# Feature Definition – Mod-Friendly Anatomy System

*Version 1.0 – June 28 2025*

---

## 1 Purpose

Deliver a **100 % data-driven** anatomy pipeline that lets modders describe creatures with arbitrary body parts, counts, and tags, while the engine assembles a valid, socket-aware graph, supports procedural variation, and handles runtime events such as limb detachment.

---

## 2 Key Concepts & Glossary

| Term | Meaning |
|---|---|
| **Recipe** | Designer-authored file that states *what* parts a creature should have (types, tags, counts, preferences, exclusions). |
| **Blueprint** | Graph definition that states *where* parts can attach (sockets, orientation, joint meta). |
| **Socket** | Attachment point declared on a part; defines allowed child types plus joint physics & naming info. |
| **Part Definition** | Entity file that represents an individual body part and advertises its own sockets (if any). |
| **Factory** | Runtime service that combines a blueprint with a recipe and spawns the final entity graph. |
| **Validator** | Dev-time service that guarantees produced graphs respect socket limits and recipe constraints. |

---

## 3 File Formats

### 3.1 Recipe (`*.recipe.json`)

jsonc
CopyEdit

```
{
  "$schema": "anatomy.recipe.schema.json",
  "recipeId": "humanoid_female",
  "slots": {
    "<slotKey>": {
```

```
      "partType": "breast",                  // required
      "preferId": "anatomy:human_left_breast", // optional exact
pick
      "tags":    ["anatomy:large","anatomy:meaty"],
      "notTags": ["anatomy:sagging"],
      "count":   { "min": 2, "max": 2 }     // or { "exact": N }
    },
    "...": {}
  },
  "constraints": {
    "requires": [ ["anatomy:heart","anatomy:artery"] ],     //
co-presence
    "excludes": [ ["anatomy:gills","anatomy:lung"] ]        //
mutual exclusion
  },
  "includes": [ "macro:standard_limbs" ]    // optional macro import
}
```

- **Counts are *soft*** – clamped by socket availability.
- **Tags / notTags** provide must-have / must-lack filtering. Note: these tags are simple components.
- **preferId** overrides randomness if the referenced entity is present.
- Schema delivered as `anatomy.recipe.schema.json` for editor linting.

---

### 3.2 Blueprint (`*.bp.json`) – graph root

jsonc
CopyEdit

```
{
  "$schema": "anatomy.blueprint.schema.json",
  "root": "anatomy:torso_female",          // entry node
  "attachments": [                         // static parent→child
edges
    { "parent":"anatomy:torso_female", "socket":"leg_left",
"child":"anatomy:leg_human" },
    { "parent":"anatomy:torso_female", "socket":"leg_right",
"child":"anatomy:leg_human" }
  ]
```

```
}
```

*Blueprints may be generated by visual tools; they never reference tags.*

---

### 3.3 Part Definition (`*.part.json`) – includes sockets

jsonc
CopyEdit

```
{
  "id": "anatomy:leg_human",
  "components": [
    { "type":"anatomy:part", "subType":"leg" },
    { "type":"anatomy:sockets",
      "sockets":[
        {
          "id":"ankle",
          "orientation":"mid",
          "allowedTypes":["foot","paw"],
          "maxCount":1,
          "jointType":"hinge",
          "breakThreshold":25,
          "nameTpl":"{{orientation}} {{type}}"
        }
      ]
    }
  ],
  "tags": ["anatomy:limb","anatomy:lower"]
}
```

#### Socket Fields (new & existing)

| Field | Type | Description |
|---|---|---|
| `id` | string | Unique within parent part. |
| `orientation` | `"left" \| "right"` | Drives name generation. |
| `allowedTypes` | string[] | Child `partType` whitelist. |
| `maxCount` | int ≥ 1 | Hard limit enforced by validator. |

| | | |
|---|---|---|
| `jointType` | `"hinge" \| "ball" \| "suture" \| "fused"` | Physics metadata. |
| `breakThreshold` | int ≥ 0 | Damage needed to sever at runtime. |
| `nameTpl` | string | Auto-name template using tokens `{{orientation}}`, `{{type}}`, `{{index}}`, `{{parent.name}}`. |

---

## 4 Runtime Algorithm (BodyBlueprintFactory)

1. **Load Blueprint** → instantiate root & static children.
2. **Depth-first over sockets** in deterministic order.
3. For each socket:
    - Find matching **recipe slot** by `partType`.
    - Build candidate list: `partType` match ∧ required `tags` present ∧ `notTags` absent.
    - If `preferId` in set ⇒ choose it; else RNG pick (seeded for reproducibility).

Instantiate child, attach via edge component:

json
CopyEdit

```
{
  "type":"anatomy:joint",
  "parentId":"leg-123",
  "socketId":"ankle",
  "jointType":"hinge",
  "breakThreshold":25
}
```

    - 
    - Name child via `nameTpl`.
4. **Clamp counts**: stop filling when socket quota reached.

5. After graph complete, run **GraphIntegrityValidator**:
    - Exceeds `maxCount` ⇒ error.
    - Recipe `requires` / `excludes` satisfied.
6. If valid ⇒ spawn entity graph; else → reject mod at load.

---

## 5 Macro Layer (optional authoring sugar)

*Any* recipe can include `isMacro: true`.
A CLI pre-processor resolves `includes`, variable substitutions, and range shorthands → outputs canonical JSON before validation. **No runtime impact.**

---

## 6 Gameplay Systems

| System | Responsibility | Uses |
| --- | --- | --- |
| **BodyGraphService** | Maintains adjacency cache; detaches sub-graphs; emits events (`LIMB_DETACHED`). | joint metadata |

---

## 7 Rules & Guarantees

1. **Blueprint truth wins** – socket counts hard-cap recipe counts.
2. **Recipes never reference mod IDs except via `preferId`**.
3. **Validation is up-front** – invalid mods halt loading; no silent failures except count clamping (logged).

4. **All joints are explicit edge components** – enables targeted damage, physics, networking.

5. **No executable code in data** – security assured; all behaviour lives in engine code.

---

## 8 Extensibility Hooks

- **Hierarchical macros**: macros may import other macros for deep composition.
- **Side numbering**: include `index` in socket meta to auto-name “Tentacle 3”.

---

## 9 Delivery Checklist

- Publish `anatomy.recipe.schema.json` & `anatomy.blueprint.schema.json`.
- Implement Blueprint & Recipe loader (JSON → javascript POCOs).
- Finish `BodyBlueprintFactory` per § 4.
- Wire `GraphIntegrityValidator` to run after factory build.

---

## 10 Non-Goals (v1)

- Vascular flow, nerve routing, metabolic simulation.
- Multi-parent geometry (e.g., clavicles requiring two simultaneous sockets) – earmarked for v1.1.
- Semantic-web / OWL reasoning.

**Ambiguities in the spec itself**

So-called “tags” are simply components. Our entities only have an “id” and “components” property. The system often uses component “markers” to depict capabilities or types.

Soft counts vs. min: Spec says counts are “soft – clamped by socket availability” What happens if min > available sockets? Fail hard or warn and drop? Answer: the system should likely log a warning; perhaps the modder or designer needs to edit either the recipe or the blueprint, or wrote in the wrong blueprint. Otherwise, the body gets assembled according to what the blueprint allows.
preferId conflicts: If preferId points to a part disallowed by allowedTypes in the socket, which wins? Answer: the list of disallowed types in the socket wins, perhaps emitting a warning, because that could be a mistake by the modder.

Macro preprocessing: Cyclic includes are possible. Need an explicit cycle-detection rule.

**Edge cases:**

**Orphan sockets** – blueprint exposes a socket but no recipe slot matches. Decide: leave empty? spawn placeholder? validation fail? Answer: the recipe is not meant to explicitly tell the blueprint all body parts that will fit all available sockets; it just lets the modder or designer express what he or she wants the final body to have. The sockets that the recipe doesn’t provide a solution for will be “filled” with body parts chosen at random, as long as they fit what that socket allows. Example: if a socket only allows “anatomy:limb”, then any random entity with “anatomy:limb” will be picked.

**Detachment cascades** – Removing a parent part with children attached (e.g., delete upper arm while forearm still present). Does `BodyGraphService` automatically detach the subtree and fire LIMB_DETACHED for each? What about event order? Answer: I think that there should be just one LIMB_DETACHED event fired, and that’s for whatever body part was detached from a socket. If there’s anything connected to that detached body part, it continues attached to the detached bodypart.

**Name collisions** – `nameTpl` using `"{{parent.name}}"` can easily generate identical strings, wreaking havoc in UI or save files. Answer: if the code that assembles the names detects that it has produced a body part with the exact same name (e.g. two “Left breast”), then a warning should be issued. Likely the modder that created the blueprint will have to fix that.

Logging & diagnostics: Modders need to know *why* their part failed the validator. Structured log with partId, socketId, rule hit, severity.

**Duplicate IDs across two mods.** Which one wins? Namespacing rules? Answer: duplicate IDs are forbidden by the mod loading system. That shouldn’t happen; it will crash the loading process.

**Blueprint references a part that itself defines a socket whose `maxCount` is zero** (or omitted). Should that be legal? Answer: No. A socket should have a minimum count of 1. Max should never be zero.

**Recursive attachment loops** (e.g., Part A lists a socket that allows Part B, Part B’s socket allows Part A) – factory depth-first algorithm could infinite-loop. Answer: this should be ensured impossible by code. Some dedicated service perhaps that ensures no cyclic situations like these exist.

**Unknown tag / component type** in recipe or part – ignore, warn, or hard-fail? Answer: hard fail. Emit a SYSTEM_ERROR_OCCURRED_ID event, and throw exception.

**Socket orientation mismatch.** Blueprint says left leg attaches to a socket whose orientation=right – allowed? Answer: we probably

shouldn’t allow body parts that by itself express their orientation. The orientation should be exclusively a matter of the socket.

*Error & Logging:*

*-The construction and assembling of a body according to a recipe and blueprint should fail fast upon errors, dispatching a SYSTEM_ERROR_OCCURRED_ID event so that the users know as soon as possible and can fix it.*