Featured

LLM TWIN COURSE: BUILDING YOUR PRODUCTION-READY AI REPLICA

An End-to-End Framework for Production-Ready LLM Systems by Building Your LLM Twin

From data gathering to productionizing LLMs using LLMOps good practices.

Paul Iusztin · Follow
Published in Decoding ML · 16 min read · Mar 16, 2024

*→ the 1st out of 12 lessons of **the LLM Twin** free course*

What is your LLM Twin? It is an AI character that writes like yourself by incorporating your style, personality and voice into an LLM.

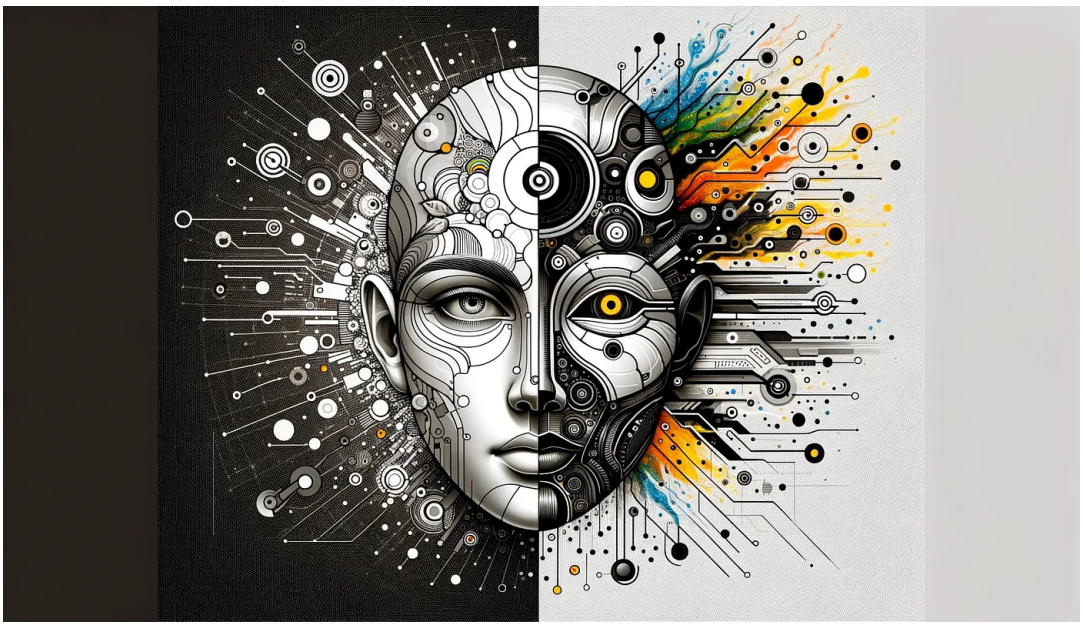


Image by DALL-E

Why is this course different?

*By finishing the "**LLM Twin: Building Your Production-Ready AI Replica**" free course, you will learn how to design, train, and deploy a production-ready LLM twin of yourself powered by LLMs, vector DBs, and LLMOps good practices.*

> *Why should you care?* 👊
>
> *→ **No more isolated scripts or Notebooks!** Learn production ML by building and deploying an end-to-end production-grade LLM system.*

What will you learn to build by the end of this course?

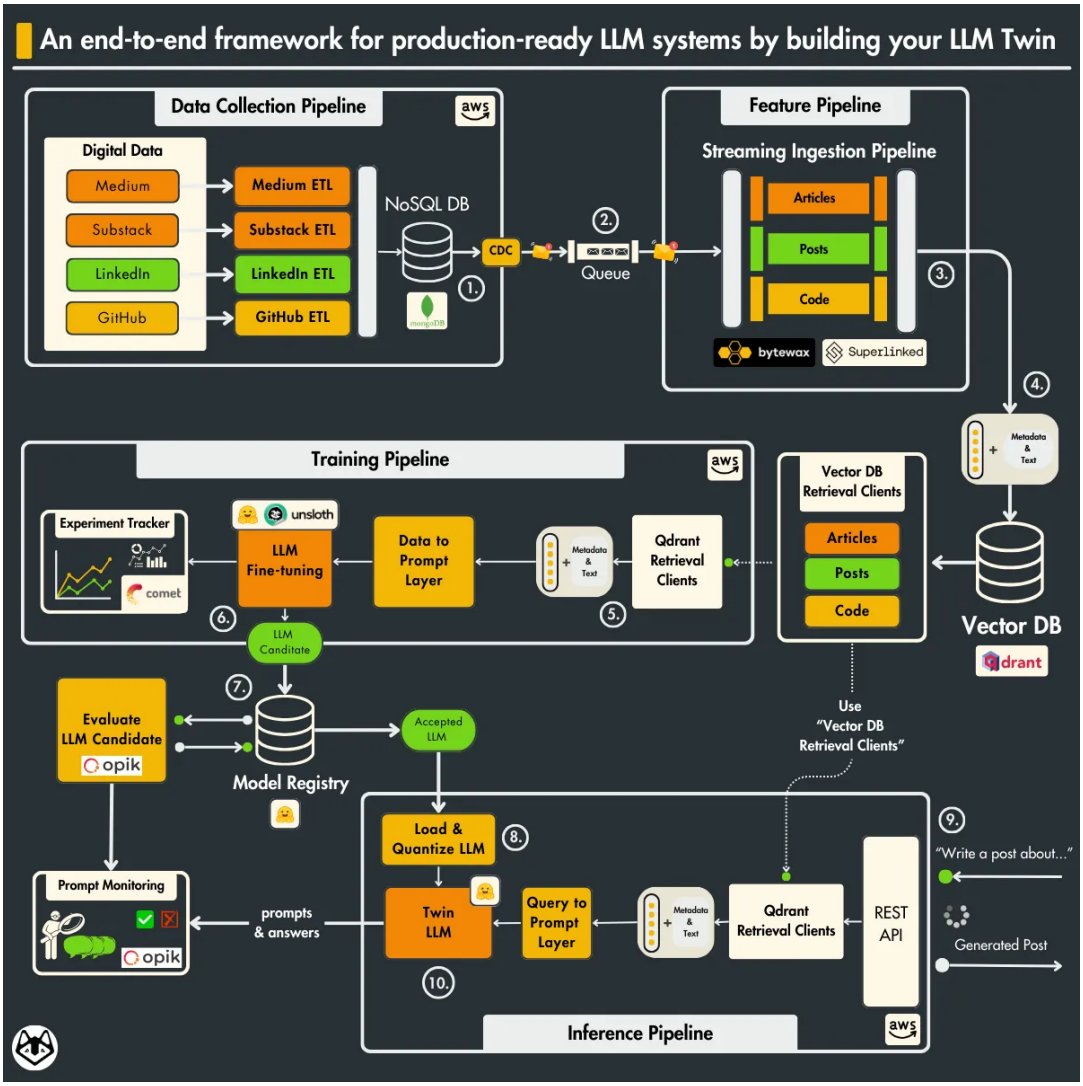
You will **learn** how to **architect** and **build a real-world LLM system** from **start** to **finish** — from **data collection** to **deployment**.

You will also **learn** to **leverage MLOps best practices**, such as experiment trackers, model registries, prompt monitoring, and versioning.

The end goal? Build and deploy your own LLM twin.

*The **architecture** of the **LLM twin** is split into **4 Python microservices**:*

1. ***The data collection pipeline*** crawls your digital data from various social media platforms. It cleans, normalizes and loads the data to a NoSQL DB through a series of ETL pipelines. Then, using the CDC pattern, it sends database changes to a queue.
2. ***The feature pipeline consumes*** messages from a queue through a Bytewax streaming pipeline. It cleans, chunks, and embeds every message and loads it to a vector DB in real-time.
3. ***The training pipeline*** creates a custom instruction dataset based on your digital data. Fine-tune an LLM using Unsloth, AWS SageMaker, and Comet ML's experiment tracker. Evaluate the LLMs using Opik and save the best model to the Hugging Face model registry.
4. ***The inference pipeline*** loads and quantizes the fine-tuned LLM from the model registry to the AWS SageMaker REST API. Enhance the prompts using RAG. Monitor the LLM using Opik. Hook the LLM Twin to a Gradio UI.



LLM Twin system architecture

Along the 4 microservices, you will learn to integrate 4 serverless tools:

- Comet ML as your experiment tracker;
- Qdrant as your vector DB;
- AWS SageMaker as your ML infrastructure;
- Opik as your prompt evaluation and monitoring tool.

Who is this for?

Audience: MLE, DE, DS, or SWE who want to learn to engineer production-ready LLM and RAG systems using LLMOps best practices.

Level: intermediate

Prerequisites: basic knowledge of Python and ML.

How will you learn?

The course contains **10 hands-on written lessons** and the **open-source code** you can *access on GitHub*, *showing how to build an end-to-end LLM system.*

Also, it includes **2 bonus lessons** on how to **improve** the **RAG system**.

You can read everything at your own pace.

→ *To get the most out of this course, we encourage you to clone and run the repository while you cover the lessons.*

Costs?

The **articles** and **code** are **completely free**. They will always remain free.

But if you plan to run the code while reading it, you must know that we use several cloud tools that might generate additional costs.

For example, AWS has a pay-as-you-go pricing plan. From our tests, it will cost you ~15$ to run the fine-tuning and inference pipelines.

We will stick to their free version for the other serverless tools, such as Qdrant, Comet, and Opik.

Lessons

The course is split into 12 lessons. Every Medium article will be its lesson:

1. **An End-to-End Framework for Production-Ready LLM Systems by Building Your LLM Twin**
2. Your Content is Gold: I Turned 3 Years of Blog Posts into an LLM Training
3. I Replaced 1000 Lines of Polling Code with 50 Lines of CDC Magic
4. SOTA Python Streaming Pipelines for Fine-tuning LLMs and RAG — in Real-Time!
5. The 4 Advanced RAG Algorithms You Must Know to Implement
6. Turning Raw Data Into Fine-Tuning Datasets
7. 8B Parameters, 1 GPU, No Problems: The Ultimate LLM Fine-tuning Pipeline
8. The Engineer's Framework for LLM & RAG Evaluation
9. Beyond Proof of Concept: Building RAG Systems That Scale
10. The Ultimate Prompt Monitoring Pipeline
11. [Bonus] Build a scalable RAG ingestion pipeline using 74.3% less code
12. [Bonus] Build Multi-Index Advanced RAG Apps

> 🔗 *Consider **checking out** the GitHub repository [1] and support us with a* ⭐

Lesson 1: End-to-end framework for production-ready LLM systems

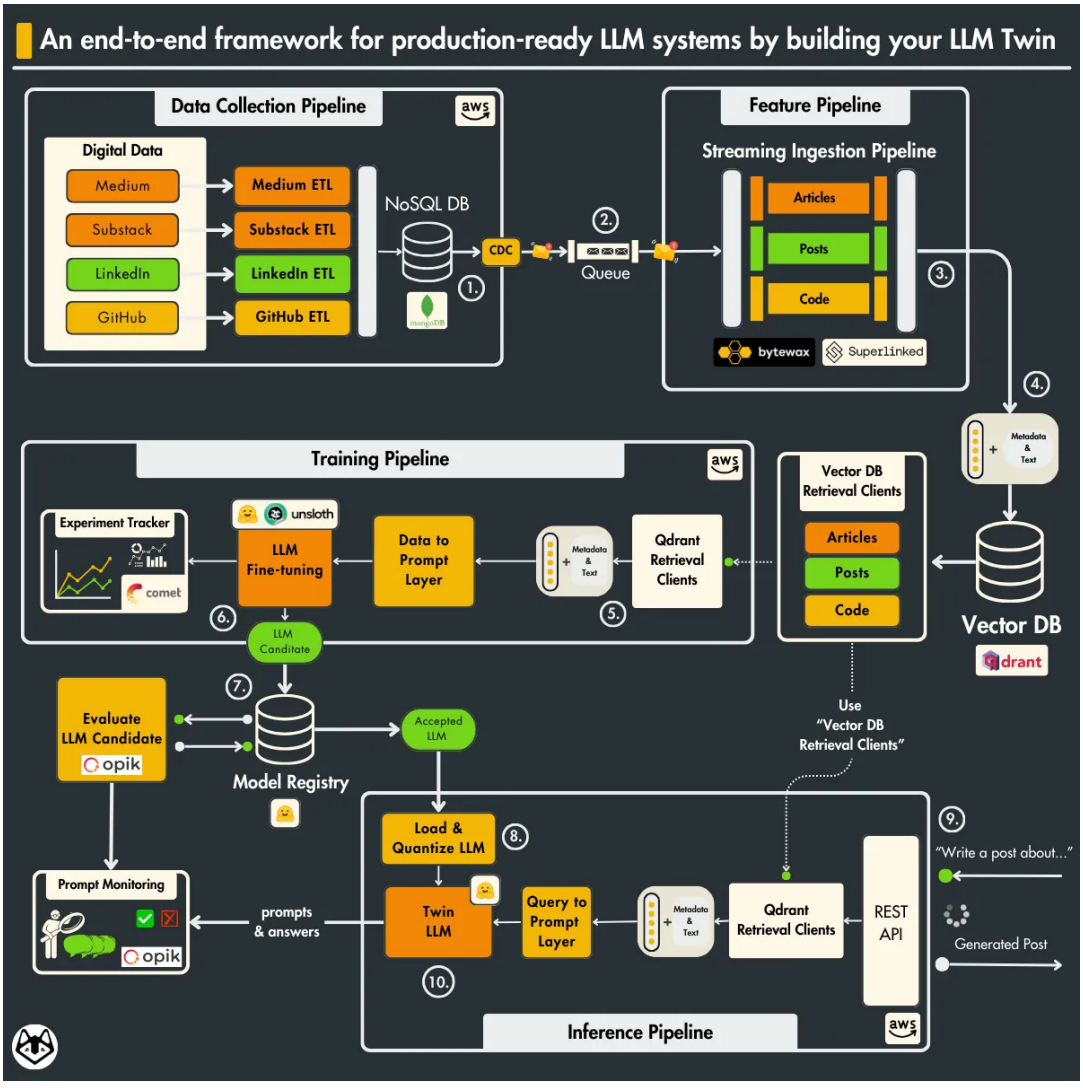
In the **first lesson,** we will **present** the **project** you will **build during the course:** *your production-ready LLM Twin/AI replica.*

Afterward, we will **explain** what **the 3-pipeline design** is and how it is applied to a standard ML system.

Ultimately, we will **dig into** the **LLM project system design.**

We will **present** all our **architectural decisions** regarding the design of *the data collection pipeline* for social media data and how we applied *the 3-pipeline architecture* to our *LLM microservices*.

In the **following lessons**, we will **examine each component's code** and learn **how** to **implement** and **deploy it** to AWS SageMaker.



The LLM Twin's system architecture

Table of Contents

> 🔗 **Check out** the code on GitHub [1] and support us with a ⭐

1. What are you going to build? The LLM twin concept

The **outcome** of this **course** is to learn to **build** your **own AI replica**. We will use an LLM to do that, hence the name of the course: ***LLM Twin: Building Your Production-Ready AI Replica***.

But what is an LLM twin?

Shortly, your LLM twin will be an AI character who writes like you, using your writing style and personality.

It will not be you. It will be your writing copycat.

More concretely, you will build an AI replica that writes social media posts or technical articles (like this one) using your own voice.

Why not directly use ChatGPT? You may ask...

When trying to generate an article or post using an LLM, the results tend to:

- be very generic and unarticulated,
- contain misinformation (due to hallucination),
- require tedious prompting to achieve the desired result.

But here is what we are going to do to fix that ↓↓↓

First, we will fine-tune an LLM on your digital data gathered from LinkedIn, Medium, Substack and GitHub.

By doing so, the LLM will align with your writing style and online personality. It will teach the LLM to talk like the online version of yourself.

Have you seen the universe of AI characters Meta released in 2024 in the Messenger app? If not, you can learn more about it here [2].

To some extent, that is what we are going to build.

But in our use case, we will focus on an LLM twin who writes social media posts or articles that reflect and articulate your voice.

For example, we can ask your LLM twin to write a LinkedIn post about LLMs. Instead of writing some generic and unarticulated post about LLMs (e.g., what ChatGPT will do), it will use your voice and style.

Secondly, we will give the LLM access to a vector DB to access external information to avoid hallucinating. Thus, we will force the LLM to write only based on concrete data.

Ultimately, in addition to accessing the vector DB for information, you can provide external links that will act as the building block of the generation process.

For example, we can modify the example above to: "Write me a 1000-word LinkedIn post about LLMs based on the article from this link: [URL]."

Excited? Let's get started 🔥

2. The 3-pipeline architecture

We all know how messy ML systems can get. That is where the 3-pipeline architecture kicks in.

The 3-pipeline design brings structure and modularity to your ML system while improving your MLOps processes.

Problem

Despite advances in MLOps tooling, transitioning from prototype to production remains challenging.

In 2022, only 54% of the models get into production. Auch.

So what happens?

Maybe the first things that come to your mind are:

- the model is not mature enough
- security risks (e.g., data privacy)
- not enough data

To some extent, these are true.

But the reality is that in many scenarios...

...the architecture of the ML system is built with research in mind, or the ML system becomes a massive monolith that is extremely hard to refactor from offline to online.

So, good SWE processes and a well-defined architecture are as crucial as using suitable tools and models with high accuracy.

Solution

→ *The 3-pipeline architecture*

Let's understand what the 3-pipeline design is.

It is a mental map that helps you simplify the development process and split your monolithic ML pipeline into 3 components:
1. the feature pipeline
2. the training pipeline

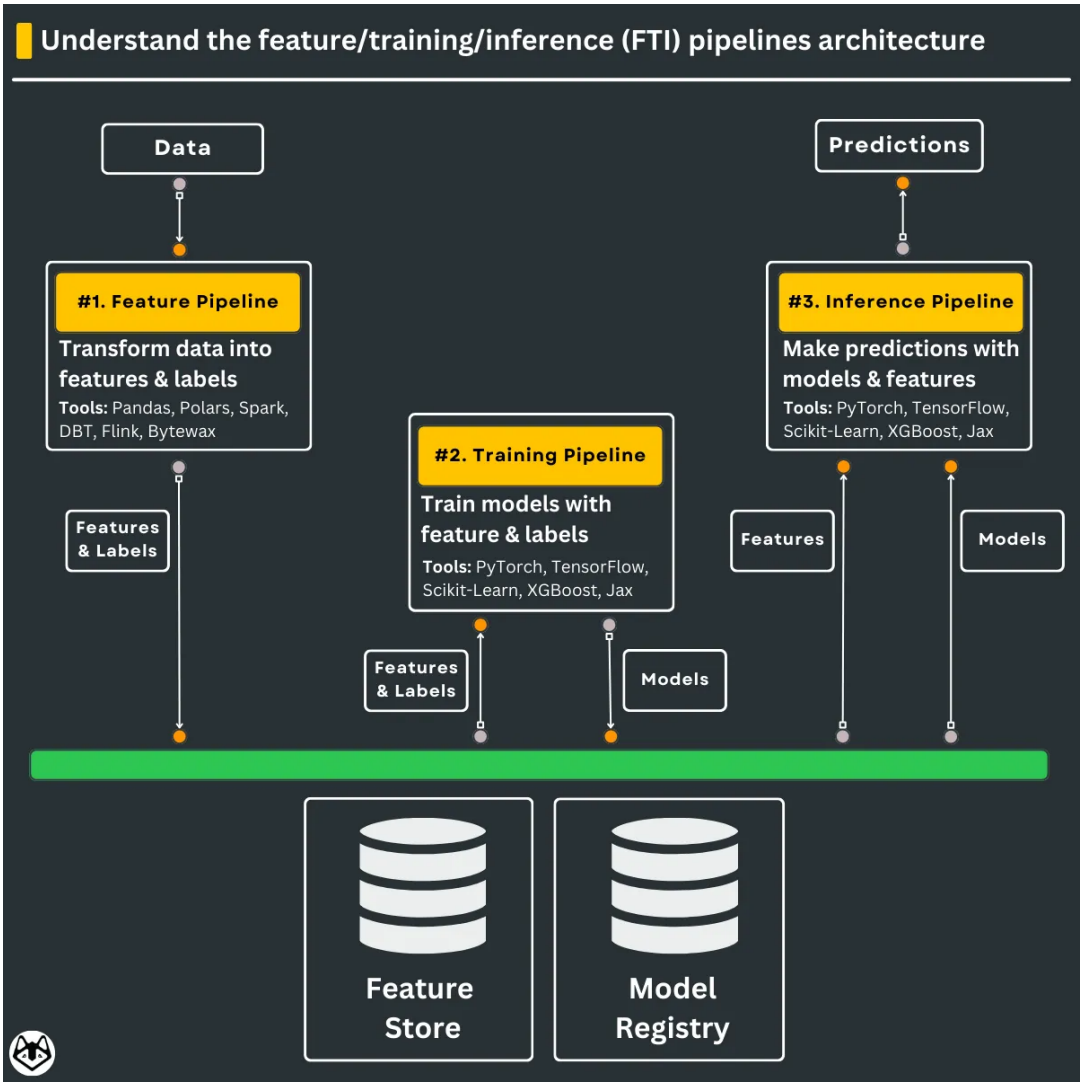
3. the inference pipeline

...also known as the Feature/Training/Inference (FTI) architecture.

#1. The **feature pipeline** transforms your data into features & labels, which are stored and versioned in a feature store. The feature store will act as the central repository of your features. That means that features can be accessed and shared only through the feature store.

#2. The **training pipeline** ingests a specific version of the features & labels from the feature store and outputs the trained model weights, which are stored and versioned inside a model registry. The models will be accessed and shared only through the model registry.

#3. The **inference pipeline** uses a given version of the features from the feature store and downloads a specific version of the model from the model registry. Its final goal is to output the predictions to a client.



The FTI architecture

This is why the 3-pipeline design is so beautiful:

- it is intuitive
- it brings structure, as on a higher level, all ML systems can be reduced to these 3 components
- it defines a transparent interface between the 3 components, making it easier for multiple teams to collaborate
- the ML system has been built with modularity in mind since the beginning
- the 3 components can easily be divided between multiple teams (if necessary)
- every component can use the best stack of technologies available for the job
- every component can be deployed, scaled, and monitored independently
- the feature pipeline can easily be either batch, streaming or both

But the most important benefit is that...

...by following this pattern, you know 100% that your ML model will move out of your Notebooks into production.

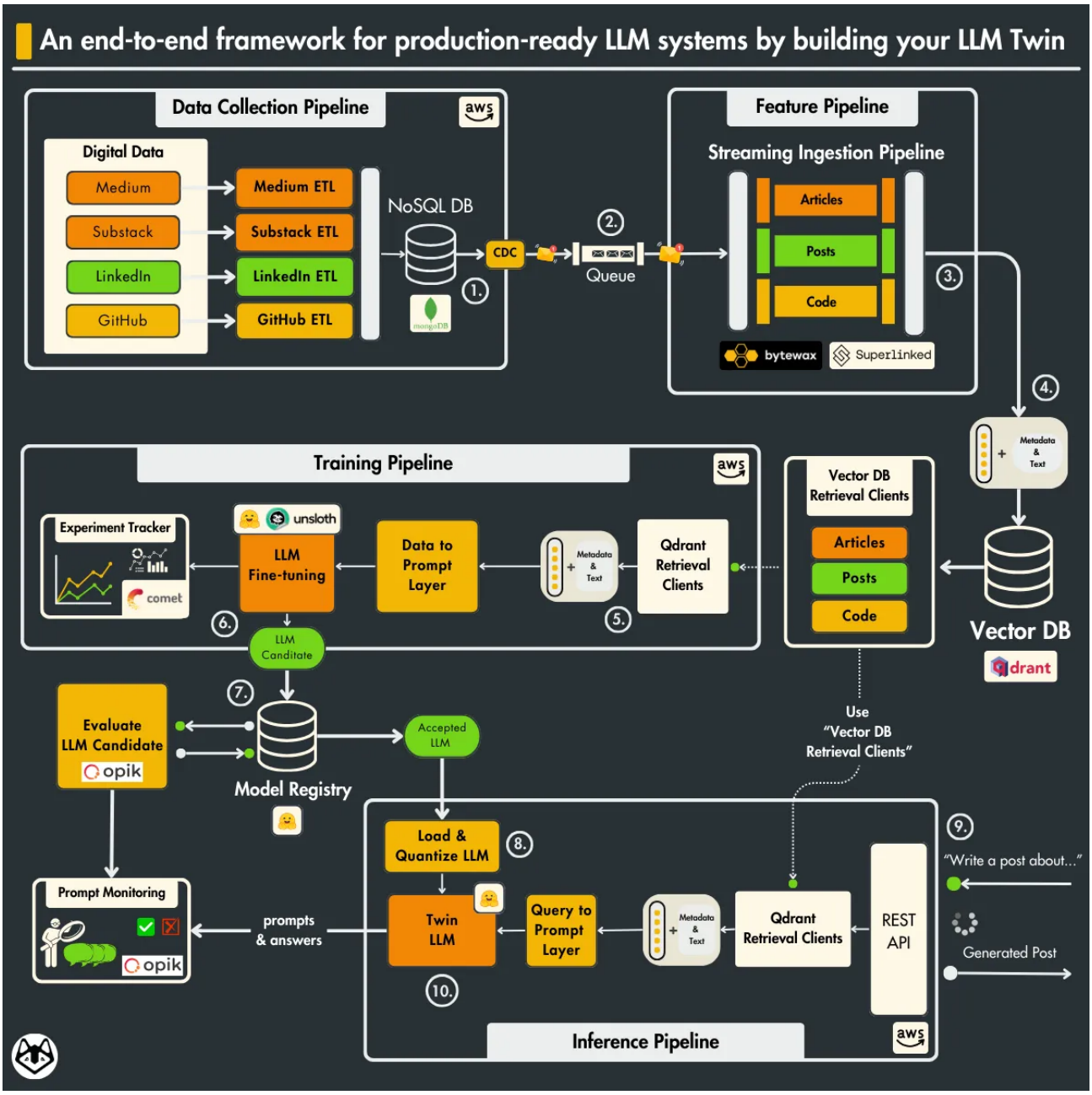
↳ If you want to *learn more about the 3-pipeline design*, I recommend this excellent article [3] written by Jim Dowling, one of the creators of the FTI architecture.

3. LLM Twin System design

Let's understand how to **apply the 3-pipeline architecture** to **our LLM system**.

The **architecture** of the **LLM twin** is split into **4 Python microservices**:

1. The data collection pipeline
2. The feature pipeline
3. The training pipeline
4. The inference pipeline



The LLM Twin's system architecture

As you can see, the data collection pipeline doesn't follow the 3-pipeline design, which is true.

It represents the data pipeline that sits before the ML system.

The data engineering team usually implements it, and its scope is to gather, clean, normalize and store the data required to build dashboards or ML models.

But let's say you are part of a small team and have to build everything yourself, from data gathering to model deployment.

Thus, we will show you how the data pipeline nicely fits and interacts with the FTI architecture.

*Now, **let's zoom in** on **each component** to understand how they work individually and interact with each other. ↓↓↓*

3.1. The data collection pipeline

Its scope is to **crawl data** for **a given user** from:

- Medium (articles)
- Substack (articles)
- LinkedIn (posts)
- GitHub (code)

As every platform is unique, we implemented a different Extract Transform Load (ETL) pipeline for each website.

🔗 1-min read on ETL pipelines [4]

However, the **baseline steps** are the **same** for **each platform**.

Thus, for each ETL pipeline, we can abstract away the following baseline steps:

- log in using your credentials
- use *selenium* to crawl your profile
- use *BeatifulSoup* to parse the HTML
- clean & normalize the extracted HTML
- save the normalized (but still raw) data to Mongo DB

Important note: We are crawling only our data, as most platforms do not allow us to access other people's data due to privacy issues. But this is perfect for us, as to build our LLM twin, we need only our own digital data.

Why Mongo DB?

We wanted a NoSQL database that quickly allows us to store unstructured data (aka text).

How will the data pipeline communicate with the feature pipeline?

We will use the **Change Data Capture (CDC) pattern** to inform the feature pipeline of any change on our Mongo DB.

🔗 1-min read on the CDC pattern [5]

To explain the CDC briefly, a watcher listens 24/7 for any CRUD operation that happens to the Mongo DB.

The watcher will issue an event informing us what has been modified. We will add that event to a RabbitMQ queue.

The feature pipeline will constantly listen to the queue, process the messages, and add them to the Qdrant vector DB.

For example, when we write a new document to the Mongo DB, the watcher creates a new event. The event is added to the RabbitMQ queue; ultimately, the feature pipeline consumes and processes it.

Doing this ensures that the Mongo DB and vector DB are constantly in sync.

With the CDC technique, we transition from a batch ETL pipeline (our data pipeline) to a streaming pipeline (our feature pipeline).

Using the CDC pattern, we avoid implementing a complex batch pipeline to compute the difference between the Mongo DB and vector DB. This approach can quickly get very slow when working with big data.

Where will the data pipeline be deployed?

The data collection pipeline and RabbitMQ service will be deployed to AWS. We will also use the freemium serverless version of Mongo DB.

3.2. The feature pipeline

The feature pipeline is **implemented using** Bytewax (a Rust streaming engine with a Python interface). Thus, in **our** specific **use case**, we will also **refer to it** as a **streaming ingestion pipeline**.

It is an **entirely different service** than the data collection pipeline.

How does it communicate with the data pipeline?

As explained above, the **feature pipeline communicates** with the **data pipeline** through a RabbitMQ **queue**.

Currently, the streaming pipeline doesn't care how the data is generated or where it comes from.

Top highlight

It knows it has to listen to a given queue, consume messages from there and process them.

By doing so, we **decouple the two components** entirely. In the future, we can easily add messages from multiple sources to the queue, and the streaming pipeline will know how to process them. The only rule is that the messages in the queue should always respect the same structure/interface.

What is the scope of the feature pipeline?

It represents the **ingestion component** of the **RAG system**.

It will **take** the **raw data** passed through the queue and:

- clean the data;
- chunk it;
- embed it using the embedding models from Superlinked;
- load it to the Qdrant vector DB.

Every type of data (post, article, code) will be **processed independently** through its own set of classes.

Even though all of them are text-based, we must clean, chunk and embed them using different strategies, as every type of data has its own particularities.

What data will be stored?

The **training pipeline** will have **access only** to the **feature store**, which, in our case, is represented by the Qdrant vector DB.

Note that a vector DB can also be used as a NoSQL DB.

*With these 2 things in mind, we will **store** in Qdrant **2 snapshots of our data:***

1. The c**leaned data** (without using vectors as indexes — store them in a NoSQL fashion).

2. The **cleaned, chunked, and embedded data** (leveraging the vector indexes of Qdrant)

The **training pipeline** needs **access** to the **data** in **both formats** as we want to fine-tune the LLM on standard and augmented prompts.

With the **cleaned data**, we will create the prompts and answers.

With the **chunked data**, we will augment the prompts (aka RAG).

Why implement a streaming pipeline instead of a batch pipeline?

There are **2 main reasons.**

The first one is that, coupled with the **CDC pattern**, it is the most **efficient** way to **sync two DBs** between each other. Otherwise, you would have to implement batch polling or pushing techniques that aren't scalable when working with big data.

Using CDC + a streaming pipeline, you process only the changes to the source DB without any overhead.

The second reason is that by doing so, your **source** and **vector DB** will **always be in sync**. Thus, you will always have access to the latest data when doing RAG.

Why Bytewax?

Bytewax is a streaming engine built in Rust that exposes a Python interface. We use Bytewax because it combines Rust's impressive speed and reliability with the ease of use and ecosystem of Python. It is incredibly light, powerful, and easy for a Python developer.

Where will the feature pipeline be deployed?

The feature pipeline will be deployed to AWS. We will also use the freemium serverless version of Qdrant.

3.3. The training pipeline

How do we have access to the training features?

As highlighted in section 3.2, all the **training data** will be **accessed** from the **feature store**. In our case, the feature store is the **Qdrant vector DB** that contains:

- the cleaned digital data from which we will create prompts & answers;
- we will use the chunked & embedded data for RAG to augment the cleaned data.

We will implement a different vector DB retrieval client for each of our main types of data (posts, articles, code).

We must do this separation because we must preprocess each type differently before querying the vector DB, as each type has unique properties.

Also, we will add custom behavior for each client based on what we want to query from the vector DB. But more on this in its dedicated lesson.

What will the training pipeline do?

The training pipeline contains a **data-to-prompt layer** that will preprocess the data retrieved from the vector DB into prompts.

It will also contain an **LLM fine-tuning module** that inputs a HuggingFace dataset and uses QLoRA to fine-tune a given LLM (e.g., Mistral). By using HuggingFace, we can easily switch between different LLMs so we won't focus too much on any specific LLM.

All the experiments will be logged into Comet ML's **experiment tracker**.

We will use a bigger LLM (e.g., GPT4) to **evaluate** the results of our fine-tuned LLM. These results will be logged into Comet's experiment tracker.

Where will the production candidate LLM be stored?

We will compare multiple experiments, pick the best one, and issue an LLM production candidate for the model registry.

After, we will inspect the LLM production candidate manually using Comet's prompt monitoring dashboard. If this final manual check passes, we will flag the LLM from the model registry as accepted.

A CI/CD pipeline will trigger and deploy the new LLM version to the inference pipeline.

Where will the training pipeline be deployed?

The training pipeline will be deployed to AWS SageMaker.

AWS SageMaker is a solution for training and deploying ML models. It makes scaling your operation easy while you can focus on building.

Also, we will use the freemium version of Comet ML for the following:

- experiment tracker;

- model registry;
- prompt monitoring.

3.4. The inference pipeline

The inference pipeline is the **final component** of the **LLM system**. It is the one the **clients** will **interact with**.

It will be **wrapped** under a **REST API**. The clients can call it through HTTP requests, similar to your experience with ChatGPT or similar tools.

How do we access the features?

To access the feature store, we will use the same Qdrant vector DB retrieval clients as in the training pipeline.

In this case, we will need the feature store to access the chunked data to do RAG.

How do we access the fine-tuned LLM?

The fine-tuned LLM will always be downloaded from the model registry based on its tag (e.g., accepted) and version (e.g., v1.0.2, latest, etc.).

How will the fine-tuned LLM be loaded?

Here we are in the inference world.

Thus, we want to optimize the LLM's speed and memory consumption as much as possible. That is why, after downloading the LLM from the model registry, we will quantize it.

What are the components of the inference pipeline?

The first one is the **retrieval client** used to access the vector DB to do RAG. This is the same module as the one used in the training pipeline.

After we have a **query to prompt the layer,** that will map the prompt and

retrieved documents from Qdrant into a prompt.

After the LLM generates its answer, we will log it to Comet's **prompt monitoring dashboard** and return it to the clients.

For example, the client will request the inference pipeline to:

"Write a 1000-word LinkedIn post about LLMs," and the inference pipeline will go through all the steps above to return the generated post.

Where will the inference pipeline be deployed?

The inference pipeline will be deployed to AWS SageMaker.

AWS SageMaker also offers autoscaling solutions and a nice dashboard to monitor all the production environment resources.

Conclusion

This is the 1st article of the *LLM Twin: Building Your Production-Ready AI Replica* free course.

In this lesson, we presented what **you will build** during the course.

After we briefly discussed how to design ML systems using **the 3-pipeline design**.

other:

1. The data collection pipeline
2. The feature pipeline
3. The training pipeline
4. The inference pipeline

In **Lesson 2**, we will dive deeper into the **data collection pipeline**, learn how to implement crawlers for various social media platforms, clean the gathered data, and store it in a MongoDB NoSQL database.

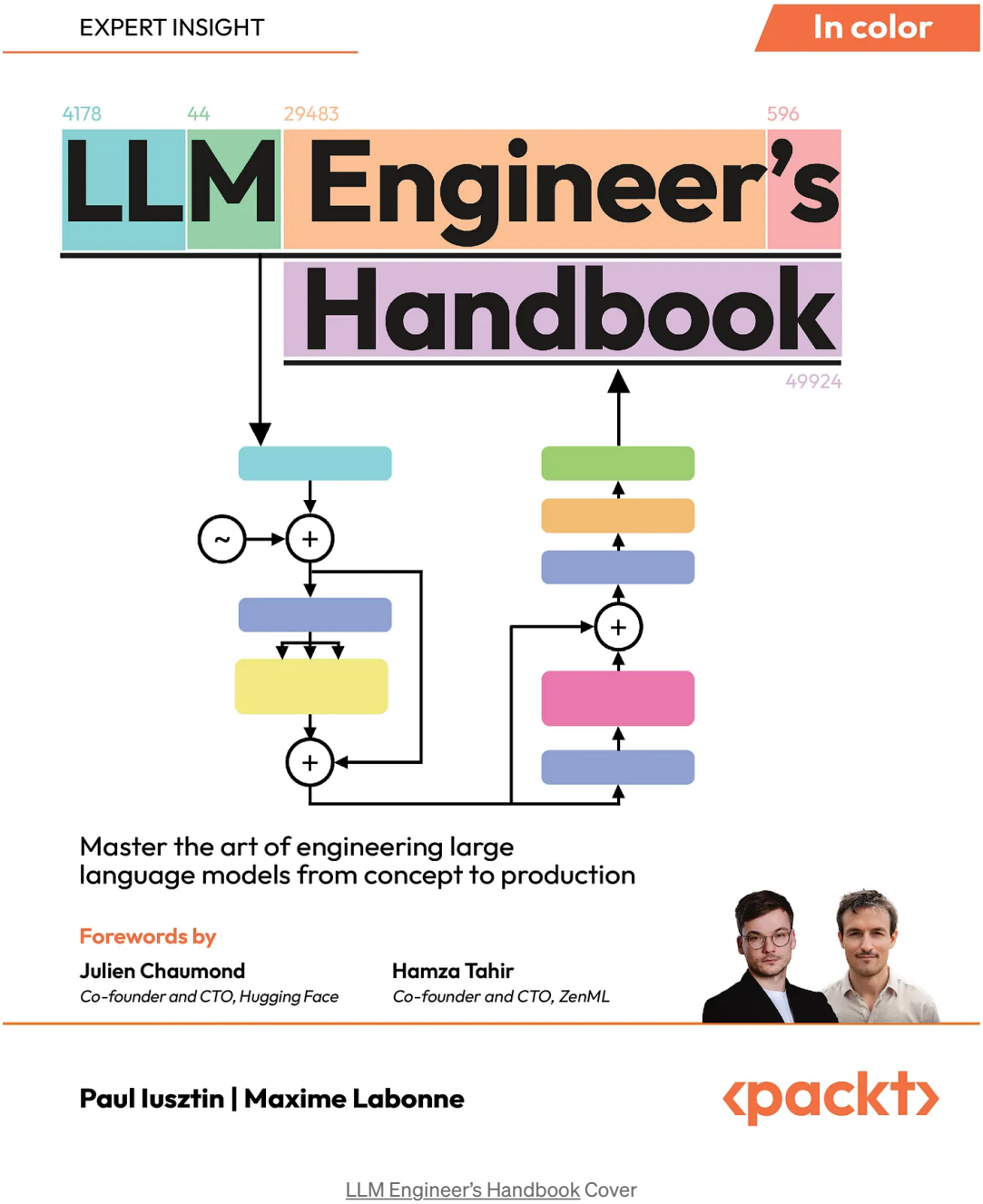
> 🔗 *Consider **checking out** the GitHub repository [1] and support us with a* ⭐

Our *LLM Engineer's Handbook* **inspired** the **open-source LLM Twin course**.

Consider supporting our work by getting our book to **learn** a **complete framework** for **building and deploying production LLM & RAG systems** — from data to deployment.

Perfect for practitioners who want **both theory** and **hands-on** expertise by connecting the dots between DE, research, MLE and MLOps:

→ Buy the LLM Engineer's Handbook (on Amazon or Packt)



LLM Engineer's Handbook Cover

Enjoyed This Article?

*Join **Decoding ML** for **battle-tested content** on designing, coding, and deploying **production-grade LLM, RecSys & MLOps systems.** Every week, a new project ↓*

Decoding ML Newsletter | Paul Iusztin | Substack

Join for battle-tested content on designing, coding, and deploying production-grade ML & MLOps systems. Every week. For...

decodingml.substack.com

References

Literature

[1] Your LLM Twin Course — GitHub Repository (2024), Decoding ML GitHub Organization

[2] Introducing new AI experiences from Meta (2023), Meta

[3] Jim Dowling, From MLOps to ML Systems with Feature/Training/Inference Pipelines (2023), Hopsworks

[4] Extract Transform Load (ETL), Databricks Glossary

[5] Daniel Svonava and Paolo Perrone, Understanding the different Data Modality / Types (2023), Superlinked

Images

If not otherwise stated, all images are created by the author.

Generative Ai | Large Language Models | Mlops | Artificial Intelligence | Machine Learning

Published in Decoding ML

1.4K Followers · Last published Nov 30, 2024

Follow

Battle-tested content on designing, coding, and deploying production-grade ML & MLOps systems. The hub for continuous learning on ML system design, ML engineering, MLOps, large language models (LLMs), and computer vision (CV).

Written by Paul Iusztin

5.4K Followers · 246 Following

Follow

Senior ML & MLOps Engineer • Founder @ Decoding ML ~ Content about building production-grade ML/AI systems • DML Newsletter: https://decodingml.substack.com

Responses (14)

What are your thoughts?

Respond

Adijsad he
Mar 20, 2024 (edited)

What if I want to feed pdfs? Can you also add an ETL to extract pdfs in a clean and clear way without losing information present in the tables, code, equation etc...?

18 2 replies Reply

M K Pavan Kumar
Mar 20, 2024

This is awesome!!

17 1 reply Reply

Daniel García
Mar 20, 2024

Great content!!

13 1 reply Reply

See all responses

More from Paul Iusztin and Decoding ML

In Decoding ML by Paul Iusztin

Your Content is Gold: I Turned 3 Years of Blog Posts into an LLM

A practical guide to building custom instruction datasets for fine-tuning LLMs

Nov 18, 2024 96

In Decoding ML by Paul Iusztin

The 4 Advanced RAG Algorithms You Must Know to Implement

Implement from scratch 4 advanced RAG methods to optimize your retrieval and post-

May 4, 2024 1.8K 15

In Decoding ML by Paul Iusztin

SOTA Python Streaming Pipelines for Fine-tuning LLMs and RAG—in

Use a Python streaming engine to populate a feature store from 4+ data sources

Apr 20, 2024 835 3

In Decoding ML by Paul Iusztin

8B Parameters, 1 GPU, No Problems: The Ultimate LLM Fine-

Master production-ready fine-tuning with AWS SageMaker, Unsloth, and MLOps best

Nov 18, 2024 110

See all from Paul Iusztin See all from Decoding ML

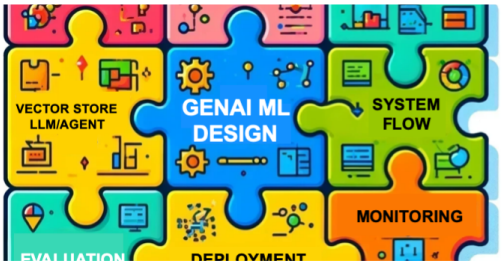
Recommended from Medium

In Decoding ML by Paul Iusztin

The 4 Advanced RAG Algorithms You Must Know to Implement

Implement from scratch 4 advanced RAG methods to optimize your retrieval and post-

May 4, 2024 1.8K 15



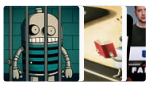
In TDS Archive by Jack Chih-Hsu Lin

Mastering GenAI ML System Design Interview: Principles &

Strategies and Insights from Both Sides of the Interview Table

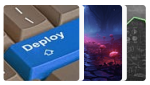
May 17, 2024 491 6

Lists



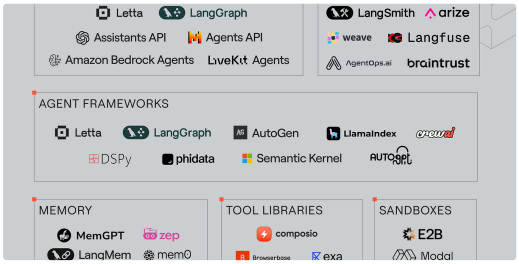
AI Regulation
6 stories · 691 saves

Natural Language Processing
1950 stories · 1599 saves



Predictive Modeling w/ Python
20 stories · 1833 saves

Practical Guides to Machine Learning
10 stories · 2209 saves



Vipra Singh

AI Agents: Introduction (Part-1)

Discover AI agents, their design, and real-world applications.

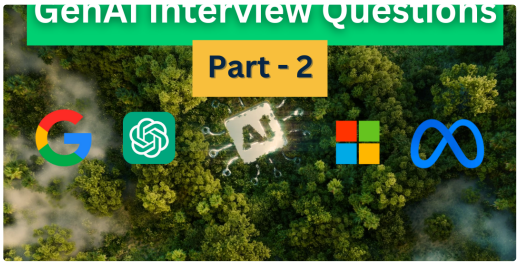
Feb 3 520 14

In Towards AI by Alex Punnen

Explaining Transformers as Simple as Possible through a Small

And understanding Vector Transformations and Vectorizations

6d ago 668 14

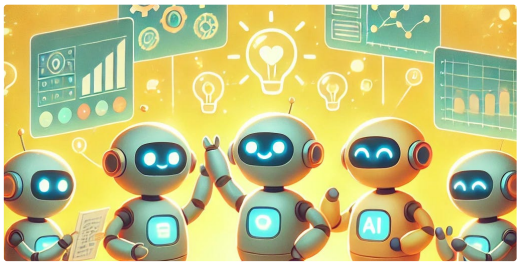


In Artificial Intelligence in Plain E by Simranjeet Si

Generative AI Interview Questions [LLM] Top 20—Part : 2

Crack you next Generative AI or LLM Interview with these Series of GenAI Interview

Aug 23, 2024 136



In Data Science Collective by Shuai Guo

Multi-Agent System Powered by Large Language Models: An

What is it, how to innovate with it, and what hidden pitfalls to avoid

Feb 13 272 1

See more recommendations