Q1.

a. Using a Git workflow, show with commands how you would create a branch, commit changes and push them to a GitHub repository.

```
git clone <url>
cd <repo>
git checkout -b feature/login-page
git status
git add .
git commit -m "Add login page"
git push -u origin feature/login-page
```

b. A college results page needs the following: a page title, a menu of links, a table of student marks, a photo of the college and an enquiry form. Choose suitable HTML5 elements for each part and state why you chose them.

Part of the page	Element chosen	Reason
Page title	`<header>` with `<h1>`	`<header>` marks the top of the page, and `<h1>` is the main heading
Menu of links	`<nav>` with `<ul>`, `<li>`, `<a>`	`<nav>` marks navigation, the list groups the links, `<a href>` makes each one clickable
Table of marks	`<table>`, `<tr>`, `<th>`, `<td>`	Marks are tabular data with rows and columns
College photo	`<img src alt>`	Shows the image, and `alt` gives a text description for accessibility
Enquiry form	`<form>` with `<input>`, `<textarea>`, `<button>`	Collects and submits user input
Main content	`<main>` and `<section>`	Semantic grouping, which helps search engines and screen readers

c. Compare localStorage, sessionStorage and cookies on lifetime, capacity and server access, and decide which one suits storing a login token.

Feature	Local Storage	Session Storage	Cookies
Lifetime	Persists until cleared manually	Until the tab or window closes	Until the expiry date (or session end if none set)

Feature	Local Storage	Session Storage	Cookies
Capacity	About 5–10 MB	About 5 MB	About 4 KB per cookie
Sent to server	No	No	Yes, automatically with every HTTP request
Access from JS	Yes	Yes	Yes, unless HttpOnly
Scope	Same origin	Same origin and same tab	Domain and path (configurable)

Login token: use a cookie with the HttpOnly, Secure and SameSite flags. JavaScript cannot read an HttpOnly cookie, so an XSS attack cannot steal the token, and the browser sends it to the server automatically. localStorage and sessionStorage are readable by any script on the page, so they are vulnerable to XSS. (Cookies need CSRF protection, which SameSite and CSRF tokens provide.)

d. Explain how Node.js handles asynchronous operations using callbacks and promises, and why this suits I/O-heavy applications.

Node.js runs JavaScript on a single thread with an **event loop**. When it meets an I/O operation (file, network, DB), it hands the work to libuv, which uses the OS or a thread pool. The main thread carries on, and when the operation completes its callback is queued and run by the event loop.

```
// Callback
fs.readFile("a.txt", "utf8", (err, data) => {
  if (err) return console.error(err);
  console.log(data);
});

// Promise
fs.promises.readFile("a.txt", "utf8")
  .then(console.log)
  .catch(console.error);

// async/await (syntactic sugar over promises)
const data = await fs.promises.readFile("a.txt", "utf8");
```

- **Callbacks** are simple but nest deeply ("callback hell") and make error handling repetitive.
- **Promises** represent a future value, can be chained and have a single .catch.
- Promise callbacks run in the microtask queue, which has priority over the macrotask queue (timers, I/O).

Why it suits I/O-heavy apps: the thread never blocks while waiting for a DB or network response, so one process can serve thousands of concurrent connections with low memory overhead. It is a poor fit for CPU-heavy work, which blocks the event loop unless moved to worker threads.

e. Explain the main data types and sources used in web analytics, and describe two data collection methods with an example each.

Data types:

- **Behavioural:** page views, clicks, scroll depth, session duration, bounce rate, conversion paths.
- **Demographic and technical:** location, device, browser, OS, screen size.
- **Acquisition:** traffic source or medium (organic, paid, referral, direct, social).
- **Transactional:** orders, revenue, cart events.
- **Qualitative:** survey responses, feedback.

Sources: server log files, JavaScript tracking tags, cookies and SDKs, third-party tools (Google Analytics), CRM and ad platforms, social media APIs.

Collection methods:

1. **Server log analysis:** the web server records each request (IP, timestamp, URL, status code, user agent). *Example:* analysing Apache or Nginx logs to find the most requested pages and 404 errors.
2. **JavaScript page tagging:** a snippet on each page sends events to an analytics server. *Example:* the Google Analytics gtag.js snippet recording page views and button clicks.

(Others: cookies, surveys, A/B tests.)

f. Explain the principles of data visualization and how visual perception (position, colour, size) influences chart choice.

Principles: clarity and accuracy (no distortion, honest axes starting at zero for bars), a high data-ink ratio (remove clutter), appropriate chart type for the data and question, effective use of colour, clear labels, titles and legends, consistency, and a focus on the message (hierarchy, annotation).

Visual perception (Cleveland and McGill's ranking of accuracy): position on a common scale > length > angle/slope > area > colour saturation/hue.

- **Position:** the most accurate channel, so use scatter plots, line charts and bar charts for precise comparison.
- **Colour:** good for categories (hue) or ordered values (a sequential lightness ramp), but poor for exact values. Use a limited palette and consider colour-blind-safe schemes.

- **Size (area):** harder to compare precisely, so use it as a secondary encoding (bubble charts) and avoid it for key comparisons.

So bar charts beat pie charts for comparing quantities, and line charts suit trends.

Q2.

a. Explain the basic MongoDB CRUD operations with their commands. Describe the role of a Mongoose schema and model in connecting an Express application to MongoDB.

CRUD operations in MongoDB (4 Marks)

Operation	Commands	Purpose
Create	`insertOne()`, `insertMany()`	Add new documents
Read	`find()`, `findOne()`	Retrieve documents, optionally with a filter
Update	`updateOne()`, `updateMany()`	Change fields of existing documents
Delete	`deleteOne()`, `deleteMany()`	Remove documents

In Mongoose, the equivalents are `Model.create()`, `Model.find()`, `Model.findById()`, `Model.findByIdAndUpdate()` and `Model.findByIdAndDelete()`.

Mongoose schema and model (4 Marks)

- **Mongoose** is a library that connects Node.js to MongoDB and adds structure to the data.
- A **schema** defines the fields of a document, their types and rules (required, minimum value, default value).
- A **model** is created from the schema and is used to perform the CRUD operations on a collection.
- This gives **validation** before data is saved, and keeps the code organized.

Connection (2 Marks): the application calls `mongoose.connect()` with the database URL (from an environment variable) using `await`, and handles a failed connection with try/catch.

b. Explain the role of Express middleware, and describe how the next() function and an error-handling middleware control the flow of a request.

1. Middleware (3 Marks)

- Middleware is a function that runs **between the request and the response**.
- It has access to the request (`req`), the response (`res`) and `next`.

- It can **read or change** the request, **end** the request by sending a response, or **pass control** to the next function.
- Examples: logging, reading JSON data (`express.json()`), checking login, and handling errors.

2. Role of `next()` (4 Marks)

- Middleware runs **in the order in which it is defined**.
- `next()` passes control to the **next middleware or route**.
- If `next()` is **not called** and no response is sent, the request **hangs**.
- If a middleware sends a response without calling `next()` (for example, 401 Unauthorized for a user who is not logged in), the chain **stops** there.
- `next(err)` skips all the normal middleware and goes straight to the **error-handling middleware**.

3. Error-handling middleware (3 Marks)

- It has **four parameters**: `(err, req, res, next)`.
- It is defined **after all the routes**.
- It logs the error, sets the **status code** (400, 404 or 500), and sends one **consistent JSON message**.
- Route handlers use **try/catch** and call `next(err)`, so all errors reach this one place.

Q3.

a. Write a TypeScript program that defines an interface for a Student, stores several students in an array and uses control structures to filter and display those who scored above a given mark.

```
interface Student {
  id: number;
  name: string;
  marks: number;
}

const students: Student[] = [
  { id: 1, name: "Asha", marks: 82 },
  { id: 2, name: "Ravi", marks: 67 },
```

```
  { id: 3, name: "Meera", marks: 91 },
  { id: 4, name: "John",  marks: 55 },
];

function displayAbove(list: Student[], threshold: number): void {
  const result: Student[] = [];
  for (const s of list) {
    if (s.marks > threshold) {
      result.push(s);
    }
  }

  if (result.length === 0) {
    console.log(`No student scored above ${threshold}`);
    return;
  }
  console.log(`Students scoring above ${threshold}:`);
  result.forEach(s => console.log(`${s.id}. ${s.name} - ${s.marks}`));
}

displayAbove(students, 70);
// Students scoring above 70:
// 1. Asha - 82
// 3. Meera - 91
```

b. Develop a TypeScript module that fetches a list of users from a REST API using the fetch API with async/await, parses the JSON response and displays the data in the DOM. Include error handling.

```
interface User {
  id: number;
```

```
  name: string;
  email: string;
}

const API_URL = "https://jsonplaceholder.typicode.com/users";

async function fetchUsers(): Promise<User[]> {
  const response = await fetch(API_URL);
  if (!response.ok) {
    throw new Error(`HTTP error: ${response.status}`);
  }
  return (await response.json()) as User[];
}

function renderUsers(users: User[]): void {
  const list = document.getElementById("user-list") as HTMLUListElement;
  list.innerHTML = "";
  users.forEach(u => {
    const li = document.createElement("li");
    li.textContent = `${u.name} (${u.email})`;
    list.appendChild(li);
  });
}

function showError(message: string): void {
  const box = document.getElementById("error") as HTMLDivElement;
  box.textContent = message;
}

async function init(): Promise<void> {
```

```
  try {
    const users = await fetchUsers();
    renderUsers(users);
  } catch (err) {
    showError(err instanceof Error ? err.message : "Something went wrong");
  }
}

init();
```

HTML:

```
<div id="error"></div>
<ul id="user-list"></ul>
<script type="module" src="users.js"></script>
```

Q4.

a. Build a controlled login form in React with validation for email and password, handling input events and displaying error messages.

```
import { useState, ChangeEvent, FormEvent } from "react";

interface Errors { email?: string; password?: string }

export default function LoginForm() {
  const [form, setForm] = useState({ email: "", password: "" });
  const [errors, setErrors] = useState<Errors>({});
  const [submitted, setSubmitted] = useState(false);

  const validate = (values = form): Errors => {
    const e: Errors = {};
```

```
  if (!values.email) e.email = "Email is required";
  else if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(values.email))
    e.email = "Enter a valid email";
  if (!values.password) e.password = "Password is required";
  else if (values.password.length < 8)
    e.password = "Password must be at least 8 characters";
  return e;
 };

 const handleChange = (ev: ChangeEvent<HTMLInputElement>) => {
  const { name, value } = ev.target;
  setForm(prev => ({ ...prev, [name]: value }));  // controlled input
 };

 const handleSubmit = (ev: FormEvent) => {
  ev.preventDefault();
  const e = validate();
  setErrors(e);
  if (Object.keys(e).length === 0) {
    setSubmitted(true);
    // call login API here
  }
 };

 return (
  <form onSubmit={handleSubmit} noValidate>
    <label>Email
      <input name="email" type="email" value={form.email}
onChange={handleChange} />
    </label>
    {errors.email && <p style={{ color: "red" }}>{errors.email}</p>}
```

```
    <label>Password
      <input name="password" type="password" value={form.password}
onChange={handleChange} />
    </label>
    {errors.password && <p style={{ color: "red" }}>{errors.password}</p>}

    <button type="submit">Login</button>
    {submitted && <p>Login successful!</p>}
  </form>
 );
}
```

b. Analyze the differences between client-side rendering in React and server-side rendering in Next.js, and compare the App router with the Pages router in terms of routing, performance and SEO.

Aspect	Client-side rendering (React/SPA)	Server-side rendering (Next.js)
Where HTML is built	In the browser via JS	On the server per request (or at build time for SSG)
Initial response	Nearly empty HTML plus a JS bundle	Fully rendered HTML
First paint (FCP/LCP)	Slower, since it waits for JS download, execution and data fetch	Faster, content visible immediately
SEO	Weaker (crawlers may not run JS well)	Strong, as content is in the HTML
Subsequent navigation	Very fast, no page reloads	Fast, with hydration and client navigation
Server load	Low (static hosting)	Higher (rendering per request, unless cached)

Aspect	Client-side rendering (React/SPA)	Server-side rendering (Next.js)
Best for	Dashboards, apps behind login	Public, content and SEO-driven sites

---

Q5.

a. An online retailer, has exported 50,000 customer-order records to orders.csv to build a model that predicts customer spending. Preliminary inspection shows:

Column	Problem observed
customer_id	About 1,200 rows appear twice (the same order was exported twice).
age	8% missing values; a few entries are 0 or 250.
city	Inconsistent entries: "Pune", "pune ", "PUNE", "Pnq".
order_date	Stored as text in mixed formats (12/03/2026, 2026-03-12).
gender	Categorical text (Male / Female / Other).
order_amount	Ranges from ₹50 to ₹2,00,000, with a few extreme values.
items_count	Ranges from 1 to 15.

(a) Identify the data-quality problems and describe the preprocessing pipeline (cleaning, transformation, normalization) you would follow, in the correct order.
(b) Explain why each step matters before analysis, and what would go wrong if it were skipped.
(c) Choose a normalization technique for order_amount and items_count, and justify your choice.

Order of steps: Inspect → Clean → Transform → Normalize → Verify

Step 1: Inspect (profile the data). Check data types, summary statistics, counts of missing values and duplicates, and the range of each column. This confirms the problems listed in the scenario and shows whether any are still hidden.

Step 2: Cleaning

Problem	Treatment
Duplicate orders	Remove exact duplicate records, keeping one copy of each.
Missing age	Fill with the median, which is not affected by extreme values.
Invalid age (0, 250)	Treat as errors and mark them as missing, then fill them with the median.
Inconsistent city	Trim spaces, use one letter case, and map aliases such as "Pnq" to "Pune".
Extreme order_amount values	Detect with the IQR rule (values beyond 1.5 × IQR from the quartiles), then verify them. Cap genuine extremes and correct or remove errors.

Step 3: Transformation

- Convert order_date from text to a proper date type using one standard format.
- Derive useful features from it, such as month, weekday and festival season.
- Encode gender and city as numbers (one-hot encoding), since models cannot use text labels.
- If order_amount is highly skewed, apply a log transformation to reduce the skew.

Step 4: Normalization. Bring numeric features to a comparable scale (see part c).

Step 5: Verify. Confirm that no missing values or duplicates remain, that ranges are valid, and that scaled columns fall in the expected range.

(b) Why each step matters (3 Marks)

Step	Why it matters	If skipped
Remove duplicates	Each order must be counted once.	Revenue and order counts are inflated, and the model over-weights repeated records.
Handle missing and invalid values	Most algorithms cannot process blanks, and impossible values distort statistics.	Errors, or a biased average (an age of 250 pulls the mean up).
Standardize categories	"Pune", "pune " and "PUNE" are the same city.	City-wise analysis splits one city into many groups and gives wrong totals.
Treat outliers	Extreme values dominate the mean and variance.	Skewed averages, and predictions pulled toward a few unusual orders.

Step	Why it matters	If skipped
Fix data types, extract features	Text dates cannot be sorted or grouped by month or weekday.	Seasonal and time-based trends cannot be found.
Encode categories	Models need numeric input.	Gender and city cannot be used, or the model assumes a false ranking among categories.
Normalize	It puts features on comparable scales (₹50–2,00,000 vs 1–15).	Distance-based and gradient-based models (KNN, k-means, neural networks) let the large-valued feature dominate.

In short, "garbage in, garbage out": an analysis can be no better than the data it is built on.

(c) Choice of normalization technique (2 Marks)

- **items_count → Min-max scaling**, which gives (x − min) / (max − min). Its range is small and bounded (1–15) with no extreme outliers, so scaling to [0, 1] is safe and keeps the shape of the distribution.
- **order_amount → Cap the outliers first, then use z-score standardization (or robust scaling).** Min-max is sensitive to outliers. One ₹2,00,000 order would squeeze almost all normal orders into a tiny range near 0. Z-score, (x − mean) / standard deviation, handles this better, and robust scaling, (x − median) / IQR, is better still when outliers are many.
- **Prevent data leakage.** Calculate the scaling values (min, max, mean) from the training data only, then apply the same values to the test data.

Technique	Formula	Best when
Min-max	(x − min) / (max − min)	Bounded data with no major outliers
Z-score	(x − mean) / std. dev.	Roughly bell-shaped data
Robust	(x − median) / IQR	Many outliers

b. an online store, has seen its revenue stay flat for two quarters although website traffic has grown. You are given 12 months of order data with these columns: order date, product category, price, quantity, discount, payment mode, city, device type and order status (Delivered / Cancelled / Returned).

(a) Outline the steps of the exploratory data analysis (EDA) you would follow.

(b) Name the descriptive statistics you would compute and the plots you would use to find patterns.

(c) State one likely insight and a recommended action.

(a) EDA steps (4 Marks)

1. **Define the goal:** find why revenue is flat although traffic is rising.
2. **Understand and check the data:** check data types and the time period, and look for missing values, duplicates and impossible values.
3. **Clean and prepare:** remove duplicates, fix order_date, and create revenue, month and weekday features.
4. **Analyze single variables:** study the distribution of order value, discount and quantity.
5. **Analyze relationships:** compare category, device, payment mode and city against revenue and cancellations.
6. **Analyze trends over time:** find the month where growth stopped.
7. **Summarize findings:** link each pattern to the business question.

(b) Statistics and plots (4 Marks)

Descriptive statistic	Purpose
Mean, median, std. deviation	Typical order value and spread
Min, max, IQR	Outliers and errors
Group-wise totals (by category, city, device)	Biggest and weakest contributors
Cancellation and return rate	Revenue lost
Correlation (discount vs quantity)	Whether discounts raise sales
Month-over-month growth	When the stagnation began

Plot	Reveals
Line chart (revenue by month)	Trend and when growth stopped
Histogram / box plot (order value)	Distribution, skew and outliers
Bar chart (revenue by category or city)	Top and weak segments
Stacked bar (order status by payment mode)	Where cancellations and returns are high
Scatter plot (discount vs quantity)	Whether discounts really increase sales

(c) Insight and action (2 Marks)

Insight: most traffic comes from mobile, but mobile orders have lower value and more cancellations, so the extra traffic is not turning into revenue.

Action: improve the mobile checkout and payment experience, and track mobile conversion every month.

Q6.

a. Explain how a real-time dashboard updates a chart every few seconds. Describe the data flow, list the steps involved and state any two benefits of such a dashboard.

1. Real-time dashboard (2 Marks)

A real-time dashboard is a web page whose charts update automatically every few seconds, without the user reloading the page. Examples are live sales, server load and stock prices.

2. Data flow (4 Marks)

Data source → Server → Browser → Chart updates

1. **Data source:** new data is created (a new order, a sensor reading).
2. **Server:** it collects the data and sends it to the browser in JSON format.
3. **Browser:** JavaScript receives the new data.
4. **Chart:** the chart library adds the new value and redraws the chart.

3. Steps to refresh the chart (3 Marks)

1. The page loads and draws the chart with the existing data.
2. A timer in JavaScript is set to run every few seconds (for example, every 5 seconds).
3. Each time the timer runs, the page asks the server for the latest data.
4. The new value is added to the chart's data, and the oldest value is removed so the chart stays readable.
5. The chart is redrawn, and the timer repeats.

4. Benefits (1 Mark)

- Users see current data and can act quickly, for example by fixing a delay.
- No manual page reload is needed.

b. Explain what an interactive multi-series line chart is. Describe how tooltips and a clickable legend work in such a chart, and state where it is useful.

1. Interactive multi-series line chart (2 Marks)

A multi-series line chart draws two or more lines on the same axes, so that several data sets can be compared over time (for example, monthly sales of three products). It is interactive when the user can hover, click or filter the chart and the chart responds, instead of showing a fixed picture. It is built with libraries such as **Chart.js** or **D3.js**.

2. Main parts of the chart (2 Marks)

Part	Purpose
X-axis and Y-axis	Show time (or categories) and values
Lines (series)	One coloured line per data set
Legend	Lists each series with its colour
Tooltip	Small box showing exact values on hover

3. How the interaction works (4 Marks)

Tooltip:

1. The user moves the mouse over a point or line.
2. The chart detects the nearest data point (a mouse event).
3. A small box appears showing the series name, the date and the exact value.
4. When the mouse moves away, the box disappears.

Legend toggle:

1. Each legend item is clickable.
2. When the user clicks an item, that series is hidden (its line is removed from the chart).
3. Clicking again shows the series.
4. The chart redraws, and the axis scale can adjust to the visible lines.

In **Chart.js**, tooltips and the legend toggle are built in and need only to be enabled. In **D3.js**, they are created by attaching mouse and click events to the chart elements.

4. Uses and benefits (2 Marks)

- **Comparison:** the user can focus on one or two series by hiding the others, so a crowded chart stays readable.
- **Exact values:** tooltips give precise numbers without cluttering the chart with labels.
- **Typical uses:** sales by product, website traffic by source, stock prices, temperature by city.

Marking scheme: definition 2, parts 2, interaction 4, uses 2.