Implementation hints for further research

Canary deployment

Use **progressive traffic shifting** to a new version, promote only if metrics stay healthy.

- *GCP (Cloud Run):* deploy new revision with `--no-traffic`, then shift 10%, to 50%, to100% via `gcloud run services update-traffic app --to-latest=10 ...`. For automation and rollback-on-alerts, use *Cloud Deploy* canary strategy with verification jobs.
- *AWS (Lambda):* publish a version and set a *Weighted Alias* (e.g., `prod` 90/10 between old/new); front with API Gateway; advance weights via CI.
- *Kubernetes (Alt):* use *Argo Rollouts* with canary steps and Prometheus/CloudWatch metrics.
- *Tip (Python flag):* pass `MODEL_ARTIFACT` / `USE_NEW=true` to route a percentage of requests to the new model; promote only if median latency is under a certain level, error rate, and **gCO_2e/1k req** meet thresholds.

Example GCP:

```
# Deploy new revision without traffic
gcloud run deploy app --image gcr.io/$PROJECT/app:$SHA --no-traffic --region=europe-west3


# Start canary: 10% to latest, 90% to stable
STABLE=$(gcloud run services describe app --region=europe-west3 --
format='value(status.traffic[0].revisionName)')
gcloud run services update-traffic app --to-latest=10 --to-revisions=$STABLE=90 --region=europe-
west3


# Promote as metrics look good
gcloud run services update-traffic app --to-latest=50 --to-revisions=$STABLE=50 --region=europe-
west3
gcloud run services update-traffic app --to-latest --region=europe-west3   # 100% to latest
```

Example Python:

```
import os, random
NEW_PCT = float(os.getenv("CANARY_PCT", "0.10"))  # 10%
def route_to_new() -> bool:
    return random.random() < NEW_PCT
# if route_to_new(): use NEW model/artifact; else use STABLE
```

Redeployment (safe + fast rollback)

Automate builds and promote immutably versioned artifacts; keep rollback one command away.

- *CI/CD:* GitHub Actions/Cloud Build, build Docker, push, deploy.
- *GCP:* `gcloud run deploy app --image gcr.io/$PROJECT/app:$SHA --no-traffic` then update traffic; rollback = point traffic back to prior revision.
- *AWS:* `aws lambda update-function-code --function-name app --image-uri ...`; rollback = alias traffic back to previous version.

Example GCP:

```
# Redeploy immutable image (no-traffic), then shift traffic once healthy
gcloud run deploy app --image gcr.io/$PROJECT/app:$SHA --no-traffic --region=europe-west3
gcloud run services update-traffic app --to-latest=100 --region=europe-west3

# Rollback instantly to previous revision if needed
PREV=$(gcloud run services describe app --region=europe-west3 \
  --format='value(status.traffic[1].revisionName)')
gcloud run services update-traffic app --to-revisions=$PREV=100 --region=europe-west3
```

Example git actions:

```
# .github/workflows/deploy.yml
name: deploy
on: { push: { branches: [main] } }
jobs:
  run:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: gcloud auth configure-docker europe-west3-docker.pkg.dev -q
      - run: docker build -t gcr.io/$PROJECT/app:${{ github.sha }} .
      - run: docker push gcr.io/$PROJECT/app:${{ github.sha }}
      - run: gcloud run deploy app --image gcr.io/$PROJECT/app:${{ github.sha }} --region=europe-
west3 --no-traffic
      - run: gcloud run services update-traffic app --to-latest=100 --region=europe-west3
```

Serverless (batch + API, carbon-aware)

Prefer *scale-to-zero* for non-urgent/variable load; burst in green windows. *Serverless* means using cloud functions or containers that run only when triggered, with no need to manage underlying servers. They scale automatically to zero when idle and up when demand increases, so you pay and consume energy only during execution.

- *GCP:* Cloud Run (API) + Cloud Run Jobs (batch). Queue with Pub/Sub; worker pulls a batch, runs inference, writes results, exits.
- *AWS*: Lambda (workers) + SQS (queue) + API Gateway (front door); use Lambda Power Tuning to pick memory vs. runtime sweet spot.
- *Python sketch:* scheduler reads carbon intensity; if HIGH: enqueue batches to serverless workers; if LOW: route to high-throughput service or release deferred jobs.

Example:

```
# pub.py
from google.cloud import pubsub_v1, storage
import json, os
topic = os.environ["TOPIC"]
pub = pubsub_v1.PublisherClient()
topic_path = pub.topic_path(os.environ["PROJECT"], topic)

def enqueue(batch_uri:str, artifact:str):
    msg = {"batch_uri": batch_uri, "artifact": artifact}
    pub.publish(topic_path, json.dumps(msg).encode())
------
# worker.py
from codecarbon import EmissionsTracker
import json, os, pandas as pd

def handle(msg: dict):
    tracker = EmissionsTracker(project_name="inference")
    tracker.start()
    df = pd.read_parquet(msg["batch_uri"])
    preds = [0]*len(df)  # placeholder inference
    kg = tracker.stop()
    print(json.dumps({"metric":"emissions","kg_co2e":kg,"n":len(df)}))

if __name__ == "__main__":
    handle(json.loads(os.environ.get("BATCH_MSG","{}")))
```

```
# scheduler.py
import os, json, random
from pub import enqueue

def carbon_intensity() -> float:
    return 150 + 200*random.random()  # simulate

HIGH=300
def tick():
    ci = carbon_intensity()
    if ci >= HIGH:
        # ECO: small batches, quantized artifact
        enqueue("gs://bucket/batch.parquet", "model-int8.joblib")
    else:
        # THROUGHPUT: larger batches, full-precision artifact
        enqueue("gs://bucket/batch.parquet", "model-fp32.joblib")

if __name__ == "__main__":
    tick()
```

Enforcing SLO (measure, gate, act)

Define SLIs (e.g., **p95 latency ≤ 300 ms**, **error rate < 1%**, **gCO_2e/1k req ≤ target**) and wire automatic gates. P95 refers to 95 percentile, i.e. in the example above 95% of queries are under 300ms.

- *GCP:* export app metrics to Cloud Monitoring; create alerting policies; in Cloud Deploy, fail canary/promote only if alerts are clear (verification step).
- *AWS:* CloudWatch metrics/alarms; use CodeDeploy canary with automatic rollback on alarm.
- *At runtime:* implement admission control (shed load or switch to lightweight model when SLOs breach), circuit breakers/timeouts, and backoff.

Example:

```
# app.py
from fastapi import FastAPI, HTTPException
from prometheus_client import Summary, Counter, Gauge, make_asgi_app
```

```
import time, os

app = FastAPI()
REQUEST_TIME = Summary("req_latency_seconds","Latency")
ERRORS = Counter("errors_total","Errors")
P95_TARGET = float(os.getenv("P95_TARGET","0.300"))  # 300 ms
MODE = Gauge("mode","0=eco,1=hp")

lat_hist = []

def p95(vals):
    if not vals: return 0.0
    s = sorted(vals); k = int(0.95*len(s))-1
    return s[max(k,0)]

@app.get("/predict")
@REQUEST_TIME.time()
def predict():
    start=time.time()
    try:
        # do inference ...
        time.sleep(0.05)  # placeholder latency
        return {"ok": True}
    except Exception:
        ERRORS.inc(); raise
    finally:
        lat = time.time()-start
        lat_hist.append(lat)
        current_p95 = p95(lat_hist[-200:])  # rolling window
        if current_p95 > P95_TARGET:
            MODE.set(0)  # switch to ECO (lighter model / cached path)
        else:
            MODE.set(1)  # high-performance profile

# expose /metrics
app.mount("/metrics", make_asgi_app())
```