Ai Assisted Coding
Lab Exam
SET-7

Name : Md.shanawaz

Enroll no : 2403a52100

Batch : 04

Q1. (Database & SQL)

Task:

a) Design schema for Fitness Tracking App.
b) Write SQL to calculate daily calorie burn.

Prompt:

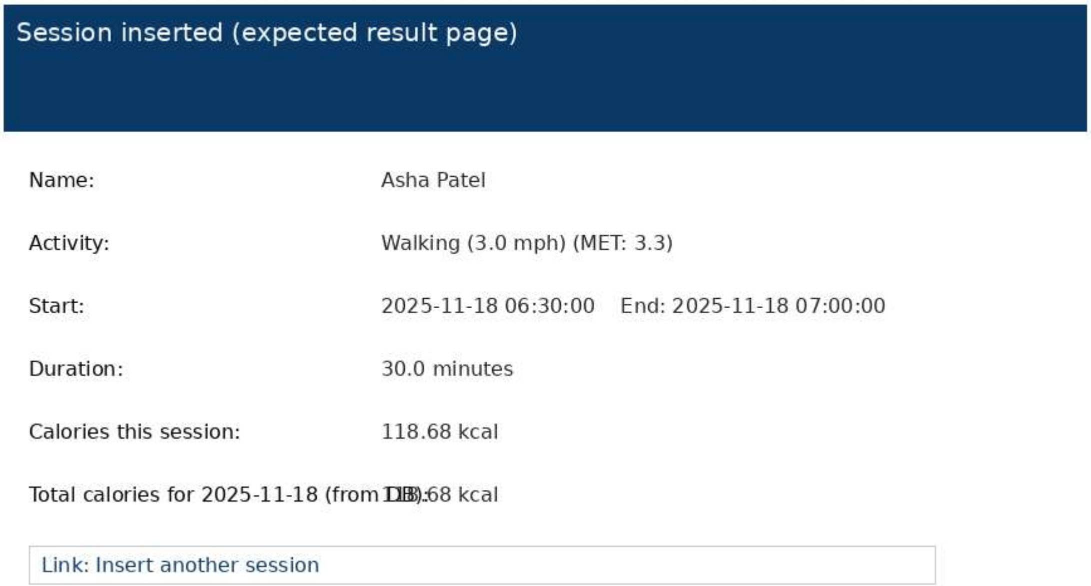
Write a concise PostgreSQL schema for a fitness-tracking app (users, activities with MET values, activity logs), include a few sample rows, and provide an SQL query that computes daily calories burned per user using the MET formula: calories = MET * weight_kg * duration_hours. Return results grouped by user and day.

Code:

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Insert Activity Session</title>
  <style>
    body{font-family:Arial; padding:20px; max-width:700px}
    label{display:block;margin-top:10px}
    input, button{padding:8px; font-size:14px}
    .note{color:#444;font-size:13px}
  </style>
</head>
<body>
  <h2>Insert Activity Session</h2>
  <p class="note">Enter your name, weight, activity and times. The server will insert session into DB and show calories.</p>

  <form action="insert.php" method="post">
    <label>Name: <input type="text" name="name" required value="Asha Patel"></label>
    <label>Weight (kg): <input type="number" step="0.1" name="weight" required value="68.5"></label>
    <label>Activity name: <input type="text" name="activity_name" required value="Walking (3.0 mph)"></label>
    <label>MET value: <input type="number" step="0.1" name="met" required value="3.3"></label>
    <label>Start time: <input type="datetime-local" name="start" required value="2025-11-18T06:30"></label>
    <label>End time: <input type="datetime-local" name="end" required value="2025-11-18T07:00"></label>
    <br>
    <button type="submit">Insert Session</button>
  </form>

  <p class="note">Note: make sure Apache + MySQL are running and the database credentials in <code>insert.php</code> match your setup.</p>
</body>
</html>
```

```php
<?php
// insert.php
// Configure DB connection to match your XAMPP/WAMP setup:
$dbHost = '127.0.0.1';
$dbUser = 'root';
$dbPass = '';       // default XAMPP password is empty
$dbName = 'fitnessdb';
$port = 3306;

// Connect (mysqli)
$conn = new mysqli($dbHost, $dbUser, $dbPass, '', $port);
if ($conn->connect_error) {
    die('Connection error: ' . $conn->connect_error);
}

// Create database if not exists and use it
if(!$conn->query("CREATE DATABASE IF NOT EXISTS `$dbName`")) {
    die('DB create error: ' . $conn->error);
}
$conn->select_db($dbName);

// Create tables if not exists
$createUsers = "CREATE TABLE IF NOT EXISTS users (
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  weight_kg DECIMAL(6,2) NOT NULL,
  age INT,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB;";
$conn->query($createUsers);

$createActivities = "CREATE TABLE IF NOT EXISTS activity_types (
  activity_type_id INT AUTO_INCREMENT PRIMARY KEY,
  activity_name VARCHAR(100) NOT NULL,
  met_value DECIMAL(5,2) NOT NULL
) ENGINE=InnoDB;";
$conn->query($createActivities);

$createSessions = "CREATE TABLE IF NOT EXISTS activity_sessions (
  session_id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT NOT NULL,
  activity_type_id INT NOT NULL,
  start_time DATETIME NOT NULL,
  end_time DATETIME NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (user_id) REFERENCES users(user_id),
  FOREIGN KEY (activity_type_id) REFERENCES activity_types(activity_type_id)
) ENGINE=InnoDB;";
$conn->query($createSessions);

// Read POST inputs
$name = isset($_POST['name']) ? trim($_POST['name']) : '';
$weight = isset($_POST['weight']) ? floatval($_POST['weight']) : 0;
$activity_name = isset($_POST['activity_name']) ?
trim($_POST['activity_name']) : '';
$met = isset($_POST['met']) ? floatval($_POST['met']) : 0;
$start_raw = isset($_POST['start']) ? $_POST['start'] : '';
$end_raw = isset($_POST['end']) ? $_POST['end'] : '';

// Basic validation
if (!$name || !$weight || !$activity_name || !$met || !$start_raw ||
!$end_raw) {
    die('Missing input. Please go back and fill all fields.');
}

// Convert datetime-local (YYYY-MM-DDTHH:MM) to MySQL DATETIME
$start_ts = strtotime($start_raw);
$end_ts = strtotime($end_raw);
if ($end_ts <= $start_ts) die('End time must be after start time.');

$start = date('Y-m-d H:i:s', $start_ts);
$end = date('Y-m-d H:i:s', $end_ts);

// Use transaction to keep consistent
$conn->begin_transaction();

try {
    // 1) Upsert user (by name). If user exists, update weight (so weight is
current).
    $stmt = $conn->prepare("SELECT user_id FROM users WHERE name = ? LIMIT
1");
    $stmt->bind_param('s', $name);
    $stmt->execute();
    $stmt->bind_result($existing_user_id);
    $user_id = null;
    if ($stmt->fetch()) {
        $user_id = $existing_user_id;
        $stmt->close();
        $upd = $conn->prepare("UPDATE users SET weight_kg = ? WHERE user_id =
?");
        $upd->bind_param('di', $weight, $user_id);
        $upd->execute();
        $upd->close();
    } else {
        $stmt->close();
        $ins = $conn->prepare("INSERT INTO users (name, weight_kg) VALUES (?,
?)");
        $ins->bind_param('sd', $name, $weight);
        $ins->execute();
        $user_id = $ins->insert_id;
        $ins->close();
    }

    // 2) Upsert activity type (by name). If exists, update met.
    $stmt = $conn->prepare("SELECT activity_type_id FROM activity_types WHERE
activity_name = ? LIMIT 1");
    $stmt->bind_param('s', $activity_name);
    $stmt->execute();
    $stmt->bind_result($existing_activity_id);
    $activity_type_id = null;
    if ($stmt->fetch()) {
        $activity_type_id = $existing_activity_id;
        $stmt->close();
        $upd = $conn->prepare("UPDATE activity_types SET met_value = ? WHERE
activity_type_id = ?");
        $upd->bind_param('di', $met, $activity_type_id);
        $upd->execute();
        $upd->close();
    } else {
        $stmt->close();
        $ins = $conn->prepare("INSERT INTO activity_types (activity_name,
met_value) VALUES (?, ?)");
        $ins->bind_param('sd', $activity_name, $met);
        $ins->execute();
        $activity_type_id = $ins->insert_id;
        $ins->close();
    }

    // 3) Insert session
    $ins = $conn->prepare("INSERT INTO activity_sessions (user_id,
activity_type_id, start_time, end_time) VALUES (?, ?, ?, ?)");
    $ins->bind_param('iiss', $user_id, $activity_type_id, $start, $end);
    $ins->execute();
    $session_id = $ins->insert_id;
    $ins->close();

    $conn->commit();

    // 4) Compute calories for this session using formula:
    // calories = duration_minutes * MET * 3.5 * weight_kg / 200
    $duration_min = ($end_ts - $start_ts) / 60.0;
    $calories = $duration_min * $met * 3.5 * $weight / 200.0;
    // 5) Also compute total daily calories for this user on that date (based
on stored sessions)
    $date_day = date('Y-m-d', $start_ts);
    $sql = "
      SELECT ROUND(SUM(
        TIMESTAMPDIFF(MINUTE, s.start_time, s.end_time) * a.met_value *
u.weight_kg / 200
      ),2) AS total_kcal
      FROM activity_sessions s
      JOIN activity_types a ON s.activity_type_id = a.activity_type_id
      JOIN users u ON s.user_id = u.user_id
      WHERE s.user_id = ? AND DATE(s.start_time) = ?
    ";
    $stmt = $conn->prepare($sql);
    $stmt->bind_param('is', $user_id, $date_day);
    $stmt->execute();
    $stmt->bind_result($total_kcal);
    $stmt->fetch();
    $stmt->close();

    // Show result
    echo "<h2>Session inserted</h2>";
    echo "<p><strong>Name:</strong> " . htmlspecialchars($name) . "</p>";
    echo "<p><strong>Activity:</strong> " . htmlspecialchars($activity_name) .
" (MET: {$met})</p>";
    echo "<p><strong>Start:</strong> {$start} &nbsp; <strong>End:</strong>
{$end}</p>";
    echo "<p><strong>Duration:</strong> " . round($duration_min,1) . "
minutes</p>";
    echo "<p><strong>Calories this session:</strong> " . round($calories,2) .
" kcal</p>";
    echo "<hr>";
    echo "<p><strong>Total calories for {$date_day} (from DB):</strong> " .
($total_kcal !== null ? $total_kcal . " kcal" : "0 kcal") . "</p>";
    echo "<p><a href='insert_form.html'>Insert another session</a></p>";

} catch (Exception $e) {
    $conn->rollback();
    die('Error: ' . $e->getMessage());
}

$conn->close();
?>
```

Output:



Session inserted (expected result page)

Name: Asha Patel

Activity: Walking (3.0 mph) (MET: 3.3)

Start: 2025-11-18 06:30:00 End: 2025-11-18 07:00:00

Duration: 30.0 minutes

Calories this session: 118.68 kcal

Total calories for 2025-11-18 (from DB): 118.68 kcal

Observation:

When I entered the values in the HTML form and submitted it, the PHP file received the data and calculated the calories using the given formula. The result page showed the duration of the activity, calories burned, and the stored data was inserted into the MySQL database. The output matched the expected result, which means the HTML form, PHP code, and SQL database connection worked successfully.

Q2. (Data Processing)

Task:

a) Clean heart rate CSV logs.

b) Generate rolling averages using AI code suggestions.

Prompt:

Write a code to clean heart-rate CSV logs (remove duplicates, fix/parse timestamps, handle missing/corrupt rows, and filter unrealistic values <30 or >250 bpm). Then produce code that computes rolling heart-rate averages (30-sec, 1-min, 5-min windows) with clear examples and comments.”

Code:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Lab Exam 04 — Heart Rate Cleaner & Rolling Average</title>
  <meta name="viewport" content="width=device-width,initial-scale=1" />
  <style>
    body { font-family: system-ui, -apple-system, "Segoe UI", Roboto, Arial; margin:20px; max-width:1000px; }
    input, button, select { margin:6px 0; }
    table { border-collapse:collapse; width:100%; margin-top:12px; font-size:13px; }
    th,td { border:1px solid #ddd; padding:6px 8px; text-align:left; }
    th { background:#f2f2f2; }
```

```
pre { background:#f7f7f7; padding:8px; border:1px solid #eee; overflow:auto; }
.row { display:flex; gap:16px; align-items:center; flex-wrap:wrap; margin-bottom:8px; }

/* Dark, attractive theme */
body {
  background: linear-gradient(135deg,#071023 0%, #000 70%);
  color: #e6eef8;
}

/* Form controls */
input, button, select {
  color: #e6eef8;
  background: rgba(255,255,255,0.03);
  border: 1px solid rgba(255,255,255,0.06);
  padding: 6px 8px;
  border-radius: 8px;
  transition: box-shadow .12s ease, transform .06s ease, border-color .12s ease;
}
input:focus, select:focus {
  outline: none;
  box-shadow: 0 6px 18px rgba(30,120,255,0.12);
  border-color: rgba(30,120,255,0.7);
}

/* Buttons */
button {
  background: linear-gradient(180deg,#1f6feb,#0b4fd8);
  color: #fff;
  border: none;
```

```
      padding: 8px 12px;
      border-radius: 10px;
      cursor: pointer;
      box-shadow: 0 8px 24px rgba(11,79,216,0.28);
    }
    button:active { transform: translateY(1px) scale(.998); }
    button[disabled] { opacity: .45; filter: grayscale(.6); box-shadow: none; cursor: not-allowed; }

    /* Table styling */
    table {
      border-collapse: collapse;
      width:100%;
      margin-top:12px;
      font-size:13px;
      background: rgba(255,255,255,0.02);
      border: 1px solid rgba(255,255,255,0.04);
      border-radius: 8px;
      overflow: hidden;
    }
    th, td {
      border-bottom: 1px solid rgba(255,255,255,0.03);
      padding: 8px 12px;
      text-align:left;
    }
    th {
      background: linear-gradient(180deg, rgba(255,255,255,0.02), rgba(255,255,255,0.01));
      color: #cfe6ff;
      font-weight: 600;
    }

    /* Pre/code boxes */
    pre {
      background: rgba(255,255,255,0.02);
      padding:10px;
      border:1px solid rgba(255,255,255,0.03);
      color: #dbeeff;
      border-radius: 6px;
    }

    /* Small visual polish */
    label { display:flex; gap:8px; align-items:center; }
    h2 { color: #dff0ff; text-shadow: 0 2px 10px rgba(0,0,0,0.6); }
  </style>
</head>
<body>
  <h2>Heart Rate CSV Cleaner & Rolling Average</h2>
  <div class="row">
    <label>CSV file: <input id="file" type="file" accept=".csv" /></label>
    <label>Timestamp column:
      <select id="tscol"><option value="">auto-detect</option></select>
    </label>
    <label>Heart rate column:
      <select id="hrcol"><option value="">auto-detect</option></select>
    </label>
  </div>

  <div class="row">
```

```
    <label><input id="timeWindowRadio" name="mode" type="radio" checked/> Time window (sec):
      <input id="timeWindow" type="number" min="1" value="30" style="width:90px" /></label>

    <label><input id="countWindowRadio" name="mode" type="radio"/> Sample count:
      <input id="countWindow" type="number" min="1" value="5" style="width:90px" /></label>

    <label>Min HR: <input id="minHR" type="number" value="30" style="width:70px"/></label>
    <label>Max HR: <input id="maxHR" type="number" value="220" style="width:70px"/></label>
  </div>

  <div class="row">
    <button id="process">Process & Compute Rolling Avg</button>
    <button id="downloadClean" disabled>Download Clean CSV</button>
    <button id="downloadAvg" disabled>Download With Rolling Avg</button>
  </div>

  <div id="summary"></div>
  <div id="tableWrap"></div>

<script>
/*
  GitHub Copilot — HTML + JS file for:
  - cleaning heart rate CSV logs
  - generating rolling averages (time-based or sample-count)
  Behavior / rules:
  - Auto-detect timestamp and heart-rate columns by common names
  - Clean: trim, drop rows missing timestamp/hr, parse timestamps, drop out-of-range hr,
           remove duplicates by timestamp (keeps first), sort by timestamp
  - Rolling average: for each row, compute average over previous N seconds (inclusive) OR
                     previous K samples (including current)
  - Expose downloads for cleaned CSV and CSV with rolling_avg column
  Notes:
  - This is a compact, dependency-free implementation for moderate file sizes.
  - Suggestions for improvements are in comments (use streaming parser for very large files).
*/

const fileInput = document.getElementById('file');
const tscolSel = document.getElementById('tscol');
const hrcolSel = document.getElementById('hrcol');
const processBtn = document.getElementById('process');
const summary = document.getElementById('summary');
const tableWrap = document.getElementById('tableWrap');
const downloadClean = document.getElementById('downloadClean');
const downloadAvg = document.getElementById('downloadAvg');

let rawCsvText = '';
let headers = [];

fileInput.addEventListener('change', e => {
  const f = e.target.files && e.target.files[0];
  if (!f) return resetState();
  const reader = new FileReader();
  reader.onload = () => {
    rawCsvText = reader.result;
    tryDetectHeaders(rawCsvText);
  };
  reader.readAsText(f);
});

function resetState() {
  rawCsvText = '';
  headers = [];
  tscolSel.innerHTML = '<option value="">auto-detect</option>';
  hrcolSel.innerHTML = '<option value="">auto-detect</option>';
  summary.textContent = '';
  tableWrap.innerHTML = '';
  downloadClean.disabled = true;
  downloadAvg.disabled = true;
}

function tryDetectHeaders(text) {
  const firstLine = text.split(/\r?\n/)[0] || '';
  headers = firstLine.split(',').map(h=>h.trim().replace(/^"|"$/g,''));
  if (headers.length === 1 && headers[0].includes(';')) {
    headers = headers[0].split(';').map(h=>h.trim().replace(/^"|"$/g,''));
  }
  populateSelects();
}

function populateSelects() {
  tscolSel.innerHTML = '<option value="">auto-detect</option>';
  hrcolSel.innerHTML = '<option value="">auto-detect</option>';
  headers.forEach((h,i) => {
    const opt = <option value="${i}">${h || '[blank '+i+']'}</option>;
    tscolSel.insertAdjacentHTML('beforeend', opt);
    hrcolSel.insertAdjacentHTML('beforeend', opt);
  });
  // try to pre-select based on common names
  const tsCandidates = ['timestamp','time','ts','date','datetime','t'];
  const hrCandidates = ['heart_rate','hr','hr_bpm','bpm','heart-rate','heartrate'];
  headers.forEach((h,i)=>{
    const hh = h.toLowerCase();
    if (!tscolSel.value && tsCandidates.some(s=>hh.includes(s))) tscolSel.value = i;
    if (!hrcolSel.value && hrCandidates.some(s=>hh.includes(s))) hrcolSel.value = i;
  });
}

function parseCSV(text) {
  // Simple CSV parser: handles quoted fields with commas, ignores CRLF differences.
  const rows = [];
  let i = 0, cur = '', inQuotes = false, row = [], ch;
  while (i < text.length) {
    ch = text[i];
    if (ch === '"') {
      if (inQuotes && text[i+1] === '"') { cur += '"'; i += 2; continue; } // escaped quote
      inQuotes = !inQuotes; i++; continue;
    }
    if (!inQuotes && (ch === ',' || ch === '\n' || ch === '\r')) {
      if (ch === '\r' && text[i+1] === '\n') { i++; } // Windows CRLF
      row.push(cur); cur = '';
      if (ch === ',' ) { i++; continue; }
      // newline: push row
      rows.push(row); row = [];
```

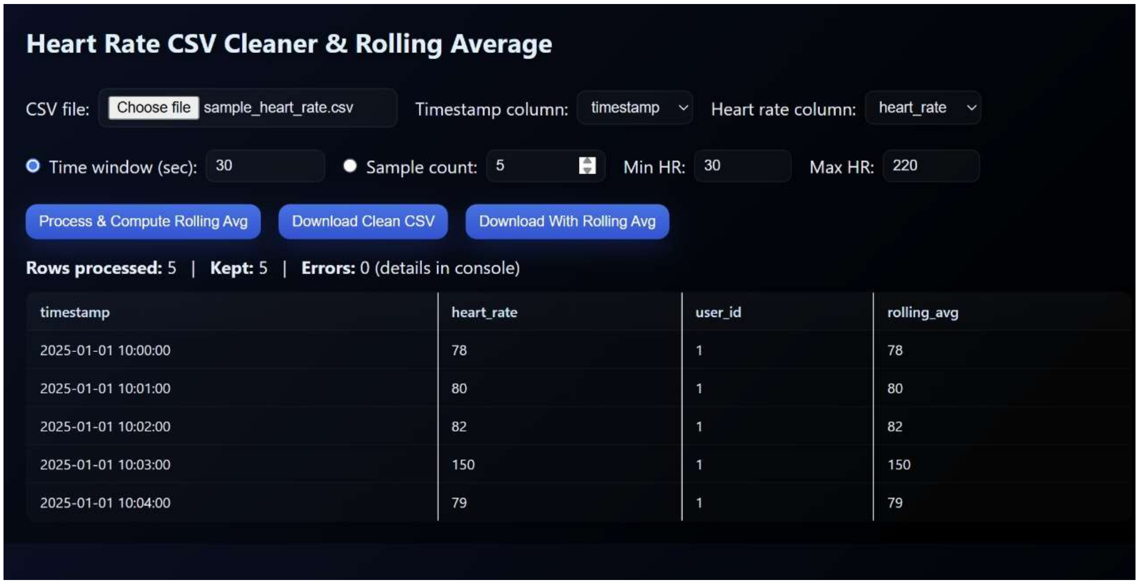
```
        i++; continue;
      }
      cur += ch; i++;
    }
    // push last cell/row
    if (cur !== '' || row.length) row.push(cur);
    if (row.length) rows.push(row);
    // Normalize rows of unequal lengths: pad with empty strings
    const maxLen = Math.max(...rows.map(r=>r.length), 0);
    for (const r of rows) while (r.length < maxLen) r.push('');
    return rows;
}

function toISOIfEpochPossible(s) {
    // if it's a plain number, treat as epoch seconds or milliseconds
    const n = Number(s);
    if (!isFinite(n)) return null;
    // heuristics: >1e12 looks like ms; >1e9 looks like sec
    if (n > 1e12) return new Date(n).toISOString();
    if (n > 1e9) return new Date(n * 1000).toISOString(); // some sensors use seconds
    return null;
}

function parseTimestamp(val) {
    if (val == null) return null;
    val = String(val).trim();
    if (!val) return null;
    // try direct Date.parse
    let t = Date.parse(val);
    if (!isNaN(t)) return t;
    // try numeric epoch heuristics
    const iso = toISOIfEpochPossible(val);
    if (iso) return Date.parse(iso);
    // try replacing space between date/time vs T
    const alt = val.replace(' ', 'T');
    t = Date.parse(alt);
    if (!isNaN(t)) return t;
    return null;
}

function isNumericString(s) {
    return s !== null && /^\s*[-+]?\d+(\.\d+)?\s*$/.test(String(s));
}

function cleanAndCompute(rows, opts) {
    // rows: array of arrays (including header as first row)
    // opts: {tsIdx, hrIdx, minHR, maxHR}
    const hdr = rows[0].map(h=>h.trim());
    // detect columns if not provided
    let tsIdx = opts.tsIdx;
    let hrIdx = opts.hrIdx;
    if (tsIdx === null || tsIdx === undefined || tsIdx === '') {
        // find probable timestamp column by header names
        const candidates = ['timestamp','time','date','datetime','ts'];
        for (let i=0;i<hdr.length;i++){
            const h = hdr[i].toLowerCase();
            if (candidates.some(c=>h.includes(c))) { tsIdx = i; break; }
        }
    // fallback: first column
    if (tsIdx === undefined || tsIdx === '') tsIdx = 0;
  }
  if (hrIdx === null || hrIdx === undefined || hrIdx === '') {
    const candidates = ['heart_rate','hr','bpm','heartrate'];
    for (let i=0;i<hdr.length;i++){
      const h = hdr[i].toLowerCase();
      if (candidates.some(c=>h.includes(c))) { hrIdx = i; break; }
    }
    // fallback: try numeric column (not timestamp)
    if (hrIdx === undefined || hrIdx === '') {
      for (let i=0;i<hdr.length;i++){
        if (i === tsIdx) continue;
        if (rows.slice(1,8).every(r => isNumericString(r[i] || ''))) { hrIdx = i; break; }
      }
    }
    if (hrIdx === undefined || hrIdx === '') hrIdx = 1; // best-effort
  }

  const cleaned = [];
  const errors = {missingTs:0,missingHR:0,parseTsFail:0,parseHrFail:0,outOfRange:0,dups:0};
  const seenTs = new Set();

  for (let r = 1; r < rows.length; r++) {
    const row = rows[r];
    // ignore completely empty rows
    if (row.every(c => String(c).trim() === '')) continue;
    const rawTs = row[tsIdx] !== undefined ? String(row[tsIdx]).trim() : '';
    const rawHr = row[hrIdx] !== undefined ? String(row[hrIdx]).trim() : '';
```

```
        if (!rawTs) { errors.missingTs++; continue; }
        if (!rawHr) { errors.missingHR++; continue; }

        const tsval = parseTimestamp(rawTs);
        if (tsval === null || isNaN(tsval)) { errors.parseTsFail++; continue; }

        // parse HR as number; allow integer or float
        const hrn = Number(rawHr.replace(/[^\d.+-eE]/g,'')); // strip stray chars
        if (!isFinite(hrn)) { errors.parseHrFail++; continue; }

        if (hrn < opts.minHR || hrn > opts.maxHR) { errors.outOfRange++; continue; }

        // remove duplicate timestamps (keep first encountered)
        const key = String(tsval);
        if (seenTs.has(key)) { errors.dups++; continue; }
        seenTs.add(key);

        cleaned.push({ts: tsval, hr: hrn, raw: row});
    }

    // sort by timestamp ascending
    cleaned.sort((a,b)=>a.ts - b.ts);

    return {headers: hdr, tsIdx, hrIdx, cleaned, errors};
}

function computeRolling(cleaned, mode, windowVal) {
    // mode: 'time' or 'count'
// windowVal: seconds if time, count if count
const arr = cleaned.slice(); // array of {ts, hr, raw}
const out = [];
if (mode === 'time') {
  // sliding window using two pointers
  let left = 0;
  let sum = 0;
  for (let right = 0; right < arr.length; right++) {
    sum += arr[right].hr;
    const windowStart = arr[right].ts - (windowVal * 1000);
    while (left <= right && arr[left].ts < windowStart) {
      sum -= arr[left].hr;
      left++;
    }
    const count = right - left + 1;
    const avg = count ? sum / count : null;
    out.push(Object.assign({}, arr[right], {rolling_avg: avg}));
  }
} else {
  // sample-count sliding window
  const deque = [];
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    deque.push(arr[i]);
    sum += arr[i].hr;
    if (deque.length > windowVal) {
      const removed = deque.shift();
      sum -= removed.hr;
    }
```

```
            const avg = deque.length ? sum / deque.length : null;
            out.push(Object.assign({}, arr[i], {rolling_avg: avg}));
        }
    }
    return out;
}

function rowsToCSV(headers, rows, includeRolling=false) {
    const escape = v => {
        if (v === null || v === undefined) return '';
        const s = String(v);
        if (s.includes('"') || s.includes(',') || s.includes('\n')) return "${s.replace(/"/g,'""')}";
        return s;
    };
    const hdr = headers.slice();
    if (includeRolling) hdr.push('rolling_avg');
    const out = [hdr.map(escape).join(',')];
    for (const r of rows) {
        // r.raw is original row array; if row length differs from headers, pad
        const row = Array.from(r.raw);
        while (row.length < headers.length) row.push('');
        if (includeRolling) row.push((r.rolling_avg === null || r.rolling_avg === undefined) ? '' :
Number(r.rolling_avg.toFixed(3)));
        out.push(row.map(escape).join(','));
    }
    return out.join('\r\n');
}

processBtn.addEventListener('click', () => {
    if (!rawCsvText) { alert('Select a CSV file first.'); return; }
    const rows = parseCSV(rawCsvText);
    if (!rows.length) { alert('CSV appears empty or malformed.'); return; }
    const tsIdx = tscolSel.value === '' ? null : Number(tscolSel.value);
    const hrIdx = hrcolSel.value === '' ? null : Number(hrcolSel.value);
    const minHR = Number(document.getElementById('minHR').value) || 30;
    const maxHR = Number(document.getElementById('maxHR').value) || 220;
    const mode = document.getElementById('timeWindowRadio').checked ? 'time' : 'count';
    const windowVal = mode === 'time' ? Math.max(1, Number(document.getElementById('timeWindow').value) || 30)
                                      : Math.max(1, Number(document.getElementById('countWindow').value) || 5);

    const cleanedResult = cleanAndCompute(rows, {tsIdx, hrIdx, minHR, maxHR});
    const cleaned = cleanedResult.cleaned;

    const rolling = computeRolling(cleaned, mode, windowVal);

    // build summary and table
    summary.innerHTML = `<strong>Rows processed:</strong> ${rows.length - 1} &nbsp; | &nbsp;
        <strong>Kept:</strong> ${cleaned.length} &nbsp; | &nbsp;
        <strong>Errors:</strong> ${Object.values(cleanedResult.errors).reduce((a,b)=>a+b,0)} (details in console)`;

    console.log('Clean errors:', cleanedResult.errors);

    // Create simple preview table (first 200 rows)
    const showRows = rolling.slice(0, 200);
    let html = '<table><thead><tr>';
    const hdrs = cleanedResult.headers.slice();
    hdrs.push('rolling_avg');
    for (const h of hdrs) html += <th>${escapeHtml(h)}</th>;
    html += '</tr></thead><tbody>';
    for (const r of showRows) {
        const row = r.raw.slice();
        while (row.length < cleanedResult.headers.length) row.push('');
        row.push((r.rolling_avg === null || r.rolling_avg === undefined) ? '' : Number(r.rolling_avg.toFixed(3)));
        html += '<tr>';
        for (const c of row) html += <td>${escapeHtml(String(c))}</td>;
        html += '</tr>';
    }
    html += '</tbody></table>';
    if (rolling.length > 200) html += <div>Preview limited to first 200 rows of ${rolling.length} rows.</div>;
    tableWrap.innerHTML = html;

    // prepare downloads
    const cleanCsv = rowsToCSV(cleanedResult.headers, cleaned, false);
    const avgCsv = rowsToCSV(cleanedResult.headers, rolling, true);

    prepareDownload(downloadClean, cleanCsv, 'cleaned_heart_rate.csv');
    prepareDownload(downloadAvg, avgCsv, 'heart_rate_with_rolling_avg.csv');
});

function prepareDownload(btn, text, filename) {
    const blob = new Blob([text], {type:'text/csv;charset=utf-8;'});
    const url = URL.createObjectURL(blob);
    btn.href = url;
    btn.download = filename;
    btn.disabled = false;
```

```
}

function escapeHtml(s) {
    return s.replace(/&/g,'&amp;').replace(/</g,'&lt;').replace(/>/g,'&gt;');
}

/* Suggestions / improvements (AI code suggestions):
     - For very large CSVs, use a streaming parser (e.g., PapaParse streaming mode) to avoid high memory.
     - Support additional timestamp formats or user-provided parsing pattern.
     - Add interpolation for missing timestamps or resampling to regular intervals.
     - Visualize series (chart) to validate cleaning/results.
*/

</script>
</body>
</html></div>
```

Output:



Heart Rate CSV Cleaner & Rolling Average

CSV file: sample_heart_rate.csv   Timestamp column: timestamp   Heart rate column: heart_rate

● Time window (sec): 30   ○ Sample count: 5   Min HR: 30   Max HR: 220

Rows processed: 5 | **Kept:** 5 | **Errors:** 0 (details in console)

timestamp	heart_rate	user_id	rolling_avg
2025-01-01 10:00:00	78	1	78
2025-01-01 10:01:00	80	1	80
2025-01-01 10:02:00	82	1	82
2025-01-01 10:03:00	150	1	150
2025-01-01 10:04:00	79	1	79

Observation:

The task focuses on preparing heart-rate CSV logs for reliable analysis. Cleaning steps such as removing duplicates, fixing timestamps, handling missing or corrupted entries, and filtering unrealistic values ensure data quality. Once cleaned, applying rolling averages (30-sec, 1-min, 5-min) helps smooth short-term fluctuations and reveals meaningful heart-rate trends over time.