**5. Data Generation**

Automate your simulator to run through all contingency scenarios end‑to‑end and capture a structured dataset for each case. Below is a recommended workflow and data schema.

**5.1 Orchestrator Loop**

In your run\_sim.py (or equivalent), wrap the contingency analysis in a master loop:

python

CopyEdit

from datetime import datetime

from grid.advanced\_grid import Contingency

from simulation import build\_pp\_net, analyze\_contingency # your functions

# Load contingencies from JSON

contingencies = load\_contingencies("data/contingencies.json")

# Prepare results container

all\_records = []

# Loop over scenarios

for cont in contingencies:

ts\_start = datetime.utcnow()

records = analyze\_contingency(cont) # returns list of dicts per violation

ts\_end = datetime.utcnow()

# Attach metadata

for rec in records:

rec.update({

"scenario\_id": cont.id,

"scenario\_type": cont.type,

"start\_time": ts\_start.isoformat(),

"end\_time": ts\_end.isoformat(),

"duration\_s": (ts\_end - ts\_start).total\_seconds()

})

all\_records.extend(records)

# Export full dataset

import pandas as pd

df = pd.DataFrame(all\_records)

df.to\_csv("outputs/simulation\_dataset.csv", index=False)

**5.2 Captured Metrics**

For **each** contingency run and each violation/mitigation event, record:

| **Field** | **Description** |
| --- | --- |
| scenario\_id | Unique contingency ID (e.g. N1\_LINE\_L\_1\_2) |
| scenario\_type | "N-1" or "N-2" |
| element\_id | Outaged element(s) (line, gen, xfmr IDs) |
| violation\_type | "thermal\_overload", "voltage\_limit", "freq\_deviation", etc. |
| severity\_value | Numeric measure (e.g., % loading, p.u. voltage, Hz deviation) |
| mitigation\_type | "redispatch", "load\_shed", "topology\_change", etc. |
| mitigation\_target | ID of the resource acted upon (bus ID, gen ID, line ID) |
| mitigation\_amount | MW dispatched/shedded or tap step or reconfiguration parameter |
| post\_severity | Severity measure **after** mitigation |
| economic\_cost | Estimated cost of action (fuel cost + VoLL for load shed, $/MW) |
| recovery\_time\_s | Time to restore normal limits (simulated or assumed) |
| start\_time | Timestamp when this scenario began |
| end\_time | Timestamp when it completed |
| duration\_s | Processing time (performance metric) |

**5.3 Economic Impact & Recovery**

* **Redispatch Cost**

Costredispatch=∑gens(ΔPgen×incremental heat rate×fuel cost) \text{Cost}\_\text{redispatch} = \sum\_{\text{gens}} (\Delta P\_\text{gen} \times \text{incremental heat rate} \times \text{fuel cost})Costredispatch​=gens∑​(ΔPgen​×incremental heat rate×fuel cost)

* **Load Shed VoLL**

Costloadshed=MWshed×VoLL$/MW \text{Cost}\_\text{loadshed} = \text{MW}\_\text{shed} \times \text{VoLL}\_\$/\text{MW}Costloadshed​=MWshed​×VoLL$​/MW

* **Recovery Time**
  + If simulating dynamic response: measure time for voltages/flows to return within limits.
  + Otherwise, assign a deterministic estimate per action type (e.g. 5 min for generator ramp, 1 min for load shed).

**5.4 Data Storage & Labeling**

* **Directory Structure**

bash

CopyEdit

outputs/

├── simulation\_dataset.csv # full run

├── per\_scenario/ # optional split files

│ ├── N1\_LINE\_L\_1\_2.csv

│ └── N2\_GEN\_G5\_LINE\_L\_2\_3.csv

└── logs/ # detailed logs

└── run\_20250608T010000.log

* **File Naming**  
  Include date/time and scenario type in filenames for easy tracking.
* **Version Control**  
  Tag your topology and mitigation-rules code versions in a metadata column or separate JSON.

**5.5 Quality Assurance**

* **Schema Validation**  
  Use jsonschema or pandas schema checks to ensure all required fields are present and typed correctly.
* **Sanity Checks**
  + No negative MW values.
  + post\_severity ≤ severity\_value.
  + duration\_s within acceptable limits (e.g., < 10 s per scenario).

Once you’ve generated **simulation\_dataset.csv**, you can open it in Excel or feed it directly to ML pipelines. This dataset will comprise:

1. **Inputs**: contingency IDs and element attributes
2. **Outputs**: violation severities, suggested mitigations, costs, and recovery metrics

…providing a rich foundation for training models or comparing performance against your conventional SE + RTCA baseline.

**5.1 Orchestrator Loop (Enhanced)**

Your run\_sim.py orchestrator should handle:

1. **Dynamic Scenario Loading**
   * Auto‑discover new JSON files in data/contingencies/
   * Support filtering (e.g. only N‑1, only generator trips)
2. **Robust Timing & Logging**

python

CopyEdit

import logging, os

from datetime import datetime

from grid.advanced\_grid import Contingency

from simulation import build\_pp\_net, analyze\_contingency, load\_contingencies

# Setup logging

log\_dir = "outputs/logs"

os.makedirs(log\_dir, exist\_ok=True)

logfile = os.path.join(log\_dir, f"run\_{datetime.utcnow():%Y%m%dT%H%M%S}.log")

logging.basicConfig(filename=logfile, level=logging.INFO)

# Load and optionally filter

all\_conts = load\_contingencies("data/contingencies/")

contingencies = [c for c in all\_conts if c.type == "N-1"] # example filter

# Prepare storage

all\_records = []

1. **Main Loop with Error Handling**

python

CopyEdit

for cont in contingencies:

logging.info(f"Starting scenario {cont.id}")

ts\_start = datetime.utcnow()

try:

records = analyze\_contingency(cont)

except Exception as e:

logging.error(f"Error in scenario {cont.id}: {e}", exc\_info=True)

continue

ts\_end = datetime.utcnow()

# Attach metadata

for rec in records:

rec.update({

"scenario\_id": cont.id,

"scenario\_type": cont.type,

"start\_time": ts\_start.isoformat(),

"end\_time": ts\_end.isoformat(),

"duration\_s": (ts\_end - ts\_start).total\_seconds()

})

all\_records.extend(records)

logging.info(f"Finished scenario {cont.id} in {(ts\_end - ts\_start).total\_seconds()}s")

# Export

import pandas as pd

df = pd.DataFrame(all\_records)

out\_csv = "outputs/simulation\_dataset.csv"

df.to\_csv(out\_csv, index=False)

logging.info(f"Wrote {len(df)} records to {out\_csv}")

**5.2 Captured Metrics (Detailed)**

Ensure each record contains:

| **Field** | **Type** | **Notes** |
| --- | --- | --- |
| scenario\_id | string | e.g. "N1\_LINE\_L\_1\_2" |
| scenario\_type | string | "N-1" or "N-2" |
| element\_id | string | Comma‑delimited if multiple (e.g. "L\_3\_4, G5") |
| violation\_type | string | "thermal\_overload", "voltage\_limit", "freq\_deviation", … |
| severity\_value | float | % loading, p.u. voltage, Hz deviation |
| mitigation\_type | string | "redispatch", "load\_shed", "tap\_change", "facts\_adjust", … |
| action\_id | string | Unique mitigation plan ID |
| action\_sequence | JSON/text | Serialized list of steps (see Section A) |
| mitigation\_target | string | Bus, generator, line, or device ID |
| mitigation\_amount | float | MW, tap steps, MVar, etc. |
| post\_severity | float | Severity after mitigation |
| economic\_cost | float | USD cost (fuel + VoLL + maintenance) |
| cost\_breakdown | JSON/text | Per‑component costs |
| recovery\_time\_s | float | Actual or estimated seconds to full recovery |
| execution\_time\_s | float | Time to execute the mitigation plan |
| start\_time | timestamp | ISO format |
| end\_time | timestamp | ISO format |
| duration\_s | float | Secs to simulate the scenario (performance metric) |
| environmental\_impact | JSON/text | CO₂ kg, particulate emissions, etc. |

**5.3 Economic Impact & Recovery (Expanded)**

Implement utility functions:

python

CopyEdit

def compute\_redispatch\_cost(gen\_deltas, heat\_rates, fuel\_price):

# gen\_deltas: dict {gen\_id: ΔP\_mw}

# heat\_rates: dict {gen\_id: HR\_mbtu\_per\_mwh}

# fuel\_price: USD\_per\_mbtu

cost = 0.0

for gid, delta in gen\_deltas.items():

cost += delta \* heat\_rates[gid] \* fuel\_price

return cost

def compute\_load\_shed\_cost(mw\_shed, voLL):

return mw\_shed \* voLL

def estimate\_recovery\_time(actions):

# actions: list of dicts with 'type' and 'parameters'

time = 0.0

for a in actions:

if a["type"] == "redispatch":

time += abs(a["parameters"]["delta\_mw"]) / a["parameters"].get("ramp\_rate\_mw\_per\_min", 10) \* 60

elif a["type"] == "load\_shed":

time += 30 # assume 30 s

# … other rules …

return time

After you generate a mitigation plan:

python

CopyEdit

cost\_rd = compute\_redispatch\_cost(gen\_deltas, heat\_rates, fuel\_price)

cost\_ls = compute\_load\_shed\_cost(mw\_shed, voLL)

total\_cost = cost\_rd + cost\_ls + other\_costs

recovery\_time = estimate\_recovery\_time(plan["action\_sequence"])

**5.4 Data Storage & Labeling (Advanced)**

1. **Directory Layout**

bash

CopyEdit

outputs/

├── simulation\_dataset.csv

├── per\_scenario/

│ ├── N1\_LINE\_L\_1\_2\_20250608T010000.csv

│ └── N2\_GEN\_G5\_LINE\_L\_2\_3\_20250608T010000.csv

└── logs/

└── run\_20250608T010000.log

1. **Filename Convention**

CopyEdit

{scenario\_id}\_{run\_timestamp}.csv

1. **Metadata Versioning**
   * **Topology Hash**: SHA‑256 of your topology JSON file, stored in a topology\_hash column.
   * **Code Version**: Git commit SHA, stored in code\_version column.
2. **Split Files**  
   If df.shape[0] grows huge, consider chunking by scenario:

python

CopyEdit

for cont\_id, group in df.groupby("scenario\_id"):

fname = f"outputs/per\_scenario/{cont\_id}\_{now}.csv"

group.to\_csv(fname, index=False)

**5.5 Quality Assurance (Deep)**

1. **Schema Validation**

python

CopyEdit

from pandera import DataFrameSchema, Column, Check

schema = DataFrameSchema({

"scenario\_id": Column(str, Check.str\_length(min=1)),

"severity\_value": Column(float, Check.in\_range(0, 500)),

"post\_severity": Column(float, Check.le("severity\_value")),

"duration\_s": Column(float, Check.in\_range(0, 30)),

# …continue for all fields…

})

schema.validate(df)

1. **Sanity Checks**
   * **No negative power**:

python

CopyEdit

assert (df["mitigation\_amount"] >= 0).all()

* + **Violation cleared**:

python

CopyEdit

cleared = df["post\_severity"] <= df["severity\_value"]

assert cleared.all()

* + **Performance bounds**:

python

CopyEdit

avg\_time = df["duration\_s"].mean()

assert avg\_time < 10 # seconds

1. **Automated Test Scripts**
   * Create a small “golden” scenario set of 5 contingencies with known results.
   * On each code change, run these and compare to stored JSON golden outputs.

**Putting It All Together**

By elaborating each sub‑step:

* **5.1** ensures robust orchestration with logging, error handling, and filtering.
* **5.2** captures a full spectrum of metrics—mechanical, economic, environmental, and performance.
* **5.3** defines precise cost and recovery calculations for every action.
* **5.4** organizes data storage for traceability, versioning, and scalability.
* **5.5** enforces data integrity with schema checks, assertions, and regression tests.

With this in place, your simulator will produce **high‑fidelity**, **richly annotated** datasets—ready for deep analysis or ML model training.