# Game Specifications

## A monitored multiplayer game for game security research and education

### Core Concepts

#### Game Type and Accessibility

- Open source Third Person Multiplayer Online Game
- Globally distributed/accessible
- Engaging gameplay to attract and retain players
- Informational / Raise Awareness

#### Unique gameplay mechanics: “Superpowers” as Cheats

- Standard third person multiplayer experience
- Players can activate different superpowers (cheats)
- These abilities presented are invoked from the game client

    - *Some abilities may be integrated within the game, while some may require hooks to external cheating programs*

### Cheat Responses/Visualization and Customization

- Clear visual feedback when activating abilities, distinguishing from normal gameplay
- Open-sourced and facilitate extensions to:
    - Improve the game by adding different levels and scenarios
    - Implement and test new cheats (superpowers) and potentially anti-cheat measures (shields)
    - Generate custom logs and datasets

### Comprehensive Logging System

- Detailed logging of all player actions, including standard gameplay, cheat activations, *and network traces*

## Technical Details

### Unreal Engine Networking Implementation

> 💡 To use different replication strategies at different scenarios

- RPCs (Remote Procedure Calls)
    - Types: Client (called on server, executed on client, e.g. updating UI elements), Server (called on client, executed on server, e.g. superpowers, shields, projectiles), NetMulticast (called on server, executed on every client, e.g. sounds, animations)
    - Reliability: Reliable (guaranteed, ordered, e.g. special abilities, projectiles) or Unreliable (faster, can be dropped, e.g. movements)
    - Used for discrete events or actions
- Replicated Properties

    - Continuously updated variables (e.g. movements received)
    - Conditional replication (e.g., owner-only (detailed stats of health), all clients (health))
- Relevancy
    - Determines which actors are replicated to specific clients
    - Can be always relevant, owner-only relevant, or distance-based
    - Optimizes network traffic

## Game Engine

- Unreal Engine (latest stable version)
- Key features to utilize:
    - Replicated Properties and Remote Procedure Calls (RPCs) for network synchronization
    - Blueprint system for visual scripting

## Network Architecture

- Core Client-Server model
- Hybrid execution:
    - Certain actions executed on the client side, then transmitted to the server (to replicate or validate)
        - Typically used for responsive behavior on the client side
        - Helps reduce perceived latency for players
    - Certain actions called from the client, but executed entirely on the server (RPCs)
        - Used for critical game logic, especially related to superpowers and shields
- Server remains the authoritative source of game state
- Careful balance required between client-side prediction and server-side validation

### Client Requirements

- Cross-Platform support:
  - Support for Windows, macOS, Linux and Consoles / Mobile Platforms in the long run
- Optimize for low hardware requirements to maximize accessibility

### Server Infrastructure

- For standard players:
  - For e.g. Cloud-based deployment using Amazon Web Services (AWS)

    > https://medium.com/@lukebrady105/building-a-multi-region-unreal-engine-server-fleet-in-aws-with-terraform-and-packer-fca19c22d1e8

  - Deployment of different game servers to minimize latency for global player base
- For researchers:
  - Ability to host a server in their own local network

## Logging in a Standard Third Person Gameplay

### Console Logging

All of these fields can be stored in a JSON format

- All the relevant game state data to be sent to the server and stored accordingly
- Player connections
  - Can be done with Unreal's PostLogin function:

```
// In ThirdPersonGameMode.cpp
AGameModeBase::PostLogin()
{
```

```
    Super::PostLogin(NewPlayer);

    FString PlayerUniqueID = NewPlayer->PlayerState->UniqueI
    UE_LOG(LogThirdPersonGameMode, Log, TEXT("Player connecte
                  *PlayerUniqueID);
}
```

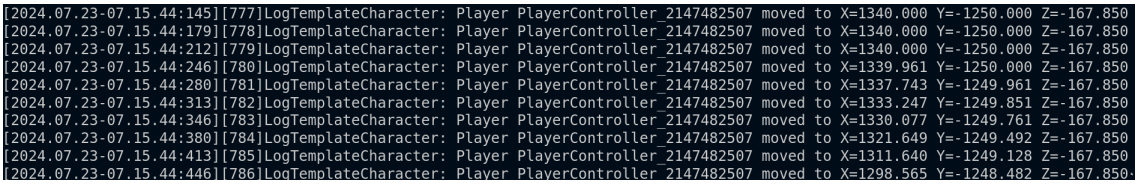
- e.g.


- Movement
  - For responsiveness, the client predicts actions locally and immediately sends an RPC to the server. The server then executes, validates, and replicates the action to other clients using replicated properties, ensuring consistent game state and logging the action for record-keeping
  - Replication is automatically set for basic movements in the Character class

```
// In ThirdPersonCharacter.h
virtual void Tick(float DeltaTime);
void LogMovement();
FString GetPlayerIdentifier() const;

// In ThirdPersonCharacter.cpp
void AThirdPersonCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (GetLocalRole() == ROLE_Authority)
    {
        LogMovement();
    }
}
```

```
void AThirdPersonCharacter::LogMovement()
{
    FVector CurrentLocation = GetActorLocation();
    UE_LOG(LogTemplateCharacter, Log, TEXT("Player %s moved t
           *GetPlayerIdentifier(), *CurrentLocation.ToString(
}

FString AThirdPersonCharacter::GetPlayerIdentifier() const
{
    APlayerController* Player = GetController<APlayerControll
    if (Player)
    {
        return Player->GetName();
    }
    return FString(TEXT("Unknown"));
}
```

- e.g



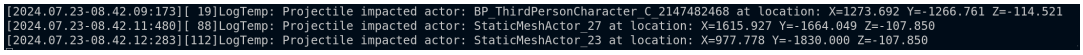

- Combat
  - Projectile Weapons:
    - These weapons are executed on the server. They spawn a separate actor which has travel time and can be affected by physics.

```
// Server-Side execution

// In ThirdPersonCharacter.h
UFUNCTION(Server, Reliable, WithValidation)
void HandleFire();
```

```
// In ThirdPersonProjectile.h
UFUNCTION(Category="Projectile")
void OnProjectileImpact(UPrimitiveComponent* HitComponent,
AActor* ImpactedActor;
FVector ImpactedLocation;

// In ThirdPersonCharacter.cpp
void AThirdPersonCharacter::HandleFire_Implementation()
{
    FVector spawnLocation = GetActorLocation() + (GetActor
    FRotator spawnRotation = GetActorRotation();
    spawnRotation.Yaw -= 3.0f;
    FActorSpawnParameters spawnParameters;
    spawnParameters.Instigator = GetInstigator();
    spawnParameters.Owner = this;
    AThirdPersonProjectile* spawnedProjectile = GetWorld()
}

//if need be (since weapon already executed on the server)
bool AThirdPersonCharacter::HandleFire_Validate()
{
    // Add any other validation checks here, e.g.
    // - Check if the character has enough ammo
    // - Check if the character is in a valid state to fi
    // - Check if the character is in a valid location


    return true;
    // if return false, player disconnected, change logic
    // Soft checking
}

// In ThirdPersonProjectile.cpp
void AThirdPersonProjectile::OnProjectileImpact(UPrimitive
{
```

```
    if (OtherActor)
    {
        ImpactedActor = OtherActor;
        // Use Hit.ImpactPoint to get the location of the
        ImpactedLocation = Hit.ImpactPoint;
        UGameplayStatics::ApplyPointDamage(OtherActor, Dar

        if (GetLocalRole() == ROLE_Authority)
        {
            UE_LOG(LogTemp, Log, TEXT("Projectile impacte
        }
    }

    Destroy();
}
```

- e.g.




- Hit-scan Weapons:
  - On the client side, these weapons use line tracing for instant hits. Bullet effects are simulated visually, but impacts are immediate. The server gets the information from the client and validates the action.
  - The logging and implementation is done similar to projectiles but using a line trace which takes impact instantly.

- Superpowers and Shields
  - Superpowers and Shields should be executed on the server, the logic of the hack could be called from different programs or software but should be implemented on the server side.

```
UFUNCTION(Server, Reliable, WithValidation)
void ActivateAbility(EAbilityType AbilityType);


void AThirdPersonCharacter::ActivateAbility_Implementatio
{
    switch (AbilityType)
    {
        case EAbilityType::Superpower:
            // Implement superpower logic here
            UE_LOG(LogTemp, Log, TEXT("Player %s activate
            break;

        case EAbilityType::Shield:
            // Implement shield logic here
            UE_LOG(LogTemp, Log, TEXT("Player %s activate
            break;

        default:
            UE_LOG(LogTemp, Warning, TEXT("Player %s atte
            break;
    }
}

FString AThirdPersonCharacter::GetPlayerIdentifier() cons
{
    if (APlayerState* PS = GetPlayerState())
    {
        return FString::Printf(TEXT("%d"), PS->GetPlayerI
    }
    return FString::Printf(TEXT("Unknown_%s"), *GetUnique
}
```

## JSON Formatted Logging

```
//Logger called on Server or on Client
//To see how client sends logs to server: check UE5_ToyExample F

void AThirdPersonCharacter::MainLogger()
{
        //Get current time
        double Time = HasAuthority() ? GetWorld()->GetTimeSecond

    //Get player ID
    FString PlayerID = GetPlayerIdentifier();

        //Get player location
    FVector PlayerLocation = GetActorLocation();

    //In the code check if it is server or client and set path c
    FString FilePath = HasAuthority() ? MainServerPath : MainCli

        //Get current health of player
        float CurrentHealthForLog = GetCurrentHealth();

        //If path exists; good, else create the directory with t
    FString DirectoryPath = FPaths::GetPath(FilePath);
    if (!FPlatformFileManager::Get().GetPlatformFile().Directory
    {
        FPlatformFileManager::Get().GetPlatformFile().CreateDire
    }

    FString FilePathString = FilePath;
    if (FilePath.IsEmpty())
    {
        UE_LOG(LogTemp, Error, TEXT("File path is empty!"));
        return;
    }

        //Make Json Objects for each field or subfield
```

```
    TSharedPtr<FJsonObject> JsonObject = MakeShareable(new FJso
    JsonObject->SetNumberField(TEXT("Time"), Time);
    JsonObject->SetStringField(TEXT("PlayerID"), PlayerID);
    JsonObject->SetNumberField(TEXT("PlayerLocationX"), PlayerLo
    JsonObject->SetNumberField(TEXT("PlayerLocationY"), PlayerLo
    JsonObject->SetNumberField(TEXT("PlayerLocationZ"), PlayerLo
    JsonObject->SetStringField(TEXT("Generated"), HasAuthority()

    //Conditional Fields
    //Checking if it is on client, if it is then also check if a
    if(!HasAuthority() && ArtDelay)
    {
        JsonObject->SetStringField(TEXT("NET.FD"), TEXT("ON"));
    }
    if(!HasAuthority() && LagSwitch)
    {
        JsonObject->SetNumberField(TEXT("Health"), CurrentHealth
        JsonObject->SetStringField(TEXT("NET.LS"), TEXT("ON"));
    }
    if(!HasAuthority() && DoS)
    {
        JsonObject->SetNumberField(TEXT("Health"), CurrentHealth
        JsonObject->SetStringField(TEXT("NET.DOS"), TEXT("ON"));
    }
    else if (HasAuthority() && DoS)
    {
        JsonObject->SetNumberField(TEXT("Health"), CurrentHealth
    }

    if(!HasAuthority() && AimBot)
    {
        JsonObject->SetStringField(TEXT("VIS.AIM"), TEXT("ON"));
    }

    //Creating the Json file
    FString JsonString;
```

```
    TSharedRef<TJsonWriter<>> JsonWriter = TJsonWriterFactory<>
    FJsonSerializer::Serialize(JsonObject.ToSharedRef(), JsonWri

    FString FileContent;
    FFileHelper::LoadFileToString(FileContent, *FilePathString)

    TArray<TSharedPtr<FJsonValue>> JsonArray;
    if (!FileContent.IsEmpty())
    {
        TSharedRef<TJsonReader<>> Reader = TJsonReaderFactory<>
        FJsonSerializer::Deserialize(Reader, JsonArray);
    }

    JsonArray.Add(MakeShareable(new FJsonValueObject(JsonObject)

    FString OutputString;
    TSharedRef<TJsonWriter<>> Writer = TJsonWriterFactory<>::Cre
    FJsonSerializer::Serialize(JsonArray, Writer);

    FFileHelper::SaveStringToFile(OutputString, *FilePathString)
    UE_LOG(LogTemp, Log, TEXT("Data successfully written to file
}
```

# Core Abilities and Specific Logging

## Categorization of Abilities

- NORM: Normal Actions
- NET: Network-Level Cheats
- INF: Information Exposure Cheats
- AIM: Aim Assistance Cheats
- BOT: Bot or Automation Cheats

## Logging Levels

- None:
  - Minimal logging, only capturing critical events or between high duration of periods
  - Logs activation and deactivation of abilities/cheats
  - One time logging of special abilities
- Moderate:
  - Balanced logging, capturing important events and periodic summaries
  - Logs activation/deactivation and periodic updates of ability usage
  - Periodic logging of normal gameplay
- Comprehensive:
  - Extensive logging, capturing detailed information about all events
  - Frequent logging of ability usage and effects
  - Regular, detailed logging of normal gameplay

💡 **Rationale for Logging Levels**

1. Hardware limitations:
   - Light logging ensures smooth gameplay on lower-end devices
2. Network constraints:
   - Reduced logging minimizes traffic for limited or unstable connections
3. Debug and Development:
   - Extensive logging aids debugging and balancing, with higher levels providing valuable player feedback

## Cheats, Observable effects and their Logs

- **Aimbots**

- **Execution:** Client-Side
- **Features**: Improved weapon accuracy, auto-target acquisition within range, enemy movement prediction
- **Effects**: High accuracy on moving targets, crosshair snapping, consistent precision shots
- **Visual Cues**:
  - Crosshair glows or changes color when the cheat is active
  - Smooth, unnatural tracking of moving targets
  - Rapid, precise snapping between multiple targets
- **Execution Details**:
  - Client activates the aimbot locally
  - Metric logging to server via RPCs (unreliable)
- **Logging Convention**:

```
{Timestamp, PlayerID, AIM,
                        {IsActive: bool,
                         TargetsAcquired: int,
                         NumHits: int,
                         Duration: float}}
```

  - *None*: Only logs IsActive (when activated/deactivated)
  - *Moderate*: Logs IsActive and TargetsAcquired every 5 seconds
  - *Comprehensive*: Logs all fields every 2 seconds

- **Game Bots**
  - **Execution**: Client-Side
  - **Features**: AI-controlled movement and action, efficiency in repetitive actions, rapid decision-making

- **Effects**: Smooth movement patterns, fast reaction times, optimal resource gathering
- **Visual Cues**:
  - Unnaturally smooth or optimized movement paths
  - Instantaneous reactions to game events
  - Perfect timing in resource collection or action sequences
- **Execution Details**:
  - AI logic runs on the client-side
  - Metric logging to server via RPCs (unreliable)
- **Logging Convention**:

```
{Timestamp, PlayerID, BOT,
                        {IsActive: bool,
                         ActionType: FString,
                         Location: FVector,
                         Duration: float}
```

  - *None*: Only logs IsActive (when activated/deactivated)
  - *Moderate*: Logs all fields every 20 seconds
  - Comprehensive: Logs all fields every 5 seconds, includes additional metrics like ResourcesGathered

- **Information Exposure**
  - **Execution**: Client-Side
  - **Features**: See enemy info (health, equipment), reveal map, see through obstacles
  - **Effects**: Visual overlays with extra info, highlighted entities through walls
  - **Visual Cues**:
    - Outlines of players/items visible through walls

        - HUD displays additional information not normally available
        - Minimap shows more details or larger area than usual
    - **Execution Details**:
        - Exposure logic runs on the client-side
        - Metric logging to server via RPCs (unreliable)
    - **Logging Convention**:

```
{Timestamp, PlayerID, INF,
                        {IsActive: bool,
                         ExposureType: FString,
                         EntitiesRevealed: int,
                         MapCoverage: float,
                         Duration: float}}
```

        - *None*: Only logs IsActive (when activated/deactivated)
        - *Moderate*: Logs IsActive and ExposureType every 10 seconds
        - *Comprehensive*: Logs all fields every 5 seconds

- **Fixed Delay**
    - **Execution**: Client-Side
    - **Features**: Add configurable delay to outgoing network packets (e.g., 1-5s)
    - **Effects**: Delayed action registration, advantageous in certain scenarios
    - **Visual Cues**:
        - Slight ghosting effect on player model
        - Delayed response to player inputs
    - **Execution Details**:
        - Client artificially introduces a delay in its outgoing packets
        - Client simulates normal actions locally

        - Delayed actions are sent after the specified time
        - Server replicates information to other clients
            - Other clients may be affected by certain actions that are already executed from attacker’s point of view and accepted by the server
        - Server-executed actions occur in real-time after delayed packets arrive
    - **Logging Convention**:

```
{Timestamp, PlayerID, NET.FixedDelay,
                        {IsActive: bool,
                         Delay: int,
                         AffectedActions: TArray<FString>}
```

        - *None*: Only logs IsActive
        - *Moderate*: No additional logging
        - *Comprehensive*: Logs all fields every 5 seconds

- **Lag Switch**
    - **Category**: NET
    - **Execution**: Client-Side Network Manipulation
    - **Features**: Temporarily disrupt network connection for configurable duration
    - **Effects**:
        - Cheating player's actions are delayed and then sent in bulk to the server
        - Other players appear to freeze or move in predictable patterns from the cheater's perspective
        - Cheater may appear to teleport or move erratically from other players' perspectives

        - Client-side actions (like hitscan weapon shots) may be accepted by the server even if the affected player moves away
    - **Visual Cues**:
        - For the cheating player:
            - Other players and objects appear to freeze in place
            - Smooth, uninterrupted movement and actions for the cheater
            - Sudden updates of other players' positions when connection resumes
        - For other players:
            - Cheating player may freeze briefly, then suddenly teleport or move rapidly
            - Unexpected hits from the cheating player, possibly from positions that seemed safe
    - **Execution Details**:
        - Client temporarily blocks outgoing network traffic
        - Client continues to simulate game locally, recording actions
        - Upon reestablishing connection, client sends batched updates to server
        - Server attempts to reconcile delayed state updates:
            - May accept certain actions (like hitscan weapon hits) if they pass basic validation
            - May reject or adjust movement if it exceeds plausible limits
            - Other clients receive delayed updates, causing the cheater to appear to teleport or move erratically
    - **Logging Convention**:

```
{Timestamp, PlayerID, NET.LagSwitch,
                        {IsActive: bool,
                         Duration: int,
```

```
                                ActionsBatched: int,
                                StateDiscrepancy: float}
```

    - *None*: Only logs IsActive (when activated/deactivated)
    - *Moderate*: Logs IsActive and Duration of attack
    - *Comprehensive*: Logs all fields after establishing reconnection

- **DoS (Denial of Service)**
    - **Execution**: Server-Side Simulation
    - **Features**: Target specific players, simulate network congestion, scalable attack intensity
    - **Effects**: Increased latency for targets, packet loss, potential disconnections
    - **Visual Cues**:
        - Target player's model appears to stutter or freeze
        - Delayed responses to target player's actions
    - **Execution Details**:
        - Client*A* (attacker) sends DoS request to server, specifying Client*B* (target)
        - Server verifies if Client*A* has necessary information to target Client*B* (e.g., PlayerID, IP)
        - If valid, server simulates DoS effects on Client*B*'s connection:
            1. Artificially increases latency for Client*B*
            2. Randomly drops packets sent to/from Client*B*
            3. In extreme cases, temporarily disconnects Client*B*
        - Server logs all DoS attempts and their effects
    - **Logging Convention**:

```
//From attacker_client to server
{Timestamp, AttackerID, NET.DoS,
                        {TargetID: FString,
                         Intensity: float}}

//From victim_client to server
{Timestamp, VictimID, NET.DoS,
                                                {PacketLos
                         Latency: float,
                         Duration: float}}
```

- *None*:
  - For attacker: Log all fields once when attack is initiated
  - For victim: Attempt to log all fields once when unusual network behavior is detected
- *Moderate*:
  - For attacker: No additional logging (server may infer ongoing attack from victim logs)
  - For victim: Attempt to log all fields every 15 seconds during the attack
- *Comprehensive*:
  - For attacker: No additional logging
  - For victim: Attempt to log all fields every 5 seconds during the attack

- ***..... (any additional cheats/shields)***

## Availability of abilities

- Abilities and their effectiveness may be available to players depending on various factors:

    - Level:
        - Unlocking newer abilities with progression
    - Provided category of Logs:
        - Attackers: Higher logging levels should grant access to more powerful abilities or enhance existing ones
        - Victims: Would be able to identify who launched an attack, what type of attack, whether they were targeted, etc.
    - *Platform*
        - *Certain platforms might have exclusive abilities or variations due to hardware capabilities or platform-specific features*

### Utilization of Logged Data

Manual analysis of the recorded logged data in a separate process i.e. data analysis

- Create heat maps of ability usage across game maps
- Track performance metrics before, during, and after ability usage
    - Measure key performance indicators (K/D ratio, objective completion rate)
    - Implement A/B testing for ability improvement
- Analyze server-client state discrepancies
    - Record instances of server-client desynchronization
    - Flag major discrepancies
- Measure impact on game balance and player performance
    - Track win rates and player scores in relation to ability usage
    - Implement a rating system for ability effectiveness

## Implementation Guidelines

### Activation System

- Clear method of activation for each ability

- UI for ability activation
    - e.g. in-game cheat menu or hotkeys (customizable)

## Balance and Fair Play

- Implement resource costs for ability activation
    - Design a dynamic cost system that adjusts based on usage frequency
    - Create strategic choices between ability use and resource conservation
- Create *fair play* zones where certain abilities are disabled
- Penalize Players when they use abilities frequently
    - Server based decision and Peer based decision
        - Cheater ability usage status
            - e.g. if the status is exhausted, penalize the player through deduction in level, XP, cheating abilities, etc
            - The status refreshes slowly as the cheats are not being used and reputation increases

## Game Environment Design

- Large, 1-2 level maps with mixed open and enclosed spaces
- Various cover types
    - e.g. walls, towers, obstacles
- Elevated positions and vantage points
- Hidden areas and shortcuts
- Interactive objects
    - e.g. buttons, levers, movable obstacles
- Hazardous elements
    - e.g. lava, traps
- Weather effects

    - e.g. wind affecting projectiles

## Gameplay Integration

- Fast-paced and strategic game modes
    - Design areas that benefit from hiding and capturing
        - e.g. information exposure in capture the flag, resource allocation modes
- Mix of hitscan and projectile weapons
    - e.g. hitscan is executed at client side, projectiles are executed at server-side
    - Hybrid weapons that change category based on distance
- Evasive techniques (crouch, jump, dodge)
- Scenarios favoring different ability usage:
    - Client-side weapon damage execution
        - Server accepts hit registration that are client-side executed
        - Design weapons that benefit from precise aim and timing
            - e.g. aimbots and client side manual network congestion on hitscan weapons
    - Server-side state validation / prioritization
        - Implement rollback mechanism that validates certain actions with previously known actions
            - e.g. network congestion on movement
    - Time-sensitive actions requiring server validation
        - Design quick-time events that test network performance
        - Create scenarios where timing is crucial
            - e.g. disarming bombs
- Both "Player vs Environment" and "Player vs Player" scenarios

- Puzzle-solving or parkour elements
- Resource gathering and management systems
    - e.g. game bots on harvestable areas that give resources