1. Analyze CPU Types (CISC vs. RISC)

Objective:

Understand processor instruction sets and their relevance to data science workflows.

Tasks:

• Define CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing).

The CPU(central processing unit) is the brain of the computer. Its job is to process instructions that tell the computer what to do(e.g. Adding numbers, saving data, etc).

Goal was to make the CPU more efficient, faster and simpler.

CISC(Complex Instruction Set Computing):

Came first – in the 1960s–70s.
Back then, memory was expensive and slow. So, engineers tried to:

- Use fewer instructions to save space.
- Make each instruction do more work (like reading from memory and processing in one step).

These instructions were complex but meant you wrote shorter programs.
Used in computers like IBM mainframes and later Intel x86 CPUs.

Main Idea was to make the CPU smarter so that it takes less code to do a task.

RISC(Reduced Instruction Set Computing):

Developed in the **1980s**.
Researchers at **IBM**, **UC Berkeley**, and **Stanford** noticed something:
Even in CISC processors, most programs only used a **small number of instructions** over and over.
Main Idea was to simplify the instruction set and make each one run **super fast and keep the CPU simple and let software do the heavy lifting**

They removed the rarely used complex instructions and kept only the **common simple ones**, optimized for speed.
This led to **RISC processors** like:

- **ARM** (used in phones/tablets)
- **MIPS**, **SPARC**, **RISC-V**

Feature	CISC	RISC
Definition	A CPU design where a single instruction can perform multiple operations (e.g., memory + processing) in one go.	A CPU design using a small set of simple instructions, each performing a single operation quickly.
Instruction Set	Large and complex (100s of instructions)	Small and simple (fewer instructions)
Instruction Complexity	One instruction can do multiple tasks	Each instruction does only one task
Execution Time	Instructions take **multiple cycles** to execute	Most instructions take **one cycle**
Memory Access	Instructions can access memory directly	Only **load/store** instructions access memory
Hardware Complexity	More complex hardware (decoding, control logic)	Simpler hardware design
Code Size	Smaller (fewer instructions needed)	Larger (more instructions to do same task)
Performance	Optimized for code density	Optimized for speed and power efficiency
Used In	Desktops, Laptops (Intel x86, AMD)	Mobile devices, Embedded systems (ARM, RISC-V)
Compiler Dependency	Less dependent on compiler	Heavily relies on compiler for optimization
Design Philosophy	Hardware does more work	Software (compiler) does more work

• Research and tabulate 2 popular CPUs for each architecture.

Architecture	CPU Model	Manufacturer	Where It's Used
CISC	Intel Core i9-13900K	Intel	High-end desktops and gaming PCs
CISC	AMD Ryzen 9 7950X	AMD	Workstations, performance desktops
RISC	Apple M2	Apple (based on ARM)	MacBooks, iPads
RISC	Qualcomm Snapdragon 8 Gen 3	Qualcomm	Android smartphones and tablets

• Discuss: For a machine learning pipeline involving a lot of matrix operations, which architecture would be theoretically better and why?

ML pipelines will require high throughput with repeated work(Simple, repetitive operations)

Matrix operations (like in deep learning: matrix multiplication, dot product, etc.) involve **simple operations** repeated millions of times (add, multiply, store).
RISC processors are optimized for this pattern - fast, simple instructions executed **in one clock cycle**.

Matrix math can be executed with **greater efficiency and speed**.
RISC designs focus on **pipelining and parallel execution**.Matrix operations benefit greatly from **parallel processing** (doing many things at once).RISC CPUs (and especially GPUs based on RISC principles like ARM cores or RISC-V) can handle **many matrix elements at the same time**.

RISC-based CPUs (like ARM) are used for **low power usage and thermal efficiency**.RISC helps reduce power consumption while still performing fast matrix calculations.

2. Explore Memory Hierarchy

Objective:

Understand how different types of memory impact performance.

Tasks:

• Draw a pyramid showing memory hierarchy: CPU Registers → Cache (L1, L2, L3) → RAM → SDD/HDD.

• Define the role of each layer in the hierarchy.

1. CPU Registers

- **Location**: Inside the CPU
- **Role**: Temporarily hold **immediate data** and **instructions** during execution.
- **Speed**: **Fastest** memory, directly accessible by the processor.
- **Size**: Very small (usually <1 KB).
- **Example**: Storing results of a calculation temporarily like A + B.

2. Cache Memory (L1, L2, L3)

- **Location**: Inside or close to the CPU
- **Role**: Stores **frequently accessed data/instructions** to avoid accessing slower RAM.
- **Levels**:
    - **L1**: Closest and fastest (per-core), but very small (~32–128 KB).
    - **L2**: Larger and slower (~256 KB–1 MB).
    - **L3**: Shared among cores (~4–64 MB), slowest among caches.
- **Speed**: Very fast, but slower than registers.
- **Use**: Acts as a **buffer** between CPU and RAM.

3. RAM (Random Access Memory)

- **Location**: On motherboard
- **Role**: Stores **active programs, data, and OS** temporarily while the system is on.
- **Speed**: Slower than cache, faster than storage.
- **Volatility**: **Volatile** – data is lost when power is off.
- **Example**: Loaded applications, browser tabs, current editing files.

4. SSD/HDD (Storage)

- **Location**: Internal drive or external storage
- **Role**: Long-term **storage of all files**, applications, and the operating system.
- **Speed**:
    - **SSD**: Much faster than HDD (up to ~5000 MB/s for NVMe).
    - **HDD**: Slowest (~100–200 MB/s), mechanical parts.
- **Volatility**: **Non-volatile** – retains data after power is off.

• Explain how cache size and speed impact data preprocessing speed for large datasets.

Cache is **small but ultra-fast memory** close to the CPU. It temporarily holds frequently used data and instructions so the CPU doesn't have to fetch them from slower RAM.

1. Speed of Cache

- **Faster cache** → CPU spends **less time waiting** for data → operations run quicker.
- When data fits in cache, access is in **nanoseconds**, while RAM or disk can take **micro- to milliseconds**.

Impact: Faster cache allows **quicker access to frequently used data chunks**, boosting preprocessing steps like normalization, filtering, or transformation.

2. Size of Cache

- Larger cache = **more data fits** = fewer trips to slower RAM or SSD.
- Preprocessing involves repeating operations on similar blocks of data (e.g., rows of a dataset, image batches).
- A small cache may constantly **evict and reload data**, slowing performance due to "cache misses".

Impact: Bigger cache reduces misses and speeds up **loop-heavy** operations (e.g., applying functions row-by-row).

3.Multithreading & Cache

When multiple cores process different data chunks (e.g., parallel processing), **each core benefits from having enough cache**. If not, they compete for memory access and performance drops.

• Explain why machine learning algorithms benefit from more RAM and faster caches.

1. **Why More RAM Helps**

RAM stores **active data** — including:

- Input datasets
- Features, labels
- Model parameters during training
- Intermediate results (e.g., gradients, activations)

Scenario	Without Enough RAM	With More RAM
Large dataset loading	Uses disk (very slow)	All fits in memory (fast access)
Model training	Frequent disk I/O	Entire training loop stays in RAM
Batching and preprocessing	Frequent reloading	Smooth chunk processing

2. **Why Faster Caches Help**

What Cache Does: Cache temporarily stores hot data — things the CPU needs right now or repeatedly, such as:

- Feature vectors
- Partial computations
- Weights being updated

Cache Use Case	Impact if Slow or Small	Impact if Fast and Large
Vector/matrix operations	Slower loops, cache misses	Fast dot products, fast loops
Gradient updates (e.g., SGD)	Waits on memory every step	Fast weight updates per batch
Repeated calculations	Redundant memory calls	Done instantly from cache

3. Understand Virtual Memory and Paging

Objective:
Learn memory management at the OS level.
Tasks:
• Define Virtual Memory, Paging, and Page Faults.

Virtual memory is a memory management technique that gives the illusion of a large, continuous memory space — even if the physical RAM is limited.
Each process thinks it has **its own large memory**.Behind the scenes, the OS maps virtual memory to **physical RAM + disk (swap space)**.

- Allows running **bigger programs** than physical RAM.
- Provides **memory protection** (processes can't access each other's memory).
- Enables **multitasking**.

Paging is the method of dividing **virtual memory** and **physical memory** into fixed-size blocks.
Pages in virtual memory (e.g., 4 KB each).
Frames in physical memory RAM (same size as pages).

- The OS keeps a **page table** to map pages ↔ frames.
- If a page is in RAM, it's mapped directly.
- If not, the page is on disk (swap area), and must be loaded.

A **page fault** occurs when a program tries to access a **page not currently in RAM(physical memory)**.

What Happens:

1. CPU requests a page.
2. OS checks the page table.
3. If it's **not in RAM**, a **page fault is triggered**.
4. OS:
    - Pauses the program.
    - Loads the page from disk (slow).
    - Updates the page table.
    - Resumes the program.

• Give real-world examples of how virtual memory helps when data scientists work with large datasets.

1. **Loading a Huge CSV with Pandas:** Suppose you only have 8 GB RAM, and the CSV takes 5–6 GB to load. Your OS might already be using RAM for other apps (browser, Jupyter).**Virtual memory** steps in by moving unused parts of memory (other processes or older dataframe rows) to **disk (swap file)**, freeing RAM for `dataframe`.
→ **load and process** the large file without a crash.

2. **Training a Deep Learning Model:** Training uses large tensors that may temporarily exceed RAM. As training progresses, **older tensors or gradients** not in use get paged out.**Virtual memory** ensures your training doesn't fail due to a brief RAM overload.

Challenge	How Virtual Memory Helps
Dataset size > physical RAM	Swaps unused parts to disk
Multiple processes/notebooks	Shares limited RAM smartly
Model training uses temporary memory spikes	Handles spikes with no crash
Concurrent data loading + visualizations	Keeps apps running without failure

Virtual memory acts like a safety net for data scientists — making sure your system stays responsive and your workflow doesn't break even when your dataset is **too big for RAM**.

• Briefly explain the impact of page swapping on model training speed**.**

Page swapping occurs when the system runs out of RAM and moves data between RAM and disk (swap space or page file). While this prevents crashes, it has a significant impact on speed, especially during model training in machine learning**.**

Training uses lots of memory (for tensors, gradients, batches).
If RAM is full, the OS **swaps out less-used memory pages** to disk.
When those pages are needed again, they must be **swapped back** into RAM.

Every time swapping occurs, the model **pauses** while waiting for data to load from disk — sometimes **thousands of times per epoch**, causing: Frequent I/O bottlenecks, Longer training epochs, **System slowdown or thrashing** if swapping is excessive.

Page swapping drastically slows down model training because accessing disk is **thousands of times slower** than RAM.
It's always better to ensure sufficient RAM or use **batching and memory-efficient techniques** to avoid swapping during training.

4. Guide for Buying a Laptop for Data Science

Objective:

Design a buying guide based on CPU and memory understanding.

Tasks:

• Recommend CPU configurations (multicore, multithreaded, cache size).
• Recommend minimum RAM (e.g., 16GB vs 32GB).
• Recommend SSD type and size (NVMe vs SATA).
• Include GPU recommendations if deep learning is a goal (mention CUDA-enabled GPU).
• Write a short guide:
"Choosing the Right Laptop for Data Science" covering all the above points clearly.

1. CPU (Processor) Recommendations

Feature	Recommendation
Cores	Minimum 6 cores (preferably 8+ for multitasking and faster model training)
Threads	12+ threads (Multithreading boosts parallel data operations)
Architecture	Choose RISC-based efficiency cores + performance cores (Apple M-series or Intel hybrid cores)
Clock Speed	At least 3.0 GHz base, boost up to 4+ GHz
Cache Size	Minimum 12 MB L3 cache (more is better for data reuse in loops and ML operations)
Recommended CPUs	Intel Core i7/i9 13th/14th Gen AMD Ryzen 7/9 7000 series Apple M2/M3 Pro or Max

2. RAM (Memory) Recommendations

Usage Type	Recommended RAM
General Data Science	16 GB minimum
Large datasets, ML pipelines	32 GB strongly recommended
Deep Learning/Heavy Multitasking	32–64 GB (for image/video/data-heavy tasks)

Prefer DDR5 or LPDDR5 for faster bandwidth. Avoid systems with non-upgradable soldered RAM unless you're getting 32+ GB pre-installed.

3. Storage (SSD) Recommendations

Feature	Recommendation
Type	NVMe SSD (PCIe Gen 4 or 5) — 5–7x faster than SATA SSD
Size	Minimum 512 GB (1 TB recommended if storing local datasets/models)
Speed	Look for 3500 MB/s+ read/write

Avoid SATA SSD for serious ML work — it bottlenecks during large I/O operations.

4. GPU

A CUDA-enabled GPU drastically improves deep learning performance by offloading matrix operations from the CPU.

Use Case	Recommendation
No DL (basic ML/pandas)	Integrated GPU is OK (Intel Iris Xe, Apple M2 GPU, etc.)
Light DL / occasional training	NVIDIA RTX **3050 / 3060**
Regular DL training / vision/NLP models	RTX **3070 / 3080 / 4060+**, or **NVIDIA A-series (Pro)**
Apple users	**M2/M3 Max** has strong ML accelerators, though limited CUDA compatibility

Important: Ensure your GPU is **CUDA-enabled** if using PyTorch or TensorFlow with GPU support.

5. Analyze Storage Interfaces and Devices

Objective:

Understand common storage interfaces and storage device characteristics relevant to data science workflows.

Tasks:

• List and describe major storage interfaces: SATA, PCI, PCIe, NVMe.

Interface	Full Form	Description	Common Use
SATA	Serial ATA (Advanced Technology Attachment)	A traditional interface used to connect hard drives and SSDs to the motherboard. Slower than PCIe but widely supported.	2.5" SATA SSDs, HDDs
PCI	Peripheral Component Interconnect	An older standard for connecting peripheral devices (e.g., network cards, GPUs, storage cards). Replaced by PCIe.	Legacy devices (rare today)
PCIe	Peripheral Component Interconnect Express	A high-speed serial expansion bus. Used for modern GPUs, SSDs, network cards. Offers faster data transfer with multiple "lanes" (x1, x4, x16).	NVMe SSDs, GPUs
NVMe	Non-Volatile Memory Express	A protocol designed for **fast SSDs** that connect via PCIe. Allows direct communication between the SSD and CPU, bypassing older bottlenecks like SATA.	M.2 NVMe SSDs, U.2 SSDs

Feature	SATA	PCI (legacy)	PCIe	NVMe
Type	Interface + Protocol	Interface	Interface	Protocol (uses PCIe)
Max Speed	~600 MB/s (SATA III)	~133 MB/s per device	Up to 32 GB/s (PCIe 5.0 x16)	Depends on PCIe version (e.g., 3.5 GB/s for PCIe 3.0 x4)
Latency	High	Very high	Low	Very low (optimized for SSDs)
Common Devices	2.5" SSDs, HDDs	Old expansion cards	GPUs, NVMe SSDs	M.2 NVMe SSDs

• Compare SSD vs. HDD based on speed, cost, durability, and use cases.

Feature	SSD (Solid State Drive)	HDD (Hard Disk Drive)
Speed	Very fast (500 MB/s to 7,000+ MB/s)Instant boot/load	Slower (~80–160 MB/s)Longer boot/load times
Cost per GB	Higher	Lower
Durability	No moving parts → More resistant to shock/vibration	Has spinning parts → Can be damaged by drops
Lifespan	Finite write cycles (but improving yearly)	Can last longer if used for light tasks
Capacity	Common: 256 GB – 2 TB (larger available at high cost)	Common: 500 GB – 8 TB (cheaper for high capacities)
Power Use	Lower (better for laptops)	Higher (more suitable for desktops)
Form Factor	Mostly 2.5", M.2, PCIe cards	Mostly 2.5" (laptops), 3.5" (desktops)

• Research and create a table of sample IOPS and throughput values for:
o SATA SSD
o NVMe SSD
o HDD (7200 RPM)

Storage Type	Typical IOPS (Random 4K Read)	Sequential Read Speed	Sequential Write Speed	Latency (Avg.)
HDD (7200 RPM)	~75–150 IOPS	~100–160 MB/s	~100–140 MB/s	~5–10 ms
SATA SSD	~75,000 – 100,000 IOPS	~500–550 MB/s	~450–500 MB/s	~0.05 ms
NVMe SSD (PCIe 3.0 x4)	~300,000 – 500,000 IOPS	~3,000 – 3,500 MB/s	~2,500 – 3,000 MB/s	~0.02 ms
NVMe SSD (PCIe 4.0 x4)	~600,000 – 1,000,000+ IOPS	~5,000 – 7,000 MB/s	~4,000 – 6,000 MB/s	~0.01–0.015 ms

IOPS (Input/Output Operations Per Second): Measures how many read/write operations the drive can handle per second — crucial for workloads like databases and random access.

Sequential Read/Write: Measures how fast large files can be read/written — important for loading datasets, videos, VMs.

Latency: Time it takes to begin responding to a request — lower latency = snappier performance.

• Discuss:
In a data science project involving constant reading/writing of 100 GB+ data, which storage type would you recommend and why?

Recommended Storage Type: NVMe SSD for High throughput, Low latency, Massive IOPS, Smooth parallel data access.

Feature	Why NVMe SSD is Ideal
Speed	NVMe SSDs offer **3,000–7,000 MB/s** read/write speed, **20–40× faster** than HDDs and **6–10× faster** than SATA SSDs.
IOPS	Excellent for random reads/writes. Can handle **300,000+ IOPS** compared to ~100 on HDDs.
Latency	Ultra-low latency (~0.02 ms), crucial for high-throughput data loops and model training steps.
Concurrent Access	Enables **parallel I/O** (good for multiprocessing data loaders like in PyTorch or TensorFlow).
Efficiency	Saves time in **disk-based operations** like saving large NumPy arrays, caching preprocessed data, or loading models.

6. Simulate Storage Bottleneck
Objective:
Understand storage impact on data pipelines.
Tasks:
• Design a simple data ingestion scenario:
o Suppose you have 10 million small files (10 KB each).
o Alternatively, suppose you have 5 files (2 GB each).
• Predict:
o Which scenario would suffer if the IOPS is low?
o Which would suffer if the Throughput is low?
• Summarize how storage design influences ETL pipeline performance.

Scenario 1:

10 million small files (10 KB each)

- Total data = ~100 GB
- Each file requires **separate open/read/close** operations
- Dominated by **random access**

Scenario 2:

5 large files (2 GB each)

- Total data = ~10 GB
- Each file read in large, continuous blocks
- Dominated by **sequential access**

Scenario	Suffers if IOPS is low	Suffers if Throughput is low
10M small files (10 KB)	Yes — **Millions of file accesses** will queue up and slow down	Not critical (files are small)
5 large files (2 GB)	Not much impact (few accesses)	Yes — **Large file reading** will be slow

7. Introduction to Containers
Objective:
Learn about containers and how they differ from VMs.
Tasks:
• Define Containers and key tools like Docker.

Containers are lightweight, portable environments that bundle an application and all its dependencies (code, libraries, tools) so it runs reliably across different computing environments.
They Share the **host OS kernel,** Are isolated from each other and the host, Are much faster and lighter than Virtual Machines.

Docker is the most popular containerization platform.It allows you to create, run, and manage containers with reproducibility and ease.You define container configurations using a **Dockerfile**.

• Create a comparison table: Containers vs Virtual Machines based on startup time, resource usage, isolation level.

Feature	Containers	Virtual Machines (VMs)
Startup Time	Seconds (near-instant)	Minutes (slower to boot)
Size	Lightweight (MBs to 100s of MBs)	Heavy (GBs, includes full OS)
Resource Usage	Efficient (shares OS kernel)	High (needs full guest OS per VM)
Isolation Level	Process-level isolation (less strict)	Full hardware-level isolation
Performance	Near-native	Overhead from hypervisor
Portability	High — runs on any OS with Docker	Lower — needs hypervisor compatibility
Use Case	Ideal for **microservices, ML models, CI/CD**	Ideal for **complete OS simulation or testing**

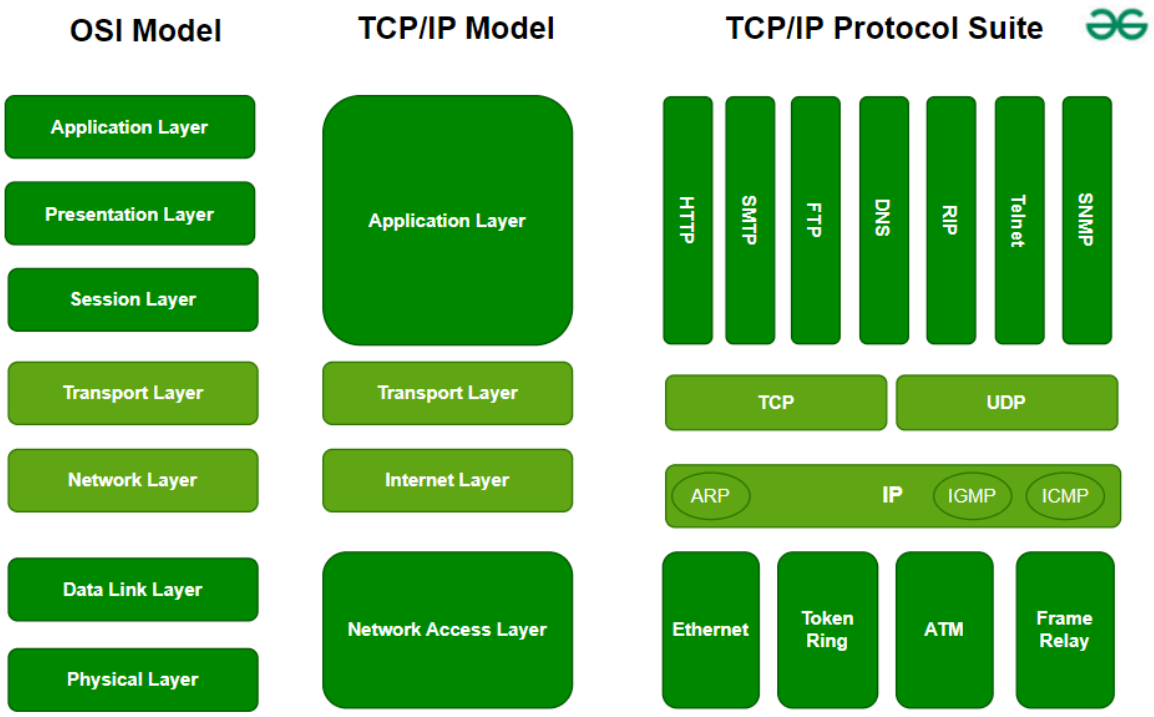
• Explain:
o Why containers are preferred for model deployment?

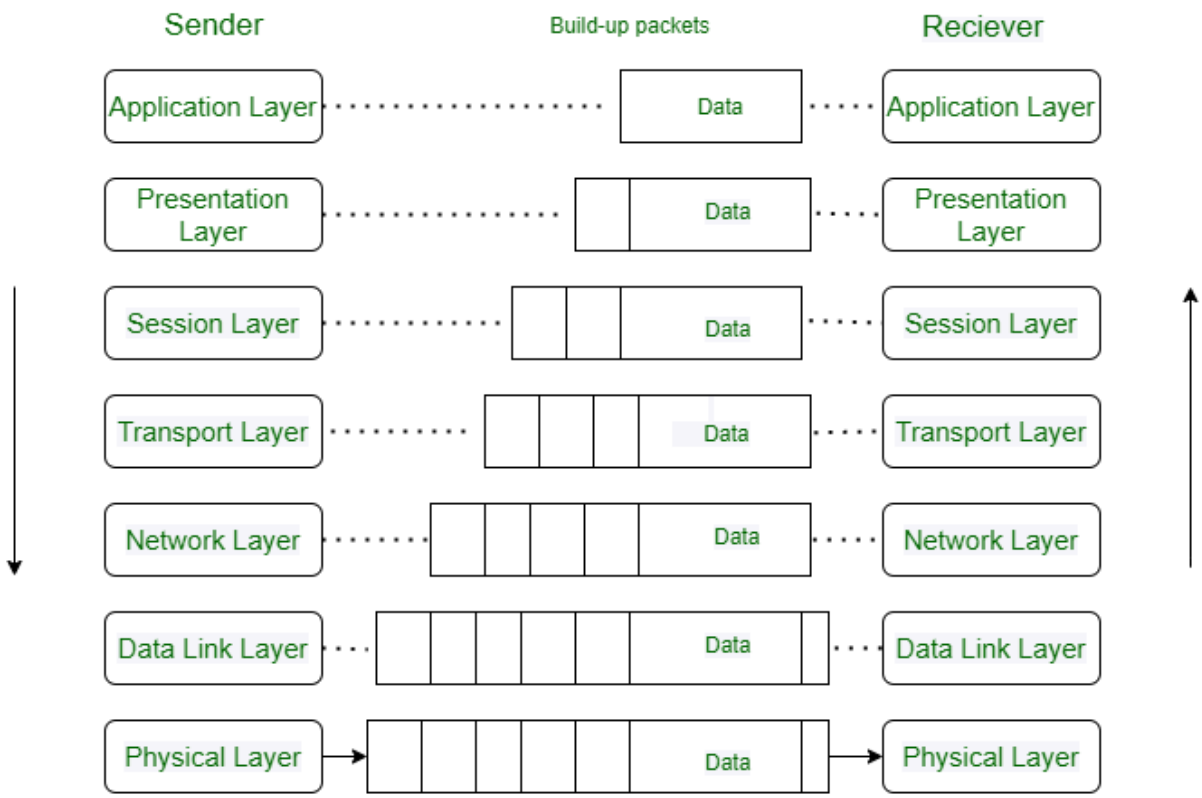
1. **Consistency** – Same environment on dev, test, and production
2. **Portability** – Run your model anywhere Docker is supported (cloud, edge, on-prem)
3. **Isolation** – Prevents conflicts with other apps or Python libraries
4. **Lightweight** – Quickly spin up or scale down model servers
5. **Reproducibility** – Ensures reproducible training and inference environments

o Give an example where you would use Docker in a data science workflow.

Deploying a trained ML model as a REST API. you train a model and save it as .pkl file. create a docker file . build and run the container. Model will run in a fully portable environment that works the same everywhere.

8. Study and Document TCP/IP Model
Create a mapping:
• How the TCP/IP model corresponds to the OSI model.
• Describe each layer's role (Application, Transport, Internet, Network Access).





Application Layer — Responsible for providing services to the user.

Presentation Layer — Take care of syntax and semantics of the information exchange between two communication system.

Session Layer — It stablish, maintain, synchronize, and terminate the interaction between sender and receiver.

Transport Layer — Responsible for process to process delivery.

Network Layer — Responsible for delivery of individual packet from source to destination.

Data Link Layer — Responsible for moving frame from one hop to next hop.

Physical Layer — Responsible for moving individual bits from one device to the next device.

TCP/IP Model Layer	Corresponding OSI Model Layers	Layer Role
Application Layer	Application (7) + Presentation (6) + Session (5)	Interfaces with user apps, manages data formatting, encryption, sessions.
Transport Layer	Transport (4)	Ensures reliable or fast delivery (TCP/UDP), manages segmentation and ports.
Internet Layer	Network (3)	Handles logical addressing (IP) and routing of packets across networks.
Network Access Layer	Data Link (2) + Physical (1)	Manages physical transmission and MAC addressing over cables, Wi-Fi, etc.

TCP/IP Model Layer Descriptions

1. Application Layer

- Combines OSI's Application, Presentation, and Session layers.
- Protocols: **HTTP, FTP, SMTP, DNS, SSH**
- **Function**:
    - Provides network services to end-user applications (e.g., browsers, email clients).
    - Handles encoding/decoding, compression, session setup.

2. Transport Layer

- Corresponds to OSI's Transport layer.
- Protocols: **TCP (reliable), UDP (fast but unreliable)**
- **Function**:
    - Manages end-to-end communication between devices.
    - Breaks data into segments and handles error correction, flow control, and port addressing.

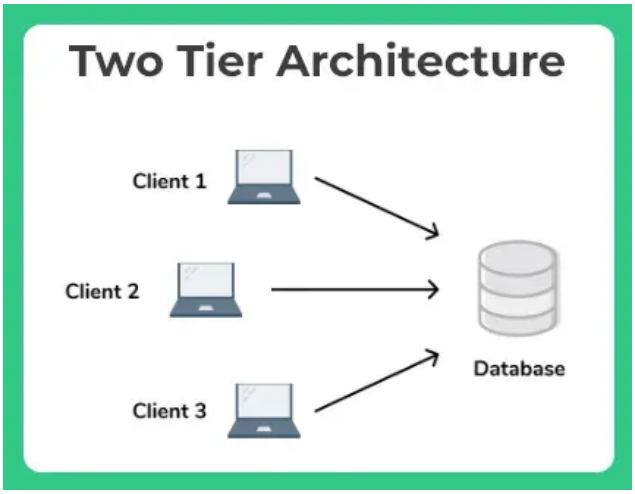
3. Internet Layer

- Corresponds to OSI's Network layer.
- Protocols: **IP (IPv4/IPv6), ICMP, ARP**
- **Function**:
    - Determines the best path for data across networks (routing).
    - Adds IP addresses to packets for source and destination.

4. Network Access Layer

- Combines OSI's Data Link + Physical layers.
- Protocols: **Ethernet, Wi-Fi, PPP**
- **Function**:
    - Handles how data is physically transmitted over the medium.
    - Manages MAC addressing, framing, error detection at hardware level.

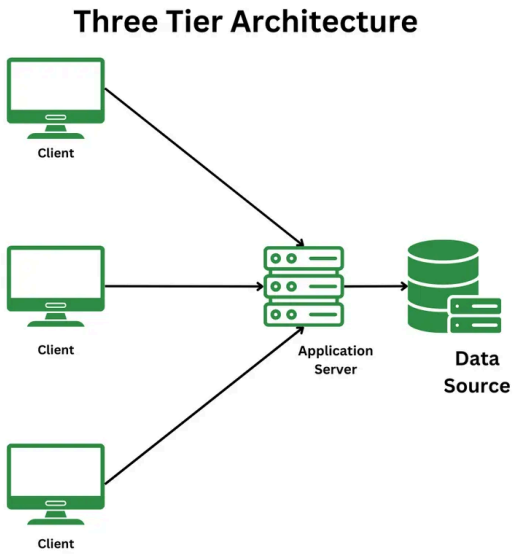
9. 2-tier vs 3-tier Architecture
Title:
Compare and design 2-tier and 3-tier architectures for an Online Food Ordering System.
Problem Statement:
Design two models:
• 2-tier model: Direct communication between user app and database.
• 3-tier model: Introduction of an application server in between.



Element	Role
Client Tier	Sends requests (e.g., place order, view menu)
Data Tier	Stores data like users, orders, restaurants, menus
Communication	Direct SQL or API-like calls to DB

Drawbacks:

- Tight coupling of client and data
- Limited scalability
- Exposes database directly — **security risks**
- Business logic on client = harder to update



Tier	Role
Presentation Tier	User interface (frontend app or browser)
Application Tier	Handles business logic (e.g., order validation, user auth) — built using Node.js, Django, Java Spring, etc.
Data Tier	Database stores all persistent data (users, orders, menus, payments)

Advantages:

- **Better Security**: DB is not exposed directly
- **Scalability**: Easily load-balance app tier
- **Maintainability**: Update logic without touching clients
  Reusability: Same app tier serves mobile + web apps

Feature	2-Tier Architecture	3-Tier Architecture
Layers	Client + Database	Client + Application Server + Database
Business Logic Location	Client-side	Application server
Security	Weaker (DB directly accessed)	Stronger (DB hidden behind app layer)
Scalability	Low	High (can scale app and DB tiers independently)
Maintainability	Poor	Good — logic changes are centralized
Use Case Suitability	Simple apps, small teams	Large apps, secure and maintainable deployments

For a **growing online food ordering business**, I would prefer the **3-tier architecture**.While the 2-tier model may be sufficient for initial prototypes, the **3-tier architecture is future-ready**. It provides **security, scalability, and maintainability** — essential traits for a platform expected to grow in traffic, feature complexity, and user base. With 3-tier architecture, you can **update your business logic independently**, support both **mobile and web platforms** through reusable APIs, and **scale horizontally** by load balancing application servers as demand increases. Though initial setup may be more demanding, it ensures the system remains robust, adaptable, and performant as the business expands.

10. Batch and Stream Processing with GCP Dataflow

Title:

Design Data Pipelines for E-commerce Transaction Data.

Problem Statement:

You are processing E-commerce transaction logs.

You have two requirements:

1. Batch processing: Analyze total sales at the end of each day.
2. Stream processing: Detect fraud in real-time.

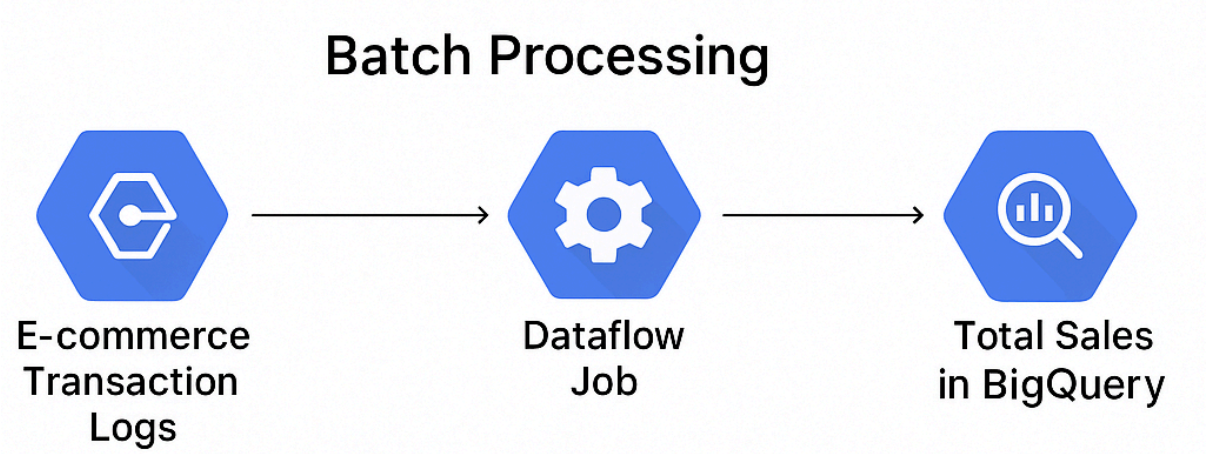
Tasks:

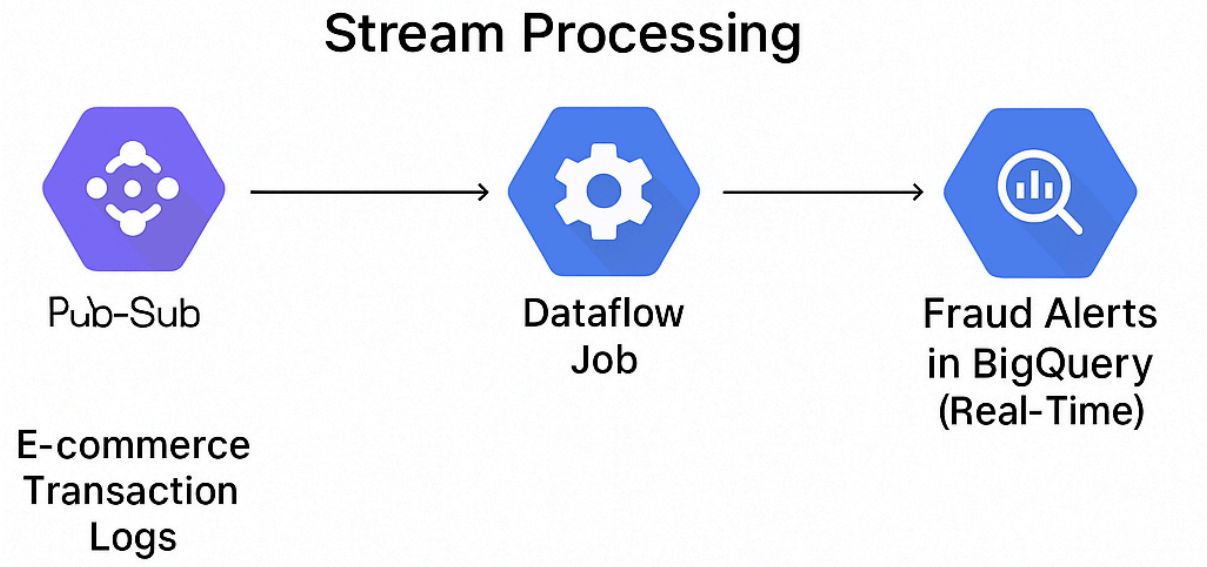
• Draw two separate pipeline diagrams using GCP Dataflow.

• For batch, mention input source (e.g., GCS), Dataflow job, output (e.g., BigQuery).

• For stream, mention input source (e.g., Pub/Sub), Dataflow job, output (e.g., BigQuery realtime table).

• List differences between batch and stream pipelines for this case.





Aspect	Batch Processing	Stream Processing
Goal	Summarize data at the end of day	Detect fraud in real-time
Input Source	GCS (stored files)	Pub/Sub (real-time messages)
Latency	High (hours)	Low (seconds or milliseconds)
Frequency	Scheduled (e.g., daily)	Continuous
Dataflow Type	Bounded (fixed data size)	Unbounded (infinite data stream)
Use Case	Reporting, business intelligence	Fraud detection, real-time alerting
Storage Target	BigQuery (append daily summary)	BigQuery (live-updated table)
Cost	Lower for small runs	Higher due to always-on processing

11. CAP Theorem in Distributed Systems

Title:
Apply CAP Theorem to a Messaging Application like WhatsApp.
Problem Statement:
WhatsApp needs to balance Consistency, Availability, and Partition Tolerance when delivering messages worldwide.
Tasks:
• Briefly explain CAP Theorem in your own words.

CAP Theorem says that in any distributed system, you can **only fully guarantee two**

Letter	Stands for	Meaning
C	Consistency	Every user sees the same data at the same time.
A	Availability	The system always responds (even if with old/stale data).
P	Partition Tolerance	The system keeps working even if there are network failures (partitions).

• Analyze which two properties WhatsApp prioritizes and why (Consistency vs Availability in case of network failure).

CAP in WhatsApp

Partition Tolerance is a Must. WhatsApp operates **globally** across mobile networks with high chances of **network partitioning(e.g., user offline, poor signal, server disconnect).**
So Partition Tolerance (P) is non-negotiable.

Consistency vs. Availability Trade-off

WhatsApp Prioritizes:

- **Availability (A)**: Users should always be able to **send messages**, even during a network partition.
- **Partition Tolerance (P)**: App should still work in disconnected scenarios.

WhatsApp *relaxes* Consistency slightly:

- Messages may be **delivered at different times** across devices.
- Message **read/delivery receipts (✓✓)** may **appear late or out of order**.

• Give a real-world example where the tradeoff happens (e.g., slight delays in message delivery to ensure consistency).

Imagine you send a message to a friend while you're in airplane mode:

- **WhatsApp lets you send the message anyway** → This shows **Availability**.
- **It stores the message locally and syncs later** → This maintains **Partition Tolerance**.
- **Your message shows only one grey tick (✓)** until delivery is confirmed → Slight **Consistency lag**.

Property	Maintained?	Reason
Consistency (C)	Relaxed	May show delays in delivery/read receipts
Availability (A)	Yes	Messages can be sent anytime, even offline
Partition Tolerance (P)	Yes	Works despite server or network issues

12. Cloud Service Models

Title:

Select the Right Service Model (IaaS, PaaS, SaaS) for a Startup Data Science Project.

Problem Statement:

A startup needs to run machine learning models on large datasets but wants minimal infrastructure management.

Tasks:

• Define IaaS, PaaS, SaaS briefly.

Model	Meaning	Responsibility Split
IaaS (Infrastructure as a Service)	Provides virtualized computing resources over the internet.	You manage: OS, runtime, ML tools, data. Provider manages: hardware, networking, VM.
PaaS (Platform as a Service)	Provides a managed platform for developing, running, and managing apps.	You manage: code and ML logic. Provider manages: infra, OS, scaling, environment.
SaaS (Software as a Service)	Ready-to-use applications managed entirely by the provider.	You just use the software; everything else is managed.

• Identify which service model fits best.

Requirement: Run ML models on large datasets with **minimal infrastructure management**.

Best Fit: PaaS (Platform as a Service)

- The startup can focus on **building and running ML pipelines**.
- No need to manage servers, OS patches, or scale manually.

• Suggest GCP services corresponding to each model (e.g., Compute Engine for IaaS, App Engine for PaaS, BigQuery for SaaS).

Cloud Model	GCP Example	Use Case
IaaS	**Compute Engine**	Full control over VMs, custom ML environments
PaaS	**Vertex AI, AI Platform**, App Engine	Train/evaluate ML models with built-in scaling
SaaS	**BigQuery**, **AutoML**, **Looker Studio**	Analyze/query data, train no-code models, dashboards

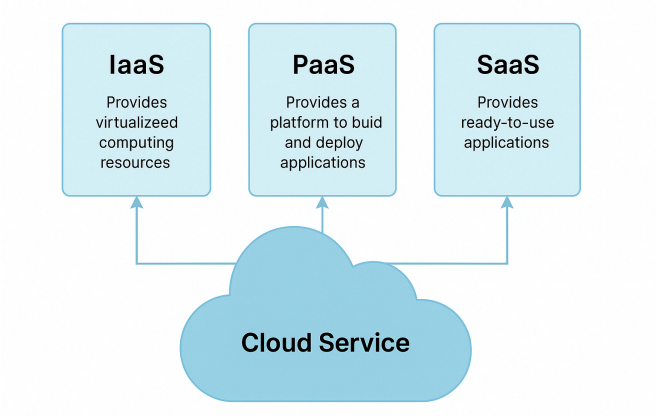
• Recommend one model and justify the choice.

Recommendation: Use PaaS as it is Ideal for a Startup Data Science Project:

- **No DevOps burden** – Focus on ML, not server setup.
- **Faster time to prototype** – Use tools like Vertex AI with Jupyter support.
- **Scales automatically** – No need to pre-provision compute power.
- **Cost-effective** – Pay-as-you-use for compute and storage.
- **Security and updates handled** – Managed by Google.

Example Workflow Using PaaS on GCP

1. Store data in **Cloud Storage** or **BigQuery**
2. Build and train models using **Vertex AI Workbench (JupyterLab)**
3. Deploy trained models using **Vertex AI Model Deployment**
4. Monitor predictions via **Cloud Logging & Monitoring**



13. GPU Usage and Distributed Computing in Data Science

Title:

Plan the Infrastructure for Training a Deep Learning Model on GCP.

Problem Statement:

You are tasked to train a deep neural network (DNN) model requiring high computational power for image classification.

Tasks:

• Describe when and why you need GPU support instead of CPU.

Scenario	CPU	GPU
Small ML models	Sufficient	Overkill
Deep Learning (CNNs, RNNs)	Very slow	Highly efficient
Matrix-heavy operations	Bottlenecks	Parallel execution
Training time	Hours to days	Minutes to hours

• Choose between using GCP services like:
  -AI Platform with GPUs
  -Compute Engine with custom GPU machines

Option	Description	When to Use
AI Platform with GPUs	Managed ML training pipeline (Vertex AI)	When you want auto-scaling, hyperparameter tuning, managed environment
Compute Engine (GPU VMs)	Full control of VM with chosen GPU (e.g., NVIDIA A100, T4)	When you want custom environments, multi-GPU clusters, or advanced control

Recommendation: Use **AI Platform (Vertex AI)** for simplicity unless:

- You need advanced customization (e.g., custom Docker images, Horovod).
- You want to control distributed training setup with frameworks like PyTorch Lightning or TensorFlow.

• Explain how CUDA cores help in parallelizing model training.

- **CUDA** is NVIDIA's GPU programming API that allows software (like TensorFlow or PyTorch) to directly use GPU power.
- A typical NVIDIA GPU has thousands of **CUDA cores**.
- Instead of training one image at a time (CPU), **CUDA-enabled GPUs** can:
    - Process **batches of images** in parallel.
    - Execute **matrix multiplications** across many cores.
    - Significantly reduce **epoch time**.

Example: A CNN that takes 10 hours to train on a CPU might take **1 hour or less on a GPU**.

• Create a basic training workflow.

Step-by-step Basic Training Workflow (on GCP)

1. **Data Preparation**
    - Upload images to **Cloud Storage (GCS)**
    - Preprocess and normalize (optionally using Dataflow)
2. **Model Development**
    - Use **Vertex AI Workbench (JupyterLab)** with GPU support
3. **Training**
    - Submit training job via **Vertex AI Custom Training**
    - Use a **pre-built TensorFlow/PyTorch container with GPU support**
4. **Evaluation**
    - Evaluate on test set stored in BigQuery or GCS
    - Visualize metrics.
5. **Model Deployment**
    - Deploy model to **Vertex AI Endpoint**
    - Enable GPU inference if real-time speed is needed
6. **Monitoring**
    - Track metrics, latency, and performance using **Cloud Monitoring**

14. Apply Software Design Principles

Title:

Design a Modular and Scalable Data Preprocessing System for Machine Learning.

Problem Statement:

You are tasked with creating a data preprocessing pipeline that can handle different datasets (CSV,
JSON, SQL databases).

Tasks:

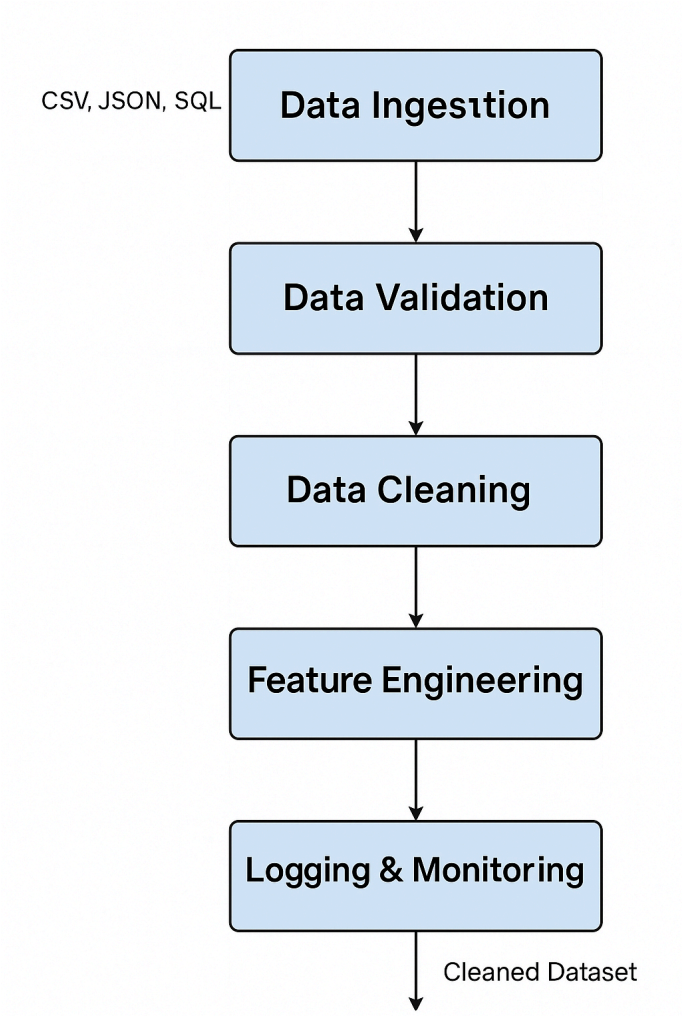
• Break down the system into independent modules (e.g., Data Ingestion, Data Cleaning, Feature Engineering).
• Explain how you will ensure Scalability as data volume grows.
• Mention how you will design for Maintainability (code updates, new formats).
• Identify parts of the system that will be Reusable across different projects.

Deliverables:

• System module description (with a diagram)
• Explanation on scalability, maintainability, and reusability

Modules:

Module	Description
1. Data Ingestion	Reads data from different sources: local files, APIs, cloud storage, or SQL databases.
2. Data Validation	Checks for schema consistency, null values, data types.
3. Data Cleaning	Handles missing values, removes duplicates, fixes outliers.
4. Feature Engineering	Performs encoding, scaling, normalization, time-series features, etc.
5. Logging & Monitoring	Logs data issues, transformation stats, tracks pipeline failures.
6. Data Output	Stores cleaned data in a staging area (e.g., DataFrame, Parquet, BigQuery).



How the System Handles Key Design Principles

1. Scalability

- **Parallel Processing**: Use Dask or Spark to parallelize ingestion, cleaning, and transformation.
- **Cloud Storage Integration**: Read/write from GCS, S3, or Azure Blob to handle TB-scale data.
- **Stream Support** (optional): Add support for Kafka/PubSub for real-time pipelines.

2. Maintainability

- **Config-Driven Design**: Each module reads from a `config file` (e.g., schema, thresholds).
- **Plug-n-Play Modules**: Use class-based or function-based design where modules can be swapped independently.
- **Version Control**: Use Git for tracking pipeline logic, requirements, and updates.

3. Reusability

- **Reusable Modules**:
  - `load_data(source_type, path_or_conn)`: works for CSV, JSON, SQL.
  - `clean_missing_values(df, method='mean')`: generic cleaning logic.
  - `encode_categorical(df, strategy='one-hot')`: reused in most projects.
- **Reusable Across Projects**:
  - Data ingestion functions
  - Feature transformers
  - Logging framework
  - Config-driven schemas

Aspect	Design Choice
Modularity	Separated ingestion, validation, cleaning, feature steps
Scalability	Spark/Dask, cloud-native design, parallel pipelines
Maintainability	Config files, versioned code, independent modules
Reusability	Core functions reused across datasets/projects