Here’s a comprehensive guide to help anyone new to \*\*Coin Clash\*\* quickly find their way around the codebase, understand how everything connects, and safely extend it for new objectives without breaking existing functionality.

---

## 🗂️ Project Structure

```text

.

├── README.txt

├── \_\_main\_\_.py

├── config.yaml

├── core/ ← Core logic and infrastructure

│ ├── config\_loader.py ← Loads & validates config.yaml

│ ├── scenario\_loader.py ← Loads scenario JSON files into memory

│ ├── models.py ← SQLAlchemy ORM models (Player, Match, Character…)

│ ├── repositories.py ← DB access layer (CRUD for matches, players…)

│ ├── engine.py ← MatchEngine: the heart of the simulation

│ ├── utils.py ← Shared helper functions

│ └── \_\_init\_\_.py

├── scenarios/ ← Prebuilt event pools, one JSON per category

│ ├── direct\_kill.json

│ ├── self.json

│ ├── environmental.json

│ ├── group.json

│ ├── story.json

│ └── comeback.json

├── new\_scenarios\_draft.md ← Master Markdown list of all scenarios

├── parse\_scenarios.py ← Converts new\_scenarios\_draft.md → scenarios/\*.json

├── data/

│ └── coin\_clash.db ← SQLite file used by repositories

├── tests/ ← (Empty) place for unit tests

└── requirements.txt

```

---

## 🔍 Core Modules & How They Tie Together

1. \*\*Configuration (`config\_loader.py` + `config.yaml`)\*\*

\* `config\_loader.py` reads `config.yaml` and validates keys (entry-fees, kill-reward rates, event weights, extra-event chances, lethal modifiers, character-purchase caps, protocol cuts, etc.).

\* All “knobs” to tune match behavior live in `config.yaml`—no hard-coding.

2. \*\*Scenario Management (`scenario\_loader.py` + `scenarios/\*.json`)\*\*

\* On startup, `scenario\_loader` reads each category JSON into named pools (direct\\_kill, environmental, story, group, self, comeback).

\* `parse\_scenarios.py` + `new\_scenarios\_draft.md` exist so you can author scenarios in Markdown and regenerate JSON via:

```bash

python parse\_scenarios.py

```

3. \*\*Data Models & Persistence (`models.py` + `repositories.py` + `data/coin\_clash.db`)\*\*

\* \*\*Models\*\* define ORM classes: `Player`, `Character`, `Match`, `EventRecord`, etc.

\* \*\*Repositories\*\* wrap SQLAlchemy sessions to CRUD players, matches, and record events—so engine logic never writes SQL directly.

\* A lightweight SQLite DB keeps history; you can swap to another RDBMS by updating `config\_loader` and SQLAlchemy connection string.

4. \*\*Match Engine (`engine.py`)\*\*

\* \*\*Initialization\*\*: seeded randomness for reproducibility (unit-test hook).

\* \*\*Purchase Phase\*\*: simulated or real players buy 1–3 characters at configured entry fee.

\* \*\*Round Loop\*\*: until only one character remains, pick next event category by weight, randomly select a scenario, apply its effect (kill, revive, cosmetic, non-lethal bonus).

\* \*\*Economic Flow\*\*: after each kill, award killer  =  `kill\_award\_rate` × entry fee; at end, winner takes pool minus protocol cut tiered by number of chars sold.

\* \*\*Extensibility Hooks\*\*: no direct I/O—just pure functions that update model objects.

5. \*\*Entry Point (`\_\_main\_\_.py`)\*\*

\* Glues it all together:

1. Load config & scenarios

2. Optionally seed RNG

3. Create simulated players (`num\_players\_default`)

4. Run one match, printing events & final payouts

6. \*\*Utilities (`utils.py`)\*\*

\* Misc helpers: random‐choice wrappers, formatted text builders, date/time stamps, etc.

---

## 🔧 How to Safely Extend or Modify

1. \*\*Tuning & New Parameters\*\*

\* \*\*Always\*\* add new “knobs” in `config.yaml` and reference them via `config\_loader`.

\* Avoid hard‐coding defaults anywhere else.

2. \*\*Adding a New Scenario Category\*\*

1. Create `scenarios/your\_category.json` with a list of scenario objects.

2. In `config.yaml` under `primary\_event\_weights:`, add your\\_category with its weight.

3. In `scenario\_loader.py`, ensure you whitelist the new category.

4. If the scenario has new effects, extend the engine’s event‐handling switch to cover them.

3. \*\*Altering Economic Rules\*\*

\* Edit kill‐award rates, protocol cuts, entry fees in `config.yaml`.

\* For more complex payout logic, modify `engine.calculate\_payouts()`—it consumes only model objects and config values.

4. \*\*Changing Match Flow\*\*

\* Round progression logic lives in `MatchEngine.run\_match()` in `engine.py`.

\* You can insert new phases (e.g., looting) by adding new methods and updating the flow diagram there.

5. \*\*Database Schema Changes\*\*

\* Update `models.py` and run an Alembic (or similar) migration script against `coin\_clash.db`.

\* Because all DB operations go through `repositories.py`, the engine code stays untouched.

6. \*\*Integrating a Messaging Adapter\*\*

\* Keep core logic pure: it returns plain‐text event strings and model snapshots.

\* Write a separate “adapter” layer (e.g. `telegram\_adapter.py`) that calls `MatchEngine` and sends messages.

\* No need to modify `engine.py`—just import and reuse.

---

## ✅ Validation Checklist

Make sure your guide (and any changes) still cover all core game points:

\* [ ] \*\*Character Purchases\*\*: 1–3 per player → entry fees enforced (`chars\_per\_player\_min/max` in config).

\* [ ] \*\*Match Start Conditions\*\*: either time cap or player cap (configurable, though time-cap not yet implemented).

\* [ ] \*\*Random Eliminations\*\*: from six scenario pools (direct\\_kill, self, environmental, group, story, comeback).

\* [ ] \*\*Event Weights & Extra Events\*\*: primary weights + non\\_lethal\\_story, extra\\_lethal, comeback chances.

\* [ ] \*\*Kill Rewards & Payouts\*\*: per-kill award rate config; winner takes pool minus protocol cut tiers.

\* [ ] \*\*Cosmetics & Items\*\*: engine hooks for non-lethal scenario buffs (e.g. outfits, discounts).

\* [ ] \*\*Seedable Randomness\*\*: `MatchEngine` accepts a seed for reproducible tests.

\* [ ] \*\*Extensibility Hooks\*\*: modular design—core logic independent of I/O or messaging.

\* [ ] \*\*Scenario Pipeline\*\*: maintain scenario text in Markdown → auto-generate JSON.

\* [ ] \*\*Persistence\*\*: full match/event history stored via repositories into SQLite.

\* [ ] \*\*Tests\*\*: directory ready for unit tests on edge cases (double-kill, empty revival pool, lethal modifiers).

---

With this map in hand, anyone handed the repository can:

1. \*\*Locate\*\* the bits they need to tweak (config vs. engine vs. scenarios).

2. \*\*Understand\*\* how data flows from config → models → engine → DB → output.

3. \*\*Extend\*\* safely by adding new YAML keys, scenario JSON, or adapter layers—without touching unaffected code.

If you’d like further detail on any piece (e.g. a deep dive into `MatchEngine.run\_round()`), let me know!