Design Document: Spawn Logic & Balancing Overhaul

1. Executive Summary

The current spawn system, while functional, centralizes its balancing logic in `SpawnManager.java`. This class contains hardcoded budgets, spawn pools (via the inferred `SpawnData` class), and complex `switch` statements for determining monster/item "colors" (tiers). This design is rigid, difficult to balance, and hard to extend. Every new item or monster variant requires recompiling Java code.

This document proposes a **data-driven refactor** that moves all spawn-related logic (budgets, pools, and variant progression) into JSON files. This will make the system consistent, highly extensible, and allow for rapid balancing by simply editing text files.

We will achieve this by:

1. **Expanding `monsters.json` and `items.json`:** We will add "variant" data to these files, defining the level progression (e.g., "Blue Ant" vs. "Pink Ant") directly in the monster/item's definition.
2. **Creating `spawntables.json`:** A new data file that defines spawn budgets, weighted spawn pools, and container contents per dungeon level.
3. **Refactoring `SpawnManager.java`:** This class will be simplified into a *spawn executor*. It will read the new JSON data and use it to build weighted tables, removing all hardcoded logic.

The `LootTable.java` system for modifiers is already excellent and data-driven; we will use it as our model and apply its principles to the rest of the spawn logic.

2. Current System Analysis & Deficiencies

1. **`MazeChunkGenerator.java`:**
   - **Problem:** Hardcodes potion spawns ('S' and 'H') during `createMazeFromText`. This is inconsistent with the `SpawnManager`'s role.
   - **Solution:** This logic will be removed. All item spawns, including potions, will be handled by the new `spawntables.json`.
2. **`SpawnManager.java`:**
   - **Problem 1: Hardcoded Budgets:** `monsterBudget` and `itemBudget` are calculated with simple, linear formulas (e.g., `3 + (level / 3)`). This doesn't allow for custom-tuning difficulty spikes or "treasure" floors.
   - **Problem 2: Hardcoded Spawn Pools:** The system relies on static lists in the `SpawnData` class (e.g., `SpawnData.TIER1_ITEMS`,

    `SpawnData.BAD_MONSTERS`). Adding a new "Tier 1" monster requires editing this Java file.
    - **Problem 3: Hardcoded Variant Logic:** The `getRandomMonsterColor` and `getRandomItemColor` methods are massive, level-based `switch` statements. This is the most brittle part of the design. If we want a "Pink Ant" to appear one level earlier, we have to modify this complex method. This logic should be part of the "Giant Ant" definition itself.
    - **Problem 4: Inconsistent Spawning:** Containers, items, and monsters all use slightly different logic paths.

3. Proposed System Architecture

Our new architecture will consist of three parts:

1. **Entity Definitions (The "What"):** `items.json` and `monsters.json`
2. **Spawn Tables (The "Where & When"):** `spawntables.json`
3. **Spawn Executor (The "How"):** The refactored `SpawnManager.java`

3.1. Proposal 1: Expand `monsters.json` and `items.json`

We will add a new `variants` list to each monster and item definition. This list will define all possible "colors" (tiers) for that entity, their spawn levels, and their spawn weight. This completely replaces the `getRandomMonsterColor` and `getRandomItemColor` methods.

Example: `monsters.json` (partial)

JSON
```
"GIANT_ANT": {
  "warStrength": 5,
  "spiritualStrength": 15,
  "armor": 1,
  "baseExperience": 10,
  "family": "BEAST",
  "texturePath": "images/monsters/giant_ant.png",
  "spriteData": [...],
  "scale": {...},
  "warDamage": 1,
  "spiritDamage": 0,
  "damageType": "PHYSICAL",
  "intelligence": 0,
  "onHitEffects": [...],
  "variants": [
    { "color": "BLUE",   "minLevel": 1, "maxLevel": 5, "weight": 10 },
    { "color": "PINK",   "minLevel": 3, "maxLevel": 10, "weight": 5 },
    { "color": "PURPLE", "minLevel": 8, "maxLevel": 99, "weight": 2 }
  ]
}
```

Example: `items.json` (partial)

JSON
```
"BOW": {
  "friendlyName": "Bow",
  "texturePath": "images/items/bow.png",
  ...
  "isWeapon": true,
  "isRanged": true,
  ...
  "variants": [
    { "color": "TAN",    "minLevel": 1, "maxLevel": 8, "weight": 10 },
    { "color": "ORANGE", "minLevel": 1, "maxLevel": 8, "weight": 8 },
    { "color": "BLUE",   "minLevel": 5, "maxLevel": 15, "weight": 5 },
    { "color": "GRAY",   "minLevel": 5, "maxLevel": 15, "weight": 5 },
    { "color": "YELLOW", "minLevel": 12, "maxLevel": 99, "weight": 3 },
    { "color": "WHITE",  "minLevel": 12, "maxLevel": 99, "weight": 3 }
  ]
}
```

3.2. Proposal 2: Create `assets/data/spawntables.json`

This new file will replace all remaining hardcoded logic from `SpawnManager` and `SpawnData`. It defines spawn *budgets* per level and the *weighted tables* for *which* entities to spawn.

Proposed `spawntables.json` structure:

JSON
```
{
  "levelBudgets": [
    { "level": 1, "monsterBudget": 3, "itemBudget": 5, "containerBudget": 1 },
    { "level": 2, "monsterBudget": 4, "itemBudget": 5, "containerBudget": 1 },
    { "level": 3, "monsterBudget": 4, "itemBudget": 5, "containerBudget": 2 },
    { "level": 4, "monsterBudget": 5, "itemBudget": 6, "containerBudget": 2 },
    { "level": 5, "monsterBudget": 6, "itemBudget": 6, "containerBudget": 2, "note": "First boss floor?" }
  ],

  "monsterSpawnTable": [
    { "type": "GIANT_ANT",       "minLevel": 1, "maxLevel": 10, "weight": 20 },
    { "type": "DWARF",           "minLevel": 1, "maxLevel": 12, "weight": 15 },
    { "type": "GIANT_SCORPION",  "minLevel": 2, "maxLevel": 14, "weight": 15 },
    { "type": "GHOUL",           "minLevel": 3, "maxLevel": 15, "weight": 10 },
    { "type": "SKELETON",        "minLevel": 3, "maxLevel": 15, "weight": 12 }
  ],

  "itemSpawnTable": [
    { "type": "BOW",             "minLevel": 1, "maxLevel": 99, "weight": 5 },
    { "type": "KNIFE",           "minLevel": 1, "maxLevel": 99, "weight": 5 },
    { "type": "AXE",             "minLevel": 1, "maxLevel": 99, "weight": 5 },
    { "type": "SMALL_SHIELD",    "minLevel": 2, "maxLevel": 99, "weight": 4 },
    { "type": "HELMET",          "minLevel": 3, "maxLevel": 99, "weight": 3 },
    { "type": "POTION_BLUE",     "minLevel": 1, "maxLevel": 99, "weight": 10 },
    { "type": "POTION_PINK",     "minLevel": 1, "maxLevel": 99, "weight": 10 },
    { "type": "SCROLL",          "minLevel": 2, "maxLevel": 99, "weight": 8 }
  ],

  "containerSpawnTable": [
    { "type": "BOX", "minLevel": 1, "maxLevel": 99, "weight": 10, "keyColor": "CONTAINER_TAN" },
    { "type": "BOX", "minLevel": 5, "maxLevel": 99, "weight": 5,  "keyColor": "CONTAINER_ORANGE" },
    { "type": "BOX", "minLevel": 12, "maxLevel": 99, "weight": 2, "keyColor": "CONTAINER_BLUE" }
  ],

  "containerLoot": {
    "default": [
      { "type": "COINS",   "minLevel": 1, "maxLevel": 99, "weight": 10 },
      { "type": "CHALICE", "minLevel": 3, "maxLevel": 99, "weight": 5 },
      { "type": "INGOT",   "minLevel": 5, "maxLevel": 99, "weight": 3 }
    ]
  }
}
```

3.3. Proposal 3: Refactor `SpawnManager.java` (The "Executor")

The `SpawnManager` class will be heavily simplified.

1. It will be loaded with the new `SpawnTableData` (parsed from `spawntables.json`) in its constructor.
2. All `spawn...` methods will be rewritten to use the same, consistent logic.
3. All hardcoded lists, budgets, and `switch` statements will be removed.

New `SpawnManager.spawnMonsters()` Pseudocode:

```
Java
private void spawnMonsters() {
    // 1. Get budget from levelBudgets table
    int budget = spawnTableData.getBudgetForLevel(this.level).monsterBudget;

    // 2. Build weighted list of possible monsters
    List<SpawnTableEntry> validMonsters = spawnTableData.monsterSpawnTable
        .filter(entry -> level >= entry.minLevel && level <= entry.maxLevel);

    WeightedRandomList<String> monsterPicker = new WeightedRandomList<>(validMonsters);

    // 3. Loop and spawn
    for (int i = 0; i < budget; i++) {
        GridPoint2 spawnPoint = getEmptySpawnPoint();
        if (spawnPoint == null) break;

        // 4. Pick a monster type (e.g., "GIANT_ANT")
        String monsterType = monsterPicker.getRandomEntry();

        // 5. Get a *variant* (color, stats) for that type from the MonsterDataManager
        // This call uses the new "variants" list from Proposal 1
        MonsterVariant variant = dataManager.getRandomVariantForMonster(monsterType,
this.level);

        if (variant == null) continue; // No valid variant for this level

        // 6. Create the monster
        Monster monster = new Monster(monsterType, spawnPoint.x, spawnPoint.y, variant.color,
...);
        monster.scaleStats(level);
        maze.addMonster(monster);
    }
}
```

This *exact same logic pattern* will be applied to `spawnItems`, `spawnContainer`, etc., achieving the "consistent random generation" goal.

---

4. Implementation Plan & Milestones

This refactor will be conducted in phases to ensure the game remains testable.

Milestone 1: Data Model Expansion (No Logic Change)

- **Task 1.1:** Create new Java helper classes to model the new JSON data:
  - `core/.../gamedata/monster/MonsterVariant.java` (stores `color`, `minLevel`, `maxLevel`, `weight`)
  - `core/.../gamedata/item/ItemVariant.java` (stores `color`, `minLevel`, `maxLevel`, `weight`)
- **Task 1.2:** Update `core/.../gamedata/monster/MonsterTemplate.java` and `core/.../gamedata/item/ItemTemplate.java` to include a `public List<MonsterVariant> variants;` and `public List<ItemVariant> variants;`.
- **Task 1.3:** Update `MonsterDataManager` and `ItemDataManager` to parse this new `variants` list from the JSON.
- **Task 1.4: (Data Entry)** Go through `SpawnManager.getRandomMonsterColor` and `getRandomItemColor` and manually add the new `variants` blocks to `assets/data/monsters.json` and `assets/data/items.json`, translating the Java logic into JSON data.
- **Task 1.5:** Add new helper methods:
  - `MonsterDataManager.getRandomVariantForMonster(String type, int level)`
  - `ItemDataManager.getRandomVariantForItem(String type, int level)`
  - These methods will filter the `variants` list by level, build a weighted random picker, and return a single `MonsterVariant`/`ItemVariant`.
- **Milestone 1 Check:** Compile and run. The game must run **exactly as before**. No logic has changed yet; we have only added new data and unused helper methods.

Milestone 2: Refactor `SpawnManager` (The "Swap")

- **Task 2.1:** Modify `SpawnManager.spawnMonsters`.
  - Instead of calling `getRandomMonsterColor(type)`, you will now call `dataManager.getRandomVariantForMonster(type.name(), level)`.

    - The returned `MonsterVariant` object's `color` field will be used to create the monster.
    - Remove the (now-empty) `getRandomMonsterColor` method.
- **Task 2.2:** Modify `SpawnManager.spawnRegularItem`.
    - Do the same as above, calling `itemDataManager.getRandomVariantForItem(type.name(), level)` and removing `getRandomItemColor`.
- **Task 2.3:** Modify `SpawnManager.spawnContainer`.
    - For now, we will leave `getContainerColorForLevel` as-is. We will replace it in Milestone 3.
- **Milestone 2 Check:** Compile and run. The game should *functionally* be identical, but the two largest and most complex methods in `SpawnManager` are now gone, replaced by data-driven calls. This is a major code-health improvement.

Milestone 3: Data-Driven Spawn Tables (The "Logic Rip")

- **Task 3.1:** Create the new file `assets/data/spawntables.json` with the structure proposed in section 3.2, using the *exact* budgets and `SpawnData` lists from the current `SpawnManager` to ensure 1:1 parity at first.
- **Task 3.2:** Create new Java classes to parse this file (e.g., `SpawnTableData`, `LevelBudget`, `SpawnTableEntry`).
- **Task 3.3:** In `Tarmin2.java` (or your main game loader), load this JSON file into an object.
- **Task 3.4:** Update `WorldManager` and `MazeChunkGenerator` to pass the `SpawnTableData` object to the `SpawnManager`'s constructor.
- **Task 3.5: Rewrite `SpawnManager.spawnMonsters()`.**
    - Remove all `SpawnData` imports.
    - Remove the hardcoded `monsterBudget` calculation.
    - Implement the new logic from the pseudocode in section 3.3, which reads from the `SpawnTableData` object.
- **Task 3.6: Rewrite `SpawnManager.spawnItems()` and `SpawnManager.spawnContainer()`.**
    - Repeat the process. This will also replace the `getContainerColorForLevel` method, as container types/colors will now be read from the `containerSpawnTable`.
- **Task 3.7: Rewrite `SpawnManager.spawnContainer()`'s loot logic.**
    - Instead of spawning a hardcoded "treasure," it will build a weighted list from `containerLoot.default` in the `spawntables.json` and spawn a random item from that.

- **Milestone 3 Check:** Compile and run. This is the most significant change. We must test carefully that all entity types (monsters, items, containers, container-loot) are spawning at the correct levels and in roughly the correct quantities.

Milestone 4: Final Clean-up

- **Task 4.1:** In `MazeChunkGenerator.createMazeFromText`, **delete** the lines that spawn potions for 'S' and 'H'. This is now handled by the `itemSpawnTable`.
- **Task 4.2:** Delete the (now un-used) `SpawnData.java` file.
- **Task 4.3:** Review `SpawnManager.java` one last time. It should be dramatically smaller, cleaner, and contain no hardcoded balancing logic.
- **Milestone 4 Check:** The refactor is complete. The game is fully functional, and we can now balance the entire game's spawn logic—from budgets to spawn rates to tier progression—by editing the three JSON files: `monsters.json`, `items.json`, and `spawntables.json`