1. **Write a simple "Hello World" program in two different programming languages of your choice. Compare the structure and syntax.**

   python

   ```
   print("Hello, World!");
   ```

   Java

   ```
   public class HelloWorld {
     public static void main(String[] args) {
       System.out.println("Hello, World!");
     }
   }
   ```

2. **Research and create a diagram of how data is transmitted from a client to a server over the internet.**

   Step-by-Step Data Journey

1. **Application Layer (Client)**
   - The user's app (web browser, email client) initiates a request—e.g., "GET /index.html" via HTTP(S).
2. **Transport Layer (TCP/UDP)**
   - TCP divides data into segments, establishes a connection via a 3-way handshake, and ensures ordered, error-free delivery.
   - Awaiting acknowledgment (ACK) from the server ensures reliability.
   - Alternatively, UDP may send packets without guaranteed delivery (e.g., for VoIP).
3. **Network Layer (IP)**
   - Segments are encapsulated into IP packets with source/destination IPs.
   - Routers forward packets across networks toward the server's IP. No guarantees on order or delivery
4. **Data Link & Physical Layers**
   - IP packets are wrapped in link-specific frames (e.g., Ethernet/Wi-Fi), including MAC addresses and error checks.
   - Frames are transmitted as electrical or wireless signals across physical media.
5. **Router/Intermediate Hops**
   - Each router strips the link-layer header, reads IP to forward, re-wraps in new link-layer frame, and sends onward.
6. **Server Reception (Reverse Flow)**
   - The server's NIC receives data, frames pass up to IP and transport layers.
   - TCP reassembles segments, checks order/integrity, and delivers the complete message to the server app (like a web server).
7. **Server Response**
   - The server processes the request, prepares a response (e.g., HTML page), and sends it back following the same multi-layer path in reverse.

3. Design a simple HTTP client-server communication in any language

Server (Python HTTP Server)

```
from http.server import BaseHTTPRequestHandler, HTTPServer


class SimpleHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)  # HTTP status code
        self.send_header("Content-type", "text/plain")
        self.end_headers()
        self.wfile.write(b"Hello from the server!")


def run(server_class=HTTPServer, handler_class=SimpleHandler, port=8080):
    server_address = ("", port)
    httpd = server_class(server_address, handler_class)
    print(f"Server running on port {port}...")
    httpd.serve_forever()


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

4. Research different types of internet connections (e.g., broadband, fiber, satellite) and list their pros and cons

1. Broadband (DSL and Cable)

DSL (Digital Subscriber Line) uses telephone lines; **Cable** uses coaxial TV cables.

Pros:

- **Widely Available**: Especially in urban and suburban areas.
- **Affordable**: Generally cheaper than fiber or satellite.
- **Stable Connection**: Good for general browsing, streaming, and video calls.

Cons:

- **Slower Speeds**: Compared to fiber.
- **Distance-Sensitive (DSL)**: Speed drops the farther you are from the provider's central office.
- **Shared Bandwidth (Cable)**: Speed may slow during peak hours.

---

2. Fiber Optic

Transmits data as light through thin glass or plastic fibers.

Pros:

- **Very High Speeds**: Often 1 Gbps or higher.
- **Low Latency**: Excellent for gaming and video conferencing.
- **Reliable Connection**: Less affected by weather or electrical interference.

Cons:

- **Limited Availability**: Mostly in cities or developed areas.
- **Higher Cost**: Installation and monthly fees can be more expensive.
- **Long Installation Times**: Especially in non-wired areas.

---

3. Satellite

Provides internet via communication satellites; ideal for remote or rural areas.

Pros:

- **Accessible in Remote Locations**: Doesn't rely on ground infrastructure.
- **Easy Setup**: No need for cables or wires.

Cons:

- **High Latency**: Due to signal travel distance to/from satellites.
- **Weather-Dependent**: Performance can degrade during storms or heavy clouds.
- **Expensive Data Plans**: Limited data with high costs.

---

4. Fixed Wireless

Connects homes to the internet using radio signals from nearby towers.

Pros:

- **Quick Deployment**: Good for areas without wired connections.
- **Reasonable Speeds**: Better than satellite in some rural cases.

Cons:

- **Line-of-Sight Required**: Obstructions can affect performance.
- **Weather Interference**: Can impact signal strength.

---

5. Mobile (3G, 4G, 5G)

Internet via cellular networks, accessed through phones or mobile hotspots.

Pros:

- **Portable and Convenient**: Works wherever there's coverage.
- **Fast (especially 5G)**: High speeds with low latency possible.

Cons:

- **Coverage Gaps**: Speed and signal vary by location.
- **Data Caps**: Many plans have limits or throttling.
- **Battery Drain**: On mobile devices.

5. Simulate HTTP and FTPrequests using command line tools (e.g., curl)

Task	Command Example
HTTP GET	curl http://example.com
HTTP POST	curl -X POST -d "a=1" http://example.com
Download file via HTTP	curl -O http://example.com/file.zip
Download file via FTP	curl -u user:pass ftp://ftp.site.com/file.txt -O

Task	Command Example
Upload file via FTP	curl -T file.txt -u user:pass ftp://ftp.site.com/

6. **Identify and explain three common application security vulnerabilities. Suggest possible solutions.**

Vulnerability	Risk	Solutions
SQL Injection	Unauthorized DB access or data corruption	Use prepared statements, validate input
XSS	Script execution in users' browsers	Sanitize input, escape output, use CSP
CSRF	Unwanted actions via authenticated sessions	CSRF tokens, SameSite cookies, re-authentication

7. **Identify and classify 5 applications you use daily as either system software or application software**

Application	Type	Explanation
Google Chrome	Application Software	A web browser used to access websites and online apps.
Microsoft Word	Application Software	A word processor used for creating and editing documents.
Windows 10/11 OS	System Software	An operating system that manages hardware and runs other software.
File Explorer	System Software	A built-in file management tool in Windows OS.
Spotify	Application Software	A music streaming app for playing audio content from the internet.

8. **Design a basic three-tier software architecture diagram for a web application.**

```
┌─────────────────────┐
│   Client Tier      │
│ (Web Browser / App) │
└─────────────────────┘

     │
     ▼

┌─────────────────────┐
│ Presentation Tier  │
│ (Web Server)       │
```

```
| - HTML/CSS/JS        |
| - Frontend Logic     |
└──────────────────────┘
          |
          ▼
┌──────────────────────┐
|   Application Tier   |
| (Backend Server)     |
| - Business Logic     |
| - API Endpoints      |
└──────────────────────┘
          |
          ▼
┌──────────────────────┐
|     Data Tier        |
| (Database Server)    |
| - MySQL, PostgreSQL  |
| - MongoDB, etc.      |
└──────────────────────┘
```

9. **Create a case study on the functionality of the presentation, business logic, and data access layers of a given software system.**

1. Presentation Layer (User Interface Layer)
Function:
This is the **front end** that interacts with users via a web browser or mobile app.
Features:

- Displays available books.
- Allows search/filtering.
- Shows the shopping cart.
- Handles user inputs (e.g., login, payment info).

Technologies Used:

- **HTML/CSS/JavaScript** (UI layout and interaction)
- **React or Angular** (for SPA functionality)
- **REST API calls** to backend

Example Action:
A user clicks **"Add to Cart"** on a book → This sends an HTTP POST request to the application layer.

---

2. Business Logic Layer (Application Layer)
Function:
Processes and enforces rules and workflows of the system (the "brains").
Features:

- Validates user login credentials.
- Calculates totals and tax.
- Checks inventory availability.
- Processes orders and triggers payment.

Technologies Used:

- **Node.js, Java, Python (Django/Flask), .NET**
- **REST API** controllers and services
- **Authentication/Authorization**

Example Action:

Receives "Add to Cart" request → Verifies book availability → Adds book to user's cart object → Returns updated cart.

---

3. Data Access Layer (Database Layer)

Function:

Responsible for interacting with the database and handling CRUD (Create, Read, Update, Delete) operations.

Features:

- Fetches book catalog.
- Stores user data and orders.
- Updates stock inventory.
- Ensures data integrity and consistency.

Technologies Used:

- **Relational DB**: MySQL, PostgreSQL
- **NoSQL DB**: MongoDB (for catalog/search)
- **ORMs**: Sequelize, Hibernate, Entity Framework

Example Action:

On "Add to Cart": Updates the user's cart in the database and reduces book inventory count by 1.

10. Explore different types of software environments (development, testing, production). Set up a basic environment in a virtual machine

Environment	**Use Case**	**Tools / Setup Example**
Development	Write and debug code	Node.js, local DB, text editors
Testing	Automated/manual testing	CI tools, test suites, mocked data
Production	Live user environment	Hardened servers, monitoring, backups

Step 2: Install Development Tools

```
sudo apt update
sudo apt install -y git curl vim build-essential
```

Step 3: Set Up a Simple Web Development Stack

Install Node.js and Nginx

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt install -y nodejs nginx
```

Create a basic web app:

```
mkdir myapp && cd myapp
npm init -y
npm install express
```

Create index.js:

```
const express = require('express');
const app = express();
app.get('/', (req, res) => res.send('Hello from VM Dev Environment!'));
app.listen(3000);
```

Run it:

```
node index.js
```

11. Write and upload your first source code file to Github

Create a file:

```
mkdir first-github-project

cd first-github-project

nano hello.py
```

Paste this Python code into hello.py:

```
# hello.py

print("Hello, GitHub! This is my first source code upload.")
```

Save and exit (in Nano: CTRL + O, then ENTER, then CTRL + X).

Step 2: Initialize Git in the Folder

bash

CopyEdit

```
git init

git add hello.py

git commit -m "Initial commit with hello.py"
```

Step 3: Create a Repository on GitHub

1. Go to https://github.com
2. Log in to your account.
3. Click **New Repository**
4. Name it: first-github-project
5. Leave it public (or private), **do not initialize with README**
6. Click **Create repository**

Step 4: Connect Your Local Repo to GitHub

You'll be given a remote URL like:

https://github.com/your-username/first-github-project.git

Add it to your Git config:

```
git remote add origin https://github.com/your-username/first-github-project.git
```

Step 5: Push Your Code to GitHub

```
git branch -M main

git push -u origin main
```

Final Result

Your file hello.py is now live on GitHub!

You can view it by going to:

https://github.com/your-username/first-github-project/blob/main/hello.py

12 . Create a Github repository and document how to commit and push code changes

Part 1: Create a GitHub Repository

1. **Go to:** https://github.com
2. **Log in** to your account.
3. **Click** the **+** icon (top right) → **New repository**
4. Fill in:
   - **Repository name**: e.g., my-first-repo
   - **Description**: (Optional)
   - Choose **Public** or **Private**
   - **Do not** initialize with a README (optional if you want to push from local)
5. Click **Create repository**

Part 2: Set Up Git Locally

1. Open terminal and create a folder

```
mkdir my-first-repo

cd my-first-repo
```

2. Create a file

```
echo "print('Hello, GitHub!')" > hello.py
```

3. Initialize Git and commit the file

```
git init

git add hello.py

git commit -m "Initial commit: Added hello.py"
```

Part 3: Connect to GitHub and Push

1. Add remote origin (from your GitHub page)

```
git remote add origin https://github.com/YOUR_USERNAME/my-first-repo.git
```

2. Push code to GitHub

```
git branch -M main   # Rename the default branch to 'main'

git push -u origin main
```

Git will ask for your credentials if you’re not using SSH or token-based authentication.

� Part 4: Commit and Push Future Changes

1. Edit your file

```
echo "# This is a comment" >> hello.py
```

2. Stage and commit

```
git add hello.py

git commit -m "Added a comment line to hello.py"
```

3. Push changes

```
git push origin main
```

13 . Create a student account on Github and collaborate on a small project with a classmate

Create a Student GitHub Account

1. **Go to** GitHub Sign Up
2. **Fill out the form** with a valid email address, username, and password.
3. **Verify your email** by clicking the link GitHub sends you.

4. (Optional) If you're a student, apply for the GitHub Student Developer Pack — it offers free access to useful developer tools.

Step 2: Create a Repository for Your Project

1. After logging in, click the **+** icon (top right) → **New repository**.
2. Name your repo, e.g., class-project.
3. Choose **Public** or **Private**.
4. Add a README if you like.
5. Click **Create repository**.

Step 3: Invite Your Classmate to Collaborate

1. Go to your repository's page.
2. Click **Settings** → **Manage access**.
3. Click **Invite a collaborator**.
4. Enter your classmate's GitHub username or email.
5. Send the invite — your classmate will accept via email or GitHub notifications.

Step 4: Clone the Repository Locally

Both of you should:

```
git clone https://github.com/your-username/class-project.git

cd class-project
```

Step 5: Start Collaborating

Common workflow:

1. Create or edit files.
2. Stage changes:

```
git add .
```

3. Commit changes:

```
  git commit -m "Add feature or fix bug"
```

4. Pull the latest changes before pushing (to avoid conflicts):

```
 git pull origin main
```

5. Push your changes:

```
git push origin main
```

Step 6: Use Branches (Recommended for Collaboration)

1. Create a branch for your work:

```
git checkout -b feature-branch
```

2. Work on your branch, commit changes.
3. Push your branch:

```
git push origin feature-branch
```

4. Open a **Pull Request** on GitHub to merge your changes into main.
5. Your classmate can review and merge.

Tips for Smooth Collaboration

- Communicate frequently about changes.
- Use issues and project boards on GitHub for task tracking.
- Resolve merge conflicts carefully.

14. Create a list of software you use regularly and classify them into the following categories: system, application, and utility software.

Software	Category	Description
Windows 10/11	System Software	Operating system managing hardware and resources
macOS	System Software	Apple’s operating system
Google Chrome	Application Software	Web browser for accessing the internet
Microsoft Word	Application Software	Word processing software
Spotify	Application Software	Music streaming application
Adobe Photoshop	Application Software	Image editing software

Software	Category	Description
File Explorer	Utility Software	File management tool on Windows
Task Manager	Utility Software	System monitoring and process management
CCleaner	Utility Software	System cleaning and optimization tool
Antivirus (e.g., Windows Defender)	Utility Software	Protects against malware and viruses

15. Follow a GIT tutorial to practice cloning, branching, and merging repositories.

Step 1: Clone a Repository

You can either clone an existing public repository or create a new one.

Option A: Clone an existing public repository

```
git clone https://github.com/octocat/Hello-World.git

cd Hello-World
```

Option B: Create and clone your own

1. Go to GitHub → Create a new repository (without README).
2. Then run:

```
git clone https://github.com/YOUR_USERNAME/YOUR_REPO_NAME.git

cd YOUR_REPO_NAME
```

Step 2: Create a New Branch

```
git checkout -b feature/my-new-feature
```

This creates and switches to a new branch called feature/my-new-feature.

Step 3: Make Changes and Commit

1. Edit or create a file:

```
echo "Hello Git World!" > hello.txt
```

2. Stage and commit your changes:

```
git add hello.txt

git commit -m "Add hello.txt with a welcome message"
```

Step 4: Switch Back to Main (or Master)

```
git checkout main
```

Step 5: Merge the Feature Branch into Main

```
git merge feature/my-new-feature
```

If there are no conflicts, the changes will merge automatically.

Step 6: Push Changes to Remote (if using GitHub)

```
git push origin main
```

(Optional) Step 7: Delete the Feature Branch

```
git branch -d feature/my-new-feature
```

16. Write a report on the various types of application software and how they improve productivity

Types of Application Software

2.1 Word Processing Software

Examples: Microsoft Word, Google Docs, LibreOffice Writer
Functions:

- Creating, editing, formatting text documents
- Spell check, grammar suggestions, templates

Productivity Benefits:

- Streamlines report writing and documentation
- Facilitates collaboration through cloud-based tools
- Reduces time spent on formatting and proofreading

2.2 Spreadsheet Software

Examples: Microsoft Excel, Google Sheets, Apple Numbers
Functions:

- Organizing data in tables
- Performing calculations, using formulas and charts
- Automating repetitive tasks with macros

Productivity Benefits:

- Enhances decision-making with data analysis
- Saves time through automation of calculations
- Useful for budgeting, scheduling, and forecasting

2.3 Presentation Software

Examples: Microsoft PowerPoint, Google Slides, Keynote
Functions:

- Creating slide decks for meetings, lectures, or proposals
- Incorporating multimedia elements (images, videos, graphs)
- Presenting information in a visual format

Productivity Benefits:

- Improves communication and idea sharing
- Speeds up the creation of professional presentations
- Encourages team collaboration and feedback

2.4 Database Management Software (DBMS)

Examples: Microsoft Access, MySQL, Oracle
Functions:

- Creating and managing structured data storage
- Running queries, reports, and automations
- Ensuring data integrity and security

Productivity Benefits:

- Increases efficiency in storing and retrieving large datasets
- Automates report generation and data validation
- Supports data-driven decision-making

2.5 Communication Software

Examples: Zoom, Microsoft Teams, Slack, Skype
Functions:

- Video conferencing, instant messaging, email
- File sharing and collaborative tools

- Integration with calendars and productivity suites

Productivity Benefits:

- Enhances remote collaboration
- Reduces travel and meeting time
- Supports real-time communication

2.6 Project Management Software

Examples: Trello, Asana, Monday.com, Microsoft Project
Functions:

- Task tracking, scheduling, resource allocation
- Team collaboration and progress monitoring
- Notifications and time management tools

Productivity Benefits:

- Organizes tasks and responsibilities
- Prevents missed deadlines and duplicated efforts
- Improves team coordination and accountability

2.7 Graphic Design and Multimedia Software

Examples: Adobe Photoshop, Canva, CorelDRAW, Final Cut Pro
Functions:

- Image editing, video production, content creation
- Design templates and multimedia integration
- Export for print, web, or social media

Productivity Benefits:

- Speeds up creative production
- Enhances quality and appeal of visual content
- Simplifies complex design tasks with templates and automation

---

2.8 Web Browsers and Internet Tools

Examples: Google Chrome, Mozilla Firefox, Safari
Functions:

- Accessing and navigating the internet

- Running web applications and extensions
- Bookmarking and managing online resources

Productivity Benefits:

- Provides access to cloud tools and online resources
- Supports research and online collaboration
- Enhances multitasking with tabs and browser extensions

17. Create a flowchart representing the Software Development Life Cycle (SDLC).

```
+--------------------------------------+
| 1. Requirement Gathering & Analysis    |
+--------------------------------------+
               |
               v
+--------------------------------------+
| 2. System Design                    |
+--------------------------------------+
               |
               v
+--------------------------------------+
| 3. Implementation (Coding)          |
+--------------------------------------+
               |
               v
+--------------------------------------+
| 4. Testing                         |
+--------------------------------------+
               |
               v
+--------------------------------------+
| 5. Deployment                       |
+--------------------------------------+
          |
          v
+-------------------
```

18. Write a requirement specification for a simple library management system

Software Requirement Specification (SRS)

Library Management System

---

1. Introduction

1.1 Purpose

The purpose of this document is to define the requirements for a simple Library Management System (LMS) that will help librarians manage book records, user accounts, and borrowing transactions efficiently.

1.2 Scope

This system will be used by library staff and members to manage book inventory, track book issues/returns, and maintain records of registered members. It will be a web-based application suitable for small to medium-sized libraries.

---

2. Overall Description

2.1 Product Perspective

This is a standalone system that does not require integration with external applications. It will have a centralized database and a web-based user interface.

2.2 User Classes and Characteristics

- **Admin**: Full access to manage books, users, and transactions.
- **Librarian**: Can issue/return books, view reports, and manage book inventory.
- **Member**: Can search books, view availability, and request books.

2.3 Operating Environment

- Web Browser (Chrome, Firefox, Edge)
- Server: Apache/Nginx
- Backend: Python/Django or Node.js
- Database: MySQL or PostgreSQL

2.4 Design and Implementation Constraints

- Must support basic CRUD operations for books and users
- System must be responsive for desktop and mobile use

---

3. Functional Requirements

3.1 User Management

- Admin can add/edit/delete librarian and member accounts
- Members can register/login with credentials

3.2 Book Management

- Add/edit/delete books (title, author, ISBN, genre, quantity)
- Search books by title, author, or genre
- Display availability status of books

3.3 Borrowing and Returning

- Librarian can issue a book to a member
- System records the issue date and due date
- Members can return books; system calculates fines if overdue

3.4 Notifications

- Send reminders for due/overdue books via email (optional)
- Notify admins about low stock or frequent requests

3.5 Reporting

- Generate reports on issued books, overdue items, and inventory
- View member borrowing history

---

4. Non-Functional Requirements

4.1 Performance

- System should support up to 100 concurrent users
- Average response time should be under 2 seconds

4.2 Security

- Authentication required for all user roles
- Role-based access control
- Passwords stored using hashing algorithms

4.3 Usability

- Intuitive and user-friendly interface
- Minimal training required for librarians

4.4 Availability

- System should be available 99% of the time
- Regular backups of the database should be maintained

---

5. Future Enhancements

- Barcode scanning support
- Integration with digital book repositories (eBooks)
- Mobile app version

19. Perform a functional analysis for an online shopping system

Functional Analysis: Online Shopping System

---

1. Objective

To analyze the core functions of an online shopping system that allows customers to browse, purchase, and manage orders for products via a web or mobile interface. The system should also provide administrative capabilities for managing inventory, users, and transactions.

---

2. Key Stakeholders and Users

- **Customers**: Individuals who browse and purchase products.
- **Admins**: Manage products, categories, orders, and user accounts.
- **Sellers (optional)**: Third-party vendors who can manage their own listings.
- **System**: Backend services, databases, and payment integrations.

---

3. Core Functional Areas

3.1 User Management

- **Register new users** (with email/password or social login)
- **Login/logout** securely
- **Manage user profile** (address, phone, preferences)
- **Password reset/recovery**

3.2 Product Management

- **Browse product catalog** by category, price, or popularity
- **Search** for products using keywords or filters

- **View product details** (description, reviews, price, availability)
- **Add/edit/delete products** (admin or seller role)

3.3 Shopping Cart & Wishlist

- **Add products to cart**
- **Update cart quantities**
- **Remove items from cart**
- **Save items for later (wishlist)**

3.4 Order Processing

- **Place an order** with selected products
- **Select shipping options**
- **Make online payments** via integrated gateways
- **Receive order confirmation** via email/SMS
- **View order history and status**

3.5 Payment Integration

- **Secure payment processing** via credit card, PayPal, or mobile wallets
- **Generate invoices and receipts**
- **Handle failed or cancelled transactions**

3.6 Inventory Management

- **Track stock levels**
- **Display product availability**
- **Automatically update stock after orders**
- **Notify admin when stock is low**

3.7 Shipping & Delivery

- **Manage shipping addresses**
- **Track shipment status**
- **Send delivery notifications**
- **Support return and refund requests**

3.8 Review & Rating System

- **Allow customers to rate products**
- **Write and edit reviews**
- **Display average ratings per product**

3.9 Admin Panel Functions

- **Dashboard** showing KPIs (orders, revenue, stock levels)
- **User management**
- **Product and category management**
- **Order and payment tracking**
- **Generate reports** (sales, user activity, popular products)

20. Design a basic system architecture for a food delivery app.

Food Delivery App – System Architecture Overview

1. Architecture Type:

Microservices-based layered architecture with RESTful APIs and optional real-time components.

---

2. Key Components

A. Client Layer (Front-End)

- **Mobile App (iOS/Android)** for Customers, Delivery Agents
- **Web App** for Admins and Restaurants

Responsibilities:

- UI/UX for browsing restaurants, ordering food
- Push notifications (order status updates)
- Location services (map, delivery tracking)

B. Application Layer (Back-End / API Gateway)

Acts as the entry point for all requests.

- **API Gateway**
  - Handles routing, authentication, rate limiting
  - Forwards requests to appropriate microservices

C. Microservices Layer

1. User Service

- Handles registration, login, user profiles
- Roles: Customer, Delivery Agent, Restaurant Owner, Admin

2. Restaurant Service

- Manage menus, restaurant details, availability
- Integration with POS systems (optional)

3. Order Service

- Handles order creation, status updates, and history
- Manages order workflow (placed → accepted → delivered)

4. Payment Service

- Integration with payment gateways (Stripe, PayPal)
- Manages transactions, refunds, and invoicing

5. Delivery Service

- Assigns delivery agents
- Tracks real-time location of orders

6. Notification Service

- Sends SMS, email, or push notifications
- Order confirmations, delivery updates, promotions

7. Review & Rating Service

- Collects and displays customer feedback
- Ratings for restaurants and delivery agents

8. Admin Dashboard Service

- Analytics, reports, user management
- Promotion and discount management

D. Data Storage Layer

- **Relational DB (e.g., PostgreSQL, MySQL)** for structured data (users, orders, restaurants)
- **NoSQL DB (e.g., MongoDB)** for flexible content (menus, user preferences)
- **Redis or Memcached** for caching frequently accessed data
- **Cloud Storage (e.g., AWS S3)** for storing images (food photos, profile pictures)

E. External Integrations

- **Payment Gateways** (Stripe, Razorpay, etc.)
- **Map APIs** (Google Maps, Mapbox) for location and routing

- **SMS/Email services** (Twilio, SendGrid)
- **Analytics tools** (Google Analytics, Mixpanel)

9. Security and Authentication

- **OAuth 2.0 / JWT** for secure token-based authentication
- **HTTPS encryption** for all communications
- **Role-based access control** (RBAC)

---

10. Deployment

- **Containerization** using Docker
- **Orchestration** using Kubernetes (for scalability)
- **CI/CD** pipelines for continuous delivery (GitHub Actions, Jenkins)
- **Cloud Hosting**: AWS, Azure, or Google Cloud

21. Develop test cases for a simple calculator program.

Test Case ID	Description	Input	Expected Output	Remarks
TC01	Add two positive integers	3 + 5	8	Basic addition
TC02	Add a positive and negative number	7 + (-2)	5	Mixed sign addition
TC03	Subtract two numbers	10 - 4	6	Basic subtraction
TC04	Multiply two numbers	6 × 3	18	Basic multiplication
TC05	Divide two numbers	20 ÷ 4	5	Basic division
TC06	Division by zero	8 ÷ 0	Error/Exception	Should handle gracefully
TC07	Add two floating-point numbers	2.5 + 3.1	5.6	Float addition
TC08	Subtract decimals	10.5 - 2.25	8.25	Float subtraction
TC09	Multiply negative numbers	-2 × -3	6	Negative multiplication
TC10	Divide negative by positive	-9 ÷ 3	-3	Sign handling

22. Document a real-world case where a software application required critical maintenance.

1. Background

Boeing's 737 MAX aircraft, a modernized version of the 737 series, relied on **software-driven flight controls**, including a system called **MCAS** (Maneuvering Characteristics Augmentation System). MCAS was designed to automatically push the aircraft's nose down in specific flight conditions to prevent stalling.

---

2. The Problem

Two fatal crashes — **Lion Air Flight 610 (Oct 2018)** and **Ethiopian Airlines Flight 302 (Mar 2019)** — revealed a **critical software flaw in the MCAS system**:

- MCAS relied on a **single angle-of-attack (AoA) sensor**.
- If this sensor gave incorrect data, the system could repeatedly push the nose of the aircraft down.
- Pilots were **not fully informed about MCAS** or trained on how to override it.
- The software **overrode manual pilot input** in critical situations.

---

3. Maintenance Response

After grounding all 737 MAX aircraft globally, Boeing was forced into **emergency and critical software maintenance**, involving:

Key Maintenance Actions

- **Redesign of the MCAS system** to use input from **two AoA sensors** instead of one.
- **Limiting MCAS authority**, so it could not repeatedly override pilot input.
- **Enhancing pilot alerts** and making MCAS behavior more transparent.
- **Developing simulator-based training** for pilots to understand the MCAS system.

---

4. Validation and Re-Certification

- Boeing's software underwent **intensive FAA testing and validation**.
- The updated MCAS system was **recertified** in **late 2020**, nearly two years after the crashes.
- Independent international aviation authorities also reviewed and accepted the fixes.

---

5. Outcome

- 737 MAX aircraft were **gradually reintroduced into service** starting in late 2020.

- The case became a landmark example of the **importance of software transparency, redundancy, and pilot training** in safety-critical systems.
- It emphasized the need for **robust maintenance protocols** and **regulatory oversight** when software impacts human lives.

---

6. Lessons Learned

- **Single-point sensor dependency is dangerous**.
- **Software updates must be communicated to end-users** (in this case, pilots).
- **Critical systems require constant validation**, even after deployment.
- **Software maintenance is not just fixing bugs**—it's about ensuring safety, compliance, and operational continuity.

23. Create a DFD for a hospital management system.

Hospital Management System – Level 1 DFD

External Entities:

- **Patient**
- **Doctor**
- **Admin**

Processes and Data Flows:

1. Patient Registration

- **Input:** Patient → Personal Information
- **Output:** Stores data in → Patient Records
- **Data Flow:** Personal Info → [Patient Registration] → Patient Records

2. Appointment Scheduling

- **Input:** Patient selects time & doctor
- **Output:** Appointment confirmation
- **Data Flow:** Appointment Request → [Appointment Scheduling] → Appointment Records
- [Appointment Scheduling] ← Doctor Availability ← Doctor Schedule

3. Medical Records Management

- **Input:** Doctor inputs diagnosis/treatment
- **Output:** Updates to medical history
- **Data Flow:** [Doctor] → Diagnosis → [Medical Records] → Patient Medical History

4. Billing

- **Input:** Treatment info + medication cost
- **Output:** Bill generated
- **Data Flow:** [Billing] ← Service Data + Pharmacy Charges ← Patient Medical History, Pharmacy
- [Billing] → Bill → Patient

5. Pharmacy

- **Input:** Prescription from doctor
- **Output:** Medicine issued and inventory updated
- **Data Flow:** Prescription → [Pharmacy] → Medicine Inventory

---

Data Stores:

- Patient Records
- Appointment Records
- Patient Medical History
- Doctor Schedule
- Medicine Inventory
- Billing Records

24. Build a simple desktop calculator application using a GUI library

I'll create a simple desktop calculator using Python and the Tkinter GUI library. This calculator will handle basic operations: addition, subtraction, multiplication, and division.

Here’s the complete code:

```
import tkinter as tk


# Create main window

root = tk.Tk()

root.title("Simple Calculator")


# Entry widget to display expressions and results

entry = tk.Entry(root, width=16, font=('Arial', 24), borderwidth=2, relief="ridge", justify='right')
```

```
entry.grid(row=0, column=0, columnspan=4, padx=10, pady=10)

# Function to add button clicks to the entry
def click_button(value):
    current = entry.get()
    entry.delete(0, tk.END)
    entry.insert(0, current + value)

# Function to clear the entry
def clear_entry():
    entry.delete(0, tk.END)

# Function to calculate the expression
def calculate():
    expression = entry.get()
    try:
        result = eval(expression)
        entry.delete(0, tk.END)
        entry.insert(0, str(result))
    except Exception:
        entry.delete(0, tk.END)
        entry.insert(0, "Error")

# Define buttons
buttons = [
    ('7', 1, 0), ('8', 1, 1), ('9', 1, 2), ('/', 1, 3),
    ('4', 2, 0), ('5', 2, 1), ('6', 2, 2), ('*', 2, 3),
    ('1', 3, 0), ('2', 3, 1), ('3', 3, 2), ('-', 3, 3),
    ('0', 4, 0), ('.', 4, 1), ('C', 4, 2), ('+', 4, 3),
    ('=', 5, 0, 4)  # Spanning 4 columns
]
```

```
# Create and place buttons
for (text, row, col, colspan) in [(*btn, 1) if len(btn) == 3 else btn for btn in buttons]:
    if text == 'C':
        action = clear_entry
    elif text == '=':
        action = calculate
    else:
        action = lambda x=text: click_button(x)
    btn = tk.Button(root, text=text, width=5, height=2, font=('Arial', 18), command=action)
    btn.grid(row=row, column=col, columnspan=colspan, sticky="nsew", padx=3, pady=3)

# Configure grid weights to make buttons expand
for i in range(6):
    root.grid_rowconfigure(i, weight=1)
for j in range(4):
    root.grid_columnconfigure(j, weight=1)

# Run the application
root.mainloop()
```

25. Draw a flowchart representing the logic of a basic online registration system

Flowchart Logic for Online Registration System

1. **Start**
2. **Display Registration Form**
3. **User Inputs Data** (Name, Email, Password, etc.)
4. **Validate Input Data**
    - If **Invalid**, show error message → Go back to step 2
    - If **Valid**, proceed to next step
5. **Check if Email Already Registered**
    - If **Yes**, show “Email already exists” message → Go back to step 2

- If **No**, proceed

6. **Save User Data to Database**
7. **Send Confirmation Email** (optional)
8. **Show Registration Success Message**
9. **End**

Symbol guide:

- **Oval:** Start/End
- **Parallelogram:** Input/Output (display form, show messages)
- **Rectangle:** Process (validate data, save to database)
- **Diamond:** Decision (valid input? email exists?)