I’ve put together a comprehensive Scenario System Guide to help any new developer understand—and safely extend—the way scenarios power “Coin Clash.” It covers:

\* \*\*Definitions & Structure\*\*: how scenarios are authored and represented

\* \*\*Draft-to-JSON Pipeline\*\*: using `parse\_scenarios.py` to convert `.md` drafts into canonical JSON

\* \*\*Runtime Loading\*\*: how `core/scenario\_loader.py` discovers, validates, and IDs scenario files

\* \*\*Engine Integration\*\*: how `MatchEngine` selects, substitutes, and logs scenarios each round

\* \*\*Best Practices for Extensions\*\*: adding or tweaking scenarios without disrupting logic

Feel free to copy this into your project wiki or README.

---

## 1. Definitions & Data Model

1. \*\*Categories\*\*

Scenarios are grouped into six buckets:

\* `direct\_kill`, `self`, `environmental`, `group`, `story` (non-lethal), and `comeback`&#x20;

2. \*\*JSON Structure\*\*

Each category’s JSON file (e.g. `scenarios/direct\_kill.json`) is a list of objects:

```json

[

{ "text": "[Character A] spins her umbrella rifle and pops [Character B] off the diving board." },

…

]

```

At load time, each item receives an `"id"` of the form `"{category}\_{index:03d}"` .

3. \*\*Placeholders\*\*

\* Use `[Character A]`, `[Character B]`, etc., to indicate where characters are injected.

\* The engine counts unique placeholders to know how many participants to sample .

---

## 2. Draft-to-JSON Pipeline

`parse\_scenarios.py` allows maintaining scenarios in Markdown drafts (`Scenarios.md` and `new\_scenarios\_draft.md`), then regenerating the JSON files:

1. \*\*Category Mapping\*\*

Headlines like `### Direct Kills` map to `"direct\_kill"`, etc., via `CATEGORY\_MAP` .

2. \*\*Parsing Logic\*\*

\* Lines starting `\* ` or numbered (`1. `) under each heading become scenario texts.

\* Strips list markers, bold titles, and trailing “(Item Find: …)” notes.

3. \*\*Combining & Emitting\*\*

\* Merges “existing” and “new” sets, then writes each non-empty category to `scenarios/{category}.json`.

> \*\*When to run\*\*: after editing drafts, execute `python parse\_scenarios.py` to refresh `core/scenarios/\*.json`.

---

## 3. Runtime Loading

The function `load\_scenarios()` in `core/scenario\_loader.py`:

1. \*\*Discovery\*\*

Scans `core/scenarios/` for `\*.json`, matching filenames against `EXPECTED\_CATEGORIES` .

2. \*\*Validation & ID Assignment\*\*

\* Loads each JSON list, filters for objects with a string `"text"` field.

\* Adds `"id": "{category}\_{i:03d}"` to each entry.

3. \*\*Result\*\*

Returns a dict `{ category: [ {id, text}, … ], … }`, with warnings for any empty categories (except `comeback`) .

---

## 4. Engine Integration

Within `MatchEngine` (`core/engine.py`):

1. \*\*Event-Type → Category\*\*

Maps round events to scenario lists via `EVENT\_TYPE\_TO\_CATEGORY`:

````python

EVENT\_TYPE\_TO\_CATEGORY = {

"direct\_kill": "direct\_kill",

"self": "self",

…,

"extra\_lethal": ["direct\_kill","environmental"],

…

}

``` :contentReference[oaicite:12]{index=12}:contentReference[oaicite:13]{index=13}.

````

2. \*\*Sampling & Substitution\*\*

\* \*\*Sample\*\* one scenario (with replacement) from the chosen category’s pool.

\* \*\*Count\*\* placeholders via regex `r"\[Character ([A-Z])\]"`.

\* \*\*Select\*\* that many distinct characters from the alive (or dead, for comeback) pool.

\* \*\*Substitute\*\* placeholders 1:1 with each character’s `display\_name` .

3. \*\*Effects & Logging\*\*

\* After substitution, the engine applies kills or revivals based on `event\_type`.

\* Each event is persisted as a `MatchEvent` (storing `scenario\_source=id`, `scenario\_text`, and `affected\_character\_ids`) .

\* The same text is appended to the in-memory `match\_log`.

4. \*\*Special Rules\*\*

\* When only two characters remain, skips any “group” or extra lethal events (enforced via early checks) .

---

## 5. Best Practices for Extension

\* \*\*Adding New Scenarios\*\*

1. Edit `new\_scenarios\_draft.md` (or your own draft) under the appropriate heading.

2. Run `python parse\_scenarios.py` to regenerate JSON.

3. Ensure your text uses only `[Character A]`, `[Character B]`, etc., matching the number of participants you intend.

\* \*\*Direct JSON Edits\*\*

\* You may directly add entries to `core/scenarios/\*.json`; just include a `"text"` field.

\* If you need a custom ID, add an `"id"` key—otherwise it will be autogenerated at load time.

\* \*\*Testing Changes\*\*

\* Because randomness is seedable (`SeedableRandom`), use a fixed seed to reproduce flows.

\* Confirm that placeholder counts match available characters; otherwise, events may be skipped with a warning.

\* \*\*Maintaining Category Integrity\*\*

\* Don’t rename JSON files or categories without updating both `EXPECTED\_CATEGORIES` and `EVENT\_TYPE\_TO\_CATEGORY`.

\* If introducing a new event type, extend `EVENT\_TYPE\_TO\_CATEGORY` and adjust `config.yaml` weights accordingly.

---

With this guide, a developer should confidently navigate scenario definitions, pipeline tools, and how these integrate into the match engine—allowing you to add new flavor without risking existing functionality. Let me know if anything needs more detail or examples!