🕹️ BrawlDev Master Notes: Roblox Systems Manual

Version: 2.0 | Focus: Scalable Architecture & Performance

📋 Table of Contents

⚖️ 1. LocalScript vs ServerScript

Aspect	LocalScript	ServerScript
Execution	Client (Player's PC)	Server (Roblox Cloud)
Visibility	Affects only the local player	Affects everyone in the server
Security	❌ High Risk (Client-side)	✅ Secure (Authoritative)

Typical Use	UI, Tweens, Player Input	Currency, Combat, Saving Data
Location	StarterGui, StarterPack	ServerScriptService

The Golden Rule: Never trust the client for logic. The client requests (FireEvent), the server validates (Check if enough coins), and then the server executes.

🗑️ 2. Instance Deletion (Memory Management)

Proper cleanup prevents memory leaks that crash servers over time.

- ❌ Deprecated: Part:Remove() – This only sets the parent to nil; the object stays in memory.
- ✅ Standard: Part:Destroy() – Locked and cleared from memory immediately.
- ✅ Delayed: game.Debris:AddItem(Instance, Time) – The "clean" way to handle temporary effects (bullets, dropped coins, blood).

🛠️ 3. Tools System

Tools represent the bridge between player interaction and game events.

🔄 The Tool Lifecycle

StarterPack → Player.Backpack (Inventory) → Character (When Equipped) → Backpack (When Unequipped).

⚙️ Key Tool Properties

- **ManualActivationOnly: If true, the tool won't fire Activated on click. Useful for custom combat systems.**
- **RequiresHandle: Uncheck if using a "Handleless" tool (script-only tools).**
- **CanBeDropped: Prevents players from losing essential items.**

⏳ 4. Task Library (Modern Timing)

Replaces legacy wait(), spawn(), and delay() for better performance.

1. **task.wait(n): Stops current thread for n seconds. Most used for cooldowns.**
2. **task.delay(n, function): Schedules a function to run after n seconds without blocking the current script.**
3. **task.spawn(function): Runs a function immediately on a separate thread (Parallelism).**
4. **task.cancel(taskThread): Essential for stopping active delays (e.g., stopping a poison tick if the player drinks an antidote).**

📡 5. Remote Events & Functions

RemoteEvents (One-Way)

- **Client → Server: Remote:FireServer(args) – "I want to swing my sword."**
- **Server → Client: Remote:FireClient(player, args) – "Your health is low, show a red screen."**

RemoteFunctions (Two-Way/Callback)

- InvokeServer: The client waits for the server to reply.
- Crucial Use Case: Purchasing items. Client: "Can I buy this?" → Server: "Checks logic..." → Server: return true.

📐 6. Vector3 & Magnitude

Roblox uses X (Red), Y (Green), and Z (Blue).

- Distance Check: (Pos1 - Pos2).Magnitude returns the distance in Studs.
- Direction: (Target.Position - Start.Position).Unit returns a vector of length 1 pointing toward the target.
- Note: Use CFrame when you need to handle rotation; use Vector3 for raw positioning and scaling.

🧵 7. Coroutines (Advanced Threading)

Coroutines allow a script to "pause" and "resume" work later.

- coroutine.yield(): Puts the thread to sleep.
- coroutine.resume(): Wakes the thread up.
- Pro Pattern: Use coroutine.wrap() to execute a function as a separate thread that starts immediately.

🏷️ 8. CollectionService (The Tag System)

Instead of putting a script inside every single "KillPart," you tag them as "KillPart" and use one script to manage them all.

- CS:AddTag(Object, "TagName")
- CS:GetTagged("TagName") — Returns a table of all objects.

- **Signal Pattern:** `CS:GetInstanceAddedSignal("Tag"):Connect(function)` – Automatically applies logic to new items that spawn with that tag.

🏗️ 9. OOP (Object Oriented Programming)

Building reusable blueprints for complex objects (like an Enemy, a Car, or a Weapon).

- **ModuleScripts:** Contain the "Class."
- **Metatables (`__index`):** Allows an object to "inherit" functions from its blueprint.
- **`self`:** A keyword that refers to the specific object being interacted with.

⌨️ 10. UserInputService (UIS)

The client-side engine for player input.

- **`gameProcessedEvent`:** A boolean that is true if the player is typing in chat or clicking a GUI. Always check this first to prevent "accidental" inputs.
- **`InputBegan`:** Fires when a button is pressed.
- **`InputEnded`:** Fires when a button is released.

💾 11. DataStore Service

Persistent storage for player data.

- **Pcall Wrapper:** DataStore requests can fail. Always wrap them in a `pcall()` to prevent the script from breaking.

- **UpdateAsync: More reliable than SetAsync because it checks the current value before changing it (prevents data overwrites).**
- **Scopes: Use scopes (e.g., "v1/PlayerStats") to version your data if you make big game updates.**

🎞️ 12. Animations (Mastering Movement)

Animations are key for game feel. They run on the Animator object inside the Humanoid.

- **R6 vs R15: You cannot play an R15 animation on an R6 character.**
- **Animation Priorities:**
  1. **Core: Default movement.**
  2. **Idle: Standing still.**
  3. **Movement: Walking/Running.**
  4. **Action: Sword swings, reloads (Overrides all others).**
- **LoadAnimation: Always load the animation onto the Animator via the server or client, then call :Play().**

🏃 13. Humanoid States & Methods

The Humanoid controls the "physics" and "status" of a character.

Common Methods

- **:MoveTo(Vector3): Directs the NPC/Character to walk to a point.**
- **:TakeDamage(amount): Bypasses ForceFields (unlike setting Health directly).**
- **:ChangeState(Enum.HumanoidStateType.Physics): Forces the humanoid into a specific state (like ragdoll).**

Key Properties

- **WalkSpeed: Default is 16.**
- **JumpPower: Default is 50.**
- **AutoRotate: Set to false for "Side-Scroller" style movement where the player shouldn't turn toward the mouse.**