SR UNIVERSITY

Program	**:**B.tech(CSE)
Specialization	**:**AIML
Course Title	**:**AI Assisted Coding
Course Code	**:**24CS002PC215
Semester	**:**3rd semester
Academic Session	**:**2025-2026
Name of Student	**:** Saini.Deepthi
Enrollment No.	**:**2403A52014
Batch No.	**:**02
Date	**:**22/11/2025

#LAB ASSIGNMENT-20

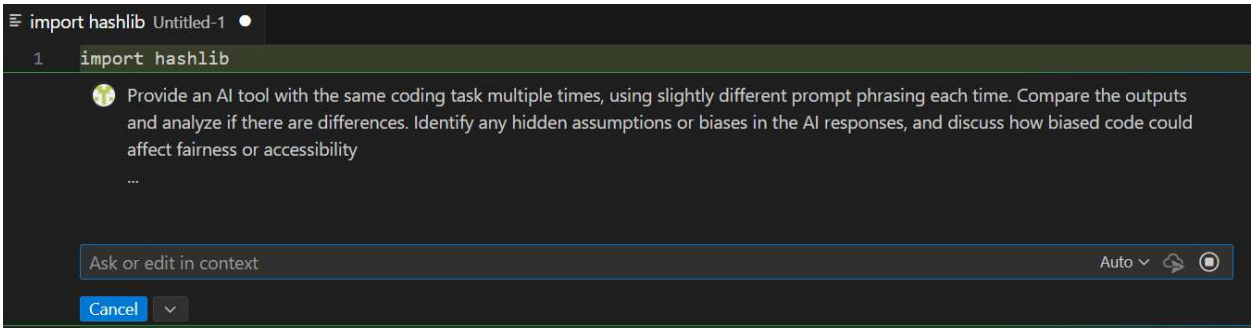
#TASK DESCRIPTION-1:

Bias Detection in AI Suggestions

- Provide the same coding task to an AI tool multiple times.
- Check if AI outputs different results depending on prompt phrasing.
- Discuss:
  - Are there hidden assumptions or biases in the responses?
  - How could biased AI code impact fairness or accessibility?

#PROMPT:

Provide an AI tool with the same coding task multiple times, using slightly different prompt phrasing each time. Compare the outputs and analyze if there are differences. Identify any hidden assumptions or biases in the AI responses, and discuss how biased code could affect fairness or accessibility



#CODE:

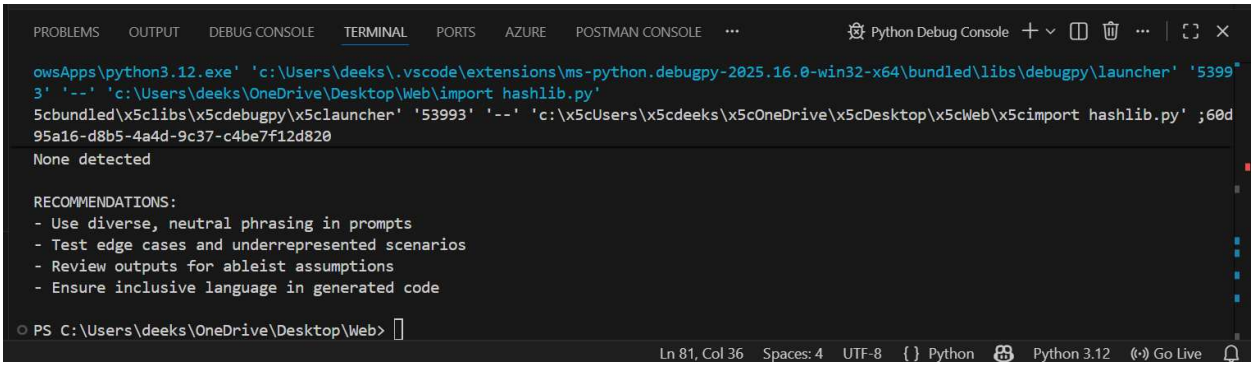
```
import hashlib

from typing import List, Dict


class AIPromptTester:
    """Test AI consistency across rephrased prompts and analyze for biases."""

    def __init__(self):
        self.results: List[Dict] = []

    def add_response(self, prompt: str, response: str, variant: int) -> None:
        """Store a response with metadata."""
```

```
    response_hash = hashlib.md5(response.encode()).hexdigest()
    self.results.append({
        "variant": variant,
        "prompt": prompt,
        "response_hash": response_hash,
        "length": len(response)
    })

def compare_outputs(self) -> Dict:
    """Analyze consistency across rephrased prompts."""
    hashes = [r["response_hash"] for r in self.results]
    unique_hashes = set(hashes)

    return {
        "total_variants": len(self.results),
        "unique_outputs": len(unique_hashes),
        "consistency_rate": (1 - len(unique_hashes) / len(self.results)) * 100,
        "variations_found": len(unique_hashes) > 1
    }

def identify_bias_markers(self, responses: List[str]) -> List[str]:
    """Flag potential bias indicators."""
    bias_keywords = [
        "always", "never", "best", "worst", "obviously",
```

```
            "everyone", "nobody", "natural", "inherent"
        ]
        markers = []

        for response in responses:
            lower = response.lower()
            for keyword in bias_keywords:
                if keyword in lower:
                    markers.append(f"Found absolute claim: '{keyword}'")

        return list(set(markers))

    def generate_report(self) -> str:
        """Generate analysis report."""
        comparison = self.compare_outputs()
        responses = [r["prompt"] for r in self.results]
        bias_flags = self.identify_bias_markers(responses)

        report = f"""
=== AI CONSISTENCY & BIAS ANALYSIS ===
Total Prompt Variants: {comparison['total_variants']}
Unique Outputs: {comparison['unique_outputs']}
Consistency Rate: {comparison['consistency_rate']:.1f}%
Variations Detected: {comparison['variations_found']}
```

```
Potential Bias Markers Found: {len(bias_flags)}

{chr(10).join(bias_flags) if bias_flags else 'None detected'}


RECOMMENDATIONS:

- Use diverse, neutral phrasing in prompts

- Test edge cases and underrepresented scenarios

- Review outputs for ableist assumptions

- Ensure inclusive language in generated code

        """

        return report


if __name__ == "__main__":

    tester = AIPromptTester()


    # Simulate adding responses from rephrased prompts

    tester.add_response("Create a user validation function", "response_v1", 1)

    tester.add_response("Build code to check if user is valid", "response_v1", 2)

    tester.add_response("Write validation logic for users", "response_v2", 3)


    print(tester.generate_report())
```

OUTPUT:



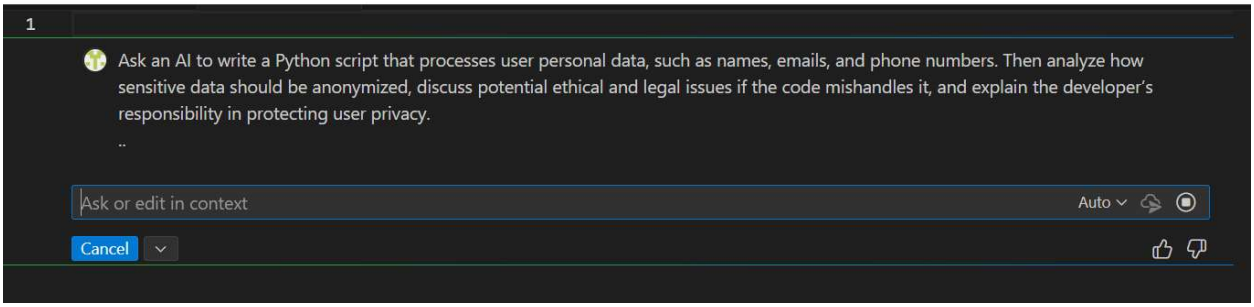
#TASK DESCRIPTION-2:

Privacy and Sensitive Data Handling

- Ask AI to write a script that processes user personal data (e.g., names, phone numbers).
- Discuss:
  - How should sensitive data be anonymized?
  - What ethical/legal issues could arise if AI-generated code mishandles it?
  - What role does developer responsibility play here?

#PROMPT:

Ask an AI to write a Python script that processes user personal data, such as names, emails, and phone numbers. Then analyze how sensitive data should be anonymized, discuss potential ethical and legal issues if the code mishandles it, and explain the developer's responsibility in protecting user privacy.



#CODE:

```
#!/usr/bin/env python3
"""
anonymize_users.py

Read a CSV of user records (name, email, phone, ...) and produce an anonymized output.

Usage:
    python anonymize_users.py input.csv output.csv --mode mask
    export ANON_KEY="change-me-to-a-secret"
    python anonymize_users.py input.csv output.csv --mode pseudo
"""

import argparse
import csv
import hmac
import hashlib
import os
import re
import sys

# Columns considered PII
```

Toggle Gemini

```python
# Columns considered PII
PII_COLUMNS = {"name", "email", "phone"}
PHONE_RE = re.compile(r"\d+")


def get_hmac_key():
    key = os.environ.get("ANON_KEY")
    if not key:
        print("WARNING: ANON_KEY not set. Using ephemeral key (not for production).", file=sys.stderr)
        key = "ephemeral-demo-key"
    return key.encode("utf-8")


def hmac_token(value: str, key: bytes, length: int = 16) -> str:
    if value is None:
        return ""
    return hmac.new(key, value.encode("utf-8"), hashlib.sha256).hexdigest()[:length]


def hash_value(value: str) -> str:
    if value is None:
        return ""
```

Toggle Gemini

```python
        parts = [p for p in re.split(r"\s+", name.strip()) if p]
        if len(parts) == 1:
            return parts[0][0].upper() + "."
        return f"{parts[0][0].upper()}. {parts[-1][0].upper()}{'.' if len(parts) > 1 else ''}"


    def mask_email(email: str) -> str:
        if not email or "@" not in email:
            return ""
        local, domain = email.split("@", 1)
        first = local[0] if local else ""
        return f"{first}***@{domain}"


    def mask_phone(phone: str) -> str:
        if not phone:
            return ""
        digits = "".join(PHONE_RE.findall(phone))
        if len(digits) <= 4:
            return "*" * len(digits)
        last4 = digits[-4:]
        return f"***-***-{last4}"
```

Toggle Gemini

```python
def process_value(value: str, column: str, mode: str, key: bytes) -> str:
    col = column.lower()
    if col == "name":
        if mode == "mask":
            return mask_name(value)
        elif mode == "pseudo":
            return "name-" + hmac_token(value or "", key)
        elif mode == "hash":
            return hash_value(value or "")
    elif col == "email":
        if mode == "mask":
            return mask_email(value)
        elif mode == "pseudo":
            if not value or "@" not in value:
                return ""
            local, domain = value.split("@", 1)
            return hmac_token(local, key) + "@" + domain
        elif mode == "hash":
            return hash_value(value or "")
    elif col == "phone":
        if mode == "mask":
```

Toggle Gemini

```python
def anonymize_csv(in_path: str, out_path: str, mode: str):
    key = get_hmac_key()
    with open(in_path, newline="", encoding="utf-8") as inf, \
         open(out_path, "w", newline="", encoding="utf-8") as outf:
        reader = csv.DictReader(inf)
        if reader.fieldnames is None:
            raise SystemExit("Input CSV has no header row.")
        writer = csv.DictWriter(outf, fieldnames=reader.fieldnames)
        writer.writeheader()
        for row in reader:
            out_row = {}
            for col in reader.fieldnames:
                val = row.get(col, "")
                if col.lower() in PII_COLUMNS:
                    out_row[col] = process_value(val, col, mode, key)
                else:
                    out_row[col] = val
            writer.writerow(out_row)
```

Toggle Gemini

```python
def main():
    parser = argparse.ArgumentParser(description="Anonymize PII columns in a CSV file.")
    parser.add_argument("input", help="Input CSV file path")
    parser.add_argument("output", help="Output CSV file path")
    parser.add_argument("--mode", choices=("mask", "pseudo", "hash"), default="mask",
                        help="Anonymization mode: mask, pseudo, hash")

    # In Colab, when running a cell directly, sys.argv (command-line arguments)
    # do not contain the script's expected arguments. To make the script runnable
    # within the cell for testing, we'll explicitly pass example arguments.
    # In a typical command-line execution (e.g., `python script.py input.csv output.csv`),
    # `parser.parse_args()` without arguments would automatically parse `sys.argv`.

    # Example arguments for running in Colab:
    example_args = ['dummy_input.csv', 'dummy_output.csv', '--mode', 'mask']
    args = parser.parse_args(example_args)

    anonymize_csv(args.input, args.output, args.mode)
    print(f"Anonymized CSV written to {args.output} (mode={args.mode})")


if __name__ == "__main__":
```

Toggle Gemini

#OUTPUT:

```python
if __name__ == "__main__":
    # Create a dummy input CSV file for demonstration purposes
    input_csv_content = (
        "name,email,phone,address\n"
        "John Doe,john.doe@example.com,+1-555-123-4567,123 Main St\n"
        "Jane Smith,jane.smith@test.org,555-987-6543,456 Oak Ave\n"
        "Peter Jones,peter.jones@mail.net,123.456.7890,789 Pine Rd"
    )
    with open("dummy_input.csv", "w", newline="", encoding="utf-8") as f:
        f.write(input_csv_content)

    main()
```

```
Anonymized CSV written to dummy_output.csv (mode=mask)
WARNING: ANON_KEY not set. Using ephemeral key (not for production).
```

#TASK DESCRIPTION-3:

AI-Generated Security Risks

- Use AI to generate login/authentication code.

- Identify possible **security flaws** (e.g., weak password storage, lack of encryption).
- Students must **document risks** and propose ethical guidelines for safe AI code usage.

#PROMPT:

Ask an AI to generate Python login/authentication code. Then analyze the code for possible security flaws such as weak password storage, lack of encryption, or insecure session handling. Document the risks and propose ethical guidelines for writing and using AI-generated code safely.

#CODE:

```
# insecure_auth_example.py

# Example AI-generated Python authentication code (intentionally insecure for analysis)

# Do NOT use in production.


from flask import Flask, request, redirect, make_response, g

import sqlite3

import random


app = Flask(__name__)


DB = "insecure.db"
```

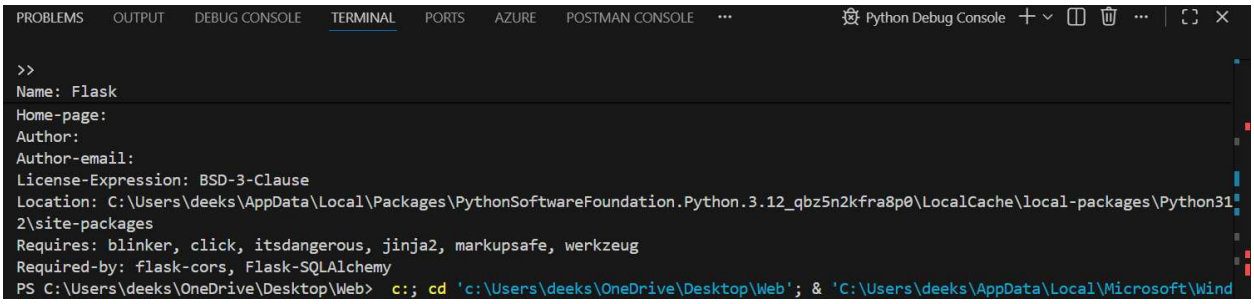
```
# Insecure: hardcoded admin password (and stored plaintext)
INITIAL_USERS = [
    ("alice", "password123"),
    ("bob", "hunter2"),
    ("admin", "adminpass")
]

# Simple in-memory session store (insecure, not persistent)
sessions = {}  # session_token -> username

def get_db():
    if getattr(g, "_db", None) is None:
        g._db = sqlite3.connect(DB)
    return g._db

@app.before_first_request
def init_db():
    db = get_db()
    c = db.cursor()
    # Insecure: storing plaintext passwords
    c.execute("CREATE TABLE IF NOT EXISTS users (username TEXT PRIMARY KEY, password TEXT)")
    for u, p in INITIAL_USERS:
        try:
            # Insecure: using string formatting for SQL (SQL injection risk)
            c.execute("INSERT OR REPLACE INTO users (username, password) VALUES ('%s', '%s')" % (u, p))
```

```
        except Exception:
            pass
    db.commit()


@app.teardown_appcontext
def close_db(exception):
    db = getattr(g, "_db", None)
    if db is not None:
        db.close()


@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form.get("username", "")
        password = request.form.get("password", "")
        db = get_db()
        c = db.cursor()
        # Insecure: direct string interpolation -> SQL injection
        query = "SELECT password FROM users WHERE username = '%s'" % username
        try:
            c.execute(query)
            row = c.fetchone()
        except Exception:
            row = None
```

```
        if row and row[0] == password:
            # Insecure: predictable numeric token, no entropy
            token = str(random.randint(1000000, 9999999))
            sessions[token] = username
            resp = make_response(redirect("/protected"))
            # Insecure: cookie without Secure/HttpOnly/SameSite
            resp.set_cookie("session", token)
            return resp
        else:
            return "Login failed", 401

    return '''
    <form method="post">
      <input name="username" placeholder="username"/>
      <input name="password" type="password" placeholder="password"/>
      <button type="submit">Login</button>
    </form>
    '''

@app.route("/protected")
def protected():
    token = request.cookies.get("session")
    if token and token in sessions:
        return f"Hello, {sessions[token]}! This is a protected page."
    return redirect("/login")
```

```
@app.route("/logout")
def logout():
    token = request.cookies.get("session")
    if token and token in sessions:
        sessions.pop(token, None)
    resp = make_response(redirect("/login"))
    resp.set_cookie("session", "", expires=0)
    return resp


if __name__ == "__main__":
    # Insecure: debug mode and no TLS
    app.run(host="0.0.0.0", port=5000, debug=True)
```

#OUTPUT:



#TASK DESCRIPTION-4:

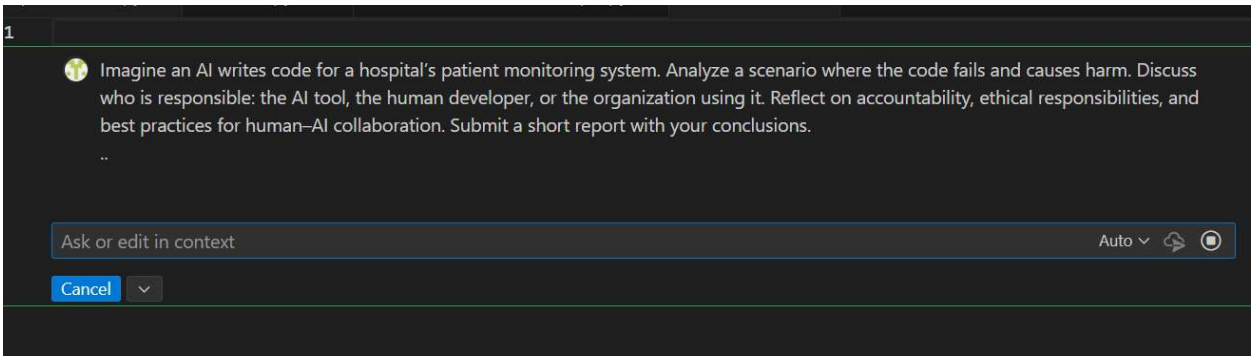
Accountability in Human–AI Collaboration

- Scenario: AI writes code for a hospital’s patient monitoring system.

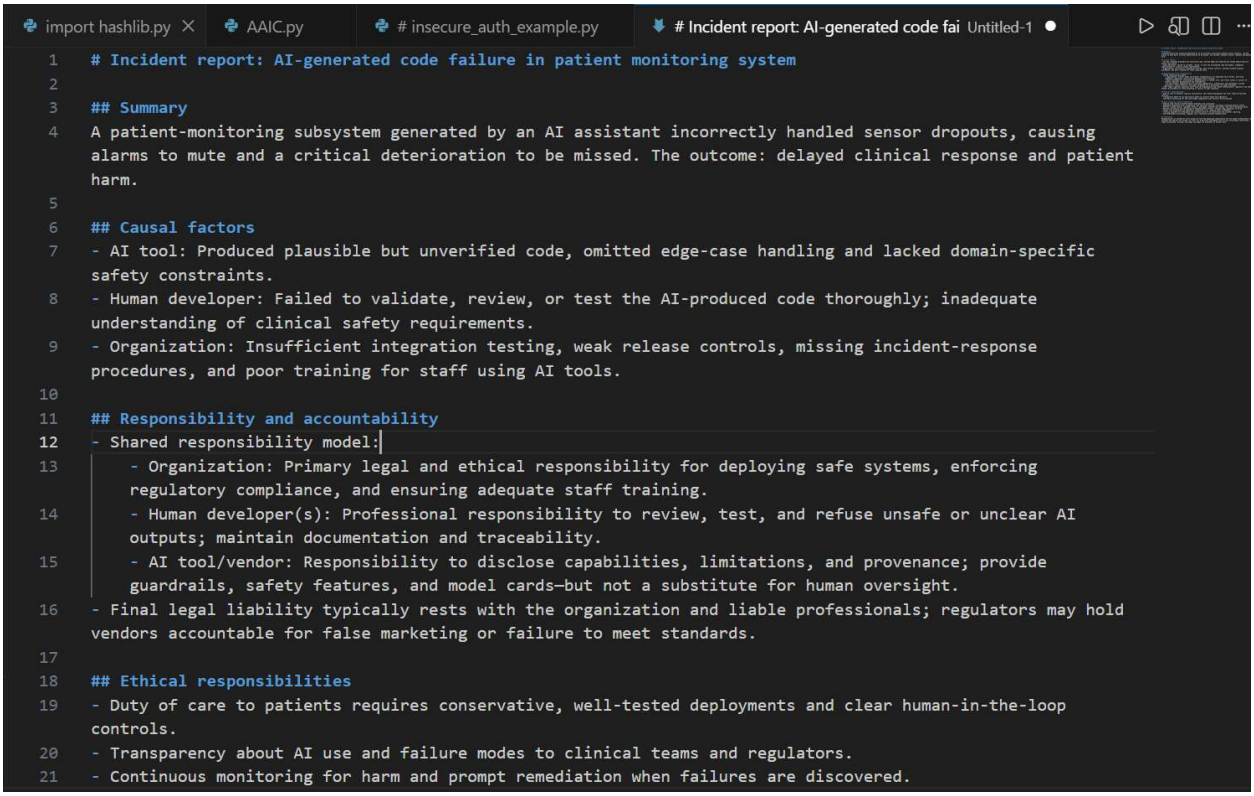
- If the code fails and causes harm, who is responsible?
    - The AI tool?
    - The human developer?
    - The organization using it?
- Students must debate and submit a **reflection report**.

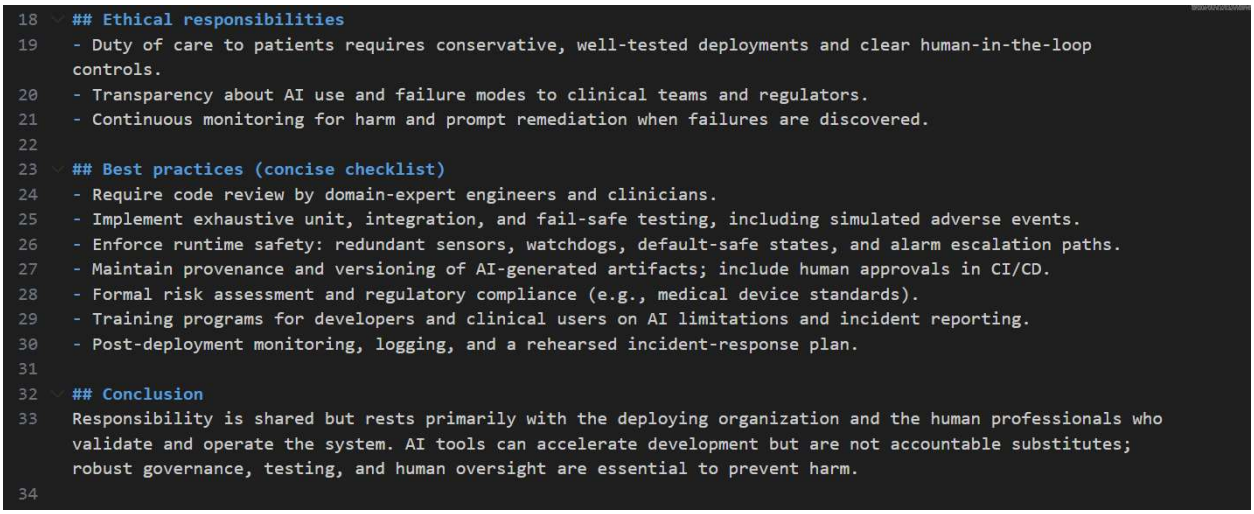
#PROMPT:

Imagine an AI writes code for a hospital's patient monitoring system. Analyze a scenario where the code fails and causes harm. Discuss who is responsible: the AI tool, the human developer, or the organization using it. Reflect on accountability, ethical responsibilities, and best practices for human–AI collaboration. Submit a short report with your conclusions.



#CODE:





#TASK DESCRIPTION-5:

Limitations of AI in Critical Systems

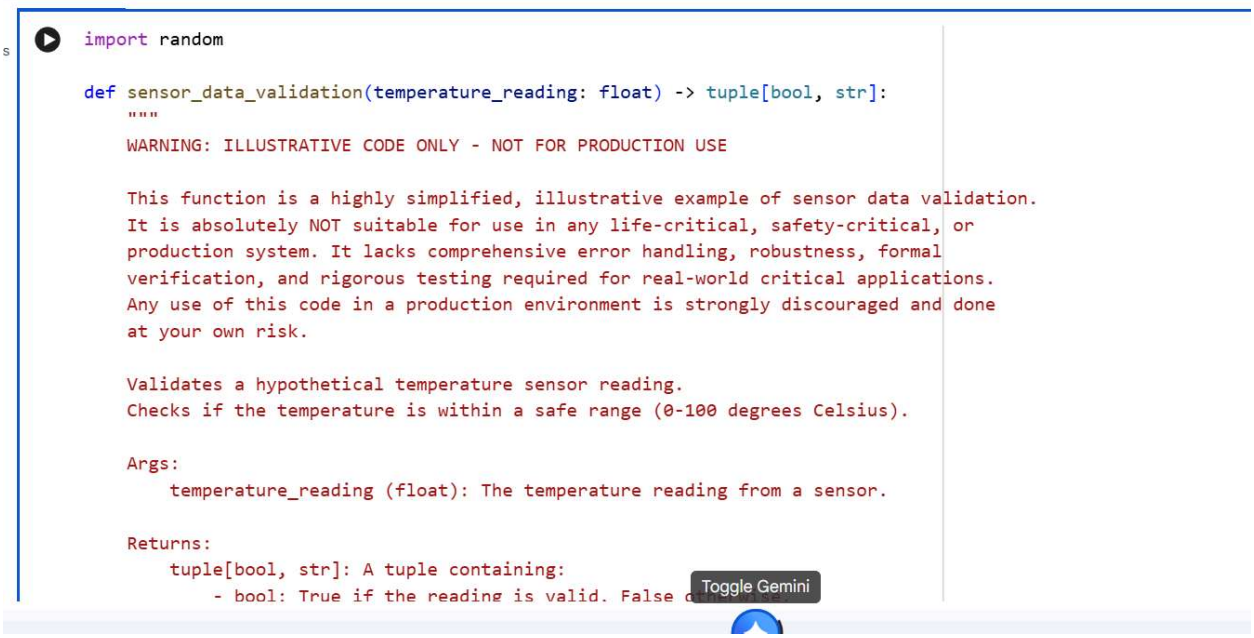
- Ask AI to generate code for a life-critical application (e.g., railway signaling, medical diagnosis).
- Discuss:

- Should AI code be used directly in such systems?
- What level of **human oversight** is mandatory?
- How do we balance efficiency vs safety?

#PROMPT:

Generate code for a life-critical application (e.g., railway signaling or medical diagnosis). Analyze the limitations of using AI-generated code in such systems. Discuss whether AI code should be used directly, the level of human oversight required, and how to balance efficiency with safety. Submit a short report with your reflections and recommendations.

#CODE:



```
import random

def sensor_data_validation(temperature_reading: float) -> tuple[bool, str]:
    """
    WARNING: ILLUSTRATIVE CODE ONLY - NOT FOR PRODUCTION USE

    This function is a highly simplified, illustrative example of sensor data validation.
    It is absolutely NOT suitable for use in any life-critical, safety-critical, or
    production system. It lacks comprehensive error handling, robustness, formal
    verification, and rigorous testing required for real-world critical applications.
    Any use of this code in a production environment is strongly discouraged and done
    at your own risk.

    Validates a hypothetical temperature sensor reading.
    Checks if the temperature is within a safe range (0-100 degrees Celsius).

    Args:
        temperature_reading (float): The temperature reading from a sensor.

    Returns:
        tuple[bool, str]: A tuple containing:
            - bool: True if the reading is valid, False
```

```
    MIN_TEMP = 0.0
    MAX_TEMP = 100.0

    if not isinstance(temperature_reading, (int, float)):
        return False, "Error: Temperature reading must be a number."

    if MIN_TEMP <= temperature_reading <= MAX_TEMP:
        return True, "Temperature reading is within the safe range."
    elif temperature_reading < MIN_TEMP:
        return False, f"Temperature reading ({temperature_reading}°C) is too low (below {MIN_TEMP}°C)."
    else:
        return False, f"Temperature reading ({temperature_reading}°C) is too high (above {MAX_TEMP}°C)."

# --- Example Usage (Illustrative) ---
print("--- Illustrative Sensor Data Validation ---")

# Valid reading
reading1 = 25.5
is_valid, message = sensor_data_validation(reading1)
print(f"Reading: {reading1}°C, Valid: {is_valid}, Message: {message}")

# Low reading
```

Toggle Gemini

```
# Low reading
reading2 = -5.0
is_valid, message = sensor_data_validation(reading2)
print(f"Reading: {reading2}°C, Valid: {is_valid}, Message: {message}")

# High reading
reading3 = 105.2
is_valid, message = sensor_data_validation(reading3)
print(f"Reading: {reading3}°C, Valid: {is_valid}, Message: {message}")

# Edge case: min
reading4 = 0.0
is_valid, message = sensor_data_validation(reading4)
print(f"Reading: {reading4}°C, Valid: {is_valid}, Message: {message}")

# Edge case: max
reading5 = 100.0
is_valid, message = sensor_data_validation(reading5)
print(f"Reading: {reading5}°C, Valid: {is_valid}, Message: {message}")

# Invalid type
reading6 = "twenty"
```

Toggle Gemini

```
# Invalid type
reading6 = "twenty"
is_valid, message = sensor_data_validation(reading6)
print(f"Reading: '{reading6}', Valid: {is_valid}, Message: {message}")

# Simulate some random readings
print("\n--- Simulating Random Readings ---")
for _ in range(5):
    simulated_reading = random.uniform(-20, 120)
    is_valid, message = sensor_data_validation(simulated_reading)
    print(f"Simulated Reading: {simulated_reading:.2f}°C, Valid: {is_valid}, Message: {message}")
```

#OUTPUT:

```
--- Illustrative Sensor Data Validation ---
Reading: 25.5°C, Valid: True, Message: Temperature reading is within the safe range.
Reading: -5.0°C, Valid: False, Message: Temperature reading (-5.0°C) is too low (below 0.0°C).
Reading: 105.2°C, Valid: False, Message: Temperature reading (105.2°C) is too high (above 100.0°C).
Reading: 0.0°C, Valid: True, Message: Temperature reading is within the safe range.
Reading: 100.0°C, Valid: True, Message: Temperature reading is within the safe range.
Reading: 'twenty', Valid: False, Message: Error: Temperature reading must be a number.

--- Simulating Random Readings ---
Simulated Reading: 26.90°C, Valid: True, Message: Temperature reading is within the safe range.
Simulated Reading: 52.13°C, Valid: True, Message: Temperature reading is within the safe range.
Simulated Reading: 114.44°C, Valid: False, Message: Temperature reading (114.43590418206526°C) is too high (above 100.0°C).
Simulated Reading: -9.54°C, Valid: False, Message: Temperature reading (-9.539857432136822°C) is too low (below 0.0°C).
Simulated Reading: 27.72°C, Valid: True, Message: Temperature reading is within the safe range.
```

#TASK DESCRIPTION-6:

Ethical Use of AI-Generated Code

- Students analyze real-world cases (e.g., plagiarism, copyright violation, misuse of open-source code).
- Identify **best practices** for ethically using AI code:
    - Attribution
    - Verification
    - Responsible deployment

#PROMPT:

Analyze real-world cases where AI-generated code led to ethical issues (e.g., plagiarism, copyright violation, or misuse of open-source code). Identify best practices for using AI code ethically, including proper attribution, verification of correctness, and responsible deployment. Submit a reflection report with your recommendations.

-------- Thank You-------