# **1) Tổng quan logic (tóm tắt)**

* Mỗi **run** = một “đời” của nhân vật. Run có seed để mọi RNG trở nên **deterministic** (giúp share và debug).
* Lặp: **chọn event → present choices → resolve (checks + RNG + modifiers) → apply effects (immediate + schedule chained events) → advance time/energy → repeat**.
* Một số event là **scripted** (canonical), phần còn lại là **procedural/AI-generated** dựa trên templates. Scripted ưu tiên, AI cached dùng khi cần.
* Giữa các run có **meta-progression**: grant points, mua talents/perks.

# **2) Danh sách stat đề xuất & vai trò (chi tiết)**

Mỗi stat kèm **range** (ví dụ 0–100), default khởi tạo theo background.

## **Visible (hiển thị cho player)**

1. **HP (Health)** — 0..100  
   * Ảnh hưởng đến khả năng sống (life end nếu <=0), khả năng hoàn thành các hành động tốn thể lực.
   * Các event về sức khoẻ, tai nạn sẽ giảm HP. Tăng bằng nghỉ ngơi, chăm sóc y tế, perks.
2. **Mood (Tâm trạng / Happiness)** — 0..100  
   * Ảnh hưởng sáng tạo, chance succeed một số social/creative checks, recovery rate.
   * Low mood có thể giảm năng lực hành động, tăng tiêu hao energy.
3. **Finance (Tiền / Tài chính)** — số (có thể âm)  
   * Dùng để chi trả chữa trị, học phí, mua items/talents. Ảnh hưởng đến lựa chọn (ví dụ phải từ chối du học vì không có tiền).
4. **Status (Địa vị xã hội / Social standing)** — 0..100  
   * Ảnh hưởng branch weight cho event về nghề nghiệp, networking, reputation-based opportunities.
5. **Relationship (Quan hệ)** — 0..100 (tổng hoặc có map cụ thể)  
   * Ảnh hưởng sự kiện xã hội, hỗ trợ, loans, job referrals.
6. **Energy** — 0..100 (mỗi milestone/mốc reset đến 100)  
   * Mỗi event có energyCost (0/1/5/10...). Hết energy → một hành động “rest” cần để tiếp tục major events.

## **Hidden / Advanced (không hiển thị hoặc show ẩn)**

1. **Luck** — -50..+50 (hoặc 0..100)  
   * Thêm modifier trực tiếp vào RNG cho events. Có thể farm qua perks/talents.
2. **Knowledge / Experience** — 0..100  
   * Mở lựa chọn đặc thù (ví dụ “apply for scholarship”), thay đổi success chance.
3. **RiskLevel** — 0..100 (ẩn)  
   * Cao → tăng tần suất event negative. Dùng để model “risky life choices”.
4. **Reputation / Karma** — 0..100  
   * Ảnh hưởng availability của certain branches (celeb, politics, career).

**Ghi chú:** bạn có thể lưu stats dưới dạng JSON { hp: 80, mood: 60, finance: 200, ... } trong DB/Run.

# **3) Cách hoạt động của event & choice (chi tiết bước-by-step)**

1. **Event selection**
   * Build candidate list:  
     + Filter events where min\_age <= current\_age <= max\_age.
     + Filter by prerequisites (stat thresholds, history flags).
     + Exclude already one-time events if is\_active=false or flagged done.
   * Compute weight for each candidate:  
     + weight = base\_branch\_weight \* rarity\_factor \* context\_modifiers
   * Use **seeded weighted random** to pick one.
2. **Present choices** (2–4 options, from event.choices).  
   * Some choices have preconditions (disabled if not met) — show as disabled or hide.
3. **Resolve choice** (core!)  
   * Two modes:  
     + **Deterministic effects**: simple stat deltas (e.g., finance += -50, knowledge += 10) — always applied.
     + **Check-based**: choices that include probabilistic success/failure or branching. For these:  
       1. Determine baseChance from event template (e.g., 50%).
       2. Compute modifiers from stats/perks/talents:  
          - e.g. luckBonus = Luck \* 0.005 (so 20 luck => +0.1 or +10%).
          - skillBonus if Knowledge > threshold etc.
       3. modifiedChance = clamp(baseChance \* (1 + totalBonus) - riskPenalty, 0.05, 0.95)
       4. Generate roll = seededRandom() → success = roll < modifiedChance.
       5. Decide resultEffects (successEffects / failureEffects). Effects are arrays of {stat, delta}.
   * **AI step (optional):** AI can propose additional effects or flavor text based on context, but only after backend validates returned JSON schema.
4. **Apply effects**
   * For each effect: stats[stat] = clamp(stats[stat] + delta, statMin, statMax).
   * Record effect in timeline with result JSON (so you can reproduce).
   * If choice triggers chained\_event\_id, enqueue for a future age or immediate processing.
5. **Advance time & energy**
   * Many designs: each event uses **energy**, not necessarily fixed time. But also have a **time progression** granularity (age in years or milestones).
   * Example: you define event.ageAdvance = 0|1|nYears or each event increments an in-run turnCounter. At end of milestone (e.g., end of adolescence), age increments + energy reset.
   * Simpler: treat each event as an atomic "turn" and map certain number of turns → 1 year (configurable).
6. **Life end detection**
   * If HP <= 0 → death early (ending type = bad).
   * If age > maxAge or reach configured end milestone → compute run score and finalize ending.

# **4) Formulas & probability model (concrete)**

Use simple, explainable math; easy to tune.

**Modifier calculation** cpp  
CopyEdit  
totalBonus = sum( statBonuses + talentBonuses + perkBonuses )

// e.g. luck contributes linearly: luckBonus = Luck \* 0.005

// skills add: if Knowledge >= required then skillBonus += 0.15

**Success chance** ini  
CopyEdit  
modifiedChance = clamp(baseChance \* (1 + totalBonus) - riskPenalty, 0.05, 0.95)

* + baseChance in [0.0..1.0], riskPenalty in [0..0.5].
  + clamp to avoid near-0 or near-1 extremes.

**Stat application** ini  
CopyEdit  
newStat = clamp(oldStat + delta, minStat, maxStat)

* **Energy consumption**
  + Event defines energyCost. If currentEnergy < energyCost, either block or force alternative (e.g., "skip" or force rest).

**Run scoring (meta points)** Example:  
  
 ini  
CopyEdit  
survivalPoints = age \* survivalFactor (e.g., age \* 10)

eventPoints = numberOfEventsTriggered \* 5

goalPoints = sum(each mini-goal completed -> assigned points)

endingMultiplier = (ending == 'good' ? 1.5 : ending == 'normal' ? 1.0 : 0.7)

runScore = floor((survivalPoints + eventPoints + goalPoints) \* endingMultiplier)

* Use this to grant meta points.

# **5) Event schema (JSON example, suggested)**

json

CopyEdit

{

"id": "evt\_011\_coach\_offer",

"type": "opportunity",

"title": "Huấn luyện viên đề nghị",

"description": "Một huấn luyện viên nhìn thấy bạn chơi...",

"min\_age": 16,

"max\_age": 25,

"prerequisites": { "stat": { "sports": 30 } },

"branch\_weight": 0.8,

"choices": [

{

"id": "c1",

"text": "Chấp nhận lời mời",

"effects\_on\_success": [ { "stat": "sports", "delta": 20 }, { "stat": "finance", "delta": -10 } ],

"effects\_on\_failure": [ { "stat": "mood", "delta": -15 } ],

"baseChance": 0.6,

"energyCost": 10,

"chained\_event\_id\_on\_success": "evt\_020\_local\_competition"

},

{

"id": "c2",

"text": "Từ chối, tập luyện tự do",

"effects": [ { "stat": "sports", "delta": 5 }, { "stat": "energy", "delta": -5 } ],

"energyCost": 5

}

],

"tags": ["sports", "opportunity"]

}

* Note: effects = deterministic; effects\_on\_success/on\_failure used with checks.

# **6) Deterministic RNG & reproducibility**

* Use seeded PRNG stored in runs.seed + a stepCounter in run state. Always pass PRNG through selection/resolution functions.
* Example library: seedrandom (JS) or Xorshift / Mulberry32.
* Implementation pattern:  
  + prng = createPRNG(seed)
  + For each pick/roll: roll = prng.next(); increment run.step.
* Store seed + timeline so any run can be replayed exactly.

# **7) Event selection algorithm (pseudocode)**

ts

CopyEdit

function selectNextEvent(runState, prng) {

candidates = events.filter(e => e.isActive && e.minAge <= runState.age && e.maxAge >= runState.age

&& prerequisitesSatisfied(e.prerequisites, runState));

// Compute weight

weights = candidates.map(e => e.branchWeight \* e.rarityFactor \* contextModifier(e, runState));

// Weighted random pick

idx = weightedRandomIndex(weights, prng);

return candidates[idx];

}

* contextModifier may boost events based on traits (e.g., art skill increases art-related events weight).

# **8) Chaining events & convergence**

* When a choice sets a flag or triggers chained\_event\_id, store it in run state as scheduledEvents with executeAtAge or executeAfterTurns. This avoids explosion.
* Use **convergence nodes**: design canonical events many chains can rejoin to keep graph manageable.

# **9) AI role & validation**

* AI generates flavored description, variant choices, or entire event objects from templates.
* **Critical**: ALWAYS validate AI output against JSON schema (Zod/AJV). If fail → retry limited times → fallback to curated event.
* Cache AI outputs in ai\_cache keyed by signature\_hash to reuse and reduce cost.

# **10) Talents & Perks (how they modify)**

* **Talent (persistent)**: e.g., "Guitar Prodigy" → increases art stat gains by 20% and increases chance to trigger music events by +15% weight.
* **Perk (spendable/persistent)**: e.g., "Lucky Charm" → adds flat +0.15 to modifiedChance for check-based choices.
* Implementation: define a function getModifiersForRun(run) that aggregates all active perks/talents to supply to resolution functions.

# **11) Mini-goals & milestones**

* Each milestone (e.g., "graduate by age 22") is represented by an object {id, type, targetStat, threshold, deadlineAge, pointsReward}.
* Evaluate at checkpoint and give goalPoints accordingly.

# **12) Data to log for balancing & analytics**

* For each choice: (runId, eventId, choiceId, playerAge, playerStatsSnapshot, resolvedOutcome, prngRoll, modifiersApplied).
* Track event frequencies and choice distributions to detect over/under-used events.
* Use this to tune branch weights, baseChance, and effect magnitudes.

# **13) Example resolveChoice pseudocode (TypeScript-like)**

ts

CopyEdit

async function resolveChoice(runId, eventId, choiceId) {

const run = await db.getRun(runId);

const event = await db.getEvent(eventId);

const choice = event.choices.find(c => c.id === choiceId);

// energy check

if (run.currentEnergy < (choice.energyCost || 0)) return { error: "Not enough energy" };

// compute modifiers

const modifiers = computeModifiers(run); // includes luck, talents, perks, stat bonuses

if (choice.baseChance != null) {

const base = choice.baseChance;

const modified = clamp(base \* (1 + modifiers.totalBonus) - modifiers.riskPenalty, 0.05, 0.95);

const roll = run.prng.next(); // 0..1

const success = roll < modified;

const effects = success ? choice.effects\_on\_success || choice.effects : choice.effects\_on\_failure || [];

applyEffects(run, effects);

if (success && choice.chained\_event\_id\_on\_success) scheduleEvent(run, choice.chained\_event\_id\_on\_success);

recordTimeline(run, event, choice, { success, roll, modified, appliedEffects: effects });

decrementEnergy(run, choice.energyCost || 0);

return { success, appliedEffects: effects };

} else {

// deterministic effects

applyEffects(run, choice.effects || []);

recordTimeline(...);

decrementEnergy(...);

return { appliedEffects: choice.effects };

}

}

# **14) Balancing & playtesting strategy**

* Start with **small event pool (30–50)**, playtest, record metrics.
* Iterate:  
  + If a stat skyrockets too fast → lower deltas.
  + If players never see some event → increase branch\_weight.
* Use role-based testing: simulate thousands of runs with automated bots to get distribution of endings, average run length, event frequency.

# **15) Tips to avoid combinatorial explosion**

* Use **parametric events** (templates): single template + different parameters (names, places) instead of distinct events.
* Use **state variables** (flags) and convergence nodes rather than exclusive long deep trees.
* Limit depth of AI-generated unique branches; prefer creating variants that map back to canonical outcomes.

# **16) Recommended default values to start with (practical)**

* Stat ranges: 0..100.
* Starting stats by background:  
  + Rich: finance 300, status 70, luck +5
  + Normal: finance 100, status 40
  + Poor: finance 10, status 20, luck -5
* Energy per milestone: 100.
* Chance clamp: min 5%, max 95%.

# **17) Debugging & reproducibility**

* Always save run.seed and timeline. To reproduce a run: re-create PRNG with seed and replay choices in timeline.
* Log PRNG outputs in timeline entries (store roll and modifiedChance).

# **18) Next concrete deliverables I can produce for you now**

* (A) **TypeScript interfaces** / Zod schemas for Event, Choice, Effect, RunState.
* (B) **TS pseudocode module** for selectNextEvent, resolveChoice, applyEffects, including seeded RNG implementation.
* (C) **Sample event JSON** set expanded from the ones you provided, including success/failure effects and chained events.
* (D) A small **test harness** script (Node) that simulates 1000 runs and outputs distribution of endings for balancing.

Bạn muốn mình bắt đầu với (A), (B), (C) hay (D) ngay bây giờ? Mình sẽ generate code/schemas/tests bạn có thể dán vào project.