ChatGPT

Pokémon Battle System Implementation Outline

Overview and Goals

This outline describes a **complete Pokémon battle system** based on the latest generation mechanics. It follows the architecture of Pokémon Showdown's simulator (a proven battle engine) as a guide, ensuring that all standard mechanics are accounted for. The system is designed in **Python** and tailored for integration with the **Evennia MUD platform**, meaning it should cleanly separate game logic from presentation and handle multiple simultaneous battles. Key goals include: - **Accuracy to official mechanics:** Use the current generation's battle rules (moves, stats, types, abilities, items, etc.) so outcomes match the real games and Pokémon Showdown. - **Support for multiple battle formats:** Handle single battles (1v1), double battles (2v2), and scalable to other formats (e.g. multi-battles or free-for-alls) by designing flexible data structures. - **Maintainable, extensible design:** Use a high-level architecture with clear separation of concerns (turn logic, damage calculation, effect handling, etc.), allowing future addition of non-standard or custom mechanics. - **Evennia compatibility:** Fit into Evennia's MUD framework by using persistent objects or scripts for battle state and leveraging Evennia's command system for player inputs. The design should enable text-based outputs for battle events and support turn-based flow in a multi-user environment.

High-Level Architecture

Main components and their responsibilities: - **Battle Engine (Battle Handler):** The core orchestrator controlling the flow of a battle. It maintains the overall state (participating trainers, active Pokémon on each side, turn order, field conditions, etc.) and processes each turn's sequence of events. In Evennia, this can be implemented as a persistent `Script` or handler object for each ongoing battle [1] [2], so it persists through server reloads and is not tied to a single player object. - **Trainer/Player Entity:** Represents a participant in the battle (could be a human player or AI). It holds a team/party of Pokémon and interacts with the Battle Engine by selecting actions each turn. In Evennia, trainers can be the player characters issuing battle commands. - **Pokémon Entity:** Represents an individual Pokémon in battle. This object holds all **Pokémon stats and state** needed during combat (level, species/base stats, current HP, stat stages, types, moveset, ability, held item, status conditions, etc.). The Pokémon entity also provides methods to handle events like taking damage, fainting, using a move, etc. - **Move Data/Logic:** A representation of Pokémon moves. Each move has properties such as name, type, category (Physical/Special/Status), base power, accuracy, priority, PP, target scope (single-target, multi-target, self, etc.), and a description of its effect. The system will include a **Move execution function** or class that applies the move's effects when used (damage calculation, status infliction, stat changes, etc.). Complex or unique move effects (e.g. multi-hit moves, two-turn moves, recoil, etc.) are handled in this logic. - **Ability Data/Logic:** Each Pokémon's ability (if any) is a passive effect that can influence battle events. The architecture will likely utilize an **event-driven system** where abilities listen for certain triggers (e.g. a "on switch-in" event for Intimidate, or "on hit by move" for abilities like Volt Absorb) and automatically apply their effects at the appropriate time. - **Item Data/Logic:** Held items can modify stats or trigger effects under certain conditions (e.g. a Berry that heals at low HP, Choice Scarf modifying Speed stat, Focus Sash preventing a KO). Like abilities, the system will handle items via event triggers or condition checks at relevant points (e.g. "before damage to check Focus

Sash", "end of turn for Leftovers healing"). Items and abilities often function similarly by hooking into battle events. - **Battle Field State:** Tracks global or side-specific conditions that affect the battle: - **Weather** (Rain, Sunlight, Sandstorm, Hail, etc.) – with turn counters and effects on Pokémon (e.g. Rain boosts Water moves, Sandstorm causes chip damage and boosts Rock-type Sp. Def). - **Terrain** (Electric Terrain, Grassy Terrain, etc.) – with turn counters and effects (e.g. blocking sleep or boosting certain move types). - **Entry Hazards** on each side (Spikes, Stealth Rock, Toxic Spikes, Sticky Web) – with counters or layers and effects when Pokémon switch in. - **Other Field Effects:** Trick Room (alters turn order), Tailwind (temporary speed boost for one side), Reflect/Light Screen (damage reduction for a side), etc. These are tracked with their remaining durations and applied in calculations. - **Turn Manager:** Logic within the Battle Engine that coordinates each phase of a turn in sequence (outlined in detail below under **Turn Cycle**). This ensures that **move selection, priority resolution, move effects, and end-of-turn effects** happen in the correct order each turn.

All these components work together to simulate battles. The design should enforce a clear **separation of data and logic**: for example, Pokémon, moves, abilities, etc., can be defined with data (possibly loaded from JSON or a Pokédex module) and the engine's code uses that data to apply mechanics. This makes it easier to update move/ability definitions or add new ones without changing core logic.

Data Model and Structures

Pokémon Entity Structure

Each Pokémon in battle will be represented by an object with fields for all relevant combat stats and conditions: - **Base Attributes:** Species identity and base stats (HP, Attack, Defense, Special Attack, Special Defense, Speed) and type(s). Base stats and level are used to calculate actual battle stats. - **Derived Battle Stats:** Current **HP** (hit points) and current **stat stages** for Attack, Defense, Sp. Atk, Sp. Def, Speed, Accuracy, and Evasion (stages typically range from -6 to +6). Stat stages modify the effective stat values by predefined multipliers (e.g. +1 Attack stage = 150% Attack, -1 = 67% Attack, etc.). - **Moveset:** A list of up to 4 moves known. For each move, track its **current PP** and whether it's been used (for moves like Struggle when out of PP). - **Ability:** The Pokémon's ability (if it has one). This may be a reference to an Ability object or identifier that the battle engine can use to trigger ability effects. - **Held Item:** The item the Pokémon holds, if any. Could be an object or identifier in an item database. - **Status Condition:** Any major status ailment the Pokémon has (e.g. Burn, Paralysis, Sleep, Poison, Freeze). This affects its behavior: - Burn reduces physical Attack and causes end-of-turn damage. - Paralysis reduces Speed and gives a chance to be "fully paralyzed" (skip move). - Sleep and Freeze prevent acting for a number of turns (sleep has a counter, freeze is probabilistic to thaw on each turn or via certain moves). - Poison deals end-of-turn damage (regular or toxic poison which increases damage each turn). - **Volatile Statuses:** Other temporary battle states like confusion, infatuation, trapping (e.g. Whirlpool), etc., including turn counters for their duration. Also flags like flinch (which is cleared at end of turn), and whether the Pokémon is mid-way through a multi-turn move (e.g. charging or semi-invulnerable in Fly/Dig). - **Current Battle State Flags:** For example: - If the Pokémon is **active** on the field or in reserve (fainted Pokémon or those not yet sent out in team). - If it has acted this turn, or if it is forced to use a certain move (e.g. due to Encore or Choice item lock). - Whether Dynamax/Gigantamax or Terastallization is active (for latest gen mechanics, if included). - Transformations or form changes (e.g. Ditto's Transform, Mega Evolution, form changes from moves or abilities). - **References to Trainer/Side:** Which trainer or side the Pokémon belongs to, to update team status or check side conditions (used for things like Light Screen affecting damage to all Pokémon on that side). - **Methods:** The Pokémon class will have methods such as: - `apply_damage(amount)` – reduces HP and handles fainting if HP ≤ 0. - `heal(amount)` – increase HP (capped at max HP). - `modify_stat(stat, stages)` – change stat stage

(e.g. Sword Dance +2 Attack, Intimidate -1 Attack) with clamping to [-6, +6]. - `can_act()` – check if Pokémon can act this turn (not fainted, not frozen, not fully paralyzed, etc.). - `use_move(move, target)` – perhaps instruct the Pokémon to execute a move on a target (the actual logic may reside in move processing but Pokemon might trigger PP decrement, etc.). - **Event hooks:** The Pokémon may have internal hooks for certain events (like "on end turn" to handle status damage or "on take damage" for abilities like Weak Armor). Alternatively, these can be handled by central battle logic checking each Pokémon's status.

Move Definition and Effects

Moves are defined by a structured dataset or class. Key fields for each move: - **Name and Type:** E.g. "Thunderbolt" (Electric type). - **Category:** Physical, Special, or Status. This determines whether Attack/Defense or Sp. Atk/Sp. Def are used in damage calculation, or if it's a non-damaging move. - **Base Power:** Numeric base damage. Might be `None` or 0 for status moves or variable-power moves. - **Accuracy:** Chance to hit (e.g. 100 means 100%, 85 means 85% hit rate). Some moves have `—` accuracy which means they never miss under normal conditions (or have special hit logic). - **Power Points (PP):** How many times the move can be used. - **Priority:** An integer indicating priority bracket. Default is 0; greater values mean the move executes *before* ordinary moves, lower (negative) means it executes later. For example, Quick Attack has +1, Protect +4, Trick Room -7, etc. This is used in turn order resolution [3]. - **Targeting:** Which slots the move can hit: - Single-target (an opponent or an ally), - Multiple targets (e.g. "all opponents" or "all Pokémon except user" as Earthquake in doubles), - Self (moves like Swords Dance), - Field (like entry hazards affecting a side). - **Effect Flags:** Various special properties: - Flinch chance, status chance (e.g. Scald's burn chance), - Critical-hit ratio stage (some moves have higher crit rate), - Whether it makes contact (important for abilities like Rough Skin), - If it's a sound-based move, punching move, etc., which certain abilities affect (e.g. Bulletproof blocks ball/bomb moves). - Charging or multi-turn behavior (Solar Beam charges one turn, attacks the next; Fly/Dig become semi-invulnerable in the interim turn). - Recoil or self-damage (Take Down recoil, Mind Blown self-damage), - Drain (Giga Drain heals user by portion of damage), - Increased priority on certain turns (e.g. Focus Punch fails if hit before execution), - etc. - **Move Resolution Logic:** For each move, the battle engine must implement what it *does*. This can be handled by a **generic move execution function** that uses the move's data and possibly a script of effects: - **Accuracy Check:** Except for certain moves, when used, perform an accuracy roll. Account for accuracy/evasion stat stages and any accuracy modifiers (e.g. weather effects like fog, abilities like No Guard or compound eyes, moves like Lock-On which guarantee hit). If the move misses, handle accordingly (missed moves usually do nothing, though some still consume PP and can fail messages). - **Target Selection:** Determine which Pokémon are affected based on targeting (in singles it's the one opponent; in doubles could be one chosen target or multiple). Validate that targets are valid (e.g., if target fainted and move can retarget, or moves that hit all). - **Damage Calculation:** If it's a damaging move, calculate damage for each target: - Use the standard Pokémon damage formula: $$ \text{Damage} = \left(\frac{2 \times \text{Level} + 10}{250} \times \frac{\text{Attack}}{\text{Defense}} \times \text{BasePower} + 2\right) \times \text{Modifier} $$ where *Attack* and *Defense* are either the Pokémon's Attack vs target's Defense for physical moves or Sp. Atk vs Sp. Def for special moves [4]. The *Modifier* is a product of factors: a random factor (0.85–1.00 range), critical hit (typically 1.5× in Gen9), STAB, type effectiveness, weather, terrain, and any other boosts or reductions [4]. Same-Type Attack Bonus (STAB) is usually 1.5× if the user's type matches the move's type (or 2× if Terastallized into that type in Gen9), and type effectiveness is 2× for super-effective, 0.5× for not very effective, 0× if immune, etc. (multiple types stack multiplicatively). Apply all these and round down at each appropriate step (to match game mechanics). - The engine should handle damage caps or modifications: for example, damage is capped at target's remaining HP (no "negative HP"). Also moves like **Fixed damage** (Night Shade, seismic toss do set damage = user's level), **One-Hit KO moves** (Sheer Cold) which ignore formula and just KO if conditions

meet, should be handled as special cases in move logic. - If a move hits multiple targets (spread moves in doubles/triples), apply the appropriate damage reduction. In official mechanics, moves hitting multiple targets in one action have their damage multiplied by 0.75 (75%) [5]. The engine should apply this modifier when a move is hitting two or more Pokémon at once. (If only one target remains, the move deals full damage [6]). - Check for **critical hits**: typically a random chance based on critical rate stage. If critical, override certain calculations: critical hits ignore negative stat drops of the attacker and positive stat boosts of the defender in damage calculation, and use the crit multiplier (usually 1.5×). Mark damage as critical for messaging. - **Apply Damage:** Subtract HP from the target(s) as calculated. If a target's HP drops to 0 or below, mark it as fainted and queue fainting procedure (to be processed after the move or at end of sequence). - **Secondary Effects:** After damage, apply any additional effects of the move: - Status ailments (e.g. Thunderbolt's 10% chance to paralyze). Check chance and apply to target if it survives and isn't already afflicted or immune. - Stat changes (e.g. Close Combat lowers user's Defense/SpDef, or Icy Wind lowers targets' Speed). Apply stage modifications accordingly, ensuring not to exceed stage limits. - Recoil or self-damage (if the user has recoil, deduct HP accordingly; if a Life Orb item recoil or Struggle recoil, handle those in a similar way after move). - Healing moves or drain: if the move absorbs HP (Mega Drain, etc.), heal the user by the appropriate amount (often 50% of damage dealt). - Other unique effects: e.g. phasing moves like Roar/Whirlwind force target switch (to be handled after move execution), multi-turn setup moves like Future Sight (which schedule damage after a delay), etc. These may require scheduling events in the battle engine for later turns. - **Miss or Immunity Handling:** If the move misses or the target is immune (due to type immunity or an ability like Levitate or an item like Air Balloon), handle the "fail" or "miss" outcome (usually just a message; some moves have consequences on miss like recoil for Jump Kick). - **Move Usage Constraints:** The engine should also enforce mechanics like: - **PP reduction:** Using a move consumes 1 PP. If a move has 0 PP, it cannot be chosen (if all moves have 0 PP, the Pokémon is forced to use Struggle). - **Choice-lock effects:** If a Pokémon is holding a Choice Scarf/Specs/Band and has acted, it cannot select a different move until it switches out. - **Encore or Torment:** If under Encore, the Pokémon must use the same move again; if under Torment, it cannot use the same move twice in a row. - **Recharge moves:** Moves like Hyper Beam cause the user to skip the next turn (enter a "recharge" state). - **Disable and Imprison:** If a move is disabled or blocked by Imprison (opponent knows the move), it cannot be selected. - **Flinch:** If the Pokémon flinched (from a prior turn's effect like Fake Out or a King's Rock), it will be unable to execute its chosen move this turn (the action is effectively skipped). - The move execution should be implemented in a clean way, possibly with a **dispatch or script system**: e.g., a dictionary of move name -> effect function, or data-driven effects that the engine interprets. This way new moves can be added without writing huge if/else blocks. Complex effects can still be coded as needed but should integrate with the event system (for example, a move might set a flag on the Pokémon or schedule a callback in future turns).

Abilities and Items

Abilities and items function as modifiers to the battle and often trigger automatically. To manage the complexity of 300+ abilities and many items, an **event-driven architecture** is recommended: - The Battle Engine can define **event hooks** for key points in the turn: e.g. *on battle start, on turn start, before move selection, on move use, on damage calculation, after damage, on faint, on turn end,* etc. Abilities or items can subscribe to these events to produce their effects. - **Ability examples:** - *Intimidate* (ability): triggers **on switch-in** of the Pokémon, lowering all opposing Pokémon's Attack by 1 stage. The engine, when a Pokémon enters battle, fires a "OnSwitchIn" event that ability handlers listen to – Intimidate's handler finds all opponents and calls their `modify_stat(Attack, -1)`. - *Levitate* (ability): provides immunity to Ground-type moves. This could be handled by checking, when a Ground move is about to hit a target, if target has Levitate then the move's effect on that target is nullified (or an event "onTryHit" that ability

intercepts to cancel the hit). - *Speed Boost*: triggers **at end of turn** for the Pokémon, raising its Speed stage by +1. The engine would fire an "EndTurn" event for each Pokémon, and Speed Boost handler applies the stat change. - *Flash Fire*: triggers when a Fire-type move is about to hit the Pokémon – instead of taking damage, the ability negates damage and raises a power boost flag. Implementation could intercept at a "onHit" or "damage calculation" phase to set damage to 0 and mark the Pokémon's state that its next Fire move will be boosted. - *Mold Breaker*: an ability that modifies how move immunity abilities are handled (it allows the user's moves to ignore abilities like Levitate). This might be implemented by flagging the Pokémon in the battle state so that when it attacks, the engine knows to ignore certain defensive abilities on targets. - **Item examples:** - *Berries*: Many have triggers like "when HP drops below 25%, heal 50%" (Sitrus Berry, etc.) or "when hit by a super-effective move, reduce damage" (e.g. Shuca Berry for Ground). The engine can check after damage if HP threshold is crossed to trigger a Berry's effect (heal and consume the item). Some can be done with events: e.g., a "AfterDamage" event where the item can react if criteria met. - *Focus Sash*: If a Pokémon would be KO'd from full HP by damage, Focus Sash leaves it at 1 HP instead and then is consumed. Implement by checking before applying damage: if damage >= current HP and current HP == full HP and item is Focus Sash, then set damage to current HP-1 (so it survives at 1) and remove the item. - *Choice Band/Specs/Scarf*: Provide a permanent stat boost (Atk, Sp.Atk, Speed respectively by 1.5×) but lock the first move choice. Stat boosts can be applied when the Pokémon enters battle or dynamically in calculation. The move lock can be enforced by storing the chosen move and disallowing different moves until switching out. - *Life Orb*: Boosts damage by 30% at the cost of 10% max HP recoil each hit. Implement by modifying the damage multiplier (e.g. in damage calc, multiply by 1.3) and then after damage, if the attacker has Life Orb and move was damaging and user is not fainted, subtract 10% max HP from the user. - *Leftovers*: Heals 1/16 of max HP at end of each turn if holder isn't at full HP. This can tie into an end-of-turn event. - *Status/Weather related items*: e.g. Heavy-Duty Boots ignore entry hazards (check on switch-in not to apply hazard damage if boots equipped), Heat Rock extends duration of Sunny Day (modify weather duration logic if holder initiated weather). - The **Ability/Item system** could be implemented similarly to move effects: a registry of ability names to their effect handlers, and item names to effect handlers. Many abilities/items are straightforward (simple condition => effect) while others are complex (see Mold Breaker or those that change type interactions). The architecture should make it easy to add these. For instance, one might have an `Ability.apply_event(event, context)` method that each ability class overrides as needed to react to events. - **Order of operations:** Real Pokémon battles have a specific order in which effects apply. For example, abilities like **Protect** or **Magic Bounce** intercept moves at certain stages, end-of-turn healing occurs before weather damage, etc. The Battle Engine's turn processing must account for these ordering rules. Typically: - Turn start: Field/weather effects (e.g. weather continues or ends if its duration expired at turn end prior). - Before moves: Abilities like Drought (summon weather on switch), or terrain start, etc., would have already applied on switch-in phase. Also check any turn-start effects like Wish (healing if one was set to happen this turn). - Move resolution (see Turn Cycle below for detailed order). - End of turn: first, apply end-of-turn effects for each Pokémon in speed order or some defined order: this includes leftover healing, Black Sludge (poison heal/harm), Shed Skin chance to cure status, then weather damage (Sandstorm/Hail) and status damage (poison/burn). The specific sequence in official games is intricate; for accuracy the engine should implement according to known mechanics. For example, Leftovers healing happens before burn damage. - Faint checks should happen immediately when HP goes to 0 (but the Pokémon might remain on field until the move or effect sequence is done). The engine can queue fainted Pokémon to be removed after the current move resolves, unless the battle ends. - After all end-turn effects, check win conditions (any team with all Pokémon fainted). - **Concurrent triggers:** There may be multiple abilities/items triggering on the same event (e.g. both active Pokémon have end-of-turn effects). The engine should handle all, usually in a specific order (often turn order or side order). For example, if two

Pokémon would faint at end of turn (say one from sandstorm and one from poison), the game typically faints them in speed order or some priority.

By using an event-driven approach for abilities and items, we mirror Pokemon Showdown's architecture where the **event system is the most important part** of the simulation. This ensures that adding a new ability just means writing its behavior in response to certain events rather than tangling it in the main flow [7].

Turn Cycle and Action Resolution

The battle proceeds in **rounds (turns)**. Each turn consists of several phases, processed in a strict order to mirror the game's simultaneous-turn system:

1. **Start of Turn Phase:**
2. **Status/Ability Effects:** Some effects happen at the very start of a turn. For example, the ability *Rain Dish* heals in rain at turn start, or *Speed Boost* actually triggers at end of turn, so start-of-turn might not have many standard events in current mechanics. One key check is if weather/terrain is set to expire, their counters decrement at turn end and we check expiration at the start of next turn (or exactly at end – depending on mechanic). In general, the engine can have a placeholder for start-of-turn triggers (e.g., check if a Pokémon's binding band deals trap damage at turn start for moves like Fire Spin – but in Pokémon, partial trap damage is end of turn).

3. **Choice of Mega Evolution / Terastallize (if applicable):** In gens with Mega Evolution (Gen6-7) or Terastallization (Gen9), players typically declare this at move selection time. This could be handled as part of move selection, but it effectively occurs at turn start (before moves execute, Mega evolution happens immediately as turn begins). If such mechanics are in scope, handle transformation of the Pokémon's form/stats/types here.

4. **Move Selection Phase:**

5. Each Trainer/AI selects an action for each of their active Pokémon. **Possible actions** per Pokémon include:
    - Use a **move** from its moveset.
    - **Switch** out to a different Pokémon (in single battles, switching one Pokémon ends that Pokémon's action for the turn; in doubles, you could switch one or both active Pokémon as your actions).
    - Use an **item** from bag (if this were a single-player RPG scenario – but in competitive multiplayer (Showdown style), item usage in battle by trainer is not allowed aside from held items. Since focusing on standard multiplayer mechanics, this can be omitted or only considered for a possible future PvE extension).
    - **Run/Forfeit:** If a trainer flees or forfeits, ending the battle (this triggers an immediate battle end).
    - Do nothing (in case of being forced by some condition, but generally there is always an action; if a Pokémon cannot move due to sleep, it still *selects* a move but then will fail to execute it).
6. The system in Evennia will likely **wait for each player's command input** (like `battle attack <move>` or `battle switch <poke>`) to gather these choices. A timer could be used to auto-

choose a default (like a random move or “Struggle”) if a player fails to respond in time, but that’s optional.

7. After both players (or all participants) have submitted their actions, lock in the choices. This corresponds to the idea of simultaneous move selection.

8. **Action Order Determination:**

9. Once actions are chosen, the battle engine determines the sequence in which they will be executed in the **Execution Phase**:

    - **Switches vs Moves priority:** In official games, switching Pokémon has the highest priority, happening before moves with normal priority. In a turn where one trainer switches and the other uses an attack, the switch occurs first. The engine should implement that if any Pokémon is switching out, those switches are processed before any moves (unless the opponent used a **pursuit** move which has a special case to hit a switching Pokémon before it leaves).
    - **Mega Evolution / form changes:** If declared, these happen now (for Mega Evolution, the Pokémon’s stats and ability update before moves execute; Terastallization changes type before move execution as well).
    - **Move Priority:** For all moves being executed, sort them by their **priority** value first (higher priority moves go first). For example, all moves with +1 (Quick Attack, Protect, etc.) happen before those with 0, which happen before those with - priority (Trick Room, etc.).
    - **Speed Check:** Within the same priority bracket, order by the Pokémon’s Speed stat (after any stat stage modifications and item boosts). The faster Pokémon acts first. If speed stats are *exactly equal* (after modifiers), break ties randomly – essentially a coin flip to decide who goes first [8]. (This random tie-break should be done each turn if needed [9], since in official mechanics a speed tie can go either way each turn.)
    - **Multi-turn considerations:** If a Pokémon is mid-two-turn move (e.g. it used Dig on the previous turn and is now executing the attack on this turn), it generally will execute now and cannot be interrupted except by being hit by certain moves. The ordering for a second-turn of a charge move is the same priority as the move normally has. For example, Dig has no priority change; if two Pokémon both are emerging from Dig, speed tie rules apply, etc.
    - **Queued Actions:** The engine should form a list of **action events** to resolve in order. For each Pokémon that is set to act, have an entry like “Pokémon A uses Move X on target Y” or “Pokémon B switches out to Pokémon C”.
    - **Edge case – multiple Pokémon fainted before action:** If a Pokémon was scheduled to act but fainted earlier in the turn (e.g. due to an opponent’s higher-priority move), its action is canceled. The engine should check before executing each action that the executor is still alive and any targets are still valid. If not, skip or adjust targets (some moves will retarget if their original target fainted, e.g. double battle targeting).
    - **Protect/Detect** (priority +4 moves): If a Pokémon used Protect, mark it as having a protect shield this turn. When resolving attacks targeting it, those attacks will *fail* (no damage) unless the attacker has an effect to bypass protect (some moves or abilities do).
    - **Pursuit special case:** If a Pokémon is switching out and an opponent used Pursuit targeting it, Pursuit will hit *before* the switch happens (and at 2× power) – potentially preventing the switch if it KOs the Pokémon. The engine should detect this scenario: when resolving actions,

if a switch is happening and the opponent has Pursuit, handle that Pursuit attack prior to completing the switch action.

10. **Action Execution Phase:**

11. Now, iterate through the sorted action list and execute them one by one:
12. **Switch Actions:** When a switch action is reached:
    - Remove the switching Pokémon from the active field and send out the new Pokémon (this could create a sub-step where the new Pokémon's ability might trigger *immediately upon entry* – e.g., new Pokémon with Intimidate will activate now, affecting opponents [10]).
    - Reset any temporary conditions on the leaving Pokémon (stat stages might be reset if game rules dictate upon switch-out; in Pokémon, stat stages, confusion, etc., are typically reset when a Pokémon switches out, but status ailments and HP remain).
    - Entry hazard effects: When a Pokémon enters, if hazards like Stealth Rock or Spikes exist on that side, apply damage: Stealth Rock damage based on type (e.g. 1/8 or 1/2 HP depending on Rock effectiveness on that Pokémon's type), Spikes (lose 1/8, 1/6, or 1/4 HP depending on layers), Toxic Spikes (poison the newcomer). If the Pokémon has an ability or item preventing this (e.g. Magic Guard avoids hazard damage, Heavy-Duty Boots ignore hazards), account for that. If a hazard causes the Pokémon to faint immediately upon switch in, queue that faint (it will be processed after all switch-in effects).
    - Weather-related abilities: If the Pokémon has weather-starting ability (Drought, Drizzle, etc.) and the weather isn't already of that type or a stronger weather in effect, set the new weather now.
    - Other switch-in abilities: Intimidate (attack drop on opponents), Electric Terrain from Electric Surge, etc., all trigger here. The event system should fire "OnSwitchIn" events to handle these.
    - If multiple Pokémon switch at once (like both trainers switched in singles, or in doubles you switch two Pokémon), handle one by one according to a defined order (likely turn order – the faster trainer's switch happens first in singles, but in practice, switching simultaneously still has an order for resolving abilities).
13. **Move Actions:** For each move action in order:
    - If the user fainted before executing (e.g. from a higher-priority move), skip this action.
    - If the user is **unable to move** due to status (sleep, freeze) or a condition (flinch, paralysis chance):
    - Sleep: decrement its sleep turn counter (if using a sleep counter) and if it reaches 0, wake up; otherwise it stays asleep and the move fails this turn.
    - Freeze: chance to thaw each turn; if thawed now, it can act, otherwise fail the action.
    - Paralysis: 25% chance to be fully paralyzed each action – roll and if true, skip the move.
    - Flinch: if flinched (typically from being hit by a flinching move last turn), skip the move. Flinch status would have been set earlier (like from an opponent's attack) and now is checked.
    - If any of these cause skipping, output appropriate message ("Pokemon is fast asleep!", "Pokemon is paralyzed! It can't move!" etc.) and continue to the next action.
    - **Execute Move:** If not skipped, carry out the move's effect using the Move resolution logic described earlier (accuracy check, damage, secondary effects, etc.). This may involve multiple substeps:
    - Display attack message (for user feedback, e.g. "Pikachu used Thunderbolt!").
    - If applicable, handle *Mega Evolution* here (though usually it already happened at turn start if declared; but in older games, mega was chosen at selection and occurs immediately as action starts, we have handled that).

- Accuracy roll: If move misses (taking into account accuracy and evasion stages, plus any accuracy modifiers like Double Team or Gravity), display “attack missed” and move on.
- If move hits, apply damage and effects to each target as per move logic.
- If a target faints due to this move, mark them fainted (but do not remove yet – fainting will be processed after the move or possibly immediately if no other actions left).
- If the move has a **special effect that ends the battle or turn** (e.g. OHKO moves end battle for that Pokémon, or moves like Explosion/Self-Destruct which faint the user as effect), handle those:
    - Explosion: user’s HP set to 0 (user faints after dealing damage).
    - Healing Wish/Lunar Dance: user faints and triggers heal on incoming ally when they switch in.
    - U-turn/Volt Switch: user deals damage then switches out immediately. The engine should queue the actual switch after the damage is done, but *before* next action in sequence. This complicates turn order because a switch mid-turn might bring in a new Pokémon – the new Pokémon does not act immediately (they didn’t have an action selected this turn) but their switch-in abilities trigger right away. The engine must handle this insertion of a switch event.
    - Red card/Eject Button (items): if target or user is forced to switch due to an item effect when hit, schedule that switch immediately after the current move resolution.
- Weather or terrain altering moves: e.g. Rain Dance sets weather (update field state with a counter, affecting future turns).
- Stat stage altering moves: ensure those changes reflect in Pokémon state (e.g. after using Dragon Dance, user’s Attack+1 and Speed+1).
- Multi-hit moves (Double Kick, Triple Axel, etc.): loop the damage calculation and hit application for the number of hits, with possible early termination if target faints or certain condition (Parental Bond, etc.).
- After move execution, handle immediate aftermath:
- Check **fainting**: If any Pokémon fainted due to this action, mark them faint and possibly remove them now *if* no further actions from them remain. In singles, if a Pokémon fainted, its trainer will be prompted to send a replacement only after the whole turn’s actions (unless the battle ended). In doubles, a fainted Pokémon can be replaced at end of turn (or immediately if both on one side faint leaving none active, the replacement is needed to continue the turn – but typically turn ends if a side has no active Pokémon to execute remaining actions).
- If the move causes **status conditions** or volatile conditions (like confusion or infatuation on the target or user), apply those to the Pokémon objects.
- If recoil fainted the user (e.g. user uses Take Down and KO’s itself from recoil), mark it fainted too.
- **Pursuit case resolution:** If we are executing a Pursuit that targeted a switching Pokémon, we handle that damage. If that causes the switching Pokémon to faint, cancel its switch (it never leaves because it died) and the trainer will need to choose a new Pokémon to send next turn (or now if we allow immediate replacement).

14. Continue until all actions in the queue are processed.

15. **End of Turn Phase:**

16. After all moves and switches for the turn have been resolved, the turn isn’t over yet. Now apply **end-of-turn effects** in a correct order:

17. Common end-of-turn effects to handle:
    - **Damage from Statuses:** If a Pokémon is Burned, it loses 1/16 of max HP; if Poisoned, it loses 1/8 max HP; if Badly Poisoned (toxic), it loses an increasing fraction of HP (1/16, then 2/16, 3/16, etc., incrementing each turn toxic status persists while active). The engine should store a toxic counter for badly poisoned and increment it each turn the Pokémon stays in battle with that status [11].
    - **Weather Damage/Healing:** Sandstorm and Hail deal 1/16 max HP to non-resistant Pokémon (those not of certain types or without safety goggles/ability). Sunlight and Rain typically don't deal damage themselves (except specific cases like Solar Power ability which is handled via ability). Also, Rain/Sun could end after 5 turns (or 8 with extending item) – the engine should decrement weather timer and possibly end the weather now, with a message ("The rain stopped.") at end-of-turn if it hits 0.
    - **Terrain effects:** like weather, terrain lasts typically 5 turns. Decrement and end if time. Terrain itself doesn't damage, but if any end-turn terrain effects existed (none standard except Grassy Terrain healing 1/16 HP for grounded Pokémon).
    - **End-of-turn healing or item effects:** Leftovers and Black Sludge (heals poison types or harms others) should activate now, healing 1/16 max HP [12] [13]. Other items like Shell Bell heal a bit (but that is immediate after an attack, not end-turn).
    - **Ailment cures:** Some abilities or items might cure status at end (Shed Skin ability: chance to cure status at end of turn; Hydration: cures status if Rain at end-of-turn, etc.). Process these after damage from status, typically.
    - **Secondary turn counters:** Moves like Perish Song reduce their counters at end of turn; if a Pokémon's Perish count reaches 0, it faints now.
    - **Delayed effects resolution:** Some moves or effects that were set earlier might resolve now. For example, Future Sight or Doom Desire (they hit at end of the turn two turns after used – if this is that turn, apply the attack now). Also, if a Pokémon used Uproar, check if Uproar continues (it lasts 3 turns, causing no Pokémon to sleep).
    - **End-turn ability effects:**
    - Speed Boost (raises Speed by 1 stage at end for Pokémon with that ability).
    - Moody (raises one random stat +2 and another -1).
    - Toxic Boost, Flare Boost (activate under status to boost attack/sp.atk – might be handled as passive multiplier rather than an event).
    - Rain Dish, Dry Skin (heal in rain, Dry Skin damage in sun).
    - Oran Berry or other pinch berries might activate at end of turn if somehow the condition only became true during end-of-turn (though usually they activate immediately on HP drop).
    - After applying each effect, check if any Pokémon fainted as a result (e.g. faint from poison or perish song at end-turn). Queue those faint events.
18. **Faint Resolution:** Now remove any Pokémon that have fainted during the turn (either during actions or at end-of-turn). For each fainted Pokémon, announce the faint (and award experience if this were RPG single-player, but in PvP that's not needed). The Pokémon is no longer active.
    - Check win conditions: if one side has no remaining Pokémon able to fight (all fainted), the battle ends declaring the other side the winner. Stop further processing.
    - If the battle is not over and this is a singles or doubles battle: for each side that has empty active slots (a Pokémon fainted), prompt that trainer to **send out a replacement** from their remaining team (if any). This usually happens **before the next turn begins**. In a simulator, it might happen immediately at end of turn: you could pause the turn cycle to wait for input of who to send in. In official games, if both sides had a Pokémon faint, traditionally the player whose Pokémon fainted last sends first, etc. (Speed of last faint might determine who

replaces first). For simplicity, the engine can either enforce one side then the other or do simultaneously if possible.
    - If a replacement is sent in, apply its switch-in effects (entry hazards, ability like Intimidate) immediately as part of this end-of-turn resolution, so that going into the next turn the field state is settled.
19. **Turn Counter:** Increment the turn count (useful for certain moves/effects that last a fixed number of turns).
20. Now the turn is complete, and the engine loops back to the **Move Selection Phase** for the next turn (unless battle ended).

Throughout all these phases, the battle engine should be sending appropriate messages to the players (via Evennia's messaging to the combatants or room) to describe what's happening: e.g., "It started to rain!", "Charizard used Flamethrower! It's super effective!", "Charizard fainted!", etc. Though not required for the logic, these messages are important for a playable experience.

Supporting Different Battle Formats

The design must be flexible to handle varying battle formats beyond the standard *1v1 single battle*. Key considerations: - **Number of Sides:** While typical battles have 2 sides (Player vs Opponent), the system could allow more (e.g. free-for-all with 4 separate players each acting independently). The architecture should not hard-code "two players" assumptions in the core logic. Instead, the Battle object can maintain a list of sides/teams, each containing one or more Pokémon active and possibly one or more trainers. - **Multiple Pokémon per side:** Doubles battles (2 Pokémon active on each side) and Triples (3 on each side, as seen in Gen5) require the engine to manage more complex target selection and move effects: - The Pokémon entity and move logic should handle **moves that target multiple Pokémon** (like Rock Slide hitting both opponents in doubles) or target choice between multiple opponents. Many moves can target any one foe or even an ally in doubles; so the UI/command should allow specifying targets. The engine should enforce targeting rules (some moves can hit only opponents vs. some can hit anyone or everyone). - Update damage calculation for spread moves as mentioned (0.75× damage when hitting multiple targets in doubles) [5] . - In doubles, turn order is determined among four Pokémon. The speed/priority sorting works similarly: all Pokémon's chosen actions (including two from one side and two from the other) are sorted by priority then speed. The engine must be able to handle interwoven turn order (it's not simply one side then the other; fastest acts first regardless of side). - Some moves have different effects in multi battles (e.g., Surf hits all other Pokémon including partner, Earthquake hits all *except* the user and partner if partner is immune? Actually Earthquake hits *all other* including partner in doubles, same for Explosion). - **Alliances and target validity:** In team battles (like 2v2 with two players per team controlling one Pokémon each, or multi-battle where you have an ally), the system should know who is ally and who is opponent to apply certain rules: - Moves that target "all foes" shouldn't hit allies. - Certain moves can target ally (Helping Hand, healing moves like Heal Pulse). - Abilities like Intimidate affect *opposing* Pokémon, not allies. - Abilities that affect partner: e.g. Flower Gift boosts ally's stats in sun, etc. - Friendly fire rules: Generally you can target your ally with moves if you want (except in some battle facilities). The engine should allow it if the format does. - **Switching in Multi-Pokémon battles:** In doubles, a trainer can switch one of their two Pokémon on a turn (or both if both use their action to switch). If one Pokémon switches and the other uses a move, that is allowed. If both switch, that uses both actions. The engine's action resolution must handle multiple switches correctly (both switches resolve before any attacks). - **Rotation battles** (Gen5) and **Battle Royal** (free-for-all 4 player) have unique rules – these can be considered non-standard and possibly omitted. But the architecture, by supporting N sides and M Pokémon per side, could accommodate free-for-all where

every Pokémon is essentially its own side for targeting logic. Free-for-all 4-way: turn order across all 4, moves that hit multiple might hit 3 opponents, etc. - **State representation:** The Battle object could use structures like: - `sides: List[BattleSide]` where `BattleSide` contains fields like `active_pokemon: List[Pokemon]` (length 1 for singles, 2 for doubles, etc.), `side_conditions` (Reflect, Light Screen, Safeguard, Tailwind counters, spikes layers, etc.), and perhaps reference to the trainer(s) on that side. - Each Pokémon knows its index or position in the side (for moves like "position-based targeting" in triples). - **Turn flow differences:** Most of the turn sequence remains the same regardless of format, but multi battles have additional complexity in replacement after fainting. For example, in doubles, if one Pokémon faints, its partner still may execute its move (unless it already did or had a higher priority action). Replacement of a fainted mon happens at turn end. If both Pokémon on a side faint in one turn, and the opponent still has an action left, the battle typically ends (because one side has no active Pokémon at that moment) – but in official games, if a side loses all active Pokémon mid-turn but still has reserves, the turn typically continues until all actions done, then that side sends replacements (assuming at least one Pokémon of the opponent was still active to target – if none, battle ended). - The engine should handle scenario where multiple Pokémon faint before getting to act – their actions are canceled as mentioned. - **Victory conditions:** In multi-team battles, victory is when all Pokémon of all opposing *teams* are fainted. In free-for-all, when 3 of the 4 players have lost all Pokémon, the remaining player wins. - The outline mainly focuses on standard singles/doubles because that covers the majority of mechanics. The architecture, however, is built so that adding support for triples or other formats is a matter of adjusting parameters (max active per side, number of sides) and ensuring all loops and targeting logic account for multiple participants.

Integration with Evennia (MUD Implementation Notes)

Integrating this system into Evennia involves bridging the game logic with the MUD's command and data structures: - **Evennia Objects:** The **Pokémon** and **Trainer** can be modeled as Evennia typeclassed objects or attributes on player characters. For example, a `Pokemon` typeclass might represent each Pokémon with persistent stats (level, moves, etc.) outside of battle. However, the battle-specific state (current HP, stat stages, etc.) might be kept in a separate Battle Pokemon instance to avoid permanently altering the base stats. - Alternatively, one could avoid making each Pokémon an Evennia game object and instead keep them as pure Python objects managed by the battle engine, loaded from a stored team roster. This might be simpler for simulation purposes and then update the real object at end of battle (e.g. apply experience gain). - **Battle Handler as Script:** As Evennia's docs suggest, implementing the battle engine as a **Script** is useful [1]. One can create a `BattleScript` (or `CombatHandler`) that is spawned when a battle starts. This script holds references to the participating trainers and Pokémon (could store either object DBrefs or the Pokemon objects themselves) [2]. The script's timers or manual triggers can be used to run turn cycles. - **Command and Turn Flow:** When a battle starts (e.g. via a command like `challenge <player>`), the system: - Instantiates a Battle handler with the two trainers (or teams). - Sends both players a message that they are now in battle mode, possibly assigning them a specific **Command set** for battle that overrides normal commands (Evennia allows swapping command sets). - On each turn's move selection phase, the battle script will either prompt players for input ("Choose a move or action for your Pokémon") or expect them to use battle commands like `attack <move>` or `switch <pokeball_slot>`. These commands can store the choice in the battle handler's state. - Once all choices are in, the battle script processes the turn. This can be done either synchronously or through a timed event. Since battles in Pokémon are not real-time, you don't strictly need a timer except for maybe a turn timeout. The script can call the execution logic immediately after receiving all inputs. - The battle script then sends out the descriptive text of what happens during the turn to all participants (and possibly observers in the same room if desired). - If the

battle continues, it goes to next turn selection. If ended, announce the result (winner) and do cleanup: - Remove the battle command set from players, return them to normal game mode. - Possibly handle post-battle effects like distributing rewards or experience if that's part of the MUD design. - Delete or archive the battle script. - **Data persistence:** Because Evennia is server-based, consider what happens on disconnect or server reboot. Using a Script (which is stored in the database) means the battle state can survive a reboot, but turn-based combat might not resume well if players left. For simplicity, one might not persist battles through reboots (just cancel them). - **Testing and Codex:** The outline is structured such that an AI coding assistant (OpenAI Codex) can systematically implement each part. For example: - It can create classes for Pokemon, Move, etc., using the described fields. - It can create the BattleEngine class with methods for each phase. - The event system can be implemented as a series of methods or signals that abilities/items hook into. - Codex should implement the damage formula precisely as given [4] and ensure all factors are included (STAB, type effectiveness, critical, random). - Edge cases described (like speed ties [8], spread move damage [5], etc.) should be coded as conditional checks in the appropriate places. - The architecture ensures nothing critical is missing: from stat changes to weather, all known mechanics have an outline here to guide coding.

Conclusion and Extensibility

This design covers the **standard battle mechanics** for a Pokémon battle system, ensuring compatibility with the latest generation's rules. By following the outline (which mirrors a lot of how Pokémon Showdown works internally), you will implement a robust simulation that can handle virtually all in-game scenarios. The high-level separation of components (battle handler, Pokemon data, moves, events for abilities/items) makes it easier to maintain and extend: - **Adding new moves/abilities** requires defining their unique effects in the event system or move logic, without altering the core engine flow. - **Non-standard mechanics** (like alternate battle modes, or previous generation gimmicks like Z-Moves, Mega Evolution, Dynamax) can be added by extending the data model (e.g., a flag for "isDynamaxed" and modifying damage if so) and hooking into the turn flow where necessary. - **Evennia integration** means players can engage in battles naturally through text commands, and the system can be integrated with the rest of the MUD (inventory for held items, character stats for trainer etc., catching new Pokémon outside of battle feeding into available team for battle, etc.).

By adhering to this outline, an implementation should cover **all essential aspects of Pokémon combat**, ensuring that nothing is missing such as priority handling, stat calculations, effect timing, and support for various battle scenarios. Each section above can be translated into code modules or classes, and unit tests can be written for specific mechanics (e.g. damage calculation test, status effect test, multi-target move test) to verify correctness. The end result will be a faithful Python-based Pokémon battle engine suitable for a text-based multiplayer game.

Sources: The design is informed by known Pokémon mechanics and the Pokémon Showdown battle simulator architecture. Key formula and rules are drawn from community-documented sources (for example, the official damage formula [4], speed tie rule [8], and multi-target move damage adjustment [5]) to ensure accuracy.

---

[1] [2] [10] Turn based Combat System — Evennia 2.x documentation
https://www.evennia.com/docs/2.x/Howtos/Turn-based-Combat-System.html

3 4 11 How to design a Pokémon-like Combat System · Davide Aversa
https://www.davideaversa.it/blog/how-to-design-a-pokemon-like-combat-system-chapter-1/

5 6 Double Battle Primer - Smogon University
https://www.smogon.com/dp/articles/double_battles

7 Best practices for managing abilities in Pokemon-esque battle game
https://www.reddit.com/r/gamedev/comments/1k15uue/best_practices_for_managing_abilities_in/

8 9 Which Pokemon moves first if they have the same Speed? - PokéBase Pokémon Answers
https://pokemondb.net/pokebase/342076/which-pokemon-moves-first-if-they-have-the-same-speed

12 13 User:FIQ/Damage calculation - Bulbapedia, the community-driven Pokémon encyclopedia
https://bulbapedia.bulbagarden.net/wiki/User:FIQ/Damage_calculation