What to say at interview start :

1. I built an end to end MLOps project : NLP Text Summarisation with MLOps.
2. It fine tunes a hugging Face Pegasus transformer on SAMSum dialogue dataset (abstractive summarisation), Evaluates it with ROUGE, and exposes realtime inference via a FastAPI.
3. The Whole pipeline - data ingestion , data validation , transformation , training , evaluation , prediction - is configuration- driven, logged, containerised with docker, and CI/CD ready.
4. I also automated builds and image publishing so the model can be deployed anywhere

Systematic Explanation

1. **Goal** : Convert long dialogues/ text to short, meaningful summaries without losing intent - using abstractive summarization (model generates new text).
2. **Core model** : Hugging Face Pegasus (a Seq2Seq transformer pre-trained for summarisation).
3. **Pipeline/MLOps focus** : make the ML work reproducible, configurable, logged, and deployable. Each stage is modular so it can be tested, rerun, or automated.

Step by step :

1. Project scaffold & packaging

1. **What :** created a project template (script template.py) to auto-generate folders & files; added requirements.txt, setup.py, README.md.
2. **How :** used os, pathlib, logging to create directories and placeholder files. setup.py configures the package metadata for professional packaging.
3. **Why:** speeds project bootstrapping, keeps consistent structure (src/, configs/, artifacts/), and prepares for packaging/distribution.

2. Logging & utils

1. **What :** src/textSummarizer/logging/__init__.py configures a logger writing to logs/running_logs.log and stdout. utils/common.py implemented helpers: read_yaml, create_directories, get_size.
2. **How :** Python logging with FileHandler + StreamHandler. read_yaml uses yaml.safe_load and returns a ConfigBox for dot access. ensure_annotations used to validate types at runtime.
3. **Why :** central logging for traceability and debugging; utility functions make code modular, reusable, reduce repetition, and ensure consistent behavior; ensure_annotations adds runtime type safety.

3. Configuration management

1. **What:** config/config.yaml and params.yaml hold paths, URLs, hyperparameters. ConfigurationManager reads these and returns typed dataclasses (DataIngestionConfig, ModelTrainerConfig, ModelEvaluationConfig, etc.).
2. **How:** read_yaml → ConfigBox → dataclasses for each stage. ConfigurationManager ensures artifact directories exist.

3. **Why:** configuration-driven approach avoids hardcoding, enables reproducibility and easy environment changes (dev/prod), and supports automated runs.

4. Data Ingestion

1. **What:** Downloaded summarizer-data.zip (SAMSum dataset) from a URL, saved to artifacts/data_ingestion, extracted it.
2. **How:** DataIngestion component: idempotent download_file() using urllib.request.urlretrieve (checks if file exists), extract_zip_file() uses zipfile. Logged headers and file size.
3. **Why:** reliable data acquisition, idempotency prevents repeated downloads, artifact storage for traceability.

5. Data Validation

1. **What:** Verified required files (train, validation, test) exist in the extracted dataset. Wrote status to artifacts/data_validation/status.txt.
2. **How:** DataValidation component reads directory contents, compares to ALL_REQUIRED_FILES, logs result.
3. **Why:** prevents training on incomplete/broken data; ensures pipeline stops early with clear logs if dataset is missing parts.

6. Data transformation / preprocessing

1. **What:** Tokenized inputs (dialogue) and targets (summary) with AutoTokenizer; converted to input_ids, attention_mask, and labels; saved transformed dataset.
2. **How:** convert_examples_to_features() applied with dataset.map(..., batched=True); used truncation (max_length=1024 for inputs, 128 for summaries) and DataCollatorForSeq2Seq for batching.
3. **Why:** Transformers expect numeric token ids and aligned labels; preprocessing ensures memory control (truncation/padding) and correct training input format.

7. Model training

1. **What:** Fine-tuned google/pegasus-cnn_dailymail with Hugging Face Trainer. Used training args from params.yaml (epochs, batch size, warmup, weight decay, gradient accumulation).
2. **How:** Initialize AutoModelForSeq2SeqLM + tokenizer, set TrainingArguments, Trainer (model, tokenizer, data_collator, train/val datasets), run trainer.train(). Observed training logs: 51 steps, final loss ~3.046.
3. **Why**: Transfer learning (fine-tuning) is faster and more effective than training from scratch. Trainer simplifies training loop, evaluation, checkpointing.

8. Training interpretation (human summary)

1. **Start training:** feed mini-batches (51 steps), compute loss, update weights via backprop. After 1 epoch final loss ~3.0 — model learned summarization patterns but room to improve with more epochs. Next steps: train more epochs, increase data, tune generation hyperparameters.

9. Model Evaluation

1. **What:** Evaluated on test split using ROUGE metrics (ROUGE-1, ROUGE-2, ROUGE-L, ROUGE-Lsum). Batch inference performed with beam search

(num_beams=8) and length_penalty=0.8. Results saved in artifacts/model_evaluation/metrics.csv. approx. Values for fine-tuned Pegasus on SAMSum: ROUGE-1 ~0.43, ROUGE-2 ~0.20, ROUGE-L ~0.36.
2. **How:** ModelEvaluation loads model+tokenizer from saved paths, loads dataset via load_from_disk, iterates batches, model.generate(), decodes outputs, accumulate metric batches, compute final ROUGE. Save metric CSV.
3. **Why:** Quantitatively assesses closeness to human summaries. Batched generation prevents memory overflow; beam search improves output quality.

10. Prediction pipeline & API

1. **What :** PredictionPipeline loads tokenizer and uses Hugging Face pipeline("summarization", model=..., tokenizer=...) to generate summaries with specified generation kwargs. app.py exposes /train and /predict routes via FastAPI. / redirects to /docs (Swagger UI).
2. **How :**
   1. POST /predict: user sends text → PredictionPipeline.predict(text) → returns model generated summary.
   2. GET /train: runs os.system("python main.py") to re-run pipeline.
3. **Why :** Provide real-time inference and easy retrain trigger via web API; FastAPI provides high performance and automatic docs.

11. Containerisation with Docker

1. **What :** Wrote Dockerfile, built image, ran container, pushed to Docker Hub. Dockerfile installs dependencies and runs app.py.
2. **How :** docker build -t text_summarizer_app . → docker run -p 8080:8080 text_summarizer_app. Pushed as deep841/text_summarizer_app:latest.
3. **Why :** Ensures reproducible runtime environment (same OS libs, Python libs). Makes the app portable and easy to deploy.

12. CI/CD

1. implemented the local Docker and pushed to Docker Hub; AWS part is ready to enable CI/CD if needed.

Definitions :

1. **Abstractive summarisation :** model generates new concise text capturing the meaning (vs extractive which picks sentences).

2. **Transformer :** Neural architecture using self-attention to relate tokens globally; create for long-range dependencies
   1. Unlike old modes (RNN , LSTMs) Transformer read the entire sentence at once.
   2. They use self attention to understand which words are related to which.
   3. Transformers preservers context and relationships (semantic meaning) btw words even they are far apart .
   4. That’s why they r perfect for : summarisation, translation, question answering, semantic analysis, chatbots.

5. **Importance** : they understand the full meaning of the text using self attention, n they also preserve the semantic relationships btw objects…which makes it perfect for text summarisation.
6. **Working** :
    1. I/P Text -> Tokens (words broken into small tokens)
    2. Self attention layer -> Finds which words r imp to each other.
    3. Encoder-Decoder Layer ->
        1. Encoder understands the text meaning.
        2. Decoder generates a shorter, meaningful version (summary)
7. In my project :
    1. Transformer performs the actual summarisation.
    2. Flask app sends text to transformer model -> gets the summary back
    3. MLOps handles data, training, deployment, and monitoring around the model.

3. **Pegasus :** A Pretrained Seq2Seq model optimized for summarization tasks.

4. Autotokenizer / AutoModelForSeq2SeqLM :
    1. AutoTokenizer : is a universal class from hugging face that automatically loads the correct tokeniser for chosen model… I won't need to manually specify the tokeniser type (like pegasus Tokeniser or bartTokenizer).
    2. AutoModelForSeq2SeqLM : AutoModel -> automatically selects the correct model architecture and Seq2SeqLM -> Sequence to sequence language model (means when one sequence is given(I/P Text) the model O/Ps another sequence (summary))

5. **Tokenization :** it means breaking a big sentence or paragraph into smaller parts called tokens - usually words, subwords, or characters

6. **Tokenizer :** a tool that : splits text into tokens , maps each to a numerical ID , adds special tokens , pads or truncates sequences to a fixed length

7. **DataCollatorForSeq2Seq :**
    1. Data collector prepares batches of data for training n ensures both I/P (dialogue) n O/P (summary) are tokenised & properly aligned
    2. It bundles (batches), pads (sequences dynamically to same length), and format data into mini-batches for training.
    3. DataCollatorForSeq2Seq is like a helper that organises data before feeding into the model

8. **ROUGE :** suite of metrics measuring overlap between machine and human summaries
    1. Rouge Evaluation :
    2. ROUGE-1 : measures overlap of single word
    3. ROUGE-2 : overlap of 2-word pairs (bigrams)
    4. ROUGE-L : measures longest common subsequence (structure similarity)
    5. ROUGE-Lsum : overall senetence-level similarity
    6. The higher the ROUGE score (closer to 1) -> the better model summaries like human.

9. **FastAPI :**

1. FastAPI is a modern, high speed web framework used to build APIs with Python
2. It lets users send input (like text) -> model processes it -> returns the result instantly
3. Used for real time ML model deployment
4. Faster than flask (because its built on ASGI instead of WSGI)….refers to web frameworks like FastAPI or Quart, which leverage the Asynchronous Server Gateway Interface (ASGI) for potentially higher performance compared to traditional Web Server Gateway Interface (WSGI) based frameworks like Flask.
5. high performance python framework for creating APIs
6. uvicorn -> web server that run FastAPI apps

10. Docker

1. It is a containerisation platform, it lets us to package entire project(code + dependencies + OS libs) into one portable unit called a container
2. Inside docker setup :
    1. Dockerfile : file telling docker about how to build app image step by step
    2. Docker Image : it is like a blueprint of app
    3. Docker Container : docker takes that image and launches a container
    4. Port Mapping : -p 8080:8080 means inside docker, FastAPI runs on port 8080, docker exposes that port to local host machine
3. Why Docker is crucial :
    1. Dependency conflicts : same environment always
    2. Works everywhere
    3. One command deploy
    4. Cloud deployment : supported by AWS, Azure, GCP, etc
    5. Reproducibility : model + code + dependencies are preserved

11. Configuration Driven

1. Configuration Driven pipeline : means that ur code does not hardcode paths or parameters - instead everything (like paths, URLS, filenames, etc) comes from config files such as config.yaml , params.yaml ; to make ML pipeline flexible &b reusable

12. **Idempotent :** Repeated runs produce the same results

13. **Reproducibility :** ability to re-run pipeline with same config and get same outputs.

14. Text Summariser

1. **Def -** It is a NLP based System that converts Long Text to short , meaningful summary without losing the main meaning of the text.
2. **Types** -
    1. Extractive Summarisation : Model select important sentences from the original text , Example: selecting key lines directly from the article , Tools: TextRank, Sumy, Spacy, etc.
    2. Abstractive Summarisation : Model understands the text and then rephrases it in its own words (like a human) , Tools: Transformers (e.g., BART, T5, Pegasus).
3. **Working :** I/P text to Model.
    1. Model Tokenises it (into words/sentences)
    2. NN processes it to understand context

3. Model generates shorter sentences capturing key meanings
4. O/P = Summary

Extra(s)

Libs :

1. **Torch :**
    1. It is the main package of PyTorch, a DL framework made by Facebook
    2. It helps in building and training NNs - just like tensor flow does for google.
    3. Works with tensor (multi-dimensional arrays like Numpy, but with GPU support)
    4. Supports automatic differentiation (autograd) -> calculates gradients automatically for back propagation.
    5. Used for training DL models (CNN, RNN, Transformers, etc)
2. **PyTorch :**
    1. An open source ML framework based on torch
    2. It provides high level APIs for building and training NNs
    3. In my project used behind transformers models - eg AutoModelForSeq2SeqLM is built on PyTorch.
3. **tqdm :**
    1. Means “progress bar”
    2. It shows progress bar when running loops.
    3. Helpful to track model training or data processing progress (when generating summaries).
4. **ntlk :**
    1. Natural Language Toolkit, py lib for NLP
    2. Helps text processing like : Tokenization, Stopword removal. Stemming/ lemmatization.
    3. Used to tokenize text into sentence before training & summarisation.
5. **Pandas :**
    1. A lib for data analysis & manipulation
    2. Provides DataFrame, like excel tables in py
    3. Great for CSV, Excel, JSON handling
    4. Used to shoe ROUGE metric scores in a DataFrame.
6. **NumPy :**
    1. A lib for numerical computing in py
    2. Provides N-dimensinoal arrays (ndarray)
    3. Performs mathematical operations super fast (vectorised)
    4. Backbone for many libs like pandas , scikit-learn
7. **Matplotlib :**
    1. A py data visualization library.
    2. For creating graphs n charts - line charts, histograms, etc
8. **Seaborn :**
    1. A statistical data visualisation library built on top of matplotlib
    2. It makes chats prettier and adds statistical insights like heatmaps, pairplots, etc

Exceptions :

1. **Exceptions** are errors that stop normal flow of program - like when something unexpected happens.
2. **Built-in (Box) Exceptions** are default exceptions provided by python. Eg : ZeroDivisionError -> Divide by zero.
3. C**ustom Exceptions** these are own custom error class, for specific use case, these controls what happens when specific project errors occur.

.yaml

1. Yet Another Markup Language
2. Used for Config files & structured data
3. Syntax : Simple key- value pairs
4. Advantage : Human-Friendly, easy to edit
5. In my project used for : Paths, parameters, and pipeline setup
6. .yaml files are like the "control center" of your project — they store all important configurations (like model paths, parameters, data folders) in one easy-to-read place.

ConfigBox

It is a special class from the python-box library that allows to access dictionary values using dot notation (.) instead of square brackets ([ ])

Data Ingestion

1. Data Ingestion is the process of collecting data from external sources (like APIs, databases, files, sensors, or URLs) and loading it into a centralized storage or pipeline (like a local folder, data lake, or database) for further processing.
2. In my project : dataset(summariser-data.zip from GitHub) ; store it locally in (artifacts/data_ingestion/data.zip)
3. Stores the raw data in a controlled folder
4. Types :
    1. Batch
    2. Real time (Streaming)
    3. Lambda.Hybrid
5. I built a modular data ingestion component with configuration management, logging, and idempotent downloads to ensure reproducibility and automation in the ML workflow.
6. Pipeline flow :
    1. Source (URl / API / DB)
    2. Data Ingestion (download + extract)
    3. Data Validation (check for missing, duplicates)
    4. Data Transformation (Tokenization, cleaning)
    5. Data Training
7. I built a modular data ingestion component with configuration management, logging, and idempotent downloads to ensure reproducibility and automation in the ML workflow

1. **Modular Data Ingestion Component** : a separate, reusable part of project that only handles data downloading and extraction
2. **Configuration Management** : managing all imp project settings (like file paths, URLs parameters) .. so that I can change values easily without touching code
3. Logging : whenever code downloads, extracts, or fails — it writes messages to a log file (e.g., running_logs.log) ; It helps in debugging and understanding what happened during a run.
4. **Idempotent** : running the same code multiple times gives the same result, without side effects
5. **Reproducibility** : Ability to reproduce (repeat) the same results again with the same configuration and code ; same O/P.
6. **Automation** : Making the entire ML workflow run automatically — no manual steps ; Each stage (ingestion → validation → training → evaluation) runs one after another through pipelines.

Data Validation

1. Data Validation checks whether the required files - train, test, and validation - exist after the data ingestion stage
2. Basically making sure that the dataset is present and not broken before training the model. See Step - 9
3. In the data validation stage , I created a configuration-driven pipeline that ensures all necessary dataset files (train, test, validation) are present after ingestion. It checks for missing files and logs the validation result. This helps prevent downstream model training failures and ensures data reliability

Data Transformation

1. It means converting raw textual data into a format that the ML model can understand
2. Step 10

Model Training

1. Model Training Means feeding the preprocessed (tokenized) data into a ML model
2. Step 11

Model Evaluation

1. It means measuring how good ur trained model performs on unseen/test data.
2. It helps to know whether model is generating accurate & meaningful summaries