CannaVille Pro - Unreal Engine 5 Technical Design Document

1. Unreal Engine 5 Project Structure and Setup

This section outlines the recommended project structure and setup procedures for developing CannaVille Pro using Unreal Engine 5. A well-organized project is crucial for efficient development, scalability, and maintainability, especially for a complex application involving hyper-realistic 3D environments, interactive gameplay, and backend integrations.

1.1. Project Creation and Initial Configuration

To begin, a new Unreal Engine 5 project should be created. Given the visual fidelity requirements and the need for custom C++ logic for game mechanics, a C++ project template is recommended over a Blueprint-only project. This provides the flexibility and performance necessary for a hyper-realistic simulation.

Steps for Project Creation:

1. **Launch Unreal Engine 5:** Open the Epic Games Launcher and launch the desired Unreal Engine 5 version (e.g., 5.3 or later for optimal features like Lumen and Nanite).

2. **New Project:** Select the

   Games category, and then choose the **Blank** template. This provides a clean slate for custom development. 3. **Project Settings:** * **Blueprint or C++:** Select **C++**. This allows for robust backend logic, complex algorithms, and direct memory management, which are beneficial for performance-critical aspects of a simulation game. * **Target Platform:** Set to **Mobile/Tablet** from the outset. This ensures that the project is configured with mobile rendering features and optimizations in mind, preventing costly refactoring later. * **Quality Preset:** Choose **Scalable 3D or 2D**. While the goal is hyper-realism, starting with a scalable preset allows for better performance tuning across a wider range of mobile devices. High-end features can be enabled selectively. * **Starter Content:** Set to **No Starter Content**. This keeps the project lean and avoids unnecessary assets. * **Raytracing:** Disable Raytracing for mobile targets, as it is generally too performance-intensive for current mobile hardware. 4. **Project Name and Location:** Name the project `CannaVillePro` and choose a suitable local storage location.

1.2. Recommended Folder Structure

A logical and consistent folder structure is paramount for managing the vast number of assets, code files, and configurations within an Unreal Engine project. The following structure is recommended:

```
CannaVillePro/
├── Config/ (Engine and Project Configuration Files)
├── Content/
│   ├── Blueprints/ (Game-specific Blueprint assets)
│   │   ├── UI/ (User Interface Blueprints - Widgets, HUDs)
│   │   ├── GameModes/ (GameMode and GameState Blueprints)
│   │   ├── Characters/ (Player and NPC Character Blueprints)
│   │   └── Environments/ (Environment-specific Blueprints)
│   ├── Cpp/ (C++ Source Files - automatically managed by UE)
│   │   ├── CannaVillePro/ (Your main game module)
│   │   │   ├── Public/
│   │   │   └── Private/
│   │   └── ThirdParty/ (External C++ libraries)
│   ├── Materials/ (All Material assets)
│   │   ├── MasterMaterials/ (Base materials for reusability)
│   │   └── Instances/ (Material instances for variations)
│   ├── Meshes/ (All Static Mesh assets)
│   │   ├── Plants/ (Cannabis plant models - seedling, veg, flower, harvest)
│   │   ├── Hydroponics/ (DWC buckets, reservoirs, pipes)
│   │   ├── Environment/ (Tent, outdoor elements, props)
│   │   └── Characters/ (Player avatars, NPCs)
│   ├── Textures/ (All Texture assets)
│   │   ├── UI/ (UI textures, icons)
│   │   ├── Environment/ (Ground, walls, props textures)
│   │   ├── Plants/ (Plant-specific textures - albedo, normal, roughness, etc.)
│   │   └── Characters/ (Character textures)
│   ├── Sounds/ (Audio assets - SFX, music)
│   ├── Maps/ (Game levels/maps)
│   ├── UI/ (UI assets - fonts, images, etc.)
│   └── VFX/ (Visual Effects assets - particles, Niagara systems)
├── Saved/ (Automatically generated files - logs, caches, etc.)
├── Source/ (C++ Source Files - for your game module)
└── ... (Other standard UE project folders)
```

1.3. Version Control Integration

For collaborative development and robust project management, integrating with a version control system (VCS) is essential. Git is highly recommended due to its widespread adoption, flexibility, and strong community support. Perforce is another viable option, particularly for larger teams with significant binary asset management needs.

Git Setup Recommendations:

1. **Initialize Git Repository:** Initialize a Git repository in the root of the `CannaVillePro` project folder.
2. **`.gitignore` Configuration:** Utilize a comprehensive `.gitignore` file to exclude unnecessary Unreal Engine generated files (e.g., `Build`, `DerivedDataCache`, `Intermediate`, `Saved` folders, and temporary files). This keeps the repository clean and manageable. Epic Games provides a good starting point for `.gitignore` files for Unreal projects.
3. **Large File Storage (LFS):** For managing large binary assets (like `.uasset` files, `.glb` models, and high-resolution textures), Git LFS is indispensable. It stores large files outside the main Git repository, replacing them with pointers, which significantly improves cloning and fetching times.
    - Install Git LFS.
    - Track relevant file types: `git lfs track "*.uasset" "*.umap" "*.glb" "*.fbx" "*.png" "*.tga" "*.jpg"`
4. **Commit Frequency:** Encourage frequent, small commits with descriptive messages to facilitate easier tracking of changes and debugging.

1.4. Plugin Management

Unreal Engine's plugin system allows for extending functionality without modifying the engine source code directly. For CannaVille Pro, several plugins will be beneficial:

- **Datasmith:** For importing complex scenes and models from external DCC (Digital Content Creation) tools, ensuring fidelity of imported assets.
- **Modeling Tools Editor Mode:** Provides in-editor mesh editing capabilities, useful for quick adjustments and prototyping.

- **Niagara:** Unreal Engine's powerful visual effects system, crucial for creating realistic water effects, plant growth animations, and environmental particles.
- **Common UI:** A framework for building robust and scalable user interfaces, especially useful for complex game UIs that need to adapt to different screen sizes and input methods (mobile, desktop).
- **Substance 3D Plugin (Optional):** If using Substance Painter/Designer for texturing, this plugin streamlines the workflow for importing and managing Substance materials.
- **Third-Party Plugins for Mobile-Specific Features:** Research and integrate plugins for mobile-specific features like in-app purchases (for Stripe integration), push notifications, and platform-specific APIs.

1.5. Build Configuration for Mobile

Proper build configuration is critical for successful mobile deployment. This involves setting up Android SDK, NDK, JDK, and Xcode for iOS development.

Android Setup:

1. **Android Studio:** Install Android Studio, which includes the necessary SDK (Software Development Kit) and NDK (Native Development Kit).
2. **SDK Manager:** Use Android Studio's SDK Manager to install specific SDK Platforms (e.g., Android 10, 11, 12) and SDK Tools (e.g., Android SDK Build-Tools, Android SDK Platform-Tools).
3. **JDK:** Ensure a compatible Java Development Kit (JDK) is installed and configured. OpenJDK 11 is generally recommended for UE5.
4. **Unreal Engine Project Settings:**
    - **Platforms > Android:** Configure SDK locations, NDK API level, and enable support for Vulkan or OpenGL ES 3.1 rendering.
    - **Android SDK:** Point to the correct SDK, NDK, and JDK paths.
    - **APK Packaging:** Set up package name, version, and signing keys.

iOS Setup:

1. **Xcode:** Install Xcode on a macOS machine. Xcode includes the iOS SDK and necessary command-line tools.

2. **Provisioning Profiles and Certificates:** Set up valid Apple Developer Program provisioning profiles and signing certificates for your application.
3. **Unreal Engine Project Settings:**
    - **Platforms > iOS:** Configure bundle identifier, version, and signing information.
    - **Mobile Provision:** Select the appropriate provisioning profile and certificate.

This foundational setup ensures that the CannaVille Pro project is robust, organized, and ready for the complex development tasks ahead, particularly focusing on the demands of hyper-realistic 3D rendering and mobile platform compatibility.

2. Asset Integration and Optimization

Achieving hyper-realistic 3D environments and characters in CannaVille Pro, especially for mobile platforms, necessitates a meticulous approach to asset integration and optimization. This section details the strategies and techniques to ensure visual fidelity while maintaining optimal performance.

2.1. Asset Pipeline for Hyper-Realism

The asset pipeline defines the workflow from content creation in Digital Content Creation (DCC) tools (like Blender, Maya, ZBrush) to their final integration and rendering within Unreal Engine 5. For CannaVille Pro, the focus is on photogrammetry-level detail and efficient asset delivery.

Recommended Pipeline:

1. **High-Poly Sculpting (ZBrush/Blender):** Create highly detailed models for plants, DWC components, environmental props, and character avatars. These models will serve as the source for normal map baking and high-fidelity renders.
2. **Retopology and UV Unwrapping (Blender/Maya):** Generate optimized low-poly meshes from the high-poly sculpts. This step is crucial for performance. Efficient UV unwrapping ensures proper texture mapping and minimizes distortion.
3. **Texturing (Substance Painter/Designer):** Utilize PBR (Physically Based Rendering) workflows to create realistic materials. This involves generating Albedo (Base Color), Normal, Roughness, Metallic, Ambient Occlusion, and

potentially Height maps. AI-generated textures, as previously discussed, can be integrated here for unique variations.

4. **Rigging and Animation (Maya/Blender):** For animated assets like growing plants or character avatars, create skeletal rigs and animations. This includes subtle plant movements (e.g., swaying in a breeze) and realistic avatar locomotion.
5. **Export to GLB/FBX:** Export assets from DCC tools in formats compatible with Unreal Engine. While FBX is a traditional choice, GLB (GL Transmission Format) is gaining traction, especially for web and mobile, due to its efficiency and PBR material support. Unreal Engine can import both.
6. **Unreal Engine Import:** Import the optimized meshes, textures, and animations into Unreal Engine 5. Ensure proper import settings are used, especially for normal maps (Tangent Space) and texture compression.

2.2. Level of Detail (LOD) Implementation

LODs are critical for performance optimization in 3D applications, particularly on mobile devices. They allow the engine to render simpler versions of a mesh when it is further away from the camera, reducing polygon count and draw calls.

Unreal Engine 5 LOD System:

Unreal Engine 5 provides robust automatic and manual LOD generation. For CannaVille Pro, a combination of both will be used to achieve optimal results.

1. **Automatic LOD Generation:** For most static meshes (e.g., DWC buckets, tent walls, soil holes), Unreal Engine can automatically generate LODs based on screen size percentages. This is a quick way to get initial optimization.
2. **Manual LOD Creation:** For critical assets like the cannabis plants and character avatars, manual LOD creation in DCC tools is recommended. This allows for more precise control over polygon reduction and ensures that important silhouette details are preserved at lower LODs. The user-provided GLB models, which include `_LOD0`, `_LOD1`, `_LOD2` suffixes, are perfectly suited for this approach.
    - **LOD0 (High Detail):** Used when the asset is close to the camera. Target polygon count: < 25,000 triangles for hero objects (e.g., a single cannabis plant).

  - **LOD1 (Medium Detail):** Used at mid-distances. Target polygon count: < 8,000 triangles.
  - **LOD2 (Low Detail):** Used at far distances. Target polygon count: < 1,000 triangles.

Implementation in UE5:

- Import each LOD level as a separate static mesh. In the Static Mesh Editor, assign these meshes to their respective LOD slots. Define the screen size thresholds at which each LOD switches.
- For animated meshes (Skeletal Meshes), similar LOD principles apply, but they are managed within the Skeletal Mesh Editor.

2.3. Material and Texture Optimization

Materials and textures significantly impact visual quality and performance. Proper optimization is key to achieving hyper-realism without sacrificing frame rate.

1. **PBR Materials:** All materials will adhere to the PBR workflow, ensuring consistent and realistic lighting responses. Master Materials will be created to define common properties (e.g., metallic, roughness, normal map application) and then instanced for specific assets, allowing for efficient material variations.
2. **Texture Resolution and Compression:**
   - Use appropriate texture resolutions (e.g., 2K or 4K for hero assets, 1K or 512 for smaller props). Avoid unnecessarily high resolutions.
   - Utilize Unreal Engine's texture compression settings. For mobile, ASTC (Adaptive Scalable Texture Compression) is the preferred format for Android, and PVRTC (PowerVR Texture Compression) or ETC2 for iOS, as they offer good quality at small file sizes.
   - Combine textures into atlases where possible (e.g., grass and soil textures into a 2K atlas) to reduce draw calls.
3. **Shader Complexity:** Keep shaders as simple as possible while achieving the desired visual effect. Avoid complex shader nodes that are computationally expensive. Utilize Material Functions for reusable shader logic.
4. **Texture Streaming:** Enable texture streaming to load textures into memory only when needed, reducing initial memory footprint and load times.

2.4. Integration of User-Provided GLB Models

The provided GLB models (`cannabis_plant_small_1.glb`, `cannabis_sativa_plant.glb`, `small_cannabis_plant.glb`, `cannabis_plant.glb`) are valuable starting points. These will be integrated and optimized as follows:

1. **Import:** Import the GLB files directly into Unreal Engine 5. UE5 has native support for GLB, which simplifies the import process.
2. **LOD Assignment:** If the GLB models contain multiple LODs (as indicated by `_LOD0`, `_LOD1`, `_LOD2` naming conventions), ensure they are correctly assigned to the respective LOD slots within the Static Mesh Editor.
3. **Material Conversion:** Verify that the imported materials are correctly converted to Unreal Engine's PBR material system. Adjust parameters (e.g., roughness, metallic) as needed to match the desired visual quality.
4. **Skeletal Mesh Conversion (if applicable):** If any of the plant models are intended to be animated (e.g., for growth stages or wind effects), they will need to be imported as Skeletal Meshes and rigged within Unreal Engine or a DCC tool.
5. **Growth Stages:** The different GLB models will be used to represent various growth stages (seedling, vegetative, flowering, harvest). This will involve swapping out the mesh or blending between different meshes/materials based on the plant's growth progress in the game logic.

2.5. Environment Specific Assets

Indoor DWC System:

- **DWC Buckets:** Optimized models (3k/1k/300 tris for LOD0/1/2) with PBR materials for plastic and water surfaces.
- **Central Reservoir & Pump Manifold:** Detailed models (5k/2k/500 tris) with realistic textures for metal and plastic.
- **PVC Pipe Network:** Modular pipe sections (1k/400/200 tris) that can be assembled in-engine. A trim sheet texture will be used for efficiency.
- **LED Quantum Boards:** Detailed models (4k/1.5k/300 tris) with emissive materials for the LEDs and realistic metal textures for the heatsink.
- **Mylar Tent Shell:** Simple primitive boxes with a highly reflective PBR material to simulate the mylar surface.

Outdoor Grid:

- **Outdoor Grass Tile:** A 1m x 1m plane with an AI-generated seamless texture. This texture will be part of a 2K atlas combining grass and soil.
- **Soil Hole Prefab:** Optimized mesh (500/150/50 tris) with realistic soil textures. Instances of this prefab will be placed across the 5x5 grid.

By following these asset integration and optimization strategies, CannaVille Pro will achieve the desired hyper-realistic visual quality while ensuring smooth performance across target mobile devices.

3. Core Gameplay and UI Design

This section details the core gameplay mechanics and user interface (UI) design for CannaVille Pro within Unreal Engine 5. The primary focus is on creating an immersive, interactive 3D environment where players can freely navigate as an avatar, inspect plants, and perform various cultivation tasks. The UI will be designed for mobile-first responsiveness and intuitive interaction.

3.1. Immersive 3D Environment and Avatar Control

The user explicitly requested a "3D environment you step into" and the ability to "walk around with avatar and inspect the plants to tend to them." This necessitates a robust first-person or third-person character controller and a well-designed level layout.

3.1.1. Character Controller Implementation:

Unreal Engine 5 provides a highly customizable Character class, which is ideal for implementing player avatars. The Character class comes with built-in movement components (CharacterMovementComponent) that handle locomotion, jumping, and falling, making it suitable for both first-person and third-person perspectives.

- **Blueprint or C++:** For core movement logic and performance, the Character class will be implemented primarily in C++, with Blueprint extensions for visual customization (e.g., avatar selection, clothing) and specific gameplay interactions.
- **Input System:** Utilize Unreal Engine 5's Enhanced Input System for robust and flexible input mapping across various platforms (mobile touch, gamepad,

keyboard/mouse). This allows for easy remapping of controls and ensures a consistent experience.

  - **Mobile Touch Controls:** Implement virtual joysticks for movement and camera control. Swipe gestures will be used for quick camera adjustments, and pinch-to-zoom for closer inspection of plants.
  - **Desktop Controls:** Standard WASD for movement and mouse for camera look.

- **Camera System:**
  - **First-Person Perspective (Default):** The camera will be attached to the character's head, providing an immersive

3.2. Technology Stack for Hyper-Real Interactive Game App

To achieve the desired hyper-realistic visuals, interactive gameplay, and efficient development workflow for CannaVille Pro, the following technology stack will be utilized:

Purpose	Tool
Engine (Core)	Unreal Engine 5
Asset Creation	Blender
Animations	Blender / Mixamo
Sound + FX	FMOD, Quixel Megascans
Scripting	Unreal Blueprints or C++
Platform Build	Android / iOS via Unreal
Publishing	Play Store, App Store

Pro Tip: Leveraging Quixel Megascans and MetaHuman Creator for Unprecedented Realism

For achieving movie-quality environments and hyper-realistic animated characters, two powerful tools integrated with Unreal Engine 5 will be extensively utilized:

- **Quixel Megascans:** This vast library of scanned real-world assets (3D models, surfaces, atlases, decals) provides unparalleled photorealism. By integrating

Megascans directly into Unreal Engine, environments can be populated with incredibly detailed rocks, foliage, soil, and other natural elements, significantly reducing asset creation time while boosting visual fidelity. This is particularly beneficial for the outdoor cultivation environments and adding intricate details to the indoor grow spaces.

- **MetaHuman Creator:** For the hyper-realistic customer avatars, MetaHuman Creator offers a revolutionary solution. This cloud-based application allows for the creation of incredibly lifelike digital humans with full facial motion capture capabilities. Integrating MetaHumans into CannaVille Pro will ensure that player avatars and potential NPCs (Non-Player Characters) exhibit a level of realism that enhances immersion and user engagement, aligning perfectly with the user's vision for

  "hyper-real customer avatars that are also hyper real."

3.3. Interactive Gameplay Mechanics

CannaVille Pro will offer a rich set of interactive gameplay mechanics that allow players to manage and nurture their cannabis plants from seed to harvest. The interactions will be intuitive and visually engaging, leveraging Unreal Engine 5's capabilities for dynamic object interaction and visual feedback.

3.3.1. Plant Interaction and Inspection:

Players will be able to interact with individual plants to perform various actions and inspect their health and growth progress. This will be achieved through a combination of raycasting (for selecting plants) and context-sensitive UI elements.

- **Selection:** When the player's avatar looks at or approaches a plant, a visual highlight will appear, indicating it is interactive. Pressing a designated interaction key (e.g., 'E' on keyboard, a dedicated button on mobile UI) will select the plant.

- **Inspection Mode:** Upon selection, the camera will smoothly transition to a closer view of the plant, allowing for detailed visual inspection. A dedicated

  UI panel will appear, displaying critical information such as: * **Plant Health Score:** A visual bar indicating overall health. * **Growth Stage:** Seedling, Vegetative, Flowering, Harvest. * **Environmental Needs:** Current light, temperature, humidity levels, and optimal ranges. * **Nutrient Levels:** Macro and micronutrient status. * **Pest/Disease Status:** Identification of any issues. * **Action Prompts:** Contextual buttons for available actions (e.g., "Water," "Apply Nutrients," "Harvest").

3.3.2. Cultivation Actions:

Players will perform various actions to tend to their plants, each with visual and auditory feedback.

- **Watering:** Players will use a watering can tool. A visual effect of water being poured and absorbed by the soil/medium will be rendered. The plant's hydration level will update, affecting its health and growth.
- **Applying Nutrients:** Different nutrient solutions will be available. Applying them will trigger visual effects (e.g., a subtle glow around the plant) and update nutrient levels. Incorrect nutrient application can lead to negative effects.
- **Pest Control:** Tools like neem oil spray or ladybugs will be used to combat pests. Visual effects of pests disappearing or ladybugs appearing will be implemented.
- **Pruning/Training:** Players can prune leaves or train branches to optimize growth. This will involve precise interaction with specific parts of the plant model.
- **Harvesting:** Once a plant reaches the harvest stage, a dedicated action will allow players to collect their yield. This will trigger a satisfying visual and auditory sequence, and the harvested product will be added to the player's inventory.

3.4. User Interface (UI) Design

The UI will be designed using Unreal Engine 5's UMG (Unreal Motion Graphics) UI Designer, focusing on a clean, intuitive, and mobile-first approach. The UI will provide essential information and controls without cluttering the immersive 3D view.

3.4.1. HUD (Heads-Up Display):

The main HUD will display critical, always-visible information:

- **Player Stats:** Money, Energy, Level, Day (as seen in the provided reference images).
- **Environment Toggles:** Buttons to switch between Indoor DWC and Outdoor environments.

- **Tool/Action Bar:** A dynamic bar at the bottom of the screen (similar to the provided image) that changes based on context (e.g., general tools, plant-specific actions).

3.4.2. Contextual UI Panels:

These panels will appear when specific interactions occur (e.g., inspecting a plant, opening the shop).

- **Plant Analysis Panel:** (As seen in the provided image) Displays detailed plant health, environmental settings (light, temperature, humidity sliders), and other relevant data. This panel will be designed to be responsive and movable on mobile devices.
- **Inventory/Shop Panel:** Allows players to manage harvested products, purchase new seeds, tools, nutrients, and other upgrades. This will integrate with the Stripe backend for real-money transactions.
- **Settings Panel:** For game options, audio, graphics, and control customization.

3.4.3. Mobile-First Responsiveness:

All UI elements will be designed with mobile screen sizes and touch input in mind. This includes:

- **Scalable Widgets:** UMG widgets will use anchors and size boxes to ensure they scale correctly across different resolutions and aspect ratios.
- **Touch-Friendly Buttons:** Buttons will be large enough for easy touch interaction.
- **Virtual Joysticks/Pads:** On-screen controls for movement and camera look.
- **Swipe Gestures:** For navigation within UI panels or camera manipulation.

3.5. Visual Feedback and Effects

To enhance immersion and provide clear feedback, various visual effects will be implemented:

- **Plant Growth Visuals:** Plants will visibly change and grow through their stages, with subtle animations for new leaves, buds, and overall size increase.
- **Health Indicators:** Visual cues on plants (e.g., wilting, discoloration) will indicate health issues, complementing the UI health score.

- **Tool Effects:** Visual effects for watering (water particles), nutrient application (subtle glow), and pest control (disappearing pests).
- **Harvest Animation:** A satisfying animation and particle effect when a plant is harvested.
- **Environmental Changes:** Dynamic lighting (day/night cycle), weather effects (rain, sun), and environmental particle systems (dust motes in indoor tent, pollen in outdoor). Niagara will be heavily utilized for these effects.

By combining robust character control, intuitive interactive mechanics, and a responsive UI, CannaVille Pro will provide an engaging and immersive cultivation experience within its hyper-realistic 3D environments.

4. Backend Integration (Stripe, Security, Database)

This section outlines the architecture and implementation details for integrating the Unreal Engine 5 frontend of CannaVille Pro with a robust Flask backend. The backend will handle critical functionalities such as user authentication, game state management, and secure payment processing via Stripe. Emphasis will be placed on data security and efficient communication between the client and server.

4.1. Backend Architecture Overview

The backend will be built using Flask, a lightweight Python web framework, due to its flexibility and suitability for building RESTful APIs. SQLAlchemy will be used as the ORM (Object-Relational Mapper) for database interactions, providing an abstraction layer over the chosen database (e.g., PostgreSQL for production, SQLite for development).

Key Backend Components:

- **Flask Application:** The core web application handling API requests.
- **Database:** Stores user accounts, game progress, inventory, and other persistent data.
- **Authentication/Authorization Module:** Manages user registration, login, and access control.

- **Stripe Integration Module:** Handles payment processing, subscriptions, and in-app purchases.
- **Game Logic Module:** Processes game state updates, plant growth calculations, and environmental effects that require server-side validation or persistence.

4.2. Database Design

The database will store essential user and game-related information. The following tables are proposed:

4.2.1. `Users` Table:

Field Name	Data Type	Description	Constraints
`id`	INTEGER	Primary Key, Unique User ID	PRIMARY KEY, AUTOINCREMENT
`username`	TEXT	Unique username for login	UNIQUE, NOT NULL
`email`	TEXT	User's email address	UNIQUE, NOT NULL
`password_hash`	TEXT	Hashed password for security	NOT NULL
`created_at`	DATETIME	Timestamp of user registration	DEFAULT CURRENT_TIMESTAMP
`last_login`	DATETIME	Timestamp of last user login	
`stripe_customer_id`	TEXT	Stripe Customer ID for payment management	UNIQUE

4.2.2. `GameStates` Table:

| Field Name | Data Type | Description | Constraints | | :------------ | :-------- | :-- | :----------------- |\n| `id` | INTEGER | Primary Key, Unique Game State ID | PRIMARY KEY, AUTOINCREMENT | | `user_id` | INTEGER | Foreign Key to `Users` table | NOT NULL, FOREIGN KEY (`user_id`) REFERENCES `Users`(`id`) | | `current_money` | REAL | Player's current in-game currency | DEFAULT 1000.0 | | `current_energy` |

INTEGER | Player's current energy points | DEFAULT 100 | | `current_level` | INTEGER | Player's current game level | DEFAULT 1 | | `current_day` | INTEGER | Current in-game day | DEFAULT 1 | | `inventory` | JSON | JSON blob storing player's inventory (seeds, tools, etc.) | | | `indoor_layout` | JSON | JSON blob storing indoor grow environment state | | | `outdoor_layout` | JSON | JSON blob storing outdoor grow environment state | | | `last_updated` | DATETIME | Timestamp of last game state save | DEFAULT CURRENT_TIMESTAMP |

4.2.3. `Plants` Table (Detailed Plant Data):

Field Name	Data Type	Description	Constraints
`id`	INTEGER	Primary Key, Unique Plant ID	PRIMARY KEY, AUTOINCREMENT
`game_state_id`	INTEGER	Foreign Key to `GameStates` table	NOT NULL, FOREIGN KEY (`game_state_id`) REFERENCES `GameStates` (`id`)
`plant_type`	TEXT	Type of cannabis strain	NOT NULL
`growth_stage`	TEXT	Current growth stage (seedling, veg, flower, harvest)	NOT NULL
`health_score`	REAL	Plant's health percentage (0-100)	DEFAULT 100.0
`hydration`	REAL	Hydration level (0-100)	DEFAULT 100.0
`nutrients`	JSON	JSON blob for nutrient levels	
`pests`	JSON	JSON blob for pest/disease status	
`position_x`	REAL	X-coordinate in 3D space	NOT NULL
`position_y`	REAL	Y-coordinate in 3D space	NOT NULL
`position_z`	REAL	Z-coordinate in 3D space	NOT NULL
`environment`	TEXT	'indoor' or 'outdoor'	NOT NULL
`planted_at`	DATETIME	Timestamp when plant was planted	DEFAULT CURRENT_TIMESTAMP

4.3. API Endpoints

The Flask backend will expose a set of RESTful API endpoints for communication with the Unreal Engine 5 frontend. All API communication will be secured using HTTPS.

4.3.1. User Authentication Endpoints:

- `POST /api/register`: User registration. Requires `username`, `email`, `password`.
- `POST /api/login`: User login. Requires `username` (or `email`), `password`. Returns JWT (JSON Web Token) for subsequent authenticated requests.
- `POST /api/logout`: User logout (invalidates JWT).
- `GET /api/user/profile`: Retrieve user profile information (requires authentication).

4.3.2. Game State Management Endpoints:

- `GET /api/game/state`: Retrieve current game state for the authenticated user. Returns `GameStates` and associated `Plants` data.
- `POST /api/game/save`: Save current game state. Requires `game_state_id` and updated game data (money, energy, inventory, layouts, etc.).
- `POST /api/game/plant/action`: Perform an action on a plant (e.g., water, apply nutrients, harvest). Requires `plant_id`, `action_type`, and relevant parameters. Server-side validation will ensure fair play and prevent cheating.

4.3.3. Stripe Payment Endpoints:

- `POST /api/stripe/create-checkout-session`: Creates a new Stripe Checkout Session for one-time payments or subscriptions. Requires `product_id` or `price_id`. Returns the session ID and URL for the frontend to redirect the user.
- `POST /api/stripe/webhook`: Stripe webhook endpoint to receive asynchronous notifications about payment events (e.g., successful payment, subscription updates). This endpoint will update the user's in-game currency or unlock premium features upon successful payment.
- `GET /api/stripe/customer-portal`: Redirects the user to the Stripe Customer Portal for managing subscriptions and billing information.

4.4. Security Measures

Robust security is paramount for protecting user data and preventing unauthorized access or cheating. The following measures will be implemented:

1. **HTTPS Everywhere:** All communication between the Unreal Engine 5 client and the Flask backend will be encrypted using HTTPS (TLS/SSL) to prevent eavesdropping and man-in-the-middle attacks.
2. **Password Hashing:** User passwords will never be stored in plain text. Instead, strong, one-way hashing algorithms (e.g., bcrypt) with appropriate salting will be used to store password hashes.
3. **JWT (JSON Web Tokens) for Authentication:** After successful login, the backend will issue a short-lived JWT. This token will be sent with every subsequent authenticated request, allowing the server to verify the user's identity without requiring repeated password submissions. Tokens will be stored securely on the client side (e.g., in secure local storage for mobile apps).
4. **Input Validation and Sanitization:** All data received from the client (e.g., user input, game actions) will be rigorously validated and sanitized on the server-side to prevent SQL injection, XSS (Cross-Site Scripting), and other common web vulnerabilities.
5. **Rate Limiting:** Implement rate limiting on API endpoints (especially login and registration) to prevent brute-force attacks and denial-of-service (DoS) attacks.
6. **CORS (Cross-Origin Resource Sharing):** Properly configure CORS headers on the Flask backend to allow requests only from trusted origins (your Unreal Engine 5 application).
7. **Server-Side Validation of Game Logic:** Critical game actions (e.g., plant growth calculations, resource consumption, inventory updates) will be validated on the server-side to prevent client-side manipulation and ensure game integrity.
8. **Environment Variables for Sensitive Data:** API keys (Stripe secret key), database credentials, and other sensitive information will be stored as environment variables on the server, never hardcoded in the codebase.
9. **Regular Security Audits:** Periodically review the codebase and infrastructure for potential security vulnerabilities.

4.5. Client-Server Communication (Unreal Engine 5)

Unreal Engine 5 provides several ways to communicate with external web services. For CannaVille Pro, the `HTTP` module and potentially `WebSockets` will be used.

- **HTTP Requests:** For most API interactions (login, save game, retrieve data), standard HTTP POST/GET requests will be made using Unreal Engine's `FHttpModule`. JSON will be the primary data format for request and response bodies.
- **JSON Parsing:** Unreal Engine has built-in JSON parsing capabilities (`FJsonSerializer`) to handle data received from the backend.
- **Error Handling:** Robust error handling will be implemented on both the client and server to gracefully manage network issues, API errors, and invalid data.
- **WebSockets (Optional):** For real-time updates (e.g., live chat, multiplayer features if implemented later), WebSockets could be considered for persistent, low-latency communication, though initial game state and actions will primarily use HTTP.

This backend architecture ensures a secure, scalable, and functional foundation for CannaVille Pro, supporting all necessary game mechanics and financial transactions.

5. Mobile Deployment and Packaging

Deploying CannaVille Pro to mobile platforms (Android and iOS) involves specific packaging and distribution steps within Unreal Engine 5. This section outlines the process to ensure the application is correctly built, signed, and ready for submission to app stores.

5.1. Packaging for Android

Packaging for Android involves configuring project settings, setting up SDKs, and building the APK (Android Package Kit) or AAB (Android App Bundle).

5.1.1. Prerequisites:

- **Android Studio:** Installed with necessary SDK Platforms, SDK Build-Tools, and Platform-Tools.

- **Java Development Kit (JDK):** OpenJDK 11 is recommended and configured in Unreal Engine.
- **Unreal Engine Android SDK Setup:** Ensure that Unreal Engine is correctly pointing to the installed Android SDK, NDK, and JDK locations in `Project Settings > Platforms > Android SDK`.

**5.1.2. Project Settings Configuration (`Project Settings > Platforms > Android`):

- **Package Name:** Set a unique package name (e.g., `com.yourcompany.cannavillepro`).
- **Store Version:** Increment this number for each new release.
- **Minimum SDK Version & Target SDK Version:** Set these based on your target audience and desired Android OS compatibility. For broader reach, start with Android 7.0 (API Level 24) or higher.
- **Supported ABIs (Application Binary Interfaces):** Select `arm64-v8a` for modern 64-bit Android devices. `armeabi-v7a` can be included for older 32-bit devices if necessary, but `arm64-v8a` is generally sufficient and recommended for performance.
- **Build Configuration:** Choose `Shipping` for final release builds to ensure maximum optimization and performance. `Development` or `Debug` builds are for testing.
- **Packaging Method:** Select `Package game as an Android App Bundle (AAB)` for Google Play Store submission. AABs are more efficient for distribution as Google Play generates optimized APKs for different device configurations.
- **Signing:**
  - **Generate Keystore:** Create a new keystore file (`.keystore`) using Java's `keytool` utility. This file is essential for signing your application and proving its authenticity.
  - **Keystore Path, Alias, Password:** Configure these details in the Project Settings. **Crucially, keep your keystore file and passwords secure and backed up.** Without them, you cannot update your application on the Play Store.

5.1.3. Packaging Process:

1. **File > Package Project > Android > Android (ETC2) or Android (Multi)**: ETC2 is a common texture compression format for Android. Multi-format packages include multiple texture formats for broader device compatibility, but result in larger initial downloads.
2. **Output Log:** Monitor the Output Log in Unreal Engine for any errors or warnings during the packaging process.
3. **Result:** Upon successful packaging, an AAB file (or APK if selected) will be generated in your project's `Saved/StagedBuilds/Android` directory.

5.2. Packaging for iOS

Packaging for iOS requires a macOS machine with Xcode installed and involves setting up provisioning profiles and certificates.

5.2.1. Prerequisites:

- **macOS Machine:** Required for Xcode and iOS development.
- **Xcode:** Installed and updated to the latest stable version.
- **Apple Developer Account:** Necessary for generating signing certificates and provisioning profiles.
- **Unreal Engine iOS SDK Setup:** Ensure Unreal Engine is correctly configured to use Xcode and the iOS SDK.

**5.2.2. Project Settings Configuration (`Project Settings > Platforms > iOS`):

- **Bundle Identifier:** Set a unique identifier (e.g., `com.yourcompany.cannavillepro`). This must match the Bundle ID in your provisioning profile.
- **Version & Build:** Increment these numbers for each new release.
- **Signing:**
  - **Automatic Signing:** If using Xcode 13+, Unreal Engine can leverage Xcode's automatic signing. Ensure your Apple Developer account is configured in Xcode.
  - **Manual Signing:** For more control, manually select your **Mobile Provision** (provisioning profile) and **Signing Certificate** that you generated from the

Apple Developer website. The provisioning profile links your app ID, devices, and certificate.

- **Supported Orientations:** Select `Portrait` or `Landscape` based on your game's design. For CannaVille Pro, `Landscape` is likely preferred for the 3D environment.

5.2.3. Packaging Process:

1. **File > Package Project > iOS:** Select this option from the Unreal Engine editor.
2. **Output Log:** Monitor the Output Log for packaging progress and any issues.
3. **Result:** A `.ipa` (iOS App Store Package) file will be generated in your project's `Saved/StagedBuilds/iOS` directory upon successful completion.

5.3. Distribution to App Stores

Once packaged, the application can be submitted to the respective app stores.

5.3.1. Google Play Store (Android):

1. **Google Play Console:** Log in to your Google Play Console account.
2. **Create New App:** Start a new application entry.
3. **Upload AAB:** Upload the generated AAB file to a release track (e.g., Internal testing, Closed testing, Open testing, Production).
4. **Store Listing:** Provide app details, screenshots, feature graphics, and a compelling description.
5. **Content Rating:** Complete the content rating questionnaire.
6. **Pricing & Distribution:** Set pricing and target countries.
7. **Review & Publish:** Submit your app for review. Once approved, it will be published.

5.3.2. Apple App Store (iOS):

1. **App Store Connect:** Log in to your App Store Connect account.
2. **Create New App:** Set up a new app entry.
3. **Upload IPA:** Use Xcode or Transporter app to upload the generated `.ipa` file to App Store Connect.

4. **App Store Listing:** Provide app metadata, screenshots, app preview videos, and a detailed description.
5. **Pricing & Availability:** Set pricing and distribution territories.
6. **Build Selection:** Select the uploaded build for submission.
7. **Review & Release:** Submit your app for Apple's review process. Once approved, you can manually release it or set it for automatic release.

5.4. Post-Deployment Considerations

- **Analytics:** Integrate analytics SDKs (e.g., Google Analytics for Firebase, GameAnalytics) to track user engagement, crashes, and performance metrics.
- **Crash Reporting:** Set up crash reporting tools (e.g., Sentry, Firebase Crashlytics) to quickly identify and resolve issues.
- **Updates:** Plan for regular updates to introduce new features, fix bugs, and improve performance. Each update will require a new packaging and submission process.
- **Marketing:** Develop a marketing strategy to promote your app on the respective app stores and other channels.

This comprehensive guide to mobile deployment and packaging ensures that CannaVille Pro can reach its target audience on both Android and iOS platforms, delivering a high-quality, hyper-realistic gaming experience.