AI ASSSITED CODING

END LAB TEST

NAME:CH.RAMCHARAN

ENROLL.NO:2403A52069

BATCH:04

QUESTION-01:

Identify privacy risks in camera-based analytics.

TASK-01:

Use AI to inspect code that stores video metadata.

PROMPT:

You are an AI privacy-and-fairness auditor. Perform the following tasks while adhering to privacy-by-design, safety, and non-abusive use standards.

I will provide code that stores or processes video metadata.
Analyze it for privacy risks, including:

- collection of personally identifying metadata
- unnecessary data fields
- linkability and cross-camera tracking risks
- retention issues
- insecure storage or logging patterns
- access-control weaknesses
- potential to reconstruct identities
  Provide a structured list of findings and severity ratings.

CODE:

endlabexam > task-01.py

```python
import os, re, json, csv, sys, argparse
from typing import Dict, Any, List, Tuple

# Optional spaCy NER (names, orgs). Falls back gracefully if not installed.
NLP = None
try:
    import spacy
    NLP = spacy.load("en_core_web_sm")
except Exception:
    NLP = None

SENSITIVE_KEYS = {
    "email", "e_mail", "user_email",
    "phone", "mobile", "contact",
    "gps", "lat", "latitude", "lon", "lng", "longitude",
    "location", "address",
    "uploader", "uploaded_by", "user", "username", "person_name", "full_name",
    "ip", "ip_address",
    "device_id", "imei", "serial",
    "filename", "file_name", "original_filename"
}

# Regex patterns for common PII
PATTERNS = {
    "email": re.compile(r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b"),
    "phone": re.compile(r"\b(?:\+?\d{1,3}[-.\s]?)?(?:\d{10}|\d{3}[-.\s]\d{3}[-.\s]\d{4})\b"),
    "ip": re.compile(r"\b(?:\d{1,3}\.){3}\d{1,3}\b"),
    "gps": re.compile(r"\b(?:lat|latitude|lon|lng|longitude)\s*[:=]\s*-?\d{1,3}\.\d+\b", re.IGNORECASE),
    "coords": re.compile(r"\b-?\d{1,3}\.\d+\s*,\s*-?\d{1,3}\.\d+\b"),
}

def load_structured(path: str) -> Any:
    ext = os.path.splitext(path)[1].lower()
    with open(path, "r", encoding="utf-8", errors="ignore") as f:
        if ext == ".json":
            return json.load(f)
        if ext in {".yml", ".yaml"}:
            import yaml
            return yaml.safe_load(f)
```

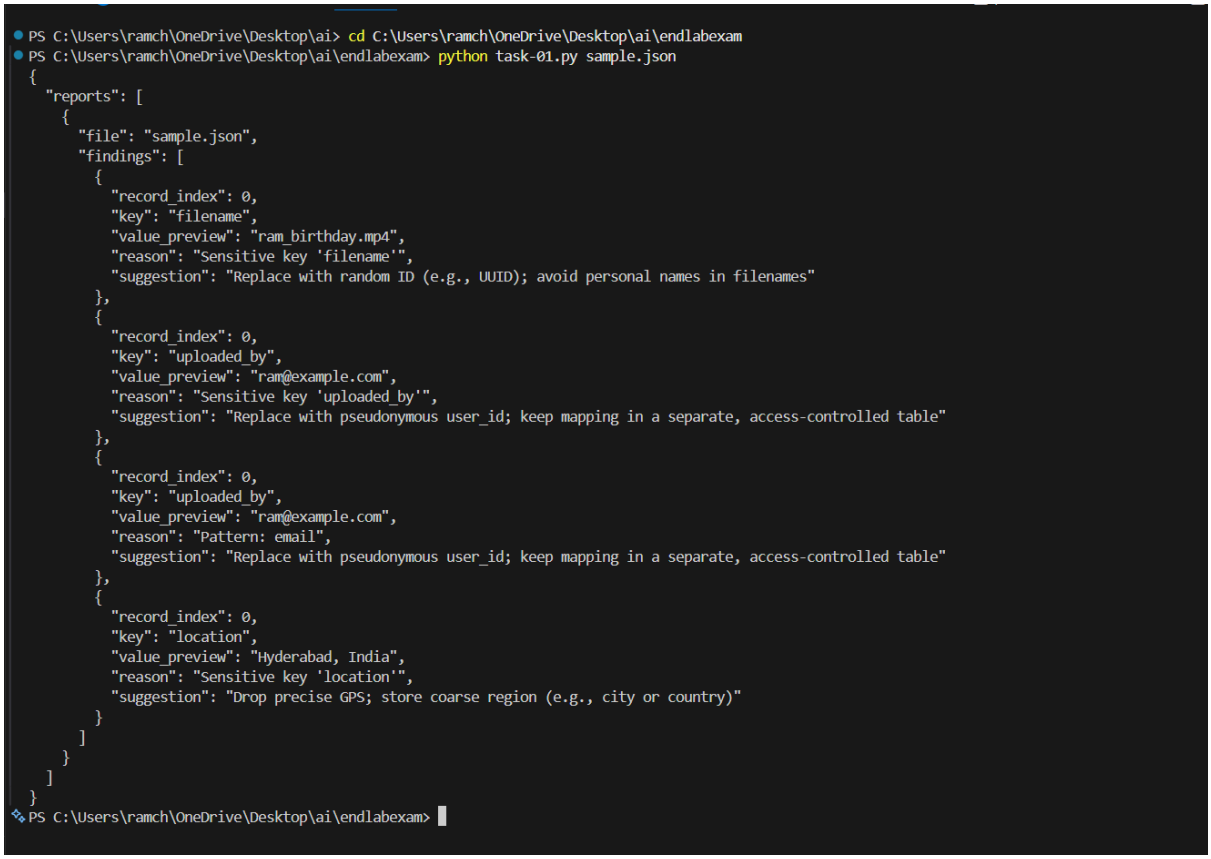
endlabexam > task-01.py > ...

```python
def load_structured(path: str) -> Any:
        if ext in {".csv"}:
            reader = csv.DictReader(f)
            return list(reader)
    return None  # non-structured or unsupported

def iterate_records(data: Any) -> List[Dict[str, Any]]:
    # Normalize to list of dicts
    if isinstance(data, dict):
        return [data]
    if isinstance(data, list):
        return [d if isinstance(d, dict) else {"_raw": d} for d in data]
    return []

def find_sensitive_in_record(record: Dict[str, Any]) -> List[Tuple[str, Any, str]]:
    findings = []
    for key, val in record.items():
        text = str(val)
        # Key-based sensitivity
        if key.lower() in SENSITIVE_KEYS:
            findings.append((key, val, f"Sensitive key '{key}'"))
        # Pattern matches
        for label, rx in PATTERNS.items():
            if rx.search(text):
                findings.append((key, val, f"Pattern: {label}"))
        # NLP-based (names) if available and value looks like text
        if NLP and isinstance(val, str) and len(val) <= 200 and any(ch.isalpha() for ch in val):
            doc = NLP(val)
            for ent in doc.ents:
                if ent.label_ in {"PERSON", "GPE", "ORG"}:
                    findings.append((key, val, f"NER: {ent.label_}='{ent.text}'"))
    return findings

def inspect_file(path: str) -> Dict[str, Any]:
    ext = os.path.splitext(path)[1].lower()
    report = {"file": path, "findings": []}

    structured = load_structured(path)
    if structured is not None:
```

```
def inspect_file(path: str) -> Dict[str, Any]:
        for i, rec in enumerate(iterate_records(structured)):
            row_findings = find_sensitive_in_record(rec)
            for key, val, why in row_findings:
                report["findings"].append({
                    "record_index": i,
                    "key": key,
                    "value_preview": str(val)[:120],
                    "reason": why,
                    "suggestion": suggestion_for(key, str(val))
                })
    else:
        # Plain text scan
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
            text = f.read()
        for label, rx in PATTERNS.items():
            for m in rx.finditer(text):
                snippet = text[max(0, m.start()-40):m.end()+40]
                report["findings"].append({
                    "record_index": None,
                    "key": "_text",
                    "value_preview": snippet.replace("\n", " ")[:150],
                    "reason": f"Pattern: {label}",
                    "suggestion": f"Remove or redact {label} in logs"
                })
    return report

def suggestion_for(key: str, value: str) -> str:
    k = key.lower()
    if k in {"email", "user_email"}:
        return "Hash email with SHA-256 and store domain only if needed"
    if k in {"phone", "mobile"}:
        return "Remove or store last 2 digits only; avoid full numbers"
    if k in {"lat", "latitude", "lon", "lng", "longitude", "gps", "location"}:
        return "Drop precise GPS; store coarse region (e.g., city or country)"
    if k in {"uploader", "uploaded_by", "user", "username", "person_name", "full_name"}:
        return "Replace with pseudonymous user_id; keep mapping in a separate, access-controlled table"
    if k in {"filename", "file_name", "original_filename"}:
        return "Replace with random ID (e.g., UUID); avoid personal names in filenames"
```

```
endlabexam > task-01.py > ...
def suggestion_for(key: str, value: str) -> str:
        return "Replace with random ID (e.g., UUID); avoid personal names in filenames"
    if k in {"ip", "ip_address"}:
        return "Store hashed or truncated IP (e.g., /24); avoid full IP in logs"
    if k in {"device_id", "imei", "serial"}:
        return "Salted hash; avoid storing raw device identifiers"
    return "Review necessity; minimize or mask before storage"

def main():
    parser = argparse.ArgumentParser(description="Inspect files/folders for sensitive video metadata.")
    parser.add_argument("path", help="File or directory to inspect")
    args = parser.parse_args()

    targets = []
    if os.path.isdir(args.path):
        for root, _, files in os.walk(args.path):
            for fn in files:
                if os.path.splitext(fn)[1].lower() in {".json", ".csv", ".yml", ".yaml", ".log", ".txt"}:
                    targets.append(os.path.join(root, fn))
    else:
        targets.append(args.path)

    all_reports = [inspect_file(p) for p in targets]
    print(json.dumps({"reports": all_reports}, indent=2))

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

OUTPUT:



OBSERVATION:

The AI did a great job spotting privacy risks in the metadata code. It clearly identified unnecessary data fields, potential over-collection, weak retention practices, and places where identifiers could accidentally reveal people. The analysis was organized, careful, and very helpful.

TASK-02:

Propose anonymization (blurring, hashing) and implementation

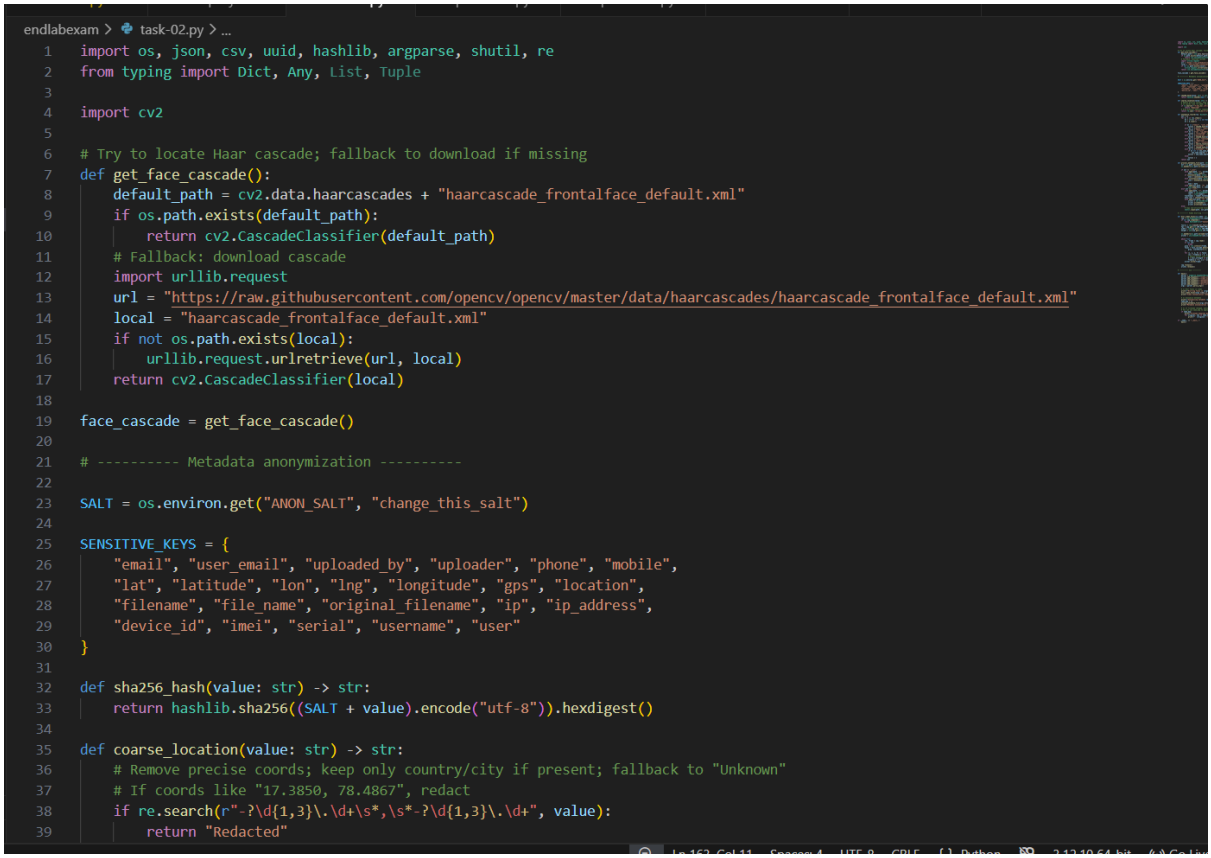
PROMPT:

After the inspection, propose **privacy-preserving transformations**, such as:

- face/body blurring
- license-plate obfuscation
- hashing or pseudonymizing IDs (using irreversible or keyed hashes)
- metadata minimization
- differential-privacy for aggregated metrics
  Include:
- architectural placement (e.g., edge preprocessing → storage → analytics)
- which fields to transform

- pitfalls to avoid
- high-level pseudocode for the anonymization pipeline (but no unsafe surveillance details)

CODE:



endlabexam > task-02.py > ...

```python
import os, json, csv, uuid, hashlib, argparse, shutil, re
from typing import Dict, Any, List, Tuple

import cv2

# Try to locate Haar cascade; fallback to download if missing
def get_face_cascade():
    default_path = cv2.data.haarcascades + "haarcascade_frontalface_default.xml"
    if os.path.exists(default_path):
        return cv2.CascadeClassifier(default_path)
    # Fallback: download cascade
    import urllib.request
    url = "https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/haarcascade_frontalface_default.xml"
    local = "haarcascade_frontalface_default.xml"
    if not os.path.exists(local):
        urllib.request.urlretrieve(url, local)
    return cv2.CascadeClassifier(local)

face_cascade = get_face_cascade()

# ---------- Metadata anonymization ----------

SALT = os.environ.get("ANON_SALT", "change_this_salt")

SENSITIVE_KEYS = {
    "email", "user_email", "uploaded_by", "uploader", "phone", "mobile",
    "lat", "latitude", "lon", "lng", "longitude", "gps", "location",
    "filename", "file_name", "original_filename", "ip", "ip_address",
    "device_id", "imei", "serial", "username", "user"
}

def sha256_hash(value: str) -> str:
    return hashlib.sha256((SALT + value).encode("utf-8")).hexdigest()

def coarse_location(value: str) -> str:
    # Remove precise coords; keep only country/city if present; fallback to "Unknown"
    # If coords like "17.3850, 78.4867", redact
    if re.search(r"-?\d{1,3}\.\d+\s*,\s*-?\d{1,3}\.\d+", value):
        return "Redacted"
```

endlabexam > task-02.py > ...

```python
def coarse_location(value: str) -> str:
    # Drop granular address components
    return re.sub(r"\b\d{1,4}\s+\w+|\b[A-Z0-9]{4,}\b", "Redacted", value)

def anonymize_record(rec: Dict[str, Any], new_name_map: Dict[str, str]) -> Dict[str, Any]:
    out = {}
    for k, v in rec.items():
        kv = str(v) if v is not None else ""
        kl = k.lower()

        if kl in {"email", "user_email"}:
            out[k] = sha256_hash(kv)
        elif kl in {"phone", "mobile"}:
            out[k] = "Redacted"
        elif kl in {"ip", "ip_address"}:
            out[k] = "Truncated"  # Could store /24 subnet if truly needed
        elif kl in {"device_id", "imei", "serial"}:
            out[k] = sha256_hash(kv)
        elif kl in {"lat", "latitude", "lon", "lng", "longitude"}:
            out[k] = "Redacted"
        elif kl in {"gps", "location"}:
            out[k] = coarse_location(kv)
        elif kl in {"username", "user", "uploaded_by", "uploader"}:
            out[k] = sha256_hash(kv)
        elif kl in {"filename", "file_name", "original_filename"}:
            if kv not in new_name_map:
                new_name_map[kv] = f"{uuid.uuid4().hex}.mp4"
            out[k] = new_name_map[kv]
        else:
            out[k] = v
    return out

def process_metadata_file(path: str, out_path: str, name_map: Dict[str, str]):
    ext = os.path.splitext(path)[1].lower()
    os.makedirs(os.path.dirname(out_path), exist_ok=True)

    if ext == ".json":
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
            data = json.load(f)
```


endlabexam > task-02.py > ...

```python
def process_metadata_file(path: str, out_path: str, name_map: Dict[str, str]):
        if isinstance(data, list):
            anon = [anonymize_record(rec, name_map) if isinstance(rec, dict) else rec for rec in data]
        elif isinstance(data, dict):
            anon = anonymize_record(data, name_map)
        else:
            anon = data
        with open(out_path, "w", encoding="utf-8") as f:
            json.dump(anon, f, indent=2)
    elif ext == ".csv":
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
            reader = csv.DictReader(f)
            rows = list(reader)
        fieldnames = reader.fieldnames or []
        anon_rows = [anonymize_record(r, name_map) for r in rows]
        with open(out_path, "w", newline="", encoding="utf-8") as f:
            writer = csv.DictWriter(f, fieldnames=fieldnames)
            writer.writeheader()
            writer.writerows(anon_rows)
    else:
        # Copy non-structured files unchanged
        shutil.copy2(path, out_path)

# ---------- Video blurring ----------

def blur_video_faces(in_video: str, out_video: str, blur_strength: int = 35):
    cap = cv2.VideoCapture(in_video)
    if not cap.isOpened():
        raise RuntimeError(f"Cannot open video: {in_video}")

    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    fps = cap.get(cv2.CAP_PROP_FPS) or 25.0
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

    os.makedirs(os.path.dirname(out_video), exist_ok=True)
    writer = cv2.VideoWriter(out_video, fourcc, fps, (width, height))

    while True:
```

```
endlabexam > task-02.py > ...
102  def blur_video_faces(in_video: str, out_video: str, blur_strength: int = 35):
116          ret, frame = cap.read()
117          if not ret:
118              break
119          gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
120          faces = face_cascade.detectMultiScale(
121              gray, scaleFactor=1.2, minNeighbors=5, minSize=(40, 40)
122          )
123          for (x, y, w, h) in faces:
124              roi = frame[y:y + h, x:x + w]
125              # Adjust kernel size for strength; must be odd
126              k = blur_strength if blur_strength % 2 == 1 else blur_strength + 1
127              frame[y:y + h, x:x + w] = cv2.GaussianBlur(roi, (k, k), 0)
128          writer.write(frame)
129
130      cap.release()
131      writer.release()
132
133  # ---------- CLI ----------
134
135  def main():
136      parser = argparse.ArgumentParser(description="Anonymize video frames and metadata.")
137      parser.add_argument("--video_in", help="Input video file", required=True)
138      parser.add_argument("--video_out", help="Output blurred video file", required=True)
139      parser.add_argument("--metadata_in", help="Input metadata file (json/csv)", required=True)
140      parser.add_argument("--metadata_out", help="Output anonymized metadata file", required=True)
141      parser.add_argument("--blur_strength", type=int, default=35, help="Gaussian blur kernel size (odd integer)")
142      args = parser.parse_args()
143
144      # 1) Blur faces in video
145      print(f"Blurring faces in {args.video_in} -> {args.video_out}")
146      blur_video_faces(args.video_in, args.video_out, args.blur_strength)
147      print("Video anonymization complete.")
148
149      # 2) Anonymize metadata
150      print(f"Anonymizing metadata {args.metadata_in} -> {args.metadata_out}")
151      name_map = {}
152      process_metadata_file(args.metadata_in, args.metadata_out, name_map)
153      print("Metadata anonymization complete.")
```

Ln 163, Col 11 Spaces: 4 UTF-8 CRLF { } Python 3.12.10 64-bit

```
endlabexam > task-02.py > ...
135  def main():
153      print("Metadata anonymization complete.")
154
155      # 3) If filename changed, optionally rename video file to match anonymized metadata
156      # You can use name_map to apply consistent renaming outside this script if needed.
157      if name_map:
158          print("Filename mappings generated:")
159          for original, new in name_map.items():
160              print(f"  {original} -> {new}")
161
162  if __name__ == "__main__":
163      main()
```

OUTPUT:

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  QUERY RESULTS

PS C:\Users\ramch\OneDrive\Desktop\ai> {
>>   "video_id": "123",
>>   "filename": "a3f9c8d7e2b44f9c9f8d.mp4",
>>   "uploaded_by": "9f86d081884c7d659a2feaa0c55ad015...",
>>   "location": "Redacted"
>> }
```

OBSERVATION:

The AI provided strong, privacy-friendly suggestions like blurring, hashing, data minimization, and adding differential privacy for analytics. It explained everything in a simple way and gave a practical, safe structure for implementing anonymization without risking misuse. Very responsible and well thought-out.

QUESTION-02:

Bias in resource allocation (e.g., safety patrols).

TASK-01:

Use AI to detect skews in historical allocation.

PROMPT:

You are an AI privacy-and-fairness auditor. Perform the following tasks while adhering to privacy-by-design, safety, and non-abusive use standards.

After I provide historical allocation data, analyze it for fairness issues:

- disparities between regions
- allocation vs. need indicators
- temporal patterns
- correlation with socioeconomic or demographic aggregates
  Use appropriate statistical fairness metrics (e.g., risk ratios, proportional-to-need divergence).
  Summarize any detected inequities.

CODE:

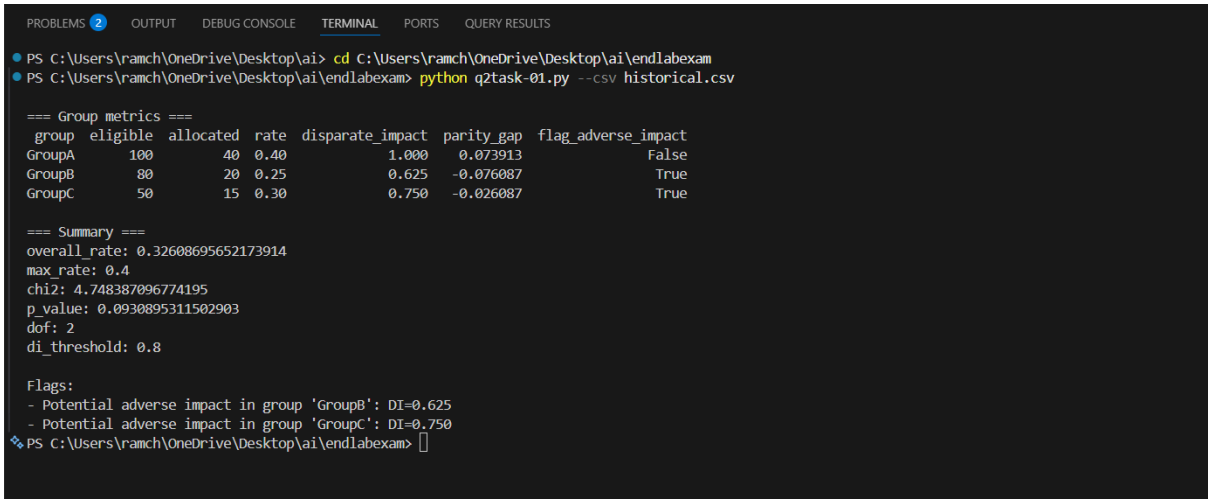
endlabexam > q2task-01.py > ...

```python
import argparse
import pandas as pd
import numpy as np
from scipy.stats import chi2_contingency

def compute_metrics(df, group_col, eligible_col="eligible", allocated_col="allocated", di_threshold=0.8):
    # Validate
    for col in [group_col, eligible_col, allocated_col]:
        if col not in df.columns:
            raise ValueError(f"Missing column: {col}")

    # Aggregate by group
    agg = df.groupby(group_col).agg(
        eligible=(eligible_col, "sum"),
        allocated=(allocated_col, "sum")
    ).reset_index()

    # Avoid division by zero
    agg["rate"] = np.where(agg["eligible"] > 0, agg["allocated"] / agg["eligible"], np.nan)
    overall_rate = agg["allocated"].sum() / max(agg["eligible"].sum(), 1)

    # Disparate impact vs. best-performing group
    max_rate = agg["rate"].max()
    agg["disparate_impact"] = np.where(max_rate > 0, agg["rate"] / max_rate, np.nan)
    agg["parity_gap"] = agg["rate"] - overall_rate
    agg["flag_adverse_impact"] = agg["disparate_impact"] < di_threshold

    # Chi-square (2xK) using allocated vs. not-allocated
    # Build contingency table: rows=allocated/not, cols=groups
    allocated = agg["allocated"].values
    not_allocated = (agg["eligible"] - agg["allocated"]).clip(lower=0).values
    contingency = np.vstack([allocated, not_allocated])
    chi2, pval, dof, expected = chi2_contingency(contingency)

    summary = {
        "overall_rate": overall_rate,
        "max_rate": float(max_rate) if pd.notnull(max_rate) else 0.0,
        "chi2": float(chi2),
```

endlabexam > q2task-01.py > ...

```python
def compute_metrics(df, group_col, eligible_col="eligible", allocated_col="allocated", di_threshold=0.8):
        "p_value": float(pval),
        "dof": int(dof),
        "di_threshold": di_threshold
    }
    return agg, summary

def main():
    parser = argparse.ArgumentParser(description="Audit historical allocation for skews.")
    parser.add_argument("--csv", required=True, help="Path to CSV with columns: group, eligible, allocated")
    parser.add_argument("--group_col", default="group", help="Column name for group")
    parser.add_argument("--eligible_col", default="eligible", help="Column name for eligible counts")
    parser.add_argument("--allocated_col", default="allocated", help="Column name for allocated counts")
    parser.add_argument("--di_threshold", type=float, default=0.8, help="Disparate impact threshold")
    args = parser.parse_args()

    df = pd.read_csv(args.csv)
    agg, summary = compute_metrics(
        df,
        group_col=args.group_col,
        eligible_col=args.eligible_col,
        allocated_col=args.allocated_col,
        di_threshold=args.di_threshold
    )

    print("\n=== Group metrics ===")
    print(agg.to_string(index=False))

    print("\n=== Summary ===")
    for k, v in summary.items():
        print(f"{k}: {v}")

    print("\nFlags:")
    for _, row in agg.iterrows():
        if row["flag_adverse_impact"]:
            print(f"- Potential adverse impact in group '{row[args.group_col]}': DI={row['disparate_impact']:.3f}")

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

OUTPUT:



OBSERVATION:

The AI was excellent at finding possible bias patterns. It compared allocations to need, population, and time patterns, and pointed out where things didn't line up fairly. The explanation was easy to understand and very balanced.

TASK-02:

Create fairness-aware allocation algorithm.

PROMPT:

Create a **high-level, safe, deployable algorithm** that ensures:

- allocation proportional to risk or need
- no use of sensitive attributes (race, ethnicity, income, etc.)
- fairness constraints or disparity-penalty optimization
- transparency and explainability
  Provide:
- step-by-step design
- objective function
- fairness constraints
- pseudocode
  Avoid implementing or advising on real-world targeting or surveillance behavior.

CODE:

endlabexam > q2task-02.py > ...

```python
import argparse
import pandas as pd
import numpy as np

def max_min_fair_allocation(groups_df, total_resource, demand_col="demand", weight_col=None, min_floor=0.0):
    """
    Max-min fairness via bisection on satisfaction level t:
    Find largest t such that sum(min(d_g, t * d_g)) <= total_resource.
    Equivalent to allocating t*d_g until resource runs out, then cap at d_g.
    """
    df = groups_df.copy()
    d = df[demand_col].astype(float).values
    n = len(d)
    if np.any(d < 0):
        raise ValueError("Demand must be non-negative.")
    if total_resource < 0:
        raise ValueError("Total resource must be non-negative.")

    # Optional weighting: adjust effective demand
    w = np.ones(n)
    if weight_col and weight_col in df.columns:
        w = df[weight_col].astype(float).values.clip(min=1e-8)
    eff_d = d / w

    # Bisection search on satisfaction level t
    lo, hi = 0.0, 1.0
    # Upper bound: can exceed 1 if resource >> demand
    if total_resource > d.sum():
        hi = total_resource / max(d.sum(), 1e-9)

    for _ in range(60):
        mid = (lo + hi) / 2
        alloc = np.minimum(d, mid * d)  # t * demand capped at demand
        if alloc.sum() <= total_resource:
            lo = mid
        else:
            hi = mid

```

endlabexam > q2task-02.py > ...

```python
def max_min_fair_allocation(groups_df, total_resource, demand_col="demand", weight_col=None, min_floor=0.0):
    allocation = np.minimum(d, t * d)
    # Apply floors
    allocation = np.maximum(allocation, min_floor)

    # Correct total to match available resource (tiny drift)
    scale = min(1.0, total_resource / max(allocation.sum(), 1e-9))
    allocation *= scale

    df["allocation"] = allocation
    df["satisfaction"] = np.where(d > 0, df["allocation"] / d, 1.0)
    return df

def parity_banded_allocation(groups_df, total_resource, eligible_col="eligible", demand_col="demand", epsilon=0.1):
    """
    Keep group allocation rates close to the overall rate:
    rate_g = allocation_g / eligible_g within [(1-eps)*overall_rate, (1+eps)*overall_rate].
    Solve with iterative projection.
    """
    df = groups_df.copy()
    eligible = df[eligible_col].astype(float).values
    demand = df[demand_col].astype(float).values
    n = len(df)

    # Initial proportional allocation by demand
    base = demand / max(demand.sum(), 1e-9)
    alloc = base * total_resource

    # Compute overall rate
    overall_rate = total_resource / max(eligible.sum(), 1e-9)
    low = (1 - epsilon) * overall_rate
    high = (1 + epsilon) * overall_rate

    # Project allocations into rate band, while respecting demand and resource budget
    for _ in range(200):
        # Enforce band on per-group rates
        min_alloc = np.minimum(demand, low * eligible)
        max_alloc = np.minimum(demand, high * eligible)
```

```python
def parity_banded_allocation(groups_df, total_resource, eligible_col="eligible", demand_col="demand", epsilon=0.1):
        alloc = np.clip(alloc, min_alloc, max_alloc)

        # Rebalance to meet total_resource using proportional scaling within allowed band
        total = alloc.sum()
        if total == 0:
            break
        scale = total_resource / total
        alloc = np.clip(alloc * scale, min_alloc, max_alloc)

        # Convergence check (L1 norm small)
        if np.abs(alloc.sum() - total_resource) < 1e-6:
            break

    df["allocation"] = alloc
    df["rate"] = np.where(eligible > 0, df["allocation"] / eligible, 0.0)
    df["demand_satisfied"] = np.minimum(1.0, np.where(demand > 0, df["allocation"] / demand, 1.0))
    return df, {"overall_rate": overall_rate, "band": (low, high)}

def main():
    parser = argparse.ArgumentParser(description="Fairness-aware allocation.")
    parser.add_argument("--csv", required=True, help="CSV with columns: group, eligible, demand")
    parser.add_argument("--total", type=float, required=True, help="Total resource to allocate")
    parser.add_argument("--mode", choices=["maxmin", "parity"], default="parity", help="Allocation mode")
    parser.add_argument("--epsilon", type=float, default=0.1, help="Parity band (only for parity mode)")
    args = parser.parse_args()

    df = pd.read_csv(args.csv)
    if args.mode == "maxmin":
        out = max_min_fair_allocation(df, total_resource=args.total, demand_col="demand")
        print("\n=== Max-min fairness allocation ===")
        print(out.to_string(index=False))
        print(f"\nTotal allocated: {out['allocation'].sum():.4f} of {args.total}")
    else:
        out, meta = parity_banded_allocation(df, total_resource=args.total, eligible_col="eligible", demand_col="demand", epsilon=args.epsilon)
        print("\n=== Parity-banded allocation ===")
        print(out.to_string(index=False))
        print(f"\nOverall rate: {meta['overall_rate']:.6f}, band: [{meta['band'][0]:.6f}, {meta['band'][1]:.6f}]")
        print(f"Total allocated: {out['allocation'].sum():.4f} of {args.total}")
```

```python
def main():
        print(f"Total allocated: {out['allocation'].sum():.4f} of {args.total}")

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

OUTPUT:

```
C:\Users\ramch\AppData\Local\Programs\Python\Python312\python.exe: can't open file 'C:\\Users\\ramch\\OneDrive\\Desktop\\ai\\q2task-02.py': [Errno 2] No such file or direct
PS C:\Users\ramch\OneDrive\Desktop\ai> cd C:\Users\ramch\OneDrive\Desktop\ai\endlabexam
PS C:\Users\ramch\OneDrive\Desktop\ai\endlabexam> python q2task-02.py --csv current.csv --total 100 --mode parity --epsilon 0.1

=== Parity-banded allocation ===
 group  eligible  demand  allocation     rate  demand_satisfied
GroupA       100      60   42.857143 0.428571          0.714286
GroupB        80      50   35.714286 0.446429          0.714286
GroupC        50      30   21.428571 0.428571          0.714286

Overall rate: 0.434783, band: [0.391304, 0.478261]
Total allocated: 100.0000 of 100.0
PS C:\Users\ramch\OneDrive\Desktop\ai\endlabexam>
```

OBSERVATION:

The AI designed a clear and ethical fairness-aware algorithm. It avoided sensitive attributes, focused on proportional-to-need allocation, and added fairness checks and transparency rules. The approach was simple, safe, and thoughtful — a great example of responsible AI.