# Clash Royale

by

Paul Hodges

March 11, 2025

# Contents

# 1 Development Schedule

- **Week 1-2:** Research and System Architecture
  - Analysis of original Clash Royale mechanics
  - Server-client architecture design
  - Technology stack selection
  - Project management setup
- **Week 3:** Problem Definition and Requirements
  - Detailed problem description documentation
  - Functional and non-functional requirements specification
  - User stories and use case development
  - Feature prioritization
- **Week 4:** Core Algorithm Design
  - Pseudocode development for game logic
  - Network protocol specification
  - Data structures design
  - Database schema planning
- **Week 5:** Algorithm Testing and Validation
  - Trace table construction for logic verification
  - Edge case identification and handling
  - Logic refinement based on test results
  - Protocol simulation testing
- **Week 6-7:** Server Implementation
  - Network communication setup
  - Game state management implementation
  - Player authentication system
  - Card deployment and battle mechanics coding
- **Week 8:** Client Implementation and Integration
  - Client-side interface development
  - Server-client integration
  - Real-time gameplay testing
  - Bug fixing and optimization
- **Week 9:** Final Testing and Submission
  - Comprehensive system testing
  - Documentation completion
  - Part 1 submission preparation
  - Project demonstration setup

# 2 Problem Outline

The purpose of this project is to develop a simplified but functional online version of Clash Royale using Python. This real-time strategy game implementation will focus on core gameplay mechanics and network functionality, demonstrating practical software development skills in a gaming context.
The game will allow players to:

- Engage in real-time, online strategic battles against other players
- Build and customize card decks from their collection
- Progress through a skill-based arena system
- Earn rewards through a chest system to expand their card collection
- Experience the core strategic elements that make Clash Royale engaging

This implementation serves as both an educational tool for understanding game development principles and a demonstration of networking, database management, and software architecture skills in a cohesive application.

# 3 Problem Description

## 3.1 Game Overview

Clash Royale is a real-time strategy game where players deploy cards representing troops, spells, and buildings onto a battlefield to defeat their opponent. The game combines elements of collectible card games, tower defense, and multiplayer online battle arenas (MOBAs) into a unique gaming experience.

## 3.2 Game Objectives

The primary objective is to destroy more of the opponent's towers than they destroy of yours within the allotted battle time (typically 3 minutes, with a 2 minute overtime if tied). The game features:

- Three towers per player: two Princess Towers and one King Tower
- Destroying a Princess Tower grants one crown
- Destroying the King Tower grants three crowns and ends the match immediately
- The player with more crowns at the end of the time limit wins

## 3.3 Core Game Rules

- Each player begins with four cards from their eight-card deck
- Cards are deployed using elixir, a regenerating resource
- Elixir regenerates at a rate of 1 unit every 2.8 seconds (doubles in the first minute of overtime and then triples in the second minute)
- Maximum elixir capacity is 10 units
- Card costs range from 1 to 8 elixir
- After playing a card, it's replaced by another from the player's deck
- Cards can only be deployed on the player's side of the arena (with exceptions for spells and certain cards)
- Units automatically move toward and attack enemy units and towers based on their targeting preferences

## 3.4 Gameplay Flow

1. Players join a match through the matchmaking system
2. Both players start with the same amount of elixir
3. Players strategically deploy cards to attack opponent towers while defending their own
4. Units and buildings have specific health, damage, speed, and targeting parameters
5. Spells cause immediate effects (damage, slow, etc.) in their area of effect
6. Destroyed units are removed from the battlefield
7. The battle continues until a King Tower falls or time expires

## 3.5 Scoring System

- Destroying a Princess Tower: 1 crown
- Destroying the King Tower: 3 crowns (automatic victory)
- Most crowns at time expiration: Victory
- Equal crowns at time expiration: Draw
- Victory awards trophies that contribute to arena progression
- Defeat results in trophy loss

## 3.6 Winner Determination

- The player who destroys the opponent's King Tower instantly wins
- If no King Tower is destroyed within the time limit, the player with more crowns wins
- If both players have the same number of crowns when time expires, the match enters overtime
- In overtime, the first player to gain a crown advantage wins
- If neither player gains an advantage during overtime, the match ends in a draw

## 3.7 Advanced Game Mechanics and Strategies

- **Elixir Management:** Efficient use of elixir is critical; overspending creates vulnerability
- **Counter Deployment:** Specific cards counter others effectively (e.g., air units counter ground-only attackers)
- **Card Synergies:** Certain combinations of cards create powerful offensive or defensive capabilities
- **Lane Pressure:** Applying pressure in one lane to force defensive card usage, then attacking the other
- **Elixir Counting:** Tracking opponent's elixir to identify advantageous attack opportunities
- **Card Cycle Management:** Cycling through cards quickly to reach key cards in your deck
- **Tower Trading:** Strategic sacrifice of a tower to gain elixir advantage for a stronger counter-push
- **Spell Value:** Using spells to eliminate multiple units simultaneously for positive elixir trades
- **Unit Placement:** Precise placement affects unit interactions and pathing
- **Timing:** Deploying cards at optimal moments to maximize their effectiveness

# 4 Requirements

## 4.1 Functional Requirements

1. **FR1: Server-Client Architecture**
   - The system shall implement a client-server model supporting real-time multiplayer
   - The server shall handle game state synchronization across clients
   - The server shall manage concurrent games with minimal latency
   - The system shall handle connection drops and reconnections gracefully
2. **FR2: Card Management System**
   - Players shall be able to view their card collection
   - Players shall be able to build decks with exactly 8 cards
   - The system shall validate deck composition rules
   - The system shall implement card rarity and level mechanics
   - Players shall be able to upgrade cards when meeting requirements
3. **FR3: Real-time Battle Simulation**
   - The system shall manage battles in real-time
   - The system shall implement elixir generation mechanics
   - The system shall handle unit deployment, movement, and targeting
   - The system shall calculate damage, health, and tower destruction
   - The system shall implement the 3-minute standard time and overtime rules
4. **FR4: Progression System**
   - The system shall track player trophies
   - The system shall implement arena progression based on trophy count
   - The system shall match players based on similar trophy counts
   - Different arenas shall unlock different cards
5. **FR5: Reward System**
   - The system shall reward players with chests after victories
   - Chests shall contain cards and gold based on arena level
   - The system shall implement various chest types with different rewards
   - The system shall track chest unlocking timers
6. **FR6: Clan System**
   - Players shall be able to create and join clans
   - Clans shall support basic member management
   - The system shall implement a clan chat feature
   - The system shall display clan member activity
7. **FR7: User Authentication**
   - The system shall securely authenticate users
   - The system shall protect user credentials
   - The system shall manage login sessions with tokens
   - The system shall enforce password security requirements
8. **FR8: Database Integration**
   - The system shall persist all player data
   - The system shall record battle history
   - The system shall implement transaction safety for critical operations
   - The system shall backup data regularly

## 4.2 Non-Functional Requirements

1. **NFR1: Performance**
   - The system shall handle at least 100 concurrent connections
   - Battle simulations shall update at least 10 times per second
   - Server response time shall not exceed 200ms under normal load
   - Client frame rate shall maintain at least 30 FPS on target devices
2. **NFR2: Reliability**
   - The system shall have 99% uptime
   - The system shall recover from crashes without data loss
   - The system shall handle network instability gracefully
   - The system shall implement data integrity checks
3. **NFR3: Scalability**
   - The architecture shall support horizontal scaling
   - The database shall handle growing player data efficiently
   - The matchmaking algorithm shall scale with player population
4. **NFR4: Maintainability**
   - The code shall follow PEP 8 style guidelines
   - The system shall use modular design patterns
   - The codebase shall include comprehensive documentation
   - The system shall implement logging for debugging
5. **NFR5: Security**
   - User passwords shall be stored using strong hashing
   - Communication shall be encrypted
   - The system shall protect against common attack vectors
   - The system shall validate all client input

# 5 Design

## 5.1 Core System Design

The game will be built using a custom Entity Component System (ECS) engine and a custom network implementation. The visual style will be 2D with a simulated 3D look achieved through layered sprites and shadow rendering.

### 5.1.1 Engine

The engine is written in Python, based on the design off of one of my previous Rust engines, "Oxidized." Key features include:

#### 5.1.1.1 ECS Architecture

The Entity Component System (ECS) architecture separates data (components) from logic (systems). Entities are simple IDs, and components are data containers (e.g., Position, Velocity, Sprite). Systems operate on entities that have the required components. This design promotes data-oriented programming, improving cache efficiency and making the code more modular and maintainable. In this Python implementation, we'll leverage dictionaries and lists to store and manage entities, components, and systems. For example, a "MovementSystem" would iterate through entities with "Position" and "Velocity" components, updating their positions based on their velocities. This decoupling of data and logic allows for flexible entity composition and avoids the complexities of traditional inheritance-based object-oriented programming.

#### 5.1.1.2 2D/3D Cameras

The engine will support both orthographic and perspective cameras. Orthographic cameras provide a parallel projection, useful for classic 2D views where depth isn't emphasized. Perspective cameras simulate real-world vision, where objects appear smaller as they recede into the distance. This is crucial for creating the simulated 3D look. The camera implementation will likely involve a transformation matrix that converts world coordinates to screen coordinates. For perspective cameras, this matrix will include a perspective projection component, while orthographic cameras will use a simple scaling and translation matrix. We will have functions to easily switch between camera types, and to adjust the field of view of the perspective camera.

#### 5.1.1.3 GameObjects and Components

GameObjects serve as containers for components. By assigning different combinations of components to a GameObject, we can create a wide variety of entities with unique behaviors. For example, a "Player" GameObject might have "Position," "Velocity," "Sprite," "Input," and "Health" components. A "Tree" GameObject might have "Position" and "Sprite" components. This modularity allows for easy creation and modification of game entities without modifying core engine code. The component data will be stored in data structures that are easily accessible by the systems.

#### 5.1.1.4 Layered Sprite Rendering

To create the illusion of depth in a 2D environment, we'll use layered sprites. Sprites representing objects closer to the "camera" will be rendered on top of sprites representing objects further away. We'll implement a sorting mechanism based on the Y-coordinate (or a custom depth value) of the entities to determine the rendering order. This will allow for overlapping sprites to create a sense of depth. Furthermore, we will have the capability to assign a layer number to each sprite, allowing for more manual control over the draw order, and enabling the creation of backgrounds, midgrounds, and foregrounds.

#### 5.1.1.5 Rendering

The rendering for the python engine are not done using conventional methods like "pygame". I use a custom library that i created myself which uses SDL2 to draw to the screen and provides more complex methods for other things like textures, images and keyboard input. I create the python bindings (for the cpython module) using pybin11. The library is compiled, the binds are compiled and then the library is linked with the binds to produce a cpython library. The stub file for the LSP is then generated using stubgen, which is shipped with python.

### 5.1.2 Network

The network uses custom socket communication with binary data transmission:

#### 5.1.2.1 Custom Binary Protocol

Instead of relying on human-readable text-based protocols, we'll use a custom binary protocol. This protocol will define the structure of packets sent between the client and server, specifying the types and sizes of data fields. Binary protocols are more compact and efficient, reducing network overhead and latency. We will use python's struct module to pack and unpack data into binary format. We will define packet types, and each packet type will have a defined data structure.

#### 5.1.2.2 Client-Server Architecture

We'll implement a client-server architecture, where the server acts as the authoritative source of game state. Clients send input to the server, and the server processes the input, updates the game state, and sends updates back to the clients. This architecture helps prevent cheating and ensures consistency across all clients. The server will maintain the master copy of the game world, and the clients will maintain local copies that are kept in sync with the server.

#### 5.1.2.3 Asynchronous Sockets

To avoid blocking the main game loop, we'll use asynchronous sockets. Asynchronous sockets allow the game to continue running while waiting for network data. We will use python's asyncio library or similar, to handle the asynchronous network operations. This will prevent the game from freezing or becoming unresponsive during network communication.

#### 5.1.2.4 Packet Listener and Broadcaster

The server will have a packet listener to receive incoming packets from clients and a packet broadcaster to send packets to all connected clients. The listener will parse incoming packets and dispatch them to the appropriate game logic. The broadcaster will efficiently distribute game state updates to all clients, minimizing network traffic. We will implement packet queuing and throttling to prevent network congestion. The broadcaster will only send data that has changed, to reduce the amount of data sent.

### 5.1.3 2D with 3D Illusion

The 3D look is achieved through:

#### 5.1.3.1 Layered Sprites

As mentioned earlier, layered sprites are crucial for creating the illusion of depth. By rendering sprites in a specific order based on their perceived depth, we can simulate the effect of objects being in front of or behind each other. This will involve sorting sprites by their Y-coordinate or a custom depth value before rendering. We will use a depth buffer, or an equivalent sort algorithm, to ensure that sprites are drawn in the correct order.

#### 5.1.3.2 Shadow Rendering

Adding shadows to objects can significantly enhance depth perception. We'll implement shadow rendering by projecting a simplified silhouette of each object onto the ground. The darkness and length of the shadow will vary based on the object's height and the light source's position. This will give objects a sense of grounding and add visual depth to the scene. We will use sprite masks to create the shadow silhouettes.

#### 5.1.3.3 Perspective Scaling

To further enhance the 3D illusion, we'll implement perspective scaling. Objects farther away from the "camera" will appear smaller, while objects closer will appear larger. This will be achieved by adjusting the sprite's scale based on its distance from the camera. We will use a perspective projection matrix to calculate the scale factor. This will be done in conjunction with the Layered Sprite rendering, to create a believable 3D effect.

## 5.2 Algorithm Design

Below are the core algorithms for the main game systems. These algorithms use proper control structures and follow logical flow to handle the key game mechanics.

### 5.2.1 Server-Side Game Management

```
FUNCTION RunServer():
    INITIALIZE server
    INITIALIZE player_pool = []
    INITIALIZE active_games = []

    WHILE server_running:
        FOR each incoming_connection:
            IF is_new_connection THEN
                player = CreatePlayer(connection)
                player_pool.APPEND(player)
                SEND_WELCOME_MESSAGE(connection)
            ELSE IF is_matchmaking_request THEN
                player = GetPlayerFromConnection(connection)
```

```
                MatchmakingQueue.ADD(player)
            ELSE IF is_game_action THEN
                game = GetGameFromPlayer(connection)
                ProcessGameAction(game, connection, action)
            END IF
        END FOR

        # Matchmaking
        WHILE MatchmakingQueue.COUNT >= 2:
            player1 = MatchmakingQueue.GET_NEXT()
            player2 = MatchmakingQueue.FIND_OPPONENT(player1)
            IF player2 != NULL THEN
                game = CreateGame(player1, player2)
                active_games.APPEND(game)
                MatchmakingQueue.REMOVE(player2)
            ELSE
                MatchmakingQueue.RETURN(player1)
                BREAK
            END IF
        END WHILE

        # Update all active games
        FOR each game IN active_games:
            UpdateGame(game)
            IF game.is_finished THEN
                ProcessGameResults(game)
                active_games.REMOVE(game)
            END IF
        END FOR
    END WHILE
END FUNCTION
```

Listing 1: Main Server Loop

### 5.2.2 Card Deployment Process

```
FUNCTION ProcessCardDeployment(game, player, card_id, position_x, position_y):
    # Validate the deployment
    IF NOT IsValidDeployPosition(game, player, position_x, position_y) THEN
        SEND_ERROR(player, "Invalid deployment position")
        RETURN False
    END IF

    # Get the card from player's hand
    card = player.GetCardFromHand(card_id)
    IF card == NULL THEN
        SEND_ERROR(player, "Card not in hand")
        RETURN False
    END IF

    # Check elixir cost
    IF player.current_elixir < card.elixir_cost THEN
        SEND_ERROR(player, "Insufficient elixir")
        RETURN False
    END IF

    # Deploy the card
    player.current_elixir = player.current_elixir - card.elixir_cost
    player.RemoveCardFromHand(card_id)
    player.AddNextCardToHand()

    # Create the unit/spell/building on the field
    IF card.type == "TROOP" THEN
        unit = CreateTroop(card, position_x, position_y, player.side)
        game.AddUnit(unit)
    ELSE IF card.type == "SPELL" THEN
        ApplySpellEffect(game, card, position_x, position_y, player.side)
    ELSE IF card.type == "BUILDING" THEN
        building = CreateBuilding(card, position_x, position_y, player.side)
        game.AddBuilding(building)
    END IF

```

```
    # Notify both players
    BROADCAST_DEPLOYMENT(game, player.id, card_id, position_x, position_y)

    RETURN True
END FUNCTION
```

Listing 2: Card Deployment

### 5.2.3 Battle Update Logic

```
FUNCTION UpdateGame(game):
    # Update game time
    game.current_time = game.current_time + TICK_RATE

    # Check for game end conditions
    IF game.current_time >= game.max_time AND NOT game.in_overtime THEN
        IF game.player1.crowns == game.player2.crowns THEN
            game.in_overtime = True
            game.overtime_end = game.current_time + OVERTIME_DURATION
            BROADCAST_OVERTIME_START(game)
        ELSE
            game.is_finished = True
            RETURN
        END IF
    END IF

    IF game.in_overtime AND game.current_time >= game.overtime_end THEN
        game.is_finished = True
        RETURN
    END IF

    # Regenerate elixir for both players
    regeneration_amount = CALCULATE_ELIXIR_REGEN(game)
    game.player1.current_elixir = MIN(game.player1.current_elixir + regeneration_amount,
     MAX_ELIXIR)
    game.player2.current_elixir = MIN(game.player2.current_elixir + regeneration_amount,
     MAX_ELIXIR)

    # Update all units on the field
    FOR each unit IN game.units:
        IF unit.health <= 0 THEN
            game.RemoveUnit(unit)
            CONTINUE
        END IF

        # Find target for the unit
        IF unit.current_target == NULL OR unit.current_target.health <= 0 THEN
            unit.current_target = FindTarget(game, unit)
        END IF

        # Update unit position and attack
        IF unit.current_target != NULL THEN
            IF IsInRange(unit, unit.current_target) THEN
                PerformAttack(unit, unit.current_target)
            ELSE
                MoveTowards(unit, unit.current_target)
            END IF
        END IF
    END FOR

    # Update all buildings
    FOR each building IN game.buildings:
        IF building.health <= 0 OR building.lifetime <= 0 THEN
            game.RemoveBuilding(building)
            CONTINUE
        END IF

        building.lifetime = building.lifetime - TICK_RATE

        # Find target for the building
        IF building.can_attack THEN
```

```
            IF building.current_target == NULL OR building.current_target.health <= 0 THEN
                building.current_target = FindTarget(game, building)
            END IF

            # Attack if target in range
            IF building.current_target != NULL AND IsInRange(building, building.current_target) THEN
                PerformAttack(building, building.current_target)
            END IF
        END IF
    END FOR

    # Check tower status
    CheckTowerStatus(game)

    # Send updated game state to both players
    BROADCAST_GAME_STATE(game)
END FUNCTION
```

Listing 3: Battle Update Logic

### 5.2.4 Target Selection Algorithm

```
FUNCTION FindTarget(game, attacker):
    closest_target = NULL
    min_distance = INFINITY

    # Define potential targets based on attacker type
    potential_targets = []

    # Filter targets based on attacker's capabilities
    IF attacker.attacks_ground AND attacker.attacks_air THEN
        potential_targets = game.GetAllUnitsAndBuildings(GetOpponentSide(attacker.side))
    ELSE IF attacker.attacks_ground THEN
        potential_targets = game.GetGroundUnitsAndBuildings(GetOpponentSide(attacker.side))
    ELSE IF attacker.attacks_air THEN
        potential_targets = game.GetAirUnits(GetOpponentSide(attacker.side))
    END IF

    # Add towers to potential targets
    towers = game.GetTowers(GetOpponentSide(attacker.side))
    potential_targets.EXTEND(towers)

    # Apply targeting preference
    IF attacker.targeting_preference == "BUILDINGS" THEN
        buildings = [target FOR target IN potential_targets IF target.type == "BUILDING" OR target.type == "TOWER"]
        IF buildings.LENGTH > 0 THEN
            potential_targets = buildings
        END IF
    END IF

    # Find closest target
    FOR each target IN potential_targets:
        distance = CalculateDistance(attacker.position, target.position)
        IF distance < min_distance THEN
            min_distance = distance
            closest_target = target
        END IF
    END FOR

    RETURN closest_target
END FUNCTION
```

Listing 4: Target Finding

### 5.2.5 Chest Reward System

```
FUNCTION GenerateChestForPlayer(player, chest_type):
    # Create a new chest with appropriate type
    chest = CreateChest(chest_type)

    # Set unlock time based on chest type
    IF chest_type == "SILVER" THEN
        chest.unlock_time = 3 * HOURS
    ELSE IF chest_type == "GOLD" THEN
        chest.unlock_time = 8 * HOURS
    ELSE IF chest_type == "MAGICAL" THEN
        chest.unlock_time = 12 * HOURS
    ELSE IF chest_type == "GIANT" THEN
        chest.unlock_time = 16 * HOURS
    END IF

    # Determine gold amount based on arena and chest type
    chest.gold = CALCULATE_GOLD_REWARD(player.arena, chest_type)

    # Determine card counts based on arena and chest type
    common_count, rare_count, epic_count, legendary_count = CALCULATE_CARD_COUNTS(player
    .arena, chest_type)

    # Generate specific cards
    available_cards = GetAvailableCardsForArena(player.arena)

    FOR i = 1 TO common_count:
        card = SelectRandomCard(available_cards, "COMMON")
        chest.AddCard(card)
    END FOR

    FOR i = 1 TO rare_count:
        card = SelectRandomCard(available_cards, "RARE")
        chest.AddCard(card)
    END FOR

    FOR i = 1 TO epic_count:
        card = SelectRandomCard(available_cards, "EPIC")
        chest.AddCard(card)
    END FOR

    FOR i = 1 TO legendary_count:
        card = SelectRandomCard(available_cards, "LEGENDARY")
        chest.AddCard(card)
    END FOR

    # Add chest to player's inventory
    player.AddChest(chest)

    # Notify player
    SEND_CHEST_RECEIVED(player, chest)

    RETURN chest
END FUNCTION
```

Listing 5: Chest Generation