GenAI AWS Architecture

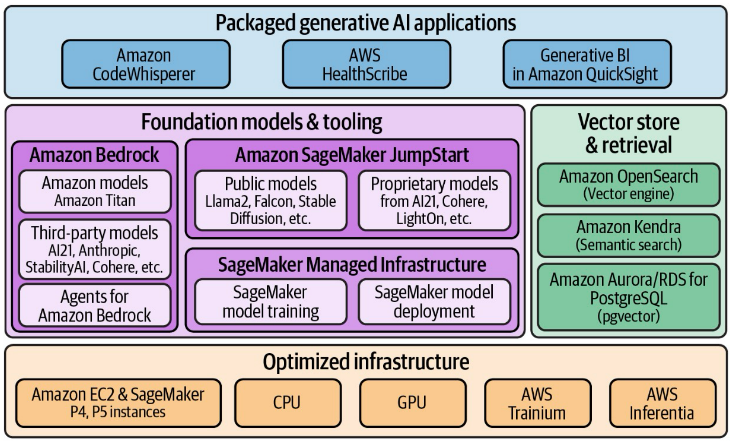

Packaged generative AI applications
Amazon CodeWhisperer
AWS HealthScribe
Generative BI in Amazon QuickSight
Foundation models & tooling
Amazon Bedrock
Amazon models Amazon Titan
Third-party models AI21, Anthropic, StabilityAI, Cohere, etc.
Agents for Amazon Bedrock
Amazon SageMaker JumpStart
Public models Llama2, Falcon, Stable Diffusion, etc.
Proprietary models from AI21, Cohere, LightOn, etc.
SageMaker Managed Infrastructure
SageMaker model training
SageMaker model deployment
Vector store & retrieval
Amazon OpenSearch (Vector engine)
Amazon Kendra (Semantic search)
Amazon Aurora/RDS for PostgreSQL (pgvector)
Optimized infrastructure
Amazon EC2 & SageMaker P4, P5 instances
CPU
GPU
AWS Trainium
AWS Inferentia

Figure 1-4. AWS services and features supporting generative AI

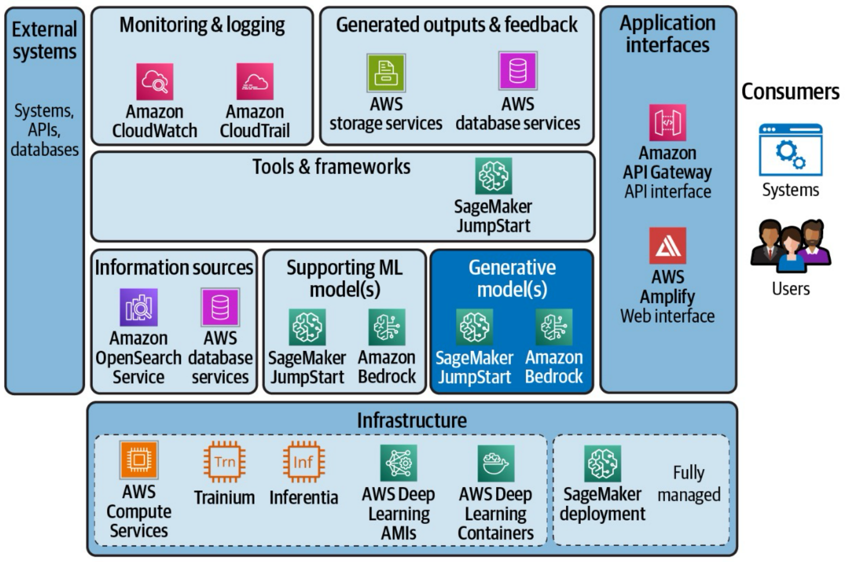


Figure 9-16. Examples of AWS services that can be used to build generative AI applications

GenAIOps, FMOps, or LLMOps

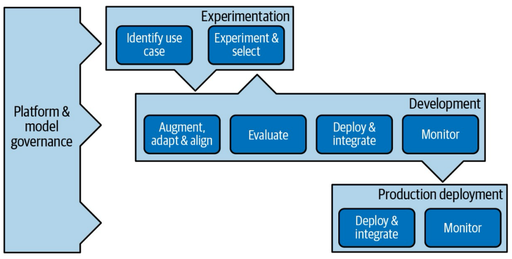


Figure 9-17. Create reliability and repeatability across stages of the generative ai project life cycle

Experiments

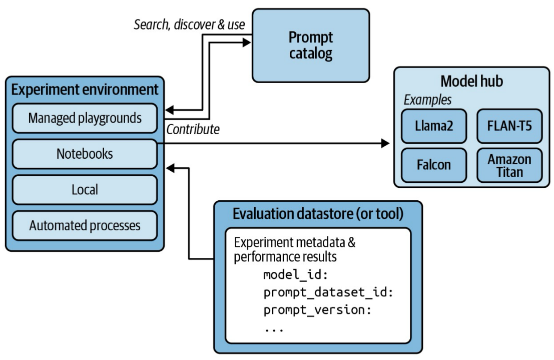


Figure 9-18. Building a repeatable mechanism to evaluate models during model selection

TRASFORMERS

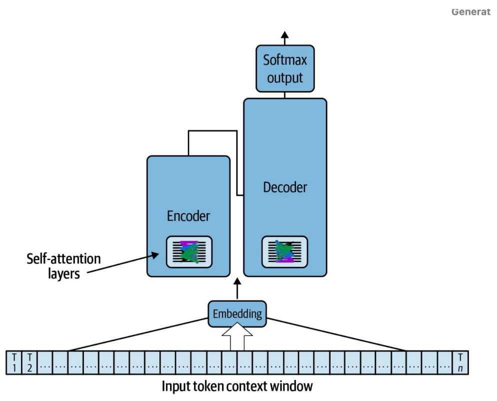


Figure 3-4. High-level Transformer architecture

QUERY, KEY,VALUE, SOFTMAX

ENCODER-ONLY

- MASKED LANGUAGE MODELING MLM

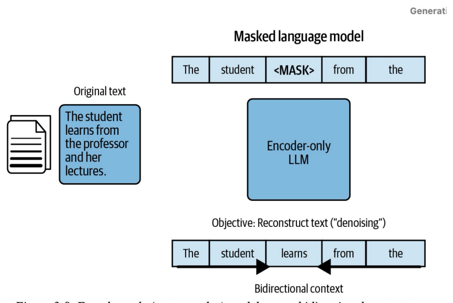


Figure 3-9. Encoder-only (autoencoder) models use a bidirectional context to reconstruct the masked input tokens

Encoder-only models are best suited text classification. They are not particularly useful for generative tasks that continue to generate more text. A well-known encoder-only model is BERT. USATI ANCHE PER RAG (SIMILARITY SEARCH)

1. **Sentiment Analysis**: Analyzing text to determine the sentiment expressed (positive, negative, neutral).
    - Model Example: BERT
    - AWS Service: Amazon Comprehend
2. **Text Classification**: Categorizing text into predefined categories, such as tagging customer feedback into topics.

    - Model Example: DistilBERT
    - AWS Service: Amazon Comprehend
3. **Named Entity Recognition (NER)**: Identifying and classifying key information (names, places, dates) in text.
    - Model Example: BERT
    - AWS Service: Amazon Comprehend
4. **Document Summarization**: Summarizing long documents into concise summaries.
    - Model Example: Longformer
    - AWS Service: There's no direct AWS service for this specific task as of my last update, but you can deploy custom models on Amazon SageMaker.
5. **Question Answering**: Extracting answers from a text given a question.
    - Model Example: BERT
    - AWS Service: Amazon Kendra for question-answering capabilities, although Kendra is more of a search service, it can be complemented with custom BERT models deployed on Amazon SageMaker for specific QA tasks.
6. **Language Modeling**: Predicting the next word or character in a sequence.
    - Model Example: GPT (Note: GPT is an encoder-decoder model but can be used in an encoder-only mode for specific tasks).
    - AWS Service: Custom models can be deployed on Amazon SageMaker.
7. **Feature Extraction**: Generating dense vector representations of text for use in various machine learning models.
    - Model Example: RoBERTa
    - AWS Service: Amazon SageMaker to deploy custom models for extracting features.
8. **Text Similarity and Clustering**: Determining how similar two pieces of text are and clustering similar texts together.
    - Model Example: Sentence Transformers
    - AWS Service: Amazon SageMaker for deploying custom models.
9. **Fake News Detection**: Identifying and flagging fake news articles.
    - Model Example: BERT
    - AWS Service: Amazon Comprehend for sentiment and entity recognition as part of a larger fake news detection pipeline, with custom logic on Amazon SageMaker.
10. **SEO Keyword Extraction**: Extracting relevant keywords from content for SEO optimization.
    - Model Example: BERT
    - AWS Service: Amazon Comprehend to extract key phrases which can then be refined for SEO purposes.

DECODER-ONLY AUTOREGRESSIVI (FLACON, GPT3, LLAMA2)

TRAINED USING CAUSAL LANGUAGE MODELING CLM, PREDICT NEXT TOKEN.

Aka Autoregressive language modeling: predicting the next word in a sequence given the previous words. The "causal" aspect refers to the fact that the model generates text based on the causal (or sequential) order of the words, where each word prediction is dependent only on the preceding words, not on any future words. This approach is foundational for many language generation models, including autoregressive transformers like GPT (Generative Pre-trained Transformer). **generate sequences of data by predicting one element at a time, using the history of previously generated elements as context**.

Tasks: generation or prediction where the output is sequential in nature.

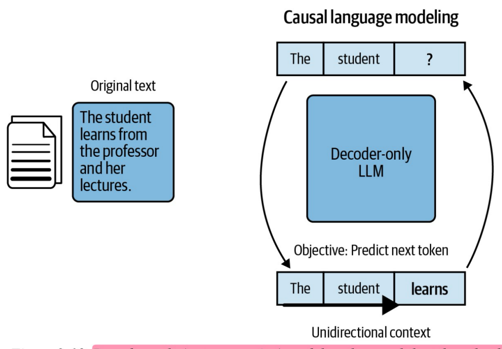


Figure 3-10. Decoder-only (autoregressive) models only reveal the tokens leading up to the token being predicted

1. **Text Generation**: Generating coherent and contextually relevant text based on a prompt.
   - Model Example: GPT-3 (Generative Pre-trained Transformer 3)
   - AWS Service: Amazon SageMaker for deploying custom GPT-3 models.
2. **Machine Translation**: Translating text from one language to another.
   - Model Example: Transformer Base or Transformer Big models (original Transformer model variants)
   - AWS Service: Amazon Translate for ready-to-use translation, or Amazon SageMaker for custom transformer models.
3. **Language Modeling**: Predicting the next word in a sentence given the previous words.
   - Model Example: GPT-2
   - AWS Service: Amazon SageMaker
4. **Code Generation**: Generating programming code based on a description of functionality.
   - Model Example: Codex (a GPT-3 variant fine-tuned for understanding and generating code)
   - AWS Service: Amazon SageMaker for deploying custom Codex models.
5. **Conversational AI and Chatbots**: Creating chatbots that can engage in human-like conversation.
   - Model Example: GPT-3
   - AWS Service: Amazon Lex for chatbots + sagemaker
6. **Music Composition**: Generating new pieces of music in a sequential manner, note by note or beat by beat.
   - Model Example: Transformer-based models tailored for music generation
   - AWS Service: Amazon SageMaker for deploying custom models.
7. **Predictive Text Completion**: Completing a user's sentence in real-time as they type, to speed up writing.
   - Model Example: GPT-3
8. **Speech Recognition**: Transcribing spoken language into text by predicting sequences of words.
   - Model Example: Wav2Vec 2.0 (While primarily a feature extractor, it can be used in an autoregressive setup for speech recognition)
   - AWS Service: Amazon Transcribe for direct speech recognition service, or Amazon SageMaker for custom models.
9. **Image Captioning**: Generating descriptive captions for images by sequentially predicting words.
   - Model Example: Transformer models that combine CNN features with autoregressive decoding.
10. **Handwriting Generation**: Producing text that mimics handwriting by generating sequences of strokes or characters.
    - Model Example: Transformer-based models designed for sequence-to-sequence tasks

ENCODE-DECODER SEQ2SEQ

Encoder-decoder architectures in transformers are designed for tasks that involve transforming an input sequence into an output sequence, where the two sequences can be of different lengths and structures. This architecture is particularly useful for tasks that require an understanding of the entire input before generating the output.
Designed for **translation**, are also very useful for **text-summarization**.

1. **Machine Translation**: Translating a text from one language to another while maintaining the context and nuances of the original language.
    - Model Example: Transformer (original model by Vaswani et al.)
    - AWS Service: Amazon Translate for direct translation services +SageMaker
2. **Text Summarization**: Creating a concise summary of a longer text that captures the main points.
    - Model Example: BART (Bidirectional and Auto-Regressive Transformers)
3. **Question Answering**: Providing answers to questions based on a given context paragraph or document.
    - Model Example: T5 (Text-to-Text Transfer Transformer)
    - AWS Service: Amazon SageMaker for deploying custom T5 models for sophisticated question-answering systems.
4. **Text-to-Speech (TTS)**: Converting written text into spoken words, generating human-like speech.
    - Model Example: Tacotron 2 (Although not a transformer, it's an encoder-decoder model used for TTS)
    - AWS Service: Amazon Polly for text-to-speech services, or Amazon SageMaker for deploying custom models.
5. **Speech-to-Text**: Transcribing spoken words into written text, accurately capturing the spoken content.
    - Model Example: DeepSpeech (While not a transformer, it's an example of an encoder-decoder model used for STT)
    - AWS Service: Amazon Transcribe for speech recognition, or Amazon SageMaker for deploying custom models.
6. **Image Captioning**: Generating descriptive text for an image.
    - Model Example: Show and Tell (a neural image caption generator)
    - AWS Service: Amazon SageMaker for deploying custom models that combine CNNs for image processing and transformers for text generation.
7. **Name Entity Recognition (NER)**: Identifying and classifying named entities in text into predefined categories such as the names of persons, organizations, locations.
    - Model Example: BERT (*Bidirectional Encoder Representations from Transformers*) for encoding, with a decoding layer for classification: trained with MLM and next sentence prediction (NSP).
    - AWS Service: Amazon Comprehend for ready-to-use NER, SageMaker
8. **Document Translation**: Translating entire documents while preserving formatting and structure.
    - Model Example: Transformer models specifically fine-tuned for document-level translation
    - AWS Service: Amazon Translate for straightforward document translation, or Amazon SageMaker for deploying and fine-tuning custom models.

9. **Dialogue Systems**: Building systems capable of conducting a conversation with human users, understanding their input, and generating appropriate responses.
    - Model Example: T5 or DialoGPT (a variant of GPT-2 optimized for dialogue)
    - AWS Service: Amazon Lex for creating conversational interfaces, integrated with custom models on Amazon SageMaker for more complex dialogue handling.
10. **Code Generation**: Automatically generating programming code from a natural language description.
    - Model Example: Codex (built on GPT-3)
    - AWS Service: Amazon SageMaker for deploying custom Codex models or other code generation models.

SCALING-LAWS: CHINCILLA PAPER

optimal training dataset size (in tokens) is 20x the number of model parameters and that anything below that 20x ratio is potentially overparameterized and undertrained.

Unveiling Transformer Learning for Trustworthy AI

Generative transformer models have become increasingly complex, with large numbers of parameters and the ability to process multiple input modalities. Current methods for explaining their predictions are resource-intensive. Most crucially, they require prohibitively large amounts of extra memory, since they rely on backpropagation that allocates almost twice as much GPU memory as the forward pass. This makes it difficult, if not impossible, to use them in production. We present AtMan (NeurIPS 2023), which provides explanations of generative transformer models (language and multimodal) at almost no extra cost. Specifically, AtMan is a modality-agnostic perturbation method that manipulates the attention mechanisms of transformers to produce relevance maps for the input with respect to the output prediction. Instead of using backpropagation, AtMan applies a parallelizable token-based search method based on cosine similarity neighborhood in the embedding space. Our exhaustive experiments on text and image-text benchmarks demonstrate that AtMan outperforms current state-of-the-art gradient-based methods on several metrics while being computationally efficient. As such, AtMan is suitable for use in large model inference deployments, making generative AI auditable and trustworthy and enabling a human expert to take responsibility, even in environments where there is no clear, easy answer.

4 MEMORY OPTIMIZATIONS IN FULL OR FINE-TUNING TRAININGS

E' O(N^2): optimizing the self-Attention Layers CON

1) QUANTIZATION, OR
2) FLASHATTENTION (SELF ATTENTION TRAINING DIVENTA O(N)) SU GPU
3) GROUPED QUERY ATTENTION: By grouping queries together before computing attention scores, this method reduces the computational complexity and memory

usage. It operates by aggregating similar or related queries into groups and then performing the attention operations on these groups instead of individual queries. This approach allows the model to focus computational resources on processing groups of queries that share common features or targets, enhancing the model's scalability and performance, especially in tasks involving large input sequences or datasets.

4) DISTRIBUTED COMPUTING
   a. DISTRIBUTED DATA PARALLEL

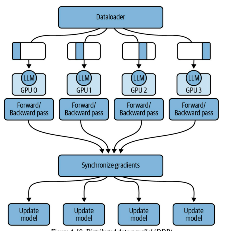


Figure 4-10. Distributed data parallel (DDP)

   b. FULLY SHARDED DATA PARALLEL (2019 ZERO PAPER): sharding the model with gradients, activations, and optimizer states—across the GPUs to achieve zero redundancy in the system.

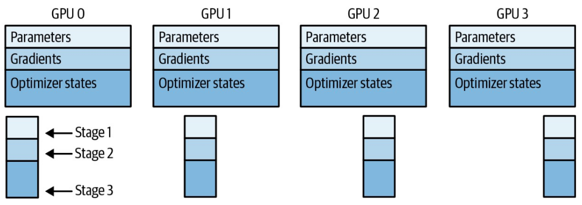
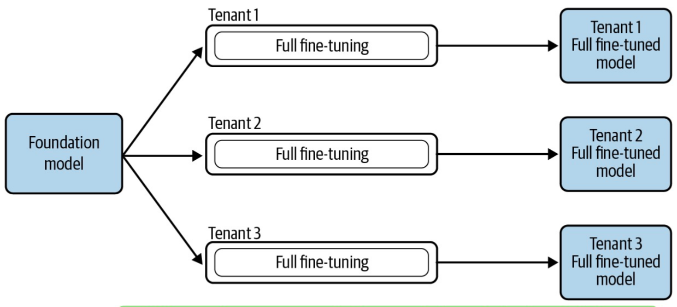


Figure 4-11. ZeRO consists of three stages depending on the GPU shards: parameters, gradients, and optimizer states

Train model on AWS Trainium hardware with AWS Neuron SDK OR Hugging Face Optimum Neuron library which integrates the Hugging Face Transformers ecosystem with the Neuron SDK.

FULL FINE-TUNING


Foundation model
Tenant 1
Full fine-tuning
Tenant 1 Full fine-tuned model
Tenant 2
Full fine-tuning
Tenant 2 Full fine-tuned model
Tenant 3
Full fine-tuning
Tenant 3 Full fine-tuned model
Figure 6-1. Full fine-tuning creates a full copy of the original model for each tenant

Instruction Fine-Tuning and Evaluation

- 500-1000 examples should be enough. If you provide instructions for just a single task (e.g., summarization) during fine-tuning, the model may experience "**catastrophic forgetting**" in which the model becomes so good at a single task that it may lose its ability to handle, or generalize to, other tasks.

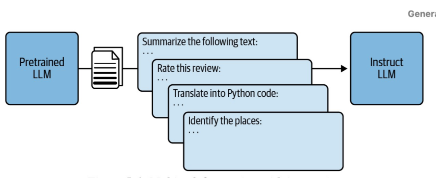


Figure 5-1. Multitask fine-tuning with instruction

- PROMPT TEMPLATES LIKE dialogue-summary. FLAN T5.TEMPLATE
- Amazon SageMaker JumpStart: scale your fine-tuning workload to a large, distributed cluster of GPU instances simply by changing a single parameter, instance_count

Mixture-of-Experts (MoE)

Mixture-of-Experts (MoE) layers enhance a language model's capacity without significantly increasing computational demands. By substituting standard layers with MoE layers, which consist of multiple specialized layers (experts) with unique parameters, the model gains flexibility and depth. A gating mechanism selectively activates these experts for specific inputs, enabling efficient, sparse computation. Originating from early research on conditional computation, MoE layers have evolved to facilitate the training of large-scale models by offering a scalable way to boost model complexity and performance, particularly beneficial in areas like language modeling where larger model capacity often translates to improved outcomes.

"As the training of giant dense models hits the boundary on the availability and capability of the hardware resources today, Mixture-of-Experts (MoE) models have become one of the most promising model architectures due to their significant training cost reduction compared to quality equivalent dense models." - Mixture-of-Experts (MoE) layers are simple and allow us to increase the size or capacity of a language model without a corresponding increase in compute. We just replace certain layers of the model with multiple copies of the layer—*called "experts"*—that have their own parameters. Then, we can use a gating mechanism to (sparsely) select the experts used to process each input. This idea has its roots in research on conditional computation in the early 1990s [15, 30] and allows us to train massive models in a tractable manner, which is helpful in domains—*such as language modeling*—that benefit from models with extra capacity. Here, we will study the MoE, its origins, and how it has evolved over the last two decades.

Model forgetting

https://www.buonaiuto.work/enhancing-multilingual-models-with-active-forgetting/

EVALUATION OF MODELS

- ROUGE (N-GRAM) FOR SUMMARIZATION TASKS,
- BLUE FOR TRANSLATIONS
- GLUE, SUPERGLUE
- HELM
- BIG-BENCH

4 PEFT PARAMETER-EFFICIENT FINE TUNING: lora, qlora, soft prompt, RLHF

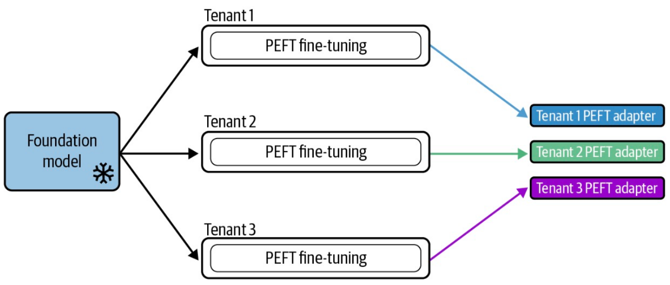
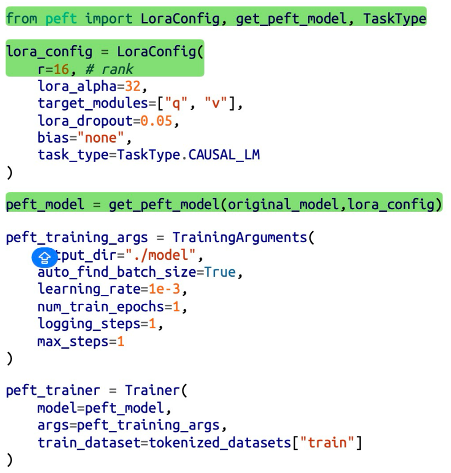


Figure 6-2. PEFT reduces task-specific model weights and can merge with original LLM at inference



```
from peft import LoraConfig, get_peft_model, TaskType

lora_config = LoraConfig(
    r=16, # rank
    lora_alpha=32,
    target_modules=["q", "v"],
    lora_dropout=0.05,
    bias="none",
    task_type=TaskType.CAUSAL_LM
)

peft_model = get_peft_model(original_model,lora_config)

peft_training_args = TrainingArguments(
    [illegible]tput_dir="./model",
    auto_find_batch_size=True,
    learning_rate=1e-3,
    num_train_epochs=1,
    logging_steps=1,
    max_steps=1
)

peft_trainer = Trainer(
    model=peft_model,
    args=peft_training_args,
    train_dataset=tokenized_datasets["train"]
)
```

- LORA ("Low-Rank Adaptation"):the number of parameters to be trained by freezing all of the original model parameters and inserting a pair of rank decomposition matrices alongside the original weights of a targeted set of modules (e.g., layers) in the model—typically the linear layers, including self-attention. keep the original weights of the model frozen and train these smaller matrices using the same supervised learning process. The size of the low-rank matrices is set by the parameter called rank (r).

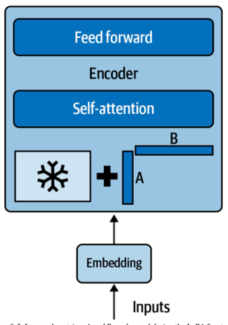


Figure 6-3. Low-rank matrices A and B are learned during the LoRA fine-tuning

- PROMPT TUNING AND SOFT PROMPTS

- RLHF FOR Helpful-Honest-Harmless

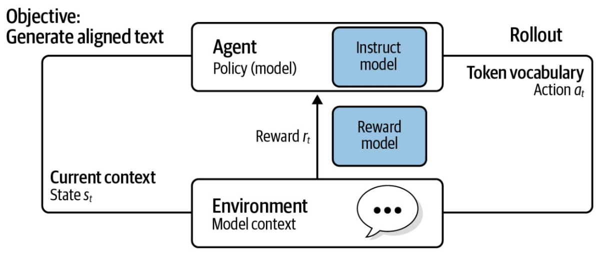


Figure 7-2. Reinforcement learning in the context of a generative AI model

  - Amazon SageMaker Ground Truth for Human Annotations
  - PPO Proximal Policy Optimization RL Algorithm to update model with new weights

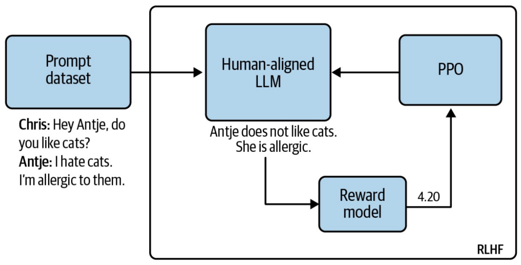

Prompt dataset
Chris: Hey Antje, do you like cats?
Antje: I hate cats. I'm allergic to them.
Human-aligned LLM
Antje does not like cats. She is allergic.
PPO
Reward model
4.20
RLHF

Figure 7-10. Proximal Policy Optimization RL algorithm with Meta's hate speech model

  - Parameter-Efficient Fine-Tuning LORA with RLHF

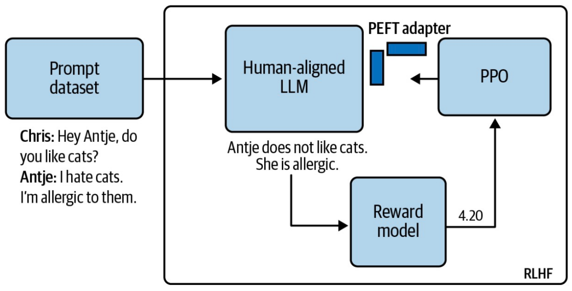

PEFT adapter
Prompt dataset
Chris: Hey Antje, do you like cats?
Antje: I hate cats. I'm allergic to them.
Human-aligned LLM
Antje does not like cats. She is allergic.
PPO
Reward model
4.20
RLHF

Figure 7-12. Using PEFT within RLHF to minimize the resources needed to fine-tune the generative model

3 Model Deployment Optimizations: prun,quantiz, distil

The size of generative AI models often presents a challenge for deployment in terms of compute, storage, and memory requirements, as well as how to ensure low-latency completions. One of the primary ways to optimize for deployment is to take advantage of techniques that aim to reduce the size of the model, typically referred to as model compression.

1) PRUNING (SPARSEGPT)
2) QUANTIZATION Post-Training Quantization with GPTQ Hugging Face Optimum library
3) DISTILLATION reduces computation and improve perf TEACHER-STUDENT. The teacher model's output is used to "distill" knowledge to the student model. The teacher models' predicted tokens are known as **soft labels**, while the student models' predicted tokens are called **soft predictions**. In parallel, you need to compare the student models' predictions (hard predictions) against the ground truth hard labels from the prompt dataset. The difference is the **student loss**. **The distillation loss and student loss are combined and used to update the student models' weights using**

standard backpropagation.

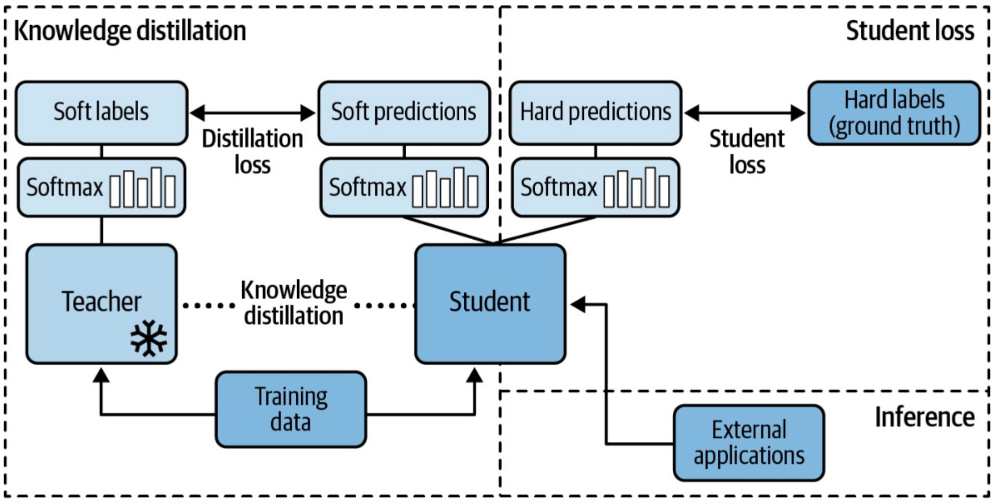


Figure 8-4. Distill knowledge from teacher to student model

a. Hugging Face Optimum library for distillation

Large Model Inference Container: INFERENTIA, NEURON

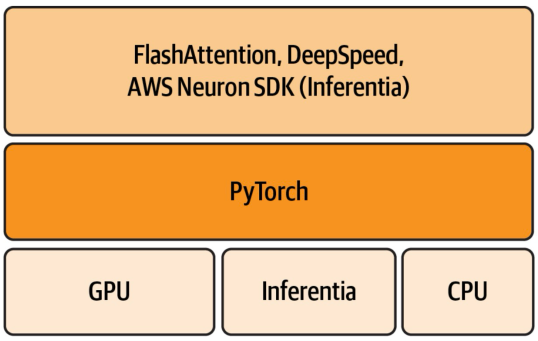


Figure 8-5. LMI container and hardware for hosting LLMs with Amazon SageMaker Endpoints

- AWS Inferentia: Purpose-Built Hardware for Inference; family of accelerators, , is purpose-built for deep learning inference workloads. The AWS Neuron SDK interacts with AWS Inferentia.
- AWS Neuron is a software development kit (SDK) enabling high-performance deep learning acceleration using AWS Inferentia and Trainium

2 *Model Update and Deployment* Strategies (AB testing,shadow deployment)

- A/B Testing
- Shadow deployment

- Amazon CloudWatch

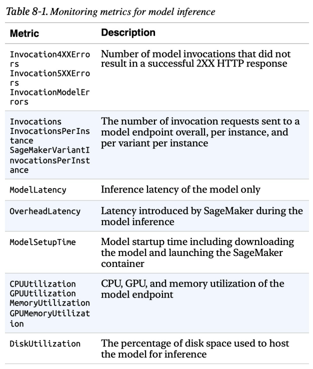


Table 8-1. Monitoring metrics for model inference

Metric	Description
`Invocation4XXErrors` `Invocation5XXErrors` `InvocationModelErrors`	Number of model invocations that did not result in a successful 2XX HTTP response
`Invocations` `InvocationsPerInstance` `SageMakerVariantInvocationsPerInstance`	The number of invocation requests sent to a model endpoint overall, per instance, and per variant per instance
`ModelLatency`	Inference latency of the model only
`OverheadLatency`	Latency introduced by SageMaker during the model inference
`ModelSetupTime`	Model startup time including downloading the model and launching the SageMaker container
`CPUUtilization` `GPUUtilization` `MemoryUtilization` `GPUMemoryUtilization`	CPU, GPU, and memory utilization of the model endpoint
`DiskUtilization`	The percentage of disk space used to host the model for inference

Context-Aware Reasoning Applications Using RAG and Agents: LANGCHAIN, REACT, PAL

RAG= retrieval-augmented generation

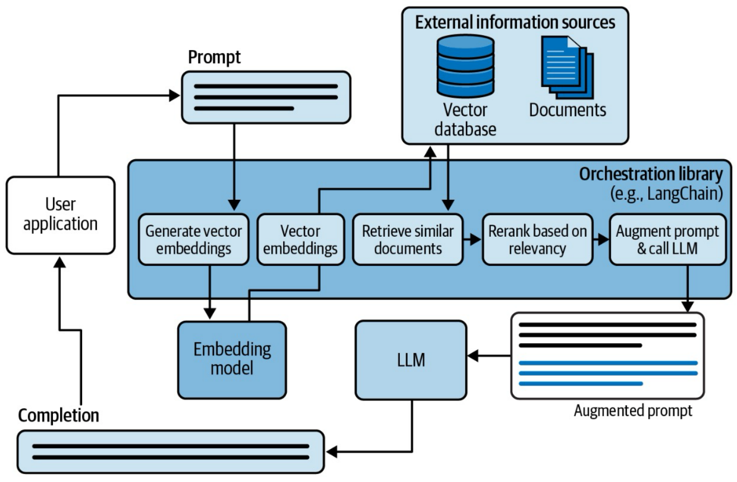

External information sources
Vector database
Documents
Prompt
User application
Orchestration library (e.g., LangChain)
Generate vector embeddings
Vector embeddings
Retrieve similar documents
Rerank based on relevancy
Augment prompt & call LLM
Embedding model
LLM
Augmented prompt
Completion

Figure 9-8. Orchestrating RAG workflows

Agents orchestrate prompt-completion workflows between user requests, foundation models, and external data sources and applications while using the foundational model as brain using ReACT chain of thoughts (COT).

Vector embeddings - `from langchain.vectorstores import`

LangChain integrates with many vector stores, such as **ElasticSearch, OpenSearch, Pinecone, and Facebook AI Similarity Search (FAISS)**

Feature/Aspect	**ElasticSearch**	**OpenSearch**	**Pinecone**	**FAISS (Facebook AI Similarity Search)**
Architecture	Distributed search engine based on Lucene	Fork of Elasticsearch, also based on Lucene	Managed vector database designed for similarity search	Library for efficient similarity search of dense vectors
Primary Use Cases	Full-text search, structured search, analytics	Full-text search, structured search, analytics	Similarity search in high-dimensional spaces	Efficient similarity search and clustering of dense vectors
Scalability	Highly scalable, distributed nature	Highly scalable, distributed nature	Built for scalability and performance at scale	Highly efficient on large datasets, but scalability depends on the hardware and integration

Feature/Aspect	ElasticSearch	OpenSearch	Pinecone	FAISS (Facebook AI Similarity Search)
Managed Service	Available via Elastic Cloud and other cloud providers	Available via AWS and other cloud providers	Fully managed service	Not a managed service; requires self-hosting and integration
Vector Search Support	Supports vector search through dense vector fields and plugins like Elasticsearch Learning to Rank	Similar support as Elasticsearch, including plugins	Native and primary focus on efficient vector search	Specialized in vector similarity search, requires integration with other systems for full search capabilities
Machine Learning Integration	Integrates with Elastic ML for anomaly detection and forecasting	Integrates with Elastic ML for anomaly detection and forecasting	Focuses on vector search, but can be used alongside ML models for enriched applications	Primarily for ML applications, especially those requiring similarity search, such as recommendation systems
Open Source	Yes, with commercial features available	Fully open source	Proprietary, with a free tier available	Open source
Optimized for	General search and analytics	General search and analytics	Similarity search in vector spaces	Dense vector similarity search and clustering
Customizability	High, through various plugins and configurations	High, similar to Elasticsearch	Configurable indices and schemas	High, but focused on algorithmic customization for search and clustering
Ease of Use	User-friendly with extensive documentation and community support	Similar to Elasticsearch	Designed for simplicity in similarity searches	Requires more specialized knowledge to implement and integrate

Feature/Aspect	ElasticSearch & OpenSearch	Pinecone	FAISS
Performance: Read/Search	High, with optimizations for distributed search	Very high, optimized for vector similarity search	Extremely high for similarity search; optimized for GPU/CPU
Performance: Write/Index	High, efficient indexing with support for bulk operations	High, supports efficient batch indexing	Variable, depends on setup; batch processing can be efficient but requires more manual management
Ease of Use: Read/Search	Relatively easy with DSL and REST APIs; extensive client libraries	Very easy for vector searches; simplified API	More complex; requires manual setup of search parameters and index loading
Ease of Use: Write/Index	Easy with REST APIs and client libraries; bulk indexing supported	Easy with simplified API for batch indexing	More complex; requires understanding of vector space partitioning and indexing techniques
LOC for 100 Reads	Low to moderate; bulk search operations can reduce LOC	Low; streamlined API for search queries	Moderate to high; complexity depends on the integration level
LOC for 100 Writes	Low to moderate; bulk indexing can significantly reduce LOC	Low; simplified API for batch updates	High; requires manual data preparation and batch processing setup

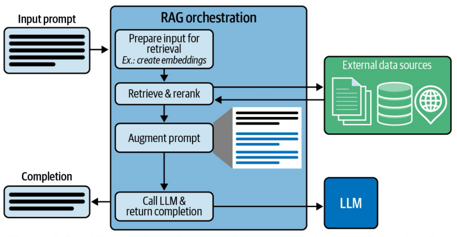


Figure 9-1. RAG provides a framework for augmenting a model with information from external sources

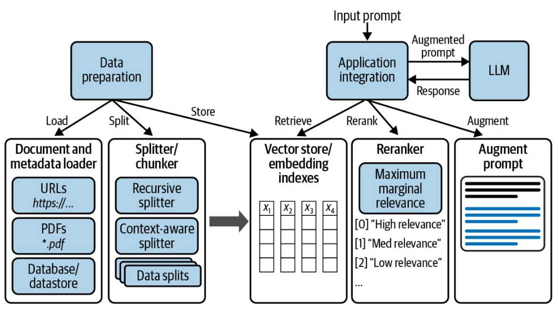


Figure 9-3. RAG architectures depend on efficient data preparation and retrieval techniques for integration into consuming applications

Creating vector embeddings that store numeric representations of text data in vector stores provides for efficient document search and retrieval techniques in RAG architectures. Documents are often large and contain varied degrees of related information on a variety of topics, some more related than others. you need to consider efficient strategies for optimizing the storage and retrieval of these documents as well as minimizing the risk of losing context. Because LLMs have fixed context window limitations, you also need to develop document storage and retrieval strategies that consider those limitations. "RAG will be used to augment the prompt with additional information prior to calling the LLM."

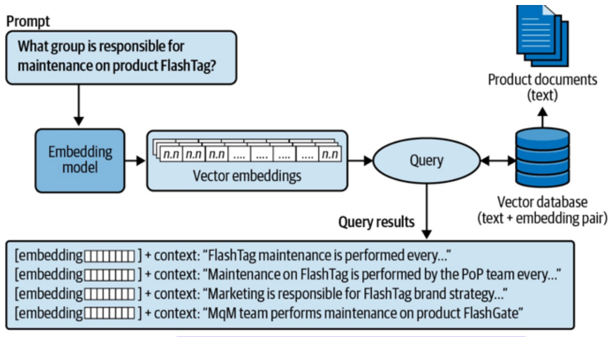

Prompt
What group is responsible for maintenance on product FlashTag?
Product documents (text)
Embedding model
Vector embeddings
Query
Vector database (text + embedding pair)
Query results
[embedding] + context: "FlashTag maintenance is performed every..."
[embedding] + context: "Maintenance on FlashTag is performed by the PoP team every..."
[embedding] + context: "Marketing is responsible for FlashTag brand strategy..."
[embedding] + context: "MqM team performs maintenance on product FlashGate"

Figure 9-6. Information retrieval based on prompt input

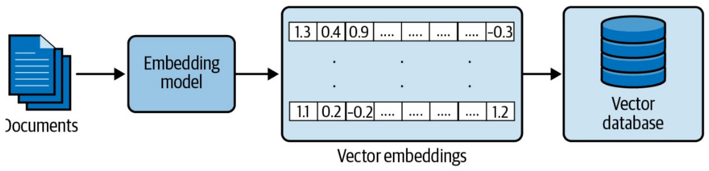


Figure 9-4. Efficient indexing of documents for quick retrieval

Chunking

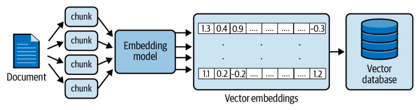


Figure 9-5. Chunking when storing and indexing documents

Reranking with "Maximum Marginal Relevance (MMR)"

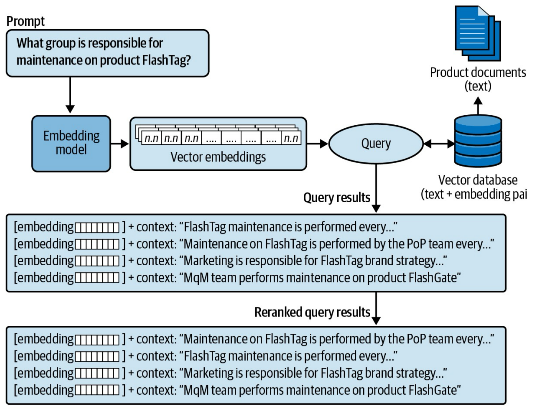

Prompt
What group is responsible for maintenance on product FlashTag?
Product documents (text)
Embedding model
Vector embeddings
Query
Vector database (text + embedding pai
Query results
[embedding] + context: "FlashTag maintenance is performed every..."
[embedding] + context: "Maintenance on FlashTag is performed by the PoP team every..."
[embedding] + context: "Marketing is responsible for FlashTag brand strategy..."
[embedding] + context: "MqM team performs maintenance on product FlashGate"
Reranked query results
[embedding] + context: "Maintenance on FlashTag is performed by the PoP team every..."
[embedding] + context: "FlashTag maintenance is performed every..."
[embedding] + context: "Marketing is responsible for FlashTag brand strategy..."
[embedding] + context: "MqM team performs maintenance on product FlashGate"

Figure 9-7. Reranking query results before augmenting the prompt

Prompt Augmentation

Augmented prompt

Completion prompt

LangChain - AWS Bedrock

LangChain is a framework designed to facilitate the creation, combination, and experimentation with different components in language models, especially in the context of building applications that leverage large language models (LLMs) for various tasks. One of the concepts within LangChain is the use of "chains," which are sequences of components linked together to perform complex tasks. These components can include language models, databases, retrieval systems, and more. The framework allows for the creation of sophisticated workflows by chaining together different functionalities.

Components of RAG:

1. **Retriever**: This component is responsible for querying a large dataset or document collection to find content that is relevant to the input question or prompt. The retrieval is usually based on semantic similarity, meaning the retriever looks for documents that semantically relate to the question, even if they don't contain the exact words.
2. **Generator**: The retrieved documents are then passed to a generative model, which synthesizes the information contained in them to generate a coherent and contextually relevant answer. This model can be based on architectures like Transformers, and it leverages the information from the retrieved documents to enhance its responses.

Chains in LangChain

Chains are essentially pipelines where the output of one component serves as the input for the next. This modular approach enables developers to build complex language processing applications by combining simpler, reusable components. For example, a chain might involve retrieving relevant documents, summarizing content, and then generating answers based on the summarized information.
https://github.com/svpino/youtube-rag/blob/main/rag.ipynb

RetrievalQA in LangChain

RetrievalQA, or Retrieval-based Question Answering, is a specific use case within LangChain where a chain is designed to answer questions by retrieving relevant information from a database or collection of documents before attempting to generate an answer. This approach mimics how humans often answer questions: by first finding relevant sources of information and then synthesizing answers based on what they've found.
How RetrievalQA Works in LangChain:

1. **Question Processing**: The chain begins with a question or prompt from the user. This input is processed to understand the context and intent.
2. **Document Retrieval**: The next step involves retrieving relevant documents or data that may contain the answer to the question. This is typically done using a retrieval system that can search through a large corpus of text based on keywords, semantic similarity, or other criteria relevant to the question.
3. **Document Processing**: The retrieved documents might be processed further, such as by summarizing them or extracting key pieces of information. This step reduces the amount of data that the next component in the chain has to handle and focuses on the most relevant information.
4. **Answer Generation**: Finally, based on the processed information from the retrieved documents, the chain generates an answer to the original question. This can involve a language model that synthesizes information from the documents into a coherent and concise answer.
5. **Feedback/Iteration**: Optionally, the system can incorporate feedback mechanisms to refine the answers or improve the retrieval process, enhancing accuracy over time.

ReACT (prompt+Questions Thought Action Observation)

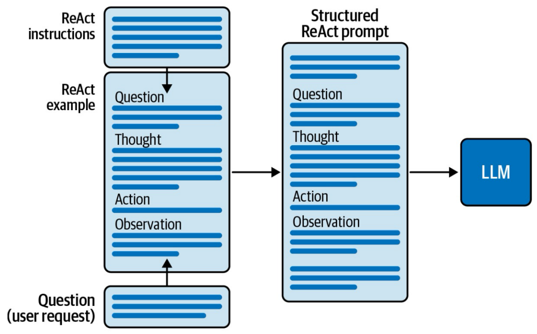
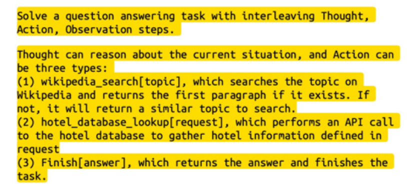


Figure 9-10. ReAct structures prompts to include instructions, ReAct examples, and the user request


```
Solve a question answering task with interleaving Thought,
Action, Observation steps.

Thought can reason about the current situation, and Action can
be three types:
(1) wikipedia_search[topic], which searches the topic on
Wikipedia and returns the first paragraph if it exists. If
not, it will return a similar topic to search.
(2) hotel_database_lookup[request], which performs an API call
to the hotel database to gather hotel information defined in
request
(3) Finish[answer], which returns the answer and finishes the
task.
```

Amazon Bedrock endpoints

https://docs.aws.amazon.com/bedrock/latest/userguide/api-setup.html

To connect programmatically to an AWS service, you use an endpoint. Refer to the Amazon Bedrock endpoints and quotas chapter in the AWS General Reference for information about the endpoints that you can use for Amazon Bedrock.

Amazon Bedrock provides the following service endpoints.

- `bedrock` – Contains control plane APIs for managing, training, and deploying models. For more information, see Amazon Bedrock Actions and Amazon Bedrock Data Types.
- `bedrock-runtime` – Contains runtime plane APIs for making inference requests for models hosted in Amazon Bedrock. For more information, see Amazon Bedrock Runtime Actions and Amazon Bedrock Runtime Data Types.
- `bedrock-agent` – Contains control plane APIs for creating and managing agents and knowledge bases. For more information, see Agents for Amazon Bedrock Actions and Agents for Amazon Bedrock Data Types.
- `bedrock-agent-runtime` – Contains control plane APIs for managing, training, and deploying models. For more information, see Agents for Amazon Bedrock Runtime Actions and Agents for Amazon Bedrock Runtime Data Types.

---

https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

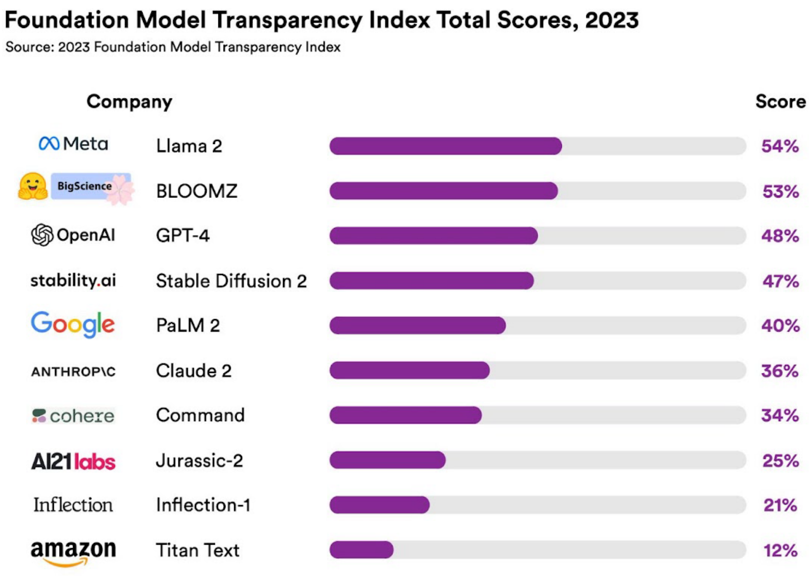
Boto3 documentation

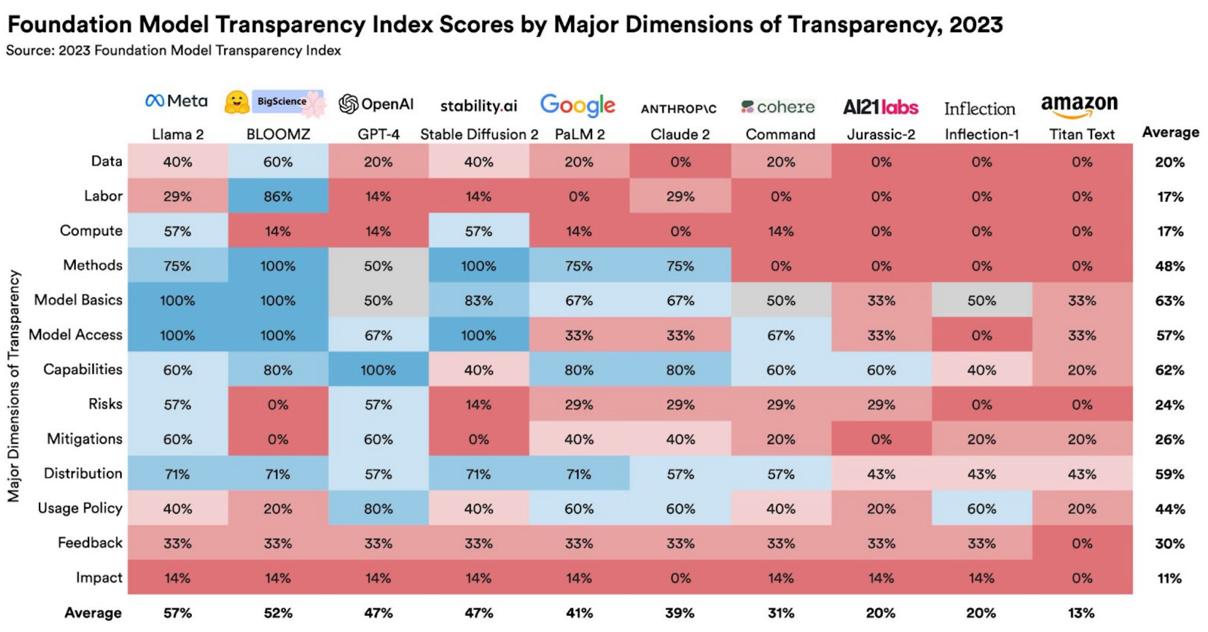
You use the AWS SDK for Python (Boto3) to create, configure, and manage AWS services, such as Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3). The SDK provides an object-oriented API as well as low-level access to AWS services.

Supported foundation models in Amazon Bedrock

Introducing The Foundation Model Transparency Index
100 indicators for transparency pdf

Less transparency makes it harder for other businesses to know if they can safely build applications that rely on commercial foundation models; for academics to rely on commercial foundation models for research; for policymakers to design meaningful policies to rein in this powerful technology; and for consumers to understand model limitations or seek redress for harms caused.




Foundation Model Transparency Index Scores by Major Dimensions of Transparency, 2023
Source: 2023 Foundation Model Transparency Index

Major Dimensions of Transparency	Meta Llama 2	BigScience BLOOMZ	OpenAI GPT-4	stability.ai Stable Diffusion 2	Google PaLM 2	ANTHROPIC Claude 2	cohere Command	AI21labs Jurassic-2	Inflection Inflection-1	amazon Titan Text	Average
Data	40%	60%	20%	40%	20%	0%	20%	0%	0%	0%	20%
Labor	29%	86%	14%	14%	0%	29%	0%	0%	0%	0%	17%
Compute	57%	14%	14%	57%	14%	0%	14%	0%	0%	0%	17%
Methods	75%	100%	50%	100%	75%	75%	0%	0%	0%	0%	48%
Model Basics	100%	100%	50%	83%	67%	67%	50%	33%	50%	33%	63%
Model Access	100%	100%	67%	100%	33%	33%	67%	33%	0%	33%	57%
Capabilities	60%	80%	100%	40%	80%	80%	60%	60%	40%	20%	62%
Risks	57%	0%	57%	14%	29%	29%	29%	29%	0%	0%	24%
Mitigations	60%	0%	60%	0%	40%	40%	20%	0%	20%	20%	26%
Distribution	71%	71%	57%	71%	71%	57%	57%	43%	43%	43%	59%
Usage Policy	40%	20%	80%	40%	60%	60%	40%	20%	60%	20%	44%
Feedback	33%	33%	33%	33%	33%	33%	33%	33%	33%	0%	30%
Impact	14%	14%	14%	14%	14%	0%	14%	14%	14%	0%	11%
Average	57%	52%	47%	47%	41%	39%	31%	20%	20%	13%	

Open LLM Leaderboard

📐 The 🤗 Open LLM Leaderboard aims to track, rank and evaluate open LLMs and chatbots.

https://github.com/EleutherAI/lm-evaluation-harness

- Over 60 standard academic benchmarks for LLMs, with hundreds of subtasks and variants implemented.

AutoGPTQ LLM Quantization

An easy-to-use LLM quantization package with user-friendly APIs, based on GPTQ algorithm (weight-only quantization).

State-of-the-art Machine Learning for JAX, PyTorch and TensorFlow

🤗 Transformers provides thousands of pretrained models to perform tasks on different modalities such as text, vision, and audio.

These models can be applied on:

- 📝 Text, for tasks like text classification, information extraction, question answering, summarization, translation, and text generation, in over 100 languages.
- 🖼️ Images, for tasks like image classification, object detection, and segmentation.
- 🗣️ Audio, for tasks like speech recognition and audio classification.

Transformer models can also perform tasks on **several modalities combined**, such as table question answering, optical character recognition, information extraction from scanned documents, video classification, and visual question answering.

🤗 Transformers provides APIs to quickly download and use those pretrained models on a given text, fine-tune them on your own datasets and then share them with the community on our model hub. At the same time, each python module defining an architecture is fully standalone and can be modified to enable quick research experiments.

🤗 Transformers is backed by the three most popular deep learning libraries — Jax, PyTorch and TensorFlow — with a seamless integration between them. It's straightforward to train your models with one before loading them for inference with the other.

https://huggingface.co/models

Current number of checkpoints: 531k

```
>>> from transformers import pipeline



# Allocate a pipeline for sentiment-analysis

>>> classifier = pipeline('sentiment-analysis')

>>> classifier('We are very happy to introduce pipeline to the transformers repository.')

[{'label': 'POSITIVE', 'score': 0.9996980428695679}]
```

---

https://www.pluralsight.com/resources/blog/data/get-started-amazon-bedrock

```
# To run this code, you first need to install the AWS SDK for Python called boto3
# From the terminal, type "pip install boto3"
import boto3
import json

# Create the client object for interacting with Amazon Bedrock
# Be sure to select a region where Amazon Bedrock is available
bedrock = boto3.client(
    service_name='bedrock-runtime',
    region_name='us-west-2'
)

# The input we'll send to the model
# TIP: You can get this info in the playgrounds by clicking "View API request" and then
updating the code below
input = {
  "modelId": "meta.llama2-13b-chat-v1",
  "contentType": "application/json",
  "accept": "*/*",
  "body": "{\"prompt\":\"I need an idea for an app to build on Amazon
Bedrock.\",\"max_gen_len\":512,\"temperature\":0.5,\"top_p\":0.9}"
}

# The response from the model
response = bedrock.invoke_model(body=input["body"],
                                modelId=input["modelId"],
                                accept=input["accept"],
                                contentType=input["contentType"])

response_body = json.loads(response['body'].read())

# Print the response from the model
print(response_body)
```

With fine-tuning, you take one of the base models (like Llama or Titan) with its general knowledge, then you augment it with your own data.

In Amazon Bedrock, you can get to this functionality by clicking on **Custom models** on the left-hand navigation, then clicking **Customize model→Create Fine-tune job**. Custom models can be very expensive, so I would not recommend going through with this unless you really know what you're doing and have the budget to support it.
https://docs.aws.amazon.com/bedrock/latest/userguide/what-is-bedrock.html

Knowledge base for Amazon Bedrock

Knowledge base for Amazon Bedrock provides you the capability of amassing data sources into a repository of information. With knowledge bases, you can easily build an application that takes advantage of *retrieval augmented generation (RAG)*, a technique in which the retrieval of information from data sources augments the generation of model responses. Once set up, you can take advantage of a knowledge base in the following ways.

- Configure your RAG application to use the RetrieveAndGenerate API to query your knowledge base and generate responses from the information it retrieves.
- Associate your knowledge base with an agent (for more information, see Agents for Amazon Bedrock) to add RAG capability to the agent by helping it reason through the steps it can take to help end users.
- Create a custom orchestration flow in your application by using the Retrieve API to retrieve information directly from the knowledge base.

Agents for Amazon Bedrock

Agents for Amazon Bedrock offers you the ability to build and configure autonomous agents in your application. An agent helps your end-users complete actions based on organization data and user input. Agents orchestrate interactions between foundation models (FMs), data sources, software applications, and user conversations. In addition, agents automatically call APIs to take actions and invoke knowledge bases to supplement information for these actions. Developers can save weeks of development effort by integrating agents to accelerate the delivery of generative artificial intelligence (generative AI) applications .

With agents, you can automate tasks for your customers and answer questions for them. For example, you can create an agent that helps customers process insurance claims or an agent that helps customers make travel reservations. You don't have to provision capacity, manage infrastructure, or write custom code. Amazon Bedrock manages prompt engineering, memory, monitoring, encryption, user permissions, and API invocation.

Agents for Amazon Bedrock enable the construction and configuration of autonomous agents within applications to assist end-users in completing actions leveraging organization data and user input. These agents orchestrate interactions among foundation models, data sources, software applications, and user dialogues, automating tasks such as processing insurance claims or making travel reservations without requiring manual capacity provisioning, infrastructure management, or custom code development. Amazon Bedrock handles aspects like prompt engineering, memory management, monitoring, encryption, user permissions, and API calls.
Key functionalities of agents include extending foundation models to parse user requests into actionable tasks, engaging in natural conversations to gather additional user information, executing API calls to fulfill requests, and enhancing performance through data source

queries. To deploy an agent, developers may optionally create a knowledge base, configure the agent for specific use cases, associate it with a knowledge base for improved performance, customize behavior through prompt templates, test the agent using the Amazon Bedrock console or API, and deploy it within their application by creating aliases for agent versions. This process significantly reduces development time for generative AI applications by automating a wide range of tasks.

BERT

BERT (Bidirectional Encoder Representations from Transformers) is a groundbreaking model in the field of natural language processing (NLP) developed by Google. It represents a significant departure from previous models due to its deep bidirectionality, allowing the model to understand the context of a word based on all of its surroundings (left and right of the word). BERT is pre-trained on a large corpus of text and then can be fine-tuned with additional output layers to perform a wide range of language tasks, such as question answering, language inference, and sentiment analysis.

The application of pre-trained language representations like BERT to downstream tasks can be approached via two main strategies: feature-based and fine-tuning.

1. Feature-based Approach: In this approach, the pre-trained representations are used as additional features for the downstream task. A common example of this strategy is the use of pre-trained word embeddings (such as GloVe or Word2Vec) where the embeddings are fixed and only the weights of the subsequent layers are trained to perform a specific task. For BERT, this would involve extracting the contextual embeddings from one of the BERT layers and then using these embeddings as input features to a separate model designed for the downstream task. The main advantage of this approach is its flexibility, as it allows for the use of pre-trained representations in a wide variety of models and tasks. However, it might not leverage the full potential of the pre-trained model since only the extracted features are used and not the model's architecture or training capabilities.

2. Fine-tuning Approach: Fine-tuning involves starting with a pre-trained model and continuing the training process on the downstream task with a much smaller dataset. For BERT, this means adding a small number of task-specific output layers on top of the pre-trained BERT model, and then training all the parameters end-to-end on the downstream task. This approach leverages the pre-trained weights as a starting point, which can significantly reduce the amount of data required to achieve high performance on a specific task. Fine-tuning can adjust both the deep pre-trained parameters and the newly added task-specific parameters, allowing the model to adapt more thoroughly to the task at hand. The fine-tuning process is generally faster and requires less data than training a model from scratch, making it a powerful strategy for applying BERT to a wide range of NLP tasks.

Both strategies have their own sets of advantages and considerations, and the choice between them depends on the specific requirements of the task, the available computational resources, and the size of the task-specific dataset. Fine-tuning has become the more popular approach for leveraging models like BERT, as it often leads to superior performance across a variety of tasks with minimal task-specific adjustments.

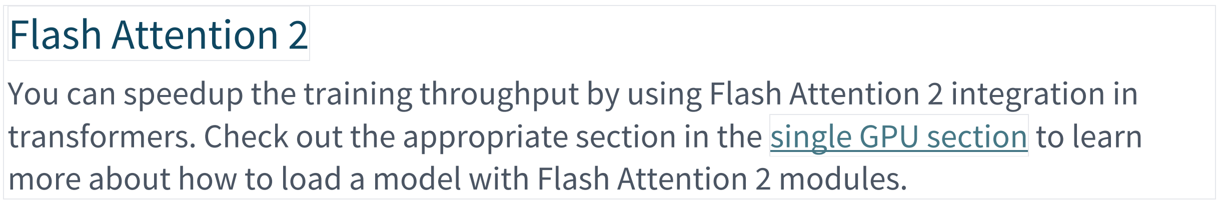
A **downstream task** in the context of machine learning, and specifically in natural language processing (NLP), refers to a specific application or problem that benefits from the use of a pre-trained model. Downstream tasks are essentially the target tasks for which the pre-trained models, such as BERT, are fine-tuned or adapted to perform specific functions or to make predictions based on the learned representations. These tasks are called "downstream" because they lie downstream in the workflow, utilizing the upstream pre-training phase's learned knowledge and capabilities.

Downstream tasks often involve specific datasets and objectives that require understanding, generating, or analyzing text. Examples of downstream tasks in NLP include:

1. Sentiment Analysis: Determining whether a piece of text expresses positive, negative, or neutral sentiment.
2. Question Answering: Providing answers to questions based on the content of a given text.
3. Named Entity Recognition (NER): Identifying and classifying key elements in text into predefined categories, such as the names of people, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.
4. Text Summarization: Generating a concise and fluent summary while retaining the key information and overall meaning.
5. Language Inference: Determining the relationship between sentences, such as whether one sentence entails another, contradicts it, or neither.
6. Machine Translation: Translating text from one language to another.

The pre-training phase involves learning general language representations from large corpora of text, which captures a wide range of language understanding capabilities. The downstream phase, on the other hand, focuses on leveraging these capabilities to perform well on specific tasks by fine-tuning the pre-trained model with task-specific data. This allows the model to adjust its pre-learned representations to better suit the nuances and requirements of the particular task at hand.

FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning



Flash Attention 2

You can speedup the training throughput by using Flash Attention 2 integration in transformers. Check out the appropriate section in the single GPU section to learn more about how to load a model with Flash Attention 2 modules.

FlashAttention-2 is an advancement in the domain of Transformers, particularly addressing the challenge of scaling Transformers to manage longer sequence lengths. This challenge is pivotal for improving performance in various applications, such as language modeling, high-resolution image understanding, and generation tasks in code, audio, and video. The core issue with scaling lies in the attention layer, where runtime and memory requirements increase quadratically with sequence length, presenting a significant bottleneck.

The original FlashAttention introduced a method that leverages the asymmetric GPU memory hierarchy to achieve significant memory savings—moving from a quadratic to a linear memory requirement—and a runtime speedup of 2-4x compared to optimized baselines, without resorting to approximation. Despite these improvements, FlashAttention's efficiency was limited, achieving only 25-40% of the theoretical maximum floating-point operations per second (FLOPs/s), primarily due to suboptimal work partitioning among the GPU's thread blocks and warps. This suboptimal partitioning led to either low occupancy or unnecessary memory operations, hindering performance.

FlashAttention-2 proposes an enhanced approach with improved work partitioning mechanisms to address these inefficiencies. Specifically, it introduces three key improvements:
1. Algorithmic Tweaks: Adjustments are made to the algorithm to reduce the number of non-matrix multiplication (non-matmul) floating-point operations (FLOPs), streamlining the process.
2. Parallelization of Attention Computation: Even for a single attention head, the computation is parallelized across different thread blocks on the GPU. This strategy increases the occupancy, utilizing the GPU's resources more efficiently.
3. Work Distribution Within Thread Blocks: Within each thread block, the workload is distributed among warps to minimize communication through shared memory, thus reducing the overhead associated with such operations.

These enhancements enable FlashAttention-2 to achieve approximately a 2x speedup over the original FlashAttention, reaching 50-73% of the theoretical maximum FLOPs/s on NVIDIA A100 GPUs. This performance is nearing the efficiency of optimized matrix-multiply (GEMM) operations, which are a cornerstone of high-performance computing in deep learning.

Empirical validation demonstrates that when FlashAttention-2 is integrated into the training process of GPT-style models, it can achieve training speeds of up to 225 teraFLOPs/s per A100 GPU, translating to a model FLOPs utilization rate of 72%. This significant improvement not only enhances the training efficiency of large-scale models but also opens up new possibilities for handling longer sequences in various applications, making it a critical advancement in the field of deep learning and Transformers.

Training LLMs on single GPUs

- Practical techniques to enhance model training efficiency on a single GPU focus on **optimizing memory utilization and training speed**.
- These techniques remain valid for multi-GPU setups, which can benefit from additional parallelism methods.

Key considerations for training large models:
- Balancing data throughput/training time with model performance.
- Maximizing throughput (samples/second) by utilizing GPU to its limit.
- Using memory optimization techniques like gradient accumulation if the desired batch size exceeds GPU memory limits.

- Determining the optimal batch size through hyperparameter tuning to maximize resource efficiency.

Methods and tools for training optimization:
- Batch Size Choice: Improves training speed and optimizes memory.
- Gradient Accumulation: Optimizes memory by allowing larger effective batch sizes without increasing GPU memory usage.
- Gradient Checkpointing: Optimizes memory by saving only a subset of activations during training.
- Mixed Precision Training: Speeds up training; may save memory depending on model size and batch size.
- Optimizer Choice: Affects both training speed and memory utilization.
- Data Preloading: Enhances training speed by ensuring efficient data feeding to the GPU.
- DeepSpeed Zero: Optimizes memory usage, particularly beneficial for large models.
- torch.compile: Boosts training speed without memory optimization.
- Parameter-Efficient Fine Tuning (PEFT): Reduces memory footprint by adding trainable parameters on top of a frozen model.

Additional optimization strategies:
- Custom Docker containers with efficient software prebuilds.
- Models utilizing Mixture of Experts (MoE) for parameter efficiency.
- Conversion to BetterTransformer for leveraging PyTorch native attention mechanisms.
- Consideration of multi-GPU setups if single-GPU optimizations are insufficient.

Notable points:
- Gradient accumulation increases effective batch size without additional GPU memory but may slow down training.
- Mixed precision training leverages lower-precision formats for speed, with potential memory savings.
- Optimizer choice, such as AdamW variants or Adafactor, impacts both speed and memory usage.
- Techniques like data preloading, DeepSpeed Zero, and torch.compile offer various efficiency improvements.

These techniques are applicable across different training frameworks, including Trainer and PyTorch loops, and can be configured with tools like 🤗 Accelerate for flexible optimization.
https://huggingface.co/docs/transformers/perf_train_gpu_one

Efficient Training on Multiple GPUs

- Transition to multi-GPU setups when single GPU limitations are reached, applying single-GPU optimization strategies beforehand.
- Parallelism forms used in multi-GPU training include data parallelism, tensor parallelism, and pipeline parallelism, tailored to specific hardware configurations.

Scalability Strategy:
- Estimate vRAM requirements using tools like the 🤗 Model Memory Calculator for models on the 🤗 Hub.

Parallelization Strategies:

For Single Node / Multi-GPU Setup:
- Model fits on a single GPU: Use Distributed DataParallel (DDP) or experiment with ZeRO for potentially faster results.
- Model too large for a single GPU: Consider Pipeline Parallel (PP), ZeRO, or Tensor Parallel (TP) strategies, depending on connectivity (e.g., NVLINK).
- Largest layer doesn't fit on a single GPU: Mandatory use of Tensor Parallel (TP) or ZeRO with additional single-GPU optimizations.

For Multi-Node / Multi-GPU Setup:
- Fast inter-node connectivity: Opt for ZeRO or a combination of PP, TP, and Data Parallel (DP) for fewer communications.
- Slow inter-node connectivity: Combine DP with PP, TP, and ZeRO for efficiency.

Data Parallelism (DP) vs. DistributedDataParallel (DDP):
- DDP is preferred over DP for its efficiency in multi-GPU setups due to reduced communication overhead and balanced workload.

ZeRO Data Parallelism:
- Splits model parameters across GPUs to reduce memory footprint, allowing each GPU to hold only a fraction of the model.

Pipeline Parallelism (PP):
- Splits model into stages across GPUs, processing different batches simultaneously to reduce idle times and increase efficiency.

Tensor Parallelism (TP):
- Divides model tensors across GPUs for parallel processing, requiring fast GPU interconnects for efficiency.

Combining Parallelism Strategies:
- Strategies like DP, PP, and TP can be combined in various configurations (e.g., 2D or 3D parallelism) for optimized performance based on specific hardware and model requirements.

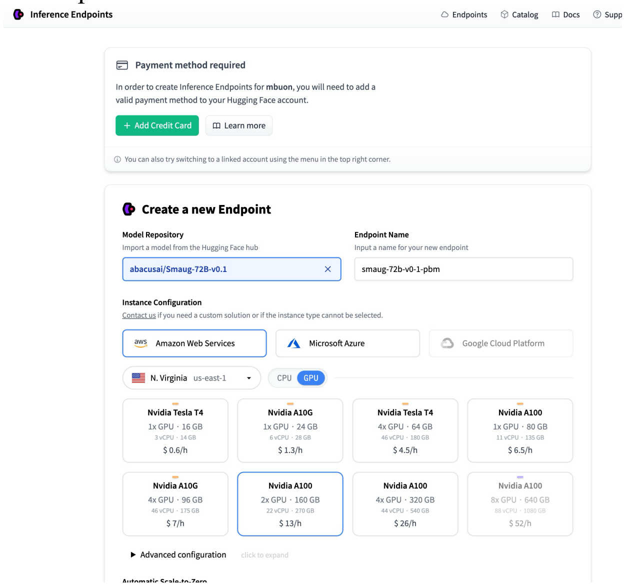
FlexFlow:
- Offers a dynamic approach to parallelism, optimizing across multiple dimensions (Sample, Operator, Attribute, Parameter) for static workloads.

GPU Selection:
- Control over the number of GPUs and their selection order can be managed through environment variables like CUDA_VISIBLE_DEVICES, optimizing resource usage according to the specific hardware setup and model requirements.

This comprehensive approach to multi-GPU training encompasses a variety of parallelism strategies and tools, allowing for tailored optimizations that leverage the full potential of available hardware resources for efficient and scalable model training.
https://huggingface.co/docs/transformers/perf_train_gpu_many

The open-source advantage

"Smaug-72B from Abacus AI is available now on Hugging Face, is on top of the LLM leaderboard, and is the first model with an average score of 80!! In other words, it is the world's best open-source foundation model," said Abacus AI CEO Bindu Reddy in a post on X.com.
"Our next goal will be to publish these techniques as a research paper and apply them to some of the best Mistral Models, including miqu (a 70B fine-tine of LLama-2)," she added. "The techniques we used specifically target reasoning and math skills, which explains the high GSM8K scores! Our upcoming paper will explain more."



Payment method required

In order to create Inference Endpoints for **mbuon**, you will need to add a valid payment method to your Hugging Face account.

You can also try switching to a linked account using the menu in the top right corner.

Create a new Endpoint

Model Repository
Import a model from the Hugging Face hub
abacusai/Smaug-72B-v0.1

Endpoint Name
Input a name for your new endpoint
smaug-72b-v0-1-pbm

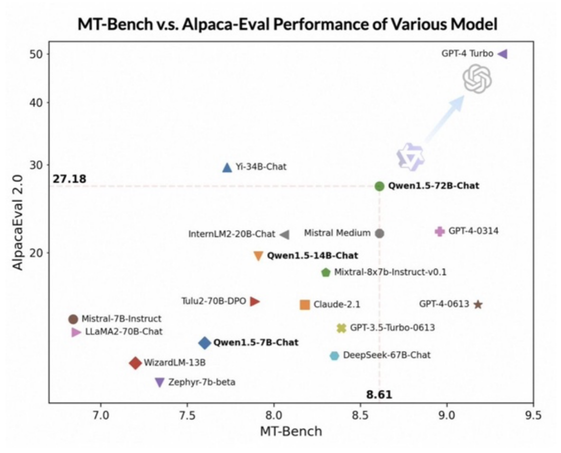
Instance Configuration
Contact us if you need a custom solution or if the instance type cannot be selected.

[✓ Amazon Web Services] Microsoft Azure Google Cloud Platform

N. Virginia us-east-1

CPU [✓ GPU]

Nvidia Tesla T4 1x GPU · 16 GB 3 vCPU · 14 GB $ 0.6/h	Nvidia A10G 1x GPU · 24 GB 6 vCPU · 28 GB $ 1.3/h	Nvidia Tesla T4 4x GPU · 64 GB 46 vCPU · 180 GB $ 4.5/h	Nvidia A100 1x GPU · 80 GB 11 vCPU · 135 GB $ 6.5/h
Nvidia A10G 4x GPU · 96 GB 46 vCPU · 175 GB $ 7/h	[✓ Nvidia A100 2x GPU · 160 GB 22 vCPU · 270 GB $ 13/h]	Nvidia A100 4x GPU · 320 GB 44 vCPU · 540 GB $ 26/h	Nvidia A100 8x GPU · 640 GB 88 vCPU · 1080 GB $ 52/h



Transformers
https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)

RNN: https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/

The Activation Function

We can use any activation function we like in the recurrent neural network. Common choices are:

- Sigmoid function: $\frac{1}{1+e^{-x}}$
- Tanh function: $\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- Relu function: max$(0, x)$

Rectified linear units find applications in computer vision[8] and speech recognition[11][12] using deep neural nets and computational neuroscience.[13][14][15]

Training a Recurrent Neural Network

The backpropagation algorithm of an artificial neural network is modified to include the unfolding in time to train the weights of the network. This algorithm is based on computing the gradient vector and is called backpropagation in time or BPTT algorithm for short.

RNN Architectures

There are different variations of RNNs that are being applied practically in machine learning problems:

Bidirectional Recurrent Neural Networks (BRNN)

In BRNN, inputs from future time steps are used to improve the accuracy of the network. It is like knowing the first and last words of a sentence to predict the middle words.

Gated Recurrent Units (GRU)

These networks are designed to handle the vanishing gradient problem. They have a reset and update gate. These gates determine which information is to be retained for future predictions.

Long Short Term Memory (LSTM) with 3 gates INPUT, OUTPUT, FORGET GATE

LSTMs were also designed to address the vanishing gradient problem in RNNs. LSTMs use three gates called input, output, and forget gate. Similar to GRU, these gates determine which information to retain.
It is a recurrent neural network trained using Backpropagation Through Time that overcomes the vanishing gradient problem. As such, it can be used to create large recurrent networks that, in turn, can be used to address difficult sequence problems in machine learning and achieve state-of-the-art results.

Instead of neurons, LSTM networks have memory blocks connected through layers.

A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A block operates upon an input sequence, and **each gate within a block uses the sigmoid activation** units to control whether it is triggered or not, making the change of state and addition of information flowing through the block conditional.

There are three types of gates within a unit:

- **Forget Gate**: conditionally decides what information to throw away from the block
- **Input Gate**: conditionally decides which values from the input to update the memory state
- **Output Gate**: conditionally decides what to output based on input and the memory of the block

Each unit is like a mini-state machine where the gates of the units have weights that are learned during the training procedure.

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid (default) or tanh activation functions are used. It can be a good practice to rescale the data to the range of 0-to-1, also called normalizing. You can easily normalize the dataset using the **MinMaxScaler** preprocessing class from the scikit-learn library.

```
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)

# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

LSTM for Regression Using the Window Method

You can also phrase the problem so that multiple, recent time steps can be used to make the prediction for the next time step.

This is called a window, and the size of the window is a parameter that can be tuned for each problem.

SageMaker JumpStart

PDFRSS

SageMaker JumpStart provides pretrained, open-source models for a wide range of problem types to help you get started with machine learning. You can incrementally train and tune these models before deployment. JumpStart also provides solution templates that set up infrastructure for common use cases, and executable example notebooks for machine learning with SageMaker.

You can deploy, fine-tune, and evaluate pretrained models from popular models hubs through the JumpStart landing page in the updated Studio experience.

https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart.html

In most cases, you will start your generative AI projects with an existing foundation model from a model hub such as Hugging Face Model Hub, PyTorch Hub, or Amazon SageMaker JumpStart. A model hub is a collection of models that typically contains detailed model descriptions including the use cases that they address.

Throughout this book, we will use Hugging Face Model Hub and SageMaker JumpStart to access foundation models like **Llama** 2 from Meta (Facebook) and ∫from the Technology Innovation Institute (TII) and **FLAN-T5** from Google.

in the journey).

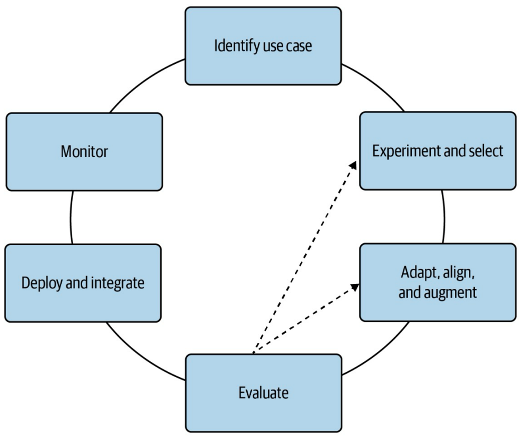


Figure 1-3. Generative AI project life cycle framework

Generative AI models are capable of carrying out many different tasks with great success. However, you will need to decide if an existing foundation model is suitable for your application needs. In Chapter 2, you will learn how to work with these existing foundation models right out of the box using techniques called prompt engineering and in-context learning.

We recommend that you try different models for your generative use case and task. Start with an existing, well-documented, relatively small (e.g., 7 billion-parameter) foundation model to iterate quickly and learn the unique ways of interacting with these generative AI models with a relatively small amount of hardware (compared to the larger 175+ billion-parameter models).

ready to scale your efforts to a larger distributed cluster, you would then migrate to SageMaker distributed training jobs to scale to a larger compute cluster using accelerators like the NVIDIA GPU or AWS Trainium

While you may be able to avoid accelerators initially, you will very likely need to use them for longer-term development and deployment of more complex models. The sooner you learn the unique—and sometimes obscure—aspects of developing with accelerators like **NVIDIA GPUs or AWS Trainium chips**, the better. Fortunately, a lot of the complexity has been abstracted by the hardware provider through the NVIDIA CUDA library and AWS Neuron SDK, respectively.

Amazon Bedrock is a fully managed service that provides access to models from Amazon (e.g., Titan) and popular third-party providers (e.g., AI21 Labs, Anthropic, Cohere, and Stability AI). This allows you to quickly get started experimenting with available foundation models. Bedrock also allows you to privately customize foundation models with your own data as well as integrate and deploy those models into generative AI”

“Adapting a model to a specific use case, task, or domain often includes augmenting the model with additional data. AWS also provides multiple implementation options for vector stores that store vector embeddings. Vector stores and embeddings are used for retrieval-augmented generation (RAG) to efficiently retrieve relevant information from external data sources to augment the data used with a generative model. The options available include vector engine for Amazon OpenSearch Serverless as well as the k-NN plugin available for use with Amazon OpenSearch Service. In addition, both Amazon Aurora PostgreSQL and Amazon Relational Database Services (RDS) for PostgreSQL include vector stores capabilities through built-in pgvector support.

If you are looking for a fully managed semantic search experience on domain-specific data, you can use **Amazon Kendra**, which creates and manages the embeddings for you.

A generative AI application includes more than generative models.

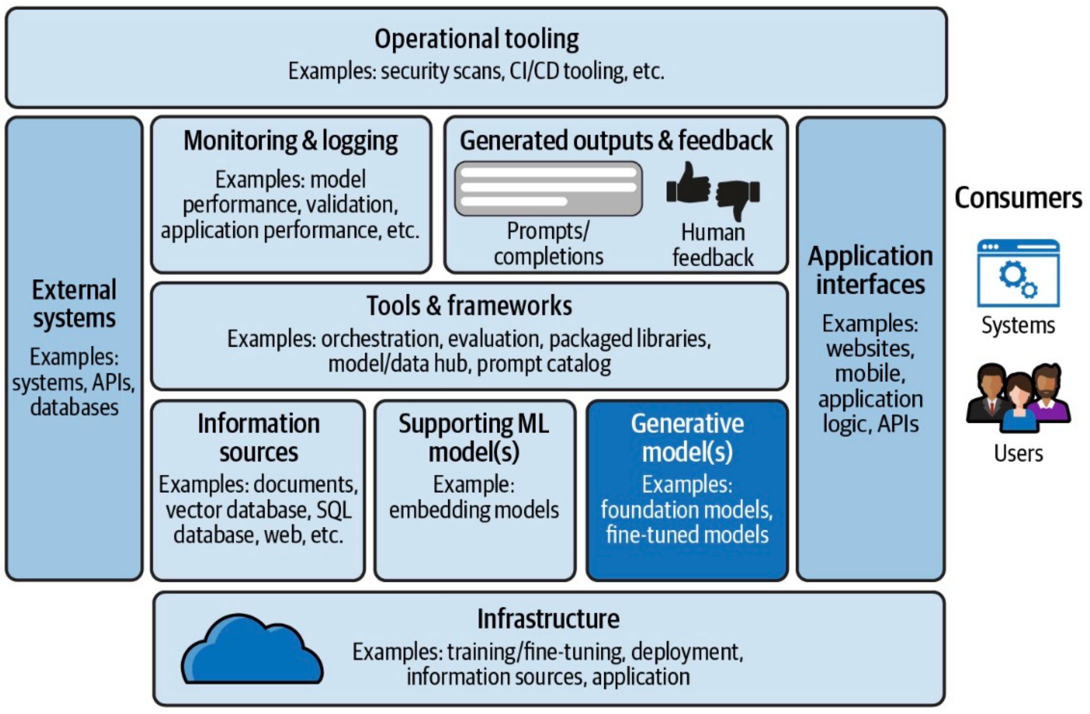

Operational tooling
Examples: security scans, CI/CD tooling, etc.
External systems
Examples: systems, APIs, databases
Monitoring & logging
Examples: model performance, validation, application performance, etc.
Generated outputs & feedback
Prompts/ completions
Human feedback
Application interfaces
Examples: websites, mobile, application logic, APIs
Consumers
Systems
Users
Tools & frameworks
Examples: orchestration, evaluation, packaged libraries, model/data hub, prompt catalog
Information sources
Examples: documents, vector database, SQL database, web, etc.
Supporting ML model(s)
Example: embedding models
Generative model(s)
Examples: foundation models, fine-tuned models
Infrastructure
Examples: training/fine-tuning, deployment, information sources, application

Figure 1-5. Generative AI applications include more than foundation models

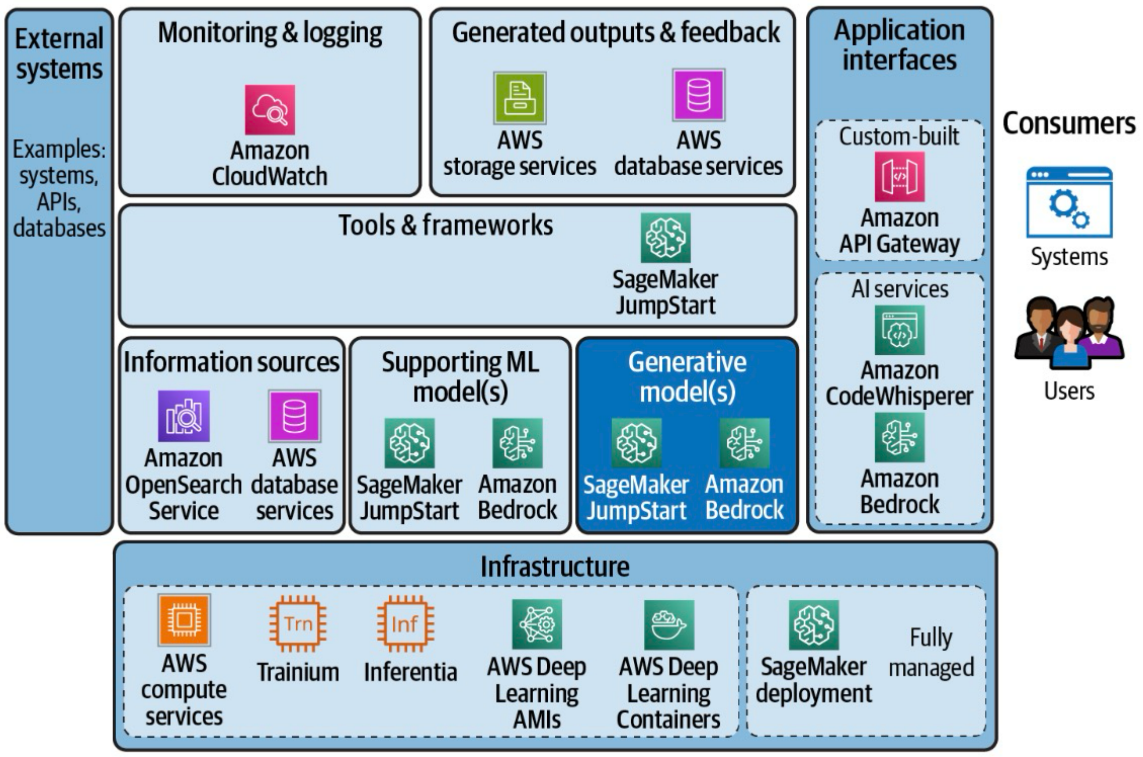


Figure 1-6. AWS breadth of service to enable customers to build generative AI applications

You will also learn how to use **in-context-learning** to pass multiple prompt-completion pairs (e.g., question-answer pairs) in the "context" along with your prompt input. This in-context learning nudges the model to respond similarly to the prompt-completion pairs in the context. This is one of the more remarkable capabilities of generative models as it temporarily alters the model's behavior for the duration of just that single request.

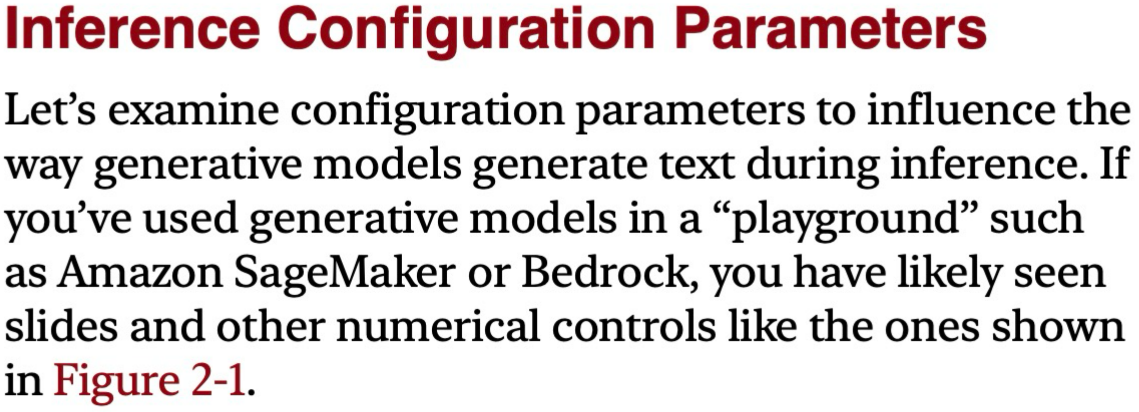
Lastly, you will learn some of the most commonly configured generative parameters like **temperature** and **top k** that control the generative model's creativity when creating content."

"It's important to note that while text-based prompts and completions are implemented and interpreted by humans as natural language sentences, generative models convert them into sequences of tokens, or word fragments. By combining many of these tokens in different elements. By combining many of these tokens in different ways, the model is capable of representing an exponential number of words using a relatively small number of tokens—often on the order of 30,000–100,000 tokens in the model's vocabulary.

“ 1.3 tokens per word,”

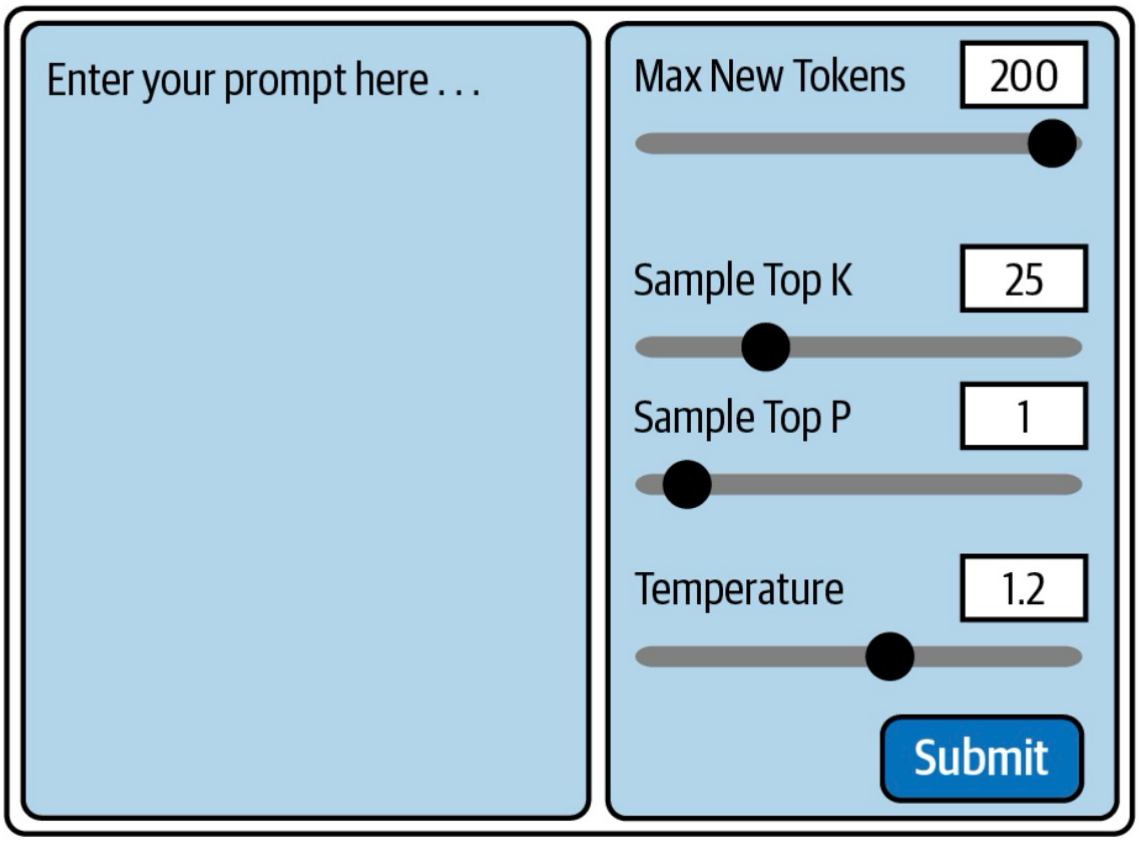
“In-Context Learning with Few-Shot Inference

A powerful technique to help your generative model produce better completions for your prompt is to include a few prompt-completion pairs inside the context portion of your prompt. This is called in-context learning with few-shot inference.”



Inference Configuration Parameters

Let’s examine configuration parameters to influence the way generative models generate text during inference. If you’ve used generative models in a “playground” such as Amazon SageMaker or Bedrock, you have likely seen slides and other numerical controls like the ones shown in Figure 2-1.



Enter your prompt here . . .

Max New Tokens 200

Sample Top K 25

Sample Top P 1

Temperature 1.2

Figure 2-1. Inference configuration parameters to control model outputs

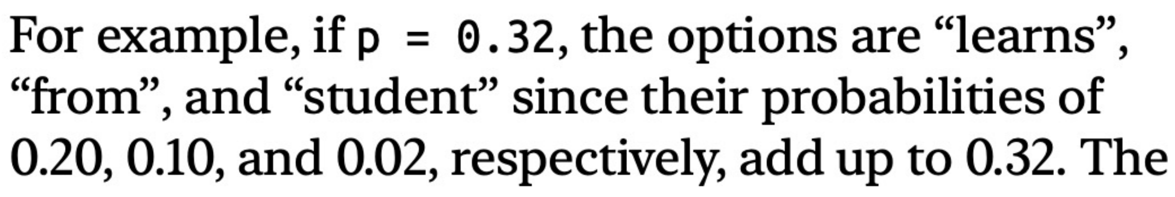
“Greedy versus random sampling”

For each inference request, you can configure the model to choose the next token using either greedy or random sampling. For greedy sampling, the token with the highest probability is selected. With random sampling, the model selects the next token using a random-weighted strategy across all predicted token probabilities. The different sampling methods are shown in Figure 2-2 for the phrase “the student learns from the professor and her lectures.”

top-p and top-k random sampling

These are the most common inference parameters when using random sampling. These parameters provide more fine-grained control for the random sample, which, if used properly, should improve the model’s response while allowing it to be creative enough to fulfill the generative task. **top-k**, as you may have guessed, limits the model to choosing a token randomly from only the top-k tokens with the highest probability. For example, if k is set to 3, you are restricting the model to choose from only the top three tokens using the weighted random-sampling strategy. ***Note that setting top-k to a higher number can help reduce repetitiveness, while setting top-k to 1 basically gives you greedy decoding.***

“top-p limits the model to randomly sampling from the set of tokens whose cumulative probabilities do not exceed p, starting from the highest probability and working down to the lowest probability. “top-p can also produce greater variability and is sometimes used if it is hard to pick a good top-k value. top-p and top-k can also be used together.”


For example, if `p = 0.32`, the options are “learns”, “from”, and “student” since their probabilities of 0.20, 0.10, and 0.02, respectively, add up to 0.32. The

temperature

This parameter also helps to control the randomness of the model output by modifying the shape of the next-token probability distribution. “In contrast to top-k and top-p, changing the temperature actually changes the next-token probability distribution, which ultimately affects the next-token prediction. “A low temperature (below 1, for example) results in stronger peaks where the probabilities are concentrated among a smaller subset of tokens. A higher temperature (above 1, for example) results in a flatter next-token probability distribution where the probabilities are more evenly spread across the tokens. Setting the temperature to 1 leaves the next-token probability distribution unaltered, which represents the distribution learned during model training and tuning."

LLM models have built a solid understanding of human language as well as a massive amount of knowledge across many domains. This is often called **parametric memory**, as the knowledge is captured in the models' parameters.

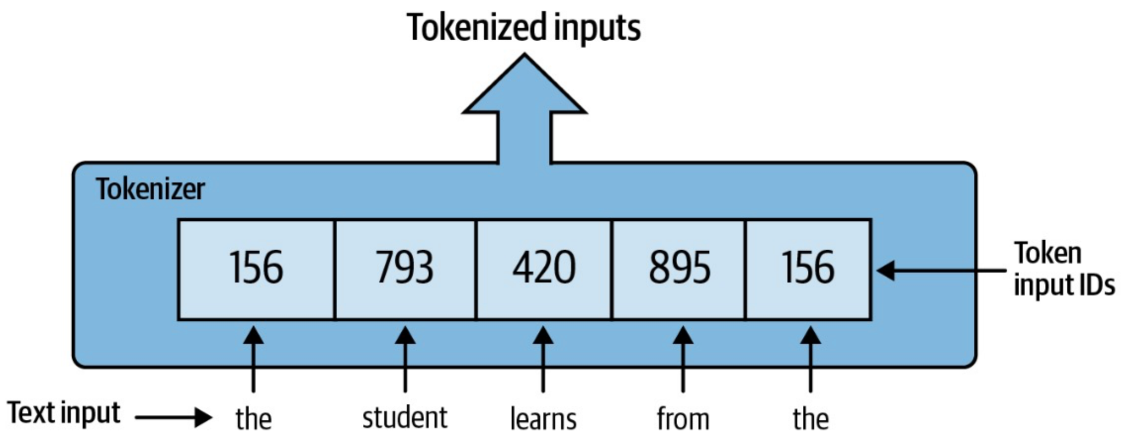


Figure 3-1. Use a tokenizer to convert text inputs into vectors for machine-readable processing

Embedding Vectors

Embedding vectors, often called the "embeddings," have been used in machine learning, information retrieval, and search use cases for decades. Embeddings are a numerical, vectorized representation of any entity of any type, including text, images, videos, and audio clips, projected into very high-dimensional vector spaces.

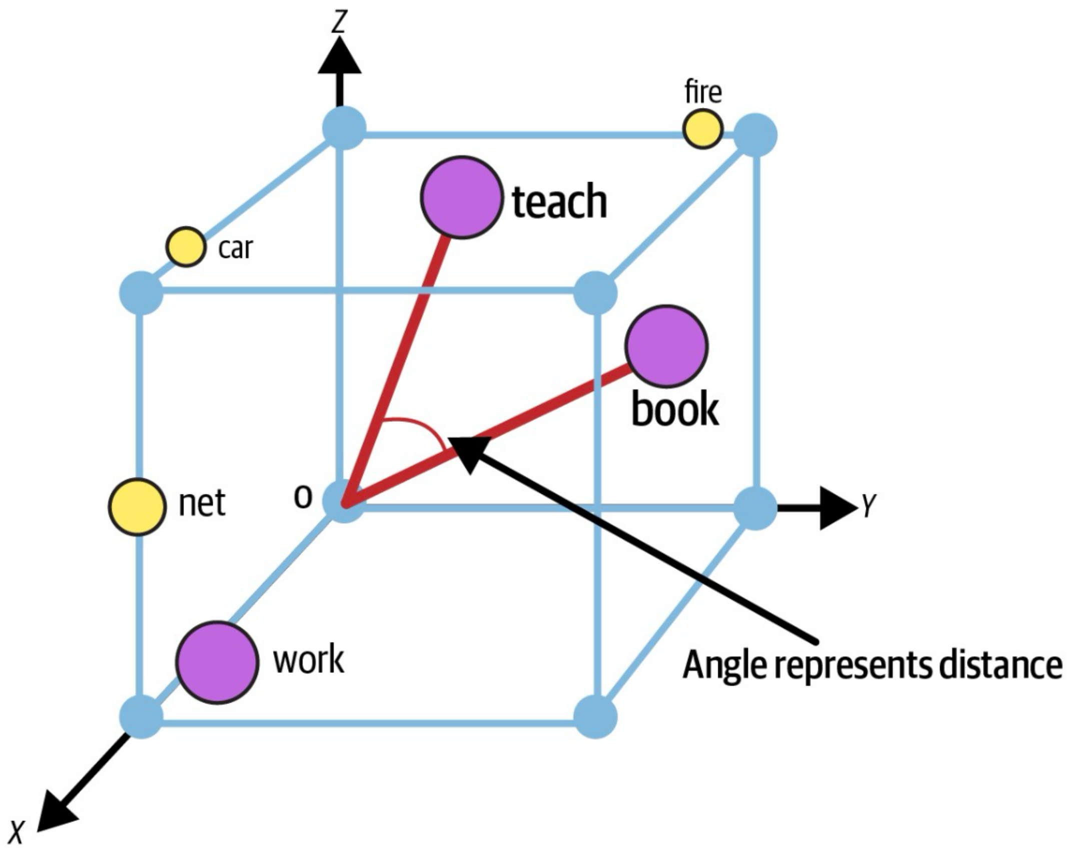


Figure 3-2. Representation of tokens in an example three-dimensional embedding space

“Since these vectors encode the meaning and context of tokens within a larger corpus of text, they allow the model to statistically represent and understand human language. The closer these tokens are to each other in the vector space, the more similar they are in semantic meaning.”

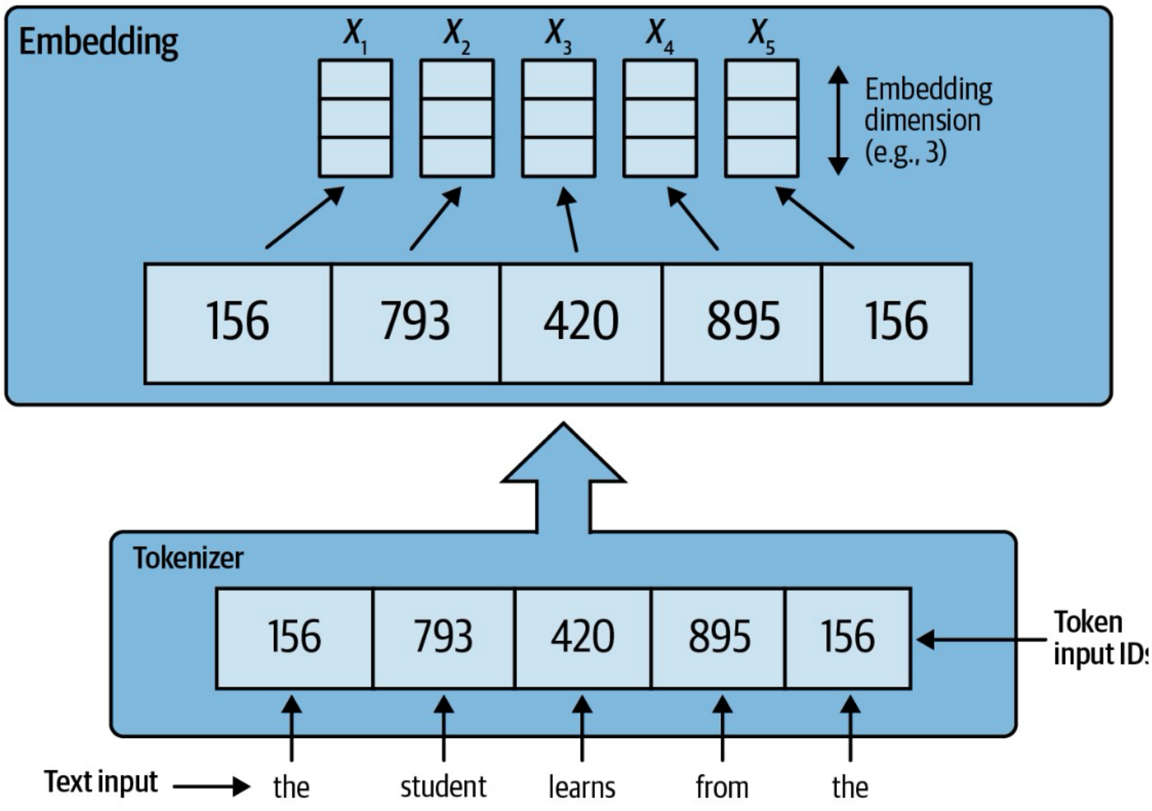


Figure 3-3. Embedding vector space of three dimensions

"Transformer is primarily focused on helping the model generate a completion to a given input prompt. During model pretraining and fine-tuning the Transformer is helping the model gain contextual understanding of the language from the input training/tuning corpus."

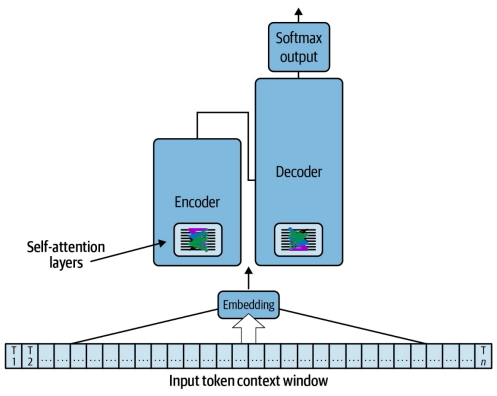


Figure 3-4. High-level Transformer architecture

Il meccanismo di "autoattenzione" associa ogni token dei dati a tutti gli altri token della sequenza di input

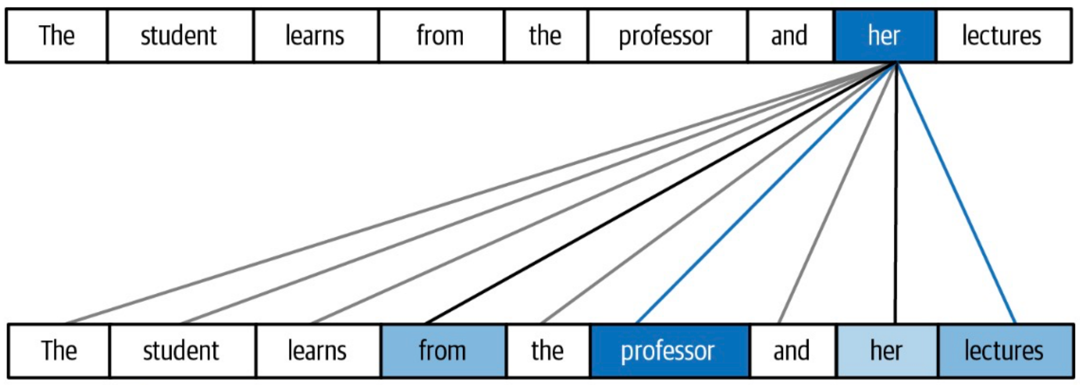


Figure 3-5. The "self-attention" mechanism attends every token in the data to all other tokens in the input sequence

This pairwise attention lets the model learn the contextual dependencies, or contextual understanding, of the input data during model pretraining. By paying attention to the whole input, the Transformer unlocks the model's ability to learn and represent language from the training documents provided. "In practice, the Transformer actually learns multiple sets of self-attention weights through multiheaded attention. Each head runs in parallel over the same input and learns different aspects of the language.

Transformers, introduced in the seminal paper "Attention is All You Need" by Vaswani et al., have become the foundation of modern natural language processing (NLP). They are designed to handle sequential data, like text, in a parallel manner, which significantly improves efficiency over traditional models that process data sequentially (e.g., RNNs and LSTMs). Transformers rely heavily on the self-attention mechanism to weigh the importance of different words within the input data. There are three main types of Transformer architectures: Autoencoders, Autoregressive models, and Encoder-Decoder models.

There are three variants of generative transformer-based models overall: encoder-only, decoder-only, and encoder-decoder. Each variant is trained with a different training objective and, during pretraining, the model weights are updated to minimize the loss of the training objectives described next for each variation. Each variant is capable of addressing different types of generative tasks, as you will see next.

Autoencoders

Autoencoder Transformers are **designed to encode input data into a compact representation and then decode this representation back into the original form or some target form.** The encoding process captures the essential information needed to reconstruct the input. In the context of NLP, autoencoder Transformers like BERT (Bidirectional Encoder Representations from Transformers) learn to predict missing words in a sentence or next sentences, enabling them to understand context and meaning from the input text. They are typically used for tasks like sentence classification, named entity recognition, and question answering, where understanding the input text is crucial.

Autoregressive Models

Autoregressive Transformers generate sequences one token at a time, where the prediction of each new token is dependent on the tokens that have been generated so far. This type of model is inherently sequential in its generation process. GPT (Generative Pretrained Transformer) is a prime example of an autoregressive Transformer. It's trained to predict the next word in a sentence and can generate coherent and contextually relevant text over long passages. Autoregressive models are particularly well-suited for tasks like text generation, language modeling, and machine translation.

Encoder-Decoder Models

Encoder-Decoder Transformers combine both encoding and decoding functionalities but are structured to handle input and output sequences that may have different lengths and are not directly aligned. The encoder processes the input sequence to create a context-rich representation, which the decoder uses to generate the output sequence. This architecture is exemplified is ideal for tasks that involve transforming input data into some output form, such as machine translation or summarization. The encoder captures the meaning of the input text, and the decoder uses this understanding to produce the target text.

Encoder-only models, or **autoencoders**, are pretrained using a technique called **masked language modeling (MLM),** which randomly mask input tokens and try to "predict the masked tokens. This is sometimes called a denoising objective. Autoencoding models use bidirectional representations of the input to better understand the full context of a token—not just the previous tokens in the sequence

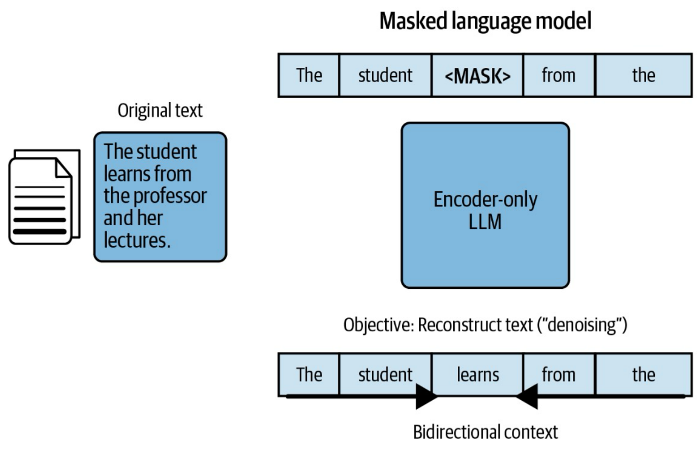


Figure 3-9. Encoder-only (autoencoder) models use a bidirectional context to reconstruct the masked input tokens

Encoder-only models are best suited for language tasks that utilize the embeddings generated by the encoder, such as text classification. They are not particularly useful for generative tasks that continue to generate more text. A well-known encoder-only model is BERT. The embedding outputs are also useful for semantic similarity search—an advanced document-search algorithm beyond simple keyword search. You will explore **semantic similarity search** more in "Retrieval-Augmented Generation"."

Decoder-only models, or autoregressive models, are pretrained using unidirectional causal language modeling (CLM), which predicts the next token using only the previous tokens—every other token is masked

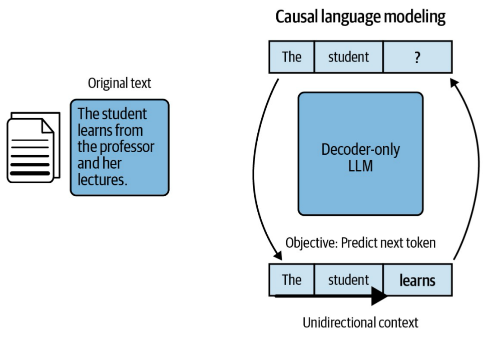


Figure 3-10. Decoder-only (autoregressive) models only reveal the tokens leading up to the token being predicted

These models are the standard for generative tasks, including question-answer. The families of GPT-3, Falcon, and LLaMA models are well-known autoregressive models.

Decoder-only, autoregressive models use millions of text examples to learn a statistical language representation by continuously predicting the next token from the previous tokens. These models are the standard for generative tasks, including question-answer. The families of GPT-3, Falcon, and LLaMA models

Encoder-decoder models, often called sequence-to-sequence models, use both the Transformer encoder and decoder. While the pretraining objectives vary from model to model, the popular T5 foundation model (e.g., FLAN-T5) was pretrained using consecutive multitoken masking called span corruption. The decoder then attempts to reconstruct the masked sequence of tokens, <X>, as shown”

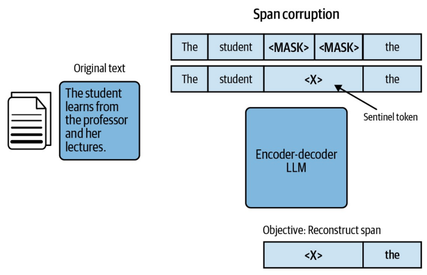


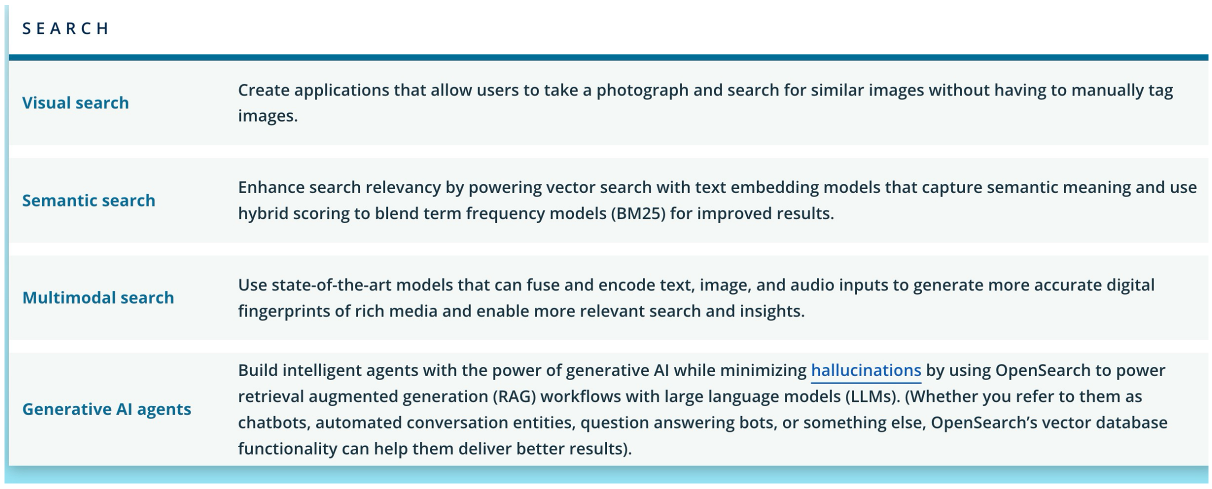
Figure 3-11. Encoder-decoder (sequence-to-sequence) models

Using **OpenSearch** as a vector database brings together the power of traditional search, analytics, and vector search in one complete package. OpenSearch’s vector database capabilities can accelerate artificial intelligence (AI) application development by reducing the effort for builders to operationalize, manage, and integrate AI-generated assets. Bring your models, vectors, and metadata into OpenSearch to power vector, lexical, and hybrid search and analytics, with performance and scalability built in. What is a vector database?

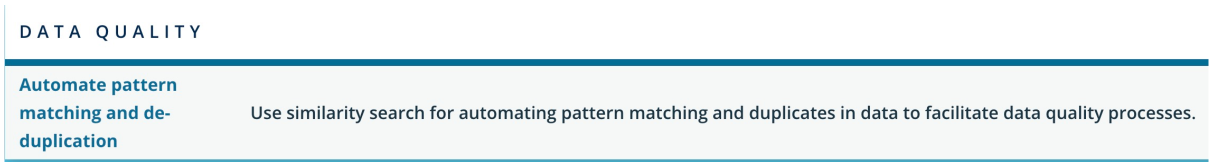
Information comes in many forms: unstructured data, like text documents, rich media, and audio, and structured data, like geospatial coordinates, tables, and graphs. Innovations in AI have enabled the use of models, or embeddings, to encode all types of data into vectors. These vectors are data points in a high-dimensional space that capture the meaning and context of an asset, allowing search tools to find similar assets by searching for neighboring data points.

Vector databases allow you to store and index vectors and metadata, unlocking the ability to use low-latency queries to discover assets by degree of similarity. Typically powered by k-NN indexes built using algorithms like Hierarchical Navigable Small Worlds (HNSW) and Inverted File (IVF) System, vector databases augment k-NN functionality by providing a foundation for applications like data management, fault tolerance, resource access controls, and a query engine.

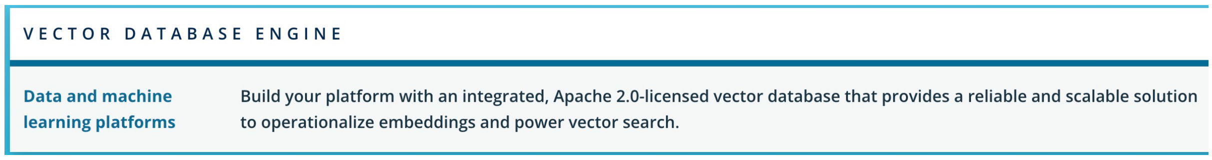
OpenSearch provides an integrated vector database that can support AI systems by serving as a knowledge base. This benefits AI applications like generative AI and natural language search by providing a long-term memory of AI-generated outputs. These outputs can be used to enhance information retrieval and analytics, improve efficiency and stability, and give generative AI models a broader and deeper pool of data from which to draw more accurate and truthful responses to queries.



SEARCH	
Visual search	Create applications that allow users to take a photograph and search for similar images without having to manually tag images.
Semantic search	Enhance search relevancy by powering vector search with text embedding models that capture semantic meaning and use hybrid scoring to blend term frequency models (BM25) for improved results.
Multimodal search	Use state-of-the-art models that can fuse and encode text, image, and audio inputs to generate more accurate digital fingerprints of rich media and enable more relevant search and insights.
Generative AI agents	Build intelligent agents with the power of generative AI while minimizing hallucinations by using OpenSearch to power retrieval augmented generation (RAG) workflows with large language models (LLMs). (Whether you refer to them as chatbots, automated conversation entities, question answering bots, or something else, OpenSearch's vector database functionality can help them deliver better results).



DATA QUALITY	
Automate pattern matching and de-duplication	Use similarity search for automating pattern matching and duplicates in data to facilitate data quality processes.



VECTOR DATABASE ENGINE	
Data and machine learning platforms	Build your platform with an integrated, Apache 2.0-licensed vector database that provides a reliable and scalable solution to operationalize embeddings and power vector search.

https://medium.com/@shankar.arunp/augmenting-large-language-models-with-verified-information-sources-leveraging-aws-sagemaker-and-f6be17fb10a8

Scaling laws

a set of scaling laws have emerged that describe the trade-offs between model size and dataset size for a fixed compute budget (e.g., number of GPU hours). These scaling laws state that **you can achieve better generative model performance by either increasing the number of tokens or the number of model parameters.**

” “Researchers have found that by increasing the training dataset size instead of the model size, you can get state-of-the-art performance that exceeds the 175 billion-parameter models with a much smaller set of weights. In fact, the “Scaling Laws for Neural Language Models” paper shows that if you hold the compute budget constant, model performance may increase when you either increase the training dataset size (and hold model parameter size constant) or increase the number of model parameters (and hold the”

“This also hints that you can improve performance for smaller models by just training them on more data.”

“The Chinchilla paper implies that the massive 100 billion-plus parameter models like GPT-3 may be overparameterized and undertrained. Additionally, they hypothesize that you could achieve 100 billion-plus parameter model performance with a small model by simply providing more training data to the smaller model.

To be more specific, the authors of the Chinchilla paper claim that the optimal training dataset size (measured in tokens) is 20x the number of model parameters and that anything below that 20x ratio is potentially overparameterized and undertrained.
“The Chinchilla paper implies that the massive 100 billion-plus parameter models like GPT-3 may be overparameterized and undertrained. Additionally, they hypothesize that you could achieve 100 billion-plus parameter model performance with a small model by simply providing more training data to the smaller model.

To be more specific, the authors of the Chinchilla paper claim that the optimal training dataset size (measured in tokens) is 20x the number of model parameters and that anything below that 20x ratio is potentially overparameterized and undertrained.

“AWS has also developed purpose-built ML accelerators, **AWS Trainium**, for high-performance and cost-efficient training of **100B+ parameter generative AI models**.

“The largest Trn1 instance, at the time of this writing, is powered by 16 AWS Trainium chips and has 512 GB of shared accelerator memory.

When you try to train a multibillion-parameter model at 32-bit full precision, you will quickly hit the limit of a single NVIDIA A100 or H100 GPU with only 80 GB of GPU RAM. Therefore, you will almost always need to use **quantization** when using a single GPU. "Quantization reduces the memory needed to load and train a model by reducing the precision of the model weights. Quantization converts your model parameters from 32-bit precision down to 16-bit precision—or even 8-bit or 4-bit.

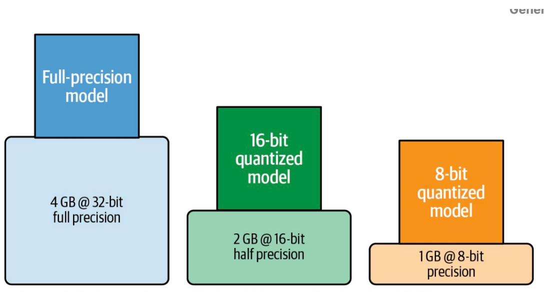


Figure 4-4. Approximate GPU RAM needed to load a 1-billion-parameter model at 32-bit, 16-bit, and 8-bit precision

"bfloat16 has become a popular alternative to fp16 as it captures the full range of fp32 with only 16-bits. This reduces numerical instabilities during model training caused by overflow.

Optimizing the Self-Attention Layers: FlashAttention and Grouped-Query Attention

FlashAttention

The Transformer's attention layer is a bottleneck when trying to scale to longer input sequences because the computation and memory requirements scale quadratically O(n2) with the number of input tokens. FlashAttention, initially proposed in a research paper,2 is a GPU-specific solution to this quadratic scaling problem.

Overall, FlashAttention increases self-attention performance by 2–4x and reduces memory usage 10–20x by reducing the quadratic O(n2) computational and memory requirements down to linear O(n), where n is the number of input tokens in the sequence. With FlashAttention, the Transformer scales to handle much longer input sequences which allows for better performance on larger input context windows

Grouped-Query Attention GQA

"GQA allows queries to be grouped into fewer key and value heads and therefore reduces memory consumption of the attention heads. In addition, GQA improves performance by reducing the number of memory reads and writes."

Parallel distribution Distributed Data Parallel – Fully Sharded Data Parallel

While these techniques are essential to pretraining large foundation models from scratch, **they are also useful for adapting foundation models to your custom datasets and use cases during a process called fine-tuning.**

For larger models, you will likely need to use a distributed cluster of GPUs to train these massive models across hundreds or thousands of GPUs.

- **Various Distributed Computing Patterns**: Includes Distributed Data Parallel (DDP) and Fully Sharded Data Parallel (FSDP).
- **Key Difference**: Lies in the method of splitting—or sharding—the model across GPUs.
- **DDP Use Case**: Chosen when model parameters can fit into a single GPU, loading a single copy of the model into each GPU.
- **FSDP Necessity**: Required if the model is too large for a single GPU, even after quantization, necessitating the sharding of the model across multiple GPUs.
- **Data Handling**: In both DDP and FSDP, data is divided into batches and distributed across all available GPUs to enhance GPU utilization and cost efficiency.
- **Trade-off**: Increased GPU utilization and cost efficiency come at the cost of some communication overhead.

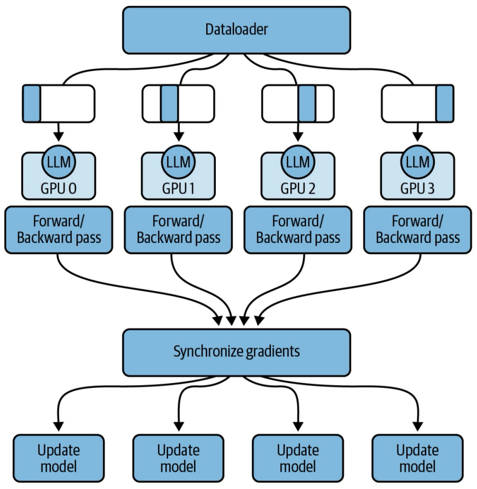


Figure 4-10. Distributed data parallel (DDP)

"PyTorch comes with an optimized implementation of DDP that automatically copies your model onto each GPU (assuming it fits into a single GPU using a technique such as quantization), splits the data into batches, and sends the batches to each GPU in parallel. In traditional data parallel training, **a complete copy of the model is maintained on every GPU,** and only the input data is divided among them. Each GPU computes gradients independently based on its subset of the data, and these gradients are then aggregated across all GPUs to update the model. While this approach is straightforward, it significantly limits the size of models that can be trained, as each GPU must have enough memory to store the entire model and the associated data for gradient computation. **FSDP addresses this limitation by dividing (sharding)**

the model's parameters across the GPUs. Each GPU stores only a portion of the model's parameters, and during the forward and backward passes of training, FSDP dynamically loads the necessary parameters into GPU memory as needed.

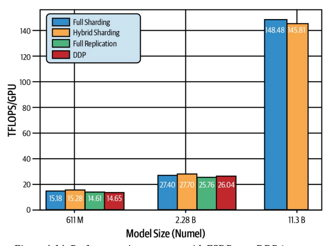


Figure 4-14. Performance improvement with FSDP over DDP (source: adapted from an image in Zhao et al.)

“FSDP is a common distributed computing strategy supported by Amazon SageMaker. T”

Fine-tuning

Instruction dataset: you can achieve very good results with instruction fine-tuning using a relatively small instruction dataset—often just 500–1,000 examples is enough.”

RAG Langchain orchestration architecture

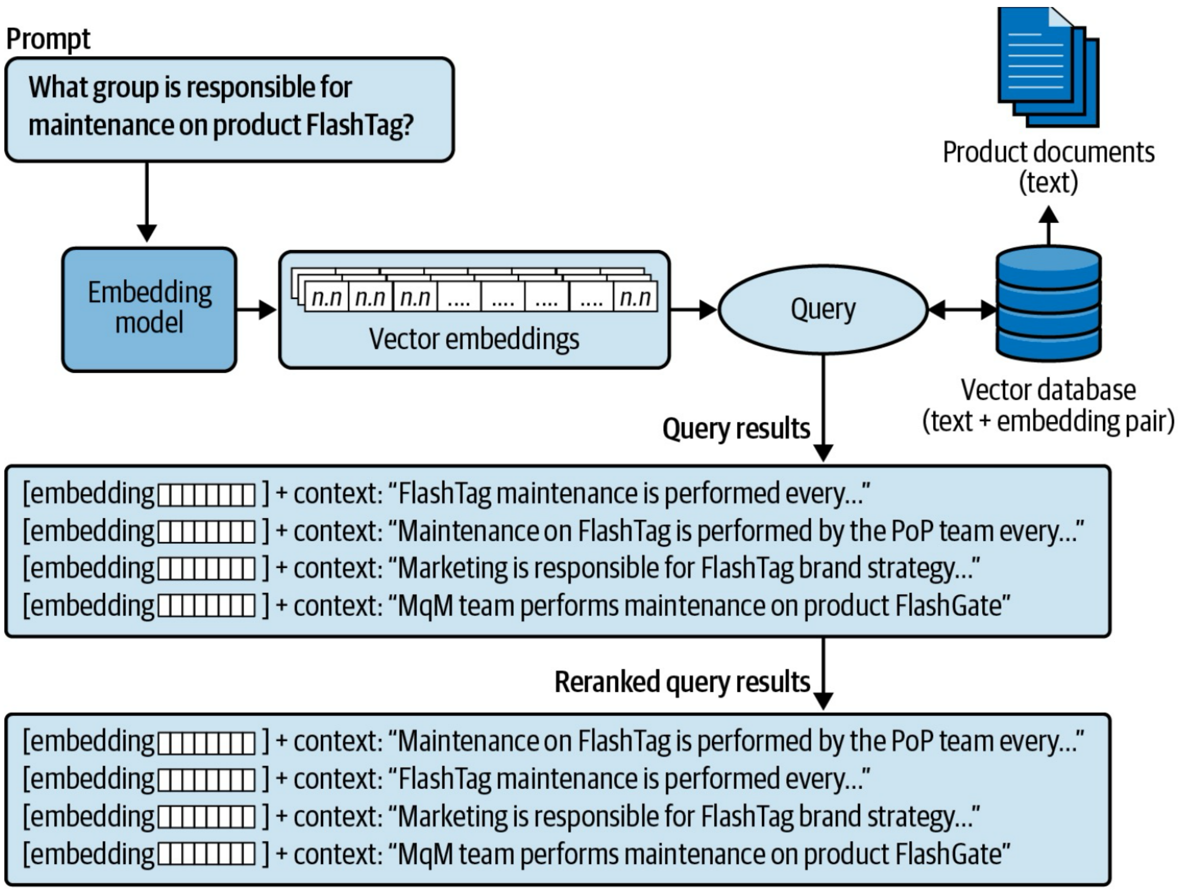

Prompt
What group is responsible for maintenance on product FlashTag?
Embedding model
n.n n.n n.n .... n.n
Vector embeddings
Query
Product documents (text)
Vector database (text + embedding pair)
Query results
[embedding] + context: "FlashTag maintenance is performed every..."
[embedding] + context: "Maintenance on FlashTag is performed by the PoP team every..."
[embedding] + context: "Marketing is responsible for FlashTag brand strategy..."
[embedding] + context: "MqM team performs maintenance on product FlashGate"
Reranked query results
[embedding] + context: "Maintenance on FlashTag is performed by the PoP team every..."
[embedding] + context: "FlashTag maintenance is performed every..."
[embedding] + context: "Marketing is responsible for FlashTag brand strategy..."
[embedding] + context: "MqM team performs maintenance on product FlashGate"

Figure 9-7. Reranking query results before augmenting the prompt

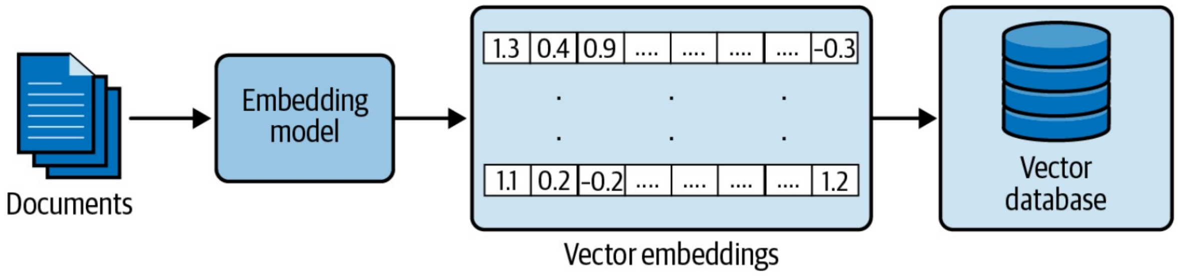


Figure 9-4. Efficient indexing of documents for quick retrieval

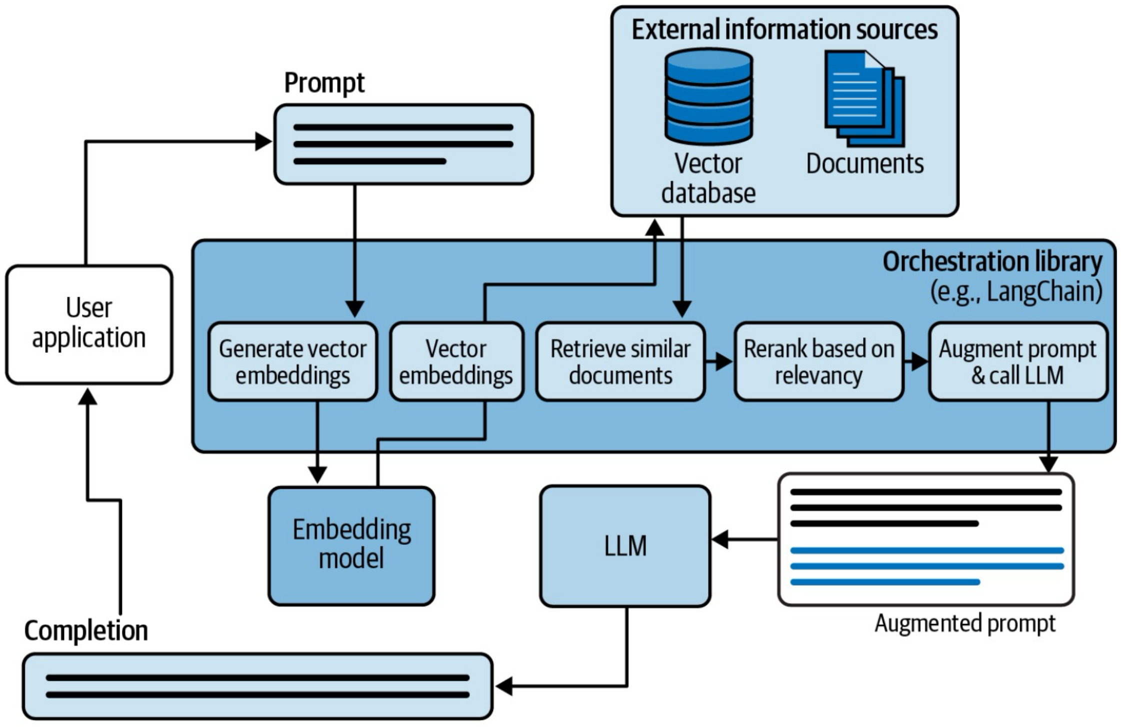

External information sources
Vector database
Documents
Prompt
User application
Orchestration library (e.g., LangChain)
Generate vector embeddings
Vector embeddings
Retrieve similar documents
Rerank based on relevancy
Augment prompt & call LLM
Embedding model
LLM
Augmented prompt
Completion

Figure 9-8. Orchestrating RAG workflows

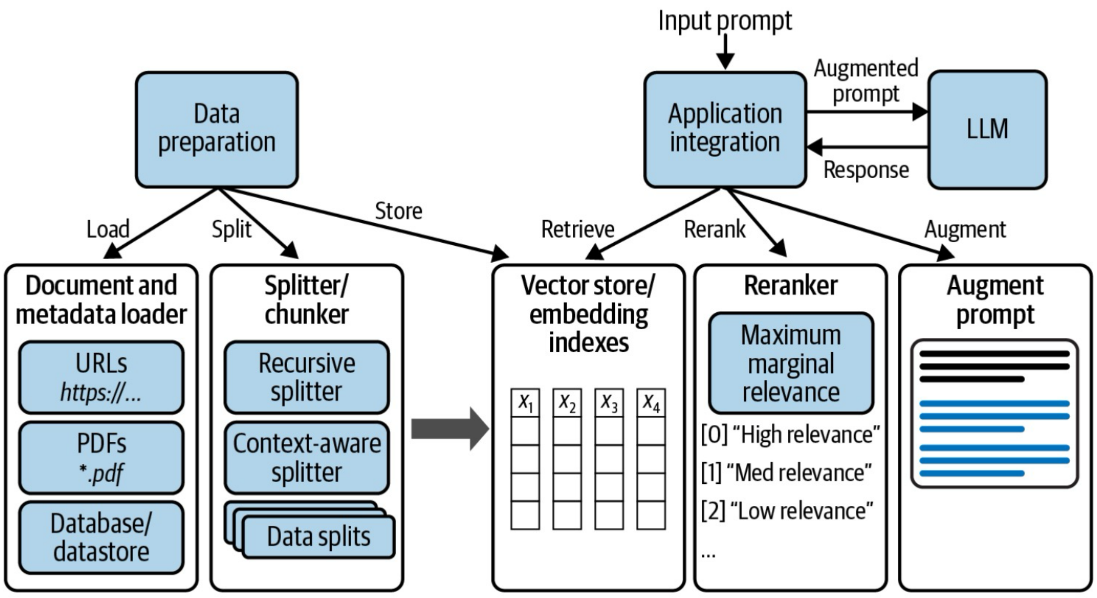


Figure 9-3. RAG architectures depend on efficient data preparation and retrieval techniques for integration into consuming applications

ReAct Framework

ReAct is a prompting strategy that combines CoT *reasoning* with *action planning*. ReAct structures prompts to include a sequence of one or more question, thought, action, and observation examples as described in the ReAct paper and shown in Figure 9-10.

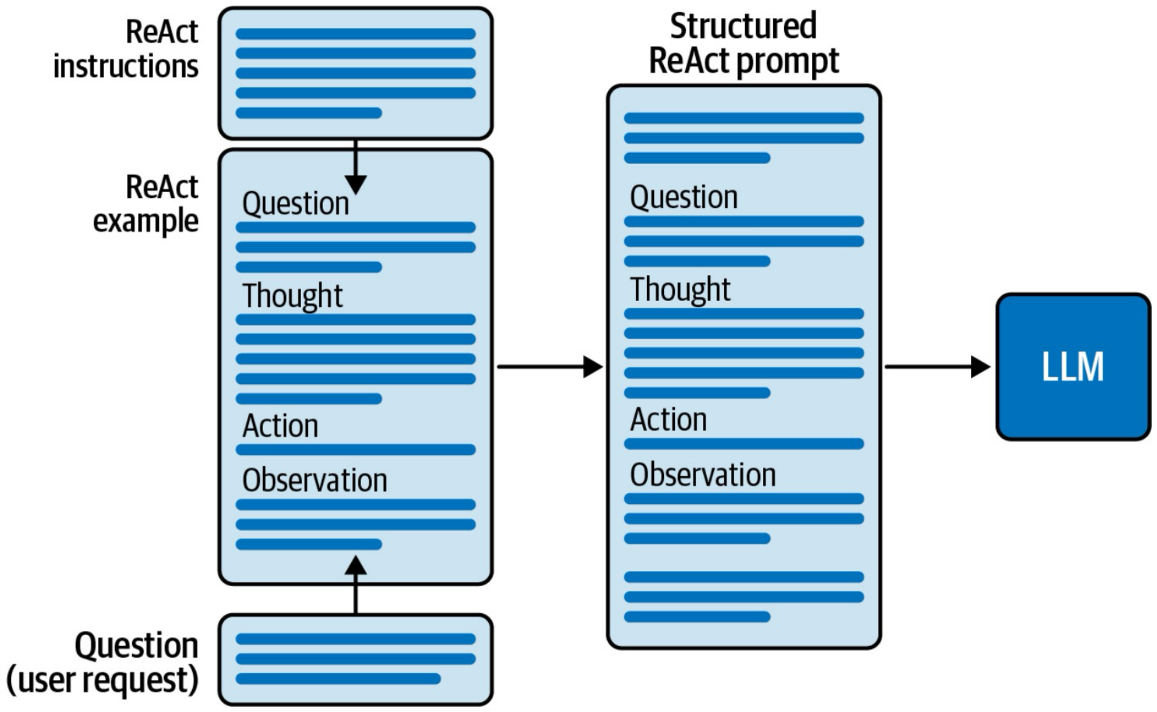


Figure 9-10. ReAct structures prompts to include instructions, ReAct examples, and the user request

Program-Aided Language Framework

PAL uses CoT reasoning to generate programs in the intermediate reasoning steps that help solve the given problem. These programs are then passed to an interpreter, for example, a Python interpreter, that runs the code and returns the result back to the foundation model (FM), as shown in Figure 9-11.

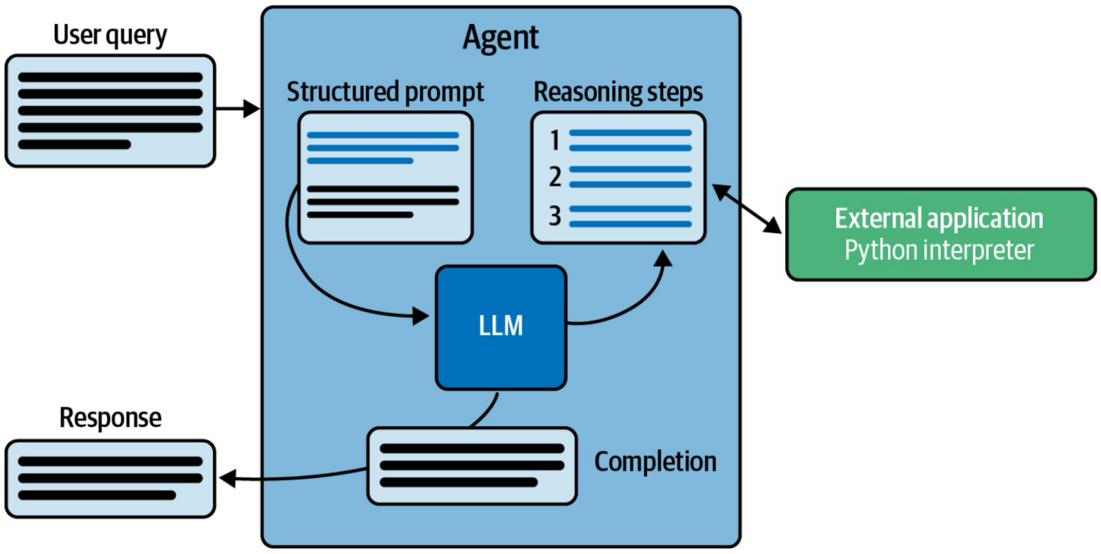


Figure 9-11. PAL connects a foundation model to an external code interpreter to perform calculations

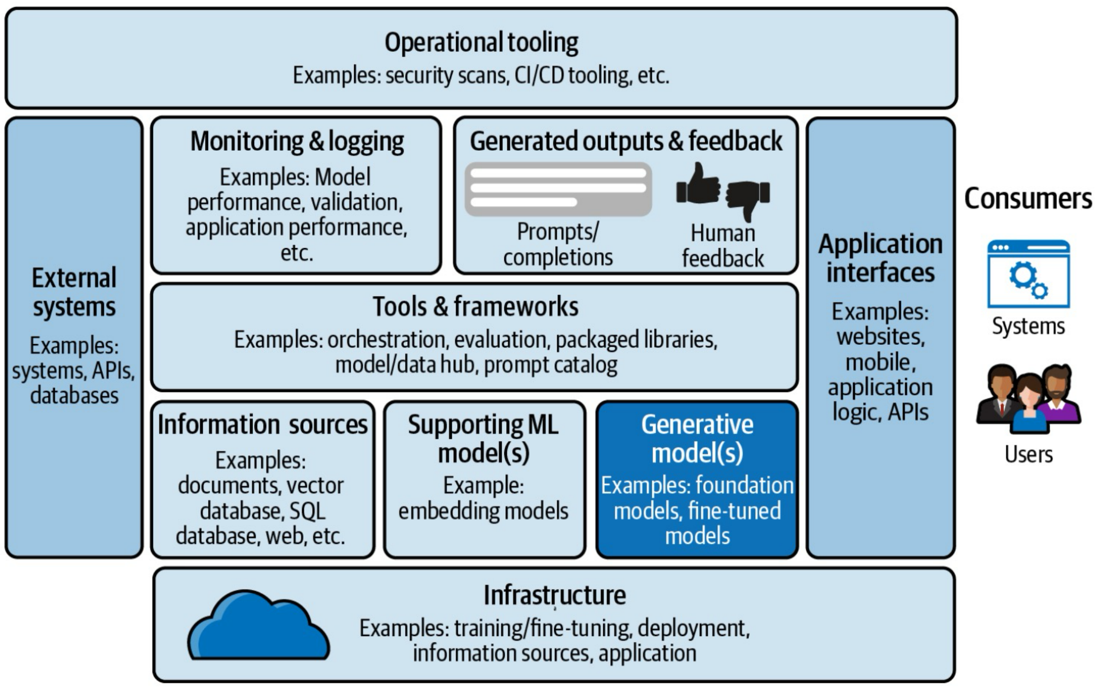

Operational tooling
Examples: security scans, CI/CD tooling, etc.
External systems
Examples: systems, APIs, databases
Monitoring & logging
Examples: Model performance, validation, application performance, etc.
Generated outputs & feedback
Prompts/ completions
Human feedback
Application interfaces
Examples: websites, mobile, application logic, APIs
Consumers
Systems
Users
Tools & frameworks
Examples: orchestration, evaluation, packaged libraries, model/data hub, prompt catalog
Information sources
Examples: documents, vector database, SQL database, web, etc.
Supporting ML model(s)
Example: embedding models
Generative model(s)
Examples: foundation models, fine-tuned models
Infrastructure
Examples: training/fine-tuning, deployment, information sources, application

Figure 9-13. Generative AI applications include more than generative models

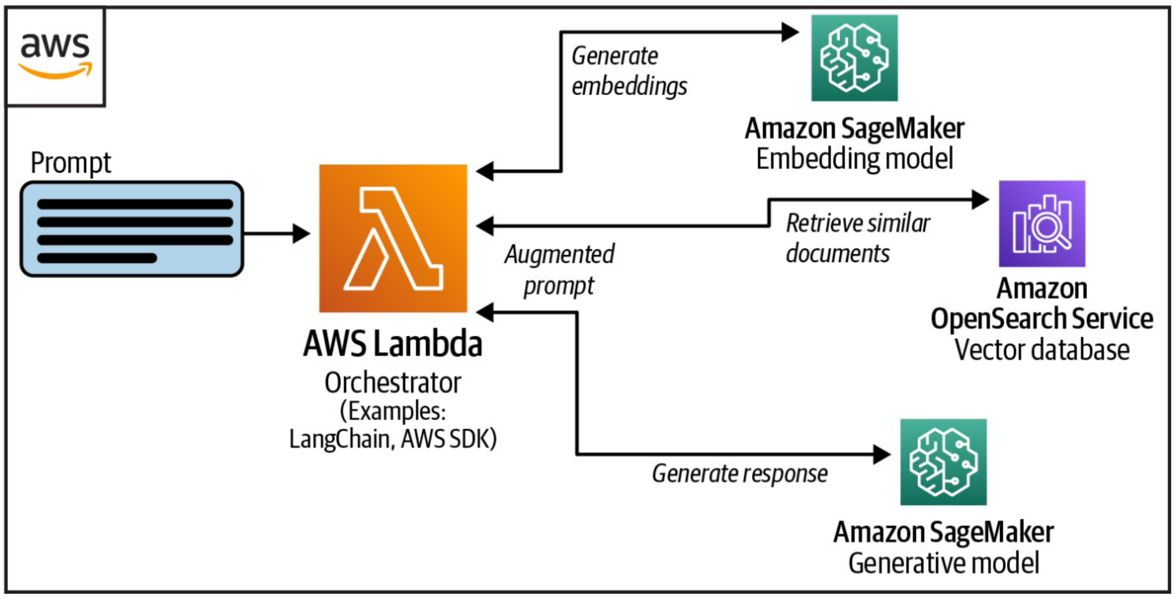


Figure 9-14. Infrastructure powers all application components

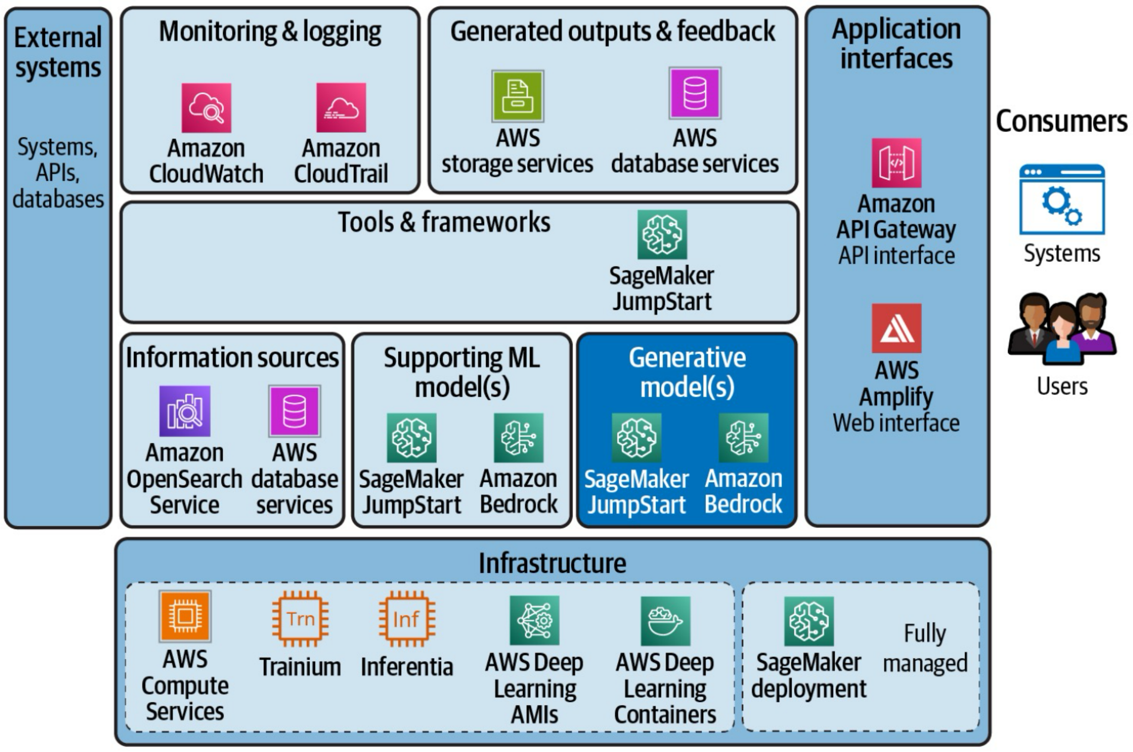


Figure 9-16. Examples of AWS services that can be used to build generative AI applications

As previously mentioned, there are prebuilt generative AI applications, such as Amazon CodeWhisperer, where all of the application components are abstracted away from the consumer and fully managed as part of a packaged application.

GenAIOps, FMOps, or LLMOps: Operationalizing the Generative AI Project Life Cycle

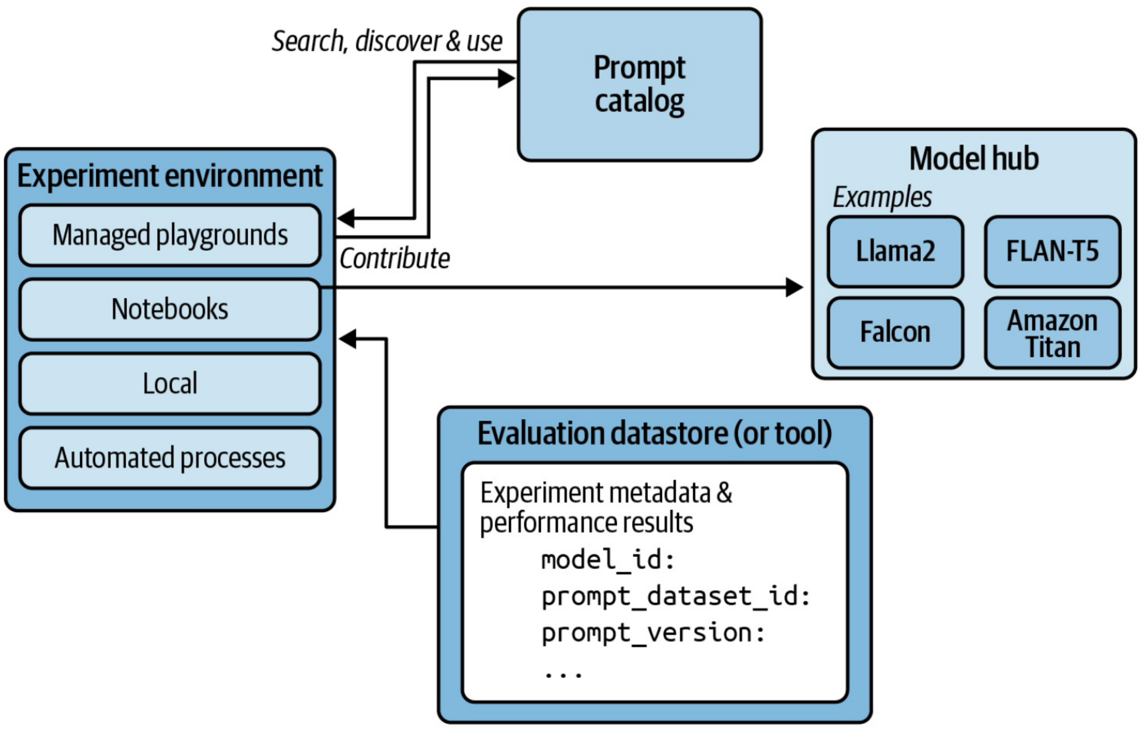
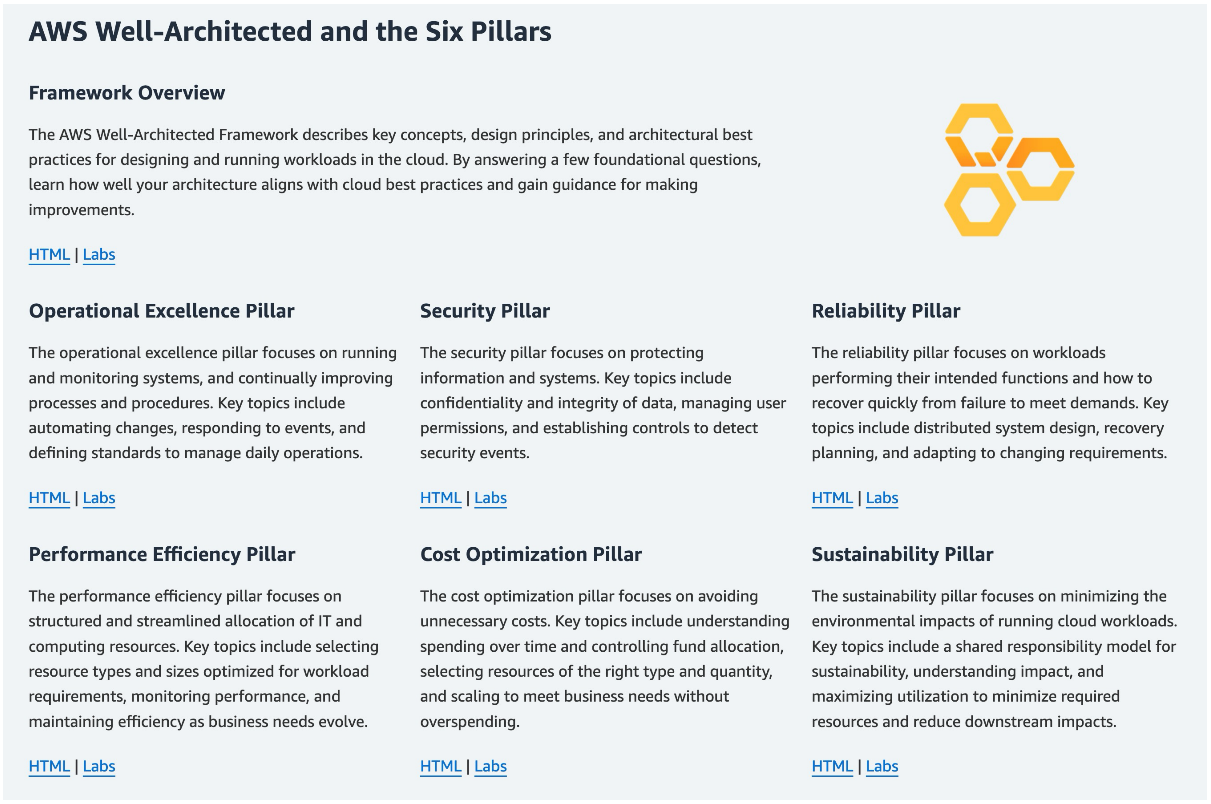


Figure 9-18. Building a repeatable mechanism to evaluate models during model selection



AWS Well-Architected and the Six Pillars

Framework Overview

The AWS Well-Architected Framework describes key concepts, design principles, and architectural best practices for designing and running workloads in the cloud. By answering a few foundational questions, learn how well your architecture aligns with cloud best practices and gain guidance for making improvements.

HTML | Labs

Operational Excellence Pillar

The operational excellence pillar focuses on running and monitoring systems, and continually improving processes and procedures. Key topics include automating changes, responding to events, and defining standards to manage daily operations.

HTML | Labs

Security Pillar

The security pillar focuses on protecting information and systems. Key topics include confidentiality and integrity of data, managing user permissions, and establishing controls to detect security events.

HTML | Labs

Reliability Pillar

The reliability pillar focuses on workloads performing their intended functions and how to recover quickly from failure to meet demands. Key topics include distributed system design, recovery planning, and adapting to changing requirements.

HTML | Labs

Performance Efficiency Pillar

The performance efficiency pillar focuses on structured and streamlined allocation of IT and computing resources. Key topics include selecting resource types and sizes optimized for workload requirements, monitoring performance, and maintaining efficiency as business needs evolve.

HTML | Labs

Cost Optimization Pillar

The cost optimization pillar focuses on avoiding unnecessary costs. Key topics include understanding spending over time and controlling fund allocation, selecting resources of the right type and quantity, and scaling to meet business needs without overspending.

HTML | Labs

Sustainability Pillar

The sustainability pillar focuses on minimizing the environmental impacts of running cloud workloads. Key topics include a shared responsibility model for sustainability, understanding impact, and maximizing utilization to minimize required resources and reduce downstream impacts.

HTML | Labs

BART vs BERT

Feature	**BART**	**BERT**
	Bidirectional and Auto-Regressive Transformers	Bidirectional Encoder Representations from Transformers
Architecture	Encoder-Decoder	Encoder-Only
Primary Use	**Text generation, such as summarization and translation**	**Text understanding, such as classification and entity recognition**
Pretraining Task	Noising and reconstructing text	Masked language model and next sentence prediction
Encoding	Bidirectional encoding of corrupted text	Bidirectional encoding of unaltered text
Decoding	Autoregressive decoding to reconstruct or generate text	Not applicable, as BERT does not generate text
Model Output	Capable of generating new text based on input	Generates representations of input text for classification or other tasks
Use Cases	Content creation, translation, summarization	Sentiment analysis, question answering, named entity recognition
Training Approach	Denoising autoencoder: learns by correcting intentionally corrupted text	Learns by predicting randomly masked words in sentences and predicting next sentences
Interactivity	Generates output text interactively, one token at a time	Analyzes input text as a whole to provide embeddings or perform specific tasks

This comparison highlights the fundamental differences in architecture and application between BART and BERT. While BART is designed for tasks that involve both understanding and generating text, BERT focuses on understanding and interpreting text, making it ideal for tasks that require insights into language structure and content without the need for generating new text.