**Phase 1: Random Forest Classifier (RF) as RTCA Recommendation Engine**

**1.1 Data Preparation**

* **Inputs (features):**
  + Tabular metrics per scenario:
    - severity\_value, line\_loading\_pct, voltage\_deviation, spare\_capacity\_mw
    - Engineered features: average loading in 2‑hop neighborhood, bus degree
  + One‑hot encode categorical fields: scenario\_type, violation\_type
* **Labels:**
  + mitigation\_type (e.g. load\_shed, redispatch, tap\_change)
  + Optionally a regression target: mitigation\_amount\_mw

**1.2 Training Pipeline**

python

CopyEdit

from sklearn.ensemble import RandomForestClassifier

from sklearn.pipeline import Pipeline

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import OneHotEncoder, StandardScaler

# 1. Preprocessor

cat\_cols = ["scenario\_type", "violation\_type"]

num\_cols = ["severity\_value", "line\_loading\_pct", "spare\_capacity\_mw"]

preproc = ColumnTransformer([

("cat", OneHotEncoder(handle\_unknown="ignore"), cat\_cols),

("num", StandardScaler(), num\_cols)

])

# 2. Pipeline

rf\_pipeline = Pipeline([

("prep", preproc),

("clf", RandomForestClassifier(n\_estimators=200, max\_depth=10, random\_state=42))

])

# 3. Train & Validate

rf\_pipeline.fit(X\_train, y\_train)

y\_pred = rf\_pipeline.predict(X\_val)

**1.3 Benchmarking & Deployment**

* **Metrics:** classification accuracy, F1‑score per action class
* **Feature Importance:** rf\_pipeline.named\_steps["clf"].feature\_importances\_ to identify key drivers
* **Real‑Time Serving:**
  + Wrap rf\_pipeline.predict() in a lightweight REST API (FastAPI or Flask)
  + Accept JSON payload of scenario features → return recommended mitigation\_type and amount\_mw

**Phase 2: Graph Neural Network (GNN) for Topology‑Aware Recommendations**

**2.1 Graph Feature Extraction**

* **Node features (x):** per‑bus metrics (voltage, load, degree, local loading)
* **Edge features (edge\_attr):** per‑line metrics (loading\_pct, impedance, capacity)
* **Graph connectivity (edge\_index):** adjacency list drawn from your topology

**2.2 Model Architecture**

python

CopyEdit

import torch

from torch\_geometric.nn import GCNConv, global\_mean\_pool

class RTCA\_GNN(torch.nn.Module):

def \_\_init\_\_(self, in\_dim, hidden\_dim, num\_actions):

super().\_\_init\_\_()

self.conv1 = GCNConv(in\_dim, hidden\_dim)

self.conv2 = GCNConv(hidden\_dim, hidden\_dim)

self.lin = torch.nn.Linear(hidden\_dim, num\_actions)

def forward(self, x, edge\_index, edge\_attr, batch):

x = torch.relu(self.conv1(x, edge\_index, edge\_attr))

x = torch.relu(self.conv2(x, edge\_index, edge\_attr))

g = global\_mean\_pool(x, batch) # graph‑level embedding

return self.lin(g) # logits over mitigation classes

**2.3 Training Loop**

python

CopyEdit

from torch\_geometric.loader import DataLoader

import torch.nn.functional as F

model = RTCA\_GNN(in\_dim=node\_feat\_dim, hidden\_dim=64, num\_actions=action\_count)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(epochs):

for batch in DataLoader(train\_dataset, batch\_size=32):

logits = model(batch.x, batch.edge\_index, batch.edge\_attr, batch.batch)

loss = F.cross\_entropy(logits, batch.y)

loss.backward()

optimizer.step()

optimizer.zero\_grad()

**2.4 Integration with RF**

* **Hybrid Feature Set:** compute a graph embedding once per scenario (g = model.embed(...)), append g to RF’s numeric features
* **End‑to‑End Replacement:** use the GNN’s final output logits as your mitigation classifier instead of RF

**Phase 3: Reinforcement Learning (RL) for Multi‑Step Sequence Optimization**

**3.1 Environment Design (GridSimEnv)**

* **State:**
  + Current PF solution: vectors of voltage, loading, reserves, previous actions
  + Encoded as a fixed‑size state vector or small graph plus global metrics
* **Action Space:**
  + Discrete: pick from a set of parameterized actions (e.g. redispatch\_G12, tap\_XFMR\_S1\_up, shed\_10mw\_bus\_15)
  + Continuous extension: parameter values (delta\_mw) via Box spaces
* **Reward Function:**

ini

CopyEdit

r = –(fuel\_cost + VoLL\_penalty + α \* total\_time) + β \* reliability\_bonus

* + Heavily penalize any remaining violation at episode end
  + Reward small, quick fixes to encourage minimal action plans

**3.2 Algorithm Choice**

* **PPO (Proximal Policy Optimization):** robust for both discrete and continuous controls
* **A2C:** simpler, synchronous alternative if sample overhead is a concern

**3.3 Training with Stable‑Baselines3**

python

CopyEdit

from stable\_baselines3 import PPO

env = GridSimEnv(grid=grid, contingencies=contingency\_list, max\_steps=5)

model = PPO("MlpPolicy", env, verbose=1, n\_steps=2048, batch\_size=64)

model.learn(total\_timesteps=300\_000)

model.save("ppo\_grid\_mitigation")

**3.4 Offline Validation**

* **Hold‑out Set:** reserve 10–20% of scenarios for evaluation
* **Metrics:**
  + % violations cleared vs. rule‑based baseline
  + Average total cost and time per scenario
* **A/B Comparison:** have RF/GNN policies and RL policy tackle the same hold‑out cases; compare costs & reliability

**Phase 4: Production Integration & Continuous Improvement**

1. **Deployment Stack:**
   * **RF & GNN** in a real‑time microservice (FastAPI + TorchServe for GNN)
   * **RL agent** runs offline in batch mode, proposing new strategies for expert review
2. **Monitoring & Feedback:**
   * Log actual grid responses and costs when deploying in a testbed or sandbox
   * Retrain RF/GNN periodically with new simulation data or real‑world event logs
3. **Iterative Refinement:**
   * Use RL‑discovered plans to expand your rule‑based engine and enrich training data
   * Gradually increase scenario complexity (N‑2 events, dynamic contingencies)

**Why This Path Works**

* **RF → GNN → RL** progression ensures you always have a **working**, **interpretable**, and **incrementally sophisticated** policy.
* You start with **low entry barriers** (RF), build **topology awareness** (GNN), and ultimately achieve **strategic planning** (RL).
* Each phase reuses code, data pipelines, and evaluation frameworks, minimizing duplication.

By following these elaborated steps, you’ll evolve from a quick RTCA classifier to a fully autonomous, cost‑optimized, multi‑step mitigation planner—ensuring continuous value and capability growth.