Patterns for Building LLM-based Systems & Products

[`llm` `engineering` `production` 🔥] · 66 min read

> Discussions on HackerNews, Twitter, and LinkedIn

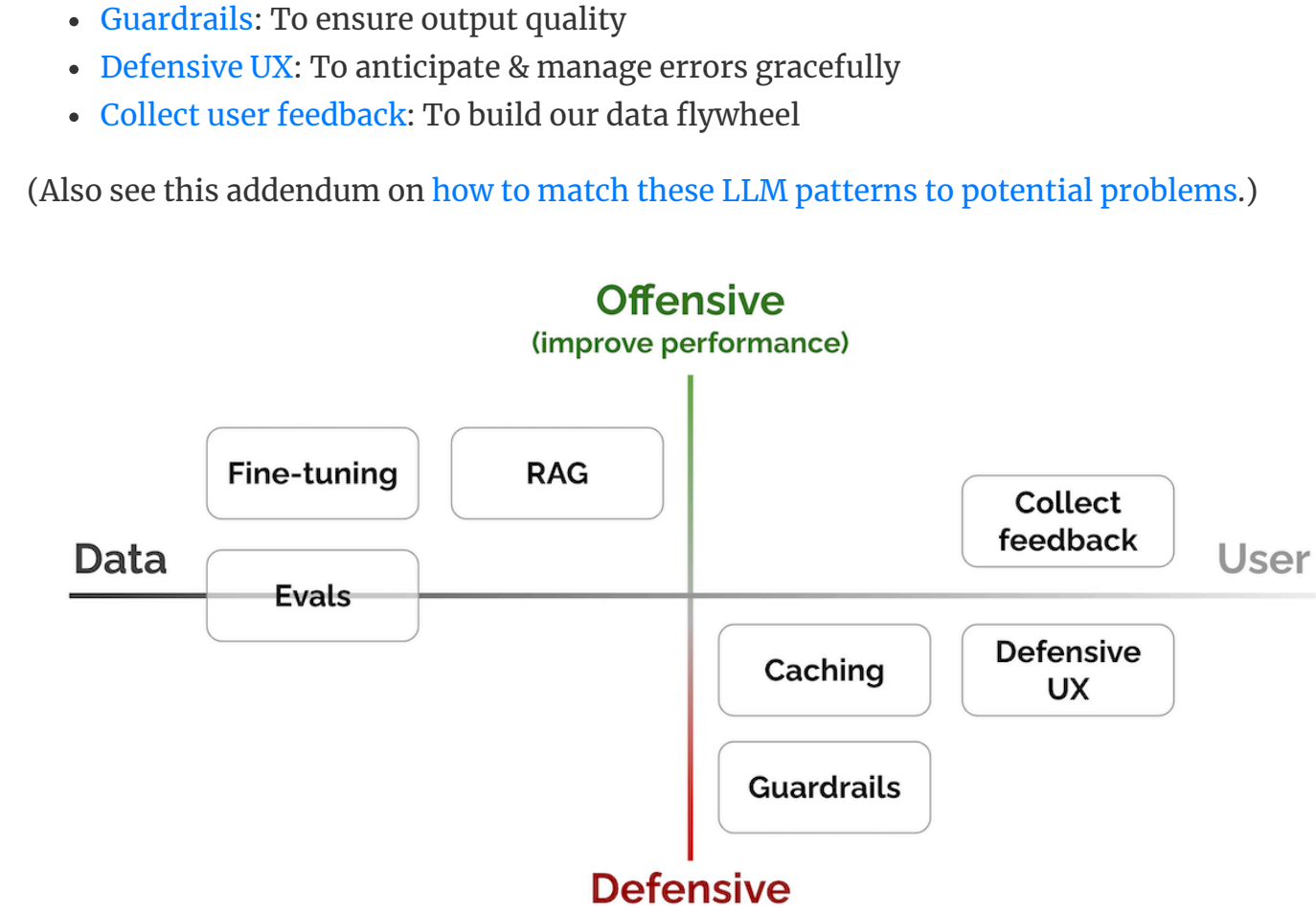
"There is a large class of problems that are easy to imagine and build demos for, but extremely hard to make products out of. For example, self-driving: It's easy to demo a car self-driving around a block, but making it into a product takes a decade." - Karpathy

This write-up is about practical patterns for integrating large language models (LLMs) into systems & products. We'll build on academic research, industry resources, and practitioner know-how, and distill them into key ideas and practices.

There are seven key patterns. They're also organized along the spectrum of improving performance vs. reducing cost/risk, and closer to the data vs. closer to the user.

- Evals: To measure performance
- RAG: To add recent, external knowledge
- Fine-tuning: To get better at specific tasks
- Caching: To reduce latency & cost
- Guardrails: To ensure output quality
- Defensive UX: To anticipate & manage errors gracefully
- Collect user feedback: To build our data flywheel

(Also see this addendum on how to match these LLM patterns to potential problems.)



LLM patterns: From data to user, from defensive to offensive (see connections between patterns)

Evals: To measure performance

Evaluations are a set of measurements used to assess a model's performance on a task. They include benchmark data and metrics. From a HackerNews comment:

> How important evals are to the team is a major differentiator between folks rushing out hot garbage and those seriously building products in the space.

Why evals?

Evals enable us to measure how well our system or product is doing and detect any regressions. (A system or product can be made up of multiple components such as LLMs, prompt templates, retrieved context, and parameters like temperature.) A representative set of evals takes us a step towards measuring system changes at scale. Without evals, we would be flying blind, or would have to visually inspect LLM outputs with each change.

More about evals

There are many benchmarks in the field of language modeling. Some notable ones are:

- **MMLU**: A set of 57 tasks that span elementary math, US history, computer science, law, and more. To perform well, models must possess extensive world knowledge and problem-solving ability.
- **EleutherAI Eval**: Unified framework to test models via zero/few-shot settings on 200 tasks. Incorporates a large number of evals including BigBench, MMLU, etc.
- **HELM**: Instead of specific tasks and metrics, HELM offers a comprehensive assessment of LLMs by evaluating them across domains. Metrics include accuracy, calibration, robustness, fairness, bias, toxicity, etc. Tasks include Q&A, information retrieval, summarization, text classification, etc.
- **AlpacaEval**: Automated evaluation framework which measures how often a strong LLM (e.g., GPT-4) prefers the output of one model over a reference model. Metrics include win rate, bias, latency, price, variance, etc. Validated to have high agreement with 20k human annotations.

We can group metrics into two categories: context-dependent or context-free.

- **Context-dependent**: These take context into account. They're often proposed for a specific task; repurposing them for other tasks will require some adjustment.
- **Context-free**: These aren't tied to the context when evaluating generated output; they only compare the output with the provided gold references. As they're task agnostic, they're easier to apply to a wide variety of tasks.

To get a better sense of these metrics (and their potential shortfalls), we'll explore a few of the commonly used metrics such as BLEU, ROUGE, BERTScore, and MoverScore.

BLEU (Bilingual Evaluation Understudy) is a precision-based metric: It counts the number of n-grams in the generated output that also show up in the reference, and then divides it by the total number of words in the output. It's predominantly used in machine translation and remains a popular metric due to its cost-effectiveness.

First, precision for various values of n is computed:

$$\text{precision}_n = \frac{\sum_{p \in \text{output}} \sum_{\text{n-gram} \in p} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{p \in \text{output}} \sum_{\text{n-gram} \in p} \text{Count}(\text{n-gram})}$$

$Count_{clip}(\text{n-gram})$ is clipped by the maximum number of times an n-gram appears in any corresponding reference sentence.

$$\text{Count}_{\text{clip}}(n\text{-gram}) = \min\left(\text{matched } n\text{-gram count}, \max_{r \in R}\left(n\text{-gram count in } r\right)\right)$$

Once we've computed precision at various n, a final BLEU-N score is computed as the geometric mean of all the $precision_n$ scores.

However, since precision relies solely on n-grams and doesn't consider the length of the generated output, an output containing just one unigram of a common word (like a stop word) would achieve perfect precision. This can be misleading and encourage outputs that contain fewer words to increase BLEU scores. To counter this, a brevity penalty is added to penalize excessively short sentences.

$$BP = \begin{cases} 1 & \text{if } |p| > |r| \\ e^{1-\frac{|r|}{|p|}} & \text{otherwise} \end{cases}$$

Thus, the final formula is:

$$\text{BLEU-N} = BP \cdot \exp\left(\sum_{n=1}^{N} W_n \log(\text{precision}_n)\right)$$

ROUGE (Recall-Oriented Understudy for Gisting Evaluation): In contrast to BLEU, ROUGE is recall-oriented. It counts the number of words in the reference that also occur in the output. It's typically used to assess automatic summarization tasks.

There are several ROUGE variants. ROUGE-N is most similar to BLEU in that it also counts the number of matching n-grams between the output and the reference.

$$\text{ROUGE-N} = \frac{\sum_{s_r \in \text{references}} \sum_{n\text{-gram} \in s_r} \text{Count}_{\text{match}}(n\text{-gram})}{\sum_{s_r \in \text{references}} \sum_{n\text{-gram} \in s_r} \text{Count}(n\text{-gram})}$$

Other variants include:

- ROUGE-L: This measures the longest common subsequence (LCS) between the output and the reference. It considers sentence-level structure similarity and zeros in on the longest series of co-occurring in-sequence n-grams.
- ROUGE-S: This measures the skip-bigram between the output and reference. Skip-bigrams are pairs of words that maintain their sentence order regardless of the words that might be sandwiched between them.

BERTScore is an embedding-based metric that uses cosine similarity to compare each token or n-gram in the generated output with the reference sentence. There are three components to BERTScore:

- Recall: Average cosine similarity between each token in the reference and its closest match in the generated output.
- Precision: Average cosine similarity between each token in the generated output and its nearest match in the reference.
- F1: Harmonic mean of recall and precision

$$Recall_{\text{BERT}} = \frac{1}{|r|} \sum_{i \in r} \max_{j \in p} \vec{i}^T \vec{j}, \quad Precision_{\text{BERT}} = \frac{1}{|p|} \sum_{j \in p} \max_{i \in r} \vec{i}^T \vec{j}$$

$$\text{BERTscore} = F_{\text{BERT}} = \frac{2 \cdot P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}$$

BERTScore is useful because it can account for synonyms and paraphrasing. Simpler metrics like BLEU and ROUGE can't do this due to their reliance on exact matches. BERTScore has been shown to have better correlation for tasks such as image captioning and machine translation.

MoverScore also uses contextualized embeddings to compute the distance between tokens in the generated output and reference. But unlike BERTScore, which is based on one-to-one matching (or "hard alignment") of tokens, MoverScore allows for many-to-one matching (or "soft alignment").

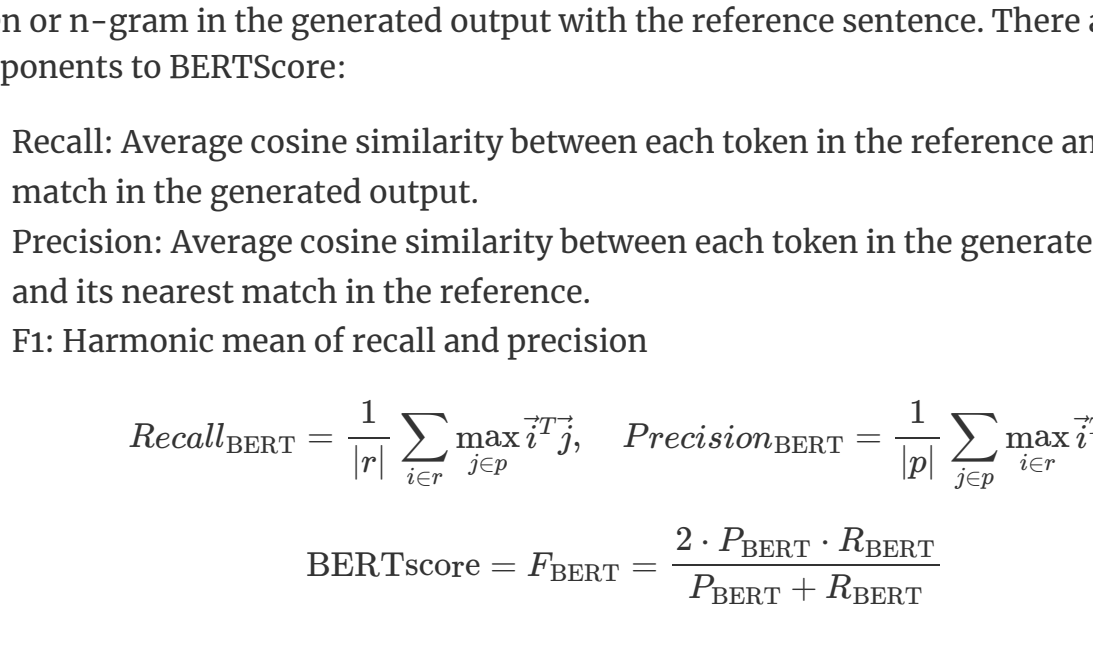

BERTScore (precision/recall)
MoverScore(unigram)
guy
sitting
man
standing
wearing
jacket
canoe
boat
red
lifevest
$\mathcal{V}$
Word Embeddings
System x: A guy with a red jacket is standing on a boat
Ref y: A man wearing a lifevest is sitting in a canoe

Figure 1: An illustration of MoverScore and BERTScore.

BERTScore (left) vs. MoverScore (right; source)

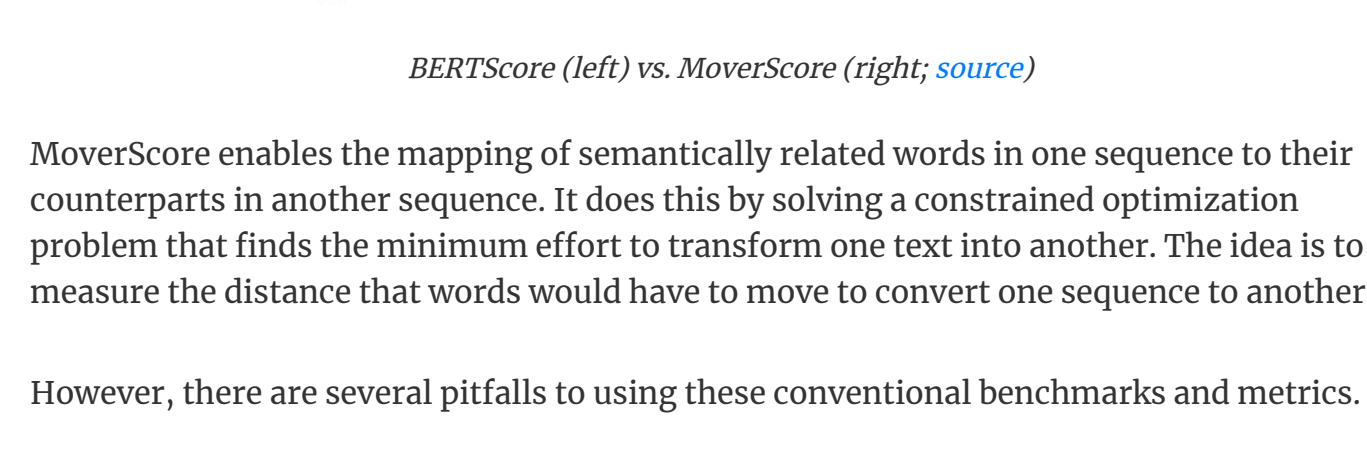
MoverScore enables the mapping of semantically related words in one sequence to their counterparts in another sequence. It does this by solving a constrained optimization problem that finds the minimum effort to transform one text into another. The idea is to measure the distance that words would have to move to convert one sequence to another.

However, there are several pitfalls to using these conventional benchmarks and metrics.

First, there's **poor correlation between these metrics and human judgments.** BLEU, ROUGE, and others have had negative correlation with how humans evaluate fluency. They also showed moderate to less correlation with human adequacy scores. In particular, BLEU and ROUGE have low correlation with tasks that require creativity and diversity.

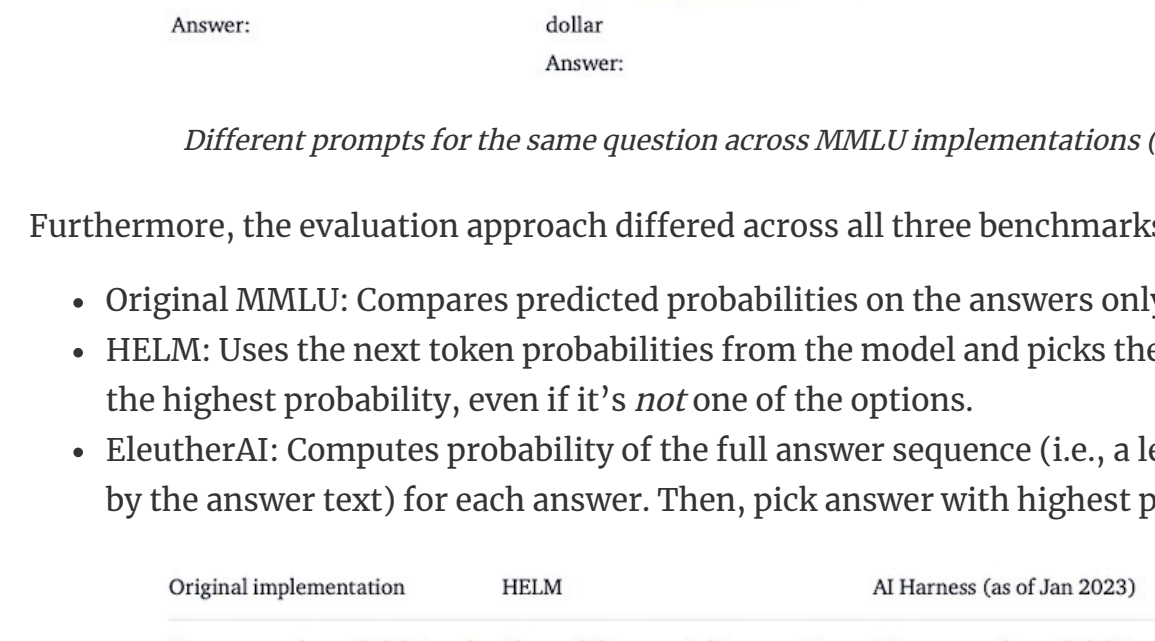
Second, these metrics often have **poor adaptability to a wider variety of tasks**. Adopting a metric proposed for one task to another is not always prudent. For example, exact match metrics such as BLEU and ROUGE are a poor fit for tasks like abstractive summarization or dialogue. Since they're based on n-gram overlap between output and reference, they don't make sense for a dialogue task where a wide variety of responses are possible. An output can have zero n-gram overlap with the reference but yet be a good response.

Third, these metrics have **poor reproducibility**. Even for the same metric, high variance is reported across different studies, possibly due to variations in human judgment collection or metric parameter settings. Another study of ROUGE scores across 2,000 studies found that scores were hard to reproduce, difficult to compare, and often incorrect because evals were often conducted with untested, incorrect ROUGE implementations.



Dimensions of model evaluations with ROUGE (source)

And even with recent benchmarks such as MMLU, **the same model can get significantly different scores based on the eval implementation**. Huggingface compared the original MMLU implementation with the HELM and EleutherAI implementations and found that the same example could have different prompts across various providers.

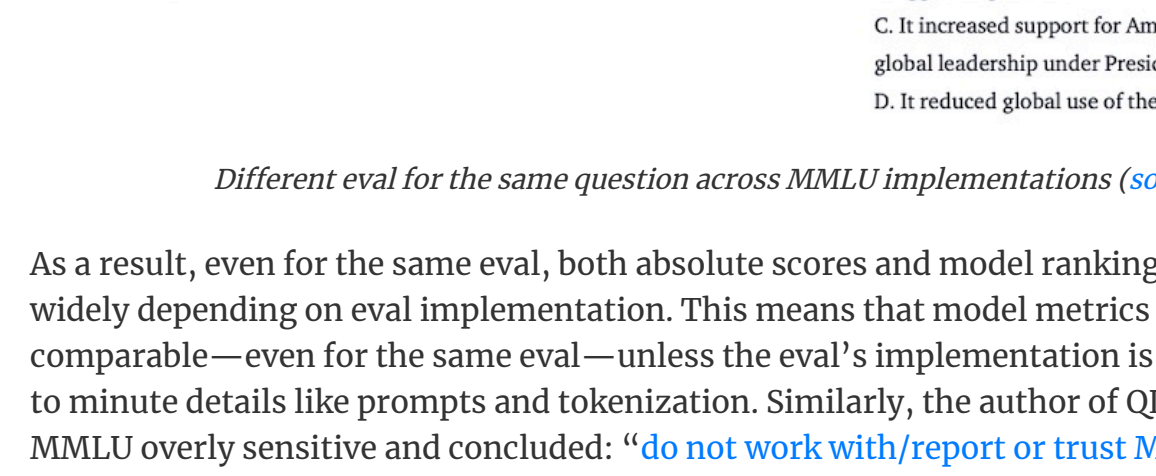


Original implementation Ollmer PR	HELM commit cab5d89	AI Harness commit e47e01b
The following are multiple choice questions (with answers) about us foreign policy. How did the 2008 financial crisis affect America's international reputation? A. It damaged support for the US model of political economy and capitalism B. It created anger at the United States for exaggerating the crisis C. It increased support for American global leadership under President Obama D. It reduced global use of the US dollar Answer:	The following are multiple choice questions (with answers) about us foreign policy. Question: How did the 2008 financial crisis affect America's international reputation? A. It damaged support for the US model of political economy and capitalism B. It created anger at the United States for exaggerating the crisis C. It increased support for American global leadership under President Obama D. It reduced global use of the US dollar Answer:	Question: How did the 2008 financial crisis affect America's international reputation? Choices: A. It damaged support for the US model of political economy and capitalism B. It created anger at the United States for exaggerating the crisis C. It increased support for American global leadership under President Obama D. It reduced global use of the US dollar Answer:

Different prompts for the same question across MMLU implementations (source)

Furthermore, the evaluation approach differed across all three benchmarks:

- Original MMLU: Compares predicted probabilities on the answers only (A, B, C, D)
- HELM: Uses the next token probabilities from the model and picks the token with the highest probability, even if it's *not* one of the options.
- EleutherAI: Computes probability of the full answer sequence (i.e., a letter followed by the answer text) for each answer. Then, pick answer with highest probability.



Original implementation	HELM	AI Harness (as of Jan 2023)
We compare the probabilities of the following letter answers:	The model is expected to generate as text the following letter answer:	We compare the probabilities of the following full answers:
A B C D	A	A. It damaged support for the US model of political economy and capitalism B. It created anger at the United States for exaggerating the crisis C. It increased support for American global leadership under President Obama D. It reduced global use of the US dollar

Different eval for the same question across MMLU implementations (source)

As a result, even for the same eval, both absolute scores and model ranking can fluctuate widely depending on eval implementation. This means that model metrics aren't truly comparable—even for the same eval—unless the eval's implementation is identical down to minute details like prompts and tokenization. Similarly, the author of QLoRA found MMLU overly sensitive and concluded: "do not work with/report or trust MMLU scores".

Beyond conventional evals such as those mentioned above, **an emerging trend is to use a strong LLM as a reference-free metric** to evaluate generations from other LLMs. This means we may not need human judgments or gold references for evaluation.

G-Eval is a framework that applies LLMs with Chain-of-Though (CoT) and a form-filling paradigm to **evaluate LLM outputs**. First, they provide a task introduction and evaluation criteria to an LLM and ask it to generate a CoT of evaluation steps. Then, to evaluate coherence in news summarization, they concatenate the prompt, CoT, news article, and summary and ask the LLM to output a score between 1 to 5. Finally, they use the probabilities of the output tokens from the LLM to normalize the score and take their weighted summation as the final result.

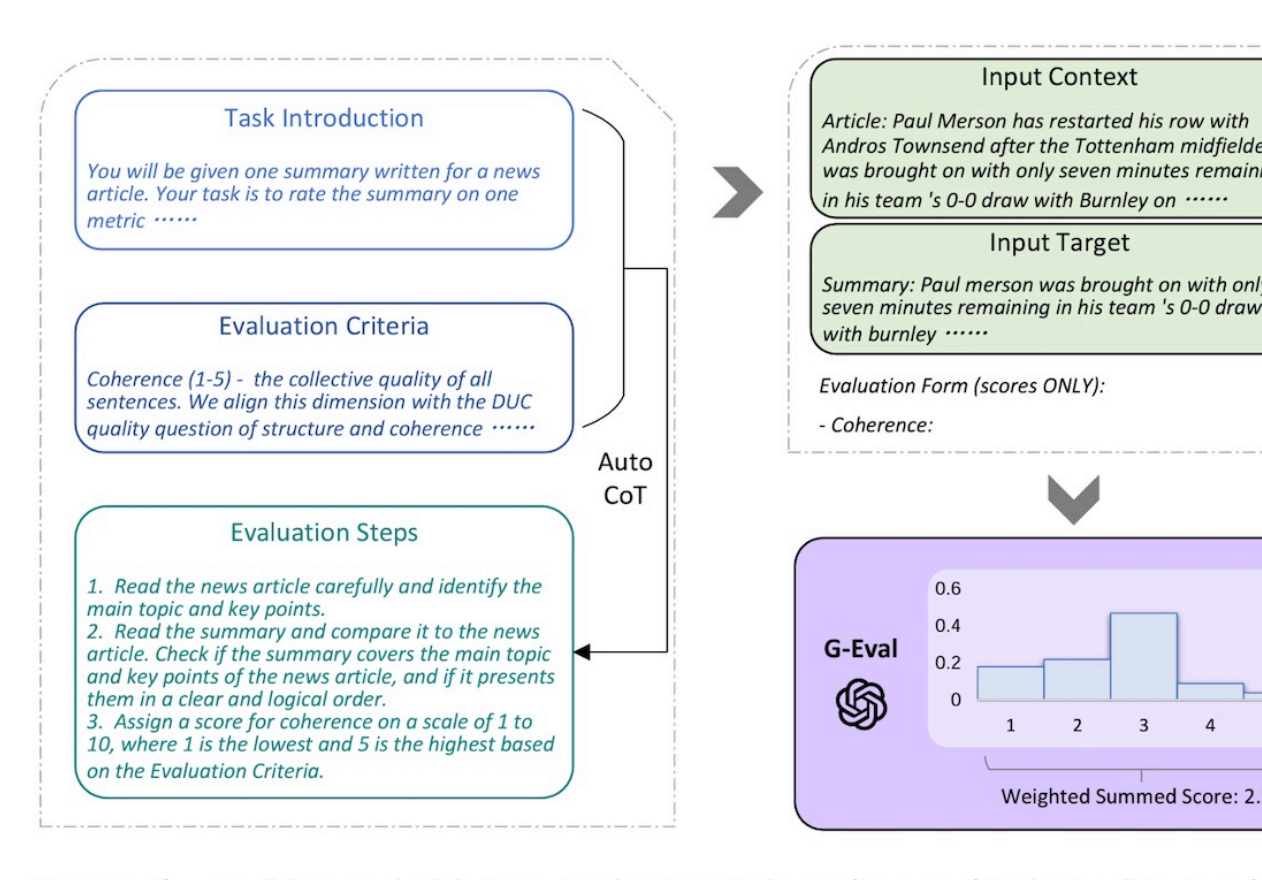

Task Introduction
You will be given one summary written for a news article. Your task is to rate the summary on one metric ······
Evaluation Criteria
Coherence (1-5) - the collective quality of all sentences. We align this dimension with the DUC quality question of structure and coherence ······
Auto CoT
Evaluation Steps
1. Read the news article carefully and identify the main topic and key points.
2. Read the summary and compare it to the news article. Check if the summary covers the main topic and key points of the news article, and if it presents them in a clear and logical order.
3. Assign a score for coherence on a scale of 1 to 10, where 1 is the lowest and 5 is the highest based on the Evaluation Criteria.
Input Context
Article: Paul Merson has restarted his row with Andros Townsend after the Tottenham midfielder was brought on with only seven minutes remaining in his team 's 0-0 draw with Burnley on ······
Input Target
Summary: Paul merson was brought on with only seven minutes remaining in his team 's 0-0 draw with burnley ······
Evaluation Form (scores ONLY):
- Coherence:
G-Eval
0.6
0.4
0.2
0
1
2
3
4
5
Weighted Summed Score: 2.59

Figure 1: The overall framework of G-EVAL. We first input Task Introduction and Evaluation Criteria to the LLM, and ask it to generate a CoT of detailed Evaluation Steps. Then we use the prompt along with the generated CoT to evaluate the NLG outputs in a form-filling paradigm. Finally, we use the probability-weighted summation of the output scores as the final score.

Overview of G-Eval (source)

They found that GPT-4 as an evaluator had a high Spearman correlation with human judgments (0.514), outperforming all previous methods. It also outperformed traditional metrics on aspects such as coherence, consistency, fluency, and relevance. On topical chat, it did better than traditional metrics such as ROUGE-L, BLEU-4, and BERTScore across several criteria such as naturalness, coherence, engagingness, and groundedness.

The Vicuna paper adopted a similar approach. They start by defining eight categories (writing, roleplay, extraction, reasoning, math, coding, STEM, and humanities/social science) before developing 10 questions for each category. Next, they generated answers from five chatbots: LLaMA, Alpaca, ChatGPT, Bard, and Vicuna. Finally, they asked GPT-4 to rate the quality of the answers based on helpfulness, relevance, accuracy, and detail.

Overall, they found that GPT-4 not only provided consistent scores but could also give detailed explanations for those scores. Under the single answer grading paradigm, GPT-4 had higher agreement with humans (85%) than the humans had amongst themselves (81%). This suggests that GPT-4's judgment aligns closely with the human evaluators.

QLoRA also used an LLM to evaluate another LLM's output. They asked GPT-4 to rate the performance of various models against gpt-3.5-turbo on the Vicuna benchmark. Given the responses from gpt-3.5-turbo and another model, GPT-4 was prompted to score both out of 10 and explain its ratings. They also measured performance via direct comparisons between models, simplifying the task to a three-class rating scheme that included ties.

To validate the automated evaluation, they collected human judgments on the Vicuna benchmark. Using Mechanical Turk, they enlisted two annotators for comparisons to gpt-3.5-turbo, and three annotators for pairwise comparisons. They found that human and GPT-4 ranking of models were largely in agreement, with a Spearman rank correlation of 0.55 at the model level. This provides an additional data point suggesting that LLM-based automated evals could be a cost-effective and reasonable alternative to human evals.

How to apply evals?

Building solid evals should be the starting point for any LLM-based system or product (as well as conventional machine learning systems).

Unfortunately, classical metrics such as BLEU and ROUGE don't make sense for more complex tasks such as abstractive summarization or dialogue. Furthermore, we've seen that benchmarks like MMLU (and metrics like ROUGE) are sensitive to how they're implemented and measured. And to be candid, unless your LLM system is studying for a school exam, using MMLU as an eval doesn't quite make sense.

Thus, instead of using off-the-shelf benchmarks, we can **start by collecting a set of task-specific evals** (i.e., prompt, context, expected outputs as references). These evals will then guide prompt engineering, model selection, fine-tuning, and so on. And as we update our systems, we can run these evals to quickly measure improvements or regressions. Think of it as Eval Driven Development (EDD).

In addition to the evaluation dataset, we **also need useful metrics**. They help us distill performance changes into a single number that's comparable across eval runs. And if we can simplify the problem, we can choose metrics that are easier to compute and interpret.

The simplest task is probably classification: If we're using an LLM for classification-like tasks (e.g., toxicity detection, document categorization) or extractive QA without dialogue, we can rely on standard classification metrics such as recall, precision, PRAUC, etc. If our task has no correct answer but we have references (e.g., machine translation, extractive summarization), we can rely on reference metrics based on matching (BLEU, ROUGE) or semantic similarity (BERTScore, MoverScore).

However, these metrics may not work for more open-ended tasks such as abstractive summarization, dialogue, and others. But collecting human judgments can be slow and expensive. Thus, we may opt to lean on **automated evaluations via a strong LLM**.

Relative to human judgments which are typically noisy (due to differing biases among annotators), LLM judgments tend to be less noisy (as the bias is more systematic) but more biased. Nonetheless, since we're aware of these biases, we can mitigate them accordingly:

- Position bias: LLMs tend to favor the response in the first position. To mitigate this, we can evaluate the same pair of responses twice while swapping their order. If the same response is preferred in both orders, we mark it as a win; else, it's a tie.
- Verbosity bias: LLMs tend to favor longer, wordier responses over more concise ones, even if the latter is clearer and of higher quality. A possible solution is to ensure that comparison responses are similar in length.
- Self-enhancement bias: LLMs have a slight bias towards their own answers. GPT-4 favors itself with a 10% higher win rate while Claude-v1 favors itself with a 25% higher win rate. To counter this, don't use the same LLM for evaluation tasks.

Another tip: Rather than asking an LLM for a direct evaluation (via giving a score), try giving it a reference and asking for a comparison. This helps with reducing noise.

Finally, sometimes the best eval is human eval aka vibe check. (Not to be confused with the poorly named code evaluation benchmark HumanEval.) As mentioned in the Latent Space podcast with MosaicML (34th minute):

> The vibe-based eval cannot be underrated. … One of our evals was just having a bunch of prompts and watching the answers as the models trained and see if they change. Honestly, I don't really believe that any of these eval metrics capture what we care about. One of our prompts was "suggest games for a 3-year-old and a 7-year-old to play" and that was a lot more valuable to see how the answer changed during the course of training. — Jonathan Frankle

Also see this deep dive into evals for abstractive summarization. It covers reference, context, and preference-based metrics, and also discusses hallucination detection.

Retrieval-Augmented Generation: To add knowledge

Retrieval-Augmented Generation (RAG) fetches relevant data from outside the foundation model and enhances the input with this data, providing richer context to improve output.

Why RAG?

RAG helps reduce hallucination by grounding the model on the retrieved context, thus increasing factuality. In addition, it's cheaper to keep retrieval indices up-to-date than to continuously pre-train an LLM. This cost efficiency makes it easier to provide LLMs with access to recent data via RAG. Finally, if we need to update or remove data such as biased or toxic documents, it's more straightforward to update the retrieval index (compared to fine-tuning or prompting an LLM not to generate toxic outputs).

In short, RAG applies mature and simpler ideas from the field of information retrieval to support LLM generation. In a recent Sequoia survey, 88% of respondents believe that retrieval will be a key component of their stack.

More about RAG

Before diving into RAG, it helps to have a basic understanding of text embeddings. (Feel free to skip this section if you're familiar with the subject.)

A text embedding is a **compressed, abstract representation of text data** where text of arbitrary length can be represented as a fixed-size vector of numbers. It's usually learned from a corpus of text such as Wikipedia. Think of them as a universal encoding for text, where **similar items are close to each other while dissimilar items are farther apart**.

A good embedding is one that does well on a downstream task, such as retrieving similar items. Huggingface's Massive Text Embedding Benchmark (MTEB) scores various models on diverse tasks such as classification, clustering, retrieval, summarization, etc.

Quick note: While we mainly discuss text embeddings here, embeddings can take many modalities. For example, CLIP is multimodal and embeds images and text in the same space, allowing us to find images most similar to an input text. We can also embed products based on user behavior (e.g., clicks, purchases) or graph relationships.

RAG has its roots in open-domain Q&A. An early Meta paper showed that retrieving relevant documents via TF-IDF and providing them as context to a language model (BERT) improved performance on an open-domain QA task. They converted each task into a cloze statement and queried the language model for the missing token.

Following that, **Dense Passage Retrieval (DPR)** showed that using dense embeddings (instead of a sparse vector space such as TF-IDF) for document retrieval can outperform strong baselines like Lucene BM25 (65.2% vs. 42.9% for top-5 accuracy.) They also showed that higher retrieval precision translates to higher end-to-end QA accuracy, highlighting the importance of upstream retrieval.

To learn the DPR embedding, they fine-tuned two independent BERT-based encoders on existing question-answer pairs. The passage encoder (E_p) embeds text passages into vectors while the query encoder (E_q) embeds questions into vectors. The query embedding is then used to retrieve k passages that are most similar to the question.

They trained the encoders so that the dot-product similarity makes a good ranking function, and optimized the loss function as the negative log-likelihood of the positive passage. The DPR embeddings are optimized for maximum inner product between the question and relevant passage vectors. The goal is to learn a vector space such that pairs of questions and their relevant passages are close together.

For inference, they embed all passages (via E_p) and index them in FAISS offline. Then, given a question at query time, they compute the question embedding (via E_q), retrieve the top k passages via approximate nearest neighbors, and provide it to the language model (BERT) that outputs the answer to the question.

Retrieval Augmented Generation (RAG), from which this pattern gets its name, highlighted the downsides of pre-trained LLMs. These include not being able to expand or revise memory, not providing insights into generated output, and hallucinations.

To address these downsides, they introduced RAG (aka semi-parametric models). Dense vector retrieval serves as the non-parametric component while a pre-trained LLM acts as the parametric component. They reused the DPR encoders to initialize the retriever and build the document index. For the LLM, they used BART, a 400M parameter seq2seq model.

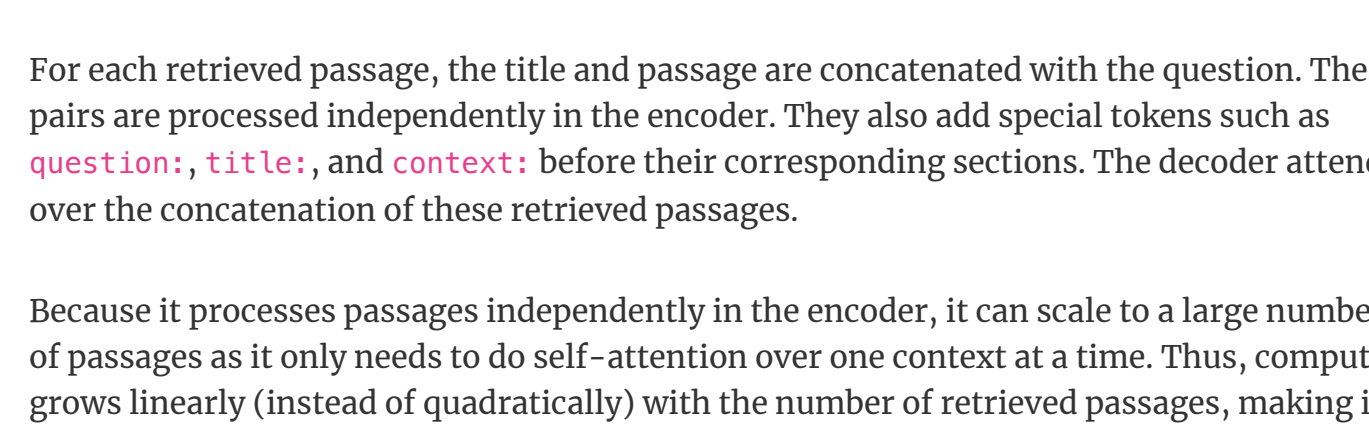

Define "middle ear" (x)
Question Answering: Question Query
Barack Obama was born in Hawaii. (x)
Fact Verification: Fact Query
The Divine Comedy (x)
Jeopardy Question Generation: Answer Query
End-to-End Backprop through q and pθ
Query Encoder
q
q(x)
Retriever pη
(Non-Parametric)
MIPS
Document Index
d(z)
z4
z3
z2
z1
Generator pθ
(Parametric)
pθ
Margin-alize
The middle ear includes the tympanic cavity and the three ossicles. (y)
Question Answering: Answer Generation
supports (y)
Fact Verification: Label Generation
This 14th century work is divided into 3 sections: "Inferno", "Purgatorio" & "Paradiso" (y)
Question Generation

Figure 1: Overview of our approach. We combine a pre-trained retriever (*Query Encoder* + *Document Index*) with a pre-trained seq2seq model (*Generator*) and fine-tune end-to-end. For query x, we use Maximum Inner Product Search (MIPS) to find the top-K documents z_i. For final prediction y, we treat z as a latent variable and marginalize over seq2seq predictions given different documents.

Overview of Retrieval Augmented Generation (source)

During inference, they concatenate the input with the retrieved document. Then, the LLM generates token_i based on the original input, the retrieved document, and the previous $i-1$ tokens. For generation, they proposed two approaches that vary in how the retrieved passages are used to generate output.

In the first approach, RAG-Sequence, the model uses the same document to generate the complete sequence. Thus, for k retrieved documents, the generator produces an output for each document. Then, the probability of each output sequence is marginalized (sum the probability of each output sequence in k and weigh it by the probability of each document being retrieved). Finally, the output sequence with the highest probability is selected.

On the other hand, RAG-Token can generate each token based on a *different* document. Given k retrieved documents, the generator produces a distribution for the next output token for each document before marginalizing (aggregating all the individual token distributions.). The process is then repeated for the next token. This means that, for each token generation, it can retrieve a different set of k relevant documents based on the original input *and* previously generated tokens. Thus, documents can have different retrieval probabilities and contribute differently to the next generated token.

Fusion-in-Decoder (FiD) also uses retrieval with generative models for open-domain QA. It supports two methods for retrieval, BM25 (Lucene with default parameters) and DPR. FiD is named for how it performs fusion on the retrieved documents in the decoder only.

Overview of Fusion-in-Decoder (source)

For each retrieved passage, the title and passage are concatenated with the question. These pairs are processed independently in the encoder. They also add special tokens such as `question:`, `title:`, and `context:` before their corresponding sections. The decoder attends over the concatenation of these retrieved passages.

Because it processes passages independently in the encoder, it can scale to a large number of passages as it only needs to do self-attention over one context at a time. Thus, compute grows linearly (instead of quadratically) with the number of retrieved passages, making it more scalable than alternatives such as RAG-Token. Then, during decoding, the decoder processes the encoded passages jointly, allowing it to better aggregate context across multiple retrieved passages.

Retrieval-Enhanced Transformer (RETRO) adopts a similar pattern where it combines a frozen BERT retriever, a differentiable encoder, and chunked cross-attention to generate output. What's different is that RETRO does retrieval throughout the entire pre-training stage, and not just during inference. Furthermore, they fetch relevant documents based on chunks of the input. This allows for finer-grained, repeated retrieval during generation instead of only retrieving once per query.

For each input chunk (C_u), the k retrieved chunks $RET(C_u)$ are fed into an encoder. The output is the encoded neighbors E_u^j where $E_u^j = \text{Encoder}(\text{RET}(C_u)^j, H_u) \in \mathbb{R}^{r \times d_0}$. Here, each chunk encoding is conditioned on H_u (the intermediate activations) and the activations of chunk C_u through cross-attention layers. In short, the encoding of the retrieved chunks depends on the attended activation of the input chunk. E_u^j is then used to condition the generation of the next chunk.

Overview of RETRO (source)

During retrieval, RETRO splits the input sequence into chunks of 64 tokens. Then, it finds text similar to the *previous* chunk to provide context to the *current* chunk. The retrieval index consists of two contiguous chunks of tokens, N and F. The former is the neighbor chunk (64 tokens) which is used to compute the key while the latter is the continuation chunk (64 tokens) in the original document.

Retrieval is based on approximate k-nearest neighbors via L_2 distance (euclidean) on BERT embeddings. (Interesting departure from the usual cosine or dot product similarity.) The retrieval index, built on SCaNN, can query a 2T token database in 10ms.

They also demonstrated how to RETRO-fit existing baseline models. By freezing the pre-trained weights and only training the chunked cross-attention and neighbor encoder parameters (< 10% of weights for a 7B model), they can enhance transformers with retrieval while only requiring 6M training sequences (3% of pre-training sequences). RETRO-fitted models were able to surpass the performance of baseline models and achieve performance close to that of RETRO trained from scratch.

Performance from RETRO-fitting a pre-trained model (source)

Internet-augmented LMs proposes using a humble "off-the-shelf" search engine to augment LLMs. First, they retrieve a set of relevant documents via Google Search. Since these retrieved documents tend to be long (average length 2,056 words), they chunk them into paragraphs of six sentences each. Finally, they embed the question and paragraphs via TF-IDF and applied cosine similarity to rank the most relevant paragraphs for each query.

Overview of internet-augmented LLMs (source)

The retrieved paragraphs are used to condition the LLM via few-shot prompting. They adopt the conventional k-shot prompting ($k = 15$) from closed-book QA (only providing question-answer pairs) and extend it with an evidence paragraph, such that each context is an evidence, question, and answer triplet.

For the generator, they used Gopher, a 280B parameter model trained on 300B tokens. For each question, they generated four candidate answers based on each of the 50 retrieved paragraphs. Finally, they select the best answer by estimating the answer probability via several methods including direct inference, RAG, noisy channel inference, and Product-of-Experts (PoE). PoE consistently performed the best.

RAG has also been **applied to non-QA tasks such as code generation**. While CodeT5+ can be used as a standalone generator, when combined with RAG, it significantly outperforms similar models in code generation.

To assess the impact of RAG on code generation, they evaluate the model in three settings:

- Retrieval-based: Fetch the top-1 code sample as the prediction
- Generative-only: Output code based on the decoder only
- Retrieval-augmented: Append top-1 code sample to encoder input before code generation via the decoder.

Overview of RAG for CodeT5+ (source)

As a qualitative example, they showed that retrieved code provides crucial context (e.g., use `urllib3` for an HTTP request) and guides the generative process towards more correct predictions. In contrast, the generative-only approach returns incorrect output that only captures the concepts of "download" and "compress".

What if we don't have relevance judgments for query-passage pairs? Without them, we would not be able to train the bi-encoders that embed the queries and documents in the same embedding space where relevance is represented by the inner product. **Hypothetical document embeddings (HyDE)** suggests a solution.

Overview of HyDE (source)

Given a query, HyDE first prompts an LLM, such as InstructGPT, to generate a hypothetical document. Then, an unsupervised encoder, such as Contriver, encodes the document into an embedding vector. Finally, the inner product is computed between the *hypothetical* document and the corpus, and the most similar *real* documents are retrieved.

The expectation is that the encoder's dense bottleneck serves as a lossy compressor and the extraneous, non-factual details are excluded via the embedding. This reframes the relevance modeling problem from a representation learning task to a generation task.

How to apply RAG

From experience with Obsidian-Copilot, I've found that hybrid retrieval (traditional search index + embedding-based search) works better than either alone. There, I complemented classical retrieval (BM25 via OpenSearch) with semantic search (`e5-small-v2`).

Why not embedding-based search only? While it's great in many instances, there are situations where it falls short, such as:

- Searching for a person or object's name (e.g., Eugene, Kaptir 2.0)
- Searching for an acronym or phrase (e.g., RAG, RLHF)
- Searching for an ID (e.g., `gpt-3.5-turbo`, `titan-xlarge-v1.01`)

But keyword search has its limitations too. It only models simple word frequencies and doesn't capture semantic or correlation information. Thus, it doesn't deal well with synonyms or hypernyms (i.e., words that represent a generalization). This is where combining it with semantic search is complementary.

In addition, with a conventional search index, we can use metadata to refine results. For example, we can use date filters to prioritize newer documents or narrow our search to a specific time period. And if the search is related to e-commerce, filters on average rating or categories are helpful. Finally, having metadata is handy for downstream ranking, such as prioritizing documents that are cited more, or boosting products by their sales volume.

With regard to embeddings, the seemingly popular approach is to use `text-embedding-ada-002`. Its benefits include ease of use via an API and not having to maintain our own embedding infra or self-host embedding models. Nonetheless, personal experience and anecdotes from others suggest there are better alternatives for retrieval.

The OG embedding approaches include Word2vec and fastText. FastText is an open-source, lightweight library that enables users to leverage pre-trained embeddings or train new embedding models. It comes with pre-trained embeddings for 157 languages and is extremely fast, even without a GPU. It's my go-to for early-stage proof of concepts.

Another good baseline is sentence-transformers. It makes it simple to compute embeddings for sentences, paragraphs, and even images. It's based on workhorse transformers such as BERT and RoBERTa and is available in more than 100 languages.

More recently, instructor models have shown SOTA performance. During training, these models prepend the task description to the text. Then, when embedding new text, we simply have to describe the task to get task-specific embeddings. (Not that different from instruction tuning for embedding models IMHO.)

An example is the E5 family of models. For open QA and information retrieval, we simply prepend documents in the index with `passage:`, and prepend queries with `query:`. If the task is symmetric (e.g., semantic similarity, paraphrase retrieval) or if we want to use embeddings as features (e.g., classification, clustering), we just use the `query:` prefix.

The Instructor model takes it a step further, allowing users to customize the prepended prompt: "Represent the `domain` `task_type` for the `task_objective`:" For example, "Represent the Wikipedia document for retrieval:". (The domain and task objective are optional). This brings the concept of prompt tuning into the field of text embedding.

Finally, as of Aug 1st, the top embedding model on the MTEB Leaderboard is the GTE family of models by Alibaba DAMO Academy. The top performing model's size is half of the next best model `e5-large-v2` (0.67GB vs 1.34GB). In 2nd position is `gte-base` with a model size of only 0.22GB and embedding dimension of 768. (H/T Nirant.)

To retrieve documents with low latency at scale, we use approximate nearest neighbors (ANN). It optimizes for retrieval speed and returns the approximate (instead of exact) top k most similar neighbors, trading off a little accuracy loss for a large speed up.

ANN embedding indices are data structures that let us do ANN searches efficiently. At a high level, they build partitions over the embedding space so we can quickly zoom in on the specific space where the query vector is. Some popular techniques include:

- Locality Sensitive Hashing (LSH): The core idea is to create hash functions so that similar items are likely to end up in the same hash bucket. By only needing to check the relevant buckets, we can perform ANN queries efficiently.
- Facebook AI Similarity Search (FAISS): It uses a combination of quantization and indexing for efficient retrieval, supports both CPU and GPU, and can handle billions of vectors due to its efficient use of memory.
- Hierarchical Navigable Small Worlds (HNSW): Inspired by "six degrees of separation", it builds a hierarchical graph structure that embodies the small world phenomenon. Here, most nodes can be reached from any other node via a minimum number of hops. This structure allows HNSW to initiate queries from broader, coarser approximations and progressively narrow the search at lower levels.
- Scalable Nearest Neighbors (ScaNN): It has a two-step process. First, coarse quantization reduces the search space. Then, fine-grained search is done within the reduced set. Best recall/latency trade-off I've seen.

When evaluating an ANN index, some factors to consider include:

- Recall: How does it fare against exact nearest neighbors?
- Latency/throughput: How many queries can it handle per second?
- Memory footprint: How much RAM is required to serve an index?
- Ease of adding new items: Can new items be added without having to reindex all documents (LSH) or does the index need to be rebuilt (ScaNN)?

No single framework is better than all others in every aspect. Thus, start by defining your functional and non-functional requirements before benchmarking. Personally, I've found ScaNN to be outstanding in the recall-latency trade-off (see benchmark graph here).

Fine-tuning: To get better at specific tasks

Fine-tuning is the process of taking a pre-trained model (that has already been trained with a vast amount of data) and further refining it on a specific task. The intent is to harness the knowledge that the model has already acquired during its pre-training and apply it to a specific task, usually involving a smaller, task-specific, dataset.

The term "fine-tuning" is used loosely and can refer to several concepts such as:

- **Continued pre-training**: With domain-specific data, apply the same pre-training regime (next token prediction, masked language modeling) on the base model.
- **Instruction fine-tuning**: The pre-trained (base) model is fine-tuned on examples of instruction-output pairs to follow instructions, answer questions, be waifu, etc.
- **Single-task fine-tuning**: The pre-trained model is honed for a narrow and specific task such as toxicity detection or summarization, similar to BERT and T5.
- **Reinforcement learning with human feedback (RLHF)**: This combines instruction fine-tuning with reinforcement learning. It requires collecting human preferences (e.g., pairwise comparisons) which are then used to train a reward model. The reward model is then used to further fine-tune the instructed LLM via RL techniques such as proximal policy optimization (PPO).

We'll mainly focus on single-task and instruction fine-tuning here.

Why fine-tuning?

Fine-tuning an open LLM is becoming an increasingly viable alternative to using a 3rd-party, cloud-based LLM for several reasons.

Performance & control: Fine-tuning can improve the performance of an off-the-shelf base model, and may even surpass a 3rd-party LLM. It also provides greater control over LLM behavior, resulting in a more robust system or product. Overall, fine-tuning enables us to build products that are differentiated from simply using 3rd-party or open LLMs.

Modularization: Single-task fine-tuning lets us to use an army of smaller models that each specialize on their own tasks. Via this setup, a system can be modularized into individual models for tasks like content moderation, extraction, summarization, etc. Also, given that each model only has to focus on a narrow set of tasks, we can get around the alignment tax, where fine-tuning a model on one task reduces performance on other tasks.

Reduced dependencies: By fine-tuning and hosting our own models, we can reduce legal concerns about proprietary data (e.g., PII, internal documents and code) being exposed to external APIs. It also gets around constraints that come with 3rd-party LLMs such as rate-limiting, high costs, or overly restrictive safety filters. By fine-tuning and hosting our own LLMs, we can ensure data doesn't leave our network, and can scale throughput as needed.

More about fine-tuning

Why do we need to fine-tune a *base* model? At the risk of oversimplifying, base models are primarily optimized to predict the next word based on the corpus they're trained on. Hence, they aren't naturally adept at following instructions or answering questions. When posed a question, they tend to respond with more questions. Thus, we perform instruction fine-tuning so they learn to respond appropriately.

However, fine-tuning isn't without its challenges. First, we **need a significant volume of demonstration data**. For instance, in the InstructGPT paper, they used 13k instruction-output samples for supervised fine-tuning, 33k output comparisons for reward modeling, and 31k prompts without human labels as input for RLHF.

Furthermore, fine-tuning comes with an alignment tax—the process can lead to **lower performance on certain critical tasks**. (There's no free lunch after all.) The same InstructGPT paper found that RLHF led to performance regressions (relative to the GPT-3 base model) on public NLP tasks like SQuAD, HellaSwag, and WMT 2015 French to English. (A workaround is to have several smaller, specialized models that excel at narrow tasks.)

Fine-tuning is similar to the concept of transfer learning. As defined in Wikipedia: "Transfer learning is a technique in machine learning in which knowledge learned from a task is re-used to boost performance on a related task." Several years ago, transfer learning made it easy for me to apply ResNet models trained on ImageNet to classify fashion products and build image search.

ULMFit is one of the earlier papers to apply transfer learning to text. They established the protocol of self-supervised pre-training (on unlabeled data) followed by fine-tuning (on labeled data). They used AWS-LSTM, an LSTM variant with dropout at various gates.

Overview of ULMFit (source)

During pre-training (next word prediction), the model is trained on wikitext-103 which contains 28.6 Wikipedia articles and 103M words. Then, during target task fine-tuning, the LM is fine-tuned with data from the domain of the specific task. Finally, during classifier fine-tuning, the model is augmented with two additional linear blocks and fine-tuned on the target classification tasks which includes sentiment analysis, question classification, and topic classification.

Since then, the pre-training followed by fine-tuning paradigm has driven much progress in language modeling. **Bidirectional Encoder Representations from Transformers (BERT; encoder only)** was pre-trained on masked language modeling and next sentence prediction on English Wikipedia and BooksCorpus. It was then fine-tuned on task-specific inputs and labels for single-sentence classification, sentence pair classification, single-sentence tagging, and question & answering.

Overview of BERT (source)

Generative Pre-trained Transformers (GPT; decoder only) was first pre-trained on BooksCorpus via next token prediction. This was followed by single-task fine-tuning for tasks such as text classification, textual entailment, similarity, and Q&A. Interestingly, they found that including language modeling as an auxiliary objective helped the model generalize and converge faster during training.

Overview of GPT (source)

Text-to-text Transfer Transformer (T5; encoder-decoder) was pre-trained on the Colossal Clean Crawled Corpus (C4), a cleaned version of the Common Crawl from April 2019. It employed the same denoising objective as BERT, namely masked language modeling. It was then fine-tuned on tasks such as text classification, abstractive summarization, Q&A, and machine translation.

Overview of T5 (source)

But unlike ULMFIt, BERT, and GPT which used different classifier heads for downstream tasks, T5 represented downstream tasks as text-to-text only. For example, a translation task would have input text starting with `Translation English to German:`, while a summarization task might start with `Summarize:` or `TL;DR:`. The prefix essentially became a hyperparameter (first instance of prompt engineering?) This design choice allowed them to use a single fine-tuned model across a variety of downstream tasks.

InstructGPT expanded this idea of single-task fine-tuning to instruction fine-tuning. The base model was GPT-3, pre-trained on internet data including Common Crawl, WebText, Books, and Wikipedia. It then applied supervised fine-tuning on demonstrations of desired behavior (instruction and output). Next, it trained a reward model on the dataset of comparisons. Finally, it optimized the instructed model against the reward model via PPO, with this last stage focusing more on alignment than specific task performance.

Overview of fine-tuning steps in InstructGPT (source)

Next, let's move from fine-tuned models to fine-tuning techniques.

Soft prompt tuning prepends a trainable tensor to the model's input embeddings, essentially creating a soft prompt. Unlike discrete text prompts, soft prompts can be learned via backpropagation, meaning they can be fine-tuned to incorporate signals from any number of labeled examples.

Next, there's **prefix tuning**. Instead of adding a soft prompt to the model input, it prepends trainable parameters to the hidden states of all transformer blocks. During fine-tuning, the LM's original parameters are kept frozen while the prefix parameters are updated.

Overview of prefix-tuning (source)

The paper showed that this achieved performance comparable to full fine-tuning despite requiring updates on just 0.1% of parameters. Moreover, in settings with limited data and involved extrapolation to new topics, it outperformed full fine-tuning. One hypothesis is that training fewer parameters helped reduce overfitting on smaller target datasets.

There's also the **adapter** technique. This method adds fully connected network layers twice to each transformer block, after the attention layer and after the feed-forward network layer. On GLUE, it's able to achieve within 0.4% of the performance of full fine-tuning by just adding 3.6% parameters per task.

Overview of adapters (source)

Low-Rank Adaptation (LoRA) is a technique where adapters are designed to be the product of two low-rank matrices. It was inspired by Aghajanyan et al. which showed that, when adapting to a specific task, pre-trained language models have a low intrinsic dimension and can still learn efficiently despite a random projection into a smaller subspace. Thus, LoRA hypothesized that weight updates during adaption also have low intrinsic rank.

Overview of LoRA (source)

Similar to prefix tuning, they found that LoRA outperformed several baselines including full fine-tuning. Again, the hypothesis is that LoRA, thanks to its reduced rank, provides implicit regularization. In contrast, full fine-tuning, which updates all weights, could be prone to overfitting.

QLoRA builds on the idea of LoRA. But instead of using the full 16-bit model during fine-tuning, it applies a 4-bit quantized model. It introduced several innovations such as 4-bit NormalFloat (to quantize models), double quantization (for additional memory savings), and paged optimizers (that prevent OOM errors by transferring data to CPU RAM when the GPU runs out of memory).

Overview of QLoRA (source)

As a result, QLoRA reduces the average memory requirements for fine-tuning a 65B model from > 780GB memory to a more manageable 48B without degrading runtime or predictive performance compared to a 16-bit fully fine-tuned baseline.

(Fun fact: During a meetup with Tim Dettmers, an author of QLoRA, he quipped that double quantization was "a bit of a silly idea but works perfectly." Hey, if it works, it works.)

How to apply fine-tuning?

The first step is to **collect demonstration data/labels**. These could be for straightforward tasks such as document classification, entity extraction, or summarization, or they could be more complex such as Q&A or dialogue. Some ways to collect this data include:

- **Via experts or crowd-sourced human annotators**: While this is expensive and slow, it usually leads to higher-quality data with good guidelines.
- **Via user feedback**: This can be as simple as asking users to select attributes that describe a product, rating LLM responses with thumbs up or down (e.g., ChatGPT), or logging which images users choose to download (e.g., Midjourney).
- **Query larger open models with permissive licenses**: With prompt engineering, we might be able to elicit reasonable demonstration data from a larger model (Falcon 40B Instruct) that can be used to fine-tune a smaller model.
- **Reuse open-source data**: If your task can be framed as a natural language inference (NLI) task, we could fine-tune a model to perform NLI using MNLI data. Then, we can continue fine-tuning the model on internal data to classify inputs as entailment, neutral, or contradiction.

Note: Some LLM terms prevent users from using their output to develop other models.

- OpenAI Terms of Use (Section 2c, iii): You may not use output from the Services to develop models that compete with OpenAI.
- LLaMA 2 Community License Agreement (Section 1b-v): You will not use the Llama Materials or any output or results of the Llama Materials to improve any other large language model (excluding Llama 2 or derivative works thereof).

The next step is to **define evaluation metrics**. We've discussed this in a previous section.

Then, **select a pre-trained model.** There are several open LLMs with permissive licenses to choose from. Excluding Llama 2 (since it isn't fully commercial use), Falcon-40B is known to be the best-performing model. Nonetheless, I've found it unwieldy to fine-tune and serve in production given how heavy it is.

Instead, I'm inclined to use smaller models like the Falcon-7B. And if we can simplify and frame the task more narrowly, BERT (340M params), RoBERTA (355M params), and BART (406M params) are solid picks for classification and natural language inference tasks. Beyond that, Flan-T5 (770M and 3B variants) is a reliable baseline for translation, abstractive summarization, headline generation, etc.

We may also need to **update the model architecture**, such as when the pre-trained model's architecture doesn't align with the task. For example, we might need to update the classification heads on BERT or T5 to match our task. Tip: If the task is a simple binary classification task, NLI models can work out of the box. Entailment is mapped to positive, contradiction is mapped to negative, while the neural label can indicate uncertainty.

Then, pick a fine-tuning approach. LoRA and QLoRA are good places to start. But if your fine-tuning is more intensive, such as continued pre-training on new domain knowledge, you may find full fine-tuning necessary.

Finally, basic hyperparameter tuning. Generally, most papers focus on learning rate, batch size, and number of epochs (see LoRA, QLoRA). And if we're using LoRA, we might want to tune the rank parameter (though the QLoRA paper found that different rank and alpha led to similar results). Other hyperparameters include input sequence length, loss type (contrastive loss vs. token match), and data ratios (like the mix of pre-training or demonstration data, or the ratio of positive to negative examples, among others).

Caching: To reduce latency and cost

Caching is a technique to store data that has been previously retrieved or computed. This way, future requests for the same data can be served faster. In the space of serving LLM generations, the popularized approach is to cache the LLM response keyed on the embedding of the input request. Then, for each new request, if a semantically similar request is received, we can serve the cached response.

For some practitioners, this sounds like "a disaster waiting to happen." I'm inclined to agree. Thus, I think the key to adopting this pattern is figuring out how to cache safely, instead of solely depending on semantic similarity.

Why caching?

Caching can significantly reduce latency for responses that have been served before. In addition, by eliminating the need to compute a response for the same input again and again, we can reduce the number of LLM requests and thus save cost. Also, there are certain use cases that do not support latency on the order of seconds. Thus, pre-computing and caching may be the only way to serve those use cases.

More about caching

A cache is a high-speed storage layer that stores a subset of data that's accessed more frequently. This lets us serve these requests faster via the cache instead of the data's primary storage (e.g., search index, relational database). Overall, caching enables efficient reuse of previously fetched or computed data. (More about caching and best practices.)

An example of caching for LLMs is GPTCache.

Overview of GPTCache (source)

When a new request is received:

- Embedding generator: This embeds the request via various models such as OpenAI's `text-embedding-ada-002`, FastText, Sentence Transformers, and more.
- Similarity evaluator: This computes the similarity of the request via the vector store and then provides a distance metric. The vector store can either be local (FAISS, Hnswlib) or cloud-based. It can also compute similarity via a model.
- Cache storage: If the request is similar, the cached response is fetched and served.
- LLM: If the request isn't similar enough, it gets passed to the LLM which then generates the result. Finally, the response is served and cached for future use.

Redis also shared a similar example, mentioning that some teams go as far as precomputing all the queries they anticipate receiving. Then, they set a similarity threshold on which queries are similar enough to warrant a cached response.

How to apply caching?

We should start with having a good understanding of user request patterns. This allows us to design the cache thoughtfully so it can be applied reliably.

First, let's consider a non-LLM example. Imagine we're caching product prices for an e-commerce site. During checkout, is it safe to display the (possibly outdated) cached price? Probably not, since the price the customer sees during checkout should be the same as the final amount they're charged. Caching isn't appropriate here as we need to ensure consistency for the customer.

Now, bringing it back to LLM responses. Imagine we get a request for a summary of "Mission Impossible 2" that's semantically similar enough to "Mission Impossible 3". If we're looking up cache based on semantic similarity, we could serve the wrong response.

We also need to **consider if caching is effective for the usage pattern.** One way to quantify this is via the cache hit rate (percentage of requests served directly from the cache). If the usage pattern is uniformly random, the cache would need frequent updates. Thus, the effort to keep the cache up-to-date could negate any benefit a cache has to offer. On the other hand, if the usage follows a power law where a small proportion of unique requests account for the majority of traffic (e.g., search queries, product views), then caching could be an effective strategy.

Beyond semantic similarity, we could also explore caching based on:

- **Item IDs:** This applies when we pre-compute summaries of product reviews or generate a summary for an entire movie trilogy.
- **Pairs of Item IDs:** Such as when we generate comparisons between two movies. While this appears to be $O(N^2)$, in practice, a small number of combinations drive the bulk of traffic, such as comparison between popular movies in a series or genre.
- **Constrained input:** Such as variables like movie genre, director, or lead actor. For example, if a user is looking for movies by a specific director, we could execute a structured query and run it through an LLM to frame the response more eloquently. Another example is generating code based on drop-down options—if the code has been verified to work, we can cache it for reliable reuse.

Also, **caching doesn't only have to occur on-the-fly.** As Redis shared, we can pre-compute LLM generations offline or asynchronously before serving them. By serving from a cache, we shift the latency from generation (typically seconds) to cache lookup (milliseconds). Pre-computing in batch can also help reduce cost relative to serving in real-time.

While the approaches listed here may not be as flexible as semantically caching on natural language inputs, I think it provides a good balance between efficiency and reliability.

Guardrails: To ensure output quality

In the context of LLMs, guardrails validate the output of LLMs, ensuring that the output doesn't just sound good but is also syntactically correct, factual, and free from harmful content. It also includes guarding against adversarial input.

Why guardrails?

First, they help ensure that model outputs are reliable and consistent enough to use in production. For example, we may require output to be in a specific JSON schema so that it's machine-readable, or we need code generated to be executable. Guardrails can help with such syntactic validation.

Second, they provide an additional layer of safety and maintain quality control over an LLM's output. For example, to verify if the content generated is appropriate for serving, we may want to check that the output isn't harmful, verify it for factual accuracy, or ensure coherence with the context provided.

More about guardrails

One approach is to control the model's responses via prompts. For example, Anthropic shared about prompts designed to guide the model toward generating responses that are helpful, harmless, and honest (HHH). They found that Python fine-tuning with the HHH prompt led to better performance compared to fine-tuning with RLHF.

Example of HHH prompt (source)

A more common approach is to validate the output. An example is the Guardrails package. It allows users to add structural, type, and quality requirements on LLM outputs via Pydantic-style validation. And if the check fails, it can trigger corrective action such as filtering on the offending output or regenerating another response.

Most of the validation logic is in `validators.py`. It's interesting to see how they're implemented. Broadly speaking, its validators fall into the following categories:

- Single output value validation: This includes ensuring that the output (i) is one of the predefined choices, (ii) has a length within a certain range, (iii) if numeric, falls within an expected range, and (iv) is a complete sentence.
- Syntactic checks: This includes ensuring that generated URLs are valid and reachable, and that Python and SQL code is bug-free.
- Semantic checks: This verifies that the output is aligned with the reference document, or that the extractive summary closely matches the source document. These checks can be done via cosine similarity or fuzzy matching techniques.
- Safety checks: This ensures that the generated output is free of inappropriate language or that the quality of translated text is high.

Nvidia's NeMo-Guardrails follows a similar principle but is designed to guide LLM-based conversational systems. Rather than focusing on syntactic guardrails, it emphasizes semantic ones. This includes ensuring that the assistant steers clear of politically charged topics, provides factually correct information, and can detect jailbreaking attempts.

Thus, NeMo's approach is somewhat different: Instead of using more deterministic checks like verifying if a value exists in a list or inspecting code for syntax errors, NeMo leans heavily on using another LLM to validate outputs (inspired by SelfCheckGPT).

In their example for fact-checking and preventing hallucination, they ask the LLM itself to check whether the most recent output is consistent with the given context. To fact-check, the LLM is queried if the response is true based on the documents retrieved from the knowledge base. To prevent hallucinations, since there isn't a knowledge base available, they get the LLM to generate multiple alternative completions which serve as the context. The underlying assumption is that if the LLM produces multiple completions that disagree with one another, the original completion is likely a hallucination.

The moderation example follows a similar approach: The response is screened for harmful and unethical content via an LLM. Given the nuance of ethics and harmful content, heuristics and conventional machine learning techniques fall short. Thus, an LLM is required for a deeper understanding of the intent and structure of dialogue.

Apart from using guardrails to verify the output of LLMs, we can also **directly steer the output to adhere to a specific grammar.** An example of this is Microsoft's Guidance. Unlike Guardrails which imposes JSON schema via a prompt, Guidance enforces the schema by injecting tokens that make up the structure.

We can think of Guidance as a domain-specific language for LLM interactions and output. It draws inspiration from Handlebars, a popular templating language used in web applications that empowers users to perform variable interpolation and logical control.

However, Guidance sets itself apart from regular templating languages by executing linearly. This means it maintains the order of tokens generated. Thus, by inserting tokens that are part of the structure—instead of relying on the LLM to generate them correctly—Guidance can dictate the specific output format. In their examples, they show how to generate JSON that's always valid, generate complex output formats with multiple keys, ensure that LLMs play the right roles, and have agents interact with each other.

They also introduced a concept called token healing, a useful feature that helps avoid subtle bugs that occur due to tokenization. In simple terms, it rewinds the generation by one token before the end of the prompt and then restricts the first generated token to have a prefix matching the last token in the prompt. This eliminates the need to fret about token boundaries when crafting prompts.

How to apply guardrails?

Though the concept of guardrails for LLMs in industry is still nascent, there are a handful of immediately useful and practical strategies we can consider.

Structural guidance: Apply guidance whenever possible. It provides direct control over outputs and offers a more precise method to ensure that output conforms to a specific structure or format.

Syntactic guardrails: These include checking if categorical output is within a set of acceptable choices, or if numeric output is within an expected range. Also, if we generate SQL, these can verify its free from syntax errors and also ensure that all columns in the query match the schema. Ditto for generating code (e.g., Python, JavaScript).

Content safety guardrails: These verify that the output has no harmful or inappropriate content. It can be as simple as checking against the List of Dirty, Naughty, Obscene, and Otherwise Bad Words or using profanity detection models. (It's common to run moderation classifiers on output.) More complex and nuanced output can rely on an LLM evaluator.

Semantic/factuality guardrails: These confirm that the output is semantically relevant to the input. Say we're generating a two-sentence summary of a movie based on its synopsis. We can validate if the produced summary is semantically similar to the output, or have (another) LLM ascertain if the summary accurately represents the provided synopsis.

Input guardrails: These limit the types of input the model will respond to, helping to mitigate the risk of the model responding to inappropriate or adversarial prompts which would lead to generating harmful content. For example, you'll get an error if you ask Midjourney to generate NSFW content. This can be as straightforward as comparing against a list of strings or using a moderation classifier.

An example of an input guardrail on Midjourney

Defensive UX: To anticipate & handle errors gracefully

Defensive UX is a design strategy that acknowledges that bad things, such as inaccuracies or hallucinations, can happen during user interactions with machine learning or LLM-based products. Thus, the intent is to anticipate and manage these in advance, primarily by guiding user behavior, averting misuse, and handling errors gracefully.

Why defensive UX?

Machine learning and LLMs aren't perfect—they can produce inaccurate output. Also, they respond differently to the same input over time, such as search engines displaying varying results due to personalization, or LLMs generating diverse output on more creative, higher temperature, settings. This can violate the principle of consistency which advocates for a consistent UI and predictable behaviors.

Defensive UX can help mitigate the above by providing:

- **Increased accessibility**: By helping users understand how ML/LLM features work and their limitations, defensive UX makes it more accessible and user-friendly.
- **Increased trust**: When users see that the feature can handle difficult scenarios gracefully and doesn't produce harmful output, they're likely to trust it more.
- **Better UX**: By designing the system and UX to handle ambiguous situations and errors, defensive UX paves the way for a smoother, more enjoyable user experience.

More about defensive UX

To learn more about defensive UX, we can look at Human-AI guidelines from Microsoft, Google, and Apple.

Microsoft's Guidelines for Human-AI Interaction is based on a survey of 168 potential guidelines. These were collected from internal and external industry sources, academic literature, and public articles. After combining guidelines that were similar, filtering guidelines that were too vague or too specific or not AI-specific, and a round of heuristic evaluation, they narrowed it down to 18 guidelines.

Guidelines for Human-AI interaction across the user journey (source)

These guidelines follow a certain style: Each one is a succinct action rule of 3 - 10 words, beginning with a verb. Each rule is accompanied by a one-liner that addresses potential ambiguities. They are organized based on their likely application during user interaction:

- Initially: Make clear what the system can do (G1), make clear how well the system can do what it can do (G2)
- During interaction: Time services based on context (G3), mitigate social biases (G6)
- When wrong: Support efficient dismissal (G8), support efficient correction (G9)
- Over time: Learn from user behavior (G13), provide global controls (G17)

Google's People + AI Guidebook is rooted in data and insights drawn from Google's product team and academic research. In contrast to Microsoft's guidelines which are organized around the user, Google organizes its guidelines into concepts that a developer needs to keep in mind.

There are 23 patterns grouped around common questions that come up during the product development process, including:

- How do I get started with human-centered AI: Determine if the AI adds value, invest early in good data practices (e.g., evals)
- How do I onboard users to new AI features: Make it safe to explore, anchor on familiarity, automate in phases
- How do I help users build trust in my product: Set the right expectations, be transparent, automate more when the risk is low.

Apple's Human Interface Guidelines for Machine Learning differs from the bottom-up approach of academic literature and user studies. Instead, its primary source is practitioner knowledge and experience. Thus, it doesn't include many references or data points, but instead focuses on Apple's longstanding design principles. This results in a unique perspective that distinguishes it from the other two guidelines.

The document focuses on how Apple's design principles can be applied to ML-infused products, emphasizing aspects of UI rather than model functionality. It starts by asking developers to consider the role of ML in their app and work backwards from the user experience. This includes questions such as whether ML is:

- Critical or complementary: For example, Face ID cannot work without ML but the keyboard can still work without QuickType.
- Proactive or reactive: Siri Suggestions are proactive while autocorrect is reactive.
- Dynamic or static: Recommendations are dynamic while object detection in Photos only improves with each iOS release.

It then delves into several patterns, split into inputs and outputs of a system. Inputs focus on explicit feedback, implicit feedback, calibration, and corrections. This section guides the design for how AI products request and process user data and interactions. Outputs focus on mistakes, multiple options, confidence, attribution, and limitations. The intent is to ensure the model's output is presented in a comprehensible and useful manner.

The differences between the three guidelines are insightful. Google has more emphasis on considerations for training data and model development, likely due to its engineering-driven culture. Microsoft has more focus on mental models, likely an artifact of the HCI academic study. Lastly, Apple's approach centers around providing a seamless UX, a focus likely influenced by its cultural values and principles.

How to apply defensive UX?

Here are some patterns based on the guidelines above. (Disclaimer: I'm not a designer.)

Set the right expectations. This principle is consistent across all three guidelines:

- Microsoft: Make clear how well the system can do what it can do (help the user understand how often the AI system may make mistakes)
- Google: Set the right expectations (be transparent with your users about what your AI-powered product can and cannot do)
- Apple: Help people establish realistic expectations (describe the limitation in marketing material or within the feature's context)

This can be as simple as adding a brief disclaimer above AI-generated results, like those of Bard, or highlighting our app's limitations on its landing page, like how ChatGPT does it.

Example of a disclaimer on Google Bard results (Note: `nrows` is not a valid argument.)

By being transparent about our product's capabilities and limitations, we help users calibrate their expectations about its functionality and output. While this may cause users to trust it less in the short run, it helps foster trust in the long run—users are less likely to overestimate our product and subsequently face disappointment.

Enable efficient dismissal. This is explicitly mentioned as Microsoft's Guideline 8: Support efficient dismissal (make it easy to dismiss or ignore undesired AI system services).

For example, if a user is navigating our site and a chatbot pops up asking if they need help, it should be easy for the user to dismiss the chatbot. This ensures the chatbot doesn't get in the way, especially on devices with smaller screens. Similarly, GitHub Copilot allows users to conveniently ignore its code suggestions by simply continuing to type. While this may reduce usage of the AI feature in the short term, it prevents it from becoming a nuisance and potentially reducing customer satisfaction in the long term.

Provide attribution. This is listed in all three guidelines:

- Microsoft: Make clear why the system did what it did (enable the user to access an explanation of why the AI system behaved as it did)
- Google: Add context from human sources (help users appraise your recommendations with input from 3rd-party sources)
- Apple: Consider using attributions to help people distinguish among results

Citations are becoming an increasingly common design element. Take BingChat for example. When we make a query, it includes citations, usually from reputable sources, in its responses. This not only shows where the information came from, but also allows users to assess the quality of the sources. Similarly, imagine we're using an LLM to explain why a user might like a product. Alongside the LLM-generated explanation, we could include a quote from an actual review or mention the product rating.

Context from experts and the community also enhances user trust. For example, if a user is seeking recommendations for a hiking trail, mentioning that a suggested trail comes highly recommended by the relevant community can go a long way. It not only adds value to the recommendation but also helps users calibrate trust through the human connection.

Example of attribution via social proof (source)

Finally, Apple's guidelines include popular attributions such as "Because you've read non-fiction", "New books by authors you've read". These descriptors not only personalize the experience but also provide context, enhancing user understanding and trust.

Anchor on familiarity. When introducing users to a new AI product or feature, it helps to guide them with familiar UX patterns and features. This makes it easier for users to focus on the main task and start to earn customer trust in our new product. Resist the temptation to showcase new and "magical" features via exotic UI elements.

Along a similar vein, chat-based features are becoming more common due to ChatGPT's growing popularity. For example, chat with your docs, chat to query your data, chat to buy groceries. However, I question whether chat is the right UX for most user experiences—it just takes too much effort relative to the familiar UX of clicking on text and images.

Furthermore, increasing user effort leads to higher expectations that are harder to meet. Netflix shared that users have higher expectations for recommendations that result from explicit actions such as search. In general, the more effort a user puts in (e.g., chat, search), the higher the expectations they have. Contrast this with lower-effort interactions such as scrolling over recommendations slates or clicking on a product.

Thus, while chat offers more flexibility, it also demands more user effort. Moreover, using a chat box is less intuitive as it lacks signifiers on how users can adjust the output. Overall, I think that sticking with a familiar and constrained UI makes it easier for users to navigate our product; chat should only be considered as a secondary or tertiary option.

Collect user feedback: To build our data flywheel

Gathering user feedback allows us to learn their preferences. Specific to LLM products, user feedback contributes to building evals, fine-tuning, and guardrails. If we think about it, data—such as corpus for pre-training, expert-crafted demonstrations, human preferences for reward modeling—is one of the few moats for LLM products. Thus, we want to be deliberately thinking about collecting user feedback when designing our UX.

Feedback can be explicit or implicit. Explicit feedback is information users provide in response to a request by our product; implicit feedback is information we learn from user interactions without needing users to deliberately provide feedback.

Why collect user feedback

User feedback **helps our models improve**. By learning what users like, dislike, or complain about, we can improve our models to better meet their needs. It also allows us to **adapt to individual preferences**. Recommendation systems are a prime example. As users interact with items, we learn what they like and dislike and better cater to their tastes over time.

In addition, the feedback loop helps us **evaluate our system's overall performance**. While evals can help us measure model/system performance, user feedback offers a concrete measure of user satisfaction and product effectiveness.

How to collect user feedback

Make it easy for users to provide feedback. This is echoed across all three guidelines:

- Microsoft: Encourage granular feedback (enable the user to provide feedback indicating their preferences during regular interaction with the AI system)
- Google: Let users give feedback (give users the opportunity for real-time teaching, feedback, and error correction)
- Apple: Provide actionable information your app can use to improve the content and experience it presents to people

ChatGPT is one such example. Users can indicate thumbs up/down on responses, or choose to regenerate a response if it's really bad or unhelpful. This is useful feedback on human preferences which can then be used to fine-tune LLMs.

Midjourney is another good example. After images are generated, users can generate a new set of images (negative feedback), tweak an image by asking for a variation (positive feedback), or upscale and download the image (strong positive feedback). This enables Midjourney to gather rich comparison data on the outputs generated.

Example of collecting user feedback as part of the UX

Consider implicit feedback too. Implicit feedback is information that arises as users interact with our product. Unlike the specific responses we get from explicit feedback, implicit feedback can provide a wide range of data on user behavior and preferences.

Copilot-like assistants are a prime example. Users indicate whether a suggestion was helpful by either wholly accepting it (strong positive feedback), accepting and making minor tweaks (positive feedback), or ignoring it (neutral/negative feedback). Alternatively, they may update the comment that led to the generated code, suggesting that the initial code generation didn't meet their needs.

Chatbots, such as ChatGPT and BingChat, are another example. How has daily usage changed over time? If the product is sticky, it suggests that users like it. Also, how long is the average conversation? This can be tricky to interpret: Is a longer conversation better because the conversation was engaging and fruitful? Or is it worse because it took the user longer to get what they needed?

Other patterns common in machine learning

Apart from the seven patterns above, there are other patterns in machine learning that are also relevant to LLM systems and products. They include:

- Data flywheel: Continuous data collection improves the model and leads to a better user experience. This, in turn, promotes more usage which provides more data to further evaluate and fine-tune models, creating a virtuous cycle.
- Cascade: Rather than assigning a single, complex task to the LLM, we can simplify and break it down so it only has to handle tasks it excels at, such as reasoning or communicating eloquently. RAG is an example of this. Instead of relying on the LLM to retrieve and rank items based on its internal knowledge, we can augment LLMs with external knowledge and focus on applying the LLM's reasoning abilities.
- Monitoring: This helps demonstrate the value added by the AI system, or the lack of it. Someone shared an anecdote of running an LLM-based customer support solution in prod for two weeks before discontinuing it—an A/B test showed that losses were 12x more when using an LLM as a substitute for their support team!

(Read more about design patterns for machine learning code and systems.)

Also, here's what others said:

> Separation of concerns/task decomposition- having distinct prompts for distinct subtasks and chaining them together helps w attention and reliability (hurts latency). We were having trouble specifying a rigid output structure AND variable response content so we split up the tasks — Erick Enriquez

> A few others that will be needed: role based access control: who can access what; security: if I'm using a DB with an LLM, how do I ensure that I have the right security guards — Krishna

> Consistent output format: setting outputs to a standardized format such as JSON; Tool augmentation: offload tasks to more specialised, proven, reliable models — Paul Tune

> Security: mitigate cache poisoning, input validation, mitigate prompt injection, training data provenance, output with non-vulnerable code, mitigate malicious input aimed at influencing requests used by tools (AI Agent), mitigate denial of service (stress test llm), to name a few :) — Anderson Darario

> Another ux/ui related: incentivize users to provide feedback on generated answers (implicit or explicit). Implicit could be sth like copilot's ghost text style, if accepted with TAB, meaning positive feedback etc. — Wen Yang

> Great list. I would add consistency checks like self-consistency sampling, chaining and decomposition of tasks, and the emsembling of multiple model outputs. Applying each of these almost daily. Dan White

> Guardrails is super relevant for building analytics tools where llm is a translator from natural to programming language — m_voitko

Conclusion

This is the longest post I've written by far. If you're still with me, thank you! I hope you found reading about these patterns helpful, and that the 2x2 below makes sense.

LLM patterns across the axis of data to user, and defensive to offensive.

We're still so early on the journey towards building LLM-based systems and products. Are there any other key patterns or resources? What have you found useful or not useful? I'd love to hear your experience. **Please reach out!**

References

Hendrycks, Dan, et al. "Measuring massive multitask language understanding." arXiv preprint arXiv:2009.03300 (2020).

Gao, Leo, et al. "A Framework for Few-Shot Language Model Evaluation." v0.0.1, Zenodo, (2021), doi:10.5281/zenodo.5371628.

Liang, Percy, et al. "Holistic evaluation of language models." arXiv preprint arXiv:2211.09110 (2022).

Dubois, Yann, et al. "AlpacaFarm: A Simulation Framework for Methods That Learn from Human Feedback." (2023)

Papineni, Kishore, et al. "Bleu: a method for automatic evaluation of machine translation." Proceedings of the 40th annual meeting of the Association for Computational Linguistics. 2002.

Lin, Chin-Yew. "Rouge: A package for automatic evaluation of summaries." Text summarization branches out. 2004.

Zhang, Tianyi, et al. "Bertscore: Evaluating text generation with bert." arXiv preprint arXiv:1904.09675 (2019).

Zhao, Wei, et al. "MoverScore: Text generation evaluating with contextualized embeddings and earth mover distance." arXiv preprint arXiv:1909.02622 (2019).

Sai, Ananya B., Akash Kumar Mohankumar, and Mitesh M. Khapra. "A survey of evaluation metrics used for NLG systems." ACM Computing Surveys (CSUR) 55.2 (2022): 1-39.

Grusky, Max. "Rogue Scores." Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2023.

Liu, Yang, et al. "Gpteval: Nlg evaluation using gpt-4 with better human alignment." arXiv preprint arXiv:2303.16634 (2023).

Fourrier, Clémentine, et al. "What's going on with the Open LLM Leaderboard?" (2023).

Zheng, Lianmin, et al. "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena." arXiv preprint arXiv:2306.05685 (2023).

Dettmers, Tim, et al. "Qlora: Efficient finetuning of quantized llms." arXiv preprint arXiv:2305.14314 (2023).

Swyx et al. MPT-7B and The Beginning of Context=Infinity (2023).

Fradin, Michelle, Reeder, Lauren "The New Language Model Stack" (2023).

Radford, Alec, et al. "Learning transferable visual models from natural language supervision." International conference on machine learning. PMLR, 2021.

Yan, Ziyou. "Search: Query Matching via Lexical, Graph, and Embedding Methods." eugeneyan.com, (2021).

Petroni, Fabio, et al. "How context affects language models' factual predictions." arXiv preprint arXiv:2005.04611 (2020).

Karpukhin, Vladimir, et al. "Dense passage retrieval for open-domain question answering." arXiv preprint arXiv:2004.04906 (2020).

Lewis, Patrick, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks." Advances in Neural Information Processing Systems 33 (2020): 9459-9474.

Izacard, Gautier, and Edouard Grave. "Leveraging passage retrieval with generative models for open domain question answering." arXiv preprint arXiv:2007.01282 (2020).

Borgeaud, Sebastian, et al. "Improving language models by retrieving from trillions of tokens." International conference on machine learning. PMLR, (2022).

Lazaridou, Angeliki, et al. "Internet-augmented language models through few-shot prompting for open-domain question answering." arXiv preprint arXiv:2203.05115 (2022).

Wang, Yue, et al. "Codet5+: Open code large language models for code understanding and generation." arXiv preprint arXiv:2305.07922 (2023).

Gao, Luyu, et al. "Precise zero-shot dense retrieval without relevance labels." arXiv preprint arXiv:2212.10496 (2022).

Yan, Ziyou. "Obsidian-Copilot: An Assistant for Writing & Reflecting." eugeneyan.com, (2023).

Bojanowski, Piotr, et al. "Enriching word vectors with subword information." Transactions of the association for computational linguistics 5 (2017): 135-146.

Reimers, Nils, and Iryna Gurevych. "Making Monolingual Sentence Embeddings Multilingual Using Knowledge Distillation." Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, (2020).

Wang, Liang, et al. "Text embeddings by weakly-supervised contrastive pre-training." arXiv preprint arXiv:2212.03533 (2022).

Su, Hongjin, et al. "One embedder, any task: Instruction-finetuned text embeddings." arXiv preprint arXiv:2212.09741 (2022).

Johnson, Jeff, et al. "Billion-Scale Similarity Search with GPUs." IEEE Transactions on Big Data, vol. 7, no. 3, IEEE, 2019, pp. 535–47.

Malkov, Yu A., and Dmitry A. Yashunin. "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs." IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 4, IEEE, 2018, pp. 824–36.

Guo, Ruiqi, et al. "Accelerating Large-Scale Inference with Anisotropic Vector Quantization." International Conference on Machine Learning, (2020)

Ouyang, Long, et al. "Training language models to follow instructions with human feedback." Advances in Neural Information Processing Systems 35 (2022): 27730-27744.

Howard, Jeremy, and Sebastian Ruder. "Universal language model fine-tuning for text classification." arXiv preprint arXiv:1801.06146 (2018).

Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

Radford, Alec, et al. "Improving language understanding with unsupervised learning." (2018).

Raffel, Colin, et al. "Exploring the limits of transfer learning with a unified text-to-text transformer." The Journal of Machine Learning Research 21.1 (2020): 5485-5551.

Lester, Brian, Rami Al-Rfou, and Noah Constant. "The power of scale for parameter-efficient prompt tuning." arXiv preprint arXiv:2104.08691 (2021).

Li, Xiang Lisa, and Percy Liang. "Prefix-tuning: Optimizing continuous prompts for generation." arXiv preprint arXiv:2101.00190 (2021).

Houlsby, Neil, et al. "Parameter-efficient transfer learning for NLP." International Conference on Machine Learning. PMLR, 2019.

Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." arXiv preprint arXiv:2106.09685 (2021).

Dettmers, Tim, et al. "Qlora: Efficient finetuning of quantized llms." arXiv preprint arXiv:2305.14314 (2023).

Williams, Adina, et al. "A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference." Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), Association for Computational Linguistics, (2018).

GPTCache (2023).

Bai, Yuntao, et al. "Training a helpful and harmless assistant with reinforcement learning from human feedback." arXiv preprint arXiv:2204.05862 (2022).

Guardrails (2023)

NeMo-Guardrails (2023)

Manakul, Potsawee, Adian Liusie, and Mark JF Gales. "Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models." arXiv preprint arXiv:2303.08896 (2023).

Guidance (2023).

Amershi, Saleema, et al. "Guidelines for human-AI interaction." Proceedings of the 2019 chi conference on human factors in computing systems. 2019.

People + AI Guidebook (2023).

Human Interface Guidelines for Machine Learning (2023).

Schendel, Zachary A., Faraz Farzin, and Siddhi Sundar. "A Human Perspective on Algorithmic Similarity." Proceedings of the 14th ACM Conference on Recommender Systems. 2020.

If you found this useful, please cite this write-up as:

> Yan, Ziyou. (Jul 2023). Patterns for Building LLM-based Systems & Products. eugeneyan.com. https://eugeneyan.com/writing/llm-patterns/.

or

```
@article{yan2023llm-patterns,
  title   = {Patterns for Building LLM-based Systems & Products},
  author  = {Yan, Ziyou},
  journal = {eugeneyan.com},
  year    = {2023},
  month   = {Jul},
  url     = {https://eugeneyan.com/writing/llm-patterns/}
}
```

Share on:

Browse related tags: [llm engineering production 🔥] Search

« Obsidian-Copilot: An Assistant for Writing & Reflecting

How to Match LLM Patterns to Problems »

Join **7,100+** readers getting updates on machine learning, RecSys, LLMs, and engineering.

Your email address... Get email updates

Twitter
LinkedIn
Threads
GitHub

Eugene Yan designs, builds, and operates machine learning systems that serve customers at scale. He's currently a Senior Applied Scientist at Amazon. Previously, he led machine learning at Lazada (acquired by Alibaba) and a Healthtech Series A. He writes & speaks about machine learning, recommenders, LLMs, and engineering at eugeneyan.com and ApplyingML.com.

© Eugene Yan 2015 - 2024 • Feedback • RSS