

✦ Member-only story

# LLM Evaluation Skills Are Easy to Pick Up (Yet Costly to Practice)

Here's how not to waste your budget on evaluating models and systems



Thuwarakesh Murallie · [Follow](#)

Published in Towards Data Science · 14 min read · 2 days ago



604



16





You can build a fortress in two ways: Start stacking bricks one above the other, or draw a picture of the fortress you're about to build and plan its execution; then, keep evaluating it against your plan.

We all know the second one is the only way we can *possibly* build a fortress.

Sometimes, I'm the worst follower of my advice. I'm talking about jumping straight into a notebook to build an LLM app.

It's the worst thing we can do to ruin our project.

Before we begin anything, we need a mechanism to tell us we're moving in the right direction — to say that the last thing we tried was better than before (or otherwise.)

In software engineering, it's called test-driven development. For machine learning, it's evaluation.

*The first step and the most valuable skill in developing LLM-powered applications is to define how you'll evaluate your project.*

Evaluating LLM applications is nowhere like software testing. I don't undermine the challenges in software testing, but evaluating LLMs isn't as straightforward as testing.

An LLM speaks a lot like our natural language. And there are a million different ways you can say the same thing. Just pull up a random WhatsApp group you are a member of and see how people say “Good Morning.” Likewise, LLMs can produce the correct answer in so many different wordings.

However, evaluating programmed outcomes often looks for a one-to-one match. A function that adds two numbers should always return three if the inputs are 2 and 1.

Thankfully, the LLM community has figured out ways to evaluate such ambiguous responses. Often, this involves another LLM that acts as an evaluator. But it's not the only way to validate your models.

LLM-assisted evaluation is faster and more helpful during development because we can run it repeatedly. But every

LLM call comes with a price tag. Though the prices seem meager on the providers' pricing pages, they add up quickly.

This post concerns the LLM-assisted evaluation techniques—for RAGs in particular. However, we'll also talk about ways to reduce the evaluation cost. Finally, we'll also talk about the evaluation techniques that don't involve an LLM.

## **To evaluate, you need a dataset.**

Let's not overcomplicate evaluation. How'd you tell if a person answered your questions correctly?



You'd already have a list of questions with correct answers — either on paper or in your mind. You'd then ask questions from this collection and decide whether the person's answers are correct.

Software testing is not too far from this approach—neither is the evaluation technique for LLM.

So we need a dataset, a list of inputs, prompts, questions, whatever you'd like to call them, and expected outputs. We can then count the generated outputs that match the

expected output to get a measurable value for the LLM's performance.

Doing this is the basics of evaluation. But an LLM can lie to us for many different reasons. We'll come to that in a moment. But first, we need to prepare the dataset.

These are the top n prompts your end user is more likely to input. Depending on your use cases, this number can be in 10s, 100s, 1000s, or even more. Also, this is a task that the ML engineer and domain experts would work on.

The following function evaluates LLM outputs for correctness against the input and expected outputs.

```
# pip install openai
# Set OPENAI_API_KEY environment variable

import os
from openai import OpenAI

# Assuming 'openai.api_key' is set elsewhere in the code
client = OpenAI(
    # This is the default and can be omitted
    api_key=os.environ.get("OPENAI_API_KEY"),
)

def evaluate_correctness(input, expected_output, actual_output
```

```
prompt = f"""
```

```
Input: {input}
```

```
Expected Output: {expected_output}
```

```
Actual Output: {actual_output}
```

```
Based on the above information, evaluate the correctness of the output.  
Provide a score from 0 to 1, where 0 is completely incorrect and 1 is completely correct.  
Only return the numerical score.
```

```
"""
```

```
response = client.chat.completions.create(
```

```
    model="gpt-4",
```

```
    messages=[
```

```
        {
```

```
            "role": "system",
```

```
            "content": "You are an AI assistant tasked with evaluating the correctness of the output.",
```

```
        },
```

```
        {"role": "user", "content": prompt},
```

```
    ],
```

```
    temperature=0,
```

```
)
```

```
    return float(response.choices[0].message.content.strip())
```

```
if __name__ == "__main__":
```

```
    dummy_input = "What is the capital of France?"
```

```
    dummy_expected_output = "Paris"
```

```
    dummy_actual_output = "Paris corner"
```

```
    dummy_score = evaluate_correctness(
```

```
        dummy_input, dummy_expected_output, dummy_actual_output
```

```
)
```

```
    print(f"Correctness Score: {dummy_score:.2f}")
```

```
>> Correctness Score: 0.50
```

# Evaluating using a framework

In the previous section, we evaluated the LLM output against the expected output using another LLM. Yet, coding them all by yourself is the last thing you'd want to do. Why should we reinvent the wheels while more evolved, sophisticated solutions are out there?

In this section, I'll use a library called Deepeval to evaluate LLM responses. But Deepeval isn't the only library that can do this. MLFlow LLM Evaluate, RAGAs, and other frameworks are widely used for LLM evaluation. I picked Deepeval because that's what I'm most comfortable with.

One benefit of using frameworks to evaluate LLMs is their many different evaluation metrics. Deepeval offers 14+ evaluation metrics. Plus, you can create custom evaluation functions if needed.

We'll talk more about evaluation metrics in the next section.

Another benefit of these frameworks is that they can generate an evaluation dataset for you. For instance, the following script will use Deepeval and generate as many as 100 Q&A pairs from a PDF.

```
from deepeval.dataset import EvaluationDataset

dataset = EvaluationDataset()
dataset.generate_goldens_from_docs(
    document_paths=['path/to/doc.pdf'],
    max_goldens_per_document=10
)
```

The dataset these frameworks generate isn't perfect. But they are a good start. You could take it to the subject matter experts and enrich your actual evaluation dataset.



Here's a basic example of how to evaluate test cases in our dataset using deepeval.

```
from deepeval import assert_test
from deepeval.test_case import LLMTestCase
from deepeval.metrics import AnswerRelevancyMetric

# Initialize the relevancy metric with a threshold value
relevancy_metric = AnswerRelevancyMetric(threshold=0.5)

# Define the test case with input, the LLM's response, and rel
test_case = LLMTestCase(
    input="What options do I have if I'm unhappy with my order
    actual_output="You can return it within 30 days for a full
    retrieval_context=["Our policy allows returns within 30 da
)
```

```
# Directly evaluate the test case using the specified metric  
assert_test(test_case, [relevancy_metric])
```

This code here creates an object called LLMTestCase. This object has all the required properties for testing. Different evaluation metrics would need different properties, but most of them need at least the input.

More on evaluation metrics will be discussed in the next section.

---

Now that we have a framework to work with and an evaluation dataset to test let's see how we can run evaluations for different

## **Evaluation metrics and RAGs**

So far in this post, we've only discussed the correctness of the LLM-generated content. In other words, the answer is directly compared against the input and the expected output and evaluated if appropriate.

It's often a must-have metric. However, there are other, still more critical metrics you need to track throughout your project.

Let's think about a RAG application. To serve the user's prompt, we query a data store for relevant information and use it to generate a more appropriate answer.

As you'd have guessed, a RAG system may produce incorrect answers in multiple ways. This post doesn't discuss how to build a RAG system. You can check out some of my previous

posts if you're interested. The focus here would only be evaluating the outputs of RAGs.

Four metrics are very popular when evaluating RAG systems: answer relevancy, contextual precision, contextual recall, and the faithfulness metric.

## **Answer Relevancy**

If someone asks you who the first man to walk on the moon was, and you answer Columbus was the first to find America, your answer is entirely irrelevant.

This is what we test with answer relevancy.

We use an LLM to test if the AI-generated text is at least relevant to the prompt.

The following code checks answer relevancy using Deepeval.

```
from deepeval import evaluate
from deepeval.metrics import AnswerRelevancyMetric
from deepeval.test_case import LLMTestCase
```

```
# Define your LLM output and test case
output = "Our working hours are Monday to Friday, 9 AM to 6 PM"
test_case = LLMTestCase(
    input="What are your business hours?",
    actual_output=output
)

# Initialize the relevancy metric
metric = AnswerRelevancyMetric(threshold=0.7)

# Measure and print the score
metric.measure(test_case)
print(f"Score: {metric.score}, Reason: {metric.reason}")
```

The above code passes the test if the score meets the threshold set to 0.7. We asked about working hours, and the answer was also about working hours. Thus, it scores high and passes the test.

Note that we don't know if the answer is correct in this test. This may be a night shop that opens only after 6 PM. Yet, it's a whole different metric to measure.

## **Contextual Precession**

Contextual precession is a crucial metric for evaluating whether your retrieval system ranks the relevant documents



higher than the others.

Suppose you ask someone about who the president of the US was during the Apollo 11 mission. In that case, that person says, “Obama was president when Bin Laden was eliminated, and John F Kennedy was president when Amstrong stepped on the Moon.”

The person has the answer but puts forward an irrelevant one first. This reduces the contextual precession score.

Here's how to do it with Deepeval.

```
from deepeval import evaluate
from deepeval.metrics import ContextualPrecisionMetric
from deepeval.test_case import LLMTestCase

# New LLM output and expected response
generated_output = "Our phone support is available 24/7 for pr
expected_response = "Premium users have 24/7 access to phone s

# Contextual information retrieved from RAG pipeline
retrieved_context = [
    "General users don't have phone support",
    "Premium members can reach our phone support team at any t
    "General users can get email support"
]

# Set up the metric and test case
metric = ContextualPrecisionMetric(threshold=0.8)
test_case = LLMTestCase(
```

```
input="What support options do premium users have?",  
actual_output=generated_output,  
expected_output=expected_response,  
retrieval_context=retrieved_context  
)  
  
# Measure and display results  
metric.measure(test_case)  
print(f"Score: {metric.score}, Reason: {metric.reason}")  
  
>> Score: 0.5, Reason: The score is 0.50 because the first and
```

As expected, the test didn't meet the threshold value of 0.8 because of incorrect ranking. However, it does have the

correct answer within the retrieved context.

## **Contextual Recall**

Contextual recall measures whether the retrieval context is sufficient to answer the problem.

Let's take the same example we used to discuss contextual precision. The second document in the retrieved context provides sufficient information to answer the user's questions. Thus, it should score high for this metric.

Here's how it works in Deepeval.

```
from deepeval import evaluate
from deepeval.metrics import ContextualRecallMetric
from deepeval.test_case import LLMTestCase

# New LLM output and expected response
generated_output = "Premium users get access to 24/7 phone support"
expected_response = "Premium users have 24/7 access to phone support"

# Contextual information retrieved from RAG pipeline
retrieved_context = [
    "General users do not have access to phone support.",
    "Premium members can reach our phone support team at any time.",
    "General users can only get email support."
]

# Set up the recall metric and test case
metric = ContextualRecallMetric(threshold=0.8)
test_case = LLMTestCase(
```

```
input="What support options do premium users have?",
actual_output=generated_output,
expected_output=expected_response,
retrieval_context=retrieved_context
)

# Measure and display results
metric.measure(test_case)
print(f"Recall Score: {metric.score}, Reason: {metric.reason}")

>> Recall Score: 1.0, Reason: The score is 1.00 because the ex
```

## The faithfulness metric

The other critical metric that helps us evaluate RAG systems is the faithfulness metric. This one purely evaluates the

ability of the final LLM to produce the output with the context provided.

In other words, Answer relevancy is an overall check, contextual precision, and recall test of the retrieval system, and the faithfulness metric tests the LLM that produces output for the user.

You'd get a high score if there isn't any contradiction between the retrieved context and the output (and vice versa)

Here's how to implement this with Deepeval:

```
from deepeval import evaluate
from deepeval.metrics import FaithfulnessMetric
from deepeval.test_case import LLMTestCase

# New LLM output and corresponding context
actual_output = "Basic plan users can upgrade anytime to the p

# Contextual information retrieved from RAG pipeline
retrieved_context = [
    "Users on the Basic plan have the option to upgrade to Pre
    "The Premium plan includes additional benefits like 24/7 s
]

# Set up the faithfulness metric and test case
```



```
metric = FaithfulnessMetric(threshold=0.75)
test_case = LLMTestCase(
    input="Can Basic plan users upgrade to Premium anytime?",
    actual_output=actual_output,
    retrieval_context=retrieved_context
)

# Measure and display results
metric.measure(test_case)
print(f"Faithfulness Score: {metric.score}, Reason: {metric.re
```

As I said, these aren't the only metrics you'd use in LLM development. For instance, Deepeval offers out-of-the-box metrics for hallucination checks, biases, and toxicity. Yet,

these are popular ones used to evaluate LLMs and RAG systems.

## **RAGAS**

A combined version of these four tests is called RAGAS. RAGAS is the average of the four tests we discussed so far. It's a single number that can be compared with different RAG systems.

In Deepeval, you can implement it with a single metric instead of manually calling all four and averaging them manually.

```
from deepeval import evaluate
from deepeval.metrics.ragas import RagasMetric
from deepeval.test_case import LLMTestCase

# LLM-generated response, expected response, and retrieved context
llm_response = "The device includes a one-year warranty with free shipping"
target_response = "This product comes with a 12-month warranty"
retrieval_context = [
    "All electronic products are backed by a 12-month warranty, including accessories."
]

# Initialize the Ragas metric with a specific threshold and model
metric = RagasMetric(threshold=0.6)

# Create a test case for the given input and output comparison
test_case = LLMTestCase(
    input="Does this product come with a warranty?",
    actual_output=llm_response,
```

```
        expected_output=target_response,  
        retrieval_context=retrieval_context  
    )  
  
    # Calculate the metric score for this specific test case  
    score = metric.measure(test_case)  
    print(f"Metric Score: {score}")  
  
>> Metric Score: 0.9768281253135719
```

While the RAGAS metric score is beneficial in comparing models and systems, you should not solely depend on it. The individual metrics would tell you where you can make improvements. You wouldn't get this with RAGAS.

# The cost of evaluation

I recently ran a RAGAS evaluation for a small 50 Q&A evaluation dataset. I used the default settings of Deepeval, which calls the OpenAI's GPT-4o model for evaluation. Let's look at the dashboard to study the cost

**Medium**

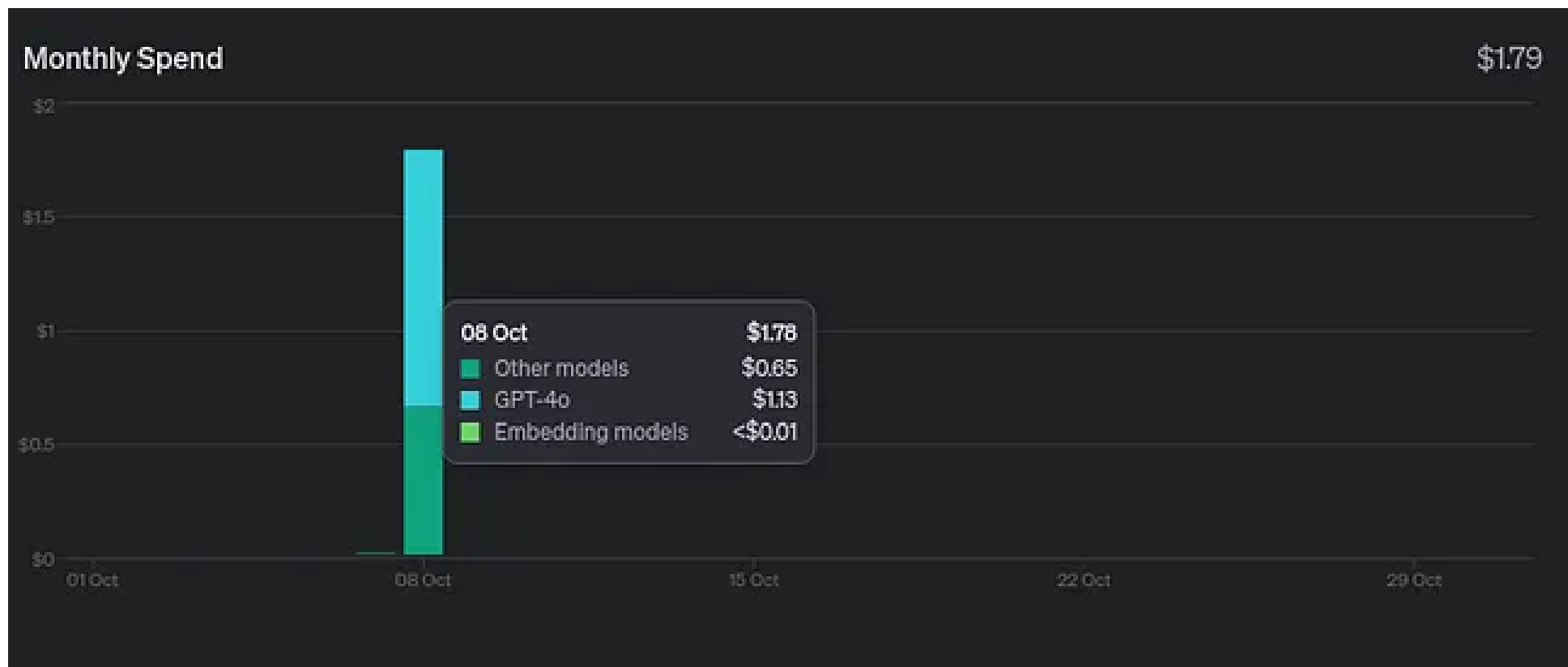


Search



Write





You'd think \$1.13 isn't significant for an evaluation. But evaluations are something you'd run very often. If you switch ten models, you'd run the evaluations ten times. But you'd be doing hundreds of improvements throughout the

project for large applications — everything should be evaluated.

This is because RAGAS triggers not one but many LLM calls. For context, my 50 test case dataset has triggered 290 LLM calls and generated 425,086 tokens in total.



If you have more context and the outputs generated are longer, you will experience an even higher cost.

Besides, the cost is only for 50 pairs of Q&A. In real projects, it's often far too significant a number. Plus, with more and more user feedback, your evaluation dataset would also start to grow.

Now, it's a cost that most smaller companies can't afford, and even the big ones shouldn't waste it for nothing.



Thankfully, there are ways you can reduce it to a smaller amount. Here are my two techniques to reduce your evaluation costs.

## **#1 Switch the model.**

Deepeval evaluates using GPT-4o by default. It was their most expensive model until the o1-preview was introduced.

The easiest cost-saving technique would be to change the model to GPT-3.5-turbo or GPT-4o-mini. GPT-4o-mini, for instance, costs only \$0.15 per 1M input tokens, whereas the same is \$2.5 for GPT-4o.

You can specify which model to use at a metric level in Deepeval. Here's how we specify it for RAGAS metric we've just used.

```
ragas = RagasMetric(threshold=0.7, model='gpt-4o-mini')  
  
# ragas = RagasMetric(threshold=0.7, model='gpt-3.5-turbo')
```

Here's a caveat, though.

Evaluation models need to have excellent reasoning capabilities. While the GPT-4o model is superior in reasoning, the mini version doesn't disappoint you in most cases.

## **#2 Switch the model provider**

I agree that Open AI is the most popular LLM API provider.

But they aren't the only ones. You can switch to a different low-cost model provider like Together.ai or Openrouter.

The plus side of going with different model providers is exploring the many open-source LLMs out there. You could try Llama 3.1 8B for roughly one-third of the cost of GPT-4o-mini. Yet, the model is good with reasoning and may be a good candidate for your evaluation process.

When choosing a provider, consider whether your evaluation framework supports it. Deepeval doesn't directly support providers such as Together AI, but it does support locally running models. We can use this feature to configure different providers if they provide an Open AI compatible API—both Together AI and Openrouter do.

```
deepeval set-local-model --model-name=<model_name> \  
  --base-url="http://localhost:11434/v1/" \  
  --api-key="TOGETHER_AI_API_KEY"
```

The above example showed how to configure deepeval for TogetherAI. All you need is an OpenAI-compatible API and a key. It's the same if you're using Openrouter.

### **#3 Host the model yourself**

The last technique I'd advise you is to host your models yourself.

There's a reason why this is last.

People claim that the locally hosted models are cheaper. But they are not. This is a topic for a future post.

Plus, setting up a local LLM requires powerful GPUs, or you might have to rent one online. This comes with an extra layer of technical overhead you don't want to deal with.

Yet, if you're already running a local model, you might as well use it for evaluation. The setup with Deepeval is similar to what we did in the last section.

# Evaluate LLMs without involving another LLM.

Evaluating LLM-powered apps is challenging because the output is sometimes different. But you can evaluate them even without an LLM evaluator.

The most popular evaluation technique would be **human-in-the-loop**. While this method is unsuitable for apps in development, it's often the best for apps already running or in the user-acceptance testing phase.

If you've used chatbots like ChatGPT, you'd see a tiny feedback section at the end of every response. You can

thumb up or down the response, which will be taken as input to improve the model in subsequent versions.

This technique is best for two reasons: it costs nothing and is the most accurate.

Another no-llm evaluation technique is to use a **similarity score**. Of course, you'd need an embedding model to do this. However, embedding models cost far less compared to text-generating models. Further, you can get a decent model like E5 running locally on a fairly good computer.



Here's a little snippet that checks the similarity between the expected and actual outputs. We use OpenAI embeddings.

```
import openai
import numpy as np

def get_embedding(text, model="text-embedding-ada-002"):
    response = openai.Embedding.create(
        input=text,
        model=model
    )
    return response['data'][0]['embedding']

def cosine_similarity(vec1, vec2):
    return np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2))
```

```
expected_output = "The data processing pipeline completed succ  
actual_output = "The data processing pipeline finished success  
  
embedding_expected = get_embedding(expected_output)  
embedding_actual = get_embedding(actual_output)  
  
similarity_score = cosine_similarity(embedding_expected, embec  
  
print(f"Similarity between expected and actual outputs: {simil
```

Scores like these are easy to compute and cheaper, which can help automate the optimization of LLM-powered applications.

## Final thoughts

Not just for LLM apps, not just for software development projects, any project needs to be evaluated in a systematic, reproducible manner. To me, it's the most valuable skill.

Trickily, evaluating LLM apps would be far more complex than projected with predictable outcomes. We have to rely on another LLM to do it for us for large-scale automated evaluation — one with excellent reasoning abilities.

Thankfully, with tools like RAGAS and Deepeval, we don't have to code our systems repeatedly. We can evaluate them

quickly using dozens of metrics.

In this post, we've discussed the key metrics we need to evaluate, especially RAG systems, how to control costs, and non-llm techniques for evaluating our projects.

• • •

*Thanks for reading, friend! Besides Medium, I'm on LinkedIn and X, too!*

Large Language Models

Retrieval Augmented

Testing

Machine Learning

Data Science



## Written by Thuwarakesh Murallie

2.9K Followers · Writer for Towards Data Science

Data Science Journalist & Independent Consultant

Follow



---

More from Thuwarakesh Murallie and Towards Data Science



Thuwarakesh M... in Towards Data S...

## Building RAGs Without A Retrieval Model Is a...

Here are my favorite techniques — one is faster, the other is more...



Sep  
17



370



2



Shaw Talebi in Towards Data Science

## 5 AI Projects You Can Build This Weekend (with...

From beginner-friendly to advanced



3d  
ago



1.1K



11





Mauro Di Pie... in Towards Data Scie...

## GenAI with Python: Build Agents from Scratch...

with Ollama, LangChain,  
LangGraph (No GPU, No APIKEY)



Sep  
29



1.5K



19



Thuwarakesh M... in Towards Data S...

## How I Used Clustering to Improve Chunking and...

It's both fast and cost-effective



Sep  
4



238



3





[See all from Thuwarakesh Murallie](#)

[See all from Towards Data Science](#)

## Recommended from Medium



Shaw Talebi in Towards Data Science

## 5 AI Projects You Can Build This Weekend (with...

From beginner-friendly to advanced



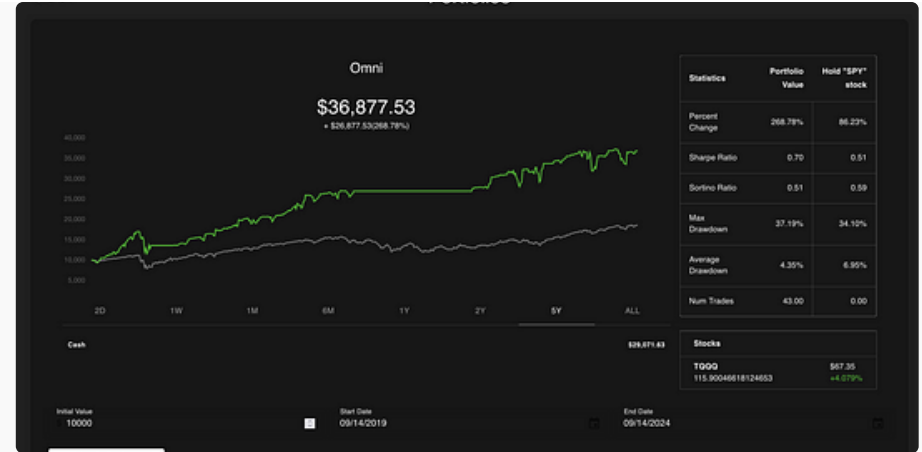
3d  
ago



1.1K



11



Austin Starks in DataDrivenInvestor

## I used OpenAI's o1 model to develop a trading strateg...

It literally took one try. I was shocked.



Sep  
16



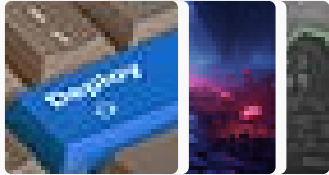
3.6K



100



## Lists



### Predictive Modeling w/...

20 stories · 1585 saves



### Natural Language Processing

1751 stories · 1345 saves



### Practical Guides to Machine Learning

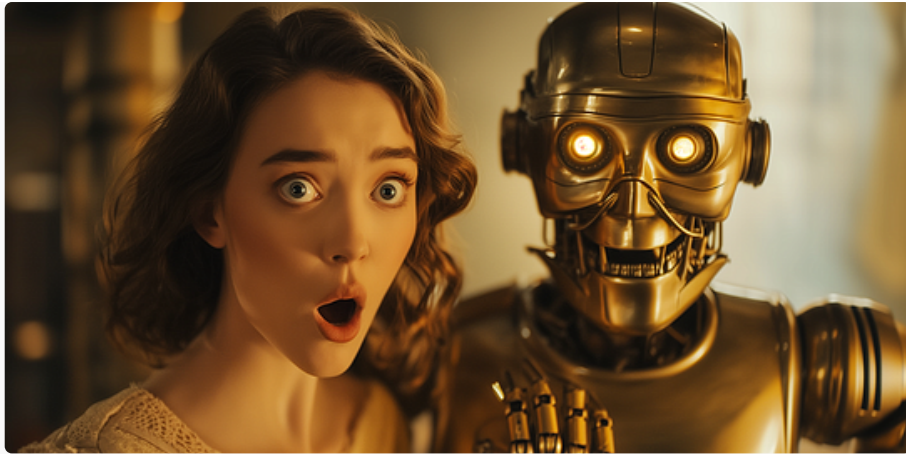
10 stories · 1929 saves



### data science and AI

40 stories · 264 saves

---



Henrique Centieir...  in Limitless I...

## AI: 99% of You Are NOT Ready for What's Coming...

AI's Insane Leap—You Likely Don't Want to Be Left Behind



6d  
ago



3K



81



SHIVAM JINDAL in Stackademic

## Why Meta is So Focused on Probability Questions in...

Recently, two of my friends sat for Software Engineer interviews at...



Sep  
28

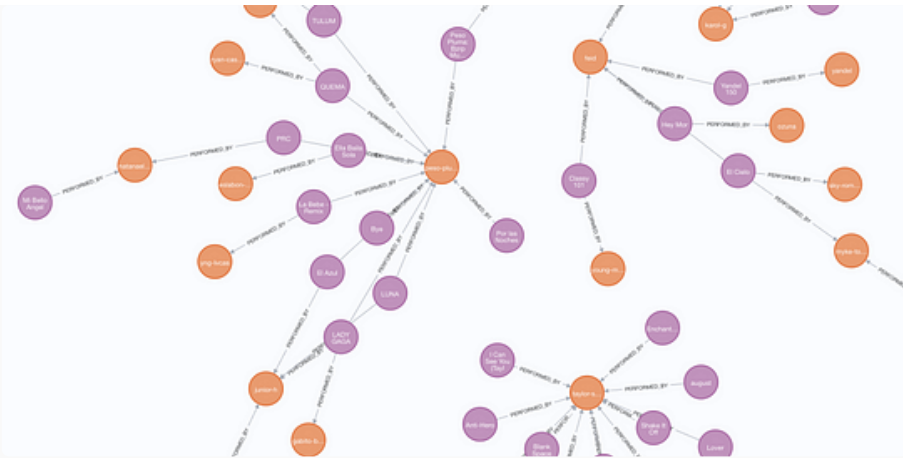


440



14






 Adam Cowl... in Neo4j Developer Bl...

## Turn your CSVs Into Graphs Using LLMs

How do LLMs fare when attempting to create graphs from...

Oct 4  184  1  

 Bryson Meiling in Level Up Coding

## Stop making your python projects like it was 15 yea...

I have a few things I've seen across companies and projects that I've...

 Sep 28  1.4K  20  

[See more recommendations](#)

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#)  
[Teams](#)