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

C languae

**#include <stdio.h>**

**int main() {**

**printf("Hello World");**

**return 0;**

**}**

**C++**

**#include <iostream>**

**int main() {**

**std:cout << "Hello World";**

**return 0;**

**}**

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

A computer and a cloud

AI-generated content may be incorrect.

[Client App]

↓ (Application Layer – HTTP Request)

[Presentation/Session]

↓ (TCP: 3-way handshake, segmentation)

[Transport Layer]

↓ (TCP segments → encapsulated in IP packets)

[Network Layer]

↓ (IP routing → internet backbone, routers)

[Data Link / Physical — medium: Ethernet/Wi‑Fi/cables]

→ travels across multiple hops →

reverses back through:

[Data Link / Network / Transport / Session / Presentation]

↑

[Server App]

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

Server (server.js)

**const http = require('http');**

**const server = http.createServer((req, res) => {**

**console.log(`📥 Request: ${req.method} ${req.url}`);**

**res.writeHead(200, { 'Content-Type': 'text/plain' });**

**res.end('Hello from Node.js server!');**

**});**

**const PORT = 3000;**

**server.listen(PORT, () => {**

**console.log(`🚀 Server running at http://localhost:${PORT}/`);**

});

Client (client.js)

**const http = require('http');**

**const options = {**

**hostname: 'localhost',**

**port: 3000,**

**path: '/',**

**method: 'GET'**

**};**

**const req = http.request(options, res => {**

**console.log(`📤 Server responded with status ${res.statusCode}`);**

**res.setEncoding('utf8');**

**res.on('data', chunk => console.log('📦 Body:', chunk));**

**});**

**req.on('error', e => console.error(`❗ Problem: ${e.message}`));**

**req.end();**

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

**Satellite internet connection**

**Pros:**

* Available in rural or remote areas
* No cables needed
* Quick Setup & Infrastructure-Free
* Moderate to High Speeds

**Cons:**

* High Latency (Ping Delay)
* Weather and Obstruction Sensitivity
* Cost: Equipment & Service
* Environmental Concerns

|  |
| --- |
|  |

|  |
| --- |
|  |

**A diagram of a satellite network

AI-generated content may be incorrect.**

**Broadband internet Connection**

**pros**

* High-Speed & Always-On
* Multiple Devices & High Bandwidth
* Reliability & Stability
* Widespread Availability

**cons**

* Higher Cost
* Limited Rural Reach
* Shared Bandwidth Issues
* Outages & Infrastructure Dependency



* Simulate HTTP and FTP requests using command line tools (e.g., curl)

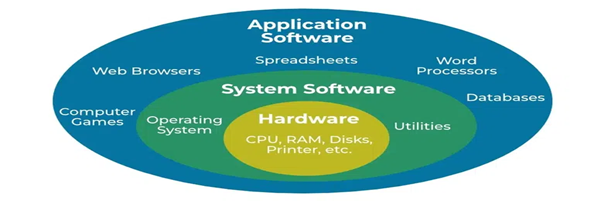
| **Command** | **Purpose** | **Description** |
| --- | --- | --- |
| **curl URL** | GET | Fetches the resource at URL via HTTP GET—default method . |
| **curl -I URL** | HEAD (headers only) | Retrieves only response headers (no body) using -I/--head . |
| **curl -X POST -d "key=val" URL** | POST form | Sends an HTTP POST request with form data; -d implies POST . |
| **curl -H "Header: Value" URL** | Custom HTTP header | Adds or overrides request headers using -H/--header . |
| **curl -v URL** | Verbose/debug info | Enables -v to display detailed connection info, including request/response details . |
| **curl --user user:pass ftp://server/path** | FTP download | Downloads a file via FTP with basic authentication; credentials via -u . |
| **curl -T file ftp://...** | FTP upload | Uploads a local file to FTP server using -T/--upload-file . |
| **curl -X "COMMAND file" ftp://...** | Custom FTP command | Sends raw FTP commands with -X (e.g., DELE filename) . |

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

| **Vulnerability** | **Cause** | **Risk** | **Mitigation Strategies** |
| --- | --- | --- | --- |
| **Injection** | Untrusted input passed as code | Unauthorized data access, data manipulation | Parameterized queries, allow-list validation, least privilege database roles |
| **XSS** | Unsanitized output in HTML | Cookie theft, session hijacking, client-side attacks | Output encoding, CSP, contextual escaping |
| **CSRF** | Trusting cross-site requests | Unauthorized transactions by authenticated users | CSRF tokens, double-submit cookies, SameSite cookies |

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

| **Software** | **Category** | **Why** |
| --- | --- | --- |
| **Windows (or Android/iOS)** | System Software | Operating systems manage hardware resources, provide interfaces, and act as platforms for other software |
| **Device Drivers (e.g., NIC/Graphics)** | System Software | Drivers bridge the OS and hardware, enabling devices to function—core system duties . |
| **Google Chrome** | Application Software | A browser used directly by users to browse the web—a typical application layer tool . |
| **Microsoft Office (or LibreOffice)** | Application Software | Suites like Word/Excel perform specific end-user tasks (documents, spreadsheets) . |
| **Antivirus Utility (e.g., Windows Defender)** | System Software (Utility) | These run at the system level to protect the OS and hardware—part of system utilities . |

****

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

**1. 🖥️ Presentation Tier (UI / Client Layer)**

* **Role:** The user-facing interface—web browser or mobile app.
* **Responsibilities:** Display content, collect input, send requests to application tier.
* **Common Tech:** HTML/CSS/JavaScript; frameworks like React, Angular, Vue.js, or native mobile UI layers

**2. ⚙️ Application Tier (Business Logic)**

* **Role:** Core processing layer—handles logic, validation, workflows.
* **Responsibilities:** Process client requests, enforce business rules, communicate with data tier, return results.
* **Common Tech:** Java/Spring, Python/Django or Flask, Node.js/Express, .NET, etc.

**3. 🗄️ Data Tier (Storage)**

* **Role:** Handles persistent data storage and retrieval.
* **Responsibilities:** Execute queries, store/retrieve data, ensure consistency & integrity, backups, schemas.
* **Common Tech:** Relational DBs (MySQL, PostgreSQL), NoSQL (MongoDB), cloud services like AWS RDS .

A diagram of a computer

AI-generated content may be incorrect.

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

**Case Study: Functionality of Software Architecture Layers**

**Overview**

In software engineering, a **three-tier architecture** separates concerns into distinct layers:

1. **Presentation Layer**: Handles user interactions and displays information.
2. **Business Logic Layer**: Processes data and enforces business rules.
3. **Data Access Layer**: Manages data storage and retrieval operations.

This separation enhances modularity, maintainability, and scalability.

**1. Presentation Layer**

**Role**: Serves as the interface between the user and the system.

**Responsibilities**:

* **User Interface (UI)**: Provides interactive elements like forms, buttons, and displays.
* **Input Validation**: Performs basic checks on user input before processing.
* **Communication**: Sends user requests to the Business Logic Layer and presents responses.

**Technologies**:

* Web frameworks (e.g., React, Angular)
* Mobile frameworks (e.g., Flutter, SwiftUI)
* Desktop applications (e.g., Electron)

**2. Business Logic Layer**

**Role**: Acts as the core of the application, processing data and enforcing rules.

**Responsibilities**:

* **Data Processing**: Transforms and manipulates data according to business requirements.
* **Rule Enforcement**: Ensures that business constraints and policies are applied.
* **Workflow Coordination**: Manages the sequence of operations and interactions between components.

**Technologies**:

* Backend frameworks (e.g., Node.js, Django, Spring Boot)
* Microservices architecture
* API development (e.g., RESTful services)

**3. Data Access Layer**

**Role**: Handles all interactions with data storage systems.

**Responsibilities**:

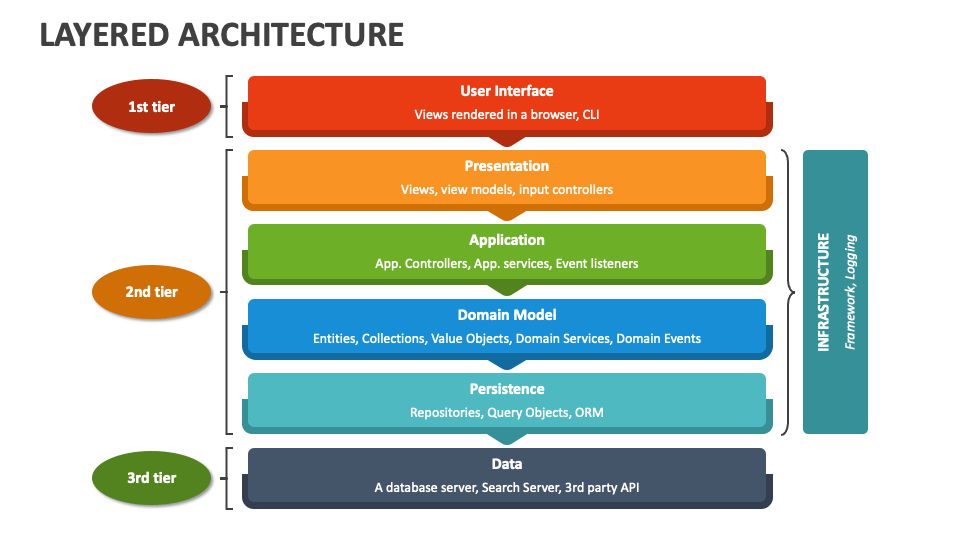
* **CRUD Operations**: Performs Create, Read, Update, and Delete operations on data.
* **Data Mapping**: Maps application objects to database records.
* **Connection Management**: Establishes and manages connections to data sources.

**Technologies**:

* Object-Relational Mapping (ORM) tools (e.g., Hibernate, SQLAlchemy)
* Database systems (e.g., MySQL, PostgreSQL, MongoDB)
* Caching systems (e.g., Redis)

**Benefits of Layered Architecture**

* **Separation of Concerns**: Each layer focuses on a specific aspect of the application, simplifying development and maintenance.
* **Scalability**: Layers can be scaled independently based on demand.
* **Maintainability**: Isolated layers make it easier to update or replace components without affecting the entire system.
* **Testability**: Individual layers can be tested separately, improving reliability.



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

**Types of Software Environments**

1. **Development Environment (Dev)**
   * Local or VM/containers where developers write and run code with rapid feedback.
   * Often IDE-based, isolated, and optimized for fast iteration
2. **Integration Environment**
   * Where modules from different developers are merged and checked.
   * Often tied to CI systems to catch early integration issues
3. **Testing/QA Environment**
   * Mirrors production configurations to run functional, performance, security, and regression tests.
   * Ensures close alignment with production to avoid false positives/negatives
4. **Staging (Pre‑Production) Environment**
   * A near-perfect replica of production used for final validation (staging/UAT/beta).
   * Ensures smooth release rollout
5. **Production Environment**
   * Live system serving end-users; must be highly reliable, secure, and monitored
6. **Specialized Testing Environments**
   * Includes performance testing, security testing, chaos testing, etc.

**Why Multiple Environments Are Important**

* Separates concerns (dev vs production risks)
* Allows early bug detection and safe experimentation
* Helps standardize deployment pipelines and environment parity
* Enables repeatable validation before live release

**Setting Up a Basic VM-Based Dev/Test Environment**

This guide uses VirtualBox + Vagrant, a common, free combo ideal for developers:

**Step 1: Install VM Tools**

* Install VirtualBox (free hypervisor).
* Install Vagrant for easy VM provisioning

**Step 2: Define a Vagrantfile**

Create a folder, then a Vagrantfile like below:

Vagrant.configure("2") do |config|

config.vm.box = "ubuntu/focal64"

config.vm.network "private\_network", ip: "192.168.56.10"

config.vm.provider "virtualbox" do |vb|

vb.memory = "2048"

vb.cpus = 2

end

config.vm.provision "shell", inline: <<-SHELL

apt-get update

apt-get install -y git build-essential

SHELL

end

**Step 3: Launch the VM**

vagrant up

vagrant ssh

**Step 4: Extend to Multiple Environments**

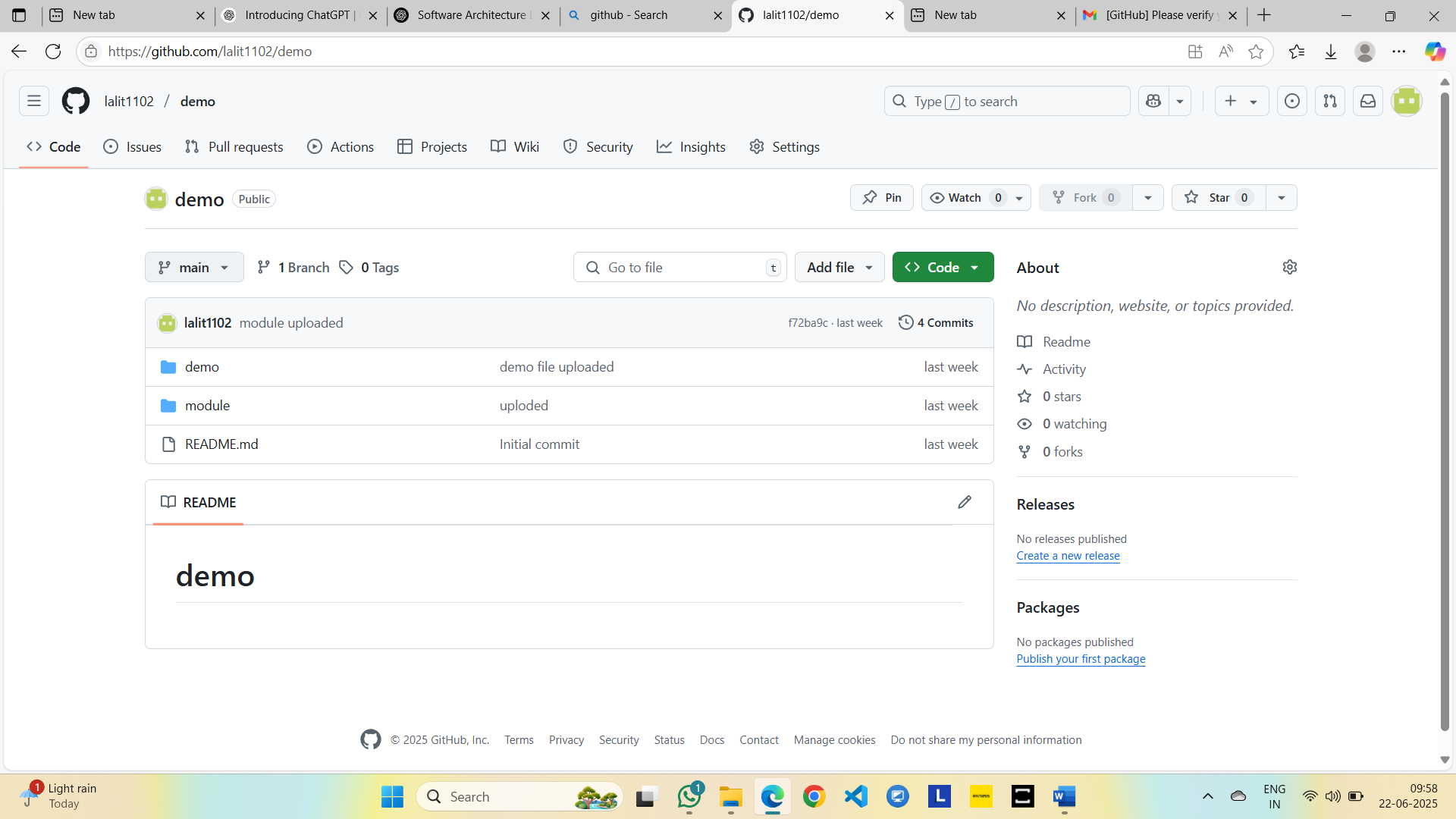
* **Test environment**: Add another VM in Vagrant or set up a separate VM/Container using Docker.
* **Staging**: Duplicate an environment with production-grade configs, data population, and environment flags.
* Use **IaC (Terraform, Ansible)** to automate reproducibility
* Write and upload your first source code file to Git hub.

**Create a GitHub repository**

1. Log in to GitHub and click **New repository** (plus icon in top right).
2. Name it e.g. **my-first-project**, add a description.
3. Choose **Public** (or Private) and check **Add a README file**.
4. Click **Create reposit**

**Upload via browser (quickest way)**

1. In your new repo on GitHub, click **Add file → Upload files**.
2. Drag and drop your module1 file (and optionally the README).
3. Add a commit message like *“module1”* and click **Commit changes**.
4. Voilà – your file is now online.

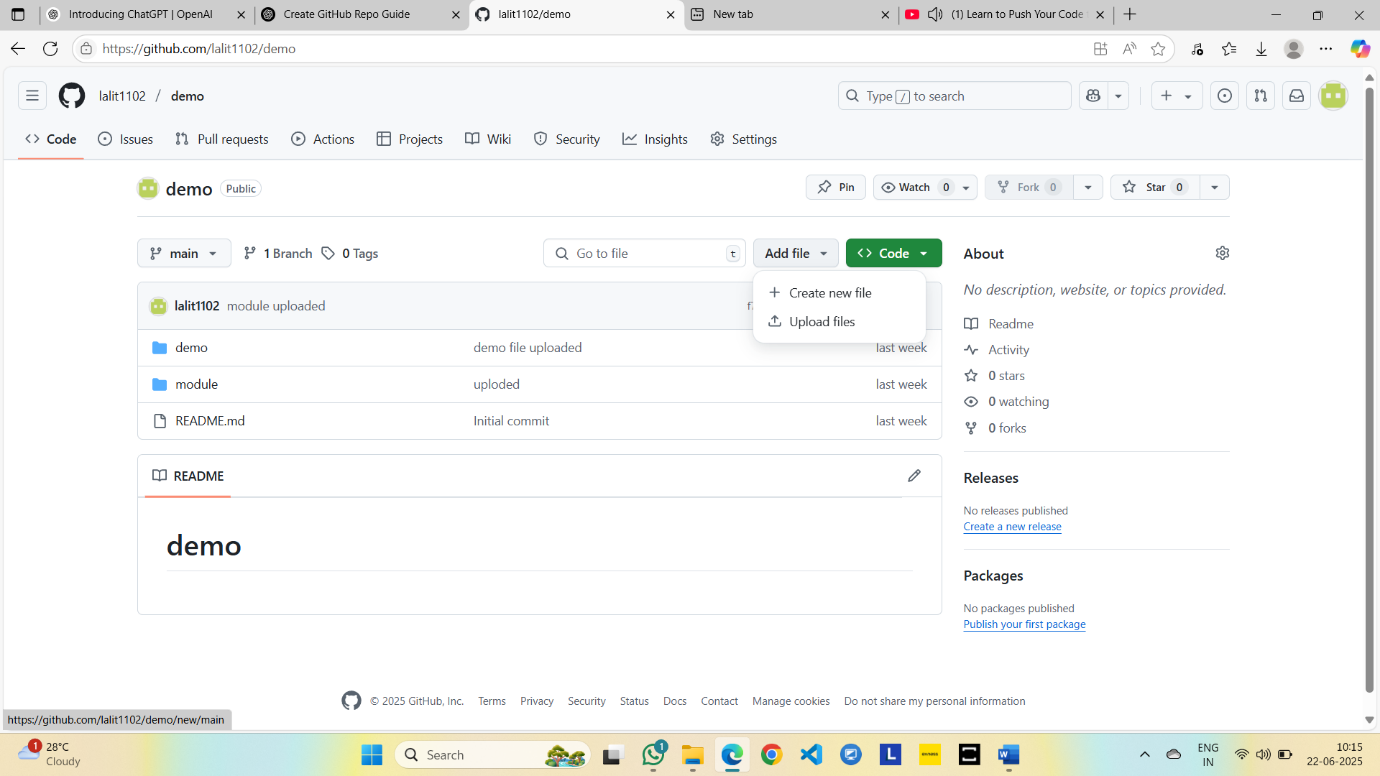
****

**A screenshot of a computer

AI-generated content may be incorrect.**

* Create a Git hub repository and document how to commit and push code changes.

Open add file.



Choose your files and commint changes

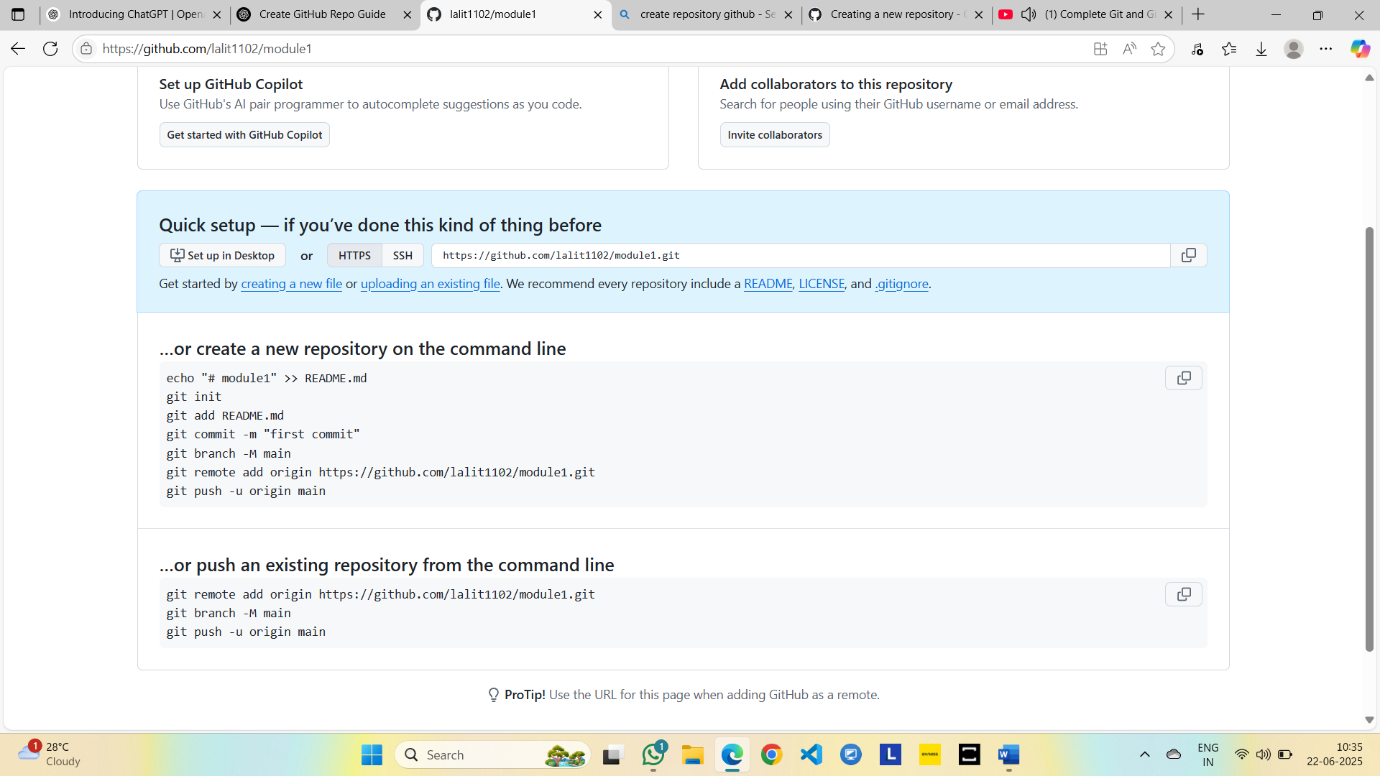
A screenshot of a computer

AI-generated content may be incorrect.

Open your file

A screenshot of a computer

AI-generated content may be incorrect.



* Create a student account on Git hub and collaborate on a small project with a classmate.

**Part 1: Create a GitHub Student Account**

1. **Visit** <https://github.com/join>
2. **Sign up** with your email, username, and password.
3. **Verify your email.**
4. **Apply for the GitHub Student Developer Pack**:
   * Go to: <https://education.github.com/pack>
   * Click “Get Student Pack.”
   * Use your **school email address** and provide proof of enrollment (e.g., student ID, school transcript, or a school-issued email).
   * Wait for approval (may take a few hours to a couple of days).

**🔹 Part 2: Set Up a Project Repository**

1. After account approval, **log in to GitHub**.
2. Click the **“+” icon** (top right) → **New repository**.
3. Name the repository (e.g., class-collab-project).
4. Select “Public” or “Private” and **initialize with a README**.
5. Click **Create repository**.

**🔹 Part 3: Collaborate with a Classmate**

1. On the repository page, go to **Settings → Collaborators**.
2. Add your classmate’s GitHub username or email to invite them.
3. Your classmate should accept the invitation from their notifications.

**🔹 Part 4: Work on the Project**

**1.Use Git or GitHub Desktop to clone the repository:**

git clone <https://github.com/your-username/class-collab-project.git>

**2.Create or edit a file, e.g., hello.py:**

print("Hello from Alice and Bob!")

**3.Save and commit your changes:**

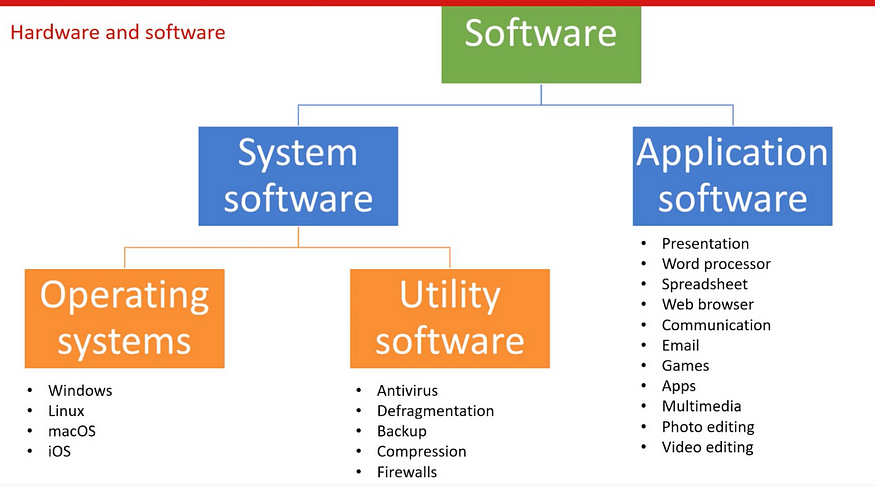
git add hello.py

git commit -m "Added hello.py"

git push

**4.Your classmate can now pull the changes, make edits, and push back.**

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



**1. System Software**

This is the foundational software that manages hardware and provides a platform for running applications Examples I use regularly:

* **Operating Systems**:
  + **Windows 10 Pro**
  + **Ubuntu 24.04 LTS**
* **Device Drivers & Firmware**:
  + NVIDIA GPU driver
  + BIOS/UEFI firmware updates

**🚀 2. Application Software**

These are programs for end-user tasks—word processing, web browsing, development, etc. My frequent ones include:

* **Web Browsers**:
  + Google Chrome
  + Mozilla Firefox
* **Office Suite**:
  + Microsoft Word, Excel, PowerPoint
* **Developer Tools**:
  + Visual Studio Code
  + IntelliJ IDEA
* **Media Players / Editors**:
  + VLC Media Player
  + Adobe Photoshop

**🛠️ 3. Utility Software**

These assist with maintenance, security, and optimization My go‑to utilities:

* **Antivirus/Security**:
  + Windows Defender
  + Malwarebytes
* **Backup & Recovery**:
  + Windows Backup & Restore
  + Mac Time Machine
* **Disk Tools & Cleaners**:
  + CCleaner
  + GParted (for Linux partitions)
* **System Monitors & Configurers**:
  + Task Manager (Windows)
  + MSConfig (Windows)
  + htop (Linux)
* **Compression/Archive Tools**:
  + 7-Zip
  + WinRAR
* Follow a GIT tutorial to practice cloning, branching, and merging repositories.

**Step 1: Install Git**

Make sure Git is installed. Run this in your terminal:

* git –version

**Step 2: Clone a Repository**

Find a public repo on GitHub or create your own. Then clone it:

* git clone https://github.com/username/repo-name.git
* cd repo-name

Or, create a local repo:

* mkdir git-practice
* cd git-practice
* git init

Step 3: **Create a New Branch**

* git checkout -b feature-branch

Make some changes (e.g., add a file)

* echo "Hello Git" > hello.txt
* git add hello.txt
* git commit -m "Add hello.txt"

Step 4: **Switch Back to Main Branch**

* git checkout main

Make a different change here too:

* echo "Main branch edit" > main.txt
* git add main.txt
* git commit -m "Add main.txt"

Step 5: **Merge the Feature Branch**

* git merge feature-branch

Step 6: **Push to Remote (Optional)**

* git push origin main
* Write a report on the various types of application software and how they improve productivity.

**1. Productivity & Office Suites**

Includes tools for generating documents, spreadsheets, presentations, and organizing tasks.

* **Examples:** Microsoft Office (Word, Excel, PowerPoint), Google Workspace (Docs, Sheets, Slides), Evernote
* **Productivity Impact:**
  + Streamlines content creation, data analysis, and information sharing.
  + Features like real-time collaboration and templates accelerate workflows.
  + Centralized platforms reduce context switching, saving time.

**💬 2. Communication & Collaboration Software**

Facilitates teamwork across distances through chat, audio/video calls, and shared workspaces.

* **Examples:** Slack, Microsoft Teams, Zoom, Skype, Google Meet
* **Productivity Impact:**
  + Enables instant messaging and group/video conferencing—cutting email chains.
  + Cohesive project coordination via document co-editing and planning tools.
  + Reduces delays in decision-making and improves remote team synergy.

**📊 3. Business & Enterprise Software**

Supports core organizational functions like finance, HR, sales, and supply chains.

* **Types & Examples:**
  + **CRM:** Salesforce, HubSpot, Zoho CRM
  + **ERP:** SAP, Oracle, Microsoft Dynamics
  + **Project/Resource Management:** Asana, Trello, Monday.com
* **Productivity Impact:**
  + Centralizes data and automates routine operations like invoicing or payroll.
  + Keeps teams aligned, reduces duplication, ensures better reporting and compliance.

**🗄️ 4. Database & Data Analysis Software**

Enables systematic storage, retrieval, and analysis of large datasets.

* **Examples:** MySQL, PostgreSQL, Oracle, MS SQL Server, MongoDB
* **Productivity Impact:**
  + Robust data management supports efficient business intelligence and analytics.
  + Web-based interfaces and BI integrations aid faster decision-making.

**🎨 5. Multimedia & Graphics Software**

Used for editing audio, video, and images to enhance communication and marketing.

* **Examples:** Adobe Photoshop, Premiere Pro, GIMP, VLC, Audacity
* **Productivity Impact:**
  + Streamlines content creation for presentations, social media, and training materials.
  + Integrated multimedia enhances visual appeal and user engagement.

**🧠 6. Educational & Reference Software**

Supports learning through interactive content, tutorials, and resource libraries.

* **Examples:** Coursera, Udemy, Khan Academy, Duolingo, Wikipedia
* **Productivity Impact:**
  + Encourages self-paced learning and skill-building.
  + Quick access to references improves research efficiency and knowledge.

**🛠️ 7. Utility & System-Enhancement Software**

Tools dedicated to optimizing performance, security, and maintenance.

* **Examples:** Antivirus software (Norton, Avast), CCleaner, backup tools (Acronis, EaseUS)
* **Productivity Impact:**
  + Keeps systems running smoothly and securely—minimizes downtime.
  + Automates cleanup and maintenance to reduce manual oversight.

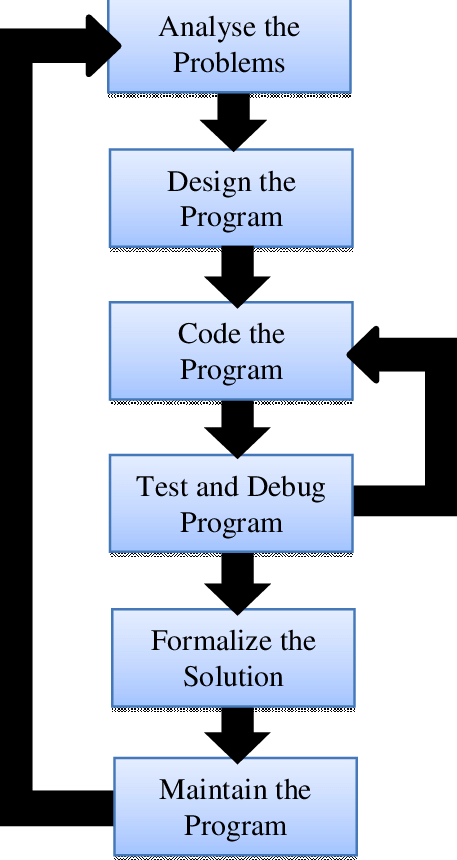
**🧩 8. Specialized & Custom Applications**

Tailored solutions for niche or industry-specific tasks.

* **Examples:** ANSYS, AutoCAD, MATLAB (engineering); Epic Systems (healthcare); legal case management .
* **Productivity Impact:**
  + Automates complex calculations and domain-specific workflows.
  + Minimizes manual errors and increases task-specific efficiency.

Productivity Gains

* **Automation:** Replacing manual steps with auto-generated reports or reminders.
* **Collaboration:** Enabling seamless teamwork and shared environments.
* **Optimization:** Improving system reliability, security, and speed.
* **Efficiency:** Cutting down repetitive tasks and accelerating creativity.
* **Knowledge Access:** Providing on-demand learning and faster decision-making.
* Create a flowchart representing the Software Development Life Cycle (SDLC).



**Requirement-Analysis**  
• Gather functional and non-functional needs from stakeholders.  
• Document them in an SRS (Software Requirements Specification).

**Feasibility-Study**   
• Evaluate technical, financial, operational, and schedule feasibility.

**Design**  
• Create High-Level and Low-Level Design, including architecture, data models, and UI mockups.

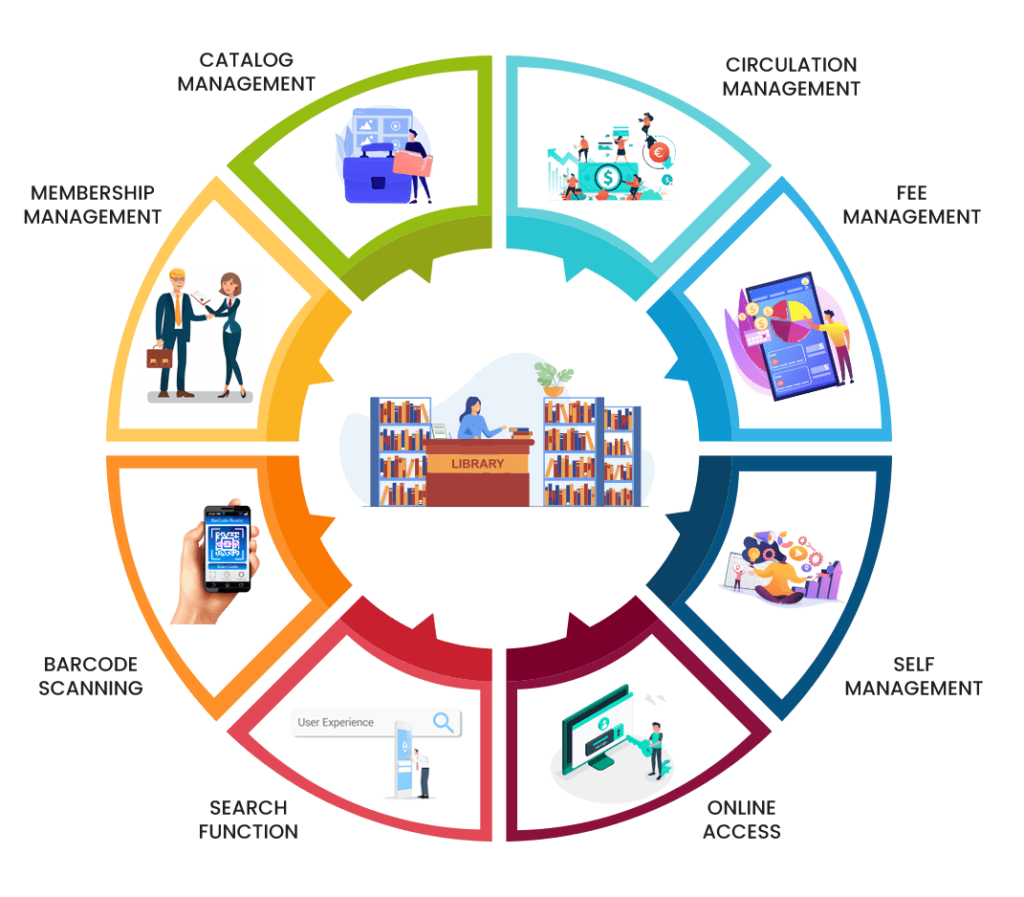
**Implementation/Coding**  
• Developers write source code to fulfill the design.  
• Use compilers, IDEs, and follow coding standards.

**Testing**  
• Perform unit, integration, system, performance, and acceptance testing.  
• Fix any defects and re-test until quality criteria are met.

**Deployment (Installation)**  
• Release the software to production or client environments.  
• Conduct user training and prepare documentation.

**Maintenance**  
• Perform ongoing support: bug fixes, updates, and enhancements

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



**1. Introduction**

**1.1 Purpose**

Define requirements for a Library Management System to manage books, users, borrowing/returning, fines, and reporting.

**1.2 Scope**

Supports these user roles: **Admin**, **Librarian**, **Patron**. Functions include:

* User registration, login, role-based access
* Catalog management (add/edit/remove books)
* Searching, borrowing, returning, renewing
* Fine calculations and payments
* Reports on inventory, loans, and overdue items

**2. Overall Description**

**2.1 Product Perspective**

Stand-alone web-based system interfacing with back-end SQL database.  
May integrate with external cataloging or RFID/barcode scanners.

**2.2 User Classes**

* **Admin**: full control — manages users, configs
* **Librarian**: manages catalog, loans, reports
* **Patron**: search catalog, manage own loans, view fines.

**2.3 Operating Environment**

Runs in browsers over LAN/WAN with backend support for Windows/Linux/Mac and databases like MySQL or PostgreSQL.

**2.4 Design Constraints**

Must support browser compatibility, secure HTTPS, responsive design, and integration with barcode/RFID hardware

**3. Functional Requirements**

**3.1 User Management**

* **Registration/Login** via email/id+password with validation
* **Role-based access**: admins/librarians can modify data; patrons view own info **3.2 Catalog/Search**
* Store detailed book info (ISBN, title, author, publisher, category, copies, location)
* Search by title, author, ISBN, keywords, availability; include advanced filters **3.3 Book Transactions**
* Issue, return, and renew books: track by ISBN and patron ID, enforce due dates
* Check limits (e.g., max books per patron), restrict in-library use as needed
* Auto-calculate fines for overdue returns; confirm fine payment before closure **3.4 Reporting**
* Generate reports for total loans, overdue items, inventory, and user borrow history
* Export downloadable reports (PDF or Excel)

**3.5 Notification**

* Automatically remind patrons of upcoming due dates or fines via email or in-system alerts.

**4. Non-functional Requirements**

* 1. **Usability**
* Simple, intuitive UI with clear fields and navigation; accessible features such as screen-reader support

**4.2 Performance**

* Support 100–250 concurrent users
* Search results in ≤ 2–3 seconds under full load

**4.3 Security**

* Encrypted communication (HTTPS) and encrypted data at rest
* Role-based authentication and authorization
* Input validation, secure password handling, comprehensive audit logs

**4.4 Reliability & Availability**

* ≥ 99% uptime with backup and data recovery mechanisms
* Graceful error handling and failover support

**4.5 Scalability**

* Scalable to multiple library branches or institutions
* Modular design to support future integration (e.g., acquisitions, serials)

**4.6 Maintainability**

* Modular code structure, separation of UI, business logic, and data layers
* Version-controlled and documented system configuration

**5. External Interfaces**

**5.1 User Interface**

* Web-based pages: login, dashboard (role-specific), catalog/search, transaction, reports
* Responsive HTML/CSS/JS UI for desktop and mobile

**5.2 Hardware Interfaces**

* Barcode/RFID scanner support for book and user IDs

**5.3 Software Interfaces**

* Integration with SQL databases (MySQL/PostgreSQL)
* Potential plug‑ins for catalog services or authentication systems

**5.4 Communication Interfaces**

* HTTPS with RESTful APIs for UI-backend interactions

**6. Data Requirements**

* Database tables: **Users**, **Books**, **Loans**, **Fines**, **Reports**, **AuditLogs**

**7. Appendix**

* **Glossary**: LMS, ISBN, RFID, OPAC, CRUD, HTTPS
* **Revision history**: concept version, author, date
* **References**: IEEE 29148:2018 SRS standards
* Perform a functional analysis for an online shopping system.

**1. User Management**

**Functions:**

* **User Registration**
  + Collect user details (name, email, password, address, etc.)
* **User Login/Logout**
  + Secure authentication
* **Profile Management**
  + Update personal and shipping information
* **Password Recovery**
  + Email or OTP-based reset

**2. Product Catalog Management**

**Functions:**

* **Product Listing**
  + Display product name, image, price, category, and stock status
* **Search & Filter**
  + Search by keyword
  + Filter by price, brand, category, rating, etc.
* **Product Details**
  + Show description, specs, images, and reviews

**3. Shopping Cart System**

**Functions:**

* **Add to Cart**
  + Store selected items with quantity
* **View/Edit Cart**
  + Change item quantity or remove items
* **Price Calculation**
  + Show subtotal, tax, and total cost

**4. Checkout and Payment**

**Functions:**

* **Shipping Information**
  + Collect address and delivery method
* **Payment Integration**
  + Support payment gateways (e.g., credit card, PayPal)
* **Order Summary**
  + Final review before placing an order
* **Order Confirmation**
  + Send confirmation email/receipt

**5. Order Management**

**Functions:**

* **Order Tracking**
  + View current order status (pending, shipped, delivered)
* **Order History**
  + Users can view past purchases
* **Cancel/Return Requests**
  + Process cancellation or return requests per policy

**6. Review and Rating System**

**Functions:**

* **Post Reviews**
  + Allow users to write reviews for purchased products
* **Rate Products**
  + 1–5 star rating system

**7. Admin Panel (Back-End Management)**

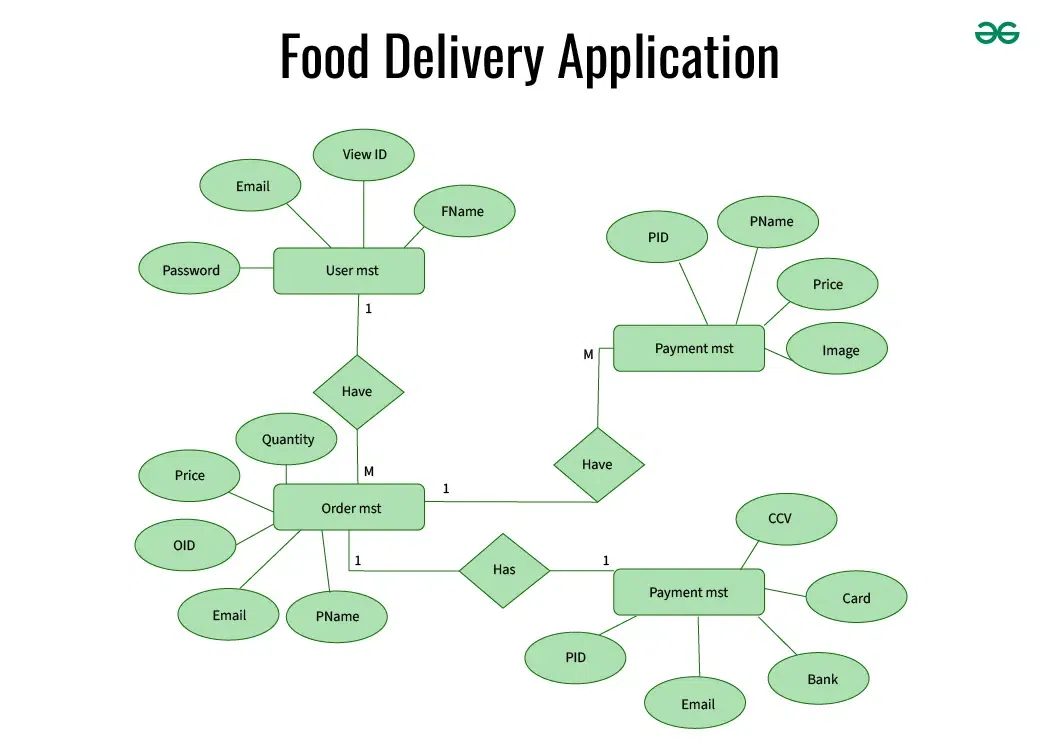
**Functions:**

* **Product Management**
  + Add, update, or remove products
* **User Management**
  + View users, deactivate accounts if needed
* **Order Management**
  + View, process, and update order statuses
* **Analytics Dashboard**
  + Track sales, user activity, and inventory

**8. Security and Performance**

**Functional Considerations:**

* Encrypted login and payment data
* Rate-limiting or captcha to prevent abuse
* Responsive design for mobile and desktop
* Design a basic system architecture for a food delivery app.

****

**🔷 1. Client Tier (Front-End)**

**a. User App (Customers)**

* Browse restaurants and menus
* Place orders
* Track delivery
* Make payments

**b. Restaurant App (Vendors)**

* Manage menu and availability
* Accept/reject orders
* Update order status

**c. Delivery App (Drivers)**

* Receive delivery assignments
* Navigate to pickup/drop-off
* Update delivery status

**🔷 2. Application Tier (Back-End Services / APIs)**

**a. User Management Service**

* Authentication (login/signup)
* Profile and address management

**b. Restaurant Management Service**

* Menu creation/editing
* Restaurant info, hours, ratings

**c. Order Management Service**

* Order placement, tracking, and status updates
* Assigning delivery partners

**d. Delivery Management Service**

* Real-time tracking of delivery agents
* Route optimization and ETA calculation

**e. Payment Service**

* Payment gateway integration
* Invoicing and refund processing

**f. Notification Service**

* In-app, SMS, or email notifications
* Order updates and promotions

**🔷 3. Data Tier (Database Systems)**

* **User DB:** Stores customer and driver data
* **Restaurant DB:** Stores menus, hours, ratings
* **Order DB:** Tracks order history and status
* **Payment DB:** Transaction records
* **Logs/Analytics DB:** User behavior, performance metrics

**🔷 4. Third-Party Integrations**

* **Maps API:** Google Maps or Mapbox for routing
* **Payment Gateway:** Stripe, PayPal, Razorpay
* **SMS/Email:** Twilio, SendGrid for communication

**🔷 5. Admin Panel (Web Interface)**

* Manage users, restaurants, and orders
* Monitor system activity
* View analytics and generate reports

**🔷 Deployment/Infrastructure**

* **Cloud Platform:** AWS, Azure, or GCP
* **Services:** Load balancer, Auto-scaling, CDN
* **Security:** HTTPS, data encryption, firewall
* Develop test cases for a simple calculator program.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Calculator</title>

<style>

body {

font-family: Arial, sans-serif;

display: flex; justify-content: center; align-items: center;

height: 100vh; background: #f0f0f0;

}

.calculator {

background: #fff; padding: 20px;

border-radius: 10px; box-shadow: 0 0 10px rgba(0,0,0,0.1);

}

.display {

width: 200px; height: 40px;

font-size: 1.5em; text-align: right;

margin-bottom: 10px; padding: 5px;

}

.buttons {

display: grid;

grid-template-columns: repeat(4, 50px);

gap: 10px;

}

.buttons button {

font-size: 1.2em; border: none;

background: #4CAF50; color: #fff;

border-radius: 5px; cursor: pointer;

}

.buttons .equals {

background: #2196F3;

grid-column: span 2;

}

</style>

</head>

<body>

<div class="calculator">

<input type="text" class="display" id="display" readonly>

<div class="buttons">

<!-- Row 1 -->

<button onclick="clearDisplay()">C</button>

<button onclick="backspace()">⌫</button>

<button onclick="appendOperator('%')">%</button>

<button onclick="appendOperator('/')">÷</button>

<!-- Row 2 -->

<button onclick="appendNumber('7')">7</button>

<button onclick="appendNumber('8')">8</button>

<button onclick="appendNumber('9')">9</button>

<button onclick="appendOperator('\*')">×</button>

<!-- Row 3 -->

<button onclick="appendNumber('4')">4</button>

<button onclick="appendNumber('5')">5</button>

<button onclick="appendNumber('6')">6</button>

<button onclick="appendOperator('-')">−</button>

<!-- Row 4 -->

<button onclick="appendNumber('1')">1</button>

<button onclick="appendNumber('2')">2</button>

<button onclick="appendNumber('3')">3</button>

<button onclick="appendOperator('+')">+</button>

<!-- Row 5 -->

<button onclick="appendNumber('0')">0</button>

<button onclick="appendNumber('.')">.</button>

<button class="equals" onclick="calculate()">=</button>

</div>

</div>

<script>

const display = document.getElementById('display');

function appendNumber(num) {

display.value += num;

}

function appendOperator(op) {

if (display.value && !/[+\-\*/%]$/.test(display.value)) {

display.value += op;

}

}

function clearDisplay() {

display.value = '';

}

function backspace() {

display.value = display.value.slice(0, -1);

}

function calculate() {

try {

display.value = eval(display.value);

} catch {

display.value = 'Error';

}

}

</script>

</body>

</html>

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

Here’s a detailed real-world incident involving a **hotel software failure** that required urgent, critical maintenance.

**Incident Summary**

In early April 2024, Omni Hotels & Resorts experienced a major **cyberattack**, leading them to shut down critical IT systems to contain the threat

**🛎️ Immediate Impact on Hotel Operations**

* **Reservation systems**, **digital door locks**, and **point-of-sale terminals** went offline across more than 50 properties in the U.S. and Canada
* Guests faced delays with check-in/out processes and room access—many had to be **escorted to rooms manually** due to non-functional digital keys
* **Payment systems were down**, forcing staff to resort to manual charge processing or defer billing

**🚑 Critical Maintenance & Recovery**

1. **Go-live mitigation**
   * Omni proactively shut down affected systems to halt the cyberattack’s impact
2. **IT restoration & containment**
   * Systems were gradually restored; antivirus scans and patches were deployed prior to rebooting services
3. **Fallback procedures**
   * Staff reverted to manual check-in/out, hand-issued room keys via front desk, and used alternate payment methods.
   * Phone and record-keeping fell back on manual or job-runaround processes

* Create a DFD for a hospital management system

A diagram of a hospital management system

AI-generated content may be incorrect.

A diagram of a medical management system

AI-generated content may be incorrect.

* Build a simple desktop calculator application using a GUI library

import tkinter as tk

# Calculator logic functions

def button\_click(value):

current = display\_var.get()

display\_var.set(current + str(value))

def clear():

display\_var.set("")

def calculate():

try:

result = str(eval(display\_var.get()))

display\_var.set(result)

except:

display\_var.set("Error")

# Set up the Tkinter window

root = tk.Tk()

root.title("Calculator")

root.geometry("312x400")

root.resizable(0, 0)

display\_var = tk.StringVar()

display\_var.set("")

# Display field

entry = tk.Entry(root, textvariable=display\_var, font=("Arial", 20), bd=10, insertwidth=2, justify="right")

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

# Buttons

btn\_texts = [

'7','8','9','/',

'4','5','6','\*',

'1','2','3','-',

'C','0','=','+'

]

row = 1; col = 0

for txt in btn\_texts:

action = clear if txt == 'C' else calculate if txt == '=' else lambda t=txt: button\_click(t)

tk.Button(root, text=txt, width=5, height=2, font=("Arial", 18),

command=action).grid(row=row, column=col, padx=5, pady=5)

col += 1

if col > 3:

col = 0

row += 1

root.mainloop()

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

A diagram of a course registration

AI-generated content may be incorrect.