[Company name] | [Company address]

[Document subtitle]

Naveen Silvester

[Year]

Contents

[Module-1: Foundation of LLMs and Generative AI 3](#_Toc204380928)

[Evolution of Language Models 3](#_Toc204380929)

[GPT (OpenAI) 3](#_Toc204380930)

[PaLM / Gemini (Google DeepMind) 3](#_Toc204380931)

[LLaMA (Meta) 3](#_Toc204380932)

[Claude (Anthropic) 4](#_Toc204380933)

[DeepSeek (China) 4](#_Toc204380934)

[Grok (xAI) 4](#_Toc204380935)

[Architecture breakdown: 6](#_Toc204380936)

[Why are Transformers essential for Generative LLMs? 7](#_Toc204380937)

[Pre-Training vs Fine-Tuning vs Instruction-Tuning 8](#_Toc204380938)

[Pre-Training (The Foundation): 8](#_Toc204380939)

[Fine-tuning (Task Specialization) 9](#_Toc204380940)

[Instruction-Tuning (Alignment with User Intent) 10](#_Toc204380941)

[Summary Table: 11](#_Toc204380942)

[Fine-Tuning Model using a Pre-trained Model 12](#_Toc204380943)

[1. Define your goal and choose your task 12](#_Toc204380944)

[2. Select Pre-trained Base Model 12](#_Toc204380945)

[3. Data Collection and Preparation (The Most Crucial Step) 12](#_Toc204380946)

[4. Tokenization: 13](#_Toc204380947)

[5. Choose a Fine-tuning Strategy 14](#_Toc204380948)

[6. Set up the Training Environment 15](#_Toc204380949)

[7. Configure Hyperparameters 15](#_Toc204380950)

[8. Train The Model 16](#_Toc204380951)

[9. Evaluate the Fine-tuned Model 16](#_Toc204380952)

[10. Deployment (Optional, but usually the goal) 17](#_Toc204380953)

[Key consideration and Best Practices 17](#_Toc204380954)

[Low-Rank Adaptation (LoRA) 18](#_Toc204380955)

[Theoretical Basis (Rank-Deficiency Hypothesis): 18](#_Toc204380956)

[Where LoRA is Applied (Insertion Points): 18](#_Toc204380957)

[Advantages Beyond Parameter efficiency: 19](#_Toc204380958)

[Hyperparameters and Tuning: 20](#_Toc204380959)

[Limitations/Considerations 20](#_Toc204380960)

[Practical Implementation (Hugging Face PEFT Library) 21](#_Toc204380961)

[Quantized Low-Rank Adaptation (QLoRA) 23](#_Toc204380962)

[The QLoRA Process 23](#_Toc204380963)

[Key Innovations in QLoRA (beyond basic 4-bit): 23](#_Toc204380964)

[Summary of Advantages: 24](#_Toc204380965)

[Understanding 32Bit representation 25](#_Toc204380966)

[BLEU (BiLingual Evaluation Understudy) 27](#_Toc204380967)

[Key Concepts in BLEU Calculation: 27](#_Toc204380968)

[Example and Step-by-Step Calculation: 29](#_Toc204380969)

[Step 1: Tokenization 29](#_Toc204380970)

[Step 2: Calculate Modified N-gram Precisions (for N=1 to 4) 29](#_Toc204380971)

[Step 3: Calculation Brevity Penalty (BP) 32](#_Toc204380972)

[Step 4: Calculate the BLEU Score 33](#_Toc204380973)

[ROUGE (Recall-Oriented Understudy for Gisting Evaluation) 34](#_Toc204380974)

[Example and Step-by-Step Calculation: 35](#_Toc204380975)

[Ethical Foundation and Governance in Generative AI 39](#_Toc204380976)

[Ethical Foundations in Generative AI 39](#_Toc204380977)

[Governance Strategies in Generative AI 41](#_Toc204380978)

[Lab: Build a mini GPT-like transformer using Hugging Face 43](#_Toc204380979)

[Lab: Practical Implementation of Fine-tuning an LLM with LoRA using Python 44](#_Toc204380980)

[Module-2: Retrieval-Augmented Generation (RAG) Systems 45](#_Toc204380981)

# Module-1: Foundation of LLMs and Generative AI

Here’s a comprehensive overview of the evolution of major language models – each representing a distinct philosophy in architecture, openness, reasoning and multimodality. These models have shaped the 2025 AI landscape across enterprise research and consumer domains.

## Evolution of Language Models

### GPT (OpenAI)

**Progression:** GPT-1 🡪 GPT-2 🡪 GPT-3 🡪 GPT-4 🡪 GPT-4o

**Highlights:**

GPT3: 175B parameters, few-shot learning

GPT-4: Multimodal (text + Image), 128k context window

GPT-4o: Real-time capabilities, improved latency, voice + vision integration

**Strengths:**

General purpose reasoning, stronger developer ecosystem, plugin support

**Use-Cases:** Coding, writing, tutoring, enterprise automation

### PaLM / Gemini (Google DeepMind)

**Progression:** PaLM 🡪 PaLM 2 🡪 Gemin 1 🡪 Gemini 2.5 Pro

**Highlights:**

Gemini 2.5 Prop: 1M token context, Deep Think mode for parallel reasoning

Native Multimodal processing (Text, Image, Audio and Video)

**Strengths:**

Google ecosystem integration, workspace productivity, video understanding

**Use-Cases:**

Research, document analysis, creative generation, enterprise workflows

### LLaMA (Meta)

**Progression:** LLaMA 🡪 LLaMA 2 🡪 LLaMA3 🡪 LLaMA4

**Highlights:**

LLaMA 4: Mixture-of-Experts (MoE), multimodal variants (Scout, Maverick)

Open weights, optimized for fine-tuning and private deployment

**Strengths:**

Open-source flexibility, multilingual support, cost-efficient scaling

**Use-Cases:**

Custom enterprise models, academic research, agentic systems

### Claude (Anthropic)

**Progression:** Claude 1 -> Claude 2 🡪 Claude 3 🡪 Claude 4

**Highlights:**

Claude 4 Opus: 1M Context, top-tier coding and reasoning

Constitutional AI for safety and alignment

**Strengths:** Long-context handling, ethical design, agentic tool use

**Use-cases:** Legal analysis, compliance, autonomous agents, technical writing

### DeepSeek (China)

**Progression:** DeepSeek R1 🡪 DeepSeek V3

**Highlights:**

Mixture-of-Experts architecture (671B total, 37B active per query)

Competitive performance at fraction of cost

**Strengths:** Technical reasoning, cost-efficiency, open-source availability

**Use-Cases:** Coding, STEM research, academic deployment

### Grok (xAI)

**Progression:** Grok1 🡪 Grok2 🡪 Grok3

**Highlights:**

Real-time integration with X (Formerly Twitter)

Specialized modes: Think Mode, DeepSearch, Big Brain Mode

**Strengths:** Truth-seeking philosophy, edgy tone, real-time data access

**Use-Cases:** Current Events, Coding, Mathematical reasoning, Informal Q&A

#### Comparative Snapshot

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model** | **Context Window** | **Multimodal** | **Open Source** | **Agentic Capabilities** | **Safety Focus** |
| GPT-4o | 128k | ✅ | ❌ | ⚠️ (limited) | ⚠️ |
| Gemini 2.5 | 1M | ✅ | ❌ | ⚠️ (emerging) | ✅ |
| LLaMA 4 | 10M (Scout) | ✅ | ✅ | ⚠️ (customizable) | ⚠️ |
| Claude 4 | 1M | ✅ | ❌ | ✅ (tool use, memory) | ✅ |
| DeepSeek R1 | 37B active | ❌ | ✅ | ⚠️ (efficient agents) | ⚠️ |
| Grok 3 | 1M | ✅ | ❌ | ✅ (real-time tools) | ⚠️ |

## Architecture breakdown:

Transformer:

At it’s core, a Transformer is a type of neural network designed to process sequential data, like text, more efficiently and effectively than previous architectures (such as RNNs). It was introduced in the groundbreaking 2017 paper “Attention is All you Need” by Google Brain Researchers.

Key Innovation of Transformers for LLMs:

The power of transformers in generative LLMs stems from a few critical innovations:

1. Self-Attention Mechanism: This is the heart of the transformer, unlike RNNs that process words one by one, the self-attention mechanism allows the model to consider all words in the input sequence simultaneously. For each word, it calculates how much “attention” it should pay to every other word in the sentence. This means it can identify relationships and dependencies between words, even if they are far apart in the sequence. For example, in the sentence “The dog chased the cat because it was hungry”, the self-attention mechanism can understand that “it” refers to” dog”, even though they are separated by other words. This ability to capture long-range dependencies is vital for understanding context and generating coherent text.
2. Parallel Processing: Because self-attention allows the model to process all part of the input sequence concurrently, Transformers are highly parallelizable. This is a massive advantage over RNNs, which process information sequentially. Parallel processing significantly speeds up training times, making it feasible to train models on truly massive datasets (Billions of words or more) and with Billions of parameters. This scalability is what enables the creation of “Large” Language Models.
3. Positional Encoding: While self-attention allows parallel processing, it loses the inherent sequential order of words. To address this, Transformers incorporate “Positional Encodings” which are numerical representations added to the work embeddings to provide information about each word’s position in the sequence. This ensures the model understands the order and relative position of words, which is crucial for grammatical correctness and meaning (e.g., “hot dog” vas “dog hot”)
4. Encoder-Decoder Structure (in some variations)

Encoder: The encoder part of a Transformer processes the input sequence and transforms it into a rich, contextualized numerical representation. It uses multiple layers of self-attention and feed-forward neural networks top refine this representation.

Decoder: The decoder then takes this encoded representation and generate the output sequence, typically one word (or “token”) at a time. It also uses self-attention (often “masked” to prevent it from looking at future words during generation) and attention to the encoder’s output. While the original Transformer paper proposed and encoder-decoder architecture, many generative LLMs (like the GPT series) primarily use a decoder-only architecture, where the entire model is focused on predicting the next token in a sequence.

## Why are Transformers essential for Generative LLMs?

**Understanding Context:** The self-attention mechanism allows LLMs to develop a deep understanding of the context of words within a sentence or even across long passages. This is crucial for generating text that is coherent, relevant and grammatically correct.

**Generating Coherent Text:** By learning relationships between words and phrases, Transformers enable LLMs to predict the most probable next word in a sequence, leading to fluid and natural-sounding text generation.

**Scalability:** The parallel processing capability is paramount. It allows researchers to train models with an unprecedented number of parameters on colossal datasets, leading to models that exhibit remarkable capabilities in language understanding and generation.

**Versatility:** Beyond just text generation, the Transformer architecture has proven incredibly versatile, finding applications, in machine translation, text summarization, question answering and even multimodal AI tasks (like generating images from text)

In essence, Transformers provide the robust and scalable architecture that allows Large Language Models to learn intricate patterns from massive amounts of text data, enabling them to generate human-like text, answer questions, translate languages, and perform a wide array of natural language processing tasks. They are the engine driving the recent revolution in generative AI.

## Pre-Training vs Fine-Tuning vs Instruction-Tuning

In the world of LLMs, the terms “pre-training”, fine-tuning” and “instructional-training” describes different stages or approaches to training these powerful models. They represent a progression from general language understanding to highly specialized and user-friendly behaviour.

### Pre-Training (The Foundation):

**Analogy:** Think of pre-training as giving a student a massive, uncurated library of all human knowledge (books, articles, websites, code etc.,) and telling them: “Read everything, and learn how language works – how words connect, grammar, facts, common patterns, and different writing styles. Don’t worry about specific tasks yet, just understand the underlying structure of text”

**Process:**

**Data:** Extremely vas and diverse datasets (trillions of words/tokens) from the internet, books, code etc., This data is largely unlabelled.

**Objective:** The primary objective is usually a self-supervised learning task, most commonly:

Casual Language Modelling (CLM): Predicting the next word in a sequence (e.g.,, as in GPT models). The model tries to complete sentences.

Masked Language Modelling (MLM): Predicting masked-out words within a sentence (e.g., as in BERT models). The model learns context from both directions.

**Result:** A “base-model” that has a broad understanding of language, facts, common sense, and various writing styles. It can generate coherent text, but it’s not yet optimized for specific instructions or conversational turns. It’s a generalist.

**Resources:** Extremely computationally expensive, requiring massive GPU clusters and months of training time. This is why only large organizations can typically “pre-train” LLMs from scratch.

**Example of Base Model Behaviour (Pre-trained only):**

If you asked an early, pre-trained-only GPT-3 model: “Summarize the article about quantum physics:”, it might continue your sentence like “Summarize the article about quantum physics: This article discusses….” Rather than actually providing summary, because its objective was just to predict the plausible tokens, not necessarily to “follow instructions”

### Fine-tuning (Task Specialization)

**Analogy:** After the student has read the entire library (pre-training), fine-tuning is like giving them a specific textbook and practice exercises on one subject, like “Medical Diagnosis” or “Legal Document Analysis” and telling them: “Now, apply your general language knowledge to become really good at this one thing”

**Process:**

**Data:** A smaller, task-specific and often labelled dataset. For example, if you want a sentimental analysis model, you would provide man sentences labelled “Positive”, “Negative” or “Neutral”

**Objective:** To adapt the pre-trained model’s general knowledge to perform a specific downstream task better. The model’s parameters are further adjusted (fine-tuned) using new data.

**Result:** A specialized model that excels at a particular task (e.g., medical, financial, legal)

**Resources:** Less computationally intensive than pre-training, but still can require significant resources, especially for “full fine-tuning” (where all model parameters are updated). Parameter-Efficient Fine-Tuning (PEFT) methods like LoRA have made this more accessible.

**Example:**

Task: Classify customer reviews as positive or negative

Fine-Tuning data: Thousands of customer reviews, each explicitly labelled “positive” or “negative”.

**Result:** The model become highly accurate at sentimental classification, even if the pre-trained model was only okay at it.

### Instruction-Tuning (Alignment with User Intent)

**Analogy:** Instruction-tuning is a specific type of fine-tuning. Building on our student analogy, it’s like teaching the student not just what they know, but how to answer question and follow directions in a helpful, conversational, and safe manner. It’s giving them a set of examples of questions and ideal answers, saying: “When someone asks you to ‘explain X’, here’s how you should structure your answer. When they ask you to ‘write a poem’, here’s what a good poem looks like.”

**Process:**

**Data:** A dataset composed of high-quality (instruction, output) pairs. These instructions can be diverse tasks (“Summarize this article,“ “Write a poem about dogs,” “Explain gravity to a 5 year old”, “Translate this to French”). The “Output” is the ideal, helpful, and safe response. This data is often collected through human annotation or generated by other LLMs (like self-instruct)

**Objective:** To align the LLMs behaviour with human instructions and preferences. It teaches the model to understand prompts, follow constraints, engage in dialogue, and generally be more useful and predictable for a user. It bridges the gap between the model’s pre-training objective (next-word prediction) and the user’s objective (getting a specific task done)

**Result:** A model that is much better at understanding and executing user commands, participating in conversations and avoiding undesirable behaviours (like generating irrelevant or harmful content). This is what makes models like ChatGPT so user-friendly.

**Resources:** Can be computationally intensive, especially for large, diverse instruction datasets, but often leverages PEFT techniques to make it more feasible.

**Example:**

Pre-trained Model: Might just complete your sentence.

Fine-Tuned Model (for Summarization): Might summarize if you specifically prompt it to.

Instruction-tuned Model: If you ask “Explain how photosynthesis works for a high school student”, it will give you a clear, structured explanation appropriate for that audience, even if it wasn’t explicitly trained on that exact phrasing. It has learned to follow instructions.

### Summary Table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Pre-training** | **Fine-tuning (General)** | **Instruction-tuning (Specific Type of Fine-tuning)** |
| **Goal** | Learn general language understanding & facts | Specialize for a specific task or domain | Align with human instructions & conversational norms |
| **Data** | Massive, diverse, largely unlabeled | Smaller, task-specific, often labeled | Curated (instruction, output) pairs; diverse tasks |
| **Objective** | Next-word prediction, masked language model | Optimize for specific task loss (e.g., classification accuracy) | Follow instructions, generate helpful/safe responses |
| **Output Model** | "Base Model" (generalist) | Specialized model for a task/domain | "Chatbot" or "Instruction-following" model (user-friendly) |
| **Cost/Resources** | Extremely High | Moderate to High (less than pre-training) | Moderate (can be reduced with PEFT) |
| **Example** | GPT-3 (base model) | BERT fine-tuned for sentiment analysis | ChatGPT, Claude, Gemini (aligned models) |

In essence, pre-training builds the raw intelligence, fine-tuning sharpens it for specific jobs, and instruction-tuning teaches it good manners and how to interpret human requests effectively.

## Fine-Tuning Model using a Pre-trained Model

Fine-tuning an LLM is the process of taking a pre-trained Large Language Model (LLM) and further training it on a smaller, task-specific dataset. The goal is to adapt the model’s vast general knowledge to excel at a particular task or within a specific domain, making it more accurate and relevant for your needs.

Here are the details of the process involved in fine-tuning an LLM.

### Define your goal and choose your task

Before doing anything else, clarify what you want the fine-tuned LLM to achieve.

What specific problem are you solving? (e.g., sentimental analysis, legal document summarization, medical question answering, customer support chatbot responses, code generation in a specific language).

What kind of output do you expect? (e.g., a classification label, a summarized text, a generated code snippet, a conversation response)

A clear objective will guide all subsequent steps.

### Select Pre-trained Base Model

You don’t start from scratch. You leverage the general intelligence of an existing LLM.

Model Architecture: Consider models like GPT-series (Generative Pre-trained Transformers), Llama, Mistral, BERT, T5, etc. The choice depends on your task (e.g., generative tasks often use decoder-only models like GPT, while classification might use encoder-decoder or encoder-only models)

Model Size: Larger models are more capable but require more computational resources for fine-tuning. Consider your hardware limitations.

Open-Source vs Proprietary: Open-source models (e.g., from Hugging Face) offer more control and customization, while proprietary APIs (e.g., OpenAI’s fine-tuning API, Google Cloud Vertex AI) offer convenience.

License: Always check the model’s license for your intended use.

### Data Collection and Preparation (The Most Crucial Step)

The quality and relevance of your data are paramount.

Collect Task specific Data: This is your “Knowledge Base” for the LLM. For Instance:

**Sentiment Analysis:** A dataset of product reviews with “positive”, “negative” or “neutral” labels

**Summarization:** Pairs of long articles and their concise summaries

**Question Answering:** (Question, Answer) pairs, potentially with relevant context passages.

**Instruction following (for Instruction Tuning):** (Instruction, Desired Output) pairs for various commands.

Data Quality:

Ensure your data is clean, consistent, and accurate. Remove noise, duplicates and irrelevant information.

#### Data Quantity:

The amount of data needed varies. For complex tasks or entirely new domains, you might need thousands to tens of thousands of examples. For Simpler tasks or slight domain shifts, a few hundred to a few thousand might suffice.

#### Format:

Your data needs to be in a format that the model can understand. This often means text-based input-output pairs. Libraries like Hugging Face’s datasets can help manage this.

Splitting:

Divide your dataset into:

**Training Set:** Used to update the model’s weights (e.g., 70-80%)

**Validation Set:** Used to monitor the model’s performance during training and tune hyperparameters (e.g., 10-15%)

**Test Set:** Used for a final, unbiased evaluation of the model after training is complete (e.g., 10-15%). This set should remain untouched during training

### Tokenization:

LLMs operate on Tokens, not raw text.

Load Corresponding Tokenizer: Every pre-trained LLM comes with its own tokenizer. It’s crucial to use the exact tokenizer that was used during the base model’s pre-training.

Process Data: Use the tokenizer to convert your data into sequences of numerical token IDs.

Padding and Truncation:

**Padding:** Add special “padding” tokens to shorter sequences to make them all the same length (required for batching).

**Truncation:** Cut off longer sequences if they exceed the model’s maximum input length.

### Choose a Fine-tuning Strategy

This step involves deciding how much of the original model you are going to modify.

#### Full Fine-Tuning:

**What it is:** Updates all the parameters (weights) of the pre-trained LLM.

**Pros:** Can achieve the highest performance for very specific tasks and can adapt the model most thoroughly

**Cons:** Very computationally intensive (requires significant GPU memory and processing power), time-consuming, and can lead to “catastrophic forgetting” (where the model forget some of its general pre-trained knowledge). Not feasible for most unless you have a significant resource.

Parameter-Efficient Fine Tuning (PEFT): This is the go-to method for most fine-tuning scenarios due to its efficiency. It modifies only a small fraction of the model’s parameters.

**What it is:** Instead of updating all billion of parameters, PEFT method introduces a small number of new, trainable parameters (or smartly modify existing ones), while keeping the vast majority of the original LLM’s weights frozen.

**Pros:** Dramatically reduces computational cost (GPU memory, training time), prevents catastrophic forgetting, and makes LLM customization accessible on more modest hardware (even consumer GPUs)

**Common PEFT Methods:**

**LoRA (Low-Rank Adaptation):** Insert small, trainable low-rank matrices into the attention layers of the transformer. When fine-tuning, only these LoRA matrices are updated.

**Prompt Tuning:** Learns “soft prompts” (continuous vectors) that are prepended to the input embeddings. The base model’s weights remain frozen, only the soft prompt is learned.

**Prefixing Tuning:** Similar to prompt tuning but adds trainable parameters to every layer of the transformer.

**Adapters:** Inserts small, narrow neural networks (adapter modules) between layers of the pre-trained model. Only the adapter modules are trained.

**QLoRA (Quantized LoRA):** An extension of LoRA that uses quantized (e.g., 4-bit) base models, further reducing memory usage.

### Set up the Training Environment

Frameworks: PyTorch (transformers library from Huggin Face is standard), TensorFlow, or JAX

Hardware: GPUs are essential. The type and numbers of GPUs depend on the model size and fine-tuning method. For PEFT, consumer-grade GPUs might suffice for smaller LLMs.

#### Libraries:

**Transformers:** For loading pre-trained models and tokenizers, and often providing a Trainer class for an easy training loop.

**Datasets:** For efficient data loading and preprocessing.

Accelerate (Hugging Face): For easily running training across multiple GPUs or machines.

**Peft (Hugging Face):** For implementing PEFT methods like LoRA

**Bitsandbytes:** For quantization techniques (useful with QLoRA)

### Configure Hyperparameters

These are settings that control the training process and significantly impact performance.

**Learning Rate:** How large of a step the model takes when updating weights. Typically small for fine-tuning LLMs (e.g., 1e-5 to 5e-5)

**Batch Size:** Number of training examples processed before the model’s weights are updated. Larger batches can lead to more stable gradients but requires more memory.

**Number of Epochs:** How many full passes over the training dataset. For fine-tuning, often just 1-3 epochs are sufficient due to the pre-trained knowledge. Over-training leads to overfitting.

**Optimizer:** Algorithms that adjust model weights (e.g., AdamW is common)

Learning Rate Scheduler: Controls how the learning rate changes over time (e.g., warm-up, linear decay)

**Weight Decay:** A regularization technique to prevent overfitting

**Dropout:** Another regularization technique.

### Train The Model

This is the iterative process of feeding data to the model and updating its weights.

**Forward Pass:** Input data goes through the model, generating predictions

Loss Calculation: A loss function (e.g., Cross-Entropy Loss for language modelling or classification) measures the difference between the model’s predications and the true labels in your dataset.

**Backward Pass (Backpropagation):** The gradients of the loss with respect to the models parameters are calculated.

**Optimizer Step:** The optimizer uses these gradients to update the model’s parameters, aiming to minimize the loss.

**Validation:** Periodically evaluate the model on the validation set to track its performance on unseen data and detect overfitting.

### Evaluate the Fine-tuned Model

After training, evaluate your model on the unseen test set.

**Metrics:** Choose appropriate metrices for your task:

**Classification:** Accuracy, Precision, Recall, F1-Score

**Generation:** BLEU, ROUGE (for Summarization), METEOR (for machine translation), human evaluation

**Question Answering:** Exact Match (EM), F1-Score

**Quantitative Evaluation:** Beyond metrics, examine some generated outputs to ensure they meet your quality standards and exhibit the desired behaviour.

**Iteration:** Fine-tuning is often an iterative process. If results are not satisfactory, revisit earlier steps, refine your data, adjust hyperparameters, try different PEFT method, or even different base model.

### Deployment (Optional, but usually the goal)

Once satisfied with the model’s performance, you can deploy it for your application.

This could involve hosting it on a cloud platform (e.g., AWS SageMaker, Google Cloud Vertex AI, Azure Machine Learning), using a managed API or integrating it into your existing software.

### Key consideration and Best Practices

**Start Small:** Begin with smaller models and dataset to quickly iterate and establish a baseline.

**High-Quality Data is king:** More data isn’t always better; better data is always better. Focus on clean, relevant and diverse examples.

**Monitor overfitting:** Keep a close eye on your validation loss. If it starts to increase while training loss decrease, you are overfitting. Early stopping is crucial.

**Leverage PEFT:** Unless you have a very specific reason and immense computational resources PEFT methods are almost always the preferred choice for fine-tuning LLMs

**Experiment with Hyperparameters:** Learning rate is often the most critical hyperparameter.

**Reproducibility:** Document your data preprocessing steps, model architecture, hyperparameters, and training pipeline to ensure you can reproduce your results.

Fine-tuning LLMs is a powerful technique to unlock their full potential for specific applications, making them highly effective tools for various real-world problems.

## Low-Rank Adaptation (LoRA)

LoRA, a highly popular and effective PEFT (Parameter-Efficient Fine-Tuning) method. Large Language Models (LLMs) are enormous, with billion of parameters. Fine-tuning for specific task by updating all these parameters is computationally prohibitive (requires massive GPUs, lots of time and huge storage) and can lead to “Catastrophic forgetting” of the general knowledge learned during pre-training. LoRA’s Big Idea: Instead of retraining all weights, why don’t we just train a small, low-rank approximation of the changes we want to make to the weights?

Theoretical Basis (Rank-Deficiency Hypothesis): The core hypothesis behind LoRA’s effectiveness is that the “update” or “change” required during fine-tuning (delta W) often has a low intrinsic rank. This means the essential information needed to adapt the model to a new task can be captured in a much lower-dimension space than the full weight matrix. This aligns with findings in neural network pruning and regularization.

### Where LoRA is Applied (Insertion Points):

LoRA is typically applied to the attention mechanism’s weight matrices in Transformers. Specifically to the Query (W\_q), Key (W\_k), Value (W\_v) and Output (W\_o) projection matrices. These are often the largest matrices and are critical for the model’s contextual understanding. You can choose to apply LoRA to all, some or just one of these. The Original paper fount it sufficient to only apply it to W\_q and W\_v.

LoRA proposes that this (delta W) the change we want to learn, can be approximated by multiplying two much smaller matrices.

Instead of directly learning (delta W), LoRA learns two smaller matrices, A and B, such that Delta W = B @ A

If W has dimensions (d\_in, d\_out) (e.g. 4069x4096):

* A has dimensions (d\_in, r)
* B has dimensions (r, d\_out)
* r is the rank (LoRA’s key Hyperparameter), and it’s chosen to be much, much smaller than d\_in or d\_out. Typically, r is between 4 and 64.

**The Math:**

Original Operation: h = x @ W

Full Fine-tuning: h = x @ (W\_orginal + delta W)

LoRA modification: h = x @ W\_orginal + x @ (B @ A)

Notice that W\_original (the large pre-trained matrix) is frozen and not updated

Only the small matrices A and B are trained.

**Parameter Savings:**

Full Delta W paramters: d\_in \* d\_out

LoRA A and B parameters = d\_in \* r + r \* d\_out

If d\_in = 4096 and d\_out = 4096, r = 8

Full Delta W: 4096 \* 4096 = 16,777,216 parameters

LoRA A and B: 4069 \* 8 + 8 \* 4096 = 32,768 + 32,768 = 65,536 parameters

That’s a massive reduction in trainable parameters (Roughly 256x fewer in this example)

For deployment, you can “merge” the trained A @ B matrices back into the original W matrix: W\_new = W\_original + B @ A. This means inference speed is identical to the original model, without no overhead.

**Technical Details of B @ A**

**A** is initialized with random Gaussian values.

**B** is initialized to zeros. This ensures that at the start of training, B @ A is zero, so the fine-tuned model initially behaves exactly like the pre-trained model (W\_original + 0). This is crucial for stable training.

The output of B @ A is scaled by a factor alpha/r. Alpha is another hyperparameter that helps control the magnitude of the learned adaption.

### Advantages Beyond Parameter efficiency:

1. Memory Savings: Fewer trainable parameters mean significant less GPU memory consumed during backpropagation, making it possible to fine-tune larger model on less powerful hardware.
2. Faster training: Fewer parameters to update means faster gradient calculations and faster optimization steps
3. No inference latency: As mentioned, the LoRA weights can be merged into the base model’s weights after training, resulting in no additional computational cost or latency during inference compared to the fully fine-tuned model.
4. Modular Storage: You can store multiple sets of small LoRA weights (e.g, one for Summarization, one for translation) and easily swap them out without needing to save multiple copies of the entire base model. This is critical for serving multiple specialized models from a single base LLM.
5. Catastrophic forgetting mitigation: By keeping the large pre-trained weights fixed, the model is less prone to forgetting its general knowledge while specializing.

### Hyperparameters and Tuning:

**r** (rank): The most critical hyperparameter. Higher r means more expressivity (can learn more complex changes) but more parameters. Common values are 4, 8, 16, 32, 64. You usually start small and increase if needed.

**alpha (scaling factor):** Often 2 \* r or a fixed value like 16 or 32. It scales the updates from B @ A.

**target\_modules:** Which linear layers in the transformer architecture you want to apply LoRA to (E.g., [“q\_proj”, “v\_proj”]

**Standard Fine-**Tuning Hyperparameters: Learning rate, batch size, number of epochs, optimizer remain important and need tuning.

### Limitations/Considerations

Approximation: Since it’s an approximation, there might be rare cases where full fine-tuning slightly outperforms LoRA, especially if the Delta W, is truly high-rank for a very specific, complex task. However, the performance difference is often negligible in practice, especially compared to the efficiency gains.

Domain Shift: If the target task is vastly different from the pre-training domain, LoRA might struggle more than full fine tuning, as the base model’s knowledge might not be sufficiently aligned.

### Practical Implementation (Hugging Face PEFT Library)

The Hugging Face PEFT library makes LoRA incredibly easy to use. You simply define a LoraConfig, pass it to get\_peft\_model, and the library handles all the patching of your base mode.

**from peft import LoraConfig, get\_peft\_model, TaskType**

**from transformers import AutoModelForCausalLM, AutoTokenizer**

**# 1. Load your pre-trained base model**

model\_name = "mistralai/Mistral-7B-v0.1" # Example model

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

model = AutoModelForCausalLM.from\_pretrained(model\_name)

**# 2. Define LoRA configuration**

# TaskType.CAUSAL\_LM for text generation

# r: rank of the update matrices

# lora\_alpha: scaling factor

# lora\_dropout: dropout probability on LoRA layers

# target\_modules: names of the modules to apply LoRA to (often specific attention layers)

config = LoraConfig(

task\_type=TaskType.CAUSAL\_LM,

r=8,

lora\_alpha=16,

lora\_dropout=0.1,

target\_modules=["q\_proj", "v\_proj"] # Common targets for attention layers

)

**# 3. Get the PEFT model**

# This "wraps" your base model with the small, trainable LoRA matrices

lora\_model = get\_peft\_model(model, config)

# Print trainable parameters to see the huge reduction

lora\_model.print\_trainable\_parameters()

# Example output might be something like:

# trainable params: 4,194,304 || all params: 7,241,185,280 || trainable%: 0.05792271830490793

**# 4. Now, you can train `lora\_model` just like any other PyTorch model**

# (e.g., using Hugging Face's Trainer or a custom training loop)

# Only the LoRA parameters will be updated.

**# 5. Save the LoRA weights (very small file!)**

# lora\_model.save\_pretrained("my\_lora\_adapter")

**# 6. Load LoRA weights and merge for inference (optional, but good for deployment)**

# from peft import PeftModel

# base\_model = AutoModelForCausalLM.from\_pretrained(model\_name)

# peft\_model = PeftModel.from\_pretrained(base\_model, "my\_lora\_adapter")

# merged\_model = peft\_model.merge\_and\_unload() # Returns a standard Hugging Face model

## Quantized Low-Rank Adaptation (QLoRA)

QLoRA is an extension of LoRA, designed to make fine-tuning even more accessible. The “Q” stands for Quantization. QLoRA is like taking an already amazing AI model, compressing it down to save a ton of space and then adding tiny, specialized LoRA adapters on top of the compressed version to teach it new tricks. This lets you train huge AI models on much less powerful computers.

**Quantization:** This is the process of reducing the precision of the numerical values (weights) in a neural network.

* Most LLMs are trained using 16-bit floating-point nuber (FP16 or BF16)
* Quantization reduces these to much lower precision, like 4-bit integers (INT4). This is a massive 4x memory reduction for the base model weights (16 bits down to 4 bits)
* The challenges with aggressive quantization is often a loss of accuracy. QLoRA introduces specific techniques to minimize this loss.

### The QLoRA Process

1. **Load Base Model in 4-Bit:** The gigantic pre-trained LLM is loaded into memory with its weights quantized to 4-bit precision. This means the majority of the model’s parameters (the “original”) now take up only ¼ the memory. Crucially, these 4-bit weights are frozen during fine-tuning.
2. Add LoRA Adapters: Just like standard LoRA, small, trainable low-rank A and B matrices are added to the attention add/or feed-forward layers of the 4-bit quantized base model.
3. Dequantization for Computation (on-the-fly): During the forward and backward passes (when the model processes data and learns), the 4-bit quantized weights are temporarily dequantized back to a higher precision (e.g., 16-bit BrainFloat, bfloat16) for actual mathematical operations. This dequantization happens dynamically, just for the calculation, and the results are then used to compute gradients.
4. Train Only LoRA Adapters: Gradients are only computed and applied to the LoRA A and B matrices, which re kept at a higher precision (e.g., 16-bit bfloat16). The 4-bit base model weights remain frozen.

### Key Innovations in QLoRA (beyond basic 4-bit):

1. **4-bit NormalFloat (NF4):** This is a new, custom 4-bit data type developed specifically for QLoRA. It’s designed to be “information-theoretically optimal” for normally distributed weights, which are common in neural networks. This means it quantizes more effectively that standard 4-bit integers or floats, preserving more accuracy.
2. **Double Quantization:** This is a clever trick to save even more memory. When you quantize, you need “Quantization constants” (like scaling factors) to know how to convert between the low precision and high-precision values. Double quantization means you then quantize these quantization constants themselves. This saves a tiny bit of memory per constant, which add up significantly across billion of parameters.
3. **Paged Optimizers:** Fine Tuning LLMs often involves memory “spikes” during gradient calculation (especially with long sequences). Paged optimizers (inspired by CPU memory paging) move optimizer states (like adam’s momentum buffers) between GPU and CPU RAM as needed. This helps prevent out-of-memory errors during training, especially with larger batch sizer or longer sequences.

### Summary of Advantages:

* **Unprecedented Memory Efficiency:** The primary benefit. Allows fine-tuning of multi-billion parameter models (e.g., 65B Llama) on single consumer-grade GPUs (e.g., 48GB A100 or even smaller cards like RTX 3090/4090).
* **Performance:** Achieves comparable performance to full 16-bit fine-tuning, despite the aggressive quantization.
* **Accessibility:** Democratizes LLM fine-tuning, making it accessible to researchers, hobbyists, and smaller companies without access to supercomputing clusters.
* **Speed:** Benefits from LoRA's faster training due to fewer trainable parameters.

## Understanding 32Bit representation

The concept of "bits" can feel abstract. Let's take a real-world number and see how it's represented in 32-bit floating-point format, which is defined by the IEEE 754 standard.

The IEEE 754 single-precision (32-bit) floating-point format breaks down the 32 bits into three parts:

* **Sign bit (1 bit):** 0 for positive, 1 for negative.
* **Exponent (8 bits):** Represents the magnitude of the number. It's stored as a biased exponent (usually with a bias of 127 for 32-bit floats).
* **Mantissa/Fraction (23 bits):** Represents the precision or the significant digits of the number. There's an *implied leading 1* for normalized numbers, giving it 24 bits of precision.

**Example: Representing the number 0.15625 as a 32-bit float**

Let's break down the decimal number 0.15625 into its 32-bit binary representation.

**Step 1: Determine the sign bit.** Since 0.15625 is positive, the sign bit is 0.

**Step 2: Convert the decimal number to binary.** To convert 0.15625 to binary:

* Multiply the fractional part by 2 repeatedly, taking the integer part at each step.
  + 0.15625 \* 2 = 0.3125 (integer part 0)
  + 0.3125 \* 2 = 0.625 (integer part 0)
  + 0.625 \* 2 = 1.25 (integer part 1)
  + 0.25 \* 2 = 0.5 (integer part 0)
  + 0.5 \* 2 = 1.0 (integer part 1) We stop when the fractional part becomes 0. So, 0.15625 in binary is 0.00101.

**Step 3: Normalize the binary number.** We need to express the binary number in "scientific notation" (normalized form), which means 1.something \* 2^exponent. 0.00101 can be written as 1.01 \* 2^-3. (We shifted the decimal point 3 places to the right, so the exponent is -3).

**Step 4: Calculate the biased exponent.** The exponent is -3. For a 32-bit float, the bias is 127. Biased exponent = exponent + bias = -3 + 127 = 124. Now, convert 124 to an 8-bit binary number:

124 (decimal) = 01111100 (binary)

**Step 5: Determine the mantissa (fractional part).** From our normalized number 1.01 \* 2^-3, the part after the 1. is the mantissa. Mantissa = 01. Since the mantissa needs to be 23 bits long, we pad it with zeros: 01000000000000000000000 (23 bits)

**Step 6: Combine all parts.**

* **Sign bit:** 0
* **Exponent:** 01111100
* **Mantissa:** 01000000000000000000000

Putting it all together, the 32-bit floating-point representation of 0.15625 is:

0 01111100 01000000000000000000000

**In chunks of 4 bits (hexadecimal representation often seen in memory dumps):** 0011 1110 0010 0000 0000 0000 0000 0000 3 E 2 0 0 0 0 0 (hexadecimal)

So, in a computer's memory, the number 0.15625 would be stored as the sequence of 32 binary digits (bits) shown above.

**Why this matters in QLoRA:**

When QLoRA talks about "32-bit computation," it means that even if the original LLM's weights are eventually represented as those compact 4-bit numbers for storage, during the actual mathematical operations (like matrix multiplications in a neural network), those 4-bit numbers are conceptually "unpacked" or "dequantized" back into a format like the 32-bit representation we just saw. This allows the calculations to be performed with the necessary precision to avoid significant errors, even though the storage is highly compressed.

It's like having a highly compressed image file (4-bit storage), but when you open it in an image editor, the software temporarily reconstructs a higher-quality version (32-bit computation) to allow for precise edits, before saving it back to a compressed format.

## BLEU (BiLingual Evaluation Understudy)

BLEU is an algorithm for evaluating the quality of text that has been machine-translated from one natural language to another. The core idea behind BLEU is that “the closer a machine translation is to a professional human translation, the better it is”. It’s a widely used and inexpensive automated metric, often correlating well with human judgement of quality.

BLEU scores range from 0 to 1, where 1 indicates a perfect match with the reference translation(s) and 0 indicates no overlap. While it was initially developed for machine translation, it’s also used to evaluate other text generation tasks like summarization and image captioning.

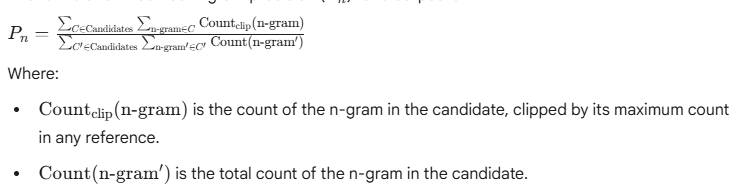
### Key Concepts in BLEU Calculation:

1. N-grams: These are contiguous sequences of ‘n’ items (words in this case) from a given text. For example, in the sentence “The cat sat on the mat”:

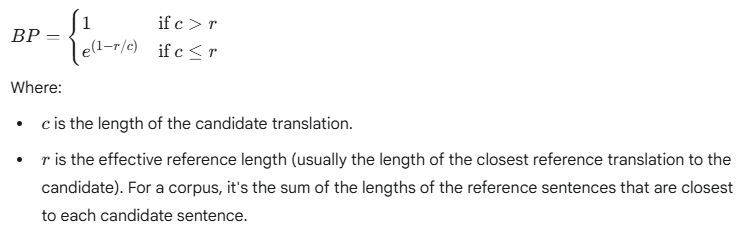
* Unigrams (1-gram): “The”, “cat”, “sat”, “on”, “the”, “mat”
* Bigrams (2-gram): “The cat”, “cat sat”, “sat on”, “on the”, “the mat”
* Trigrams (3-gram): “The cat sat”, “cat, sat on”, “sat on the”, “on the mat”
* 4-grams: “The cat sat on”, “cat sat on the”, “sat on the mat”

1. Modified N-gram Precision: This is the core of BLEU. Instead of simple precision (which can be inflated by repetition), BLEU uses a “clipped” count. For each n-gram in the candidate translation, its count is clipped by the maximum count of the same n-gram in any of the reference translations. This prevents translations that repeat common words from getting an artificially high score.

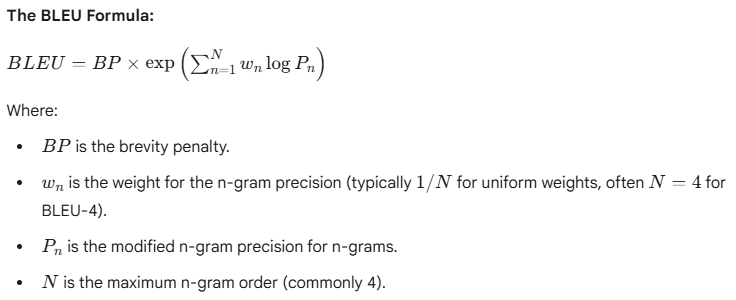
The formula for modified n-gram precision (Pn) for a corpus is:



1. Brevity Penalty (BP): BLEU penalizes translations that are too short compared to the reference translations. This is because a very short translation could have high precision if all its few words are present in the reference, but it might miss a lot of information. The brevity penalty prevents this.



1. Geometric Mean: The individual modified n-gram precisions (typically for n=1 to 4) are combined using a geometric mean. This ensures that if any single n-gram precision is zero (meaning no match for that specific n-gram length), the overall score becomes zero, which is desirable.



## Example and Step-by-Step Calculation:

Let’s consider a simple example with one candidate translation and one reference translation. For real-world scenarios, BLEU is typically calculated over a corpus (multiple sentences) and often with multiple reference translations.

Candidate Translation ( C ): “The cat sat on the mat”

Reference Translation ( R ): “A cat was sitting on the mat”

### Step 1: Tokenization

First, tokenize the sentences (convert them into lists of words). Punctuation is usually removed or treated as separate tokens, and casing is often normalized (e.g., lowercase everything).

C: [“the”, “cat”, “sat”, “on”, “the”, “mat”] (length = 6)

R: [“a”, “cat”, “was”, “sitting”, “on”, “the”, “mat”] (length = 7)

### Step 2: Calculate Modified N-gram Precisions (for N=1 to 4)

Unigram (n=1) Precision (P1):

**Count of unigrams in C:**

“the”: 2

“cat”: 1

“sat”: 1

“on”: 1

“mat”: 1

Total unigrams in C: 6

**Count of unigrams in R:**

“a”: 1

“cat”: 1

“was”: 1

“sitting”: 1

“on”: 1

“the”: 1

“mat”: 1

**Clipped Count of unigrams in C (based on R):** For each unigram in C, count how many times it appears in C, but don’t exceed its maximum count in R.

“the”: Max count in R is 1. In C it’s 2. So, clipped count is min(2,1) = 1

“cat”: Max count in R is 1. In C, it’s 1. So, clipped count is min(1,1) = 1

“sat”: Max count in R is 0. In C, it’s 1. So, clipped count is min(1,0) = 0

“on”: Max count in R is 1. In C, it’s 1. So, clipped count is min(1,1) = 1

“mat”: Max count in R is 0. In C, it’s 1. So, clipped count is min(1,0) = 1

Total clipped unigram matches = 1 + 1 + 0 + 1 + 1 = 4



**Bigram (n=2) Precision (P2)**

**Bigrams in C:**

“the cat”: 1

“cat sat”: 1

“sat on”: 1

“on the”: 1

“the mat”: 1

Total bigrams in C: 5

**Bigrams in R:**

“a cat”: 1

“cat was”: 1

“was sitting”: 1

“sitting on”: 1

“on the”: 1

“the mat”: 1

**Clipped Count of bigrams in C (based on R):**

“the cat”: Max count in R is 0. Clipped: min(1,0) = 0

“cat sat”: Max count in R is 0. Clipped: min(1,0) = 0

“sat on”: Max count in R is 0. Clipped: min(1,0) = 0

“on the”: Max count in R is 1. Clipped: min(1,1) = 1

“the mat”: Max count in R is 1. Clipped: min(1,1) = 1

Total clipped bigram matches = 0 + 0 + 0 + 1 + 1 = 2



**Trigram (n=3) Precision (P3):**

**Trigrams in C:**

“the cat sat”: 1

“cat sat on”: 1

“sat on the”: 1

“on the mat”: 1

Total trigrams in C: 4

**Trigrams in R:**

“a cat was”: 1

“cat was sitting”: 1

“was sitting on”: 1

“sitting on the”: 1

“on the mat”: 1

**Clipped Count of trigrams in C (based on R):**

“the cat sat”: Max count in R is 0. Clipped: min(1,0) = 0

“cat sat on”: Max count in R is 0. Clipped: min(1,0) = 0

“sat on the”: Max count in R is 0. Clipped: min(1,0) = 0

“on the mat”: Max count in R is 0. Clipped: min(1,1) = 1

Total clipped trigram matches = 0 + 0 + 0 + 1 = 1



**4-gram (n=4) Precision (P4):**

**4-grams in C:**

“the cat sat on”: 1

“cat sat on the”: 1

“sat on the mat”: 1

Total 4-grams in C: 3

**4-grams in R:**

“a cat was sitting”: 1

“cat was sitting on”: 1

“was sitting on the”: 1

“sitting on the mat”: 1

**Clipped Count of 4-grams in C (based on R):**

“the cat sat on”: Max count in R is 0. Clipped: min(1,0) = 0

“cat sat on the”: Max count in R is 0. Clipped: min(1,0) = 0

“sat on the mat”: Max count in R is 0. Clipped: min(1,0) = 0

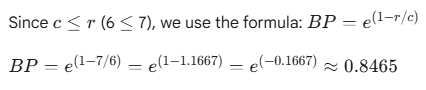
Total clipped 4-gram matches = 0 + 0 + 0 = 0



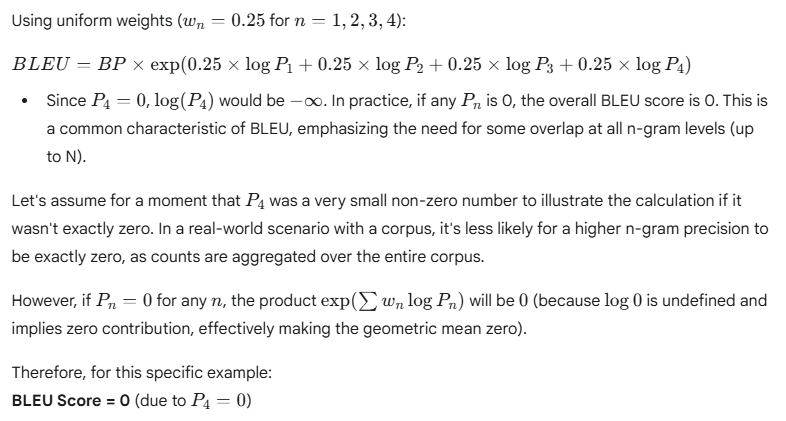
### Step 3: Calculation Brevity Penalty (BP)

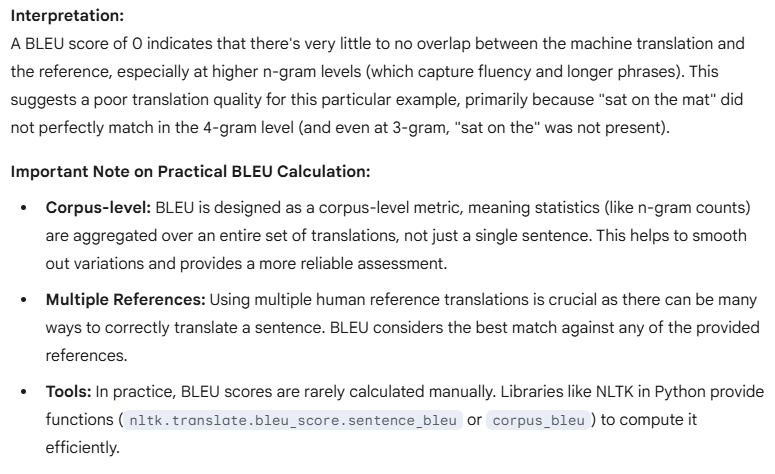
Length of Candidate ( C ) = 6

Length of Reference ( R ) = 7 (This is the length of the closest reference. Since there is only one, its 7)



### Step 4: Calculate the BLEU Score





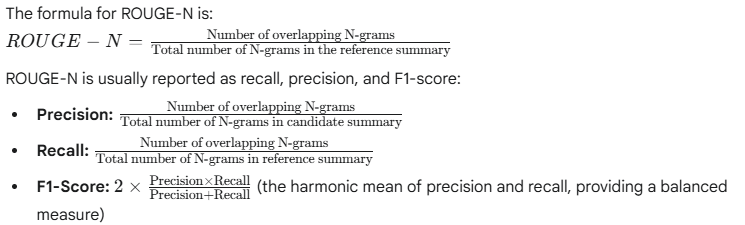
## ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

ROUGE is a set of metrics used for evaluating the quality of summaries and translations generated by machines, primarily in Natural Language Processing (NLP). Unlike BLEU, which focuses on precision (how much of the candidates is in the reference), ROUGE focuses on recall (how much of the reference is captured by the candidate). This makes it particularly well-suited for summarization tasks, where the goal is often to include as much important information from the original text as possible in the summary.

ROUGE compares an automatically produced summary or translation against a set of human-produced reference summaries or translations. Scores range from 0 to 1, with higher scores indicating greater similarity.

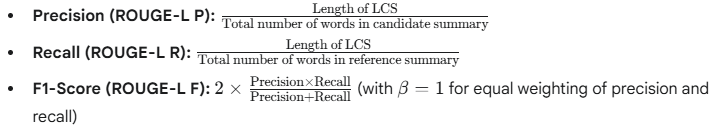
There are several types of ROUGE metrics, each focusing on a different aspect of overlap:

1. **ROUGE-N:** Measures the overlap of n-grams between the candidate and reference summaries.
   1. ROUGE-1: Overlap of unigrams (single words). This captures word-level similarity.
   2. ROUGE-2: Overlap of bigrams (pair of consecutive words). This captures the fluency and short phrase matches.
   3. ROUGE-3, ROUGE-4, etc.: Overlap of trigrams, 4-grams and so on. Higher N-grams capture longer consecutive matches, indicating better fluency and adherence to the reference phrasing.



1. **ROUGE-L:** Measures the Longest Common Subsequence (LCS) between the candidate and reference summaries. The LCS is the longest sequence of words that appear in both texts, in the same order, but not necessarily consecutively. This metric naturally accounts for sentence-level structure and word order, even if words are interrupted.

The formula for ROUGE-L is also typically based on Precision, Recall, and F1-score of the LCS length:



1. **ROUGE-S (Skip-bigram):** Measures the overlap of “skip-bigrams”. A skip-bigram is any pair of words in sentence order, allowing for arbitrary gaps between them. For example, in “the quick brown fox,” “the fox” is a skip-bigram with a skip distance of 2. This is useful for capturing paraphrasing or when word order might change slightly but the core concepts are still present.

## Example and Step-by-Step Calculation:

Let’s calculate ROUGE-1, ROUGE-2, and ROUGE-L for the following

Candidate Summary ( C ): “The cat sat on the mat”

Reference Summary ( R ): “A cat was sitting on the mat”

**Step-1: Tokenization and Normalization**

Convert sentences to lists of words (tokens) and typically lowercase them and remove punctuation for consistent comparison.

C : [“the”, “cat”, “sat”, “on”, “the”, “mat”] (length = 6 words)

R : [“a”, “cat”, “was”, “sitting”, “on”, “the”, “mat”] (length = 7 words)

**Calculation of ROUGE-1 (Unigram Overlap):**

**Step-2a: Identify Overlapping Unigrams**

Count the common unigrams (words) between C and R

Common words: “cat”, “on” “the”, “mat”

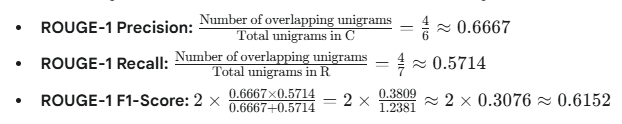
“the” appears twice in C, once in R. For overlap, we count the minimum occurrences. So, “the contributes 1 to the overlap.

Total overlapping unigrams: 4 (“cat”, “on”, “the” (1st instance), “mat”)

**Step-2b: Calculate Precision, Recall, and F1-score**

Total unigrams in Candidate ( C ): 6 (“the”, “cat”, “sat”, “on”, “the”, “mat”)

Total unigrams in Reference ( R ): 7 (“a”, “cat”, “was”, “sitting”, “on”, “the”, “mat”)



**Calculation of ROUGE-2 (Bigram Overlap)**

**Step-3a: Extract Bigrams**

**Bigrams in C:**

(“the”, “cat”)

(“cat”, “sat”)

(“sat”, “on”)

(“on”, “the”)

(“the”, “mat”)

Total bigrams in C: 5

**Bigrams in R:**

(“a”, “cat”)

(“cat”, “was”)

(“was”, “sitting”)

(“sitting”, “on”)

(“on”, “the”)

(“the”, “mat”)

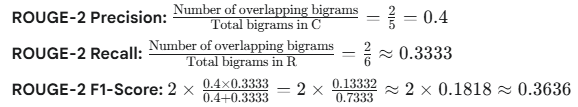
Total bigrams in R: 6

**Step-3b: Identify Overlapping Bigrams**

Common bigrams: (“on”, “the”), (“the”, “mat”)

Total overlapping bigrams: 2

**Step-3c: Calculate Precision, Recall, and F1-Score**



**Calculation of ROUGE-L (Longest Common Subsequence – LCS):**

**Step-4a: Find the Longest Common Subsequence (LCS)**

The LCS is the longest sequence of words that appear in both sentences in the same order, but not necessarily contiguously.

C: [“the”, “cat”, “sat”, “on”, “the”, “mat”]

R: [“a”, “cat”, “was”, “sitting”, “on”, “the”, “mat”]

Let’s trace:

“cat” is common.

After “cat”, “sat” in C does not match “was”, “sitting” in R.

“on” is common

“the” is common

“mat” is common

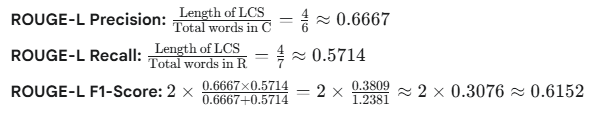
The LCS is: “cat”, “on”, “the”, “mat”

Length of LCS = 4 words.

**Step-4b: Calculate Precision, Recall, and F1-Score**

Total words in Candidate ( C ): 6

Total words in Candidate ( R ): 7



**Summary of Scores for this example:**

**ROUGE-1 F1**: 0.6152

**ROUGE-2 F1:** 0.3636

**ROUGE-L F1:** 0.6152

**Interpretation:**

* A ROUGE-1 score of around 0.62 indicates a decent word overlap
* A lower ROUGE-2 score of around 0.36 suggests that while many individual words matches, the specific two-word phrase did not align as well, indicating some difference in sentence structure or phrasing.
* ROUGE-L being similar to ROUG-1 implies that the common words were largely in the correct relative order, even if there were insertions or deletion that affected consecutive n-grams.

Like BLEU, ROUGE is typically computed over a large corpus and often against multiple reference summaries to provide a more robust evaluation. Python libraries like rouge\_score or evaluate (from Hugging Face) are commonly used for practical calculations.

## Ethical Foundation and Governance in Generative AI

Generative AI (GenAI) presents immense opportunities but also unique ethical challenges due to its ability to create new content. Establishing strong ethical foundations and robust governance strategies is crucial for its responsible development deployment.

### Ethical Foundations in Generative AI

The ethical foundations of GenAI are rooted in broader AI ethics, but with specific emphasis on aspects related content generation. Key principles include:

1. **Transparency and Explainability**

Disclosure: Users should be informed when content is AI generated (e.g., text, images, audio, video). This helps maintain trust and prevents deception.

Traceability: Where feasible, AI systems should provide insights into their logic or justifications, especially in high-stakes contexts. This helps users understand capabilities, limitations and intended use.

Documentation: Clear documentation of training data, model architecture, and deployment decisions is essential for understanding how the system works

1. **Fairness and Non-discrimination**

Bias Mitigation: GenAI models must be trained on diverse and representative datasets to avoid perpetuating or amplifying harmful biases related to race, gender, culture, or socio-economic status.

Regular Auditing: Continuous testing and auditing are needed to detect and mitigate algorithmic bias, ensuring equitable outcomes and preventing the marginalization of vulnerable groups.

Inclusivity: AI systems should be designed to empower everyone and engage all people, regardless of their backgrounds and reflect a vast range of human identities and experiences.

1. **Accountability and Human Oversight**

Huma-in-the-loop: In sensitive domains (e.g., healthcare, education, finance), mechanisms for human oversight and intervention are critical. Humans must be able to monitor generated content and intervene when necessary.

Responsibility: Clear designation of responsibility for managing AI outcomes, including legal and ethical implications, is vital. Organizations and individuals must be accountable for the AI systems they develop and deploy.

Audit Trails: Logging prompts, model responses, user interventions, and decision rationales ensures transparency and accountability.

1. **Data Privacy and Informed Consent**

Ethical Data Sourcing: GenAI should only use data that has been ethically sourced with explicit consent.

Output Privacy: Systems must be designed not to produce outputs that compromise individual privacy. Techniques like differential privacy and federated learning can enhance data protection.

Secure Handling: Implementing robust data security and privacy standards to mitigate risks from breaches, unauthorized access, and non-compliance.

1. **Safety and Robustness**

Harm Prevention: GenAI systems should be designed to prevent the generation of harmful outputs, such as misinformation, hate speck, or inappropriate material.

Vulnerability Testing: Rigorous testing for vulnerabilities like adversarial prompts, phishing potential, or disinformation generation is essential.

Reliability: AI systems should perform reliably and safely under various conditions, with mechanisms to override, repair, or decommission them if they pose undue harm.

1. **Intellectual Property and Creative Integrity**

Copyright Compliance: Users and developers must comply with copyright laws and avoid plagiarizing original work.

Attribution: When referencing existing works, proper attribution should be provided.

Acknowledging AI Use: Users should clearly acknowledge when AI is used to produce work fostering trust and credibility.

1. **Environment Impact**

Energy Consumption: Acknowledging the high power consumption required for training large GenAI models and exploring methods for energy efficient AI development.

### Governance Strategies in Generative AI

Governance strategies translate ethical principles into actionable frameworks and practices. They involve defining policies, roles and processes to ensure responsible AI development and deployment.

1. **Establishing a Comprehensive AI Governance Framework**

Policy Development: Create clear, organization-wide policies defining acceptable and prohibited uses of GenAI, aligned with organizational values, legal constraints, and industry standards.

Risk Assessment and Management: Implement structured frameworks to identify asses, and mitigate operational, technical, and reputational risks associated with GenAI. This includes continuous monitoring for biases, security vulnerabilities, and ethical violations

Clear Roles and Responsibilities:  Define specific roles and responsibilities for individuals and teams involved in the GenAI lifecycle, from development to deployment and monitoring.

**Governance Committee/Board:** Establish an interdisciplinary committee (including legal, compliance, data science, product, and operations) to oversee GenAI initiatives and vet new deployments.

1. **Lifecycle Management and Monitoring**

Model Monitoring: Continuously track model performance, behavioral drift, and output accuracy over time.

Audit Trails and Documentation: Maintain detailed logs of prompts, model responses, user interventions, and decision rationales to ensure transparency, accountability, and compliance readiness.

Continuous Improvement: Implement mechanisms for ongoing evaluation and improvement of AI models, adapting to changing circumstances, new risks, and evolving ethical standards.

1. **Data Governance**

Quality and Integrity: Focus on the availability of high-quality, well-annotated data to minimize errors and biases in GenAI outputs.

Access Control: Implement strict access control policies, including role-based access, to sensitive tools and data used for training and inference.

Encryption and Anonymization: Utilize data encryption and anonymization techniques to protect sensitive information.

Data Lineage: Implement tools to track the lineage of data used in GenAI models.

1. **Regulatory Compliance**

Stay Updated: Continuously monitor and align with evolving regional and international AI regulations and standards (e.g., EU AI Act, NIST AI Risk Management Framework, OECD AI Principles, GDPR).

Legal Risk Assessment: Conduct regular assessments to identify and address potential legal risks associated with GenAI use, including copyright infringement and privacy violations.

1. **Organizational Culture and Training**

Ethical AI Education: Provide regular training for all employees involved with GenAI on ethical considerations, bias awareness, responsible use, and compliance policies.

Culture of Responsibility: Foster a culture within the organization that prioritizes ethical AI development and deployment, encouraging open discussion and reporting of concerns.

Stakeholder Engagement: Engage diverse stakeholders, including the public, in discussions about AI policy decisions and the direction of technological developments to ensure governance frameworks reflect societal values.

1. **External Auditing and Red Teaming**

Independent Audits: Conduct independent audits for high-impact or large-scale GenAI deployments to verify compliance with ethical guidelines and identify potential issues.

Red Teaming: Proactively test GenAI systems for vulnerabilities and potential misuse by simulating adversarial attacks to identify weaknesses and implement safeguards.

By integrating these ethical foundations and governance strategies, organizations can work towards building and deploying generative AI systems that are not only innovative and powerful but also responsible, fair, and beneficial to society.

## Lab: Build a mini GPT-like transformer using Hugging Face

Building a “mini GPT-like transformer” from scratch with Hugging Face to understand the core components of large language models. The Hugging Face transformers library make this process much more accessible than implementing everything from scratch.

Here’s a step by step guide to building and training a simple, small causal language model (like GPT) using Hugging Face. We will focus on the essential Components:

**Tokenizer, Model Configuration, Model Architecture, Dataset Preparation, and Training.**

**Prerequisites:**

Make sure you have necessary libraries installed.

pip install transformers datasets accelerate tokenizers torch

accelerate is good for efficient training, and tokenizers is for custom tokenizer training. torch is the backend of our model.

Step-1: Prepare Your Dataset

Step 2: Train a Custom Tokenizer

Step 3: Configure Your Mini GPT Model

Step 4: Preprocess the Dataset for Training

Step 5: Set up Training Arguments and Trainer

Step 6: Generate Text with Your Mini GPT



## Lab: Practical Implementation of Fine-tuning an LLM with LoRA using Python

(Here's the complete Python script for fine-tuning an LLM with LoRA using your custom text file, along with instructions on how to run it.)



# Module-2: Retrieval-Augmented Generation (RAG) Systems

Retrieval Augmented Generation (RAG) is an AI framework that enhances the capabilities of Large Language Models (LLMs) by giving them access to external, up-to-date, and domain specific information beyond their initial training data. This helps LLMs generate more accurate, relevant, and grounded responses, and reduces issues like “hallucinations” (Where LLMs produce plausible but factually incorrect information).

Here's a breakdown of how RAG works and why it’s beneficial:

## What RAG is Needed:

**LLMs have limitations:** Traditional LLMs are trained on massive datasets, but their knowledge is static – it’s limited to the information they were trained on up to a certain cutoff date. This means they can provide outdated information or “hallucinate” when asked about specific, real-time or proprietary data.

**Cost and complexity of retraining:** Retraining or fine-tuning an LLM with new data is computationally expensive and time consuming. RAG offers a more cost-effective and agile solution.

**Need for factual accuracy and context:** Many applications require LLMs to provide factually accurate and contextually relevant responses, especially in domain like customer service, research, or internal knowledge management.

## How RAG works (The Process):

### Data Preparation (External Knowledge Base):

**Collect Data:** Gather relevant external data sources, which can include documents, databases, web pages, APIs, internal company policies etc., This data can be in various formats.

**Chunking:** The collected data is broken into smaller, manageable “chunks” (e.g., sentences, paragraphs). This helps in retrieving highly specific information.

**Embedding:** Each chunk of text is converted into a numerical representation called a “vector embedding”. These embeddings capture the semantic meaning of the text.

**Vector Database:** These vector embeddings are stored in a specialized database called a vector database, which allows for efficient similarity searches.

### User Query Handling (Retrieval Phase):

**User Input:** When a user submits a query (e.g., a question), this query is also converted into a vector embedding using the same embedding model used for the external data.

**Similarity Search:** They system performs a similarity search in the vector database to find the chunks of external data whose embeddings are most similar to the user’s query embeddings. This identifies the most relevant information.

Ranking (Optional): Retrieved documents might be further ranked based on their relevance to ensure the most pertinent information is selected.

### Augmented Generation Phase:

**Prompt Augmentation:** The retrieved relevant information (the “chunks”) is then combined with the original user query. This augmented prompt provides the LLM with necessary context and facts. This is sometimes called “prompt stuffing”

**LLM Generation:** The augmented prompt is fed to the LLM. The LLM uses this new, external information, along with its pre-existing training data, to generate a comprehensive, accurate, and contextually relevant response.

**Output:** The LLM delivers the generated response, often with citations to the external sources for verification.

## Key Benefits of RAG:

* **Improved accuracy and Factual Grounding:** RAG significantly reduces “hallucinations” by grounding the LLM’s responses in verifiable, external information.
* **Access to Up-to-Date information:** RAG enables LLMs to access real-time data and proprietary knowledge bases, keeping responses current and relevant.
* **Cost-Effective:** It’s more efficient and less expensive than constantly retraining or fine-tuning LLMs with new data.
* **Domain-Specific Knowledge:** RAG allows LLMs to specialize in particular domains by providing them with relevant knowledge bases, without needing to modify the core model.
* **Transparency and Trust:** By providing sources, RAG allows users to verify the information increasing trust in the AI’s output.
* **Reduced Bias:** By relying on vetted external sources, RAG can help mitigate biases present in the LLMs original training data.

## Applications of RAG:

RAG is widely used in various applications, including:

* **Question-Answering Systems:** Building smart chatbots and virtual assistants that can answer specific questions based on an organization’s internal documents or up-to-date external information.
* **Content Generation:** Generating reports, articles, or summaries that require factual accuracy and current data.
* **Customer Service:** Providing accurate and personalized responses to customer queries based on product manuals, FAQs and customer data.
* **Internal Knowledge Management:** Allowing employees to quickly access information from internal documents and databases.
* **Research:** Assisting research in synthesizing information from large corpuses of academic papers.

In essence, RAG acts as an “open-book exam” for LLMs, allowing them to refer to a vast library of knowledge before providing an answer, making them more powerful, reliable and versatile.

## Embedding Techniques

Embeddings are numerical representations (vectors) of real-world objects like text, images, audio, or even entire documents. The magic of embeddings lies in their ability to capture the semantic meaning and relationships between these objects. In a high-dimensional vector space, similar items are located closer to each other, while dissimilar items are farther apart.

### How Embeddings Work:

1. Transformation: An embedding model takes raw data (e.g., a sentence) and transforms it into a fixed-length array of numbers (the vector)
2. Semantic capture: During the training of these models, they learn to encode the meaning, context and relationships of the input data. For instance, in text embeddings, words like “king” and “queen” would have similar vectors and the relationship between “king” and “man” might be similar to “queen” and “woman” in terms of vector arithmetic.
3. Dimensionality Reduction (Implicit): While the original data might be very high-dimensional (e.g., individual pixels in an image, or a vast vocabulary of words), embeddings typically reduce this to a much lower, more manageable dimension (e.g., 128, 768, or 1536 dimensions), while retaining essential information.

## Why Embeddings are Crucial:

* **Machine Readability:** ML models can’t directly understand raw text or images. Embeddings convert them into a numerical format that models can process.
* **Semantic Search:** Enables searching based on meaning rather than just keywords. If you search for “fast car”, an embedding-based search can also find documents talking about “speedy automobiles”
* **Similarity and Relationships:** Allows for easy comparison of items. Similar items have similar vectors.
* **Foundation for AI tasks:** Essential for tasks like semantic search, recommendation systems, anomaly detection, clustering and Retrieval Augmented Generation (RAG)

## Popular Embedding Models/Techniques:

### OpenAI Embeddings (e.g., text-embedding-3-large, text-embedding-ada-002):

**Description:** These are highly powerful, transformer-based embedding models provided by OpenAI via their API. They are trained on vast amounts of diverse text data.

**Strengths:**

* **High Semantic Accuracy:** Excellent at capturing nuanced semantic relationships and understanding context.
* **General Purpose:** Perform well across a wide range of NLO tasks
* **Ease of Use:** Simple API integration.

**Weakness:**

* **API-based:** Requires an internet connection and incurs costs per token
* **Lack of Transparency:** Models are proprietary, so you can’t fine-tune them directly or inspect their internal workings.

**Ideal Use Cases:** Semantic search, RAG pipelines, question-answering, document clustering, and classification where high accuracy and ease of integration are paramount.

### Cohere Embeddings (e.g., embed-english-v3, embed-multiligual-v3):

**Description:** Cohere specializes in large language models and offers robust, high-quality embedding model through their API. They are known of their performance in semantic tasks and efficiency.

**Strengths:**

* **High quality Semantic embeddings:** Optimized for semantic search, retrieval and classification.
* **Flexible Output Types:** Support various embedding types like float, int8, binary and ubinary, which can save memory and improve retrieval speed.
* **API-driven:** Easy to integrate into applications

**Weakness:**

* **API-based:** Similar to OpenAI, requires and API and associated costs.

**Ideal Use cases:** Semantic search, RAG pipelines, document classification, and application that benefit from efficient embedding types.

### FastEmbed (by Qdarnt):

**Description:** FastEmbed is lightweight, performant, and local-first Python library that provides access to state-of-the-art embedding models. It leverages optimized ONNX models from Hugging Face.

**Strengths:**

* **Local Execution:** Runs locally on your machine, no internet connection or API keys required, ensuring data privacy and reducing latency
* **Fast and Efficient:** Optimized for speed and low memory footprint.
* **Open-Source Models:** Utilizes various open-source transformer models (e.g., from SentenceTransformers family)
* **Cost-Effective:** No per-token costs as it runs locally.

**Weaknesses:**

* **Model Selection:** While it uses good open-source models, they might not always match the absolute cutting-edge performance of the largest proprietary models from OpenAI/Cohere for all tasks.

**Ideal Use Cases:** Projects requiring local processing, offline capabilities, privacy-sensitive applications, cost-sensitive environments, and rapid prototyping of RAG systems.

## Vector Databases: Storing and Searching Embeddings

A Vector Database (also known as vector store or vector search engine) is a specialized database designed to store, manage and index high-dimensional vector embeddings. Its primary purpose is to enable extremely fast and efficient similarity searches (also called “nearest neighbor” or “approximate nearest neighbor” – ANN search).

### How Vector Databases Work:

1. **Ingestion & Vectorization:** Raw data (text, images, etc.,) is first converted into vector embeddings using an embedding model.
2. **Storage:** These numerical vectors are stored in the vector database. Often, metadata associated with the original data (e.g., title, author, timestamp) is also stored alongside the vector or linked to it.
3. **Indexing:** To enable fast searches in high-dimensional spaces, vector databases use specialized indexing algorithms. Unlike traditional databases that use B-trees or hash tables, vector database employ techniques like:

Hierarchical Navigable Small Worlds (HNSW): Creates a graph structure where each node is a vector and connections represent proximity. Efficiently traverses the graph to find neighbors.

Inverted File Index (IVF): Divides the vector space into clusters and stores inverted lists of vectors within each cluster.

Product Quantization (PQ): Compresses vectors to reduce memory footprint and speed up distance calculations.

These algorithms allow for Approximate Nearest Neighbor (ANN) search, which sacrifices a tiny bit of accuracy for massive speed improvements, especially with billions of vectors.

1. **Querying:** When a user submits a query (e.g., “What is a vector database?”), that query is also converted into an embedding. This “query vector” is then sent to the vector database.
2. **Similarity Search:** The database uses its indexing structures and distance metrics (e.g., cosine similarity, Euclidean Distance, dot product) to find the ‘k’ most similar vectors to the query vector. These similar vectors correspond to the most relevant data chunks.
3. **Retrieval:** The actual data (or a pointer to it) associated with the top “k” similar vectors is retrieved and passed on (e.g., to an LLM in a RAG system)

### Why Vector Databases are Crucial:

**Scalability:** Efficiently handles vast amounts of high-dimensional data (billions of vectors)

**Performance:** Enables real-time or near real-time similarity searches, which is critical for interactive AI applications.

**Semantic Search:** Powers the ability to search by meaning, overcoming the limitations of keyword-based search.

**Core of RAG:** They are the “memory” or “knowledge base” that LLMs can query to get external relevant information.

### Popular Vector Databases:

#### FAISS (Facebook AI Similarity Search):

Description: Developed by Meta FAISS is an open-source library (not a standalone database) for efficient similarity search and clustering of dense vectors. It’s written in C++ with Python wrappers.

**Strengths:**

* **Extremely Fast:** Optimized for performance and can handle billions of vectors, leveraging GPUs for accelerated search.
* **Highly Flexible:** Offers a wide array of indexing algorithms (IVF, HNSW, Product Quantization etc.) that can be combined and configured.
* **Memory Efficient:** Provides techniques to compress vectors for lower memory usage.
* **Battle-tested:** Used in production by Meta for large-scale similarity tasks

**Weaknesses:**

* **Library, Not a Database:** Lacks traditional database features like persistence, data management, direct filtering on metadata, or network APIs. You need to manage storage and access around it.
* **Steep Learning Curve:** Can be complex to set up ad optimize for specific use cases.

**Ideal Use Cases:** Pure raw, high-performance similarity search within an existing ML/AI pipeline, particularly for very large-scale datasets where you manage data persistence separately.

#### Weaviate:

**Description:** Weaviate is an open-source, cloud-native vector database the combines vector search with a knowledge graph. It’s designed to be a full-fledged database solution for AI applications.

**Strengths:**

* **Hybrid Search:** Supports both semantic (vector) search and symbolic (keyword/lexical) search, often allowing for more refined queries.
* **Schema-based:** Allows defining a schema for your data, including metadata, enabling powerful structured filtering alongside vector search.
* **GraphQL API:** Provides a powerful and flexible API for querying.
* **Scalable:** Built for horizontal scalability to handle large deployments.
* **Modular ML integration:** Supports on-the-fly vectorization and multi-modal data (text, images etc)

**Weaknesses:**

* Can be more resource-intensive than simpler libraries like FAISS
* The schema and knowledge graph features might add complexity if you only need basic vector search.

Ideal Use Cases: Building semantic search engines, recommendation systems, and RAG applications that require sophisticated querying, structured filtering, and potentially multi-modal data, especially for enterprise-level deployments.

#### Chroma

**Description:** Chroma is an Open-Source Embedding database that focuses on being lightweight and developer-friendly. It aims to simplify the process of building AI applications with embeddings.

**Strengths:**

* **Easy to use:** Designed for Quick setup and integration, making it popular for prototyping and smaller-to-medium scale projects.
* **Developer-friendly:** Python-native API that’s intuitive
* **Integrates Well:** Good integration with popular ML frameworks and libraries like LangChain and LlamaIndex
* **Lightweight:** Can run embedded within an application (in-memory) or as a client-server.

**Weaknesses:**

* **Scalability:** While it can scale, it might not offer the same raw performance or handle billions of vectors as efficiently as FAISS or large Weaviate Deployments
* May lack some of the advanced features (like hybrid search or complex schema definitions) or more comprehensive vector databases.

**Ideal Use Cases:** Quick prototyping of RAG pipelines, small to medium-sized AI-powered search or QA Systems, and educational projects where ease to use and rapid deployments are prioritized.

## LangChain Pipeline Construction for RAG

LangChain is a powerful framework designed to simplify the development of applications powered by Large Language Models (LLMs). It provides a structured way to connect LLMs with external data sources, enabling complex workflows like Retrieval Augmented Generation (RAG). Let’s break down the construction of a LangChain RAG pipeline, focusing on multi-folder ingestion, chunking, and retriever + LLM orchestration.

A typical RAG pipeline in LangChain involves two main phases: Ingestion (or Indexing) and Querying.

### Phase-1: Ingestion (Building the Knowledge Base)

This phase focuses on taking your raw data, processing it, and storing it in a way that allows for efficient retrieval.

#### Multi-Folder Ingestion (Document Loading)

The first step is to load your data into LangChain’s Document format. LangChain provides various DocumentLoaders to handle different data sources. For multi-folder ingestion, you would typically iterate through your directories and load files based on their type.

Concept: DocumentLoaders are responsible for reading data from a source (e.g., PDF, TXT, HTML databases) and converting it into LangChain’s Document objects. A Document object usually has page\_content (the text itself) and metadata (additional information like file path, title, etc.,)

**Multi-folder strategy:**

Globbing/Directory Loader: LangChain has DirectoryLoader which can load documents from a specified directory. You can combine this with Glob patterns to select specific file types or traverse subdirectories.

Custom Script: For more complex scenarios (e.g., conditional loading based on file names, specific parsing logic per folder), you can write a Python script that iterates through your folders using os.walk or pathlib, identifies files, and then calls the appropriate LangChain DocumentLoader for each file.

**Example (Conceptual):**

import os

from langchain\_community.document\_loaders import PyPDFLoader, TextLoader, CSVLoader

from langchain\_core.documents import Document

def load\_documents\_from\_folders(base\_path):

all\_documents = []

for root, \_, files in os.walk(base\_path):

for file in files:

file\_path = os.path.join(root, file)

loader = None

if file.endswith(".pdf"):

loader = PyPDFLoader(file\_path)

elif file.endswith(".txt"):

loader = TextLoader(file\_path)

elif file.endswith(".csv"):

loader = CSVLoader(file\_path)

# Add more loaders for other file types as needed

if loader:

try:

docs = loader.load()

# Optionally add custom metadata (e.g., original folder path)

for doc in docs:

doc.metadata["source\_folder"] = root

all\_documents.extend(docs)

except Exception as e:

print(f"Error loading {file\_path}: {e}")

return all\_documents

# Example usage:

# documents = load\_documents\_from\_folders("./my\_data\_directory")

#### Chunking (Text Splitting)

Once documents are loaded, they are often too large to fit into an LLM’s context window or to be effectively searched. Chunking is the process of breaking down these large documents into smaller semantically meaningful pieces.

**Concept:** TextSplitters in LangChain divide a Document into smaller Document objects (chunks), aiming to preserve contextual integrity.

##### Why Chunking is important:

* **LLM Context Window:** LLMs have token limits. Chunking ensures that the retrieved pieces of information fit within these limits when passed to the LLM.
* **Relevance:** Smaller, more focused chunks are more likely to be highly relevant to a specific query, improving retrieval accuracy.
* **Efficiency:** Searching through smaller chunks is faster and more memory-efficient vector databases.

##### Common Chunking Strategies (LangChain TextSplitters):

* RecursiveCharacterTextSplitter:This is often the default and most recommended. It attempt to split text using a lit of separators ([“\n\n”, “\n”, “ ”, “”]) in order, trying to keep chunks together until a certain chunk\_size is exceeded. It then moves to the next separator. This helps preserve semantic meaning by prioritizing paragraph breaks over sentence breaks,etc.
  + **Chunk\_size:** The maximum number of characters (or tokens, depending on the length function) for a chunk
  + **Chunk\_overlap:** The number of characters (or tokens) that overlap between consecutive chunks. This help maintain context across chunks
* CharcterTextSplitter: A simpler splitter that divides text by character (e.g., “\n\n”, “”). Less sophisticated than RecursiveCharacterTextSplitter but useful for basic splitting.
* SentenceTransformersTokenTextSplitter / TokenTextSplitter : Splites text based on tokens, often using a pre-trained tokenizer from models like SentenceTransformers. Useful for ensuring chunks adhere to token limits of specific embedding models.
* Markdown/HTML/JSON Splitter: Specialized splitters (MarkdownHeaderTextSplitter, HTMLHeaderTextSplitter, RecursiveJsonSplitter) that understand the structure of these formats and split accordingly, preserving logical sections.
* Sematic Chunking (Advanced): This method involves embedding sentences or paragraphs and then splitting based on semantic similarity. If the semantic similarity between adjacent segments drops below a threshold, it indicates a good breakpoint. LangChain provides tools that can integrate with this, but it often involves a custom approach using embedding models.

Example (Conceptual)

from langchain.text\_splitter import RecursiveCharacterTextSplitter

# Assuming 'documents' is a list of LangChain Document objects from ingestion

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=1000,

chunk\_overlap=200,

length\_function=len, # Use len for character count, or a tokenizer for token count

add\_start\_index=True,

)

chunks = text\_splitter.split\_documents(documents)

#### Embedding and Vector Storage

After chunking, each chunk is converted into a numerical vector (embedding), and these vectors are stored in a vector database.

**Concept:**

**Embeddings:** As discussed previously, these are numerical representational of text that capture semantic meaning. LangChain provides an Embedding interface.

**Vector Store:** A database optimized for storing and querying vector embeddings using similarity search. LangChain integrates with many vector stores (FAISS, Weaviate, Chroma, Pinecone etc)

**Process:**

1. Initialize Embedding Model: Choose an Embedding modle (OpenAI, Cohere, HuggingFace Embeddings via HuggingFaceEmbeddings)
2. Initialize Vector Store: Select and initialize your chosen vector database.
3. Add Documents: Pass your chunks and embedding model to the vector store to generate embeddings and store them.

from langchain\_openai import OpenAIEmbeddings # Or from langchain\_community.embeddings import FastEmbedEmbeddings

from langchain\_community.vectorstores import Chroma # Or FAISS, Weaviate, etc.

**Concept**

# 1. Initialize Embedding Model

embeddings = OpenAIEmbeddings(model="text-embedding-ada-002") # Or FastEmbedEmbeddings(model\_name="BAAI/bge-small-en-v1.5")

# 2. Create/Load Vector Store

# For a new collection:

vectorstore = Chroma.from\_documents(chunks, embeddings, collection\_name="my\_rag\_collection")

# For an existing collection:

# vectorstore = Chroma(embedding\_function=embeddings, collection\_name="my\_rag\_collection")

### Phase 2: Querying (Retrieval and Generation)

This phase handles user queries, retrieves relevant information, and uses LLM to generate a response.

#### Retriever

The Retriever is the component responsible for fetching relevant documents (chunks) from the vector store based on a user’s query.

**Concept:** LangChain’s Retriever interface takes a query string and returns a list of Document objects that are most relevant.

**Types of Retrievers (and Orchestration):**

* **Vector Store Retriever (most common):** This is the simplest and most widely used. It directly interfaces with a vector store (like Chroma or FAISS) and performs a similarity search with the query embeddings to retrieve-top-k similar chunks.

Concept:

# Get a retriever from the vector store

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 5}) # Retrieve top 5 similar chunks

* Multi-Query Retriever: Generates multiple variations of the user’s query (e.g,, paraphrases, decomposed sub-questions) using an LLM, then perform retrieval for each generated query, and combines the results. This can improve recall.
* Contextual Compression Retriever: Retrieves more documents initially and then uses a BaseLLMCompressor (like LLMChainExtractor or LLMChainFilter) to filter or summarize the retrieved documents, passing only the most salient information to the final LLM.
* Parent Document Retriever: Indexes smaller chunks for granular search but retrievers the larger “parent” document to provide more context to the LLM. Useful when smaller chunks are better for search but the LLM needs a broader view.
* Ensemble Retriever: Combines multiple different retrievers (e.g., vector store retriever and a keyword-based retriever like BM2Retriever) and merges their results, often with weighted scores. This can leverage the strengths of different retrieval methods.
* Self-Query Retriever: Allows the LLM to construct a structured query (e.g., with metadata filters) for the vector store based on the natural language user query. This enables more precise retrieval when metadata is rich.
* Time-weighted VectorStore Retriever: Prioritizes more recent documents in its similarity search.

#### LLM Orchestration (Chains and LCEL)

LangChain provides “Chains” and the LangChain Expression Language (LCEL) to orchestrate the flow of data from the retriever to the LLM and beyond.

**Concept:**

* **LLM: The** Large Language Model itself (e.g., ChatOpenAI, Ollama)
* **Prompt Template:** Defines the structure of the input given to the LLM, including placeholders for the user’s question and the retrieved context.
* **Chain:** A sequence of components that process input and produce output. LangChain’s chain simplify complex LLM workflows
* **LangChain Expression Language (LCEL):** A declarative way to build complex, highly flexible, and production-ready chains by composing runnables using the pipe (|) operator. If offers features like streaming, async support and easy parallelism.

Orchestration Steps:

1. Define the LLM:

from langchain\_openai import ChatOpenAI # Or from langchain\_community.llms import Ollama

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.1) # or Ollama(model="llama3")

1. Define the Prompt Template: This is crucial for guiding the LLM. It typically includes the retrieved context and the user’s question.

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder

template = """You are an AI assistant for answering questions about documents.

Use the following retrieved context to answer the question.

If you don't know the answer, just say that you don't know, don't try to make up an answer.

Use as much detail as possible from the context.

{context}

Question: {question}

"""

prompt = ChatPromptTemplate.from\_template(template)

1. Build the RAG Chain (using LCEL): This is where the retriever and LLM are combined.

from langchain\_core.runnables import RunnablePassthrough, RunnableParallel

from langchain\_core.output\_parsers import StrOutputParser

# Helper function to format retrieved documents for the prompt

def format\_docs(docs):

return "\n\n".join(doc.page\_content for doc in docs)

rag\_chain = (

RunnableParallel(

{"context": retriever | format\_docs, "question": RunnablePassthrough()}

)

| prompt

| llm

| StrOutputParser()

)

# Example of how the chain works:

# 1. RunnableParallel executes 'context' and 'question' branches concurrently.

# - 'context' branch: Takes the input question, passes it to the 'retriever',

# then 'format\_docs' formats the retrieved documents into a single string.

# - 'question' branch: Simply passes the original input question through.

# 2. The combined output (a dict with 'context' and 'question' keys) is passed to the 'prompt'.

# 3. The prompt template is rendered with the context and question.

# 4. The rendered prompt is sent to the 'llm'.

# 5. The LLM's raw output is passed to 'StrOutputParser' to get a clean string answer.

1. Invoke the Chain

# query = "What are the main features of LangChain?"

# response = rag\_chain.invoke(query)

# print(response)

### Summary of the LangChain RAG Pipeline Flow:

1. **Load Documents:** Raw data from various sources (multi-folder) is loaded into Document objects using DocumentLoaders.
2. **Chunk Documents:** Large documents are broken into smaller, contextually coherent chunks using TextSplitters.
3. **Create Embeddings & Store:** Each chunk is converted into a vector embedding using an Embeddings model, and these embeddings (along with metadata) are stored in a VectorStore.
4. **User Query:** A user submits a question.
5. **Retrieve Documents:** The Retriever (often built on top of the VectorStore) performs a similarity search with the query's embedding to find the most relevant chunks.
6. **Augment Prompt:** The retrieved chunks are incorporated into a PromptTemplate along with the original user query.
7. **Generate Response:** The augmented prompt is sent to the LLM, which generates a natural language response based on the provided context.
8. **Output:** The LLM's response is presented to the user.

LangChain's modular design and LCEL make it highly flexible, allowing developers to experiment with different loaders, splitters, embedding models, vector stores, retrievers, and LLMs to optimize their RAG pipeline for specific use cases and data types.

**Lab:** Here is the code and associated files



## Lab: RAG Summarization Pipeline with fallback

1. Imports and Setup
2. Ingestion Phase : Loading and chunking of documents, Using Chroma for efficient storage and retrieval of document embeddings
3. LLMs & Prompt Definitions: Leveraging Ollama to run Large Language Models locally for both generation and sub-tasks
4. Retrieval Layer Functions: Multi-Query Retrieval to improve recall by generating multiple interpretations of a query. Contextual Compression – Enhancing precision by extracting only the most relevant parts of retrieved documents
5. Fallback Layer
6. Orchestration with LangChain Expression Language (LCEL)
7. Main Execution of Chatbot



# Module-3: Agentic AI and Autonomous Task Planning

Agentic AI and Autonomous Task planning are two closely and often overlapping concepts in the field of Artificial Intelligence, both aiming to enable AI systems to perform tasks with increasing levels of independence and intelligence.

## Agentic AI

Agentic AI refers to AI systems designed to act as “agents” that can perceive their environment, reason about their goals, make decisions, take actions, and learn from their experiences to achieve those goals with minimum human intervention. Unlike traditional AI, which often operates on a direct input-output model, agentic AI takes initiative, breaks down complex problems, and orchestrates various tools and sub-tasks to reach a higher-level objective.

AI agent is more than just a piece of code that responds to an input; its an autonomous entity that can perceive, reason, act and learn to achieve goals in its environment.

Let’s break down how memory, planning, tools and reflexes contribute to this agentic behaviour.

1. **Memory:**

The ability of an AI agent to store and recall past experiences, observations, and decisions. This can range from short-term memory (context within a single conversation or task) to long-term memory (knowledge accumulated over many interactions or persistent knowledge base.)

**Why its crucial for an agent:**

**Context and Coherence:** Allows the agent to maintain context over multi-turn conversations or multi-step tasks. Without memory, each interaction would be treated as isolated, leading to repetitive questions or nonsensical responses.

**Learning and Adaptation:** Enables the agent to learn from past successes and failures. It can recall what worked or did not work in similar situations and adjust its future behaviour accordingly. This is vital for improvement over time.

**Personalization:** For user-facing agents, memory allows for a personalized experience, remembering user preferences, past interactions, and specific needs.

**Knowledge Base:** Long-term memory often forms the basis of knowledge base, allowing the agent to access factual information or learned procedures.

**Examples:**

* A customer service AI remembering previous support tickets and resolutions for a specific user.
* A coding agent recalling previously written code snippets or design patterns
* A chatbot remembering the topic of the current conversation to stay on track.

1. **Planning:**

The capability of an AI agent to strategize and determine a sequence of actions to achieve a specific goal. This involves breaking down complex goals into smaller sub-tasks, considering preconditions and effects of actions, and anticipating potential outcomes.

**Why its crucial for an agent:**

**Goal-Oriented Behavior:** Allows the agent to be proactive and pursue long-term objectives rather than just reacting to immediate stimuli.

**Complex Task Handling:** Enables the agent to tackle multi-step problems that required a series of coordinated actions.

**Efficiency and Optimization:** Helps the agent find the most efficient or optimal path to a goal, considering constraints or resources.

**Problem Solving:** If an initial plan fails, a capable agent can often replan or adjust its strategy.

**Examples:**

* An agent tasked with “book me a flight to London next month” might plan steps like – Check Dates, Compare Prices on different Airlines, Find accommodation, Book Tickets, Send confirmation.
* A Content generation agent planning an article outline before writing the content
* A robotic arm planning a sequence of movements to assemble a product.

1. **Tools:**

The ability of an AI agent to use external functions, APIs, or other Software components to extend its capabilities beyond its core reasoning and language generation. Tools allow agents to interact with the real world or specific digital environments.

**Why its crucial for an agent:**

**Expanded Capabilities:** Overcomes the limitations of an LLM’s static training data. Tools provide access to up-to-date information (web search), perform calculations, send emails, interact with databases, control devices or execute code.

**Grounding:** Connects the agent’s abstract reasoning to concrete actions and real-world data, preventing “hallucinations” by providing factual sources.

**Actionable Intelligence:** Transforms knowledge and reasoning into tangible outcomes in the environment.

**Modularity:** Allows for a flexible architecture where new tools can be easily integrated without retraining the core AI model.

**Examples:**

* Using a search engine tool (like Tavily in your provided code) to get real-time information.
* Calling a calculator tool to perform arithmetic
* Interacting with a calendar API to schedule appointments
* Executing Python Code through a code interpreter tool

1. **Reflexes (or Reactive Behaviour):**

The ability of an AI agent to respond immediately and automatically to specific, predefined environmental stimuli or conditions, often without extensive planning or deep reasoning. These are “if-then” rules that tigger instant actions.

**Why it’s crucial for an agent**

**Speed and Efficiency:** For simple, frequently occurring situations, a quick reflexive response is more efficient than going through full planning cycle.

**Safety and Immediate Action:** In critical systems (like autonomous vehicles or industrial controls), immediate reactions to hazard or specific triggers are paramount.

**Foundational Layer:** Even complex agents often have underlying reflexive layers for basic behaviors.

**Handling Known Scenarios:** For Situations that have clear, unambiguous optimal responses, a reflex can be hardcoded or learned.

**Examples:**

* A self-driving care immediately braking if a sudden obstacle appears (a high-priority reflex)
* A thermostat turning on the heating when the temperature drops below a set point.
* A Chatbot immediately acknowledging “hello” with a greeting.
* An agent immediately re-trying a failed API call if the error code indicates a transient issues.

These four components don’t operate in isolation; they work together in an intelligent agent. A truly sophisticated AI agent integrates all these elements to achieve robust, autonomous, and intelligent behaviour in dynamic environments.

## Frameworks: LangChain Agents, CrewAI, AutoGPT, OpenAgents

### 1. LangChain Agents

**Description:** LangChain is not strictly an agent framework itself, but rather a powerful *orchestration framework* for building applications powered by Large Language Models (LLMs). Within LangChain, the concept of "Agents" is a specific module designed to enable LLMs to reason about which tools to use and in what sequence, given a user's input and a set of available tools. It essentially provides the "brain" for an agent to decide its next action.

**How it fits Agentic AI:** LangChain agents are fundamental building blocks for creating Agentic AI systems. They provide the core **planning** and **tool-use** capabilities. An agent in LangChain is typically an LLM that, through prompt engineering, is given:

* A goal or question.
* A list of available tools with their descriptions.
* Access to its own internal **memory** (often via a ChatMessageHistory or similar mechanism).

The LLM then "thinks" (via chain-of-thought prompting) about the problem, decides if it needs a tool, which tool, what arguments to pass, and then executes that tool. This process can be iterative until the goal is achieved.

**Key characteristics in the context of Agentic AI:**

* **Flexible Planning:** Supports various agent types (e.g., zero-shot-react-description, OpenAIFunctionsAgent, AgentExecutor) that implement different reasoning strategies for tool selection.
* **Extensive Tool Integration:** LangChain's rich ecosystem of integrations means agents can easily access a vast array of tools (web search, calculators, APIs, databases, etc.), significantly enhancing their capabilities.
* **Memory Management:** Provides robust ways to integrate memory (e.g., ConversationBufferMemory, VectorStoreRetrieverMemory) for context-aware interactions.
* **Composability (LCEL):** Agents are often built using LangChain Expression Language (LCEL), allowing for highly flexible and customizable agentic workflows.
* **Developer-Centric:** Offers programmatic control, making it a preferred choice for developers to build custom agentic solutions.

**Limitations (in comparison to higher-level frameworks):** While powerful, LangChain agents often require developers to manually orchestrate the overall agent loop, error handling, and more complex multi-agent interactions. They provide the "how to act" but not always the "what to do next" across a long-running, complex autonomous process without additional code.

### 2. CrewAI

**Description:** CrewAI is a newer framework specifically designed for orchestrating **multi-agent systems**. It focuses on creating "crews" of autonomous agents that collaborate to achieve a shared goal. Each agent in a CrewAI "crew" has a defined role, a set of goals, and specific tools it can use. The framework handles the communication and task delegation between these agents, allowing for complex workflows to be broken down and executed collaboratively.

**How it fits Agentic AI:** CrewAI directly addresses the orchestration of multiple specialized AI agents, moving beyond a single agent's capabilities. It emphasizes:

* **Collaborative Planning:** Agents within a crew collectively work on a problem, leveraging their specialized roles.
* **Role-Based Autonomy:** Each agent is designed with a specific persona, tools, and responsibilities, enabling efficient division of labor.
* **Automated Workflow:** The framework manages the flow of information and tasks between agents, reducing the need for explicit step-by-step programming.

**Key characteristics in the context of Agentic AI:**

* **Multi-Agent Orchestration:** Its primary strength is facilitating complex interactions between multiple agents, each with distinct responsibilities.
* **Role-Playing & Specialization:** Encourages defining agents with specific role, goal, and backstory, which helps in guiding their behavior and problem-solving approach.
* **Task Management:** Provides mechanisms to define tasks and assign them to specific agents or groups of agents.
* **Process Automation:** Automates the communication and delegation loops between agents, allowing for sophisticated, automated workflows.
* **Simplified Collaboration:** Abstracts away much of the complexity of inter-agent communication and state sharing.
* **Built on LangChain:** Leverages LangChain's underlying agent capabilities, tools, and LLM integrations.

**Limitations:** Being built on LangChain, it inherits some of its underlying complexities. While it simplifies multi-agent orchestration, designing the optimal roles and tasks for a complex problem still requires careful thought.

### 3. AutoGPT

**Description:** AutoGPT was one of the early viral examples of an "autonomous agent" that captured the public imagination. It's a system designed to achieve a user-defined goal by autonomously generating and executing a sequence of thoughts, actions, and self-correction steps. It iteratively plans, acts, reviews its progress, and corrects itself until the goal is met.

**How it fits Agentic AI:** AutoGPT epitomizes the core tenets of Agentic AI:

* **Extreme Autonomy:** Once given a goal, it attempts to operate with minimal human intervention.
* **Iterative Planning & Execution:** It maintains an internal loop of "plan-act-reflect" until the objective is achieved.
* **Self-Correction:** It can identify when its actions fail or lead to undesired results and adjust its plan accordingly.
* **Dynamic Tool Use:** It can decide which tools (like web search, file writing, code execution) are needed at each step.

**Key characteristics in the context of Agentic AI:**

* **Loop-based Architecture:** Operates on a continuous loop of thinking, executing, and self-reflecting.
* **Goal-Driven:** Highly focused on achieving a high-level, often abstract, goal provided by the user.
* **"Thinking" Process:** Its prompts encourage the LLM to articulate its thoughts, reasoning, and plans, making its decision-making process more transparent (and debuggable).
* **Broad Capabilities:** By combining web access, file system interaction, and code execution, it can perform a wide range of digital tasks.

**Limitations:**

* **Cost and Speed:** The iterative nature can be slow and expensive due to numerous LLM calls and tool invocations.
* **Reliability/Hallucinations:** Can sometimes get stuck in loops, pursue irrelevant paths, or "hallucinate" non-existent information or successful outcomes, leading to unpredictable results.
* **Safety Concerns:** Its ability to write and execute code, and interact with the internet, raises significant safety and security concerns if not properly contained.
* **Complexity for Simple Tasks:** Can be overkill for straightforward tasks where a linear chain would suffice.

### 4. OpenAgents

**Description:** OpenAgents is a more recent and comprehensive **open-source platform/framework** for building, deploying, and managing LLM-powered agents. It aims to provide a unified environment for researchers and developers to experiment with and build robust agentic systems. It often focuses on benchmarking and enabling reproducibility of agentic research.

**How it fits Agentic AI:** OpenAgents provides the infrastructure to not just *create* agents, but to *manage* and *evaluate* them. It's less about a specific agent architecture (like AutoGPT's loop) and more about providing a framework that supports various agent designs, including:

* **Agent Deployment & Management:** Tools for deploying agents and managing their lifecycle.
* **Benchmarking & Evaluation:** Emphasizes methodologies for testing agent performance on various tasks.
* **Tool Integration:** Like LangChain, it facilitates connecting agents to diverse external tools.
* **Modular Design:** Supports building agents from various components and allows for experimentation with different planning, memory, and tool-use strategies.

**Key characteristics in the context of Agentic AI:**

* **Platform-Oriented:** More of an ecosystem or platform for agent development and deployment, rather than just a library for building a single agent.
* **Research & Development Focus:** Often includes features useful for researchers, such as evaluation metrics, benchmarking tools, and support for different agent architectures.
* **Reproducibility:** Aims to make it easier to reproduce agent experiments and build upon existing research.
* **Extensibility:** Designed to be highly modular and extensible, allowing developers to plug in their own LLMs, tools, memory systems, and planning algorithms.
* **Potential for Multi-Agent Systems:** While not its exclusive focus like CrewAI, the platform nature of OpenAgents naturally supports building and managing multiple interacting agents.

**Limitations:** As a broader platform, it might have a steeper learning curve for beginners compared to a simpler library. Its focus on research and generalized solutions might mean that for very specific, simple use cases, it could be considered more heavyweight than necessary.

In summary:

* **LangChain Agents** provide the fundamental mechanisms for an LLM to choose and use tools (the "brain" of an agent).
* **CrewAI** specializes in orchestrating *multiple* LangChain-powered agents to collaborate on complex tasks, focusing on roles and teamwork.
* **AutoGPT** pioneered the concept of a fully *autonomous, iterative* agent that self-corrects towards a goal.
* **OpenAgents** aims to be a comprehensive *platform* for building, deploying, and evaluating various types of LLM-powered agents, supporting research and robust development.

They represent different layers and approaches to building increasingly autonomous and intelligent AI systems.