Data Engineering Lifecycle: It describes how data is captured, stored, processed, and delivered. Main stages: Generation → Ingestion → Storage → Transformation → Serving. Undercurrents (always present): Security, Data Management, DataOps, Data Architecture, Orchestration, Software Engineering.

Type A Data Engineers (Abstraction): They keep architecture simple and rely on managed services and off-the-shelf tools. Their focus is running the lifecycle efficiently without building custom systems. **Type B Data Engineers (Build):** They build custom data systems for scale or unique, mission-critical needs when existing services cannot meet the requirements.

Internal-Facing Data Engineer: They focus on building and maintaining data systems that support internal teams such as analytics, BI, and machine learning. Their work improves data access, quality, and reliability inside the organisation. **External-Facing Data Engineer:** They build data systems that power products used by customers or partners. This includes handling application data, APIs, and features where data engineering directly affects external users.

Upstream: Software Engineers, Data Architects, DevOps. **Downstream:** Data Analysts, Data Scientists, ML Engineers, **Business Leaders** (CIO/CTO/CDO).

Data Pipeline: A system that collects, processes, and moves data from sources to storage or applications. It often cleans, transforms, and organizes data so it can be effectively used for analytics, reporting, or machine learning. **Key stages:** Data Sources → Ingestion → Storage → Transformation → Serving → Analytics/ML.

Storage Considerations: No one-size-fits-all: Different data types and workloads need different storage solutions. Schema flexibility: Some systems need fixed schemas; others handle evolving or semi-structured data. Storage vs. query: Some store data cheaply, others let you query it directly. Performance and scalability: Consider read/write speed, ability to scale, and reliability guarantees (SLA).

Data acquisition/ingestion: process of collecting raw data from various sources and bringing it into a system for storage and processing. It's often the main bottleneck in a data pipeline due to volume, velocity, and diversity of data. **Key Consideration** Reusability: Can the data be used for multiple purposes? Batch vs. Streaming and Method: Push or pull? On-demand ingestion needed. Volume and Velocity: How much data and how fast does it arrive. Data Format: What format is the incoming data in?

Data cleaning and transformation: process of converting raw data into a usable, high-quality form for downstream applications. This stage is where the data starts generating real value for users or systems. **Key Considerations:** Data Quality: Does the data meet downstream quality requirements? System Compatibility: Can existing systems handle the current format, or are transformations needed? Simplicity: Are transformations simple and self-contained? Cost vs. ROI: What is the cost and return on investment of the transformation?

Filesystem Navigation: pwd – show current directory. cd ¡name¿ – change directory. cd – go home. cd - – previous directory. cd / – root directory. ls [¡name¿] – list contents. ls -al – detailed list. ls -R – recursive list. mkdir ¡name¿ – create directory. mv ¡old¿ ¡new¿ – move/rename. cp ¡source¿ ¡destination¿ – copy file

File Content: cat filename – show file content. more filename / less filename – page-by-page view. head filename – first 10 lines. tail filename – last 10 lines. wc filename – lines, words, characters. sort filename – sorted content.

Piping: Treats everything as a stream of data. Output of one command can be passed directly to another using —. Example: cat filename — sort — head -3 — reads, sorts, and shows the first 3 lines.

Grep: A command-line tool used to search for patterns in files and display matching lines.**AWK:** A text-processing language for extracting, transforming, and manipulating data from files line by line.

Data Acquisition: File Access – Data from existing files or downloaded datasets. Formats: CSV, Excel, XML, JSON, text, images. **Database / Application Access** – Data fetched programmatically via APIs or scraping, Formats: XML, JSON. **Change Data Capture (CDC) / Logs** – Data from messages, streams, or **publish/subscribe systems**. Formats: JSON, log files, event streams. **File-Based Approach** – Data stored in files for local analysis. Formats: CSV, spreadsheets; analyzed via formulas, charts, or Python/pandas.

Push: Source actively sends data to target as it happens. Difference: Source initiates transfer.Ex: Notifications, Kafka, MQTT. **Pull:** Target requests data from source when needed. Difference: Target initiates transfer, Ex: File download, database query, API call. **Poll:** Target repeatedly checks source for updates at intervals. Difference: Target checks periodically rather than continuous or on-demand., Ex: Web crawling, periodic database checks.

Bounded Data: Fixed-size, finite datasets processed as a batch.Ex: CSV files, API data pulls. **Unbounded Data:** Continuous, infinite data streams arriving over time; must be windowed or batched for processing. Ex: Sensor readings, log streams.

Throughput: Amount of data processed in a given time; high throughput ensures large volumes are handled efficiently. Ex: Processing thousands of log entries per second. **Latency:** Time delay between data generation and its availability for use; critical for real-time streams. Ex: Time from a sensor reading to dashboard update. **Scalability:** System's ability to handle increasing data volumes or bursts without performance loss. Ex: Adding nodes to a cluster to manage higher traffic

ETL (Extract → Transform → Load): Extract data from source (batch-pull or push). Transform/clean before loading. Load into storage (e.g., data warehouse)

ELT (Extract → Load → Transform): Load raw data into a data lake or warehouse. Transform/clean for each use case within the storage layer

ETL Process: Extract: Purpose: Collect data from multiple sources. Methods: Custom scripts using APIs, open-source or SaaS products. Examples: Ad platforms (Facebook Ads, Google Ads), backend databases, sales CRMs. Challenges: Requires technical skill, maintenance of extraction scripts. **Transform:** Purpose: Normalize and model raw data. Methods: Programming languages (Python, Scala), technologies (Spark, Hadoop), or GUI-based ETL platforms. Common Transformations: Rename columns, cast fields to correct types, convert timestamps. **Load:** Purpose: Load transformed data into storage like a data warehouse for BI and end-user access. Characteristics: Only transformed data resides in warehouse; supports analytics. Challenges: Must ensure transformation accuracy since raw data is not stored in the warehouse.

ETL: Pros: Ensures data is cleaned and transformed before entering the warehouse, maintaining quality. Supports complex business logic and transformations outside the warehouse. Reduces storage of unnecessary raw data. **Cons:** Requires separate transformation infrastructure, adding complexity. Slower for very large datasets due to pre-loading transformations. Less flexible for ad-hoc queries or changing analytics needs.

ELT: Pros: Leverages warehouse processing power, scalable for large datasets. Preserves raw data, allowing reprocessing or different analyses. Faster initial loading since transformations happen after loading. **Cons:** Warehouse must be powerful enough to handle transformations. Raw data storage can be large before transformation. Managing transformations inside the warehouse can be complex.

Undercurrents in Data Acquisition: **Security:** Data ingestion can expose pipelines to risks like unauthorized access or man-in-the-middle attacks. Use VPNs, private connections, or encryption in transit. Data Ethics, Privacy, and Compliance: Only collect necessary data, especially sensitive/personal information. Prefer anonymized or masked data for testing and training. **Data Management:** Handle schema changes and ensure you are notified when source data structures change. **DataOps:** Methodology to streamline and automate data pipelines, ensuring reliable, fast, and collaborative data delivery. **Data Architecture:** Blueprint for storing, organizing, and managing data to ensure consistency, scalability, and accessibility throughout its lifecycle.**Orchestration:** Schedule and manage complex data pipeline tasks as complete workflows, not individual jobs. **Software Engineering:** Use the best available tools, leveraging managed services to simplify and secure pipelines.

Data Quality Issues: Data quality depends on users; errors are common. Metadata often separate, hard to keep in sync. Backup and sharing are manual; multiple copies create redundancy. Hard to prevent inconsistent changes or maintain data integrity.

Data Quality Attributes: Believable: Trusted, credible, and regarded as true. Value-added: Provides benefits or advantages when used. Relevant: Useful and applicable for the task. Accurate: Correct, reliable, and error-free. Interpretable: Clear, understandable, and well-defined.

Data Cleaning: Missing Values: Cause: Not recorded (survey non-response, sensor faults). Impact: Can bias results if not random. Detection: Use built-in functions (e.g., Pandas). Remedy: Remove or impute carefully.

Default Values: Cause: Set values instead of blanks (e.g., "NA", -1). Impact: Can distort analysis (e.g., averaging text values). Detection: Domain knowledge, metadata, frequency checks. Remedy: Treat as missing values.

Incorrect Values: Cause: Device errors, unreliable responses, data entry mistakes. Impact: Distorts accuracy. Detection: Domain knowledge, statistical tests (e.g., Benford's law). Remedy: Remove or correct if possible.

Inconsistent Values: Cause: Free-text input variations (e.g., "Röhm" vs "Roehm"). Impact: Underrepresentation or misclassification. Detection: Cluster analysis or variant detection. Remedy: Reconcile to a single standard value.

Reading CSV in Python: The csv module has csv.reader which returns rows as lists and csv.DictReader which returns rows as dictionaries. Manual type conversion may be needed. Pandas provides pd.read_csv which reads data into a DataFrame, handles missing data, data types, filtering, and basic statistics, and is preferred for analysis.

Python Missing Values: Use the na_values parameter in pd.read_csv() to mark placeholders. After import, dropna() removes rows with missing values, fillna() fills missing values, and replace() changes specific values.**Type Conversion:** Use int(), float(), and datetime.strptime() in Python. In Pandas, astype() converts column types.**Data Visualization:** Use Matplotlib for bar, line, and scatter plots. Pandas also allows quick plotting directly from DataFrames or Series.

DBMS: is software that stores, manages, and allows efficient retrieval, insertion, and updating of data in a structured way. Pros: Centralized management ensures data security, integrity, and consistency. Supports multiple users and concurrent access efficiently. Cons: Can be resource-intensive and complex to administer. May require specialized knowledge to design and maintain.

Relational Database (Structured Data): A type of DBMS that stores data in tables (relations) with rows (tuples) and columns (attributes). Each table has a schema defining its structure and data types, and tables can be related via keys. Pros: Ensures data integrity and consistency through constraints and ACID transactions. Powerful querying with SQL, supporting complex joins and aggregations. Cons: Rigid schema; changing structure requires altering tables. Scaling horizontally is difficult; large joins can impact performance.

Primary Key: A unique attribute or combination of attributes that identifies each row in a table. Cannot be NULL. **Foreign Key:** An attribute in one table that refers to the primary key in another table, creating a link between the two tables.

Schema Diagrams: Graphical representations of a relational database showing tables, columns, and relationships. Normalised schemas avoid data redundancy by storing each fact only once. Includes notations like Entity-Relationship Diagrams (ERDs).

Schema Design: Automatic schema generation: Creates tables from DataFrames or CSVs but usually lacks foreign keys, constraints, and proper normalization. Manual schema design: Define tables with SQL DDL (CREATE TABLE) to control data types, relationships, and integrity; ensures consistency but makes loading raw data harder. Practical approach: Use a normalized schema with code-based loading to balance control and ease of use.

Relational vs NoSQL: Relational uses fixed schemas; NoSQL allows flexible or dynamic schemas. Relational stores structured data in tables; NoSQL handles unstructured or semi-structured data. Relational enforces ACID transactions; NoSQL often prioritizes scalability and eventual consistency.

Analytical vs Operational: Analytical databases are optimized for complex queries and reporting; operational databases handle real-time transactions. Analytical stores historical or aggregated data; operational stores current transactional data. Analytical focuses on read-heavy workloads; operational focuses on frequent inserts, updates, and deletes.

Commercial vs Open-Source: Commercial databases require licenses and paid support; open-source databases are free and community-supported. Commercial offers proprietary features and enterprise optimizations; open-source relies on community contributions. Commercial provides enterprise-grade security and SLAs; open-source may need extra setup for similar guarantees.

Disk-Based vs Main-Memory vs Cloud: Disk-based stores data on disks; main-memory stores in RAM for faster access; cloud-based is hosted remotely and scalable. Disk/main-memory rely on local infrastructure; cloud provides managed services and high availability. Disk-based is slower for large queries; main-memory is faster but limited by RAM; cloud offers flexible scaling.

Querying PostgreSQL from Python: Use a connection (conn) to execute SQL statements. Pandas can load query results into a DataFrame using pd.read_sql(). Use a utility function, for example query(), to distinguish between commands and queries that return data.

Bulk-Loading Data from Storage: Load an entire table from the database into Python for processing, for example using pd.read_sql_table('measurements', conn). **Pro:** Easy to use, works across systems, and gives full control in Python. **Cons:** Cannot leverage database optimizations and must fit the data in memory. Advanced database features, such as user-defined functions or stored procedures, can improve performance but may cause platform lock-in.

Data Storage Approach 1: External GUI or command line tools: Load data directly from files using tools like psql or SQL browsers such as pgAdmin. Pros: Fast, straightforward, no programming needed. Cons: Only one-to-one CSV-to-table mapping, stops at the first error, no data transformation.**Approach 2:** Programmatically using SQL: Generate SQL INSERT statements via scripts or code, for example Python, to load data. Pros: Flexible, allows data cleaning and transformations, can be optimized for the use case. Cons: Requires development time; schema or transformation changes must be handled manually.**Approach 3:** Pandas loading code: Use a Pandas DataFrame and the to_sql() function to load data into a database. Pros: Flexible, integrates Pandas data cleaning and transformations. Cons: Needs custom coding; setup of tables and constraints may be required.

Operational Systems (OLTP): Purpose: Handles day-to-day transactions, Data Volume: Small, current data, Query Type: Simple, short queries, Updates: Frequent inserts, updates, deletes, Design: Highly normalized. Example: Bank account transfer: Start transaction, check sender's balance, Debit sender's account, Credit receiver's account, Commit transaction or rollback if error.

Analytical Systems (OLAP): Purpose: Supports analysis and decision-making, Data Volume: Large, historical data, Query Type: Complex, long queries, Updates: Mostly read-only, Design: Denormalized for faster querying. Example: Retail sales reporting: Extract sales data for the past year, Transform: group by product category, Aggregate: calculate total revenue per category, Sort results by revenue, Load into reporting dashboard

OLAP Operations: Roll-up: Summarize data along hierarchy (e.g., city → country). Drill-down: View more detailed data (e.g., year → month). Slice: Select a single dimension (e.g., products × cities for January). Dice: Select multiple dimensions to create a smaller cube. Pivot: Rotate dimensions for a new perspective (e.g., switch rows/columns).

Data Warehouse: A separate database for analytics that combines data from multiple operational systems to solve data silos. Uses schemas (star, snowflake) optimized for analytical queries, allowing complex analysis without affecting OLTP operations. **Common Issues:** Semantic Integration: Resolve mismatches like different units, currencies, or schema names. Heterogeneous Sources: Consolidate data from varied formats, DBMSs, and repositories. Load, Refresh, Purge: Periodically update data and remove outdated records. Data Cleaning: Eliminate errors and inconsistencies before loading. Metadata Management: Track sources, loading times, and descriptive info for all records.

Database (OLTP): Purpose: Handles day-to-day transactions, Data: Current, operational data, Queries: Simple, short, frequent, Design: Highly normalized, Impact: Affects live operations.

Data Warehouse (OLAP): Purpose: Supports analytics and decision-making, Data: Historical, aggregated data from multiple sources, Queries: Complex, long-running, Design: Star/snowflake schemas, denormalized for analysis, Impact: Does not affect operational systems.

Fact Table: The central table in a data warehouse storing numeric measures for analysis. Each measure is linked to dimensions via foreign keys. Example: Sales amount by Market_ID, Product_ID, Time_ID.

Dimension Table: Holds descriptive attributes of dimensions, often with hierarchical levels for aggregation.Ex: Market → City, State, Region; Product → Name, Category, Price; Time → Year → Quarter → Month → Week → Date.

Star Schema: A schema with the fact table at the centre and dimension tables surrounding it. Enables efficient analytical queries by joining measures with descriptive attributes. **Snowflake Schema:** A schema where dimension tables are normalized into multiple related tables around the central fact table. Reduces redundancy but makes queries slightly more complex.

Data Lake: A centralized repository that stores raw, structured, semi-structured, and unstructured data in its native format until needed for analysis. Pros: Can handle large volumes and diverse types of data. Flexible for multiple analytics and AI/ML use cases. Cons: Harder to enforce data quality, governance, and security. Query performance can be slower than structured warehouses.

Data Lakehouse: data architecture that combines elements of data lakes and data warehouses. It stores raw, semi-structured, and structured data like a data lake, but also supports structured querying and analytics like a warehouse. Provides flexibility, scalability, and analytics on all types of data in one platform.

Key Difference: Warehouse: structured and processed before storage. Lake: raw and unprocessed, structured only when read. Data Warehouse: Stores structured, cleaned, and processed data. Optimized for analytics and reporting using predefined schemas. Supports fast, complex queries but less flexible for new data types. **Data Lake:** Stores raw, structured, semi-structured, and unstructured data. Schema-on-read, flexible for multiple analytics and machine learning use cases. Can handle large volumes but may have slower query performance and harder governance.

Web API (Application Programming Interface) is a set of rules and protocols that allows applications to communicate over the internet. It enables one software system to request data or services from another using standard web protocols like HTTP.

Web Scraping: The process of extracting data from websites, either by downloading files or parsing data embedded in web pages.**General Approach:Reconnaissance:** Identify the source, structure, and terms of service.**Webpage Retrieval:** Download pages, often using scripts that generate URLs.**Data Extraction:** Parse content and extract raw data.**Cleaning and Transformation:** Convert data into the required format.**Storage and Analysis:** Save or combine with other datasets.**Notes:** Web pages are written in HTML, which may be semi-structured or poorly formatted. Browsers are forgiving with HTML, but parsers may fail on incorrect markup.**Tools and Libraries:** Unix commands (curl, grep, awk), Pandas (read_csv), BeautifulSoup, Scrapy, Import.io, Google Sheets ImportHTML, and commercial web-crawling services.

Web Scraping Tips: Pay attention to URL formats and parameters; they may be looped over for multiple datasets (e.g., monthly or yearly files). Look for patterns in links to automate data access efficiently. Check for access tokens or API keys if required. Inspect the web page structure to locate the desired elements. Use browser page inspector to identify HTML elements for extraction. Complex structures may require tokenizing child nodes, which can be handled with libraries like BeautifulSoup in Python.

Web Page Retrieval in Python: Request Types: GET – fetches a webpage, optionally with parameters. POST – sends data to a web form or endpoint. Python Requests Library: Simple page: requests.get (URL). Page with parameters: requests.get (URL, params=key:value). POST form: requests.post (URL, params=key:value)

URLs: Uniform Resource Locator, which is the address of a resource. Format: protocol://site/path_to_resource. Protocol can be http, https, or ftp. Site is the domain or IP (optional port). Path is the resource location and may include query parameters.

Website Crawling: Scrapy: Python framework for multi-page crawling (spiders follow links and extract data). Selenium: Programmable browser to simulate user actions, useful for interactive pages and running JavaScript.

Selecting Content in a Webpage: Text Patterns: Simple string matching; limited for complex structures. DOM Navigation: Traverse the document object model to find elements. CSS Selectors: Use tags, classes, and IDs to select elements; easy but depends on consistent CSS usage. XPath Expressions: Powerful way to navigate the document tree and select nodes or subtrees; supports filters on attributes and values.

CSS Selectors:Class: `table.data` selects elements with class `data`.**ID:** `#results` or `div#results` selects the element with ID `results`.**Position:** `:first-child`, `:last-child`, `:nth-child(n)`, `:nth-of-type(n)` select elements by order.

Web API Programming: Getting Data via APIs: Many websites provide APIs to request data programmatically. Common formats: JSON and XML.

Web Service Standards: RESTful APIs: Stateless operations using standard HTTP methods (GET, PUT) with responses in HTML, XML, or JSON. XML Web Services (SOAP): XML-based messaging standard, more complex and powerful than REST.

JSON: Purpose: Lightweight, semi-structured data exchange; widely used in APIs. Structure: Nested key–value pairs, arrays, and objects. Syntax: → objects, [] → arrays; flexible and easy to parse. Example: "name": "John", "age": 21, "courses": ["Math","Physics"] → easy to store or transfer data.

HTML: Purpose: Display content in a browser; focuses on layout and presentation. Tags: Predefined (e.g., ¡p¿, ¡div¿, ¡img¿). Structure: Not strict; mainly for human-readable pages. Example: ¡p¿Welcome to the website! ¡/p¿ → shows a paragraph on a webpage.

XML: Purpose: Store and exchange data; machine-readable. Tags: User-defined (e.g., ¡student¿, ¡order¿). Structure: Strict hierarchy; must be well-formed; can be validated with DTD/schema. Example: ¡student¿¡name¿John¡/name¿¡age¿21¡/age¿¡/student¿ → structured data for processing.

Logical Document Structure (XML): XML data is made of elements. The highest-level element is the root element. Elements are wrapped in opening and closing tags. Tags are case-sensitive. Tags cannot overlap (strict tree structure). Invalid: ¡A¿…¡B¿…¡/A¿…¡/B¿ Empty elements use a shorthand form: ¡AUD/¿. XML documents always form a tree, not a graph. Ex: ¡book¿ ¡title¿Autobiography of Benjamin Franklin¡/title¿ ¡price¿ ¡AUD/¿8.99 ¡/price¿ ¡/book¿.

Document Type Definition (DTD): XML lets you invent your own tags, but a DTD defines which tags are allowed. A DTD acts like the grammar of the XML document. It can specify: Which elements exist Their allowed structure Attributes and their types Which elements must/may appear. You can link a DTD to an XML document so the document can be validated against it. Well-formed XML: Follows XML syntax rules (proper nesting, matching tags, one root). Valid XML: Well-formed and follows a DTD/schema.

How to Query or Filter XML: DOM navigation: Walk through the XML tree programmatically using the Document Object Model. XPath: A concise way to select nodes, attributes, or entire subtrees inside one XML document. XQuery: Built on XPath; a full query language for querying multiple XML documents. CSS selectors: Mainly for HTML, not general XML.

XPath Document Model (Tree View): XPath treats an XML document as a tree. A special synthetic root node sits above the document's actual root element. Internal nodes = XML elements.

Semi-Structured Data: Data that does not follow a fixed schema but carries tags or keys to describe its structure. It can have nested, hierarchical (tree-like) organization and mixed data types. **Pros:** Flexible schema allows records with varying fields. Supports nested or hierarchical data without complex joins. **Cons:** Less strict structure can lead to inconsistent data if not managed carefully. Querying can be less efficient than in structured relational tables.

Semi structured Data in Databases: Relational DBMS Support – Some SQL databases like PostgreSQL can store and query JSON and XML data natively. XML Support – Uses SQL/XML standard; provides XML data type and functions for creation and querying (XPath/XQuery). JSON Support – PostgreSQL offers JSON (exact copy) and JSONB (binary, optimized for faster queries). Queries can extract and filter nested data. NoSQL Databases – Databases like MongoDB are designed for JSON storage and retrieval, handling flexible, nested structures efficiently. EX: JSON: "name": "Bob", "degree": "BSc (CS)", "courses": [...] stored in a JSON/JSONB column.

NoSQL: Designed for scale-out on many simple servers; cloud-friendly. Simpler data model and queries; weaker consistency and integrity. Often open-source, less resource-intensive than traditional SQL DBs. Modern NoSQL added some SQL features; traditional DBs added support for diverse data. Ex: MongoDB, Redis. **Pros:** Highly scalable; optimized for horizontal scaling across servers. Flexible schema supports evolving or unstructured data. **Cons:** Weaker consistency guarantees (eventual consistency in many systems). Complex queries and joins can be less efficient or require workarounds.

NoSQL Types: Document Stores – Nested key-value pairs, e.g., MongoDB, XML databases. Column Stores – Large-scale column-oriented storage, e.g., BigTable, HBase. Key-Value Stores – Simple key-value mapping, e.g., Redis, DynamoDB, Cassandra. Graph Databases – Nodes and relationships, e.g., Neo4J. No standard model or API across NoSQL systems.

Schema-on-Read: Data can be ingested without predefined schema. Structure is interpreted when reading/querying. Flexible; good for semi-structured or evolving data (JSON, XML). Example: MongoDB storing different JSON documents in the same collection.

Schema-First: Schema must be defined before inserting data. Structure is fixed and enforced by the database. Ensures strong integrity and consistency. Example: Relational databases like PostgreSQL or MySQL.

MongoDB: Flexible schema; documents can differ in structure. Collections correspond to tables, documents to rows, and fields to columns. Each document has a unique _id (user-defined or auto-generated). Relationships are managed via embedded sub-documents or references. **Example:** A student document contains name, age, degree, and courses all in one document.**Pros:** Flexible schema, easy to handle evolving or semi-structured data. Supports nested documents and arrays, reducing the need for joins.**Cons:** Complex relationships can be harder to query efficiently. Weaker consistency guarantees, with eventual consistency in sharded setups.

Querying and Operations: Create: insert JSON documents into collections. Read: find () with filters, projections, sorting; supports predicates (gt,or, etc.). Flexible Queries: implicit equality and conjunction; can combine conditions. Ex: Document: "name": "Sue", "age": 26, "status": "pending". Query: Find students older than 20 in Sydney.

MongoDB: Aggregations: Pipeline of stages for filtering, grouping, sorting, computing. Combines complex operations in one flow. Used for analytics and reporting. **Scalability and Architecture:** Auto-sharding: partitions collections across servers. Single-leader replication for updates; followers serve reads (eventual consistency). Load balancing and high availability via distributed architecture. Ex: Aggregate sales totals per region; student data split across shards.

Graph Database: Stores data as nodes (entities), edges (relationships), and properties (attributes). Optimized for complex, many-to-many relationships. Schema-less or flexible; allows evolving data structures. Ex: Users and their friendships in a social network; nodes = users, edges = friendships. Properties: Attributes or metadata

Neo4j (Graph Database): Property graph model: nodes (entities), edges (relationships), and properties (attributes). Flexible schema: nodes and edges can have different properties. Features include ACID compliance, high availability, and the Cypher query language. Advanced features: indexing, full-text search, graph algorithms, and visualization using Neo4j Bloom.**Example:** `(:Person)-[:ACTED_IN]->(:Movie)` represents actors in movies. **Pros:** Optimized for complex, many-to-many relationships and fast graph traversals. Flexible schema allows evolving data, and built-in graph algorithms support analytics **Cons:** Not ideal for simple, tabular data—overkill for basic CRUD tasks. Can become resource-intensive with very large graphs if not properly indexed or sharded.

Managing Python Packages: Install a package using `pip install package_name`. Save dependencies with `pip freeze > requirements.txt`. Install from requirements using `pip install -r requirements.txt`.**Usage Scenario:** Python-only projects, simple dependency management, quick prototyping, and small applications. **Best Practices:** Use a separate virtual environment for each project. Keep `requirements.txt` updated. Avoid installing project packages globally.

Spatial Data: Definition: Data about objects or entities that have a location or geometry. Geospatial Data: Specifically identifies geographic locations or boundaries on Earth (e.g., cities, suburbs). Example: OpenStreetMap API provides JSON data with latitude, longitude, bounding boxes, and place details.

SDBMS (Spatial Database Management System): Stores large amounts of spatial data. Built-in spatial semantics in queries. Specialized indexing for fast spatial access. GIS (Geographic Information System): Client of SDBMS, focused on analysis and visualization. Organizes data in layers (roads, stations, suburbs). Allows exploration and integration across layers; relies on SDBMS for large datasets.

Spatial Data Types: Point Data: Represents individual points in space. Examples: GPS locations from apps, raster pixels in satellite images, feature vectors from text. Region Data: Represents objects with spatial extent (area and boundary). Typically stored as vector data using polygons, line segments, etc.

Models of Spatial Information: Object-Based (Entity-Based) Model: Represents distinct, identifiable objects relevant to an application. Objects have attributes (spatial: location; non-spatial: name, type, etc.) and operations (functions on attributes). Example: Road objects with location, type, number of lanes; operation: calculate length or intersection. Field-Based (Space-Based) Model: Represents spatial phenomena as continuous fields over space. Example: Forest stand map viewed as continuous field rather than individual polygons.

Classifying Spatial Objects: Simple Objects: Point (0D): Road intersection Curve (1D): River Surface (2D): Country. Collections: Polygon collections, e.g., boundaries of Japan or Hawaii.

Coordinate Systems and SRID: SRID: Spatial Reference Identifier; ensures spatial data is interpreted in the correct coordinate system. Types of Coordinates: Cartesian: 2D plane positions from a defined origin. Geographic/Geodetic: Latitude and longitude. Geocentric: 3D Earth-centered Cartesian coordinates. Projected: Planar coordinates from mapping Earth to a plane. Projections: Flattening the Earth introduces distortions (e.g., Mercator projection exaggerates high latitudes).

WGS84: Standard geodetic datum used by the global GPS system. **GDA94:** Official geodetic datum for Australia, based on ITRF but fixed to reference points in Australia. **Differences:** Both are geocentric and use almost identical spheroids, so coordinates are mostly interchangeable for mapping and GIS. For precise surveys, small differences exist, approximately 45 cm in 2000, and change slowly over time.

PostGIS: PostgreSQL extension for spatial data. Geometry: planar shapes; Geography: spherical, large-scale. Supports spatial functions (area, distance, intersection) and R-Tree indexing. Import/export via GeoJSON, KML; coordinate transformations supported.

Spatial Operations: Set-Based: union, intersection, containment. Topological: touches, disjoint, overlap. Directional: north, east, etc. Metric: distances, lengths.

Topological Relationships: Relationships unchanged under deformation (no tearing/merging). Interior, boundary, exterior define object space. Queries: e.g., relationship between A and B, or find objects related to A.

Nine-Intersection Model: 3×3 matrix of intersections (interior, boundary, exterior of A and B). Values: 0 = empty, 1 = non-empty. Defines 8 possible 2D relationships for objects without holes.

DuckDB: is a high-performance, in-process SQL database for analytics and data science, supporting columnar storage and large datasets without a separate server. **Pros:** Fast and efficient for large datasets. Easy integration with Python, R, and columnar formats **Cons:** Limited multi-user support. Lacks advanced RDBMS features (triggers, extensions).

GeoPandas: Python library for geographic data, building on Shapely, Fiona, and Geos. Reads/writes various geo-formats (Shapefiles, MapInfo TAB, etc.). Install via conda install geopandas. GeoPandas Data Types: GeoSeries: Series of geometric objects (Point, Line, Polygon, MultiPoint, MultiLine, MultiPolygon). GeoDataFrame: Table-like structure built on GeoSeries, allows attributes + geometry.

GeoSeries: Attributes include `area`, `bounds`, `total_bounds`, `geom_type`, and `crs`. Methods include `distance()`, `centroid`, `to_crs()`, and `plot()`. Relationship tests include `contains()`, `intersects()`, and others.**Coordinate Reference System (CRS):** Defines how coordinates relate to Earth. Commonly used CRS is WGS84 (epsg:4326). You can set the CRS using `set_crs()` or reproject using `to_crs()`.

GeoDataFrame: Combines tabular data with one special geometry column (GeoSeries). Spatial methods act on the geometry column. Supports attribute joins (merge ()) and spatial joins (sjoin ()).

Plotting: Use the `plot()` function for GeoSeries or GeoDataFrame. You can customize colors, markers, legends, and overlay multiple layers (all layers must have the same CRS).**Reading GeoSpatial Data:** Shapefiles, MapInfo, and GeoJSON can be read using `read_file()`. For PostGIS, use `read_postgis()`, which requires a database connection.

Geo-aware Web APIs: RESTful APIs allow fast responses for modern and mobile applications. Can return spatial data in JSON or XML formats. Useful for location-based services like maps, navigation, and local search.

Spatial Data Exchange Formats: GeoJSON: JSON-based format for geographic data; supports points, lines, polygons, and multi-geometries, along with properties. Example: Representing a location with coordinates and a name. KML (Keyhole Markup Language): XML-based format for maps, placemarks, polygons, and 3D models; used by Google Earth and mapping software. Example: Describing a city's location and details for map visualization. OSM XML: OpenStreetMap standard for exchanging map data. Other OGC standards: WMS (Web Map Service), WFS (Web Feature Service), WCS (Web Coverage Service). Proprietary formats: MapInfo TAB, ESRI Shapefiles.

Temporal Data: Data marked with time, like sensor readings, medical records, or transactions. Helps analyze trends and sequences. Historic ("Long") Data: Examining data over long periods (10–30 years) reveals patterns and insights missed by short-term analysis.

Temporal database: is a database that tracks data with respect to time, storing historical, current, and future information, and supports time-based queries. **Pros:** Enables historical analysis and auditing. Supports reasoning with past, present, and future data. **Cons:** More complex to design and maintain than conventional databases. Can have higher storage and processing overhead.

Temporal Data Types: Instant types: Represent a single point in time. DATE: Day, month, year. TIMESTAMP: Date + time with fractions of a second. TIME: Hours, minutes, seconds (no date). Interval types: Represent a duration of time. Example: Year-Month intervals (e.g., 1 year 2 months). Period Data Type: Represents a time span with a start and end (same type). Includes start, excludes end. Used for validity or active periods.

Kinds of Time: User-defined time: simple, uninterpreted value (e.g., birthdate). Valid time: when a fact is true in reality; can move forward/backward. Transaction time: when a fact is stored in the DB; only moves forward, enables rollback/auditing.

Temporal Tables: Store and manage data with respect to time, supporting time-based queries. Include transaction-time and/or valid-time columns, typically using period data types. Types: Transaction-time table has a transaction-time column. Valid-time table has a valid-time column. Bitemporal table has both transaction-time and valid-time columns. Nontemporal table has neither column.

Basic Representation: SQL provides DATE, TIME, and TIMESTAMP types. Time is treated as a set of instants (points) with a partial order. Time attributes can be added to any table; interpretation is application specific. Examples: Person (id, name, birthday DATE) → stores birthdates. Project (id, title, start DATE) → project start dates. TripLog (user, car, when TIMESTAMP, distance) → logs trips at specific times.

Operations: Comparisons include equal, less than, greater than, less than or equal to, and greater than or equal to. Constants include `CURRENT_DATE` and `CURRENT_TIME`. To extract components, use `EXTRACT(year FROM date)`. Arithmetic with intervals can be done by adding intervals to dates, for example, adding 36 hours to a date.

Semi-Temporal Tables: Add a since timestamp to track when a fact became true. Example: Employees (tfn, name, city, since) → [since, now) represents current validity.

Valid-Time Tables / History: Track periods of validity using two DATE columns, `start_date` and `end_date`. For example, in `EmpPosition (tfn, pos, start_date, end_date)`, intervals defined by `start_date` and `end_date` capture role history. This approach allows SQL databases to manage current facts, semi-temporal data, and full historical tracking even without a native period type.

PostgreSQL Range Indexes: GiST indexes Generalised Search Tree speed up queries on ranges (e.g., periods) and combinations with other attributes. Useful for predicates like EQUAL, OVERLAP, CONTAINS. Helps efficiently query temporal or spatial data.

Timeseries Data: Sequential observations recorded at discrete, usually equal, time intervals. Examples: Weather data, trip logs, road usage, sleep patterns, cytometry/panel data. Workflow: Ingress: File import, database access, web scraping. Storage: Standard file formats or databases. Analysis and Reporting: In-database or via applications (e.g., Python), plus visualization.

Timeseries Data Storage: Approaches include point-based storage, where each row represents one time point. This is simple, easy to compare and index, and may require multiple rows if values repeat. Sequence-based storage uses a single row to hold arrays of time points, which requires array operations such as `UNNEST` and `array_agg`. Dedicated time-series databases include TimescaleDB, InfluxDB, and SciDB.

Point-Based Representation: Stores each timestamp as a separate, atomic row. Supports comparisons, filtering, aggregation, and joins efficiently. Ideal for OLAP queries and analytics. Uses B-Tree indexing for fast access to individual time points. Working: Query a specific timestamp or range easily, e.g., sum of sales per day.

Sequence-Based Representation: Stores multiple time points as an array in a single row, which violates strict 1NF. Time points can be explicit, meaning stored timestamps, or implicit, meaning regular intervals. Supports array operations such as `array_agg()` and `UNNEST()`, and allows bulk updates. Uses GIN indexing for fast searches within arrays. For example, daily temperatures for a month can be stored compactly in one row.

Text Data: Text usually has no fixed structure. It's unmodelled, messy, and full of ambiguous elements like dates, numbers, and names, which makes it harder to process than structured database data.

NLP and Feature Extraction: NLP tries to interpret the meaning of text by analysing how words and sentences relate. It's used for tasks like sentiment analysis, topic modelling, translation, and entity recognition. Traditional ML models need numeric features, so text is converted using methods like Bag-of-Words, TF-IDF, word embeddings, or other feature representations.

Image Data Basics: Images can be vector or raster. Raster = a grid of pixels. Each pixel stores brightness/color info. Color depth = how many color values a pixel can represent. Raster images come from cameras, scanners, sensors, medical devices, etc. Types of Images: RGB / TrueColor: 3 channels: Red, Green, Blue. Typically 24-bit (8 bits per channel) $\longrightarrow$ 16.7 million colors Grayscale: 1 channel. Each pixel is a shade of gray. 8-bit = 256 shades, 16-bit = 65k shades. Binary: 1 channel, only 0 or 1 (black or white). Often used as masks.

Supervised Learning: Uses labelled data (ground truth). Goal is to predict an output for new, unseen data. Typical tasks: classification and regression. Classification: predict categories (e.g., spam/not spam). Regression: predict numeric values (e.g., price, temperature).

Unsupervised Learning: Works with unlabelled data. Goal is to discover structure or patterns in the data. Typical tasks: clustering, distribution estimation, dimensionality reduction. Clustering – Grouping data points into similar clusters without any labels. Estimating distributions – Learning the underlying probability distribution of the data. Association patterns – Finding rules that show how features tend to co-occur. Dimensionality reduction – Compressing data by keeping only the most important features.

Unstructured Data Analysis (general flow): Data acquisition – collect raw text/images/audio. Preprocessing – clean and format it. Feature extraction – convert raw data into useful numerical features. ML tasks – classification, similarity search, object detection, sentiment analysis, etc.

Feature Extraction: The process of transforming raw data into meaningful input variables (features) for machine learning. Helps models learn patterns more effectively by summarizing, encoding, or deriving relevant information from the original data.

Text $\longrightarrow$ Feature Vectors: Word Embeddings – Models like Word2Vec, GloVe, or BERT map words into dense numeric vectors that capture semantic meaning. TF-IDF – Converts text into sparse vectors based on how often words appear in a document vs. across the whole corpus. Bag-of-Words basics – Tokenise text, count word frequencies, ignore grammar and order. Feature Vector meaning – A list of numbers showing which words appear and how often.

Tokenisation: Clean and normalise text (lowercase). Remove stop-words. Apply stemming/lemmatisation to reduce words to their base form. Output is a list of tokens ready for BOW or TF-IDF.

Vector Space Model: Each document becomes a vector where each dimension represents a token (usually stemmed). Values are term frequencies or TF-IDF weights. IDF reduces weight for very common words to highlight more informative ones.

TF Feature Extraction (Python, scikit-learn): Converts text into numeric vectors representing word occurrences. Can handle unigrams, bigrams, lowercasing, stop-word removal, and frequency-based filtering. Produces structured features from raw text for machine learning.

BERT (Bidirectional Encoder Representations from Transformers): Bi-directional – considers context from both left and right. Designed for understanding text (classification, QandA, sentiment analysis). Pre-trained with Masked Language Modeling (predicts masked words). Uses encoder-only transformer architecture.

GPT (Generative Pre-trained Transformer): Uni-directional – reads text left-to-right. Designed for text generation (completion, conversation, story writing). Pre-trained with Autoregressive Language Modeling (predicts next word). Uses decoder-only transformer architecture.

Feature Extraction in Images: Extract patterns, color info, or metadata to represent images as vectors. Enables mathematical operations like similarity comparison. Image Similarity Search: Convert images to feature vectors, index them, and compare vectors to find visually similar images.

Feature Extraction Methods: White-box algorithms (Image descriptors): Use gradients or intensity/color changes. Black-box algorithms (Neural networks): Learn complex features automatically Python Support: Libraries like skimage help implement feature extraction.

Image Data Analysis Outcomes: Classification: Categorize images based on features. Image Similarity Search: Find visually similar images using feature vectors. Object Detection: Locate and identify objects within images. Intensity-Based Analysis: Measure brightness or color intensity patterns. Data Aggregation: Combine results from multiple images for raw data extraction.

Meta Data Analysis: Unstructured data like images, videos, and PDFs often include metadata. Common formats: EXIF, XMP (standards by iptc.org). Typical information: capture time, device settings, GPS location, author, copyright. Tools: GUI (Photoshop, Lightroom), web (metadata2go.com), command-line (exiftool). Uses: key-value extraction, geolocation, author info, keywords. Without metadata, analysis must rely on the actual content, which is more complex.

Data Streams: continuous, potentially unbounded sequence of rapidly changing data tuples. Transactional streams: Track events or interactions, e.g., credit card purchases, manufacturing steps, supply chain logs. Measurement streams: Monitor evolving states, e.g., network traffic, sensor readings, road conditions. Widely used in real-time monitoring and analytics applications. Stream Processing: "Data in motion" – data is processed as it flows, without persistent storage.

Data Stream Approach: Driven by applications and technology for real-time analysis. Handles large volumes of transactional and measurement data. Uses DSMS to process streams continuously. Continuous queries return incremental results as data flows.

Publish/Subscribe: Messaging pattern where publishers send messages to a broker under specific topics. Subscribers receive messages by subscribing to those topics. Decouples senders and receivers: publishers don't send directly to subscribers. Common in real-time systems for scalable, flexible communication.

Apache Kafka: A distributed streaming platform for building real-time data pipelines and streaming applications. It handles high-throughput, fault-tolerant message storage and processing. Pros: Extremely high throughput and low latency. Durable and fault-tolerant with distributed log storage. Cons: Complex to set up and manage at scale. Not ideal for complex event processing by itself (needs integration with tools like Flink/Spark).

MQTT (Message Queuing Telemetry Transport): A lightweight messaging protocol for IoT and constrained devices, using a publish/subscribe model. Pros: Very lightweight; ideal for low-bandwidth, low-power devices. Simple publish/subscribe architecture for easy IoT integration. Cons: Not designed for high-volume big data scenarios Minimal built-in message durability and persistence (depends on broker).

Apache Hive: A data warehouse system built on top of Hadoop for querying and managing large datasets using a SQL-like language (HiveQL). Pros: Simplifies big data querying with familiar SQL syntax; integrates with Hadoop ecosystem. Cons: Not ideal for real-time processing; query latency can be high for very large datasets.

Hadoop: An open-source framework for distributed storage and processing of large datasets across clusters of computers. Pros: Handles massive data volumes; fault-tolerant via data replication. Cons: Batch-oriented (not real-time); complex to configure and manage.

MapReduce: A programming model and processing engine in Hadoop for parallel computation on large datasets. Pros: Automatically handles parallelization, fault tolerance, and load balancing; scalable. Cons: High latency for small jobs; less suitable for iterative or real-time processing

Apache Flink: A stream-processing framework for real-time analytics, supporting event-time processing, windowing, and stateful computation. Pros: True real-time streaming with low latency. Advanced features like exactly-once processing and state management. Cons: Steeper learning curve than batch frameworks. Resource-intensive for very large clusters.

Apache Spark: A unified analytics engine for large-scale data processing, supporting batch, stream, machine learning, and graph processing. Pros: Supports both batch and stream processing (via Spark Streaming). Rich ecosystem for ML, graph analytics, and SQL queries Cons: Spark Streaming is micro-batch, not true real-time. High memory usage; can be resource-heavy for small clusters.

Apache Parquet: A columnar storage file format designed for efficient data storage and retrieval in big data processing frameworks. Optimized for analytical workloads and large-scale data processing. Pros: Highly efficient compression and encoding, reducing storage and I/O. Excellent performance for column-based analytics and queries. Cons: Not ideal for frequent row-level updates or transactional workloads. Requires compatible processing frameworks to fully leverage its benefits.

Stream Query Processing: Challenge: Streaming data is unbounded and constantly in motion, unlike static datasets assumed in traditional query planning and execution. Continuous Queries: Queries are executed continuously as new data arrives. Processing Steps: Parsing $\longrightarrow$ Planning $\longrightarrow$ Optimization $\longrightarrow$ Execution.

Stateful: Retains information about past interactions; requires session management, more memory, and careful development. Less scalable and fault tolerant. **Stateless:** Treats each request independently; simpler, scalable, more fault-tolerant, and resource-efficient.

Windowing Technique: Windows are defined using stream attributes like timestamps, sequence numbers, tuple counts, or explicit markers. Converts infinite data streams into finite segments for real-time processing. **Types:** Ordering-Attribute Windows – Based on an attribute that defines tuple order. Sliding Window: Overlapping, continuously moving segments. Tumbling Window: Non-overlapping, fixed-size segments. Tuple-Count Windows – Contains a fixed number of tuples; may produce non-deterministic results if timestamps repeat. Marker-Based Windows – Defined by explicit events or markers in the stream Variants: Tuples can be partitioned within a window for finer-grained analysis.

Event Time: Timestamp assigned to events, independent of processing (wall-clock) time. **Challenge:** Out-of-order and late-arriving events. **Watermarks:** Progress indicators in event time; declare that all events up to a certain timestamp are expected to have arrived. Usage: Operators advance their event time clocks on receiving watermarks; late events can be discarded or handled separately. EX(Flink): Tumbling event-time windows with allowed lateness; watermarks generated periodically to handle out-of-order events.

Stream-Table Join (Enrichment): Definition: Combines streaming data with static or reference tables for richer analysis. Challenge: Keeping reference data updated without affecting performance. Solution: Use in-memory storage for frequent lookups or external stores like Redis. Example: Real-time transaction stream joined with product data.

Stream-Stream Join (Windowed): Definition: Combines two streams within a defined time window. Challenge: Ensuring proper ordering and matching of records across streams. Solution: Use a fixed window (e.g., 15 seconds) and process records in timestamp order. Example: Matching sensor readings from two devices within a 15-second window.

Scale-Up: Upgrade a single machine with more CPU, RAM, or storage. Simple to implement; no changes to software architecture. Limited by maximum hardware capacity. High-end hardware can be expensive. coordination overhead

Scale-Out: Add more machines (nodes) to a cluster. Enables massive parallelism and handles big data. Requires distributed architecture and coordination. More nodes increase potential points of failure and communication overhead.

System Design Principles Loosely Coupled: Components operate independently for easier scaling. Separate Compute and Storage: Scale storage or compute independently. Availability Risk: More nodes $\longrightarrow$ higher failure probability. Latency Overhead: Communication and non-local disk access can slow requests.

Goals of Scalable System: Scale-Agnostic Data Management: Sharding: Split data for performance. Replication: Ensure availability. Transparent to applications. Scale-Agnostic Data Processing: Parallelize across hundreds or thousands of CPUs. Grow resources as needed (10×, 1000×...). Performance: Efficient parallel query processing. Availability: Handle failures transparently. Elasticity: Scale up or down dynamically. Key Paradigm: Map/Reduce

Storage Optimizations: Single Machine: Indexing: Speeds up filtering and joins. Partitioning: Split large tables across disks. Column vs Row Store: Choose based on query type. RAID: Redundant disks for performance and reliability. Cluster of Machines: Sharding: Distribute data across nodes. Replication: Ensure availability. Caching: Fast access to frequent data (e.g., memcached).

Row-Oriented Database: Definition: Stores data by records (rows). Pros: Fast for reading and writing full rows; good for transactional workloads. Cons: Less efficient for analytical queries that access only a few columns. Examples: Postgres, MySQL.

Column-Oriented Database: Definition: Stores data by attributes (columns). Pros: Efficient for querying and computing on specific columns; ideal for analytics. Cons: Slower for inserting or updating full rows; more complex storage design. Examples: Google BigQuery. Pure Column Organization: Each page stores values from a single column. Mixed Column Organization: A page can contain values from multiple columns.

Data Partitioning: Dividing a dataset into smaller subsets (partitions) stored in different locations to improve manageability, performance, and scalability. **Pros:** Easier management of large datasets. Improved availability: failure of one partition doesn't affect others. **Cons:** Added complexity in query processing. Can introduce uneven load if partitions are imbalanced. **Types of Partitioning:** Horizontal Partitioning (Sharding): Subsets of rows. Vertical Partitioning: Subsets of columns. How to Partition Data: Round-Robin: Distribute partitions evenly across nodes in turns. Hash Partitioning: Assign partitions based on a hash function of a key. Range Partitioning: Assign partitions based on value ranges of a key.

Data Replication: Storing copies of the same data at multiple locations to improve availability and reliability in distributed systems. Pros: Increased availability; if one site fails, another replica can be read. Load balancing for faster read queries. Cons: Updates become slower, especially if replicas must stay consistent. Managing conflicts and consistency can be complex in multi-leader setups.

Replication Types: Synchronous (Eager) Replication: All replicas updated within the original transaction. Strong consistency, Slower performance. Asynchronous (Lazy) Replication: Updates applied to replicas separately. Faster performance. Possible temporary inconsistencies.

Replication Architectures: Primary-Copy (Single Leader): Only one authoritative copy is updated; replicas are read-only. Simple consistency management. Less flexible for writes. Multi-Leader: Multiple replicas can be updated; changes propagated to others. Flexible updates across sites. Conflicts may require resolution, deadlocks possible with eager replication.

How It Works: Primary-Copy Asynchronous: Capture changes in primary, then apply to secondary replicas. Multi-Leader Asynchronous: Updates can happen anywhere; conflicts resolved if needed.

Goals of Distributed Data Management: Strong Consistency: Every read reflects the most recent write; all copies appear as a single, up-to-date dataset. High Availability: The system remains operational even if some nodes fail. Partition Tolerance: System continues functioning despite network splits or communication failures.

CAP Theorem: A distributed system can provide at most two of the following three guarantees: Consistency (C): Every read sees the most recent write. Availability (A): Every request receives a response, even if some nodes fail. Partition Tolerance (P): The system continues operating despite network splits. Common Trade-offs: CP: Prioritizes correctness over availability (e.g., financial systems). AP: Prioritizes uptime over consistency (e.g., social media platforms).

Big Data Analytics Stack: Scale-out Infrastructure: Cluster of machines for storage and compute. Storage: Distributed storage systems (e.g., HDFS, Azure Storage). Data Processing: Frameworks for parallel processing (e.g., MapReduce, Spark). Serving / Application: Tools and APIs for querying, reporting, or serving results to applications.

HDFS (Hadoop Distributed File System): Distributed file system providing transparent access to files across multiple machines. Pros: Fault-tolerant via block replication. Handles large datasets efficiently with scale-out storage. Cons: Not suitable for low-latency, small-file access. Single NameNode can be a bottleneck.

HDFS Read Flow: Client requests file from NameNode. NameNode returns block handles and replica locations. Client caches this info. Client requests blocks from a nearby DataNode. DataNode sends the data blocks to the client.

Spark Program Pattern: Initialize runtime environment. Load or create source data. Specify data transformations (select, filter, orderBy, groupBy, join, etc.). Specify output destination. Execute

Flink Program Pattern: Initialize runtime environment Load or create source data Specify data transformations (can include UDFs) Specify output destination Execute. PySpark SQL: Definition: Module in Spark for working with structured data using SQL queries.

Data Architecture: Design decisions that shape how data is collected, stored, processed, and used to meet an organization's evolving needs. Good architecture is flexible, maintainable, and aligned with business strategy. **Aspects:** Operational Architecture: Defines what data processes are needed, data quality management, and latency requirements. Technical Architecture: How data is ingested, stored, transformed, and served (the data pipeline). Principles of Good Architecture: Design for automation Plan for availability/failure Prioritize security and privacy. Architect for scalability (favor loosely coupled systems). Continuously evolve architecture Examples: **Modern Data Stack**: Cloud-based, modular, plug-and-play tools for pipelines, storage, transformation, governance, monitoring, visualization, and exploration; emphasizes self-service and agility.

Data Streaming Architectures Lambda Architecture: Purpose: Reduces latency in batch-oriented systems by combining batch and real-time processing. **Layers:** Batch Layer (Cold Path): Stores all raw data; performs batch processing $\longrightarrow$ batch views. Speed Layer (Hot Path): Processes data in real time; low latency but may be less accurate. Serving Layer: Merges batch and real-time views for analytics and client queries. Kappa Architecture: Purpose: Simplifies streaming by using a single stream-processing path for all data. Features: Event-driven; all data flows through the same path; eliminates separate batch layer.

DataOps: Collaborative practices for automating and managing data workflows, inspired by DevOps; ensures faster delivery, high-quality data, and cross-team collaboration. **Core Components:** Continuous Deployment / Continuous Integration CI/CD): Auto-test, integrate, and deploy code/models. Pipeline Orchestration: Automates ETL and data flows. Data Quality Monitoring: Ensures accurate and reliable data. Governance and Security: Maintains compliance and data protection. Self-Service Access: Enables independent data exploration. **Challenges:** Ensuring data quality and validation Managing complex pipelines from ingestion to model training Scaling efficiently across distributed systems **Principles:** Iterative development, continuous feedback, adaptive pipelines. Benefits: Faster delivery, reliable data, better collaboration, and regulatory compliance.

DataOps Lifecycle: Plan: Define data quality metrics and availability goals. Develop: Build data products and models collaboratively. Integrate: Connect pipelines to the tech stack for seamless operation. Test: Validate data accuracy and integrity with automated tests. Deploy: Move data models to production. Monitor: Continuously track pipeline performance and data quality.

Key Technologies: Data Ingestion: Apache NiFi, Kafka Pipeline Orchestration: Apache Airflow, Luigi. Data Transformation: Apache Spark. Data Cataloging: Alation, Apache Atlas

Automating Data Curation: Data Curation Automates cleaning, transforming, and organizing raw data. Ensures data is accurate, consistent, and usable. **Metadata Management** Tracks data lineage, transformations, and usage. Active Metadata: Updates in real time to provide insights into data assets. **Governance Automation** Enforces data quality rules and access control policies. Ensures compliance and proper data usage. **Security Automation** Applies encryption, access control, and monitoring to protect data **Master Data Management (MDM)** Maintains consistency and accuracy of critical data across systems. Automates deduplication and synchronization to create a single source of truth.

Data Observability: The Five Pillars: Freshness: Timely updates and ingestion. Distribution: Data values within expected ranges. Volume: Detect missing or incomplete data. Schema: Track structural changes. Lineage: Understand data flow and transformations. Ex: Ensuring accurate marketing attribution by tracking data lineage.

Apache MADlib: An open-source library for scalable in-database analytics, providing machine learning, statistical, and graph algorithms that run directly within SQL databases. **Pros:** Executes analytics close to the data, reducing data movement. Supports large-scale data and parallel processing inside the database. **Cons:** Limited to algorithms implemented in the library. Requires compatible database systems and setup.

Autoregressive Integrated Moving Average (ARIMA): Used to forecast future values in time series. Input is a table with timestamp, value, and optional grouping columns. Functions include `arima_train()` to train the model and `arima_forecast()` to make predictions. Parallelism is supported in Greenplum, with limited parallel support in PostgreSQL.

Feature: An input signal (feature vector) used by machine learning models to make predictions. **Feature Store:** A system for managing, storing, and serving features for operational ML, bridging raw data and model consumption.**Challenges:** Ensuring feature consistency between training and serving. Handling real-time and batch data infrastructure requirements. **Benefits:** Simplifies ML deployment into production. Centralizes feature definitions, improving reuse and reliability.

Lifecycle / Components: Transformation: Processes features from raw data (batch, streaming, on-demand). Storage: Maintains online (real-time) and offline (historical) features. Serving: Delivers features to models for training and inference. Monitoring: Tracks feature quality, correctness, and usage. Feature Registry: Central source of truth for feature definitions and metadata.

Serving Feature Data for ML Models: Consistency: Features in production must match those used during training to avoid skew. Offline Serving: Access historical features for training with point-in-time accuracy. Online Serving: Deliver fresh features in real time via low-latency APIs.

Data Storage in a Feature Store: Offline Storage: Stores large volumes of historical data for training (e.g., S3, BigQuery, Snowflake). Online Storage: Optimized for low-latency retrieval during inference (e.g., DynamoDB, Redis). Entity-Based Data Model: Each feature is linked to an entity (e.g., user) and timestamp.

Feature Transformations: Batch Transform: Applied to data at rest (e.g., historical user purchase data for marketing). Streaming Transform: Applied to live/streaming data (e.g., clicks per user in last 30 min for recommendations). On-Demand Transform: Computed at prediction time (e.g., checking user location or similarity scores for search).

Monitoring: Data Monitoring: Detect drift, ensure training-serving consistency Operational Monitoring: Track system metrics (storage availability, capacity, staleness). Benefit: Guarantees ML models use accurate, up-to-date features

Machine Learning Model Registry: Centralized Registry: Single source of truth for feature definitions, metadata, and lineage; main interface for feature store interactions. Collaboration: Teams can easily discover, share, and reuse features. Version Control: Tracks feature versions, simplifying debugging and ensuring compliance. Automation: Schedules feature ingestion, transformation, and serving jobs automatically.

Alternatives to Feature Stores: Custom ETL Pipelines: Pros: High flexibility and control; good data security for in-store ML. Cons: Complex; inconsistent feature serving across teams. Data Warehouses and Lakes: Pros: Integrates with existing infrastructure (e.g., S3). Cons: Not optimized for real-time ML; usually lacks native feature transformations. In-House Feature Management Systems: Pros: Fully customizable for business needs. Cons: High maintenance; scalability challenges as teams grow.

Agile: A project management approach that breaks work into small, iterative cycles called sprints, encourages continuous feedback, collaboration, and quick adaptation to changes. It focuses on delivering value in small increments instead of one big final release.

Data Mesh: A decentralised architecture where each domain manages its own data as a product, while a shared platform provides standards and tools to keep everything consistent and easy to use.

Data Privacy: Data privacy laws protect how personal data is collected, used and shared. Key examples include: GDPR (Europe): Strict rules for handling personal data and transferring it outside the EU. CCPA (California): Similar to GDPR, giving consumers more control over their data. Australian Privacy Principles (APPs): Core privacy rules under the Privacy Act 1988, enforced by the OAIC.

Personally Identifiable Information (PII): According to the OAIC, PII is any information that can identify a person on its own or when combined with other data. Examples include Name, signature, address, phone number, date of birth, Sensitive personal details, Financial or credit information, Employee records.

Sensitive Information: The OAIC defines sensitive information as personal data that is more private and requires stronger protection. It includes details about a person's: Racial or ethnic background, Political views or associations, Religious or philosophical beliefs, Trade union membership, Sexual orientation or behaviour.

Special Data Protection Rules: Certain types of data are protected by stricter laws that control where the data can be stored and how it must be handled. These rules are designed to reduce risk and prevent misuse. Examples include Tax file numbers, social security numbers, Credit card information.

Data Minimalism: Collect only the data you genuinely need. Avoid gathering sensitive information unless it is essential. When designing pipelines and storage, consider potential risks and ensure access credentials are always kept secure.

Right to be Forgotten: Historic or unnecessary PII should be erasable. While not legally required in Australia, GDPR provides a right to erasure (Article 17) and objection to processing for non-essential purposes (Article 21). Data systems should allow easy identification and deletion of outdated data.

Principle of Least Privilege: Access should be limited to what is necessary for a task. Apply to both people and systems, restrict access duration, use granular controls (row, column, cell), mask sensitive data, and provide views with only the required information. Processes: Focus on real security, not just compliance. Embed strong privacy governance, keep procedures simple, and conduct regular security training for the team.

Data Analytics and Australian Privacy Principles: Use de-identified data, apply privacy-by-design, conduct Privacy Impact Assessments, be transparent, know and justify data collection, handle sensitive info carefully, provide user options, ensure accuracy, and protect data according to risk.

Security Scope: Assets: Data objects (files, tables, views, rows) and the systems managing them. Threats: Power failures, employee fraud, denial of service, unauthorized access. Security Objectives: Protect assets, detect breaches, minimize loss, and enable recovery. Note: Not all data is sensitive, but critical data requires robust protection.

Threats to Data Security: Unauthorized modification: sabotage, crime, or mistakes. Unauthorized disclosure: leaks of sensitive data. Loss of availability: DoS attacks. Commercial sensitivity: internal fraud. Personal privacy: legislative requirements, audit trails, careful handling.

Technology and Measures: Patch and update systems regularly. Encrypt data at rest and in transit. Implement logging, monitoring, and alerting; include anomaly detection. Restrict network access; avoid publicly accessible cloud storage, secure APIs and SSH access.

Encryption: Protects data by converting it into unreadable format, ensuring confidentiality both when stored (at rest) and transmitted (in transit). At Rest: Encrypt all stored data on servers, databases, and cloud storage. Ensure laptops have full-disk encryption and backups are encrypted. In Transit: Use HTTPS for cloud APIs. Handle API keys securely and avoid vulnerable protocols like FTP.

Other Privacy Aspects: Data Storage Security: Encrypt data at rest in databases, data lakes, and cloud storage. Implement role-based or attribute-based access controls (RBAC/ABAC). Data Access Management: Authenticate and authorize users/systems (MFA, OAuth). Apply least privilege: grant minimal access needed for roles. Data Masking and Anonymization: Mask sensitive data when full access isn't required. Anonymize datasets to prevent identification of individuals. Network Security: Use firewalls, VPNs, and ACLs to restrict network access. Adopt Zero Trust: verify every device/user before granting access. Data Integrity and Validation: Use checksums and hashing to ensure data isn't altered. Apply quality controls for accuracy and consistency. Data Governance and Compliance: Follow regulations like GDPR, HIPAA, CCPA. Define retention policies and securely delete data when not needed. Incident Response and Disaster Recovery: Have data breach response plans to contain and mitigate damage. Regular backups and recovery plans to handle failures or attacks.

Disaster Prevention – Backups: Regularly back up data following the 3-2-1 rule: At least 3 copies on 2 different media, 1 copy off-premises. Protects against failures, accidents, and ransomware attacks.

Backup: Copy of data to protect against loss, corruption, or failure. **Recovery:** Restoring data after a loss event. **Importance:** Ensures business continuity, data integrity, and availability. **Types of Backups:** Full Backup: Complete copy; simple to restore, storage intensive. Incremental Backup: Only changes since last backup; fast, less storage, complex restoration. Differential Backup: Changes since last full backup; easier to restore than incremental, moderate storage.

Backup Strategies: Frequency: Daily, weekly, monthly based on criticality. Automation: Automated backups reduce errors and ensure consistency. Location: On-Premises: Local drives, NAS. Cloud: AWS, Azure, etc. Hybrid: Combines on-premises and cloud for redundancy.

Recovery Point Objective (RPO): Maximum acceptable data loss (time-based). Ex: RPO 1 hour $\longrightarrow$ backup every hour. Importance: Determines backup frequency; shorter RPO reduces data loss but costs more. Use Case: Critical systems may need minutes; less critical data can tolerate hours. **Recovery Time Objective (RTO):** Maximum time to restore operations after failure. Ex: RTO 2 hours $\longrightarrow$ full system restored within 2 hours. Importance: Guides disaster recovery planning. Use Case: Online services need short RTO; internal systems can allow longer.

Backup Solutions for Distributed Systems: Challenges: Data spread across multiple locations and databases. Maintaining consistency between backups across nodes. Concurrency control during backups to avoid conflicting transactions. Solutions: Distributed Backup Tools: e.g., Google Cloud Spanner, Amazon Aurora. Replication: Maintain multiple copies across nodes. Eager Replication: Immediate consistency, lower data loss, higher overhead. Lazy Replication: Asynchronous, eventual consistency, potential data loss.

Types of Recovery: Crash Recovery: Restore system after a crash by undoing incomplete transactions using logs to achieve a consistent state. Disaster Recovery: Recover from large-scale failures (hardware, natural disasters, cyberattacks) using failover systems and separate disaster recovery sites. Point-in-Time Recovery (PITR): Restore the database to a specific moment to undo accidental deletions or corruption.

Backup and Recovery Best Practices: Regular Testing: Periodically verify backups for successful restoration. Multiple Copies: Maintain backups in multiple locations (on-premises and cloud) for redundancy. Encryption: Protect backup data at rest and in transit. Version Control: Keep multiple restore points to enable rollback to specific data versions.

Tools for Backup and Recovery: Database-specific tools include MySQL (`mysqldump`, MySQL Enterprise Backup), PostgreSQL (`pg_dump`, continuous archiving with `pg_basebackup`), and SQL Server (native full, differential, and transaction log backups). Cloud-based solutions include AWS Backup, which provides automated backups across AWS services; Google Cloud Backups, which offers managed backups for distributed databases such as Cloud Spanner and Bigtable; and Azure Backup, which provides scalable, cloud-native backup solutions.

Monolith: Monolith $\longrightarrow$ single system with all layers (Data, Business, UI). Features: Self-contained system **Pros:** Simple, fewer moving parts, less context switching **Cons:** Brittle, hard to update or migrate, slow releases **Modular:** Modular $\longrightarrow$ multiple microservices handling specific tasks. Features: Decoupled components communicating via APIs. **Pros:** Swappable components, easy to adopt new technologies **Cons:** Many systems to maintain, higher management complexity.

Optimising Cost and Business Value: Goal: Maximise business value while minimising Total Cost of Ownership (TCO) and opportunity cost. Cost Monitoring: Track expenses to avoid surprises. Expense Types: CapEx: Up-front investment in hardware/software. OpEx: Gradual, flexible spending, common in cloud services.

Build: Pros: Full control over the solution. **Cons:** Requires resources and expertise. When Worthwhile: Provides a competitive advantage. **Buy: Pros:** Avoids reinventing the wheel **Cons:** Dependent on vendor/community When Worthwhile: System isn't a key advantage

Types of Purchased Software: Open Source: Free, licensed for general use. Community Managed: Evaluate maturity, popularity, roadmap, documentation (e.g., Spark). Commercial: Assess value, pricing, support (e.g., Databricks). Independent/Proprietary: Less transparent, can still be effective. Consider: Interoperability, documentation, pricing, longevity

Location: On-Premises: Company owns hardware/software, manages upgrades, and handles peak loads. Cloud: Rent hardware/services (AWS, Azure, Google Cloud); allows rapid project launch and dynamic scaling. Hybrid Cloud: Mix of on-premises and cloud. Multi cloud: Deploy across multiple clouds to leverage system benefits. Use only when there's a strong justification.

Interoperability: Describes how systems connect and exchange data. Data pipelines often use multiple technologies, so evaluate how easily they integrate, whether natively or requiring manual configuration. Design pipelines modularly to allow swapping technologies and use APIs for seamless communication, for example, the Canvas API.

Speed to Market: In technology, delivering quickly matters. For data pipelines, select tools that enable fast, reliable, and secure feature delivery. Focus on delivering value early, iterate using agile approaches, and maintain high quality and security standards.

Team Size and Capabilities: Architecture should guide technology choices, not the other way around. Select tools that fit the team's skills and bandwidth. Small teams benefit from familiar technologies or SaaS solutions. Learning new tools is worthwhile only if it clearly adds value and will be used in production.

Apache Beam: Purpose: Unified model for batch and streaming pipelines Execution: Executes on multiple runners (e.g., Flink, Spark, Dataflow) Scope: Flexible pipelines for stream/batch data. **Apache Airflow:** Purpose: Workflow orchestration and scheduling Execution: Executes DAGs across workers to manage task dependencies Scope: Scheduling and monitoring of data workflows. **Apache NiFi:** Purpose: Dataflow automation and management Execution: Automates data movement between systems Scope: Low-latency data integration **Apache HBase:** Purpose: Distributed NoSQL database (columnar storage). Execution: Provides real-time reads/writes for big data. Scope: Random access to large datasets