Here’s a comprehensive, end-to-end guide to integrating SUI (and, eventually, Solana) into your “Coin Clash” project’s economic flow. It’s organized so a new engineer can pick it up, understand where and how to hook in blockchain payments, and extend or modify without breaking existing game logic.

---

## 1. Project Structure & Data Flow Overview

```

.

├── \_\_main\_\_.py # Entry point: loads config, repos, engine

├── config.yaml # Game economics parameters (fees, reward rates, etc.)

├── parse\_scenarios.py # Builds/validates scenario JSON from markdown drafts

├── new\_scenarios\_draft.md # Human-editable scenario source

├── scenarios/ # Event JSON files (group, direct\_kill, self, …)

│ ├── direct\_kill.json

│ ├── environmental.json

│ └── …

├── engine.py # Core MatchEngine: orchestrates rounds & applies game logic

├── scenario\_loader.py # Loads JSON scenarios into memory

├── repositories.py # SQLAlchemy repos: Players, Characters, Rounds, Stats

├── models.py # ORM models (Player, Character, RoundEvent, etc.)

├── utils.py # Helpers (random seed, logging, CLI parsing)

└── data/coin\_clash.db # SQLite persistence for reproducible runs

```

1. \*\*`\_\_main\_\_.py`\*\* wires everything:

\* Loads `config.yaml` via `config\_loader.py`

\* Instantiates repositories (backed by `coin\_clash.db`)

\* Builds a `MatchEngine`, passing in repos & config

\* Kicks off `engine.run\_match()`

2. \*\*`engine.py`\*\* drives the game:

\* \*\*Entry‐fee phase\*\* → \*(hook here)\*

\* Randomly selects & purchases characters for each player

\* Loops through rounds: picks events (lethal/non-lethal) via `scenario\_loader.py`

\* Applies kills, awards via configs (`kill\_award\_rate\_\*` in `config.yaml`)

\* When one player remains → \*\*payout phase\*\* → \*(hook here)\*

3. \*\*`repositories.py`\*\* persists everything:

\* You’ll extend it to record on-chain transaction metadata.

4. \*\*`models.py`\*\* contains your domain objects; you’ll add a `Transaction` model.

---

## 2. Current Economic Flow (Off-Chain)

1. \*\*Configure\*\* entry fees and reward rates in `config.yaml`:

```yaml

min\_fee: 0.5 # SUI (or future Sol) denom units

default\_fee: 1.0

max\_fee: 5.0

kill\_award\_rate\_default: 0.1 # fraction of pool awarded per kill

…

```

2. \*\*Collect fees\*\* (purely simulated today) before match starts.

3. \*\*Track kills\*\* in-memory; compute award per kill as

```

award = kill\_award\_rate\_default × total\_pool

```

4. \*\*Distribute awards\*\* at end to survivors / killers.

---

## 3. High-Level SUI Integration Architecture

Adopt an \*\*Adapter pattern\*\* so you can plug in multiple blockchains:

```

blockchain/

├── \_\_init\_\_.py

├── adapter.py # defines IBlockchainAdapter interface

├── sui\_adapter.py # implements SUI-specific calls

└── solana\_adapter.py # (future) Solana implementation

```

\* \*\*`IBlockchainAdapter`\*\* exposes:

```python

class IBlockchainAdapter(ABC):

@abstractmethod

collect\_entry\_fee(self, match\_id: str, player\_id: str, amount: Decimal) -> str

@abstractmethod

distribute\_reward(self, match\_id: str, player\_id: str, amount: Decimal) -> str

@abstractmethod

get\_balance(self, address: str) -> Decimal

…

```

\* In \*\*`\_\_main\_\_.py`\*\*, read `blockchain` section from `config.yaml` to choose adapter:

```yaml

blockchain:

provider: sui # or "solana" in future

sui:

rpc\_endpoint: "https://fullnode.sui.io"

coin\_type: "0x2::sui::SUI"

…

```

\* Pass the instantiated adapter into `MatchEngine`:

```python

adapter = get\_adapter(config.blockchain)

engine = MatchEngine(repos, config, blockchain\_adapter=adapter)

```

---

## 4. Detailed Integration Points

Below is a map of each economic hook—where to call your adapter—and the files/methods to update.

| Flow Step | File / Class / Method | Integration Action |

| --------------------------- | -------------------------------------------- | --------------------------------------- |

| \*\*1. Entry Fee Collection\*\* | `MatchEngine.\_collect\_entry\_fees()` (create) | After loading players, iterate and call |

> `tx\_id = adapter.collect\_entry\_fee(match\_id, player.id, fee)`

> Save `tx\_id` via new `repositories.record\_transaction(...)`. |

> \| \*\*2. Character Purchase\*\* | `MatchEngine.purchase\_characters()` | (Optional) If you monetize character buys beyond entry, wrap each purchase in

> `adapter.collect\_entry\_fee(...)` |

> \| \*\*3. Kill Award Allocation\*\*| `MatchEngine.\_allocate\_kill\_award(killer, victim)` | After computing on-chain award amount:

> `tx\_id = adapter.distribute\_reward(match\_id, killer.id, award)` |

> \| \*\*4. Final Payout\*\* | `MatchEngine.end\_match()` | Compute survivor shares; loop survivors:

> `adapter.distribute\_reward(match\_id, player.id, survivor\_share)` |

> \| \*\*5. Transaction Persistence\*\* | `models.py` → add `Transaction` model

```python

class Transaction(Base):

\_\_tablename\_\_ = "transactions"

id = Column(Integer, primary\_key=True)

match\_id = Column(String, index=True)

player\_id = Column(String, index=True)

tx\_type = Column(String) # ENTRY\_FEE, KILL\_REWARD, FINAL\_PAYOUT

amount = Column(Numeric)

tx\_hash = Column(String, unique=True)

status = Column(String) # PENDING, CONFIRMED, FAILED

timestamp = Column(DateTime, default=datetime.utcnow)

```

\| Also extend `repositories.py` with `record\_transaction(...)`, `update\_transaction\_status(...)`.|

---

## 5. Making It Modular for Solana

1. \*\*Implement `SolanaAdapter`\*\* in `blockchain/solana\_adapter.py`:

\* Use Solana Web3 SDK

\* Differences: coin decimals, transaction signing, RPC endpoints

\* Keep method signatures identical to SUI adapter.

2. \*\*Factory in `adapter.py`:\*\*

```python

def get\_adapter(blockchain\_cfg):

if blockchain\_cfg.provider == "sui":

return SuiAdapter(blockchain\_cfg.sui)

elif blockchain\_cfg.provider == "solana":

return SolanaAdapter(blockchain\_cfg.solana)

else:

raise ValueError("Unsupported provider")

```

3. \*\*Config schema\*\* in `config.yaml`:

```yaml

blockchain:

provider: sui

sui:

rpc\_endpoint: …

solana:

rpc\_endpoint: …

commitment: "confirmed"

```

---

## 6. Validation & Testing

\* \*\*Unit tests\*\* in `tests/`:

\* Mock `IBlockchainAdapter` to simulate on-chain responses.

\* Verify that `MatchEngine` correctly calls adapter for each hook.

\* Validate that your `Transaction` records appear in SQLite with correct statuses.

\* \*\*End-to-end\*\*:

\* Run a dry-run match against a test SUI devnet.

\* Confirm balances before/after match for a test wallet.

\* Stress-test concurrent matches to ensure idempotency of fee collection.

---

## 7. Completeness Checklist

\* [ ] \*\*Entry fees\*\*: Hook created and tested in `MatchEngine.\_collect\_entry\_fees()`.

\* [ ] \*\*Character purchases\*\*: Payment adapter wired (if monetized).

\* [ ] \*\*Kill awards\*\*: On-chain disbursement in `MatchEngine.\_allocate\_kill\_award()`.

\* [ ] \*\*Final payout\*\*: On-chain disbursement in `MatchEngine.end\_match()`.

\* [ ] \*\*Transaction model & repo\*\*: `Transaction` table + CRUD in `repositories.py`.

\* [ ] \*\*Config driven\*\*: All endpoints and coin types in `config.yaml`.

\* [ ] \*\*Adapter pattern\*\*: `IBlockchainAdapter` interface + SUI/Solana implementations.

\* [ ] \*\*Tests\*\*: Mock adapter tests + integration test against devnet.

---

With this guide, a new developer can trace from \*\*config → engine → adapters → blockchain\*\* and safely add or modify economic flows—whether on SUI today or Solana tomorrow—without disturbing core battle-royale logic. Let me know if you’d like code snippets or deeper details on any section!