a farmi sentire bambino. Grazie per non farmi smettere di sognare, e sopratutto, ti ringrazio davvero tanto, perchè continui ad accettare la mia assenza, continui ad accettare che il lavoro e gli obiettivi che inseguo a volte mi allontanano da te, ma sappi che il mio cuore avrà sempre spazio per te. Ringrazio i miei genitori, perchè hanno sempre creduto in me, hanno sempre avuto fiducia nelle mie scelte, e non hanno mai opposto resistenza alle mie ambizioni. E anche se a volte non lo do a vedere, è anche grazie a loro che ho fede nel futuro.
Ringrazio mia madre, perchè lei mi ha insegnato la libertà, mi ha insegnato che nessuno, più di noi stessi, sa cosa è giusto, e niente vale di più di combattere una battaglia in cui si crede davvero.

Ringrazio mio padre, perchè mi ha insegnato cosa significa il sacrificio per un bene superiore, mi ha insegnato che nella vita nulla è mai dato per scontato, e che ognuno di noi ha la forza per farsi carico delle proprie responsabilità.
Un ringraziamento speciale va a mia zia Monica, che mi ha accolto nella sua casa quando ne avevo più bisogno, senza chiedere nulla in cambio. Grazie a lei ho capito cosa vuol dire fare del bene per il bene, e non per ricevere qualcosa in cambio, una lezione che non scorderò mai. Inoltre, grazie a lei ho imparato che l'amore supera ogni barriera, che l'amore può salvare il mondo, e anche se non lo farà, salverà le persone che lo provano.
Un ringraziamento anche a Federico, che in questo ultimo anno ha quasi sempre cucinato per me, quando ero impegnato con gli esami o con il lavoro, in lui, ho trovato un amico, grazie, non lo dimenticherò mai.
Ringrazio mia cugina Jessica, sempre pronta a farmi ragionare nei momenti in cui sono meno lucido, sempre pronta a svegliare le mie sinapsi per riportare il Riccardo che conosce alla realtà, anche lei, una forza della natura, il mio specchio, una donna che ammiro profondamente.
Ringrazio Roberto, suo marito, per essere un amico vero, una persona con cui poter parlare di tutto, e un compagno di viaggio insostituibile, aspetto con ansia il nostro prossimo concerto.
Ringrazio tutti i piccolini della nostra famiglia; Sveva, Ettore ed Ameliè, per riempirmi il cuore di calore ogni volta che sono in loro compagnia, per farmi credere ancora una volta che il futuro è luminoso.
Ringrazio i miei nonni, che non smettono mai di fare un passo verso di me e comprendere questo "nerd informatico" per loro incomprensibile. Loro mi hanno insegnato che le grandi cose richiedono tempo e pazienza, e la bellezza non risiede in automobili lussuose od orologi sgargianti, la vera bellezza si nasconde nei momenti, negli attimi vissuti insieme e raccontati 40 anni dopo, di nuovo, nell'amore e nella fiducia verso l'altro. La vita è anche costruzione, e questo, non lo dimenticherò mai.
Un ringraziamento va anche a mio cugino Thomas, con il quale posso sempre scambiare profonde conversazioni sulla vita, l'universo e tutto quanto, un'inestimabile fonte di confronto.
Grazie per essere la mia famiglia.

Vorrei dedicare qualche altra riga per ringraziare tutti gli amici che mi sono stati vicini in questo lunghissimo percorso, ho sempre considerato gli amici come una seconda famiglia, ed è proprio su questo principio che 7 anni fa abbiamo fondato il gruppo *Arfamo la Rotonda*, ringrazio ognuno di loro perchè nonostante io sia sempre via, sempre in giro, e indaffarato con le mie cose, ogni volta che torno nel mio paese e sono insieme a loro sembra che non sia passato un momento, li ringrazio perchè hanno sempre festeggiato insieme a me i piccoli successi di questo

percorso, e ho sempre avuto la possibilità di condividere la mia passione per i videogiochi insieme a loro.
Voglio dedicare qualche riga per ringraziare il mio amico Federico, maestro pasticcere che non demorde mai, e porta la sua passione per la pasticceria ad una raffinatezza eccelsa, grazie, sei molto d'ispirazione.

Ringrazio tutti i miei amici del mio paese, che continuano a fare il tifo per me, che mi riempono di calore sui social, e con cui posso sempre scambiare qualche conversazione motivazionale.

Ringrazio tutti i miei amici di Torino, con cui ho condiviso i momenti più belli di questa magistrale, in cuor mio, spero di portarvi per sempre insieme a me.
Ringrazio i miei colleghi del team Bugstards, con i quali ho condiviso l'esame di game desgin che poi mi ha portato alla realizzazione di questo progetto. Ringrazio Gianluca e Simone, inestimabili compagni di tesi, senza di loro, questo lavoro non sarebbe stato possibile, grazie mille, spero di tornare a lavorare con voi.
Un ringraziamento speciale va a Monica, con la quale ho condiviso tutti i progetti d'esame del percorso magistrale, un'inestimabile compagna di viaggio, tu, in questa Torino, sei stata la mia Virgilio, grazie davvero.

Ringrazio tutti i miei coinquilini per avermi sempre sopportato, e aver accettato tutte le mie lune, grazie, non è stato per niente scontato per me.

Inoltre, vorrei ringraziare il mio professore, sempre presente e paziente, disposto a darci consigli di ogni tipo e a metterci in contatto con esperti del settore per migliorare sempre di più il nostro lavoro, la ringrazio davvero, lei è stato un punto importante della mia carriera, e spero continuerà ad esserlo.

Infine, voglio ringraziare Benedetta, con cui ho condiviso 5 fantastici anni della mia vita, mi ha sempre sostenuto, più di quanto lo facessi io, un abnegante di natura, sempre disponibile ad accogliere i miei dubbi e farmi ragionare nel modo giusto, sono certo che senza di lei non sarei qui. Mi ha insegnato davvero tanto, e sono estremamente grato che abbia fatto parte della mia vita. Avrai sempre un posto nel mio cuore, Grazie.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Compiere delle scelte è sempre difficile, sia per l'individuo, che per la grande azienda, entrambi, si sono affidati all'istinto per moltissimi anni, e anche se questo può indirizzarci correttamente, non è sempre l'unico modo per compiere la scelta giusta. Negli ultimi anni, abbiamo sentito parlare di Big data in ogni salsa, e ne sentiremo parlare sempre di più.
In questi tempi, più che mai, abbiamo bisogno di compiere delle scelte utilizzando i dati a nostra disposizione, e il valore che questi possono rivelare è immenso. Il mondo dei videogiochi non è certamente rimasto immune dalla generazione dei dati, sopratutto nei videogiochi multiplayer di stampo competitivo, dove il bilanciamento delle statistiche di gioco che caratterizzano il gameplay è fondamentale per far percepire un senso di equilibrio ai giocatori. Si pensi a *League of legends*, che negli ultimi anni ha raggiunto la quota di 180 milioni di videogiocatori attivi in tutto il mondo. I big data, all'interno di questa realtà, sono fondamentali per diversi aspetti, che coinvolgono sia i giocatori che gli sviluppatori. Come sarebbe stato possibile aggiornare il gioco ogni anno, rendendolo sempre più appetibile, e robusto, se non compiendo delle scelte *Data-Driven*?
Partendo dallo sviluppo del gioco, e di un sistema di abilità robusto, sviluppato in Unreal Engine, sfruttando il plugin "Gameplay Ability System", in questa tesi si vorranno analizzare gli aspetti che un Data Engineer deve affrontare per costruire una pipeline di analisi dei dati sfruttando tecnologie all'avanguardia, in previsione del fatto che questa possa crescere in maniera indefinita. Inoltre, si analizzeranno alcuni degli strumenti che offrono la possibilità di costruire una pipeline di questo tipo, come **MongoDB, PySpark e Databricks**.

1.1 Motivazione

Da che ho memoria, sono stato appassionato di videogiochi. Sono sempre stato affascinato dal fatto che il videogioco fosse un medium che offrisse un così grande spazio interpretativo. Può essere puro divertimento come il classicissimo *Super Mario*, può essere un'opera a metà tra l'informatica e il cinema come *Metal Gear Solid*, oppure, può essere un'arena dove più giocatori si sfidano tra di loro come in un vero e proprio stadio che pulula di tifosi. Crescendo ho iniziato a sviluppare anche altri tipi di interessi legati al mondo della tecnologia, che viene utilizzata egregiamente nel mondo videoludico.
L'idea di questo videogioco viene principalmente per la mia passione per League of Legends, e, più in generale, per il mondo Esport. Un mondo che, purtroppo, in Italia viene ancora poco ascoltato, ma che offre delle possibilità che ancora non ci immaginiamo, e quando ad Ottobre 2021 ho partecipato al primo evento di business Esport alle OGR di Torino, insieme al mio relatore, il prof. Mazzaglia, ho capito che questa fosse la strada da intraprendere. In particolare, mi sono sempre interessato di analisi delle partite, ma in generale, nella mia vita fare affidamento ai fatti concreti, prendere decisioni basate su osservazioni che potevo dimostrare, è sempre stato un punto di forza.
Quindi, anche per questa scelta, ho fatto esattamente la stessa cosa, ho guardato i dati che avevo a disposizione, un team di persone all'avanguardia disposte ad intraprendere questa strada insieme, un prof. volenteroso a seguire il progetto, e una passione per i videogiochi che tutt'ora è ardente in me, la scelta è stata ovvia.

1.2 Outline

I capitoli sono organizzati nel modo seguente:

- **Capitolo 2 - Stato dell'arte:** Viene analizzata la storia del fine-tuning dei videogiochi, partendo dai primi giochi, in cui le decisioni venivano prese più istintivamente piuttosto che *Data-driven*, fino ai moderni giochi online, con migliaia di persone connesse nella stessa partita, in cui vengono catturati tutti gli eventi che gli utenti compiono all'interno del gioco, permettendo di conoscere sempre di più le criticità che affliggono il prodtto videoludico. Infine, si descriveranno brevemente le tecnologie che hanno guidato questo sviluppo.

- **Capitolo 3 - Prodotto videoludico:** Descrizione del videogioco e dei suoi concetti chiave, seguita dalla descrizione dei ruoli ricoperti all'interno del progetto.

- **Capitolo 4 - Studio correlato:** Studio dei problemi principali nella creazione di una data pipeline, delle architetture e delle soluzioni tecnologiche più utilizzate, e spiegazione delle scelte prese per lo sviluppo del progetto.
- **Capitolo 5 - Game design:** Analisi del percorso che ha portato dall'ideazione, al raffinamento del concetto iniziale del gioco, approfondendo le modifiche che il playtesting ha imposto all'idea concepita inizialmente.
- **Capitolo 6 - Unreal Engine: Gameplay Ability System:** Approfondimento sulle funzionalità offerte da Unreal Engine e motivazioni su alcune scelte programmatiche. Analisi del Gameplay Ability System, e del ruolo fondamentale che ha avuto nello sviluppo del nostro videogioco.
- **Capitolo 7 - Data Pipeline:** Macro e micro approfondimento sull'architettura generale di cattura e analisi dei dati. Quali scelte sono state fatte e seguendo quali criteri, quali tecnologie sono state coinvolte nello sviluppo, punti migliorabili e di forza.

Capitolo 2

Stato dell'arte

La cattura dei dati generati dai giocatori è diventata una parte essenziale all'interno dei videogame. Questa è la porta d'ingresso per vari tipi di analisi che possono essere effettuati in un prodotto videoludico. L'unione tra l'attività di cattura dei dati e i vari processi di analisi, che su di essi possono essere costruiti, prende il nome di *Game analytics*. Quest'attività può essere inserita nel cappello dell'analisi comportamentale, che a sua volta, è una disciplina che fa capo alla *business analytics*, che ha lo scopo di rilevare dei pattern comportamentali volti a prendere decisioni *data-driven*.

La prima volta che la parola *business intelligence* è stata usata in un'opera scritta è stato nel 1865, da Richard Miller Devens.

L'autore usò queste parole per descrivere come il banchiere Henry Furnese riuscì a contrastare la concorrenza collezionando, analizzando e usando tutte le informazioni a sua disposizione per compiere migliori scelte di business.

L'importanza di utilizzare la parola *business intelligence* in quel tempo, risiede nel fatto che venga utilizzata per descrivere l'utilizzo di dati e evidenze empiriche, piuttosto che fare affidamento su istinto e superstizioni, per implementare strategie di business. Questo segna l'ingresso del metodo scientifico nei problemi di scelta legati al business.

Dopo un secolo, agli albori della rivoluzione digitale, nel 1950, la *business intelligence* diventò un processo scientifico indipendente adottato dagli imprenditori. In particolare, nel 1956 IBM inventò il primo *hard disk*, che all'epoca conteneva non più di 5 MB, fu un cambio di paradigma importante che aprì le porte alla digitalizzazione delle informazioni, gettando le basi per l'automatizzazione dei processi decisionali. Due anni dopo, Hans Peter Luhn, un ricercatore dell'IBM pubblicò un paper che illustrava un sistema di business intelligence applicativo a problemi reali, con sistemi, per il tempo, innovativi.

"Abstract: An automatic system is being developed to disseminate information to

the various sections of any industrial, scientific or government organization. This intelligence system will utilize data-processing machines for auto-abstracting and auto-encoding of documents and for creating interest profiles for each of the "action points" in an organization. Both incoming and internally generated documents are automatically abstracted, characterized by a word pattern, and sent automatically to appropriate action points. This paper shows the flexibility of such a system in identifying known information, in finding who needs to know it and in disseminating it efficiently either in abstract form or as a complete document. "

All'epoca, le tecnologie erano ancora distanti da quanto potesse essere offerto dalla digitalizzazione delle informazioni, solo negli anni 60 IBM segnò un nuovo punto di svolta in questa disciplina, introducendo il primo *Database management system* o DBMS, chiamato IMS. In questo periodo si iniziò a parlare di indipendenza, sicurezza, integrità e organizzazione dei dati.

Negli anni 80, l'ingresso delle tecnologie di data warehousing, segnarono l'inizio di una nuova era per l'analisi dei dati. Il data warehouse ha dominato in ambito di business intelligence, fino agli anni 10 del 21esimo secolo. Con i nuovi paradigmi di acquisizione dei dati, nati dopo l'avvento degli smartphone e dei vari dispositivi mobili interconnessi, sono stati evidenziati i limiti delle tecnologie utilizzate, è in questo scenario che un nuovo paradigma ha iniziato a farsi spazio. Inoltre, l'avvento delle tecnologie di Machine learning e AI ha introdotto la necessità di un accesso ai dati diretto, e non basato su SQL.

Il data lake entra in gioco per ovviare ai problemi del data warehouse, ma non necessariamente sostituendolo, spesso, integrandosi nelle architetture esistenti.

Con il tempo, anche il data lake ha mostrato il suo tallone d'achille, è per questo che dal 2020 si sta parlando sempre di più di data lakehouse, un'architettura, che nel migliore dei modi, cerca di unire i punti di forza del data warehouse e del data lake.

Nel nostro caso specifico, le statistiche elaborate sono state utilizzate per raffinare le scelte di design. In particolare, gran parte del processo di sviluppo di un videogioco è dedicato alla correzione e al miglioramento delle meccaniche previste. Per molto tempo, questo è stato fatto manualmente, in due modi:

- Osservazione diretta dei giocatori
- Compilazione di questionari con domande mirate

Solitamente, queste tecniche sono utilizzate in contemporanea. Purtroppo, presentano molti limiti, specialmente, quando il videogioco è già stato rilasciato. post-rilascio il prodotto ha bisogno di manutenzione, deve essere aggiornato per mantenere il suo *appeal*, in questa fase, più di ogni altra, la cattura dei dati

si rivela fondamentale per il ciclo di vita del prodotto. La mole di dati cresce e risulterebbe confusionario e limitante basarsi solo su osservazioni pratiche e feedback degli utenti. Le aziende videoludiche non possono fare a meno di implementare un'attenta pipeline di collezione dei dati, che sfrutti le giuste tecnologie.

2.1 Data warehouse

Un data warehouse è un raccoglitore di informazioni pronte per l'analisi, con lo scopo di compiere scelte consapevoli. Un data warehouse è organizzato a strati, il più intellegibile è lo strato di presentazione, che si interfaccia con lo strato di computazione, che, a sua volta, si interfaccia con il database dove vengono effettivamente immagazzinate le informazioni.

Il data warehouse è popolato da varie fonti, da sistemi transazionali a database relazionali, tipicamente, a cadenza regolare. I dati prima di essere salvati passano attraverso un processo di *ETL*. Un processo in cui i dati vengono estratti da varie fonti, e salvati nel data warehouse, pronti per la fase di presentazione.

Non tutti i data warehouse mantengono la stessa architettura, di fatto, negli anni si sono viste molte varianti dell'architettura classica. Ad esempio, è pratica comune aggiungere un layer di staging intermedio, in cui vengono immagazzinati i dati prima di essere effettivamente scritti nel data warehouse.

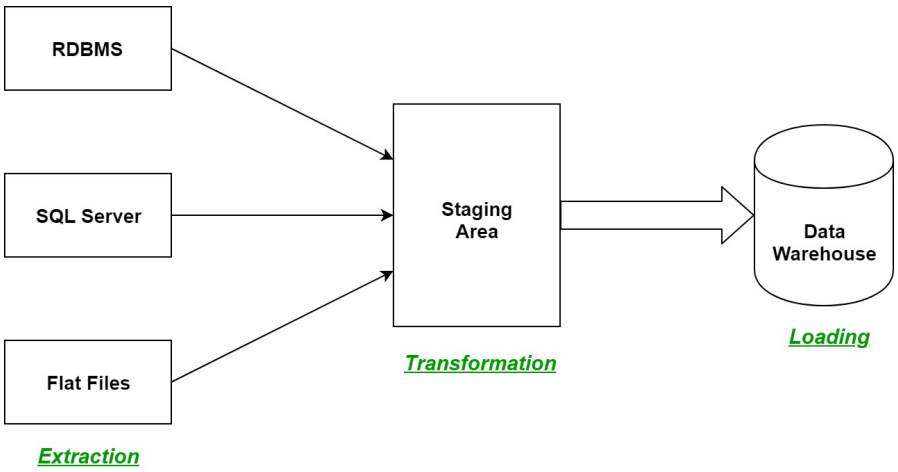

Figura 2.1: ETL process

Il processo di ETL si divide in tre fasi:

1. **Extraction**: Il primo passo del processo di ETL, in questo passo, i dati provenienti da diverse fonti vengono estratti, possono essere in diversi formati,

è fondamentale che i dati di questo formato non entrino direttamente nello storage.

2. **Transformation**: In questo step vengono applicate una serie di regole e funzioni ai dati estratti, rendendoli digeribili dal data warehouse. Possono essere coinvolti più processi di trasformazione come:
   - *Filtraggio*
   - *Pulizia*
   - *Join*
   - *Divisione*
   - *Ordinamento*
3. **Loading**: L'ultimo step è il processo di caricamento. I dati trasformati vengono caricati nel data warehouse. La cadenza di caricamento può variare da intervalli brevi giornalieri, a processi mensili.

Oppure, è possibile aggiungere un layer di organizzazione in più, aggangiando al data warehouse dei *data marts*, composti da sotto-insiemi dei dati contenuti nel data warehouse che fanno capo ad una specifica funzione o area dell'organizzazione aziendale.

Infine, con l'aggiunta dei data marts, l'architettura può seguire principalmente due approcci:

- **Bottom-up**: in cui i dati vengono immagazzinati nei data marts, che poi andranno a popolare il data warehouse.

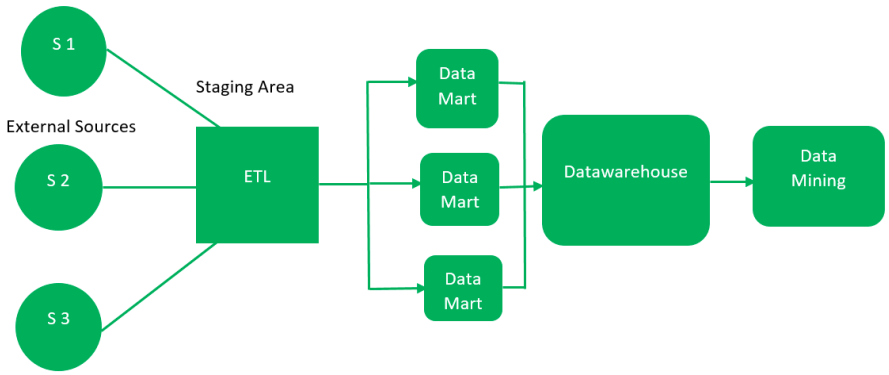

Figura 2.2: Bottom-up architecture

- **Top-down**: in cui i dati vengono immagazzinati direttamente nel data warehouse e i data marts sono ricavati da essi.

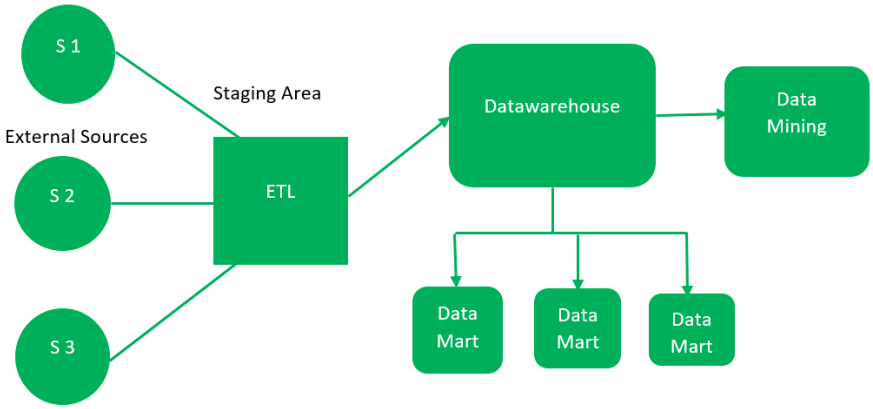

Figura 2.3: Top-down architecture

2.2 Data lake

Il data lake è un contenitore di vari tipi di dato, grezzi, non strutturati, semi-strutturati e strutturati, provenienti da fonti diverse e senza uno schema preciso. I dati grezzi sono dati non ancora processati e senza uno scopo ben definito, ad esempio, per un'applicazione social, gli eventi generati sulla piattaforma sono dati grezzi, da cui possono essere estrapolate diverse informazioni. I data lake sono stati concepiti per contenere dati che "potrebbero avere valore", il loro valore viene definito col tempo, al variare delle richieste e sfruttando comparti di ricerca e sviluppo. Permettono ai data scientist di minare e analizzare grosse moli di dati, i cosi detti *Big data*, termine coniato per fare riferimento ad una quantità di dati tale da essere ingestibile con le classiche metodologie SQL. nel 2008, il framework *Hadoop* ha aperto le porte alla gestione dei big data. Nel 2010, James Dixon, coniò il termine *data lake*, contrapponendolo a quello di data mart:

"If you think of a Data Mart as a store of bottled water, cleansed and packaged and structured for easy consumption, the Data Lake is a large body of water in a more natural state. The contents of the Data Lake stream in from a source to fill the lake, and various users of the lake can come to examine, dive in, or take samples."

I data lake necessitano di un sistema di governance e manutenzione, che li renda accessibili e utilizzabili.
Il rischio è di lasciare che i dati diventino spazzatura, costosi e inutili. Un data lake che cade in questo stato è chiamato *data swamp*.

Capita spesso che data lake e data warehouse vengano confusi, ma non sono la stessa cosa, ed inoltre, hanno scopi differenti. L'unica similarità è che mantengono dati. Molte imprese usano queste architetture sinergicamente. Un data warehouse

fornisce un modello di dati strutturati, pronti per la business intelligence, di contro, il data lake non necessità di un processo strutturato di ETL per mantenere i dati. Infatti, non è raro trovare architetture in cui il data lake si rivela essere la fonte di un data warehouse.

Un data lake ha un'architettura meno strutturata, piatta, vista la natura dei dati che mantiene. Anche per questo offrono una scalabilità molto ampia fino all'ordine degli *exabyte*. Questo è importante perchè, solitamente, non si è a conoscenza della mole di dati che verranno caricate.

2.2.1 Data lake and Data warehouse differences

La necessità di effettuare un confronto tra le due tecnologie non nasce con scopi sostituivi, ma integrativi. Evitando confusione, il data lake non sostituisce il data warehouse, ma è un paradigma innovativo nato da necessità differenti, che arricchisce l'architetture di collezionamento dei dati possibili.

- **Data collection**: al contrario del data warehouse, il data lake non richiede una struttura dei dati pre-definita.
- **Data processing**: nel data warehouse, la struttura del database è definita come una priorità, il dato è scritto in modo strutturato, e letto con la consapevolezza dello schema presente. Nel data lake, invece, i dati sono acquisiti nel loro formato nativo, ogni elemento riceve un identificativo e una serie di metadati.
- **Agilità e flessibilità**: essendo molto strutturato, cambiare la struttura di un data warehouse risulta essere molto oneroso. Nel data lake, è possibile cambiare i modelli in modo flessibile.
- **Costi**: vista la sua natura più "fluida", il data lake risulta essere più economico a parità di dati mantenuti, di contro, il data warehouse fornisce dei dati più solidi in termini di intellegibilità dell'informazione.

2.3 Data lakehouse

Le aziende hanno scoperto velocemente che alcuni obiettivi delle architetture basate su data lake non sono stati raggiunti per via della mancanza di alcune caratteristiche fondamentali: scarso supporto di transazionalità, data quality o governance scadente e scarse ottimizzazioni di performance. In particolare, anche l'unione di data lake e data warehouse ha delle limitazioni, che vanno oltre la manutenibilità:

- Mancanza di flessibilità: i data warehouse mantengono i dati in formati proprietari, che aumentano i costi in caso di migrazioni ad altri sistemi. Nella maggior parte dei casi, i data warehouse forniscono un accesso basato su SQL, rendendo difficile eseguire con altri motori computazionali come sistemi di machine learning.

- Supporto limitato per il machine learning: nessuno dei più popolari sistemi di machine learning, come TensorFlow o PyTorch, funzionano bene con i data warehouse. A differenza della business intelligence, che estrae piccole porzioni di dato, i sistemi di machine learning hanno bisogno di grandi quantità di dati che sfruttano codice non-SQL. per risolvere questo problema, i fornitori di data warehouse suggeriscono di esportare i dati, ma questo processo aggiunge un layer di complessità all'architettura.

- Compromessi forzati tra data lake e data warehouse: più del 90% dei dati aziendali vengono immagazzinati in data lake per via della loro facilità di accesso e riduzione dei costi. Per contrastare le basse performance e i problemi di qualità dei dati dei data lake, le aziende necessitano di un processo di ETL da eseguire su sotto-insiemi di dati, e inserire i risultati su data warehouse per quanto concerne decisioni più importanti. Questa dualità dei sistemi richiede una continua manutenzione del processo, il che risulta essere costoso.

è in questo scenario che è iniziato ad emergere un nuovo paradigma architetturale chiamato *data lakehouse*, che ha l'obiettivo di integrare al meglio possibile la flessibilità e la velocità del data lake, con l'affidabilità e la data governance del data warehouse. Il data lakehouse cerca di risolvere i problemi discussi precedentemente in questo modo:

- Mancanza di flessibilità:
  - Open file format: costruiti su formati standardizzati come Apache Parquet.
  - Open API: Permette l'accesso ai dati diretto senza la necessità di motori proprietari.
  - Supporto di differenti linguaggi: Supporto per le librerie di machine learning di R e Python.

- Supporto limitato per il machine learning:
  - Supporto per diversi tipi di dato: può immagazzinare, raffinare e analizzare diversi tipi di dato dalle immagini ai dati strutturati.
  - Lettura non-SQL efficiente.

  - Versionamento dei dati utilizzati per gli esperimenti di machine learning: Rende disponibili delle "fotografie" dei dati, permettendo ai team di data science e machine learning di accedere a versioni dei dati precedenti.

- Compromessi forzati tra data lake e data warehouse:

  - Ottimizzazione delle performance: Utilizzo di varie tecniche di ottimizzazione, come il caching, clustering multi-dimensionale e *data skipping*.

  - Supporto transazionale: Supporto per transazioni ACID, così da assicurare la consistenza.

  - Archiviazione a basso costo: tipicamente le architetture lakehouse sono costruite utilizzato storage in cloud a basso costo come Amazon S3, Azure blob storage o Google Cloud Storage.

Cerchiamo di riassumere tutto in un formato più leggibile:

	Data warehouse	Data lake	Data lakehouse
formato dei dati	chiuso, formati proprietari	Formato apero 23	Formato aperto
tipo di dati	Dati strutturati, con supporto limitato per i dati semi-strutturati	Tutti i tipi	Tutti i tipi
accesso ai dati	solo SQL, no accesso diretto	API aperte per accesso diretto con differenti linguaggi	API aperte per accesso diretto con differenti linguaggi
affidabilità	Qualità alta, affidabili con supporto per tx ACID	Qualità bassa, rischio data swamp	affidabili con supporto per tx ACID
governance e sicurezza	Sicurezza e governance ben definite sia a livello di tabella che di righe e colonne	Scarsa governance la sicurezza necessità di essere applicata a files e item	governance ben definite sia a livello di tabella che di righe e colonne
performance	Alte	Basse	Alte
scalabilità	Scaling molto costoso	Scala per mantenere ogni grandezza di dato a basso costo, al di la del tipo	Scala per mantenere ogni grandezza di dato a basso costo, al di la del tipo
supporto per differenti scenari	limitata alla BI, SQL e al supporto decisionale	Limitato al machine learning	Comprensivo di BI, SQL e machine learning

Capitolo 3

Prodotto videoludico multigiocatore

Lo scopo è sviluppare un prototipo avanzato di un videogioco multiplayer in real-time. La presente tesi tratta di un progetto di gruppo svolta assieme a due studenti magistrali del Politecnico di Torino. Il team, dal nome 3Bugstards, ha lavorato su tre fronti principali, ciascuno dei quali supervisionato e curato da uno dei tre membri. I tre ambiti sono di seguito elencati:

- **Tech Art:** caratteristiche estetiche del videogioco.
  - Modellazione
  - Animazione
  - UI Design
  - VFX
  - Audio design
  - Shader Programming
- **Engineering:** caratteristiche funzionali del videogioco.
  - Gestione del multiplayer
  - Ottimizzazione
  - Collezione e pulizia dei dati di gioco
  - Gestione dell'IA
  - Gameplay Programming
- **Design:** figura essenziale che fa da regia nella creazione del prodotto e si occupa del design del videogioco.

3.1 Descrizione

Il prodotto videoludico realizzato è **BugBall**, un *RTS-sportivo*, ovvero uno strategico in tempo reale che richiede ai giocatori capacità meccaniche e strategiche, individuali e di squadra.
L'ispirazione proviene dal Dodgeball, ma con la modifica di gran parte delle regole. Il campo della partita è suddiviso in due aree, ogni area ospita una squadra di tre giocatori. Trattasi dunque di un 3 vs 3, ove l'obiettivo è spingere fuori i componenti del team avversario per accumulare punti e vincere la partita. Ogni giocatore eliminato deve aspettare un tempo di respawn per tornare in campo. Le partite sono a tempo e si svolgono al meglio dei 3 incontri, pertanto si può ottenere una vittoria matematica concludendo in trionfo 2 match su 3. Le condizioni di vittoria del singolo match sono le seguenti:

- Eliminare tutti e tre gli avversari prima dello scadere del tempo.
- Allo scadere del tempo, aver totalizzato un numero di punti maggiore della squadra avversaria
- In caso di pareggio, si giocheranno i tempi supplementari e vincerà la squadra che eliminerà tutti i giocatori avversari (un giocatore eliminato durante i supplementari non può rientrare)

La peculiarità del videogioco è il concept dei personaggi realizzati, rappresentati da insetti umanoidi. Ognuno di essi ha caratteristiche diverse dall'altro, può essere più difensivo, più portato all'attacco o equilibrato. In aggiunta, il personaggio ha a disposizione tre slot abilità e un'abilità passiva unica che lo contraddistinguono. Le abilità sono varie e tra le più importanti ci sono gli attacchi speciali contro l'avversario (Skillshot) o attachi ad area (AoE), dando anche la possibilità di fornire supporto al team con dei potenziamenti (Buff).
Prima dell'inizio di ogni match la squadra ha a disposizione del tempo per pensare ad una strategia e impostare il set di abilità, scegliendo quelle più appropriate per scardinare il team avversario.

3.2 Ruolo primario

3.2.1 Data Engineer

Il ruolo di primo piano svolto all'interno del team è il Data Engineer. E' una figura professionale in ambito IT ormai necessaria in ogni realtà, la funzione principale è preparare i dati per usi analitici o operativi, garantendone la disponibilità, nonché

la loro qualità e fruibilità a chi li utilizza per metterli al servizio, il più delle volte, del business. In questo caso particolare, i dati sono messi a disposizione del gioco, dei suoi sviluppatori, e degli utenti. Per la realizzazione del prodotto videoludico multigiocatore, e più in generale, di un'applicazione user-centered, è indispensabile avere a disposizione più informazioni possibili riguardo l'utilizzo dell'applicativo, in questo modo possono essere effettuate delle scelte precise, scovando i punti critici e i punti di forza del prodotto sviluppato. Si sente parlare spesso di *Data-Driven Design*, e il data engineer è colui che rende possibile ad una realtà di essere data-driven. Senza una pipeline di acquisizione dei dati ben costruita, è impossibile avere accesso alla conoscenza che questi nascondono. Normalmente il data engineer e il data analysts sono due figure separate, ma in questo caso i dati acquisiti hanno avuto uno scopo ben definito: aiutare i game designer nell'individuazione di errori di design.

3.3 Ruoli secondari

Lo svolgimento della tesi prevede in aggiunta all'attività di Data Engineer, l'impiego di tutti i ruoli inerenti alla sezione progettazione e sviluppo del videogioco. Trattandosi di un piccolo team di tre persone, è stato ritenuto necessario che ognuno ricoprisse dei ruoli secondari, descritti qui di seguito.

3.3.1 Game Designer

Il Game designer si occupa di fornire la visione del gioco e delle meccaniche che lo comporranno. Il game designer nel mondo videoludico è la controparte del regista nel mondo cinematografico. In piccole realtà come 3Bugstards, il Game designer ricopre di solito ruoli diversi che spaziano da quelli artistici a quelli, come in questo caso, tecnici. Poichè quasi tutto il resto delle attività dipendevano dagli output di questo ruolo, le prime tre settimane di lavoro sono state dediche quasi totalmente ad attività legate a questo ruolo. In corso d'opera sono stati necessari diversi raffinamenti che sono tutt'ora in corso. Il design del gioco prende spunto da diverse fonti, tra le più intellegibili troviamo:

- **League of Legends**
- **Dodgeball**
- **Windjammers**
- **Blaston (logica di scontro tra i proiettili)**

3.3.2 Gameplay Programmer

Il Gameplay Programmer è il responsabile dell'implementazione programmatica delle meccaniche di gioco, e sistemi ad esse annessi. Il gameplay Programmer lavora a stretto contatto con il game designer, sopratutto in team di piccole dimensioni. Il mio ruolo come G. Programmer in BugBall è stato espletato ricoprendo due macro aree principali dello sviluppo del gioco, l'implementazione dell'ability system e delle regole generali del match di gioco. Nella prima è stato sfruttato il plugin *Gameplay Ability System* messo a disposizione da unreal engine, in questa fase sono state create le basi per tutte le abilità presenti all'interno del gioco e relativa logica di gestione (cooldowns, consumo del mana, danni, status, etc). Nella seconda è stata implementato il flusso di gioco all'interno di una partita, le regole generali, la gestione dei respawn, la gestione dello scoring system e della mappa di gioco.

3.3.3 Project Manager

Quando si lavora in gruppo, è fondamentale avere una figura che si faccia carico della gestione dei compiti, e che, tramite l'ausilio di strumenti adatti, cerchi di definire delle linee guida generali circa il procedimento dei lavori. Si è cercato di adottare un approccio lean e agile. Essendo stato il target molto ambizioso, il modello waterfall sarebbe stato obsoleto e limitante, poichè avevamo necessità di testare man mano il prodotto e di migliorare strada facendo, ci siamo concentrati il più possibile su avere qualcosa di giocabile in tempi rapidi, e questo vale anche per la parte di data engineering. Per questo, ci siamo approcciati allo sviluppo con una metodologia Agile, l'obiettivo è stato quello di raggiungere dei checkpoint ogni due settimane. Di fatto, gli incontri con il nostro relatore sono stati a cadenza, il più delle volte bisettimanali. Tutti i componenti del team hanno avuto un doppio compito, e nelle due settimane di sprint ognuno di noi faceva capo ad una serie di task centrali, e gestiva individualmente la sua parte specifica del progetto.

Capitolo 4

Studio correlato

Il lettora avrà ormai chiaro che la cattura dei dati in un prodotto videoludico multigiocatore, ma più in generale in qualsiasi applicazione digitale, è un punto fondamentale per l'evoluzione e il miglioramento. Le tecnologie stesse evolvono di pari passo alle necessità degli utenti e dei designer.
Inizialmente non era necessario catturare ogni singola informazione, e il fine-tuning si limitava ad una semplice osservazione dei giocatori o dei componenti del team stessi.
Per prodotti di alto livello che puntano al mercato competitivo è fondamentale avere un solido sistema di telemetria, che permetta di individuare le criticità nella maniera più veloce possibile.
Lo studio del prototipo videoludico della tesi ha come obiettivo finale la costruzione di una pipeline di cattura e pulizia dei dati che garantisca alcune caratteristiche fondamentali:

- Facilità di utilizzo anche per figure non-tecniche.
- Taglio dei costi di implementazione.
- Flessibilità implementativa.
- Scalabilità.

4.1 Facilità di utilizzo

Non è raro trovarsi in situazioni in cui delle figure non tecniche (Designer, animatori, artisti), vorrebbero monitorare un determinato parametro all'interno del gioco.
Il problema è che spesso questo non è immediato, per situazioni specifiche può capitare che si crei un collo di bottiglia sulle figure tecniche che si occupano della cattura dei dati.

La soluzione presentata in questo elaborato mira a risolvere questa problematica rendendo disponibile, in blueprint, le API di interrogazione del sistema di cattura dei dati.
In questo modo, anche per chi volesse solo prototipizzare alcune nuove meccaniche, è possibile agganciare il sistema telemetrico ed, eventualmente, analizzarne i dati in un secondo momento.

4.2 Riduzione dei costi

Le tecnologie utilizzate sono di tipo open-source, che garantiscono delle risorse gratuite entro una certa mole di utilizzo. L'idea è quella di permettere a chiunque abbia bisogno di una pipeline di cattura dei dati di poterla utilizzare in tempi brevi e a costi zero.

4.3 Flessibilità implementativa

Molto spesso le soluzioni custom e a costo zero soffrono di un'elevata complessità di implementazione.
In generale, la soluzione più semplice dal punto di vista implementativo sarebbe quella di agganciarsi ad un servizio terzo che si occupi della cattura dei dati, utilizzando API assodate (es: Google Analytics), ma spesso queste soluzioni vanno in contrasto con il punto precedente, poichè i principali vendors di data analytics offrono delle prove limitate del loro servizio o poco customizzabili. Per questo, la soluzione cerca di rendersi il più manutenibile possibile anche per i non coinvolti nello sviluppo iniziale. In particolare, non è richiesta nessun tipo di configurazione ai collaboratori del team che non si occupano dello sviluppo. Questo è reso possibile anche dell'utilizzo esclusivo di tecnologie cloud, in particolare, i dati catturati vengono inviati (tramite server) a *Mongo DB Atlas*, Un Database-as-a-Service (DBaaS), un servizio che permette di configurare e deployare un database senza preoccuparsi dell'hardware che lo compone e i relativi update necessari.

4.4 Scalabilità

La scalabilità descrive l'elasticità di un sistema.
Solitamente, viene associata all'abilità di un sistema di crescere al crescere dell'utilizzo che se ne fa di esso, ma una definizione di questo tipo non è esaustiva. Lo scaling può anche essere effettuato a ritroso, togliendo risorse non necessarie.

La scalabilità di una risorsa può avvenire principalmente in due modi:

- **Verticale**: Lo scaling di tipo verticale si riferisce all'aumento della potenza computazionale del singolo server o cluster. Sia i database relazionali che non possono scalare verticalmente, naturalmente con dei limiti massimi di potenza e throughput. Inoltre, i costi di uno scaling verticale non sono lineare e dipendenti dal costo di hardware più performante.

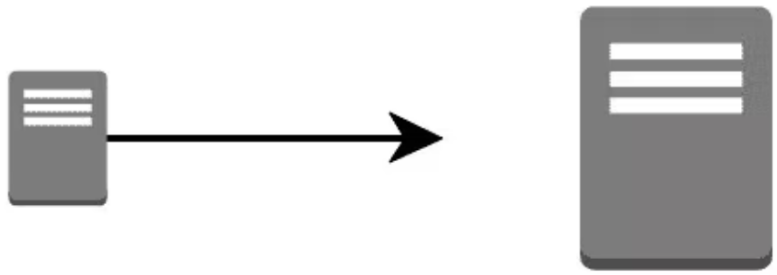

Figura 4.1: Scaling Verticale

- **Orizzontale**: Viene chiamato anche *scale-out*, è un processo nel quale vengono aggiunti dei nodi che condividono il carico di lavoro. è molto difficile con i database di tipo relazionale vista la necessità di distribuire le informazioni sui vari nodi della rete. Con i database non-relazionali, è più semplice dato che le collezioni sono *self-contained* e non accoppiate in modo relazionale. Questo permette di distribuirle tra i nodi facilmente, dato che non c'è bisogno di compiere delle *join*.

Figura 4.2: Scaling Orizzontale

Lo scaling orizzontale di Mongo DB è ottenuto attraverso due tecniche:

  - **Sharding**: I dati vengono distribuiti tra diversi nodi che formano il database. Ogni nodo contiene un sotto-insieme di tutti i dati. Questa tecnica è particolarmente efficace per aumentare il throughput in casi in cui è richiesto un grande quantitativo di scritture, dato che ogni operazione

coinvolge solo uno dei nodi. In *MongoDB Atlas* lo sharding avviene in modo automatico. Con il passare del tempo, i dataset che formano i database non crescono in modo uniforme, e diverse *shards* hanno un tasso di crescita maggiore di altre, sarà necessario ribilanciare i dati periodicamente assicurando una distribuzione uniforme. Questa distribuzione dei dati sbilanaciata è risolta da Mongo attraverso un processo chiamato *shard balancing*.

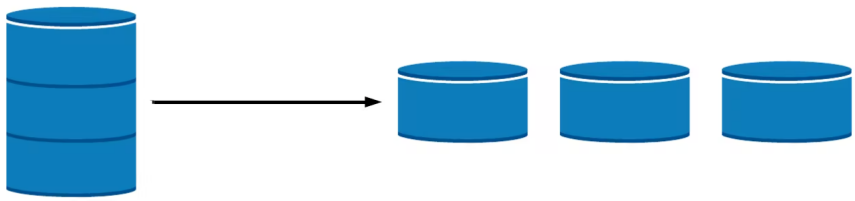

Figura 4.3: Sharding scaling

– **Replica Sets**: Può sembrare simile allo sharding, ma la differenza sostanziale è che il dataset è duplicato piuttosto che distribuito. La replicazione permette di avere una disponibilità del dato molto alta, una gestione puntuale delle ridondanze, e una diminuzione di colli di bottiglia nell'accesso ai dati. Purtroppo, replicare tutti i dati presenta grossi problemi nel caso di molte operazioni di scrittura, dato che ogni update deve essere propagato attraverso tutti set replicati.

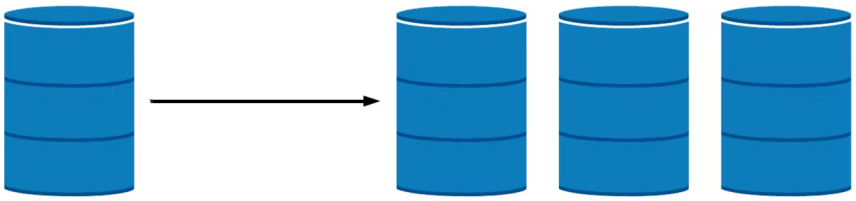

Figura 4.4: Replica set scaling

La scalabilità della soluzione nel nostro caso è garantita da due fattori:

- Auto-scaling di mongo db: Al crescere dell'applicazione, ogni pezzo che la compone deve scalare con essa, in particolare, i servizi di telemetria che hanno l'onere di catturare sempre più eventi. Storicamente, lo scaling dei database

è sempre stato un punto doloroso per applicazioni di grandi dimensioni, le soluzioni erano spesso limitate e costose.
Mongo DB ha un vasto range di opzioni di scalabilità, questa funzione viene espletata automaticamente da Mongo DB Atlas.

- Buffering delle richieste inviate a mongo: Inizialmente, ogni evento che veniva generato durante la partita veniva catturato e inviato immediatamente al database.
  Questo metodo, sicuramente di facile implementazione, presenta vari punti di criticità dal punto di vista delle performance. Per questo, la soluzione è stata migliorata aggiungendo un buffer di eventi che faccia da tramite tra i dati catturati e le richieste inviate a Mongo DB.

Capitolo 5

Game Design

Come in ogni sistema, prodotto o opera d'arte, è necessaria una fase creativa e di progettazione, in cui gli attori coinvolti nel progetto ideano e progettano quanto si vorrà creare.
In questo caso specifico, gli attori sono i game designer.

5.1 Game Concept

Nel nostro caso, l'ispirazione proviene dal gioco del Dodgeball, e viene modificata da meccaniche provenienti da diversi giochi e diverse piattaforme come:

- **League of legends (PC Gaming)**
- **Blaston (VR Gaming)**
- **Fifa (PC, Console)**
- **Biomutant (PC, Console)**
- **Pokemon Unite (Mobile Gaming, nintendo switch)**
- **Super Smash Bros Brawl (Nintendo Gaming)**
- **WindJammers (NeoGeo, Pc)**

Naturalmente ogni gioco ha contribuito ad ispirare il progetto in maniera differente, ad esempio, da Blaston è stata ripresa l'idea di conferire dei punti vita ai proiettili emessi dalle skill, da Super Smash l'idea di conferire una resistenza ai colpi, ma verrà spiegato tutto nelle sezioni successive.

5.2 Descrizione e Regolamento

In questa sezione saranno presentate le regole di gioco e verranno descritti tutti gli elementi che possiamo trovare nel gioco (personaggi, abilità, ambiente, ecc.).

Possiamo definire il gioco come un *RTS-sportivo*, che richiede ai giocatori capacità meccaniche e strategiche, individuali e di squadra. Il gioco prevede tre modalità di gioco differenti:

- 1vs1
- 2vs2
- 3vs3

è prevista una visuale isometrica come nei classici giochi sportivi o Moba (Fifa, Lol, pokemon unite).

Le due squadre si divideranno sul campo di gioco, come in una partita di dodgeball classica, occupando ognuna la sua metà campo.
L'obiettivo principale è quello di accumulare punti spingendo fuori dal campo i componenti della squadra avversaria.
Ogni giocatore eliminato dalla partita dovrà aspettare un tempo di respawn per tornare in campo. La partita sarà automaticamente vinta se una delle due squadre eliminerà contemporaneamente tutti gli avversari, di fatto, nella modalità 1 vs 1 la partità sarà automaticamente vinta da se si riesce ad eliminare l'avversario.
Per portare a termine gli obiettivi del gioco, ogni giocatore avrà a disposizione delle abilità individuali (attive e passive) e una palla neutra di gioco che potrà essere utilizzata come un attacco base.

5.3 Condizioni di vittoria

Le partite si svolgeranno al meglio dei 3, e i giocatori potranno cambiare il loro set di abilità al termine di ogni match in modo da organizzarsi per un assetto più aggressivo o difensivo a seconda della necessità. Le condizioni di vittoria del singolo match sono le seguenti (dove ogni punto rappresenta la vincita automatica di una delle due squadre):

- Eliminare tutti gli avversari della squadra avversaria prima dello scadere del tempo.
- Allo scadere del tempo, aver totalizzato un numero di punti maggiore della squadra avversaria.

- In caso di pareggio, si giocheranno i tempi supplementari, vincerà la squadra che eliminerà tutti i giocatori avversari (un giocatore eliminato durante i supplementari non può rientrare).

La condizione di vittoria dell'intera partita, naturalmente, è quella di vincere 2 match su 3.

5.4 Sistema delle abilità

Ogni personaggio avrà a disposizione 3 slot abilità e un'abilità passiva unica. I 3 slot potranno essere utilizzati pescando da un pool di abilità che ogni personaggio avrà a disposizione. Ad ogni inizio del match i giocatori avranno a disposizione 60 secondi per sistemare il loro set di abilità e scegliere quelle che secondo loro saranno più adatte alla composizione del team scelta e allo scontro con gli avversari. Ogni abilità avrà un cooldown che varierà con l'aumentare del tempo di gioco (ogni minuto passato il cooldown diminuirà un pochino, in modo da velocizzare le ultime fasi della partita). Il sistema di abilità sarà governato da parametri variabili.

5.4.1 Tipi di abilità

In generale, le abilità si dividono in abilità **permanenti**, che applicano il loro effetto in modo permanente, e **temporanee**, che applicano un effetto per un periodo di tempo limitato. Le abilità possedute dai giocatori potranno dividersi in varie categorie come segue:

- **Skillshot**: Abilità proiettile che segue un path predefinito, sarà caratterizzata da una lunghezza e da una larghezza (il cosiddetto hitbox). In base alla nostra visione di gioco, gli skillshot saranno abilità utilizzate, in maggior parte, per far perdere resistenza agli avversari e cercare di eliminarli (spingendoli fuori dal campo), quindi, skill offensive.

- **AoE**: Abilità ad area che hanno effetto su tutti i giocatori che si trovano nel range dell'abilità (ancora una volta, hitbox) nel momento dell'attivazione. Questo tipo di skill si presta bene ad essere sia una skill di tipo offensivo che difensivo, esempio granata e area di cura rispettivamente.

- **Buff/Debuff**: Abilità (solitamente a target, quindi selezionando direttamente il giocatore a cui applicarla) che fornisce un potenziamento (o un depotenziamento) ad una (o più) delle statistiche di gioco dei personaggi (es: velocità di movimento, scudo, vita, res regen, etc.). Da notare che anche uno skillshot o un AoE possono applicare buff o debuff sul colpo.

- **Environment manipulation**: Abilità che interagiscono con l'ambiente permettendo di manipolare alcuni punti dell'area di gioco. Ad esempio: Creazione di una barriera che non permetta alle skill avversarie di passare o barriera che blocchi gli avversari in specifici punti della mappa.
- **Untargetable ability**: Abilità che rendono l'utilizzatore "immune" da tutte le abilità per un determinato periodo di tempo (Si pensi alla E di Fizz in league of legends).
- **Map-Through**: Abilità che attraversano tutta la mappa di gioco.
- **Attacco base**: Abilità comune a tutti i personaggi, utilizzare una delle palle neutre in giro per la mappa per effettuare degli attacchi base, alcuni personaggi potranno sfruttare la loro abilità passiva per effettuare degli attacchi base speciali (es: attacco a base che rallenta per un secondo).

Le abilità avranno parametri differenti in base alla loro tipo.
Alcuni parametri saranno comuni a tutte le abilità:

- **Range**: Raggio d'azione entro il quale può essere lanciata una skill, può andare da 0 (skill che posso lanciare solo su me stesso) a tutta la mappa.
- **Cooldown**: Tempo di ricarica necessario per un nuovo utilizzo della skill.
- **Velocità di lancio**: il tempo necessario impiegato dall'abilità per essere lanciata.
- **Hitbox**: l'area d'effetto dell'abilità (uguale a 0 se si parla di un'abilità a target).

Le abilità di **AoE**, **Skillshot** e **Map-Through** di natura offensiva avranno anche i seguenti parametri:

- **Punti danno**: il valore grezzo (al quale andrà scalato un valore dipendente dall'armour dell'avversario) del danno da applicare alla resistenza dell'avversario.
- **Potenza di spinta**: valore che determina la forza di spinta all'indietro dell'abilità.

Nel caso in cui si stesse parlando di abilità di Buff (Debuff), i parametri da tenere in considerazione sono:

- **Buff (Debuff) all'armour**: Aumento (diminuzione) di un valore assoluto all'armour di un alleato (nemico) per un determinato periodo di tempo.

- **Buff(Debuff) alla velocità di tiro**: Aumento (diminuzione) di un valore assoluto alla velocità di tiro di un alleato (nemico) per un determinato periodo di tempo.
- **Buff(Debuff) tasso di rigenerazione/s della resistenza**: Aumento (diminuzione) di un valore assoluto al tasso di rigenerazione/s della resistenza di un alleato (nemico) per un determinato periodo di tempo.
- **Buff(Debuff) velocità di movimento**: Aumento (diminuzione) di un valore assoluto alla velocità di movimento di un alleato (nemico) per un determinato periodo di tempo.

Per quanto riguarda le skill di tipo Environment manipulation, possono essere divise in:

- **Offensive**: Avranno un effetto sulla parte di mappa avversaria, con l'intento di limitare i movimenti avversari, il parametro durata determinerà la loro permanenza in campo.
- **Difensive**: Avranno un effetto sulla propria regione di mappa con l'intento di proteggersi dai danni avversari (es: barriera di legno), le abilità di questo tipo avranno il parametro punti vita, indicherà la quantità di danno che possono subire dalle skill avversarie prima di rompersi, e il parametro durata che determinerà la loro rottura dopo un determinato periodo pur non avendo esaurito tutti i loro punti vita.

Una meccanica fondamentale del gioco sarà quella di scontro tra gli skillshot. Questa meccanica è stata ripresa dal gioco in Virtual Reality *Blaston*, in cui è possibile usare ogni arma sia in modo offensivo che difensivo.

Scenario d'esempio in Blaston: Ogni arma genera un tipo diverso di Shot, ogni shot ha, come nel nostro gioco, dei punti danno, se due shot si scontrano si va incontro alle seguenti situazioni:

- *Danno A = Danno B*: gli shot vengono annullati ed entrambi eliminati.
- *Danno A > Danno B (Danno A < Danno B)*: Lo shot B (A) viene eliminato e lo shot A (B) continua la sua cavalcata con punti danno pari a Danno A - Danno B (Danno B - Danno A).

Da notare che, sia le palle neutre, che gli skillshot, saranno soggetti a questa logica, con l'unica differenza che: se la palla neutra avrà meno punti danno dello skillshot questa sparirà e respawnerà nel campo opposto del tiratore.

A livello tecnico il sistema delle abilità sarà sviluppato sfruttando il plugin *Gameplay Ability System* messo a disposizione da Unreal.

5.4.2 Palla neutra

La palla neutra è un oggetto che si muoverà con costanza all'interno del campo di gioco. Ce ne saranno in totale quattro, per tutta la durata della partita. La palla neutra può essere raccolta dai partecipanti alla partita, e utilizzata (come skillshot) contro gli avversari. La palla neutra è gestita passando attraverso tre stati diversi, ogni stato ha un tempo massimo di permanenza, allo scadere del tempo la palla spawnerà automaticamente nel lato di gioco avversario:

- **Spawned**: in questo stato la palla è appena spawnata nel campo di gioco, i giocatori avranno a disposizione 5 secondi per poterla raccogliere ed utilizzarla contro gli avversari.
- **Picked up**: questo è lo stato che intercorre quando un giocatore raccoglie la palla, è anche lo stato in cui la palla può essere effettivamente lanciata, dura 5 secondi.
- **Throwed**: lo stato di lancio, in cui la palla cade dopo esser lanciata, in questo stato la palla sarà un vero e proprio skillshot e tutte le logiche che si applicano agli skillshot vengono applicate anche alla palla, dura 3 secondi.

5.4.3 Parametri di gioco

Ogni personaggio avrà dei parametri di gioco con valori di base fissi, differenti tra i vari personaggi, che potranno essere modificati solo dal sistema di abilità (Buff o Debuff).

- **Armour**: parametro difensivo, indica quanto un determinato personaggio è in grado di sopportare gli attacchi avversari, e di conseguenza quanto questi avranno influenza sul parametro resistenza.
- **Resistenza**: il parametro resistenza è l'equivalente dei punti vita dei personaggi, come mai non si chiamano punti vita? A differenza della barra della vita, se la barra della resistenza scende a 0 il personaggio non morirà ma entrerà nello stato di Iper-Stun (Vedi sezione Stati di gioco).
- **Tasso di rigenerazione/s della resistenza**: Parametro che determina il tasso di rigenerazione della resistenza ogni secondo.
- **Velocità di movimento**: Determina quanto velocemente (di base) un personaggio può muoversi per la mappa.
- **Potenza di tiro**: parametro che veicola la quantità di spinta che gli attacchi possiedono. In particolare, una potenza di tiro molto alta farà più danni al

nemico e lo spingerà molto più indietro, mentre una potenza di tiro bassa il contrario.

- **Velocità di tiro**: parametro che gestisce la velocità con cui viene percorsa la traiettoria della palla, non ha influenza su danni o su spinta ma influisce sulla capacità di schivata o meno per gli avversari.

5.4.4 Stati di gioco

Gli stati di gioco sono gli status in cui si può trovare un determinato personaggio durante la partita. Sono stati previsti i seguenti stati:

- **Normale**: nessun deficit o benefit.

- **Root**: Alcune abilità avranno la possibilità di fermare l'avversario in un punto specifico (pur mantenendo la possibilità di utilizzare skill/attacchi base, ma non skill di movimento come il dash), la durata dipenderà dalle skill avversarie.

- **Stun**: Stato in cui il giocatore non può ne muoversi, ne lanciare abilità, ha durata variabile a seconda dell'abilità che lo innesca.

- **Iper-Stun**: Stato in cui si trova un personaggio che ha perso tutta la sua resistenza, ha lo stesso effetto dello *stun* con la differenza che le abilità che lo colpiscono hanno potenza di spinta doppia, ha una durata limitata ed uguale per tutti.

- **Avvelenamento**: In questo stato il personaggio perde punti resistenza periodicamente, ha una scadenza variabile a seconda dell'effetto che l'ha scaturito.

5.5 Game Flow

Il game flowchart è formato dalle sezioni di gioco esplorabili interconnesse tra loro seguendo un determinato ordine. Il diagramma di flusso che segue raffigura tutti i percorsi di gioco formanti l'applicativo, rispondendo alla domanda "Come mi muovo tra le schermate?". Ogni schermata avrà una struttura a sé stante.

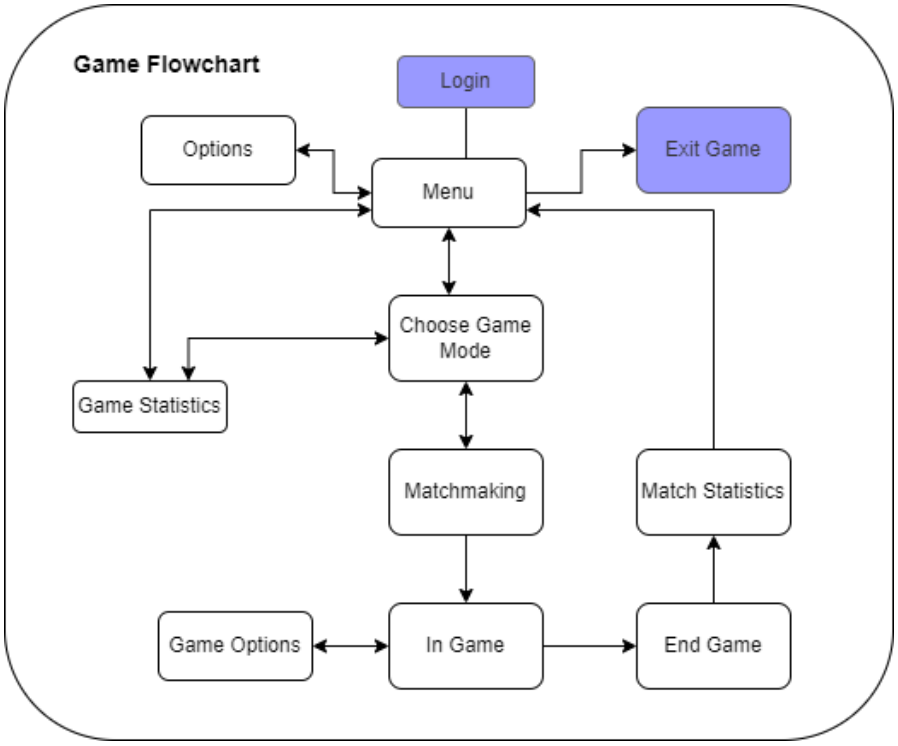

Figura 5.1: Game flow schema

Scendiamo brevemente nel dettaglio delle schermate:

- **Game Statistics**: In questa sezione è possibile visionare lo storico di alcuni parametri di gioco accumulati dal giocatore durante le partite. Verranno presentati dati come:
  - Partite perse/vinte
  - Personaggi giocati di più
  - Punteggio in classifica
  - Skills + utilizzate
- **Menu**: Nel menu principale sarà l'hub centrale del gioco in cui si avrà la possibilità di accedere a varie schermate (vedi figura), il nodo centrale di collegamento.
- **Choose game mode**: Schermata di scelta della modalità di gioco, 3 possibili modalità sono:
  - 3 vs 3
  - 2 vs 2

  - 1 vs 1

- **Matchmaking**: Schermata di ricerca della partita.
- **In Game**: Schermata di partita, in cui il giocatore entrerà effettivamente nello stato di game flow.
- **End Game**: Schermata di fine partita (su lol usata per onorare i compagni di squadra).
- **Match Statistics**: Presentazione delle statistiche specifiche per quella partita.
- **Options**: Schermata delle opzioni pe modificare parametri di configurazione come grafica/suono ecc.
- **Game Options**: Lo stesso, solo per il menu in game.

Capitolo 6

Unreal Engine: Gameplay Ability System

un prodotto videoludico multigiocatore richiede uno sviluppo accurato di un sistema di abilità complesso, capace di rispettare i principi di networking, necessari al corretto funzionamento delle logiche di replicazione.
In particolare, in un videogioco, con meccaniche MOBA come BugBall, è fondamentale avere a disposizione un robusto meccanismo di gestione delle abilità, seguendo queste necessità, la scelta delle tecnologie da utilizzare per lo sviluppo del prototipo è stata direzionata verso **Unreal Engine 4**.
Unreal engine è un motore di gioco scritto in C++, di fatto, è anche il linguaggio con cui si sviluppano le logiche di funzionamento del gioco con questa tecnologia. Supporta un grandissimo range di piattaforme desktop, mobile, console e di realtà virtuale. Inoltre, il sistema è open source, e disponibile su *GitHub*.
Epic Games, l'azienda produttrice del motore, rinuncia alle royalties fino ad un fatturato di un milione di dollari, lasciando spazio di pubblicazione a studi di sviluppo più piccoli. Attualmente è il miglior motore di gioco con una gestione del multiplayer nativa.
Sfruttando il plugin *Gameplay ability System*, abbreviato in *GAS*, nativo della tecnologia, è stato possibile implementare un sistema di abilità molto potente con logiche già ben definite da Epic Games, la casa produttrice del motore.
Infine, Unreal offre un'interfaccia di sviluppo di alcune funzionalità di gameplay, e non, chiamata *Blueprint*, una sorta di programmazione visuale che, sfruttando dei nodi, permette di implementare logiche di gioco. In questo modo, risulta molto rapido prototipare, anche per figure non tecniche, come designer o artisti.

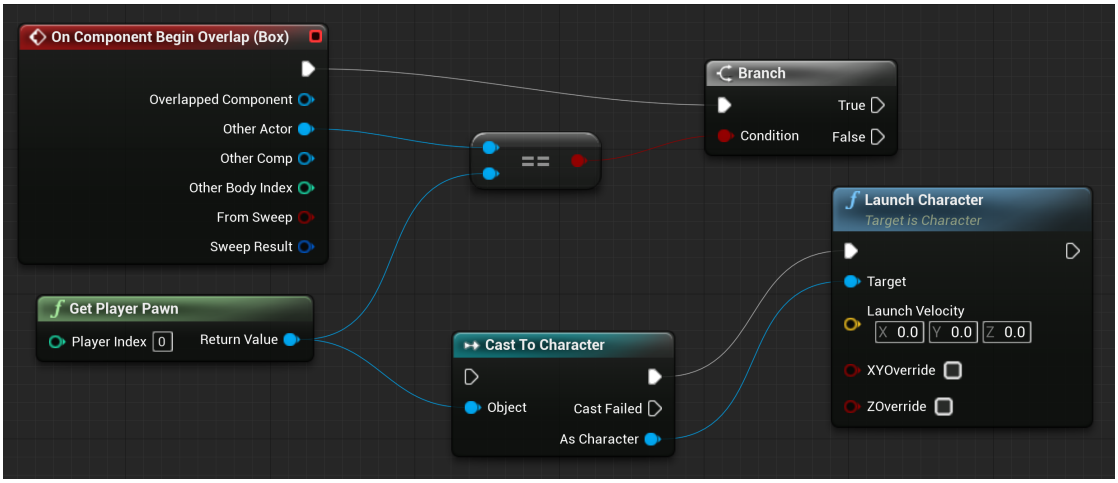

Figura 6.1: Un esempio di utilizzo di programmazione blueprint

Il contro è che, rispetto ad un motore come *Unity*, Unreal ha una barriera d'ingresso piuttosto alta, seppur la curva d'apprendimento sia molto ripida, sono stati necessari circa 30/40 giorni lavorativi per iniziare uno sviluppo del gioco vero e proprio.

6.1 Gameplay ability system

Il gameplay ability system è un framework utilizzato per sviluppare abilità e attributi di gioco, maggiormente per titoli MOBA o RPG.
Permette di sviluppare abilità, attive o passive, da associare a qualunque personaggio si voglia, purchè rispetti i paradigmi del GAS (*Gameplay Ability System*), ha un supporto per i cambiamenti di stato, ed è possibile associare una serie di effetti ad ogni abilità. Possono essere associati degli effetti audio-visivi coordinatamente all'attivazione dell'abilità.
In sintesi, questo sistema aiuta nel processo di progettazione e implementazione di tutto il sistema di abilità.

Per una comprensione più approfondita delle parti successive, sarà necessario presentare alcuni concetti fondamentali del gameplay ability system, il lettore già avvezzo a questo framework può saltare questa parte e dirigersi alla sezione **6.2**. Inoltre, questo elaborato da per scontato alcuni concetti fondamentali del funzionamento di unreal engine, in ogni caso, il lettore potrà far affidamento alla bibliografia per eventuali chiarimenti.

6.1.1 Ability system component

L'*Ability system component* è il cuore del Gameplay ability system. è un *UActorComponent* che gestisce tutte le interazioni con il sistema. Ogni attore che voglia utilizzare un'abilità, avere attributi, o ricevere *GameplayEffects* con questo sistema, deve possedere questo componente. Questi oggetti vivono tutti all'interno del, sono gestiti e replicati dal, Ability system component.

L'attore con l'ability system component associato è chiamato *OwnerActor* dell'Ability system component. la rappresentazione fisica dell'attore è chiamata *AvatarActor*.
L'owner e l'avatar possono essere gli stessi nel caso di semplici AI o minion. Risultano essere differenti nel caso di personaggi controllati dagli utenti come in un MOBA, in questo caso l'OwnerActor è il *PlayerState* e l'AvatarActor è la classe del personaggio in gioco (questo serve a dividere la logica dallo stato).
Si pensi ad una struttura MVC, il model è il PlayerState, o OwnerActor, il Controller è l'AvatarActor, e il view sono le animazioni/VFX/UI utilizzate per visualizzare il tutto.

6.1.2 Gameplay Tags

Gli *FGameplayTags* è un sistema di tag gerarchici nella forma *Parent.Child.Grandchild* che sono registrati con il *GameplayTagManager*. Questi tag sono molti utili per scopi di classificazione e descrizione dello stato di un oggetto. Per esempio, se un personaggio è stordito, possiamo associargli un tag che rappresenti il suo stato per tutta la durata dello stun, questo, a livello di gestione degli stati e delle meccaniche di gioco ha un grande valore.

Fornire dei tag ad un oggetto, significa associarli ad un ability system component, in modo che questi possano essere letti dal Gameplay Ability system, il che implica che tutti gli attori che possiedono un ability system component possono leggere quel tag. In particolare, L'*UAbilitySystemComponent* implementa l'*IGameplayTagAssetInterface* che fornisce le funzioni necessarie alla manipolazione dei Gameplay tags.

6.1.3 Attributes

Gli attributi sono valori di tipo *float* definiti dalla struttura *FGameplayAttributeData*. Possono rappresentare qualsiasi cosa misurabile, dalla salute del personaggio, al suo livello, al numero di pozioni rimaste nell'inventario. Ogni valore numerico associato ad un attore che faccia parte del gameplay probabilmente sarà un attributo. Vista la natura inter-connessa dell'ability system component, è buona pratica che

gli attributi vengano modificati esclusivamente dai GameplayEffects. Gli attributi sono definiti e vivono nell'*AttributeSet*.

AttributeSet

L'*AttributeSet* definisce, mantiene e gestisce i cambi di attributi. è buona pratica estendere la classe *UAttributeSet*.

6.1.4 Gameplay Effects

I *GameplayEffects (GE)* sono i mezzi attraverso il quale le abilità cambiano il valore degli attributi e rimuovono/aggiungo GameplayTags. Possono causare cambiamenti immediati di attributi, come danni o cure, oppure possono applicare effetti a lungo termine e duraturi come *buff* o *debuff* come un potenziamento alla velocità di movimento o uno stordimento. La classe *UGameplayEffect* è stata concepita per essere una classe di soli dati, che definisce un singolo effetto di gioco. Rispecchiando al meglio le meccaniche di gioco, un GameplayEffect non detiene nessun tipo di logica, ma solo l'effetto stesso, hanno tre tipi di durata:

- **Instant**: applicazione istantanea se le condizioni di applicazione vengono soddisfatte.
- **Duration**: applicazione a durata variabile, ma finita.
- **Infinite**: effetto permanente.

I tipi duration e infinite permettono di applicare effetti periodici. I GameplayEffects cambiano gli attributi attraverso i *modificatori* e le *esecuzioni*, costrutti utilizzati per gestire la replicazione dei cambi di valori degli attributi in modo coerente.

6.1.5 Gameplay Abilities

Le *GameplayAbilities (GA)* sono ogni azione o abilità che un attore può fare nel gioco. Possono essere attivate più gameplay ability in contemporanea, ad esempio, lo scatto più utilizzo di un'abilità, il sistema supporta sia l'implementazione via blueprint, sia l'implementazione tramite C++.

Qualche esempio di abilità che si potrebbe implementare con questo sistema:

- **Sprinting**
- **Sparare con una pistola**
- **Usare una pozione**

- **Aprire una porta**
- **Collezionare una risorsa**

Tutte le *GameplayAbilities* avranno la loro funzione *ActivateAbility()* sovrascritta con logica custom. Può essere aggiunta ulteriore logica sfruttando la funzione *EndAbility()* che viene eseguita quando la Gameplay Ability viene completata o cancellata.

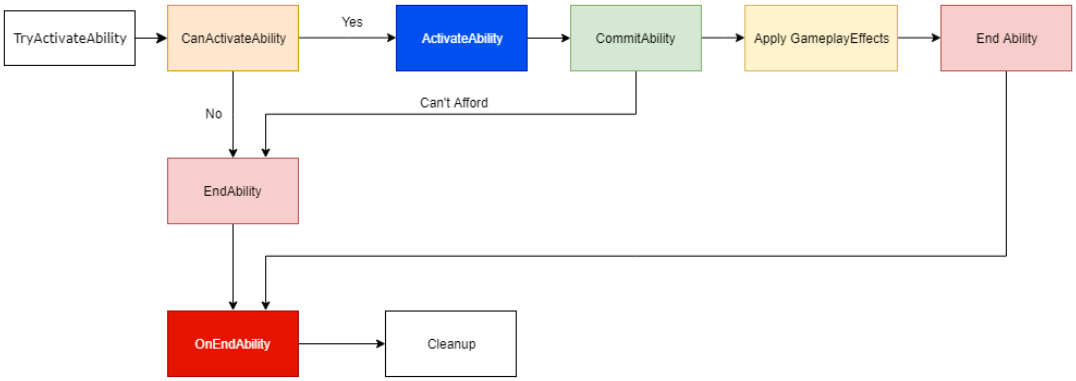

Figura 6.2: Flusso di esecuzione di una semplice Gameplay Ability

Le abilità più complesse possono essere implementate usando più Gameplay Ability che interagiscono tra loro (attivazione, cancellamento, etc.) con le altre.

6.1.6 Ability Task

Le GameplayAbilities vengono eseguite in un singolo frame. Questo limite non garantisce nessun tipo di flessibilità, per effettuare azioni che vengono eseguite al passare del tempo o che devono essere eseguite in un secondo momento vengono utilizzati gli *AbilityTasks*. La classe in questione è chiamata *UAbilityTask*, nel suo costruttore è inserito un limite massimo di ability task che possono essere eseguiti contemporaneamente, ovvero, mille.

6.1.7 Gameplay Cues

I *GameplayCues (GC)* eseguono cose non strettamente legate al gameplay, come:

- Effetti sonori.
- Effetti particellari.
- etc.

6.2 Le abilità di bugball

Inizialmente lo sviluppo ha incontrato delle difficoltà legate alla scarsissima documentazione presente in rete.
Come suggerito da Epic Games stessa, il miglior modo per utilizzare il GAS è quello di studiare progetti esistenti, dato che è uno strumento molto vasto, ogni sviluppatore può utilizzarlo in modo leggermente diverso. è stato fondamentale per noi seguire la documentazione scritta da *tranek*, un utente diventato noto tra gli appassionati per la completezza, seppur non ufficiali, dei suoi appunti.
Prima di scendere nel dettaglio delle abilità implementate, facciamo un zoom-in del flusso di esecuzione dell'abilità.

6.2.1 Flusso di esecuzione di un'abilità

Il flusso è innescato da un input dell'utente, che viene catturato dall'Ability system component dell'oggetto rappresentante il personaggio all'interno di Unreal.
Dopo aver individuato l'abilità corrispondente all'input viene effettuato un controllo sui tag posseduti dal giocatore al momento dell'innesco, ad esempio il sistema di assicura che il giocatore non possieda un debuff sulle abilità o se l'abilità eseguita non possieda il tag *isInCooldown*, se i requisiti non vengono soddisfatti l'attivazione dell'abilità viene cancellata, altrimenti si prosegue con il flusso.
A questo punto è necessario un altro controllo, quello delle risorse di gioco, se il giocatore ne ha abbastanza, il processo continua, altrimenti viene cancellata l'attivazione.
Ad esempio, se un'abilità per essere lanciata richiede il consumo di un determinato quantitativo di punti abilità, è necessario controllare che il lanciatore ne possieda abbastanza prima del lancio. Una volta che tutti i controlli sono andati a buon fine, l'ability system component applica una serie di Gameplay Effect al lanciatore dell'abilità, che hanno lo scopo di consumare le risorse e/o applicare i tag di lancio.
I gameplay effect di questo tipo vengono utilizzati per vari scopi, quello principale è l'innesco del cooldown dell'abilità, fin tanto che il giocatore lanciante avrà quel determinato *GE*, non sarà in grando di lanciare di nuovo quell'abilità.
Ma non solo, alcune abilità potrebbero attivare dei *deficit* al giocatore, o un potenziamento, e così via.
Il prossimo step è l'esecuzione della logica dell'abilità, che concerne tutta l'esecuzione di animazioni, effetti sonori, generazione di oggetti all'interno del gioco e/o controlli sul target dell'abilità (nemico o amico). Infine, vengono applicati i gameplay effect e i cambi dei valori degli attributi a tutti gli attori, possedenti l'Ability system component, coinvolti nel flusso di esecuzione, possono essere giocatori, compagni o avversari o altri oggetti di gioco.

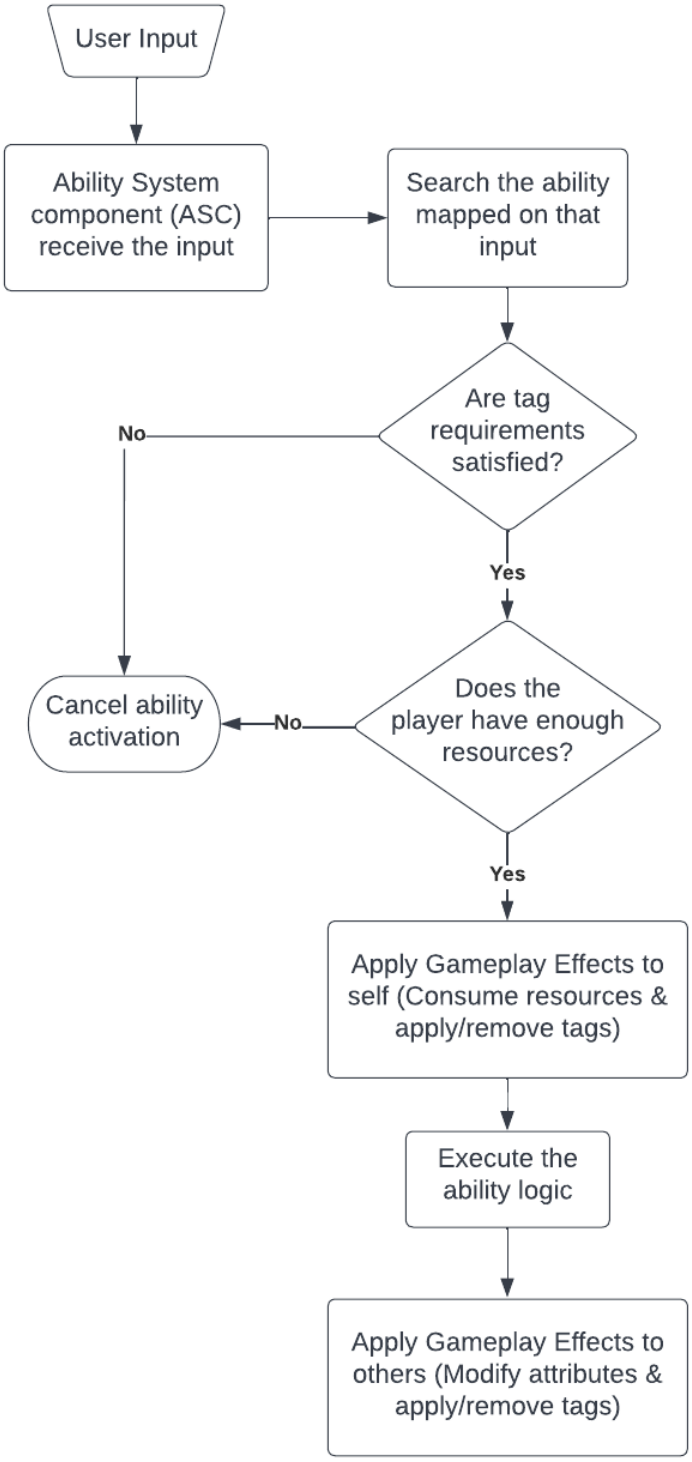

Figura 6.3: Flusso concettuale di esecuzion di una semplice Gameplay Ability

All'interno del prodotto videoludico, sono state implementate varie abilità:

- **Tiro di base**
- **Skillshot**
- **Scatto in avanti**
- **Granata**
- **Cura di squadra**
- **Scudo**
- **Potenziamento alla velocità**

In questa sede non si scenderà nei dettagli implementativi di tutte le abilità, si esploderà solo la logica implementativa dello skillshot e dello scatto in avanti.

6.2.2 Skillshot

Nel nostro caso, il flusso seguito dal lancio di un'abilità è leggermente diverso, se ne discuteranno solo le differenze. Non abbiamo previsto la possibilità di attivare contemporaneamente più abilità, sarà necessario attendere che un'abilità finisca il suo flusso d'esecuzione per lasciare spazio alla prossima. Questo viene gestito tramite un GameplayTag speciale chiamato *isCastingAnAbility*. Per rendere il gioco più veloce e dinamico, abbiamo scelto di non utilizzare un sistema di punti magia o punti abilità, quindi ad ogni lancio l'unico limite è il cooldown delle abilità. In questo modo viene eliminato il controllo delle risorse da parte dell'Ability System Component. L'esecuzione della logica dell'abilità è diversa per ogni abilità, in questo caso specifico, l'abilità si occupa di spawnare un proiettile, rendendolo il veicolo con cui espletare le volontà dell'abilità. In particolare, anche il proiettile avrà tra i suoi componenti L'ability system component, il che lo rende a tutti gli effetti soggetto ai cambi di tag, al consumo di risorse, etc. Se il proiettile colpisce un avversario lungo la sua traiettoria possono essere applicati vari effetti. Lo skillshot di base, da cui estendono tutti gli altri, si limita ad applicare un *GE* istantaneo di danneggiamento, che consuma la barra resistenza del giocatore colpito. Un altra conseguenza dell'essere colpiti dallo skillshot è quella di essere spinti all'indietro, questa meccanica non è stata gestita con l'ability system component, poichè non prevede un sistema di spostamenti fisici. Di fatto, il GAS si limita esclusivamente alla parte numerica/logica.

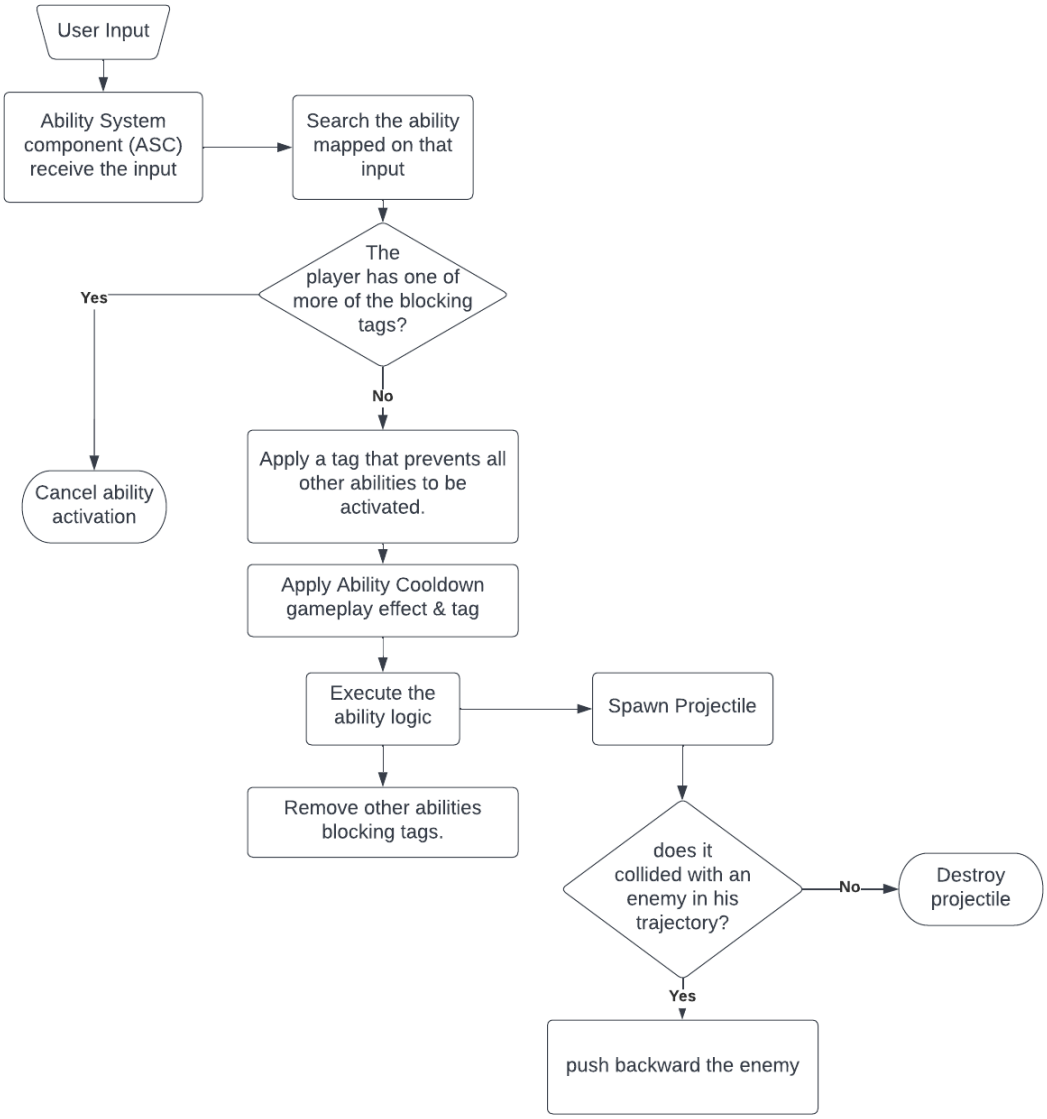

Figura 6.4: Flusso concettuale di esecuzion dello skillshot

Scendendo nei dettagli implementativi vediamo come si traduce quanto appena descritto in termini programmatici. Prima di tutto, è stato necessario definire delle logiche di sviluppo chiare e condivise. In particolare, la fase di sviluppo di un'abilità passa attraverso due step:

- **Sviluppo in C++**: sfruttando le logiche di sviluppo messe a disposizione dalle macro di unreal engine, nel C++ sono stati definiti alcuni attributi fondamentali per il funzionamento dello skillshot.

- **Estensione in Blueprint**: Una volta definite le proprietà di base in C++, sono state definite in blueprint alcune logiche di esecuzione dell'abilità. In questo caso, contenute all'interno della classe Projectile.

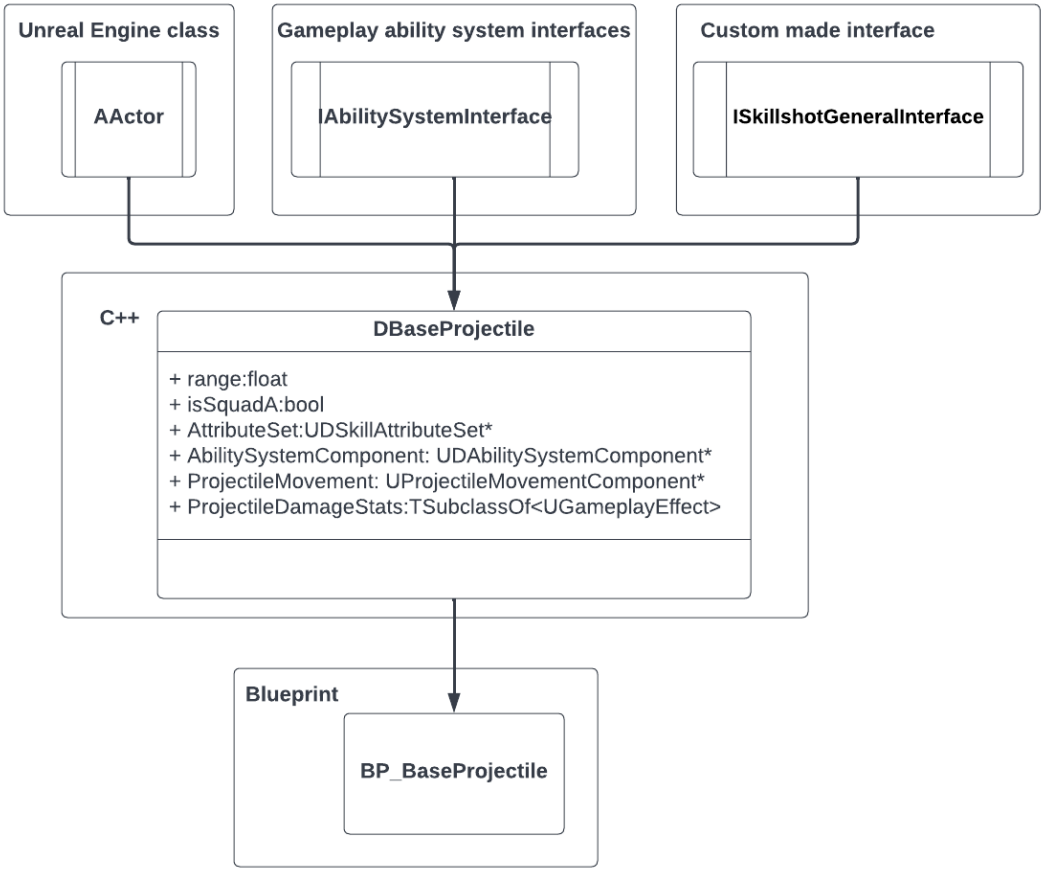

Figura 6.5: Struttura delle classi coinvolte nello skillshot

Partendo dalla parte di programmazione pura, analizziamo la classe *DBase-Projectile*, che è la classe di base, da cui estendono tutti i proiettili, a sua volta questa classe estende da alcune classi. La prima da menzionare è sicuramente la classe **AActor**, un attore è un oggetto che può essere usato all'interno di un livello, come le camere, le mesh statiche o il punto di partenza di un giocatore. Gli attori supportano le classiche trasformazioni 3D come traslazione, rotazione o scaling. Le logiche di comportamento che contraddistinguono gli attori possono essere implementate sia in blueprint che in C++. Inoltre DBaseProjectile deve necessariamente implementare alcune interfacce del GAS. L'interfaccia **IAbilitySystemInterface**

ha una funzione che deve essere sovrascritta: **UAbilitySystemComponent* GetAbilitySystemComponent() const**, ritorna un puntatore all'Ability system component posseduto dall'attore. I vari Ability system component interagiscono tra di loro internamente sfruttano l'implementazione di questa interfaccia. Infine la classe DBaseProjectile possiede alcuni attributi specifici che le garantiscono un comportamento:

- **Range**: definisce la distanza massima di percorrenza del proiettile, oltre la quale questo cessa di esistere.
- **IsSquadA**: flag di comodo per identificare la squadra di appartenenza del lanciatore dell'abilità.
- **AttributeSet**: mantiene tutti gli attributi che definiscono le caratteristiche del proiettile (popolati nel blueprint)
- **AbilitySystemComponent**: Componente che tutti gli attori che fanno parte del sistema di abilità devono utilizzare.
- **ProjectileMovement**: gestisce la logica di movimento del proiettile all'interno dell'ecosistema unreal.
- **ProjectileDamageStats**: Contenitore del gameplay effect che definisce il comportamento del proiettile alla collisione con il nemico.

DBaseProjectile viene poi estesa da BPBaseProjectile, la classe blueprint che implementa la logica di funzionamento vera e propria per questo tipo di proiettile specifico. In questo modo, il workflow seguito per implementare qualsiasi tipo di proiettile è immediato e altamente customizzabile. Al blueprint sono associati tre gameplay effect diversi:

- **BPSkillshotCooldownGE**: Gameplay effect che sancisce il tempo di recupero di una skill attraverso l'attributo duration. Un gameplay effect che gestisce un cooldown deve avere una duration policy di tipo *HasDuration*.
- **BPBasicSkillshotStatsGE**: Le stats, non sono altro che i valori che popolano gli attributi di gameplay del proiettile, in questo caso la potenza di *KnockBack*, quanto sarà forte la spinta che il proiettile applicherà all'avversario in caso di collision, e *Damage*, quanti punti resistenza il proiettile toglierà all'avversario. PEr gameplay effect di questo tipo si applica una duration policy di tipo *Instant*.
- **BPSkillshotDamageGE**: Si occupa di gestire effetti aggiuntivi alla collisione con l'avversario, da notare che il danno base è inserito nelle stats. Di fatto, in questo gameplay effect troviamo un effetto di rallentamento, associato a sua

volta ad un altro blueprint chiamato *BPSkillShotSlowDownGE*, che mantiene delle informazioni sull'entità dello slow down e della durata.

La gestione dei gameplay effect in questo modo ne permette un ampio riutilizzo. Inoltre, la modularità che se ne guadagna permette di combinare la creazione di abilità differenti con uno sforzo contenuto. Ad esempio, il gameplay effect che produce un rallentamento può essere riutilizzato anche da altre abilità non necessariamente di tipo skillshot.

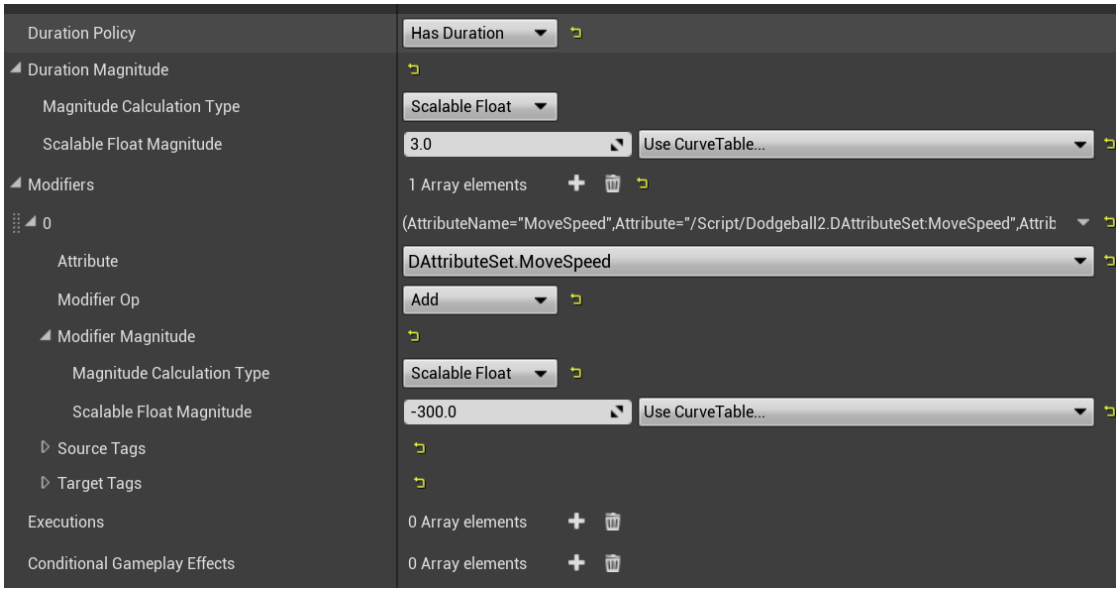

Figura 6.6: Parametri di configurazione della classe blueprint del cooldown

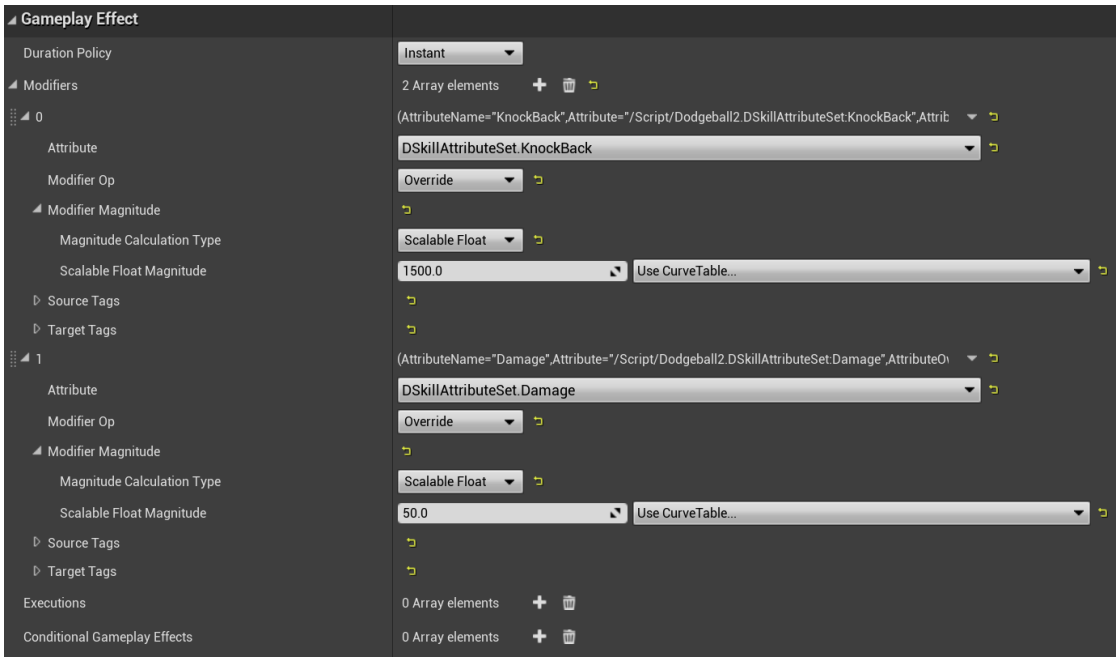

Figura 6.7: Parametri di configurazione della classe blueprint delle statistiche dello skillshot

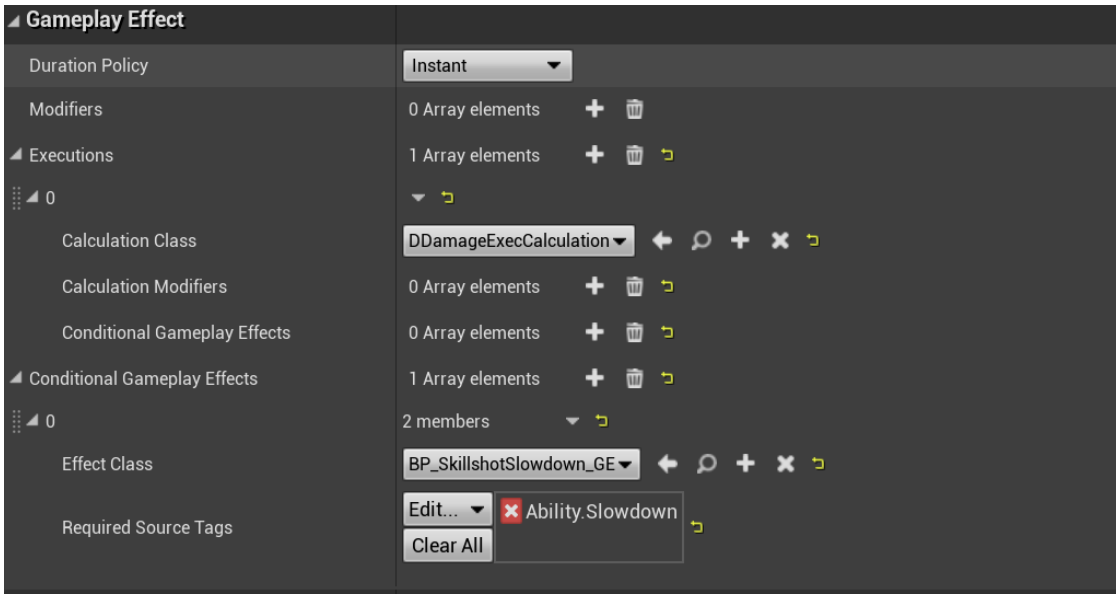

Figura 6.8: Parametri di configurazione della classe blueprint degli effetti aggiuntivi

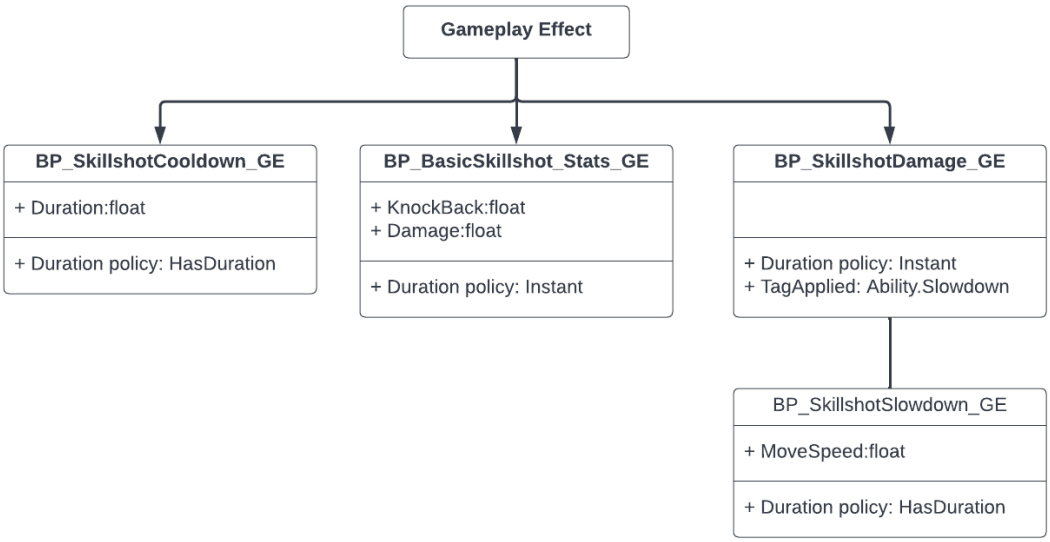

Figura 6.9: Struttura dei gameplay effect coinvolti nello skillshot

Scendiamo più nei dettagli implementativi del blueprint che compone il proiettile. Al momento della creazione dell'oggetto, il proiettile verrà associato al **BPBasicSkillshotStatsGE**.

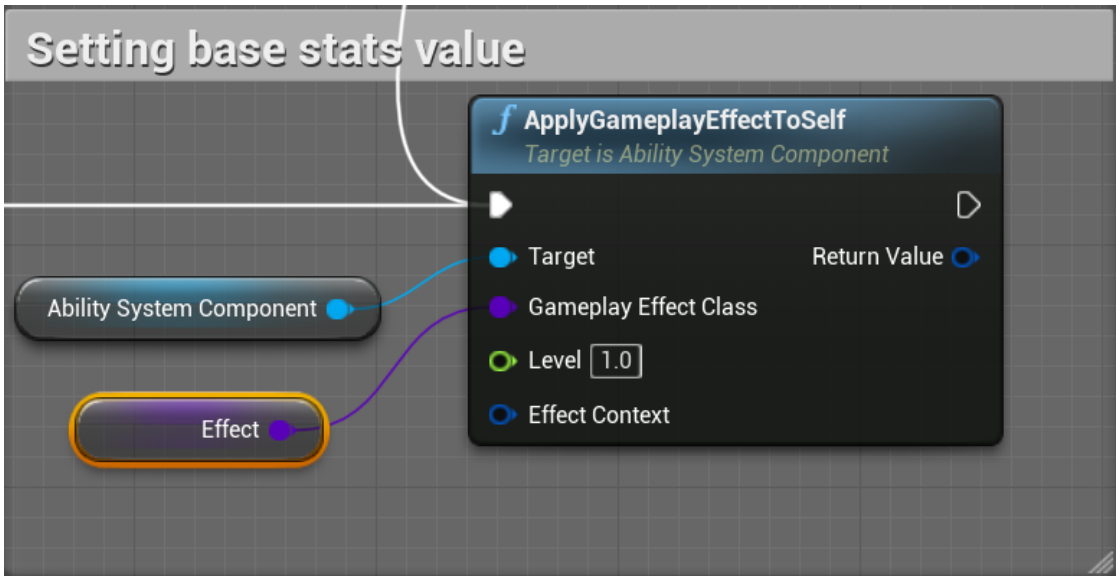

Figura 6.10: Nodi blueprint che configurano il gameplay effect relativo alle statistiche di gioco

Quando il proiettile riceve un evento di collisione, si avvia una procedura di controllo per determinare la natura dell'oggetto con cui si è colliso.

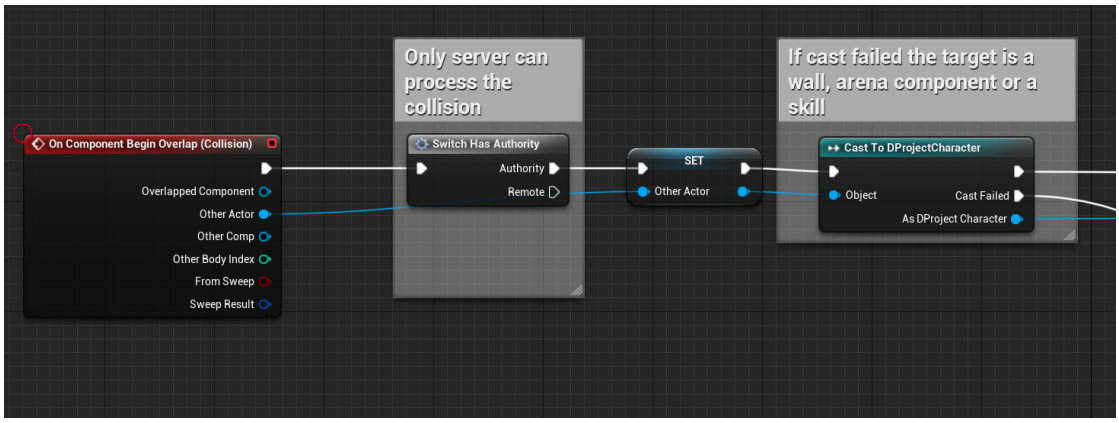

Figura 6.11: Controllo sulla collisione

Una volta essersi assicurati di aver colliso con un giocatore, occorre verificarne la squadra di appartenenza, nel caso in cui si collida con un compagno di squadra, la logica di esecuzione si interrompe e il proiettile continuerà il suo percorso.

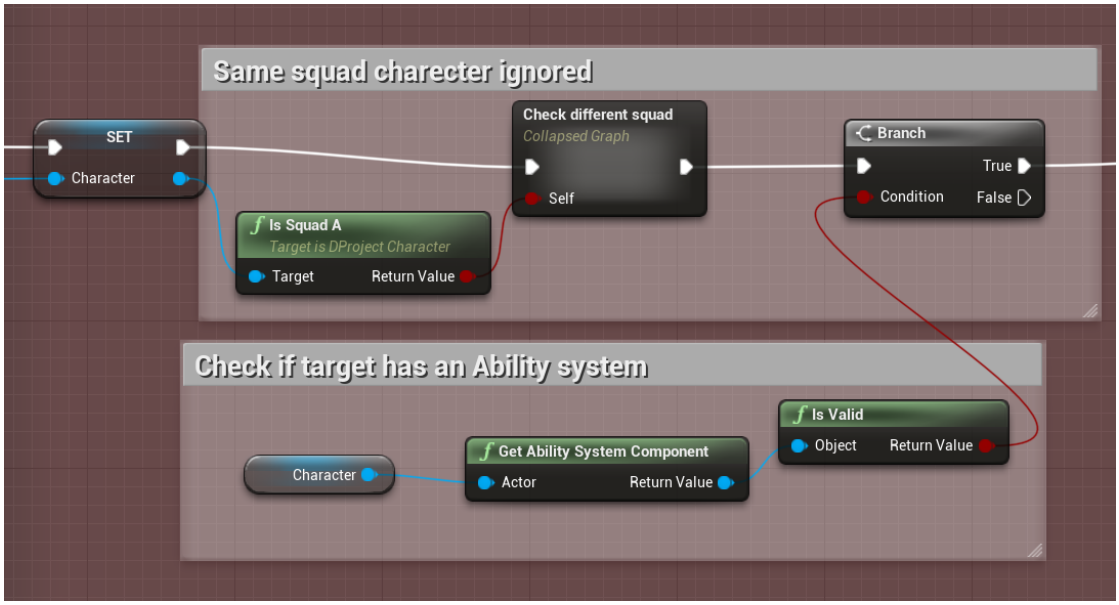

Figura 6.12: Controllo squadra del giocatore colliso

Se il giocatore colpito è della squadra avversaria si può proseguire con l'applicazione degli effetti di gameplay.

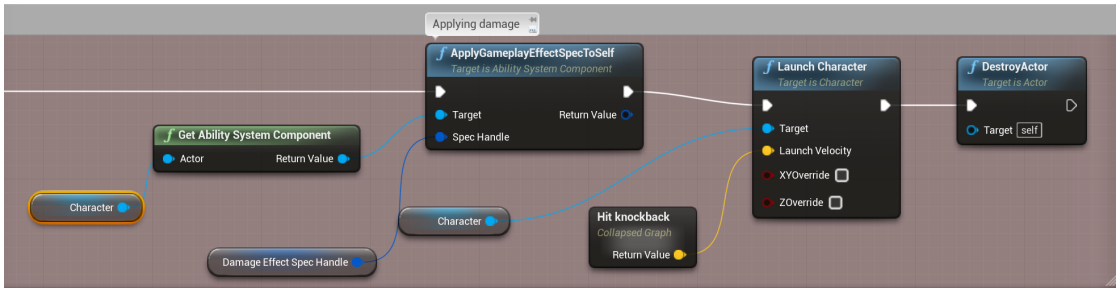

Figura 6.13: Applicazione della logica in seguito alla collisione

Viene catturato l'Ability system component del giocatore colpito, e passato al nodo *ApplyGameplayEffectToSelf*, funzione del GAS che si occupa di applicare un effetto ad uno specifico ability system component: il *Damage Effect Spec Handle* che gestisce l'applicazione dei danni.
Inoltre, viene chiamata in causa la funzione *Launch Character*, che applica una forza che spinge un attore in una determinata direzione.
L'attributo *LaunchVelocity* è la velocità con cui questa forza spingerà l'attore, è stata calcolata sfruttando la funzione *Hit KnockBack*, che ottiene il valore necessario sfruttando la velocità del proiettile (contenuta nell'attributo *ProjectileMovement*) e dal valore KnockBack dell'attribute set del proiettile.

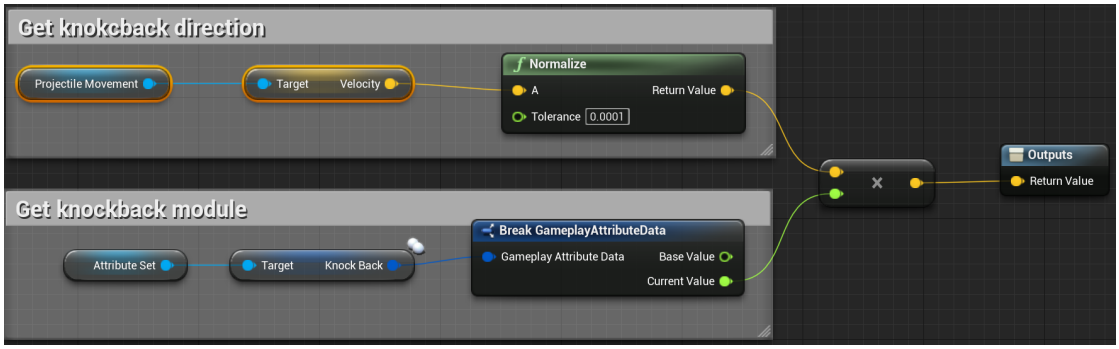

Figura 6.14: Calcolo della potenza di knockback

Infine, viene distrutto l'attore che veicola questi effetti, ovvero, il proiettile.
Nel caso in cui la collisione avvenga con un attore che non sia un giocatore, il blueprint seguirà una strada differente. Ad esempio, se l'attore colpito è un altro skillshot, verrà effettuato un calcolo sui punti vita dei due proiettili, e il più resistente sopravviverà allo scontro con potenza ridotta.

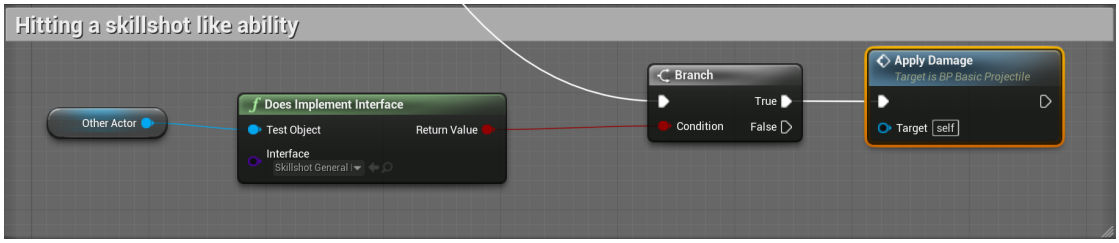

Figura 6.15: Collisione con un altro poiettile

6.2.3 Scatto in avanti

Un'altra abilità interessante da analizzare è lo scatto in avanti, che può essere usato sia in termini difensivi che offensivi.
In particolare, tutto il flusso di esecuzione rimane più o meno lo stesso, l'unica differenza è la logica di esecuzione.
In questo caso, non avremo più il coinvolgimento di altri giocatori o altri skillshot, quest'abilità si limita a spostare molto velocemente il giocatore in una determinata direzione.
Rifacendoci al grafico in **Figura 6.4**, l'unica differenza è che l'esecuzione della logica si limita a spostare il lanciatore.

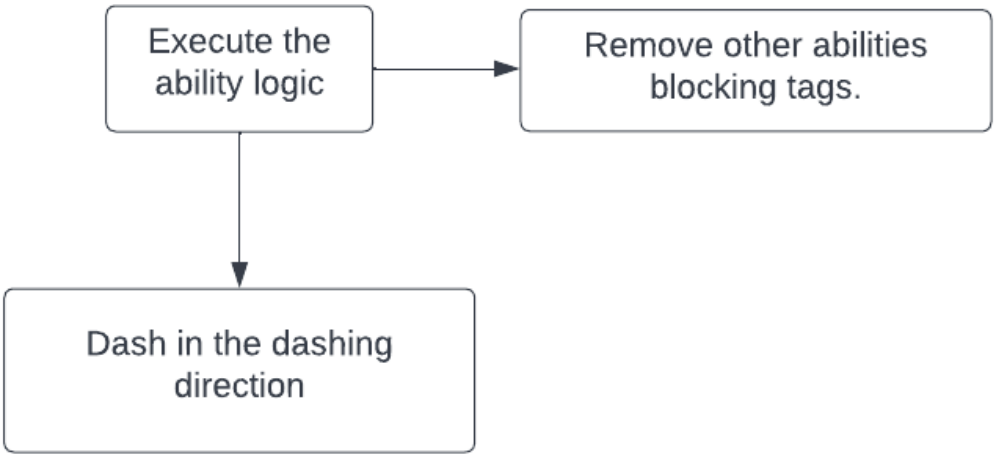

Figura 6.16: Differenza logica di esecuzione dell'abilità di dash

L'implementazione dell'abilità di dash è stata sviluppata interamente in blueprint.

Come si può notare dalla figura successiva, una volta attivata l'abilità, viene eseguita una logica di trasformazione al lanciatore per permettere al dash di essere eseguito nella giusta direzione.
In unreal, il movimento dei giocatori è gestito dalla classe *PlayerController*, che implementa le funzionalità di rispondere agli input degli utenti restituendo azioni concrete, come il movimento, l'uso di oggetti, ecc.
Una volta ottenuto il PlayerController del lanciatore, viene definita la direzione di rotazione tramite il nodo *GetCursorDirection*, da cui viene ritornato un vettore.
Il vettore viene dato come input alla funzione *RotationFromXVector*, da cui si ottiene un *FRotator*, un contenitore per le informazioni di rotazioni, in cui tutti i valori rotazionali sono memorizzati in gradi. Infine, queste informazioni vengono passate alla funzione *SetActorRotation* che ruota l'attore nella direzione desiderata istantaneamente.

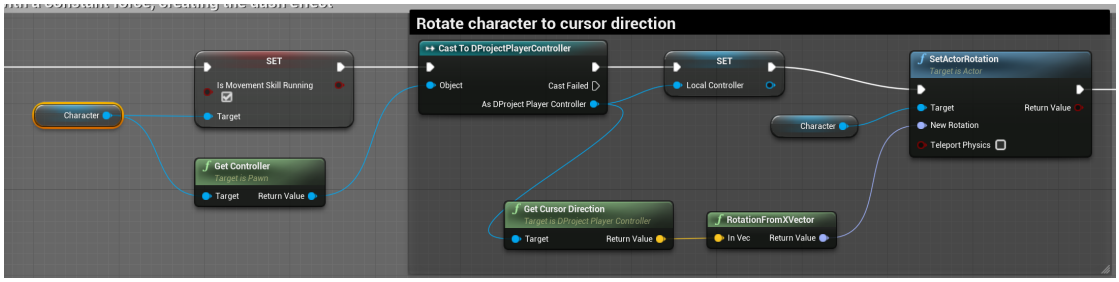

Figura 6.17: Logica di rotazione del dash

Prima di eseguire lo scatto vero e proprio, è necessario disabilitare il percorso che il player controller avrebbe dovuto perseguire, altrimenti, si rischia di ottenere una posizione finale non consistenze, sopratutto in ottica di networking.
In particolare, *Disable Path* e *Reset Path* gestiscono i problemi dovuti alla propagazione delle informazioni di percorrimento attraverso i vari partecipanti alla partita.

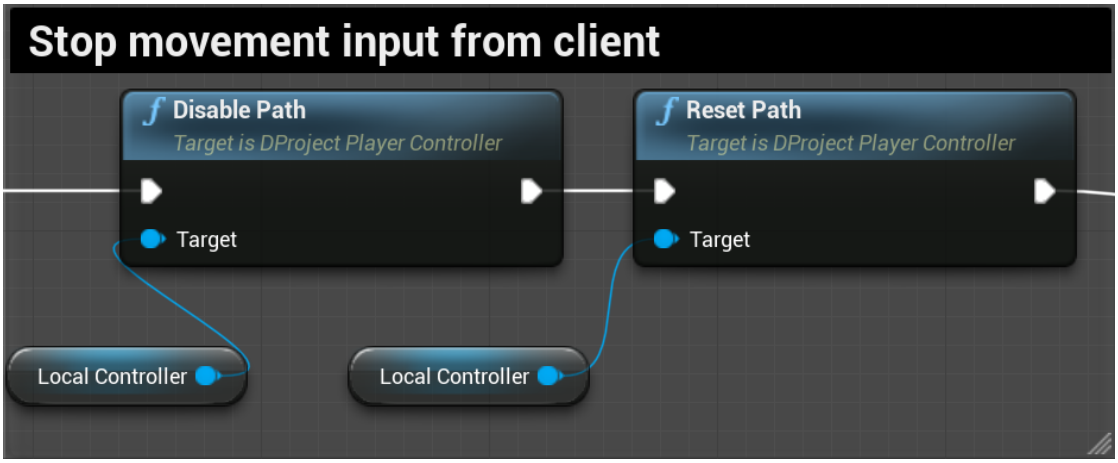

Figura 6.18: Disabilita il percorso che il personaggio stava attualmente seguendo

La forza viene applicata sfruttando la funzione *Apply Root Motion Constant Force*, che, banalmente, applica una forza di una potenza di valore *Strength*, in una direzione specifica.

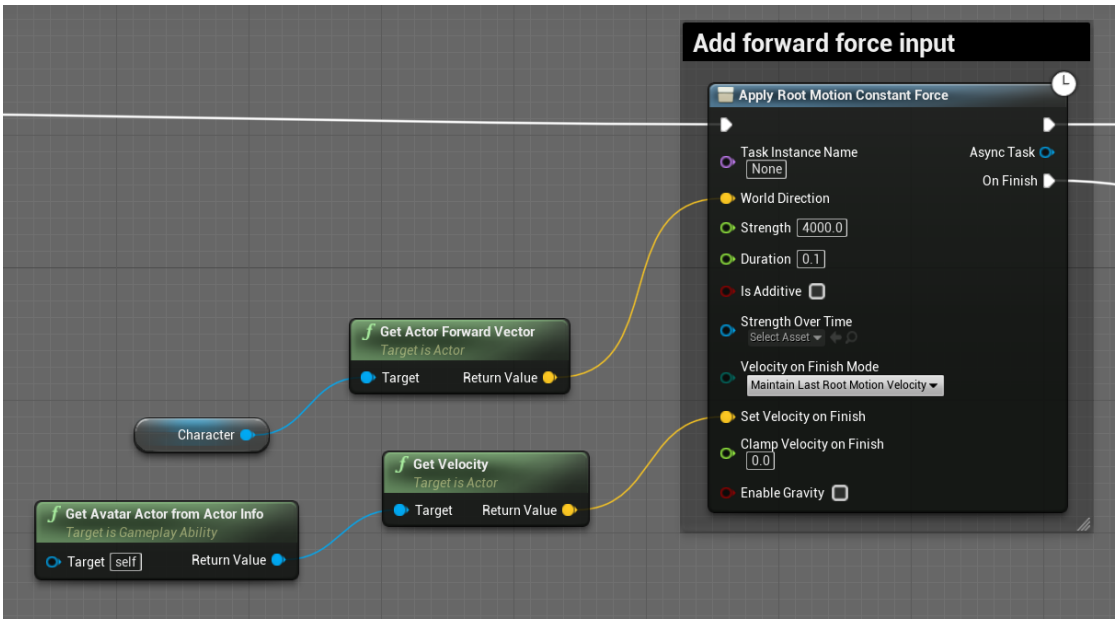

Figura 6.19: Dash Calcolo della forza da applicare al player

Attraverso il blueprint viene gestita anche l'esecuzione dell'animazione associata all'abilità, in questo caso utilizzando la funzione *Play Montage and Wait for Event*.

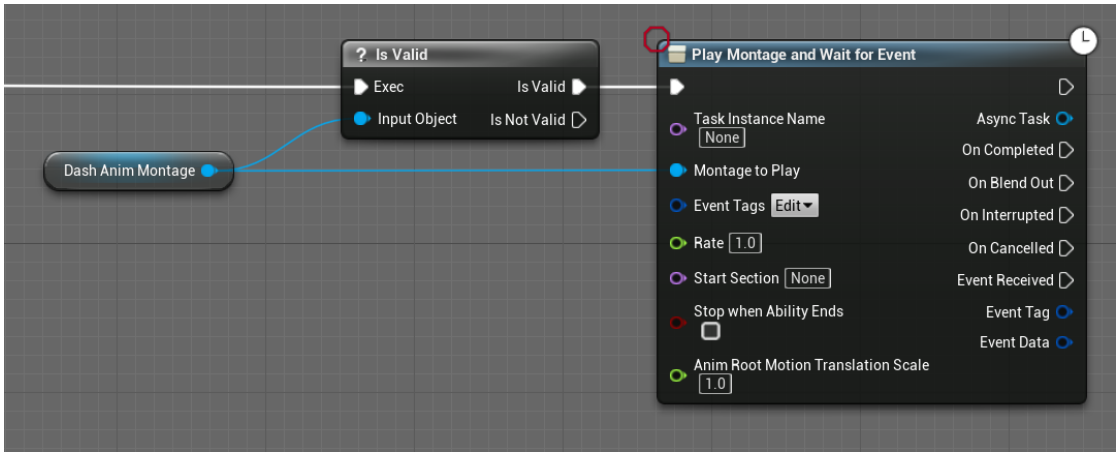

Figura 6.20: Esegue l'animazione associata al dash

Una volta eseguita tutta la logica relativa allo scatto, la conclusione dell'abilità viene gestita sfruttando il nodo *EndAbility*, del pacchetto Gameplay Ability System. Infine, viene riattivata la possibilità di calcolare i percorsi seguendo il normale flusso di input output.

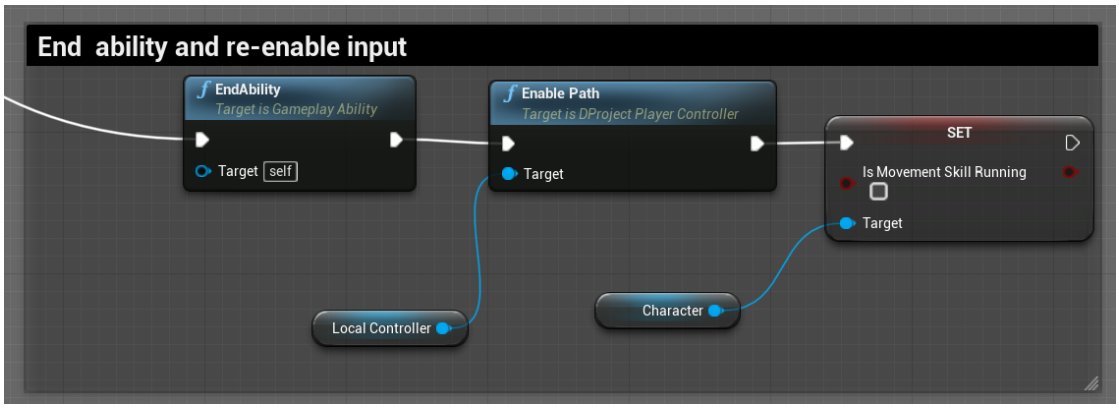

Figura 6.21: Conclude l'abilità

6.3 Osservazioni Conclusive

Naturalmente, le possibilità sono infinite, e quelle riportate sono solo alcune delle applicazioni del GAS.
In Bugball, sono state implementate molte altre abilità, tutte rispettano il pattern C++/Blueprint.

Come il lettore avrà notato, gli obiettivi che il GAS ci ha permesso di perseguire sono stati:

- **Modularità**: Permettere di sfruttare le classi, idealmente, in più di un'abilità, si pensi al gameplay effect del cooldown, o al gameplay effect dell'avvelenamento.
- **Flessibilità**: Permettere anche a figure non tecniche di gestire i blueprint e quindi di testare varie configurazioni per una stessa abilità in maniera rapida e indipendente. Di fatto, in fase di prototipizzazione di un videogioco risulta fondamentale la velocità di implementazione e poter testare diverse configurazioni nello stesso giorno. Lo svantaggio è la perdita di ottimizzazione, ma questo diventa un problema solo in fase di rilascio dell'applicazione.

Capitolo 7

Data Pipeline

Con Data pipeline si intende il processo di trasferimento dei dati da una sorgente ad una destinazione (es: un data warehouse).
Lungo la strada i dati possono essere trasformati e ottimizzati, arrivando in uno stato facilmente leggibile e utilizzato per sviluppare statistiche di business e non solo.
Durante la costruzione di una data pipeline le informazioni possono essere coinvolte in processi di aggregazione, organizzazione e spostamento. Le moderne data pipeline automatizzano molti degli step manuali coinvolti nel processo di trasformazione.
Tipicamente, questo include il caricamento dei dati grezzi in una tabella di staging (o intermedia) accessibile solo dal team data science, e un passaggio successivo ad una tabella ottimizzata pronta per collegarsi allo strato di visualizzazione.
Per analizzare tutti i dati che vengono prodotti da un'applicazione, è necessario avere un luogo centrale dove poter accedervi. Quando i dati vengono immagazzinati in più luoghi, può risultare difficoltoso accedervi e leggerne il contenuto, sopratutto, se c'è la necessità di fare analisi incrociate.
Un'altra caratteristica importante che deve possedere il flusso di dati è l'affidabilità, purtroppo, ci sono molti punti durante il trasporto, in cui possono esserci delle corruzioni o dei colli di bottiglia.
Questi sono alcuni dei motivi per cui avere una buona data pipeline è importante. Eliminerebbe gran parte degli step manuali che, paradossalmente, ancora oggi, vengono fatti per analizzare i dati, rendendo il processo più fluido e automatico. Inoltre, l'analisi manuale dei dati rende impossibile l'analisi in real-time o near real-time. Una data pipeline è composta da tre elementi essenziali:

- **Sorgenti**: Le sorgenti sono la fonte dei dati. Possono essere di vario tipo, dipendentemente dalla pipeline, database, CRMs, sensori, etc. In relazione al prodotto videoludico multigiocatore, la sorgente dei dati è il videogioco stesso, o meglio, gli eventi che nel videogioco si è scelto di catturare.

- **Fase di elaborazione**: In generale, i dati sono estratti da varie sorgenti, manipolati e trasformati attenendosi ai bisogni del caso, e poi, depositati nella destinazione. I passi di elaborazione, solitamente, includono:
  - Trasformazione
  - Arricchimento
  - Filtraggio
  - Raggruppamento
  - Aggregazioni
- **Destinazione**: La destinazione è dove i dati arrivano alla fine del percorso di elaborazione, tipicamente, un data lake o un data warehouse. Nel nostro caso, in un data lakehouse.

7.1 Data Pipeline in BugBall

Nel nostro caso, abbiamo cercato il giusto compromesso tra disponibilità dei dati, trasparenza del sistema utilizzato e flessibilità alle modifiche.
Non volevamo avere i dati estratti nel nostro database con troppo ritardo, ma allo stesso tempo non volevamo appesantire l'applicazione e gli step successivi della pipeline, sopratutto, perchè abbiamo utilizzato tecnologie a costo zero, o versioni di prova di alcuni prodotti.
Durante il processo di sviluppo ci sono stati vari cambiamenti ed evoluzioni della pipeline, tutt'ora è ancora in fase di sviluppo, potenzialmente non c'è un vero e proprio limite alle modifiche, poichè in prodotti di questo tipo gli eventi generati in app possono cambiare velocemente, quindi è necessaria una manutenzione costante.
è stato necessario mettere in comunicazione più servizi diversi, facendo scorrere i dati da una risorsa all'altra nel modo migliore. Generalmente, in casi di questo genere, è necessario rendere i dati provenienti da una sorgente del formato desiderato dalla destinazione. Inoltre, alcuni nodi della pipeline possono essere sia sorgenti che destinazioni, dipendentemente da che punto del processo che si sta osservando o sviluppando.
L'obiettivo è quello di scendere sempre più in profondità nei dati posseduti, arrivando ad uno stato finale in cui questi possano essere presentati a chiunque (figure tecniche e non tecniche) e osservati tramite delle dashboard.

7.1.1 Bronze, Silver e Gold Layer

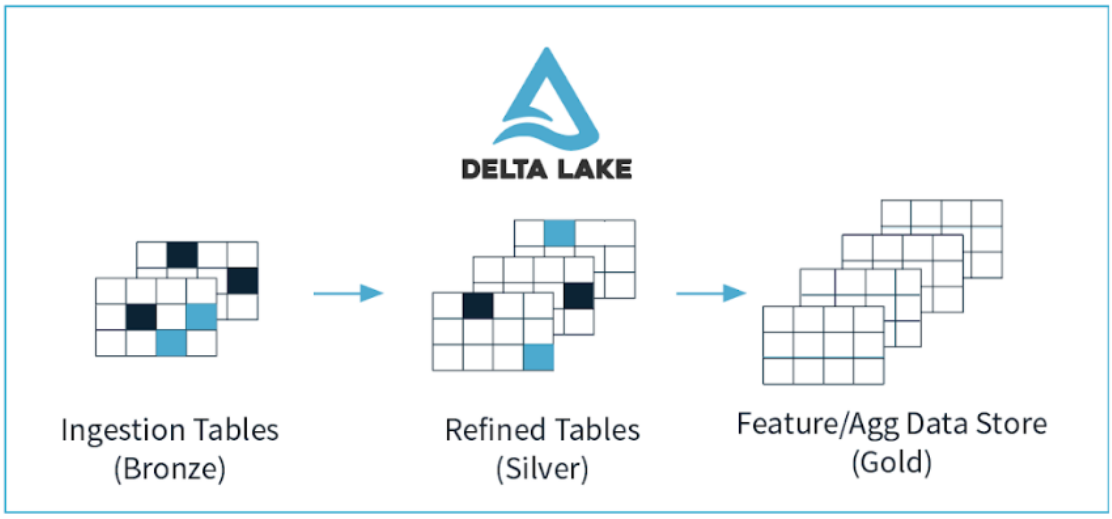

Figura 7.1: Paradigma delta lake

Il Bronze/Silver/Gold layer rappresentati nella figura, sono degli stati in cui i nostri dati si presentano nel data lake.
Con layer Bronzo definiamo i dati grezzi, non elaborati. Il layer silver rappresenta i dati che sono già passati attraverso un primo processo di pulizia e filtraggio.
Infine, il gold layer è il layer in cui possiamo trovare le aggregazioni business-level.
Nella realizzazione della pipeline di analisi delle partite è stato preso come riferimento questo pattern, adattandolo agli scopi di design di un videogioco. Di fatto, non abbiamo dei veri e propri dati di business, o meglio, il nostro *Business layer* sono tutti quegli insights utili ai designer del gioco, che hanno lo scopo di raffinarne le meccaniche.
Alcune domande che possono essere risposte sfruttando queste analisi sono:

- Quale parte della mappa è la più percorsa? e perchè?
- Dove vengono lanciate la maggior parte delle abilità?
- Qual è la percentuale delle abilità che vanno a buon segno rispetto alla totalità delle abilità lanciate?
- Esiste una sequenza di abilità, o combo, che se utilizzata ha un tasso di scoring maggiore delle altre?

7.1.2 Flusso generale

Nel nostro progetto, i vari layer dell'architettura delta lake sono stati rivisti e adattati ai nostri bisogni, organizzandoli in questo modo:

- **Bronze Layer**: Mongo DB Atlas è il nostro bronze layer, la scelta di utilizzare mongo è dovuta dalla sua flessibilità e velocità di utilizzo, si rimanda al capitolo 4 per le motivazioni specifiche, questa è l'unica differenza tra la nostra soluzione e il paradigma delta lake consigliato da databricks, di fatto, secondo la loro visione, anche il bronze layer dovrebbe far parte delle delta table.

- **Silver Layer**: Il Silver layer è stato costruito direttamente su Databricks, sfruttando *Spark on Python* per il collegamento con mongo.

- **Gold Layer**: sempre tramite Pyspark, i dati processati e salvati allo step precedente sono stai presi e utilizzati per costruire il layer di presentazione.

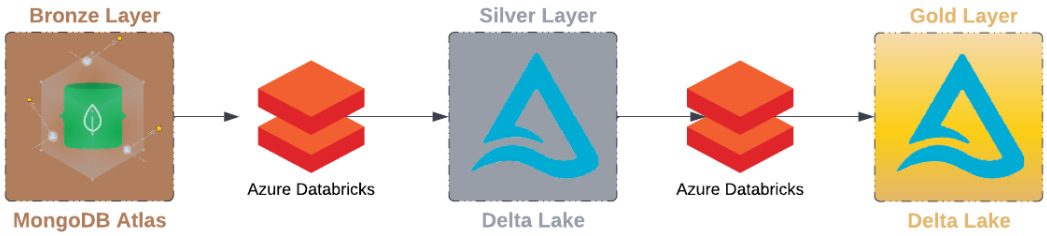

Figura 7.2: Bronze, Silver e Gold layer in bugball

Inizialmente, la distinzione non era così netta, il Silver layer e il gold layer non erano stati inseriti nell'architettura e sono stati necessari vari sprint per arrivare al risultato attuale. il modo migliore per capire come si è giunti alla soluzione è effettuare un veloce excursus nelle scelte implementative e nelle fasi dello sviluppo.

7.2 Acquisizione dei dati

L'acquisizione dei dati è il processo con il quale gli eventi di gioco prodotti durante una partita vengono catturati in una forma organizzata e pronta per essere immagazzinata in un Database. Il contesto ideale è quello in cui gli eventi da dover catturare arrivino già in una forma organizzata, e leggibile dal database

su cui si vogliono immagazzinare i dati. Purtroppo, il più delle volte questo non è vero, sopratutto in un videogioco che può presentare un elevatissimo tasso di customizzazione degli eventi.
Di conseguenza, per ogni evento catturato è stato necessario organizzare i dati in dei formati specifici sfruttando delle strutture dati messe a disposizione da Unreal. La tipologia di eventi che si è deciso di catturare sono tutti quelli che avvengono durante il gameplay vero e proprio, quindi, durante la partita. Tutte quelle azioni che, se analizzate, possono portare i designer a trarre delle conclusioni *Data-Driven*, e ad intercettare le criticità del videogioco.
In particolare, gli eventi sui quali la data pipeline è in ascolto sono:

- **IperStunEvent**: evento in cui un personaggio perde tutta la sua resistenza e viene stordito per qualche secondo.
- **PlayerLocation**: ogni 500ms viene catturata un'informazione sulla posizione del personaggio, in questo modo, è possibile fare osservazioni circa la navigabilità della mappa, e i pattern di movimento più utilizzati dai giocatori.
- **PlayerRespawnEvent**: utile per capire in quali momenti della partite il player è più soggetto a ritornare in campo.
- **ScoringEvent**: fornisce informazioni sul momento in cui una delle due squadre segna un punto.
- **SkillEvent**: Forse il tipo di evento più importante di tutti, detiene informazioni circa l'esecuzione di un'abilità all'interno del gioco, per ogni abilità vengono memorizzate informazioni diverse.
- **ThrowOutPlayerEvent**: memorizza informazioni riguardanti gli eventi di eliminazione, Posizione, status dell'eliminato prima di esserlo stato, ecc.
- **TimerEvent**: mantiene info riguardo al timer di gioco.
- **TryOutWithImmunityEvent**: quando un personaggio torna in gioco, gli è garantito un tempo di invulnerabilità, non alle abilità avversarie, ma alla barriera di eliminazione. Qui vengono catturati gli eventi in cui viene toccata la barriera di fondo campo con l'invulnerabilità attiva.
- **WinningEvent**: cattura informazioni riguardanti la vittoria di una squadra, uno per ogni partita.

Ma andiamo per gradi, il primo quesito a cui è stato necessario far fronte è stato se implementare l'acquisizione dei dati in blueprint o in C++. Dato che si è scelto di gestire le abilità sfruttando il blueprint (si rimanda al capitolo 6), per rimanere coerenti con questa scelta si è implementata la logica di catture degli

eventi direttamente in blueprint. Dopo aver catturato gli eventi, si è sfruttata una classe C++, chiamata *UTelemetrySystemConnector* per l'elaborazione di queste informazioni e per la connesione al DB su cui immagazzinarli.

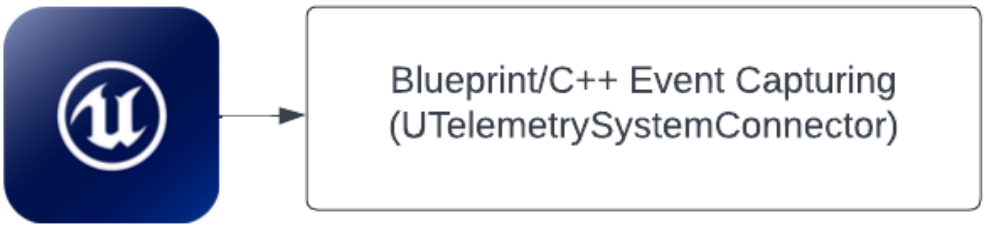

Figura 7.3: Architettura acquisizione dei dati dagli eventi unreal

Un evento viene catturato sfruttando la struttura dati *TMap* nativa di unreal. Le TMaps sono mappe definite da due tipi, un tipo chiave e un tipo valore, che sono immagazzinati come coppie all'interno della mappa.
Nel progetto, I due tipi utilizzati per catturare le informazioni sono *<FString, FString>*, ovvero una coppia di stringhe, questo è stato necessario per permettere la scrittura su database.
ogni volta che un evento interessante avviene, una TMap viene popolata con le informazioni più interessanti riguardanti l'evento, alcune di queste possono essere: La posizione all'interno della mappa di gioco in cui l'evento è avvenuto, Il nome del giocatore che ha innescato l'evento, se presente, ecc. Analizziamo un esempio di cattura delle informazioni di un evento, in particolare, analizziamo la gameplay ability skillshot descritta nel capitolo 6. Quando l'abilità viene attivata, viene innescato un flusso di cattura di alcune informazioni.
In gioco, l'attivazione dello skillshot si presenta in questo modo:

Figura 7.4: Rappresentazione in game del momento di direzionamento dello skillshot

Figura 7.5: Rappresentazione in game del momento dello spawn del proiettile

Ma ciò che avviene in background, è questo:

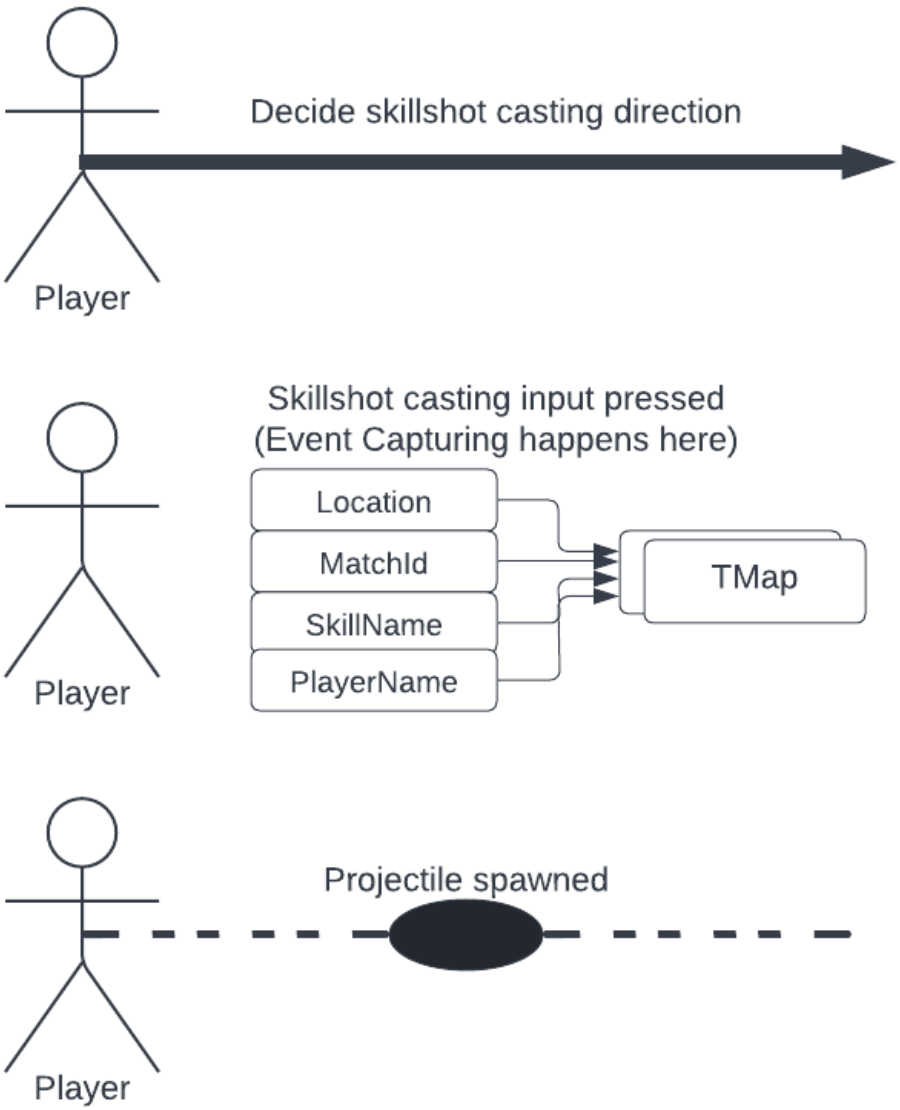

Figura 7.6: Flusso di esecuzione del lancio di un'abilità e cattura delle informazioni

Come si può notare, la cattura delle informazioni avviene al momento del commit dell'abilità da parte del giocatore. In questo caso specifico, si sono utilizzate delle informazioni d'esempio, vengono immagazzinate solo le seguenti informazioni:

- Location: la posizione del giocatore al momento del lancio dell'abilità.

- MatchId: un Id unico che identifica la partita in corso, fondamentale in fase di analisi dei dati.

- SkillName: il nome dell'abilità utilizzata, anche questa informazione essenziale durante la fase di analisi.

- PlayerName: il nome del lanciatore dell'abilità, non del giocatore umano, ma il nome che rappresenta il character utilizzato per lanciare l'abilità, ad esempio, se viene utilizzata l'ape, questo campo avrà il valore di *BPBeeCharacter"*.

Nota bene: in questa fase vengono catturate le informazioni, e immagazzinate in memoria, il momento di scrittura su database viene ritardato.
Nell'immagine successiva, vengono riportati i nodi che si occupano della cattura delle informazioni che compongono l'evento che si sta analizzando.

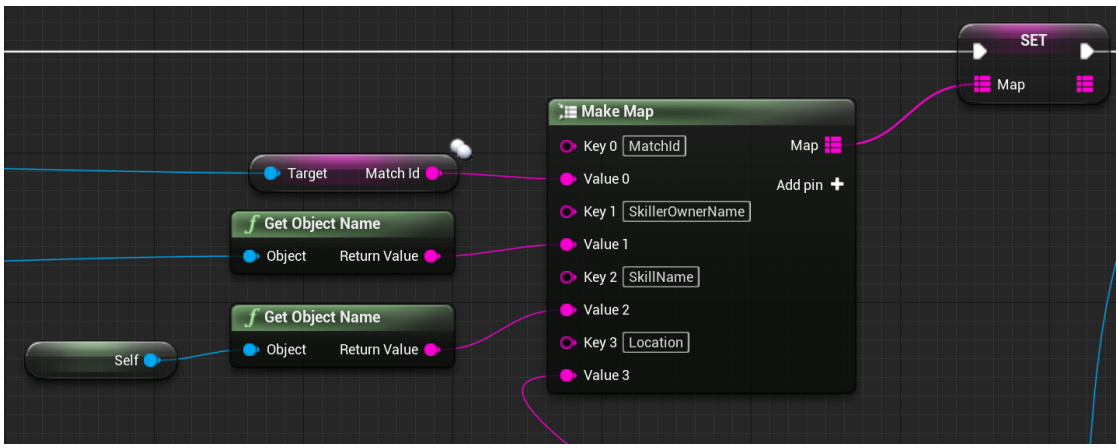

Figura 7.7: Creazione della TMap in blueprint per la cattura degli eventi

Il valore del *MatchId* viene generato randomicamente nella *Game Mode*, una classe presente in ogni gioco sviluppato in Unreal, che detiene le regole del gioco e gestisce il flow generale della partita.
Non è lo scopo di questo capitolo scendere nei dettagli della Game Mode, basti sapere che molte informazioni generali riguardanti il gioco come: Durata di una partita, secondi passati dall'inizio del match, punteggi, Id della partita, ecc, sono contenute nella game mode.
Inoltre, sfruttando un riferimento al lanciatore dell'abilità e all'abilità stessa, basta sfruttare la funzione *Get Object Name*, per ottenere le informazioni desiderate.
L'informazione riguardante la posizione del giocatore ha richiesto un grado di elaborazione in più per essere memorizzata in un formato corretto nella mappa.

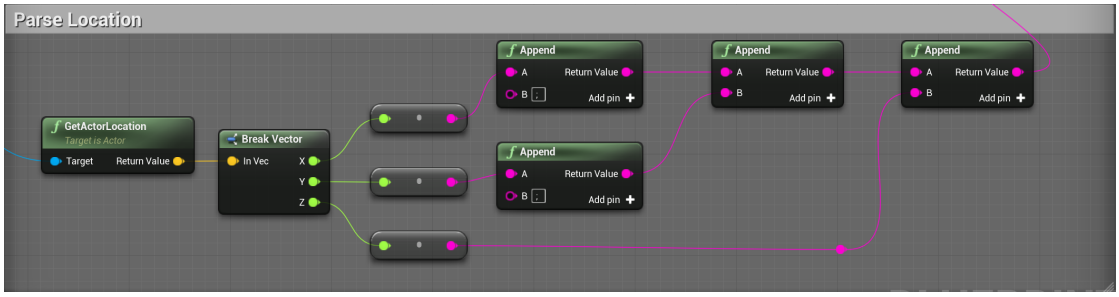

Figura 7.8: Parsing della location per essere inserita nella mappa

Come si può notare, il vettore rappresentante la posizione viene spianato e passato alla mappa non più come vettore, ma come una stringa, Questo facilità il processo di scrittura su DB, poichè in C++ queste informazioni verranno ulteriormente modificate per ottenere il formato giusto. Quella coppia, si presenterà come una coppia di stringhe in questo modo:

Key:	"Location"
Value:	xValue;yValue;zValue

Figura 7.9: Chiave e valore della posizione nella TMap

Una volta catturate e organizzate le informazioni, vengono essere processate dalla classe che si occupa della connesione a Mongo DB, in modo da memorizzarle in streaming al momento della loro generazione.
Questa era la prima soluzione realizzata per il salvataggio degli eventi, ogni volta che ne veniva generato uno, si sfruttava la connessione a mongo per salvarlo istantaneamente.
Questo metodo non ha dato problemi fino a che la connessione era instaurata con un'istanza di mongo db eseguita in locale sulla macchina del tester.

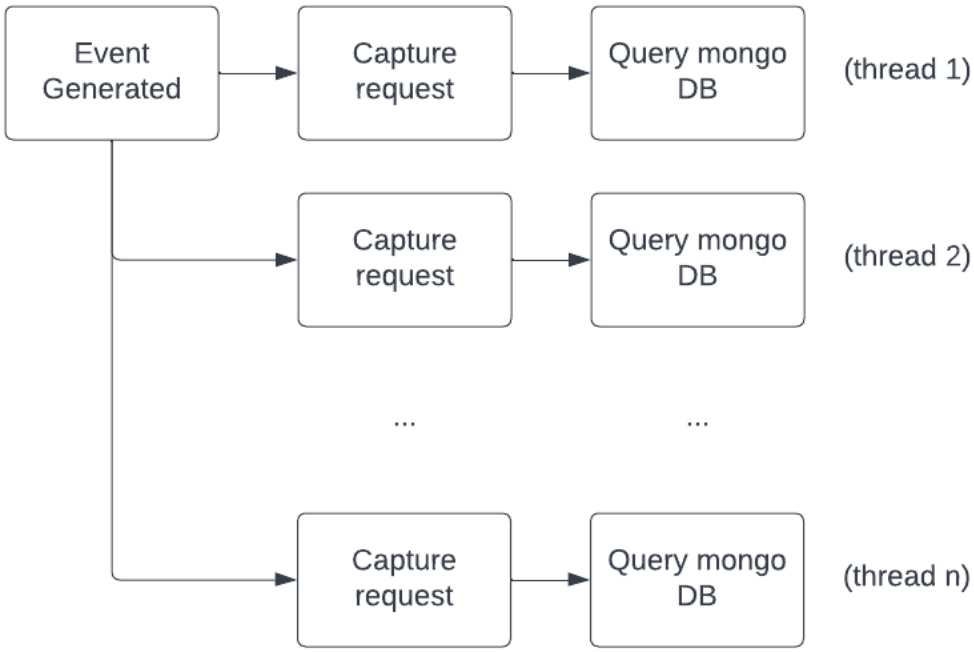

Figura 7.10: Scrittura su mongo in real-time

Quando il gioco ha visto l'ingresso del multiplayer e della connessione ad un'istanza in cloud di mongo DB, la soluzione ha iniziato a presentare vari problemi, tra cui, la necessità di mantenere una connessione costante con un'istanza il database, il processamento di funzioni aggiuntive durante l'esecuzione della partita, e l'impossibilità di gestire le eccezioni in unreal engine.
Questo problema è stato superato sfruttando una scrittura su DB di tipo batch, e l'utilizzo di una struttura dati intermedia tra l'acquisizione dei dati e la scrittura su mongo DB.

7.2.1 Multi-producer single-consumer queue

La struttura dati intermedia utilizzata per bufferizzare le richieste di scrittura a mongo DB è una **coda multi-produttore e singolo consumatore illimitata e parallela**.
Praticamente, una coda che permette l'inserimento di più oggetti in contemporanea e il consumo sequenziale degli stessi.

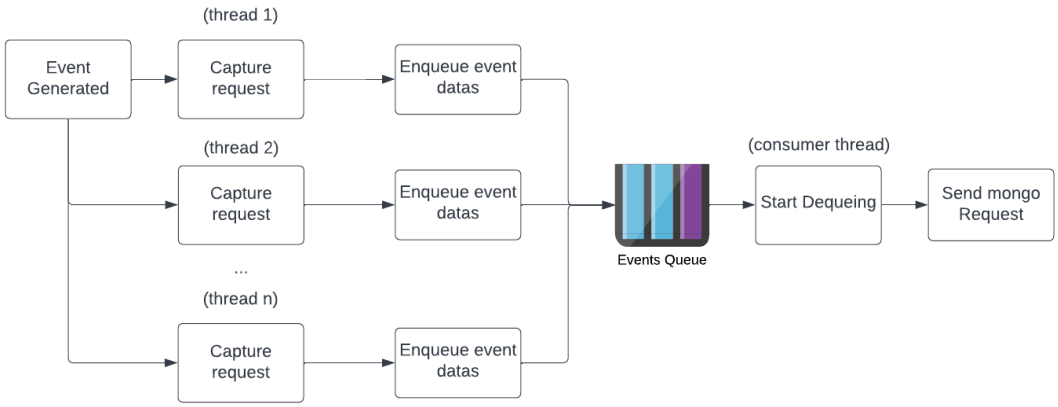

Figura 7.11: Pipeline di funzionamento della coda MPSC

Una struttura di questo tipo ha i seguenti vantaggi:

- Produttori veloci e senza attese.
- Consumatore estremamente veloce, senza necessità di sincronizzazione poichè è una lettura single-thread.
- l'operazione di pop è sempre O(1), non c'è bisogno di riordinare.

In Unreal Engine 4 non esiste una struttura di questo tipo, esiste in Unreal Engine 5, gli sviluppi del gioco si trovavano già in uno stato avanzato al momento della decisione di usare una struttura di questo tipo, è stata fatta un'integrazione del codice sorgente di unreal Engine 5 con quella del nostro motore.
Banalmente, è stata integrata la classe *MpscQueue.h*, proveniente dal codice sorgente di Unreal Engine 5, nel codice sorgente di Unreal Engine 4.
Questa struttura dati può essere utilizzata in due modi:

- **Scrittura periodica su DB**: data una determinata frequenza di scrittura, ogni intervallo di tempo prefissato si avvia una fase di dequeuing, ad esempio, ogni minuto pulisci la coda da tutti gli elementi inseriti nel minuto precedente.
- **Scrittura su DB alla fine della produzione degli eventi**: In questo caso la coda accumula elementi fino alla fine della partita, per poi venire svuotata al termine della produzione degli eventi.

Nel progetto, si sono testate entrambe le soluzioni, si è scelta la seconda per evitare eventuali situazioni di sovraccarichi ogni volta che inizia la fase di scrittura.

7.3 Mongo DB

Una volta catturati i dati, è necessario che questi vengano immagazzinati in una struttura stabile e non volatile.
è stato scelto Mongo DB per adempiere a questo compito. Mongo DB è un sistema open-source di gestione database NoSQL. I database NoSQL vengono utilizzati come alternativa ai classici databse relazionali. MongoDB è uno strumento in grado di gestire informazioni *document-oriented*, supporta varie forme di dato, è una delle tecnologie noSQL più utilizzate in ambito big data, sopratutto in quei casi in cui i dati non seguono un *Data model* rigido.
Piuttosto che utilizzare tabelle e righe, come in un database relazionale classico, l'architettura di MongoDB è fatta di collezioni e documenti.

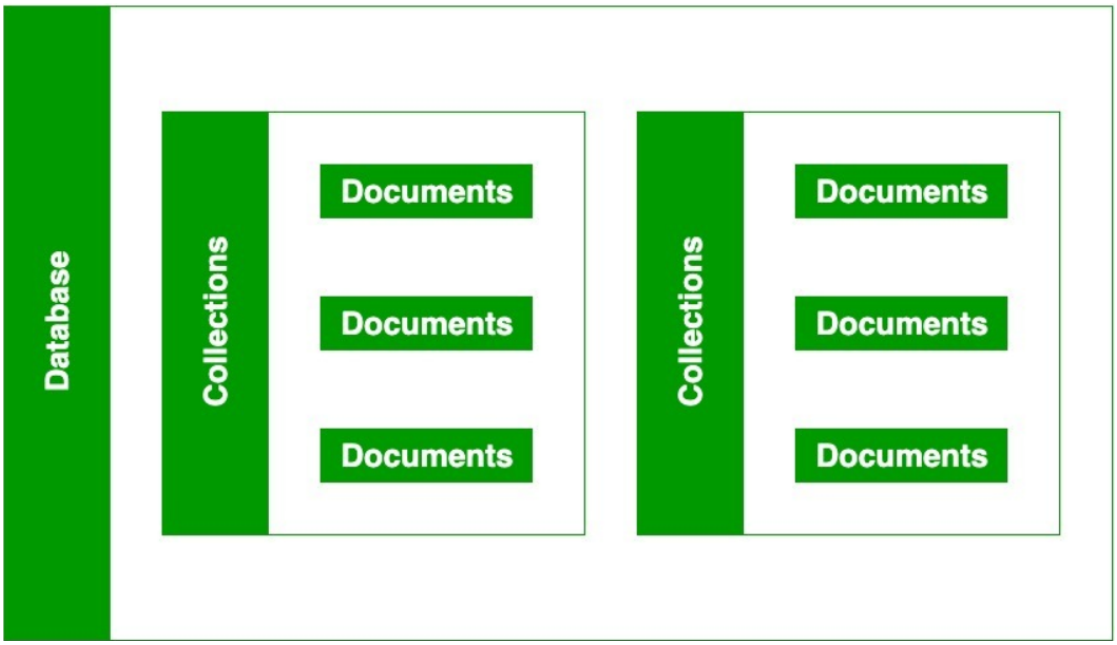

Figura 7.12: Struttura delle informazioni contenute in mongoDB

Un documento è una struttura dati composta da coppie chiave-valore, sono la unità base di dati in MongoDB. I documenti sono molto simili alla JavaScript Object Notation, nota anche come *JSON*, ma formalmente sono chiamati *Binary JSON* (BSON).
Un JSON non riconosce il tipo di dato contenuto all'interno di uno dei suoi campi, il BSON si. Ad esempio, le informazioni di un evento di skillshot, in mongoDB vengono rappresentate in questo modo:

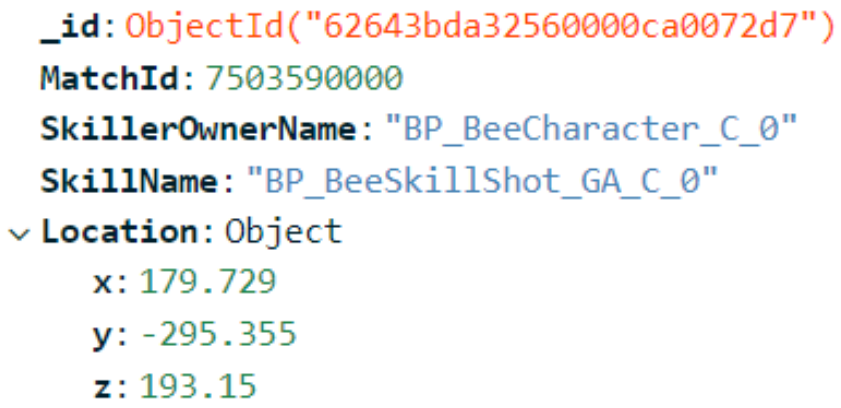

Figura 7.13: Struttura del BSON rappresentate l'evento di skillshot

Si può notare che sono le stesse informazioni catturate e inserite nella mappa nel blueprint.
Un altro esempio utile è quello del dash, il processo di lettura, scrittura su database e visualizzazione in formato BSON è lo stesso presentato per lo skillshot.

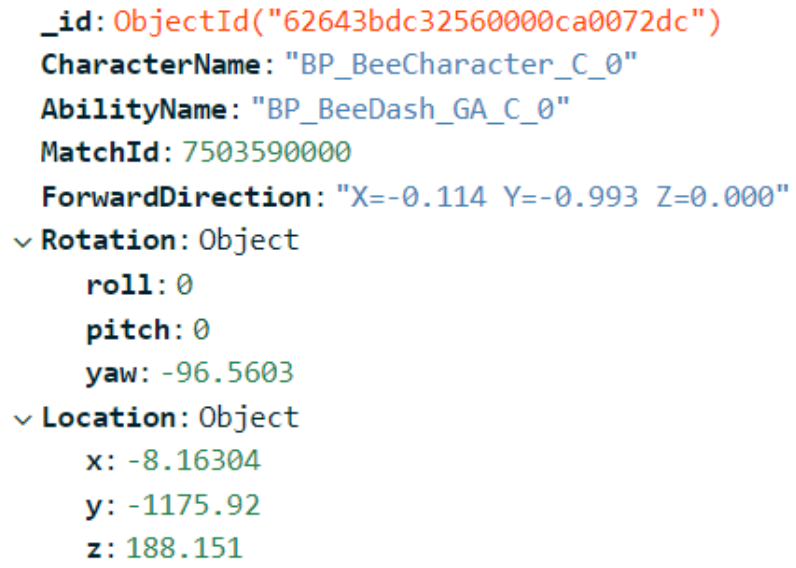

Figura 7.14: Struttura del BSON rappresentate l'evento di dash

I campi in questi documenti sono simili alle colonne in un database di tipo relazionale.

I valori contenuti possono essere vari tipi di dato, altri documenti, array, interi, stringhe, ecc.
Ogni documento sarà accompagnato da un'id che fungerà da identificatore univoco, assegnata in automatico da mongoDB al momento della scrittura.
Un insieme di documenti è chiamato **collezione**, che ha la stessa funzione della tabella nei database relazionali.
Le collezioni possono contenere ogni tipo di dato, i documenti contenuti in una collezione possono avere strutture differenti tra loro, non è necessario che rispettino una struttura specifica.
Questo garantisce all'utilizzatore un alto grado di flessibilità per creare qualsiasi numero di campi nel documento, rendendo facile scalare il database in confronto ai database di tipo relazionale. Un vantaggio nell'utilizzo dei documenti è che i loro campi hanno un mapping nativo con i tipi di dato di moltissimi linguaggi di programmazione. Inoltre, avere documenti innestati riduce di molto il numero di join necessari, il che riduce i costi.

7.3.1 Connessione con Unreal

MongoDB supporta la connessione con vari linguaggi di programmazione attraverso i *MongoDB Driver*, librerie ufficiali che permettono di utilizzare le funzionalità di mongo in vari ambienti. Piuttosto che sviluppare tutto il sistema di interrogazione a MongoDB da zero, si è scelto di utilizzare un plugin di unreal, che utilizza i driver C++ di mongo per rendere disponibili delle funzionalità (sia in blueprint, che in C++) per interrogare MongoDB. Il plugin si chiama *MongoDB Driver*, sviluppato dal team *Pandores*.

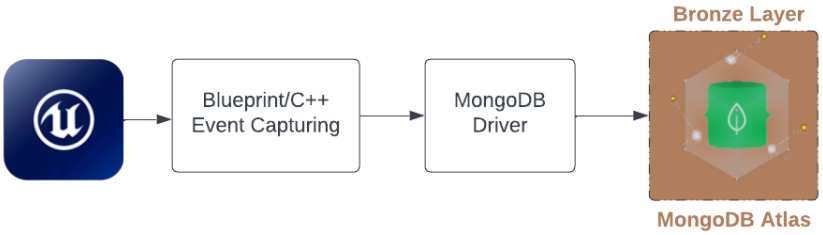

Figura 7.15: Connessione Da Unreal a MongoDB (Bronze Layer)

Prima di iniziare ad interrogare il DB, sarà necessario instaurare una connessione. Al momento della richiesta di connessione, è possibile specificare il numero minimo e massimo di connessioni parallele che il sistema può tenere attive. Ogni connessione è gestita da un thread diverso, per questo si consiglia di non eccedere. Nel nostro

caso, la connessione è mono-thread, si è scelto di delegare ad un solo thread la scrittura, poichè le macchine utilizzate sono low-priority, inviando troppe richieste in contemporanea si rischierebbe che alcune di esse non vengano prese in carico.

In prima istanza, è stato utilizzato un database locale, accessibile tramite il metodo *CreatePool*, che richiede di specificare alcune informazioni come la porta e l'indirizzo a cui si trova l'istanza locale di mongo, restituisce un puntatore al gestore della connessione, che verrà utilizzato per interrogare mongo.
Per visualizzare e interagire con il db in locale è stato utilizzato *MongoDB Compass*, uno strumento che tramite un sistema di interfacce grafiche permette di interagire con il db in modo molto intuitivo.

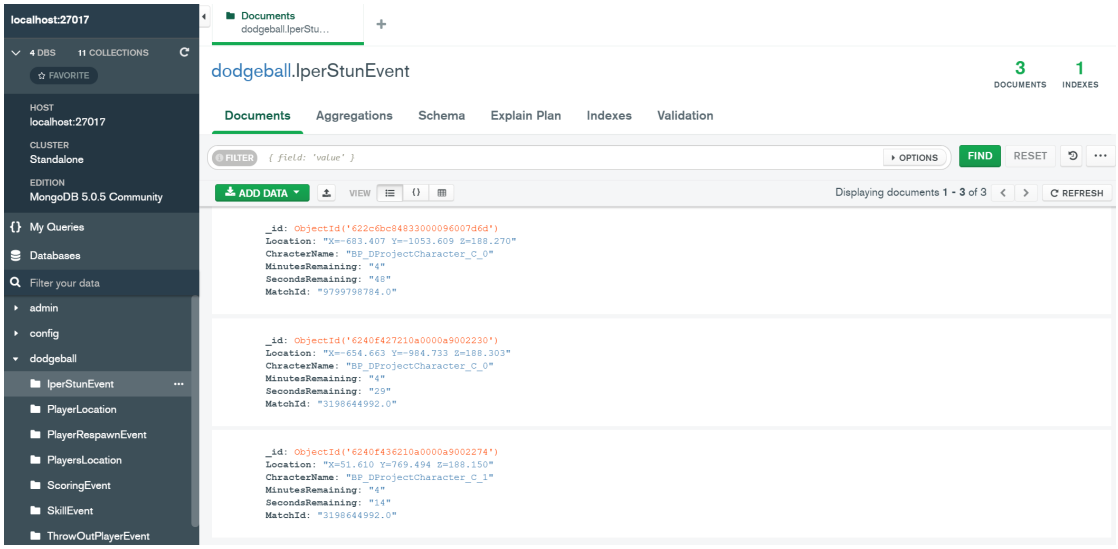

Figura 7.16: GUI di MongoDB Compass

Naturalmente, questa soluzione è stata utilizzata solo come test per verificare il corretto funzionamento della connessione unreal-mongo.
Lo step successivo è stato quello di rendere accessibile mongo a tutti gli sviluppatori, in modo da non perdere nessun dato di telemetria, nemmeno in fase di testing.
Per risolvere questo problema è stata utilizzata un'istanza cloud di mongo, **MongoDB Atlas**. La logica di funzionamento è praticamente la stessa, con la differenza che l'hardware su cui viene eseguito Mongo non è più on-premises, ma in cloud.

7.3.2 Struttura del database

In ogni collezione è contenuto un tipo diverso di evento, sfruttando la possibilità di inserire record con strutture differenti tra loro, questa è risultata essere una

strategia vincente. In questo modo, all'interno della collezione *SkillEvent*, è possibile trovare tutti gli eventi che contengono ogni singolo momento in cui è stata utilizzata un'abilità all'interno del gioco.
In questo modo si mantiene lo stesso flow per ogni abilità che viene catturata, e tramite Python è possibile effettuare differenti tipi di analisi, escludendo o includendo i tipi di abilità desiderati.
Alcune delle domande che possono trovare risposta sfruttando questa strategia sono:

- Quale tipo di skill è la più utilizzata?
- Quante abilità di quelle lanciate vanno a segno?
- Qual è la percentuale di skillshot rispetto al totale delle abilità lanciate?

7.4 Elaborazione dei dati

La fase finale della pipeline è quella di elaborazione ed analisi dei dati. Una volta che il processo di cattura degli eventi è in piedi, è essenziale rendere i dati memorizzati il più leggebili possibili, ed estrarre, dal risultato di quest'ultima fase, delle analisi che portino a delle conclusioni utili al design del prodotto.
Fin da subito la scelta delle tecnologie da utilizzare per l'analisi dei dati è ricaduta su Python.
In particolare, si è scelto di utilizzare *PySpark*, API Python per l'utilizzo di **Apache Spark**, un framework open-source e distribuito, orientato alla manipolazione di dati real-time, e di grosse dimensioni, sfruttando il calcolo parallelo e sistemi batch.
Il tipo di dato chiave usato in PySpark è il **dataframe**. Questo oggetto può essere visto come una tabella distribuita tra vari cluster, ogni operazione che viene compiuta sui dataframe avviene in modo distribuito.

7.4.1 Analisi locale

Inizialmente la pipeline di lettura e analisi dei dati è stata scritta in locale, sfruttando un'istanza di Spark installata su sistema operativo Windows 11.
Veniva instaurata una connessione con il bronze layer, e tramite PySpark venivano creati dei data frame relativi alle varie collection presenti nel Database.
Infine, sfruttando *Pandas on PySpark*, le API per utilizzare Pandas con la libreria PySpark, è stato possibile compiere delle semplici analisi.

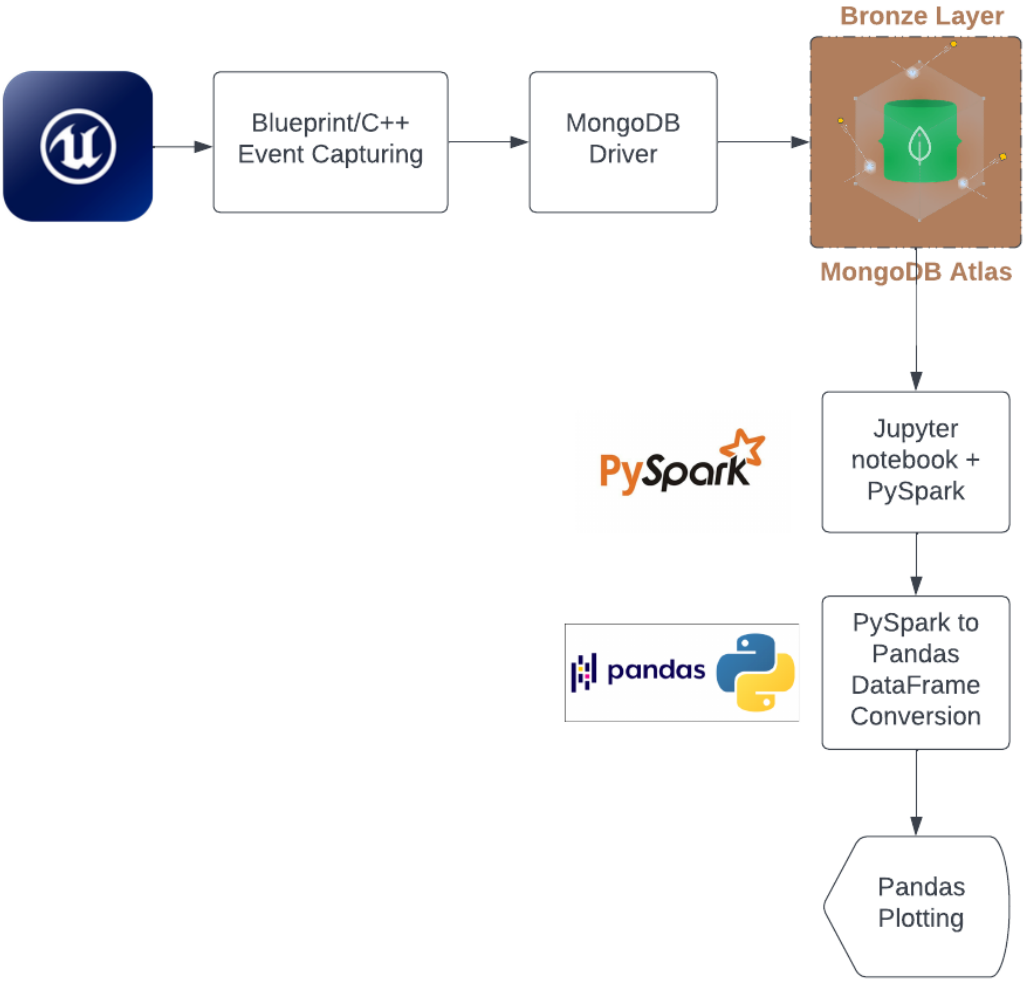

Figura 7.17: Pipeline di dati con l'ingresso di Pyspakr (in locale)

Per rappresentare le collezioni che troviamo in mongoDB sottoforma di Data frame è necessario specificare uno *Schema*, una collezione di colonne tipizzate. Tutto lo schema è immagazzinato come un *StructType* e le colonne che ne definiscono la struttura sono *StructFields*.
Praticamente, è necessario definire la struttura dei dati che verranno letti da Mongo.
Prendiamo come esempio la collezione *Player Location Event*, dove ogni entry rappresenta la posizione dei giocatori nella mappa in un momento specifico della partita.
In questa collezione alcuni campi sono strutturati, come il campo *Location*, che

presenta i 3 valori posizionali x, y e z.
Per far si che la lettura avvenga, è possibile definire alcuni campi dello StructType come StructType stessi, creando di fatto una struttura nidificata. Definire uno schema di questo tipo di eventi può essere fatto come segue:

```
#-----------------Player Location--------------------
playerLocationSchema = StructType([
    StructField("Location", StructType([
        StructField("x", DoubleType(), True),
        StructField("y", DoubleType(), True),
        StructField("z", DoubleType(), True)
    ])),
    StructField("CharacterName", StringType(), True),
    StructField("MatchId", IntegerType(), True),
    StructField("MinutesRemaining", IntegerType(), True),
    StructField("SecondsRemaining", IntegerType(), True),
    StructField("Rotation", StructType([
        StructField("roll", DoubleType(), True),
        StructField("pitch", DoubleType(), True),
        StructField("yaw", DoubleType(), True)
    ]))
])
```

Figura 7.18: Schema del Data Frame relativo agli eventi di player location

In questo momento, abbiamo tutti gli elementi necessari alla creazione del data frame, e si prosegue sfruttando i seguenti metodi:

- read: per comunicare a spark di abilitare la lettura ad una fonte di dato ancora non specificata.
- format: comunica il formato della fonte di dato, può essere un csv, txt, o altro. In questo caso è *com.mongodb.spark.sql.DefaultSource*.
- option: permette di aggiungere delle opzioni di configurazioni aggiuntive, in questo caso, specifichiamo la stringa di connessione a mongo e il nome della collezione.
- schema: viene specificato la struttura dei dati che devono essere letti.

```
playerLocationDF = (sqlContext
                    .read
                    .format("com.mongodb.spark.sql.DefaultSource")
                    .option("uri", mongo_ip + ".PlayerLocation")
                    .schema(playerLocationSchema)
                    .load())
```

Figura 7.19: Snippet del codice che permette la lettura della collezione sottoforma di data frame

In questa fase, i dati all'interno del data frame possono essere visualizzati, interrogati e manipolati come se fossero contenuti in un formato tabellare.

```
display(playerLocationDF)
```

Location	CharacterName	MatchId	MinutesRemaining	SecondsRemaining	Rotation
{-40.0005, -1370....	BP_BeeCharacter_C_0	2147483647	0	0	{0.0, 0.0, 0.0}
{-30.8657, -1087....	BP_BeeCharacter_C_0	2147483647	4	59	{0.0, 0.0, 114.624}
{-40.0005, -1370....	BP_BeeCharacter_C_0	2147483647	0	0	{0.0, 0.0, 0.0}
{-38.3416, -1354....	BP_BeeCharacter_C_0	2147483647	4	59	{0.0, 0.0, 74.667}
{-273.437, -413.1...	BP_BeeCharacter_C_0	2147483647	4	58	{0.0, 0.0, 70.5148}
{-138.668, -148.5...	BP_BeeCharacter_C_0	2147483647	4	58	{0.0, 0.0, 43.2591}
{132.582, -102.65...	BP_BeeCharacter_C_0	2147483647	4	57	{0.0, 0.0, -16.5907}
{272.081, -144.41...	BP_BeeCharacter_C_0	2147483647	4	57	{0.0, 0.0, -18.7617}
{179.729, -295.35...	BP_BeeCharacter_C_0	2147483647	4	56	{0.0, 0.0, -127.8}
{-3.53706, -532.8...	BP_BeeCharacter_C_0	2147483647	4	56	{0.0, 0.0, -128.147}
{-222.987, -708.1...	BP_BeeCharacter_C_0	2147483647	4	55	{0.0, 0.0, -83.2256}
{-80.876, -949.62...	BP_BeeCharacter_C_0	2147483647	4	55	{0.0, 0.0, 67.0676}
{-77.7494, -952.8...	BP_BeeCharacter_C_0	2147483647	4	54	{0.0, 0.0, 67.0676}
{-54.9596, -999.3...	BP_BeeCharacter_C_0	2147483647	4	54	{0.0, 0.0, -63.8023}
{-69.796, -1775.8	BP_BeeCharacter_C_0	2147483647	4	53	{0.0, 0.0, -91.5144}

Figura 7.20: Rappresentazione dei dati contenuti nel data frame

Da questo momento in poi, si aprono moltissime possibilità di analisi, di qualsiasi tipo. Nonostante l'analisi dei dati catturati non sia stato il focus principale di questa tesi, si riporta comunque un esempio di possibile analisi che può essere eseguita con una struttura di questo genere.
Si immagini di voler capire quale area della mappa è la più utilizzata dai giocatori, sapendo che nel dataframe *PlayerLocationDF* troviamo l'informazione **Location**, possiamo visualizzare in un rettangolo rappresentante la nostra area di gioco tutti

i punti in cui il giocatore si è trovato nelle ultime 10 partite.
Nel nostro caso, la partita si svolge in una mappa 2D, per mantenere le cose semplici, prendiamo i quattro punti rappresentanti i vertici della mappa:

- A = (780, -2080)
- B = (-780, -2080)
- C = (-780, 2080)
- D = (780, 2080)

Possiamo affrontare il problema in tre passi:

1. Estrarre tutte le coppie (X,Y) dalla colonna "Location".
2. Normalizzare vertici e punti di posizione, mantenendo i valori compresi tra [-1, 1], utile per facilitarci i calcoli.
3. Plottare tutti i punti estratti in un piano 2D, con l'obiettivo di capire dove passa la maggior parte del tempo il giocatore.

Per estrarre i valori dalla colonna location possiamo utilizzare la funzione *select*, probabilmente la funzione più utilizzata quando si ha a che fare con i dataframe, permette di selezionare una porzione di dati specifica, è possibile anche specificare delle condizioni, rendendo l'interrogazione del data frame molto simile ad una query SQL.

```
pointsExtractedDF = playerLocationDF.select("Location.x", "Location.y")
pointsExtractedDF.show(truncate = False)
```

```
+--------+------+
|x       |y     |
+--------+------+
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
|-5.99E-4|1470.0|
+--------+------+
```

Figura 7.21: Estrazione delle posizioni X ed Y di tutti gli eventi di location catturati, e visualizzazione dei primi 20

Dalla figura si può notare che il valore di X per i dati estratti risulta essere sempre lo stesso, probabilmente perchè, per pura casualità, sono stati visualizzati gli eventi catturati ad inizio partita, quando il giocatore ancora non può muoversi. Per risolvere il problema, si potrebbe fare un'ulteriore pre-processing, eliminando tutte le entry con tempo di acquisizione pari all'inizio della partita.
Ma comunque non risulta problematico ai fini dell'analisi poichè sono solo una piccola parte rispetto alla totalità dei dati.
Per quanto riguarda l'estrazione, sono state create due *user defined function on spark* che si occupano della normalizzazione dei valori contenuti nelle colonne in modo parallelo.
Le user defined function in Pyspark, sono funzioni scritte in linguaggio python utilizzabili sui data frame, sfruttando tutte le funzionalità messe a disposizione da Spark.

```
def divide_by_maximum_x_value(number):
    return number / 780.0

def divide_by_maximum_y_value(number):
    return number / 2080.0

divide_by_maximum_x_value_udf = funcs.udf(divide_by_maximum_x_value, DoubleType())

divide_by_maximum_y_value_udf = funcs.udf(divide_by_maximum_y_value, DoubleType())
```

Figura 7.22: Definizione delle funzioni di normalizzazione

Verranno poi utilizzate in questo modo:

```
normalizedPointsDF = (pointsExtractedDF.withColumn('normalizedX', divide_by_maximum_x_value_udf('x'))
                      .withColumn('normalizedY', divide_by_maximum_y_value_udf('y'))
                      .select('normalizedX', 'normalizedY'))

normalizedPointsDF.show(truncate = False)
```

```
+--------------------+------------------+
|normalizedX         |normalizedY       |
+--------------------+------------------+
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
|-7.67948717948718E-7|0.7067307692307693|
+--------------------+------------------+
```

Figura 7.23: Utilizzo delle udf di normalizzazione

Arrivati a questo punto, i dati sono pronti per essere plottati.
PySpark non offre molte possibilità di plotting, per questo è stata utilizzata la libreria *Pandas on Spark*, che ci permette di utilizzare la funzione *scatter*, che disegna un grafico scatter dei punti estratti. All'atto pratico, viene convertito il data frame spark, in un data frame pandas, che viene utilizzato per la visualizzazione.

```
verticesPoints = spark.createDataFrame([(1,1), (1,-1), (-1,1), (-1,-1)], ['normalizedX', 'normalizedY'])
dfWithVertices = normalizedPointsDF.union(verticesPoints)

pandasDF = dfWithVertices.toPandas()

pandasDF.plot.scatter(x='normalizedX', y='normalizedY')
```

Figura 7.24: Codice dello scatter plot

Del risultato di questo codice si hanno a disposizione due plot differenti, il primo relativo ad una fase di testing in cui non si avevano ancora molti dati da analizzare a disposizione, il secondo relativo ad una fase avanzata dello sviluppo del gioco in cui si possedevano molti più dati a disposizione.

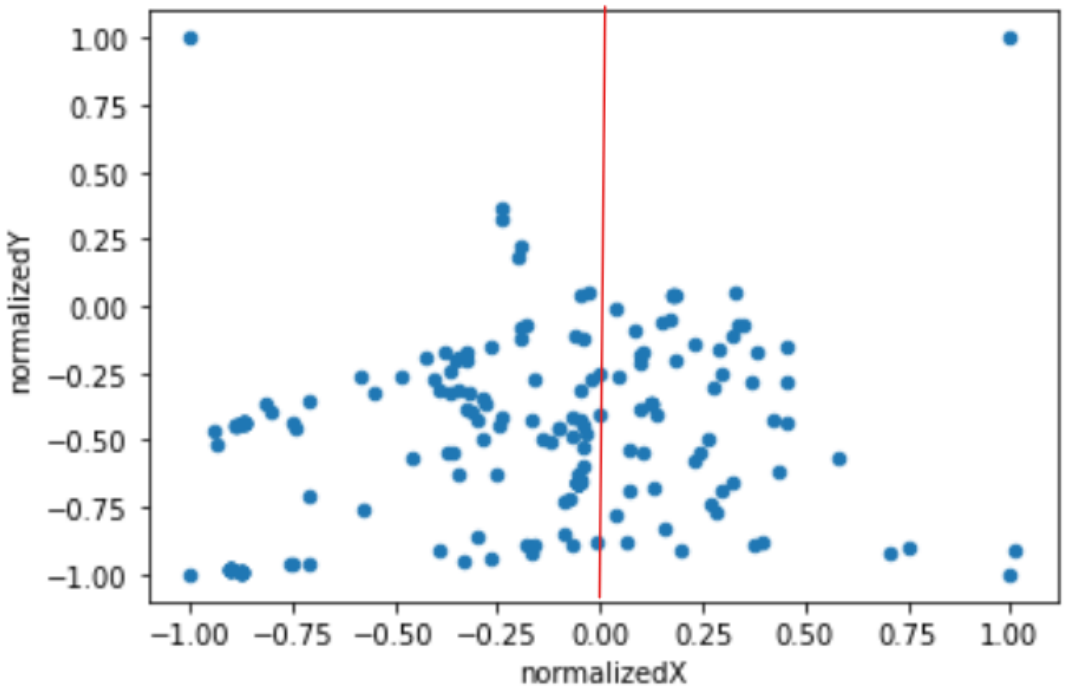

Figura 7.25: Plot della posizione del giocatore in fase di testing (circa 100 eventi catturati)

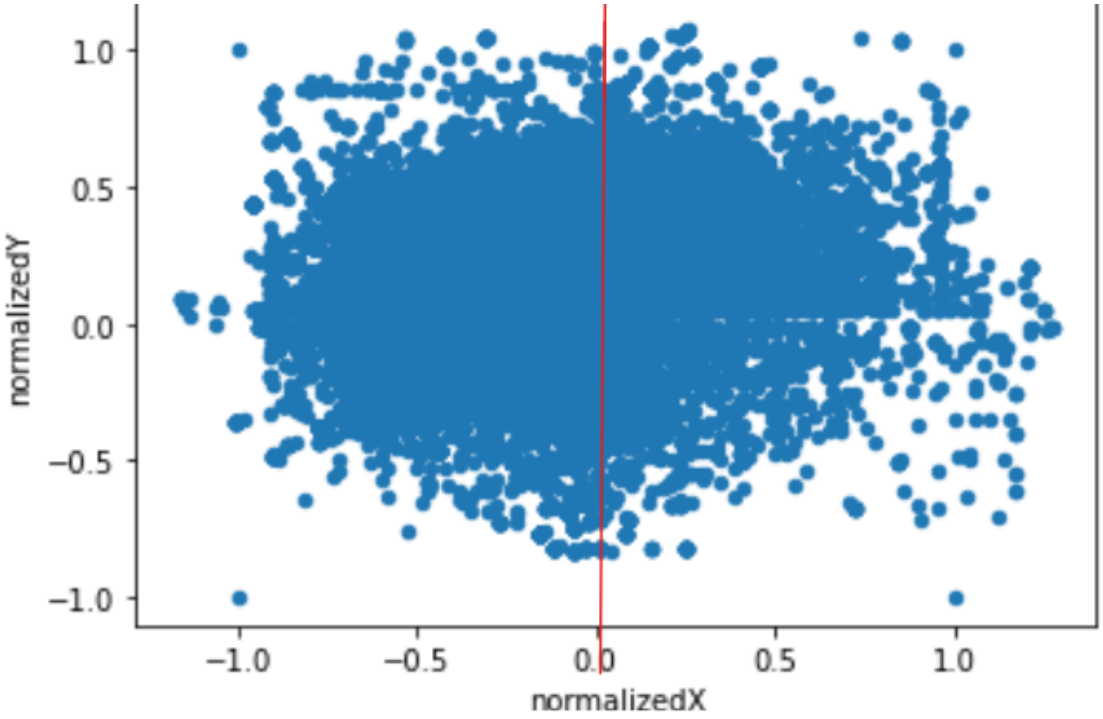

Figura 7.26: Plot della posizione del giocatore in fase avanzata dello sviluppo del gioco (circa 35000 eventi catturati)

Come si può notare dalla seconda figura, il grafico suggerisce che la mappa sia ben utilizzata dai giocatori, ma questo tipo di plot ha un limite, all'aumentare dei punti, tutta la mappa sembra utilizzata allo stesso modo. Per questo all'aumentare degli eventi si dovrebbe optare per un altro tipo di grafico.
Una scelta interessante potrebbe essere quella di utilizzare una heatmap, dove si visualizzano le aree della mappa percorse dagli utenti con colorazioni diverse pesate per la quantità di eventi che si possono trovare in quell'area specifica.
In questo modo, si avrebbero delle informazioni più parlanti rispetto alle aree non utilizzate.
Ma questo, è solo un piccolo esempio delle infinite possibilità che questa pipeline mette a disposizione.

7.4.2 Databricks

L'ultima fase dello sviluppo della pipeline è stato quello di proseguire con il paradigma Bronze-Silver-Gold layer.
La pipeline di analisi sfruttando PySpark è risultata essere una buona strategia, ma mantenere la computazione in locale è un grosso limite, sopratutto in ottica

di scalabilità, per questo, si è optato per Databricks, un framework basato sul paradigma lakehouse, che permette di sfruttare le potenzialità della programmazione a notebook, in un ambiente totalmente cloud.
Il codice utilizzato su databricks è molto simile a quello utilizzato in locale sfruttando i notebook di jupyter.
L'enorme differenza, oltre che l'utilizzo di un ambiente cloud, è quella di poter accedere ai vantaggi architetturali messi a disposizione dal paradigma lakehouse, creando una vera e propria pipeline che comprenda silver e gold layer.

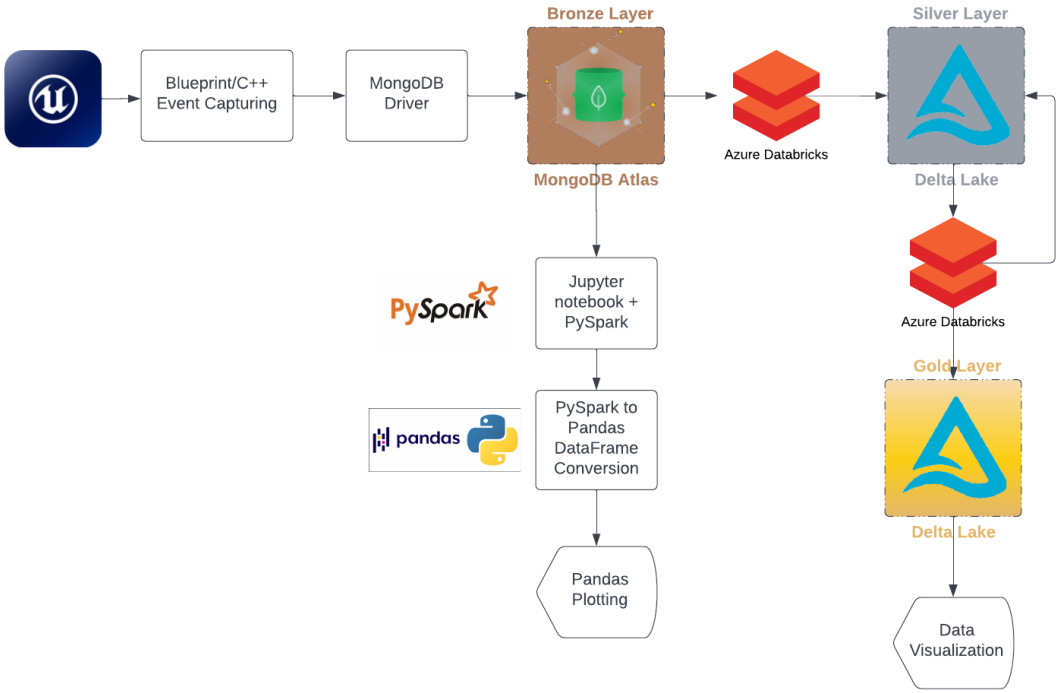

Figura 7.27: Pipeline Generale di acquisizione e immagazzinamento dati

Prendendo come esempio il caso precedente, i risultati intermedi ottenuti dall'estrazione dei valori di X ed Y, verrebbero immagazzinati sotto forma di delta table in databricks, nel cosi detto *Silver layer*.
Infine, i valori normalizzati verrebbero inseriti nel *Gold layer*, pronti per essere visualizzati.
Quest'ultima parte di progetto, è stata solo progettata e testata per pochissimi casi, come appunto, quello della posizione del giocatore.
Non è stato il focus dell'elaborato occuparsi nello specifico dell'analisi e la pulizia di tutti i dati, poichè questa fase risulta essere abbastanza meccanica.
Piuttosto, si è cercato di implementare tutta l'infrastruttura che permette al dato

di navigare tra vari sistemi e arrivare nelle mani di un'eventuale team di data scientists, che abbia lo scopo di manipolare i dati e trovarne le gemme nascoste.
Facendo uno Zoom-out dell'architettura si riesce a capire come si inseriscono i layer all'interno del prodotto videoludico sviluppato, e quale siano stati i punti di maggior interesse di questo capitolo:

1. **Cattura dei dati da Unreal**: dato che tutte le abilità sono state implementate in blueprint, e ogni attore coinvolto nella partita è un blueprint, anche la cattura dei dati è stata espletata tramite blueprint.

2. **Collegamento con MongoDB**: Una volta catturati i dati, è necessario instaurare una connessione con mongo DB per scrivere negli spazi appositi, questo viene fatto tramite C++.

3. **Collegamento con Databricks**: I dati immagazzinati su Mongo DB sono in uno stato molto grezzo, è stato sfruttato databricks per presentarli in uno stato migliore, qui, si è lavorato con *Python*, in particolare, tramite *PySpark*.

4. **Salvataggio dei dati in uno stadio intermedio sfruttando il paradigma delta lake**: Una volta puliti i dati possono essere salvati in uno stato migliore sfruttando l'architettura lakehouse che databricks mette a disposizione tramite le Delta table.

5. **Ulteriore trasformazione che porta al gold layer**: Il gold layer è uno dei tre layer che databricks prevede nel paradigma Lakehouse, scendiamo nel dettaglio più avanti.

Capitolo 8

Conclusioni

il presente lavoro ha cercato di illustrare un nuovo modo di cattura dei dati per un prodotto videoludico real-time in cloud, in grado di contenere i costi, e che consenta di lavorare in un ambiente flessibile, permettendo la modifica *on-the-fly* di eventuali informazioni non catturate come si vorrebbe.

8.1 Considerazioni generali

Durante l'elaborazione del progetto, si è tenuto conto di tutto il processo di sviluppo, dall'ideazione del videogioco, allo sviluppo dell'applicativo, all'implementazione della pipeline che catturasse quante più informazioni possibili.
Un obiettivo centrale è stato quello di mantenere i costi del lavoro molto bassi o pari a 0, molto spesso, sopratutto per sviluppatori indpendenti o studi di sviluppo molto piccoli, abbattere i costi è fondamentale per crescere, di conseguenza, sistemi "secondari" (come potrebbe essere la pipeline di dati), sono i primi a subire le conseguenze di lavorare a basso budget.
Nel mercato esistono già soluzioni che permettono di catturare le informazioni che si vuole, anche con una certa velocità, ma molto spesso sono soluzioni a pagamento, non inaccessibili, ma sicuramente scomode per quelle piccole realtà che devono fare i conti con la precarietà.
Inoltre, la flessibilità di utilizzo delle infrastrutture cloud presentate nel progetto, permette di adattarsi pienamente al paradigma di lavoro basato su sprint settimanali, ormai fondamentale per sopravvivere in un'ambiente dove è richiesto un continuo rinfrescamento del prodotto. Naturalmente, come la realtà insegna, la flessibilità a volte è scambiata con qualcos'altro, e in questo caso, è scambiata con la robustezza della soluzione.
Essendo una soluzione non ancora testata su grossi numeri, sarebbe necessario

testarla su grosse moli di dati dell'ordine delle centinaia di migliaia al giorno, se non, milioni al giorno, per decretarne la scalabilità e individuarne i punti deboli.

8.2 Sviluppi Futuri

L'architettura lakehouse offre molte funzionalità che spaziano dall'interrogazione dei dati tramite SQL, al collegamento con strumenti di visualizzazione, al supporto per il machine learning.
Un punto fondamentale dell'evoluzione di questa soluzione è sfruttare al massimo quanto detto, ad esempio: ogni mese, i dati elaborati e trasferiti nel gold layer, potrebbero essere immagazzinati in un data warehouse che faccia da memoria storica dei dati acquisiti.
Immaginando una situazione in cui un gioco di questo tipo venga rilasciato, per mantenere l'appeal, ogni dato spazio di tempo, sarà necessario introdurre nuove funzionalità o nuovi personaggi.
Questo si traduce con l'ingresso di nuove abilità ed eventi da catturare. Ogni volta che questo accadrà, sarà necessario gestirne la cattura dei dati, pur mantenendo una congruenza con quelli vecchi.
Perciò, potrebbe nascere la necessità di mantenere delle strutture dati pre e post ingresso della nuova feature. Tutti questi dati potrebbero essere utilizzati per creare delle dashboard di analisi del profilo dei giocatori.
In questo modo, ogni utente visualizzerà i suoi punti di forza e i suoi punti deboli, potrebbero essere forniti degli hints ai giocatori per permettergli di migliorare i punti deboli o di valorizzare quelli di forza.
Tutte queste statistiche possono essere utilizzate per la realizzazione di un sistema di matchmaking custom, senza doversi necessariamente agganciare a sistemi già implementati.
Insomma le possibilità che si aprono avendo alla base una struttura di questo tipo sono infinite, l'elaborato ha implementato solo un mattoncino di tutto quello che potrebbe venir fuori dall'implementazione di un sistema di cattura e analisi dei dati complesso.

Bibliografia

Sitografia

[1] *Scatter Plot*. URL: https://en.wikipedia.org/wiki/Scatter_plot.

[2] *What is mongoDB?* URL: https://www.techtarget.com/searchdatamanagement/definition/MongoDB.

[3] *GAS Documentation*. URL: https://github.com/tranek/GASDocumentation#concepts-at.

[4] *Data Lake definition*. URL: https://atlantic-technologies.com/en/blog/what-is-data-lake/.

[5] *The evolution of lakehouse paradigm*. URL: https://databricks.com/blog/2021/05/19/evolution-to-the-data-lakehouse.html.

[6] *Differences between data mart and data warehouse*. URL: https://www.geeksforgeeks.org/difference-between-data-warehouse-and-data-mart/.

[7] *data warehouse Architecture*. URL: https://www.geeksforgeeks.org/data-warehouse-architecture/.

[8] *ETL process*. URL: https://www.geeksforgeeks.org/etl-process-in-data-warehouse/.

[9] *What is a data warehouse?* URL: https://www.talend.com/resources/what-is-data-warehouse/.

[10] *Unreal Engine*. URL: https://docs.unrealengine.com/4.27/en-US/.

[11] *Ten great game telemetry reads*. URL: https://gameanalytics.com/news/10-great-game-telemetry-reads/.

[12] *Better Game Design Trhough Data*. URL: https://www.gamasutra.com/view/feature/2816/better_game_design_through_data_.php.

[13] *What is game telemetry?* URL: https://gameanalytics.com/blog/what-is-game-telemetry/.

[14] *MongoDB on Databricks*. URL: https://docs.databricks.com/data/data-sources/mongodb.html.

[15] *Introduction to pyspark*. URL: https://medium.com/the-researchers-guide/introduction-to-pyspark-a61f7217398e.