Readers can expect to gain a deeper understanding of how to collect and prepare data for LLMs, fine-tune models for specific tasks, optimize inference performance, and implement RAG pipelines. They will learn how to evaluate LLM performance, align models with human preferences, and deploy LLM-based applications. The book also covers essential MLOps principles and practices, enabling readers to build scalable, reproducible, and robust LLM applications.

Who this book is for

This book is intended for a wide range of technology professionals and enthusiasts interested in the practical applications of LLMs. It's ideal for software engineers aiming to transition into AI projects. While some familiarity with software development is beneficial, the book explains many concepts from the ground up, making it accessible even to those who are new to AI and machine learning.

For those already working with machine learning , this book will enhance your skills in implementing and deploying LLM-based systems. We provide a deep dive into the fundamentals of MLOps, guiding you through the process of creating a minimum viable product using an open-source LLM to solve real-world problems.

What this book covers

Chapter 1, *Understanding the LLM Twin Concept and Architecture*, introduces the LLM Twin project, which is used throughout the book as an end-to-end example of a production-level LLM application, and defines the FTI architecture for building scalable ML systems and applies it to the LLM Twin use case.

Chapter 2, *Tooling and Installation*, presents Python, MLOps, and cloud tools used to build real-world LLM applications, such as an orchestrator, experiment tracker, prompt monitoring and LLM evaluation tool. It shows how to use and install them locally for testing and development.

Chapter 3, *Data Engineering*, shows the implementation of a data collection pipeline that scrapes multiple sites, such as Medium, GitHub and Substack and stores the raw data in a data warehouse. It emphasizes collecting raw data from dynamic sources over static datasets for real-world ML applications.

Chapter 4, *RAG Feature Pipeline*, introduces RAG fundamental concepts, such as embeddings, the vanilla RAG framework, vector databases, and how to optimize RAG applications. It applies the RAG theory by architecting and implementing LLM Twin's RAG feature pipeline using software best practices.

Chapter 5, *Supervised Fine-Tuning*, explores the process of refining pre-trained language models for specific tasks using instruction-answer pairs. It covers creating high-quality datasets, implementing fine-tuning techniques like full fine-tuning, LoRA, and QLoRA, and provides a practical demonstration of fine-tuning a Llama 3.1 8B model on a custom dataset.

Chapter 6, *Fine-Tuning with Preference Alignment*, introduces techniques for aligning language models with human preferences, focusing on **Direct Preference Optimization (DPO)**. It covers creating custom preference datasets, implementing DPO, and provides a practical demonstration of aligning the TwinLlama-3.1-8B model using the Unsloth library.

Chapter 7, *Evaluating LLMs*, details various methods for assessing the performance of language models and LLM systems. It introduces general-purpose and domain-specific evaluations and discusses popular benchmarks. The chapter includes a practical evaluation of the TwinLlama-3.1-8B model using multiple criteria.

Chapter 8, *Inference Optimization*, covers key optimization strategies such as speculative decoding, model parallelism, and weight quantization. It discusses how to improve inference speed, reduce latency, and minimize memory usage, introducing popular inference engines and comparing their features.

Chapter 9, *RAG Inference Pipeline*, explores advanced RAG techniques by implementing methods such as self-query, reranking, and filtered vector search from scratch. It covers designing and implementing the LLM Twin's RAG inference pipeline and a custom retrieval module similar to what you see in popular frameworks such as LangChain.

Chapter 10, *Inference Pipeline Deployment*, introduces ML deployment strategies, such as online, asynchronous and batch inference, which will help in architecting and deploying the LLM Twin fine-tuned model to AWS SageMaker and building a FastAPI microservice to expose the RAG inference pipeline as a RESTful API.

Chapter 11, *MLOps and LLMOps*, presents what LLMOps is, starting with its roots in DevOps and MLOps. This chapter explains how to deploy the LLM Twin project to the cloud, such as the ML pipelines to AWS and shows how to containerize the code using Docker and build a CI/CD/CT pipeline. It also adds a prompt monitoring layer on top of LLM Twin's inference pipeline.

Appendix, *MLOps Principles*, covers the six MLOps principles used to build scalable, reproducible, and robust ML applications.

To get the most out of this book

To maximize your learning experience, you are expected to have, at the very least, a foundational understanding of software development principles and practices. Familiarity with Python programming is particularly beneficial, as the book's examples and code snippets are predominantly in Python. While prior experience with machine learning concepts is advantageous, it is not strictly necessary, as the book provides explanations for many fundamental AI and ML concepts. However, you should be comfortable with basic data structures, algorithms, and have some experience working with APIs and cloud services.

Familiarity with version control systems like Git is assumed, as this book has a GitHub repository for code examples. While this book is designed to be accessible to those who are new to AI and LLMs, if you have some background in these areas, you will find it easier to grasp the more advanced concepts and techniques we present.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/LLM-Engineers-Handbook`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781836200079`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "In the `format_samples` function, we apply the Alpaca chat template to each individual message."

A block of code is set as follows:

```
def format_samples(example):
    example["prompt"] = alpaca_template.format(example["prompt"])
    example["chosen"] = example['chosen'] + EOS_TOKEN
    example["rejected"] = example['rejected'] + EOS_TOKEN
    return {"prompt": example["prompt"], "chosen": example["chosen"],
"rejected": example["rejected"]}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def format_samples(example):
    example["prompt"] = alpaca_template.format(example["prompt"])
    example["chosen"] = example['chosen'] + EOS_TOKEN
    example["rejected"] = example['rejected'] + EOS_TOKEN
    return {"prompt": example["prompt"], "chosen": example["chosen"],
"rejected": example["rejected"]}
```

Any command-line input or output is written as follows:

```
poetry install --without aws
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "To do so, go to the **Settings** tab at the top of the forked repository in GitHub. In the left panel, in the **Security** section, click on the **Secrets and Variables** toggle and, finally, click on **Actions**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *LLM Engineer's Handbook, First Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

`https://packt.link/free-ebook/9781836200079`

2. Submit your proof of purchase.
3. That’s it! We’ll send your free PDF and other benefits to your email directly.

1

Understanding the LLM Twin Concept and Architecture

By the end of this book, we will have walked you through the journey of building an end-to-end **large language model (LLM)** product. We firmly believe that the best way to learn about LLMs and production **machine learning (ML)** is to get your hands dirty and build systems. This book will show you how to build an LLM Twin, an AI character that learns to write like a particular person by incorporating its style, voice, and personality into an LLM. Using this example, we will walk you through the complete ML life cycle, from data gathering to deployment and monitoring. Most of the concepts learned while implementing your LLM Twin can be applied in other LLM-based or ML applications.

When starting to implement a new product, from an engineering point of view, there are three planning steps we must go through before we start building. First, it is critical to understand the problem we are trying to solve and what we want to build. In our case, what exactly is an LLM Twin, and why build it? This step is where we must dream and focus on the "Why." Secondly, to reflect a real-world scenario, we will design the first iteration of a product with minimum functionality. Here, we must clearly define the core features required to create a working and valuable product. The choices are made based on the timeline, resources, and team's knowledge. This is where we bridge the gap between dreaming and focusing on what is realistic and eventually answer the following question: "What are we going to build?".

Finally, we will go through a system design step, laying out the core architecture and design choices used to build the LLM system. Note that the first two components are primarily product-related, while the last one is technical and focuses on the "How."

These three steps are natural in building a real-world product. Even if the first two do not require much ML knowledge, it is critical to go through them to understand "how" to build the product with a clear vision. In a nutshell, this chapter covers the following topics:

- Understanding the LLM Twin concept
- Planning the MVP of the LLM Twin product
- Building ML systems with feature/training/inference pipelines
- Designing the system architecture of the LLM Twin

By the end of this chapter, you will have a clear picture of what you will learn to build throughout the book.

Understanding the LLM Twin concept

The first step is to have a clear vision of what we want to create and why it's valuable to build it. The concept of an LLM Twin is new. Thus, before diving into the technical details, it is essential to understand what it is, what we should expect from it, and how it should work. Having a solid intuition of your end goal makes it much easier to digest the theory, code, and infrastructure presented in this book.

What is an LLM Twin?

In a few words, an LLM Twin is an AI character that incorporates your writing style, voice, and personality into an LLM, which is a complex AI model. It is a digital version of yourself *projected* into an LLM. Instead of a generic LLM trained on the whole internet, an LLM Twin is fine-tuned on yourself. Naturally, as an ML model reflects the data it is trained on, this LLM will incorporate your writing style, voice, and personality. We intentionally used the word "projected." As with any other projection, you lose a lot of information along the way. Thus, this LLM will not *be you*; it will copy the side of you reflected in the data it was trained on.

It is essential to understand that an LLM reflects the data it was trained on. If you feed it Shakespeare, it will start writing like him. If you train it on Billie Eilish, it will start writing songs in her style. This is also known as style transfer. This concept is prevalent in generating images, too. For example, let's say you want to create a cat image using Van Gogh's style. We will leverage the style transfer strategy, but instead of choosing a personality, we will do it on our own persona.

To adjust the LLM to a given style and voice along with fine-tuning, we will also leverage various advanced **retrieval-augmented generation** (**RAG**) techniques to condition the autoregressive process with previous embeddings of ourselves.

We will explore the details in *Chapter 5* on fine-tuning and *Chapters 4* and *9* on RAG, but for now, let's look at a few examples to intuitively understand what we stated previously.

Here are some scenarios of what you can fine-tune an LLM on to become your twin:

- **LinkedIn posts and X threads**: Specialize the LLM in writing social media content.
- **Messages with your friends and family**: Adapt the LLM to an unfiltered version of yourself.
- **Academic papers and articles**: Calibrate the LLM in writing formal and educative content.
- **Code**: Specialize the LLM in implementing code as you would.

All the preceding scenarios can be reduced to one core strategy: collecting your digital data (or some parts of it) and feeding it to an LLM using different algorithms. Ultimately, the LLM reflects the voice and style of the collected data. Easy, right?

Unfortunately, this raises many technical and moral issues. First, on the technical side, how can we access this data? Do we have enough digital data to project ourselves into an LLM? What kind of data would be valuable? Secondly, on the moral side, is it OK to do this in the first place? Do we want to create a copycat of ourselves? Will it write using our voice and personality, or just try to replicate it?

Remember that the role of this section is not to bother with the "What" and "How" but with the "Why." Let's understand why it makes sense to have your LLM Twin, why it can be valuable, and why it is morally correct if we frame the problem correctly.

Why building an LLM Twin matters

As an engineer (or any other professional career), building a personal brand is more valuable than a standard CV. The biggest issue with creating a personal brand is that writing content on platforms such as LinkedIn, X, or Medium takes a lot of time. Even if you enjoy writing and creating content, you will eventually run out of inspiration or time and feel like you need assistance. We don't want to transform this section into a pitch, but we have to understand the scope of this product/project clearly.

We want to build an LLM Twin to write personalized content on LinkedIn, X, Instagram, Substack, and Medium (or other blogs) using our style and voice. It will not be used in any immoral scenarios, but it will act as your writing co-pilot. Based on what we will teach you in this book, you can get creative and adapt it to various use cases, but we will focus on the niche of generating social media content and articles. Thus, instead of writing the content from scratch, we can feed the skeleton of our main idea to the LLM Twin and let it do the grunt work.

Ultimately, we will have to check whether everything is correct and format it to our liking (more on the concrete features in the *Planning the MVP of the LLM Twin product* section). Hence, we project ourselves into a content-writing LLM Twin that will help us automate our writing process. It will likely fail if we try to use this particular LLM in a different scenario, as this is where we will specialize the LLM through fine-tuning, prompt engineering, and RAG.

So, why does building an LLM Twin matter? It helps you do the following:

- Create your brand
- Automate the writing process
- Brainstorm new creative ideas

What's the difference between a co-pilot and an LLM Twin?

A co-pilot and digital twin are two different concepts that work together and can be combined into a powerful solution:

- The co-pilot is an AI assistant or tool that augments human users in various programming, writing, or content creation tasks.
- The twin serves as a 1:1 digital representation of a real-world entity, often using AI to bridge the gap between the physical and digital worlds. For instance, an LLM Twin is an LLM that learns to mimic your voice, personality, and writing style.

With these definitions in mind, a writing and content creation AI assistant who writes like you is your LLM Twin co-pilot.

Also, it is critical to understand that building an LLM Twin is entirely moral. The LLM will be fine-tuned only on our personal digital data. We won't collect and use other people's data to try to impersonate anyone's identity. We have a clear goal in mind: creating our personalized writing copycat. Everyone will have their own LLM Twin with restricted access.

Of course, many security concerns are involved, but we won't go into that here as it could be a book in itself.

Why not use ChatGPT (or another similar chatbot)?

This subsection will refer to using ChatGPT (or another similar chatbot) just in the context of generating personalized content.

We have already provided the answer. ChatGPT is not *personalized* to your writing style and voice. Instead, it is very generic, unarticulated, and wordy. Maintaining an original voice is critical for long-term success when building your brand. Thus, directly using ChatGPT or Gemini will not yield the most optimal results. Even if you are OK with sharing impersonalized content, mindlessly using ChatGPT can result in the following:

- **Misinformation due to hallucination**: Manually checking the results for hallucinations or using third-party tools to evaluate your results is a tedious and unproductive experience.
- **Tedious manual prompting**: You must manually craft your prompts and inject external information, which is a tiresome experience. Also, the generated answers will be hard to replicate between multiple sessions as you don't have complete control over your prompts and injected data. You can solve part of this problem using an API and a tool such as LangChain, but you need programming experience to do so.

From our experience, if you want high-quality content that provides real value, you will spend more time debugging the generated text than writing it yourself.

The key of the LLM Twin stands in the following:

- What data we collect
- How we preprocess the data
- How we feed the data into the LLM
- How we chain multiple prompts for the desired results
- How we evaluate the generated content

The LLM itself is important, but we want to highlight that using ChatGPT's web interface is exceptionally tedious in managing and injecting various data sources or evaluating the outputs. The solution is to build an LLM system that encapsulates and automates all the following steps (manually replicating them each time is not a long-term and feasible solution):

- Data collection
- Data preprocessing

- Data storage, versioning, and retrieval
- LLM fine-tuning
- RAG
- Content generation evaluation

Note that we never said not to use OpenAI's GPT API, just that the LLM framework we will present is LLM-agnostic. Thus, if it can be manipulated programmatically and exposes a fine-tuning interface, it can be integrated into the LLM Twin system we will learn to build. The key to most successful ML products is to be data-centric and make your architecture model-agnostic. Thus, you can quickly experiment with multiple models on your specific data.

Planning the MVP of the LLM Twin product

Now that we understand what an LLM Twin is and why we want to build it, we must clearly define the product's features. In this book, we will focus on the first iteration, often labeled the **minimum viable product** (**MVP**), to follow the natural cycle of most products. Here, the main objective is to align our ideas with realistic and doable business objectives using the available resources to produce the product. Even as an engineer, as you grow up in responsibilities, you must go through these steps to bridge the gap between the business needs and what can be implemented.

What is an MVP?

An MVP is a version of a product that includes just enough features to draw in early users and test the viability of the product concept in the initial stages of development. Usually, the purpose of the MVP is to gather insights from the market with minimal effort.

An MVP is a powerful strategy because of the following reasons:

- **Accelerated time-to-market**: Launch a product quickly to gain early traction
- **Idea validation**: Test it with real users before investing in the full development of the product
- **Market research**: Gain insights into what resonates with the target audience
- **Risk minimization**: Reduces the time and resources needed for a product that might not achieve market success

Sticking to the *V* in MVP is essential, meaning the product must be *viable*. The product must provide an end-to-end user journey without half-implemented features, even if the product is minimal. It must be a working product with a good user experience that people will love and want to keep using to see how it evolves to its full potential.

Defining the LLM Twin MVP

As a thought experiment, let's assume that instead of building this project for this book, we want to make a real product. In that case, what are our resources? Well, unfortunately, not many:

- We are a team of three people with two ML engineers and one ML researcher
- Our laptops
- Personal funding for computing, such as training LLMs
- Our enthusiasm

As you can see, we don't have many resources. Even if this is just a thought experiment, it reflects the reality for most start-ups at the beginning of their journey. Thus, we must be very strategic in defining our LLM Twin MVP and what features we want to pick. Our goal is simple: we want to maximize the product's value relative to the effort and resources poured into it.

To keep it simple, we will build the features that can do the following for the LLM Twin:

- Collect data from your LinkedIn, Medium, Substack, and GitHub profiles
- Fine-tune an open-source LLM using the collected data
- Populate a vector database (DB) using our digital data for RAG
- Create LinkedIn posts leveraging the following:
    - User prompts
    - RAG to reuse and reference old content
    - New posts, articles, or papers as additional knowledge to the LLM
- Have a simple web interface to interact with the LLM Twin and be able to do the following:
    - Configure your social media links and trigger the collection step
    - Send prompts or links to external resources

That will be the LLM Twin MVP. Even if it doesn't sound like much, remember that we must make this system cost effective, scalable, and modular.

Even if we focus only on the core features of the LLM Twin defined in this section, we will build the product with the latest LLM research and best software engineering and MLOps practices in mind. We aim to show you how to engineer a cost-effective and scalable LLM application.

Until now, we have examined the LLM Twin from the users' and businesses' perspectives. The last step is to examine it from an engineering perspective and define a development plan to understand how to solve it technically. From now on, the book's focus will be on the implementation of the LLM Twin.

Building ML systems with feature/training/inference pipelines

Before diving into the specifics of the LLM Twin architecture, we must understand an ML system pattern at the core of the architecture, known as the **feature/training/inference (FTI)** architecture. This section will present a general overview of the FTI pipeline design and how it can structure an ML application.

Let's see how we can apply the FTI pipelines to the LLM Twin architecture.

The problem with building ML systems

Building production-ready ML systems is much more than just training a model. From an engineering point of view, training the model is the most straightforward step in most use cases. However, training a model becomes complex when deciding on the correct architecture and hyperparameters. That's not an engineering problem but a research problem.

At this point, we want to focus on how to design a production-ready architecture. Training a model with high accuracy is extremely valuable, but just by training it on a static dataset, you are far from deploying it robustly. We have to consider how to do the following:

- Ingest, clean, and validate fresh data
- Training versus inference setups
- Compute and serve features in the right environment
- Serve the model in a cost-effective way
- Version, track, and share the datasets and models
- Monitor your infrastructure and models
- Deploy the model on a scalable infrastructure
- Automate the deployments and training

These are the types of problems an ML or MLOps engineer must consider, while the research or data science team is often responsible for training the model.

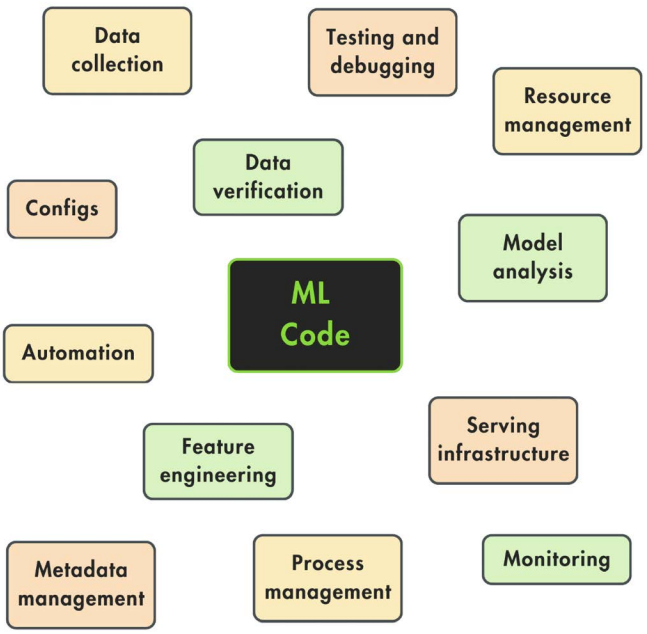

Figure 1.1: Common elements from an ML system

The preceding figure shows all the components the Google Cloud team suggests that a mature ML and MLOps system requires. Along with the ML code, there are many moving pieces. The rest of the system comprises configuration, automation, data collection, data verification, testing and debugging, resource management, model analysis, process and metadata management, serving infrastructure, and monitoring. The point is that there are many components we must consider when productionizing an ML model.

Thus, the critical question is this: How do we connect all these components into a single homogenous system? We must create a boilerplate for clearly designing ML systems to answer that question.

Similar solutions exist for classic software. For example, if you zoom out, most software applications can be split between a DB, business logic, and UI layer. Every layer can be as complex as needed, but at a high-level overview, the architecture of standard software can be boiled down to the previous three components.

Do we have something similar for ML applications? The first step is to examine previous solutions and why they are unsuitable for building scalable ML systems.

The issue with previous solutions

In *Figure 1.2*, you can observe the typical architecture present in most ML applications. It is based on a monolithic batch architecture that couples the feature creation, model training, and inference into the same component. By taking this approach, you quickly solve one critical problem in the ML world: the training-serving skew. The training-serving skew happens when the features passed to the model are computed differently at training and inference time.

In this architecture, the features are created using the same code. Hence, the training-serving skew issue is solved by default. This pattern works fine when working with small data. The pipeline runs on a schedule in batch mode, and the predictions are consumed by a third-party application such as a dashboard.

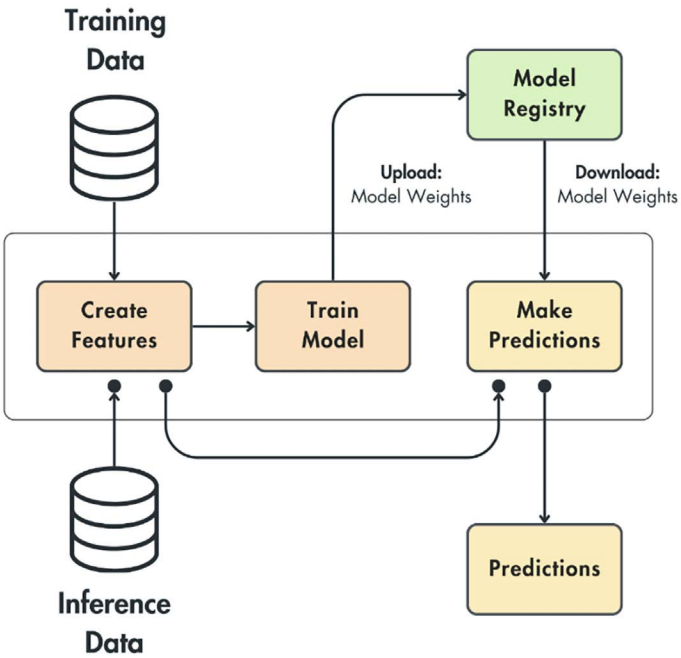

Figure 1.2: Monolithic batch pipeline architecture

Unfortunately, building a monolithic batch system raises many other issues, such as the following:

- Features are not reusable (by your system or others)
- If the data increases, you have to refactor the whole code to support PySpark or Ray
- It's hard to rewrite the prediction module in a more efficient language such as C++, Java, or Rust

- It's hard to share the work between multiple teams between the features, training, and prediction modules
- It's impossible to switch to streaming technology for real-time training

In *Figure 1.3*, we can see a similar scenario for a real-time system. This use case introduces another issue in addition to what we listed before. To make the predictions, we have to transfer the whole state through the client request so the features can be computed and passed to the model.

Consider the scenario of computing movie recommendations for a user. Instead of simply passing the user ID, we must transmit the entire user state, including their name, age, gender, movie history, and more. This approach is fraught with potential errors, as the client must understand how to access this state, and it's tightly coupled with the model service.

Another example would be when implementing an LLM with RAG support. The documents we add as context along the query represent our external state. If we didn't store the records in a vector DB, we would have to pass them with the user query. To do so, the client must know how to query and retrieve the documents, which is not feasible. It is an antipattern for the client application to know how to access or compute the features. If you don't understand how RAG works, we will explain it in detail in *Chapters 8* and *9*.

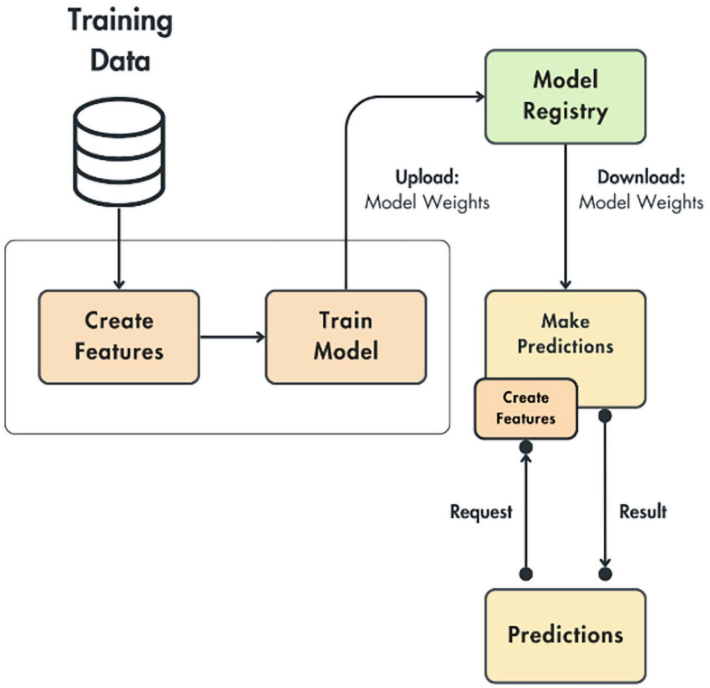

Figure 1.3: Stateless real-time architecture

In conclusion, our problem is accessing the features to make predictions without passing them at the client's request. For example, based on our first user movie recommendation example, how can we predict the recommendations solely based on the user's ID? Remember these questions, as we will answer them shortly.

Ultimately, on the other spectrum, Google Cloud provides a production-ready architecture, as shown in *Figure 1.4*. Unfortunately, even if it's a feasible solution, it's very complex and not intuitive. You will have difficulty understanding this if you are not highly experienced in deploying and keeping ML models in production. Also, it is not straightforward to understand how to start small and grow the system in time.

The following image is reproduced from work created and shared by Google and used according to terms described in the Creative Commons 4.0 Attribution License:

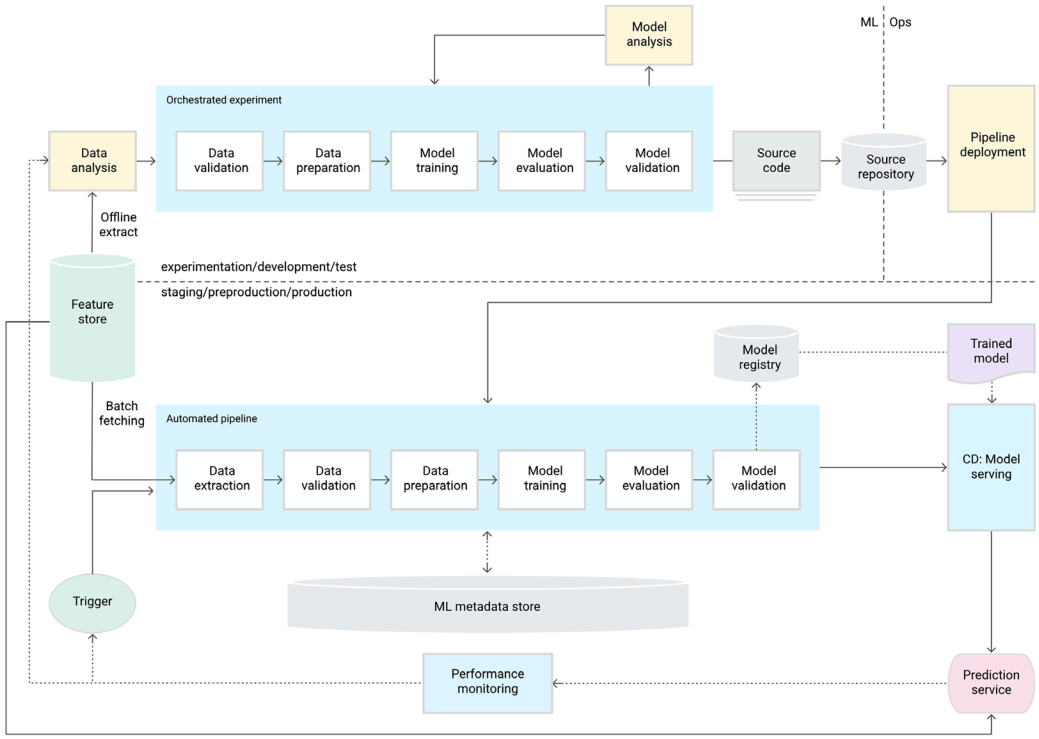

Figure 1.4: ML pipeline automation for CT (source: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning)

But here is where the FTI pipeline architectures kick in. The following section will show you how to solve these fundamental issues using an intuitive ML design.

The solution – ML pipelines for ML systems

The solution is based on creating a clear and straightforward mind map that any team or person can follow to compute the features, train the model, and make predictions. Based on these three critical steps that any ML system requires, the pattern is known as the FTI pipeline. So, how does this differ from what we presented before?

The pattern suggests that any ML system can be boiled down to these three pipelines: feature, training, and inference (similar to the DB, business logic, and UI layers from classic software). This is powerful, as we can clearly define the scope and interface of each pipeline. Also, it's easier to understand how the three components interact. Ultimately, we have just three instead of 20 moving pieces, as suggested in *Figure 1.4*, which is much easier to work with and define.

As shown in *Figure 1.5*, we have the feature, training, and inference pipelines. We will zoom in on each of them and understand their scope and interface.

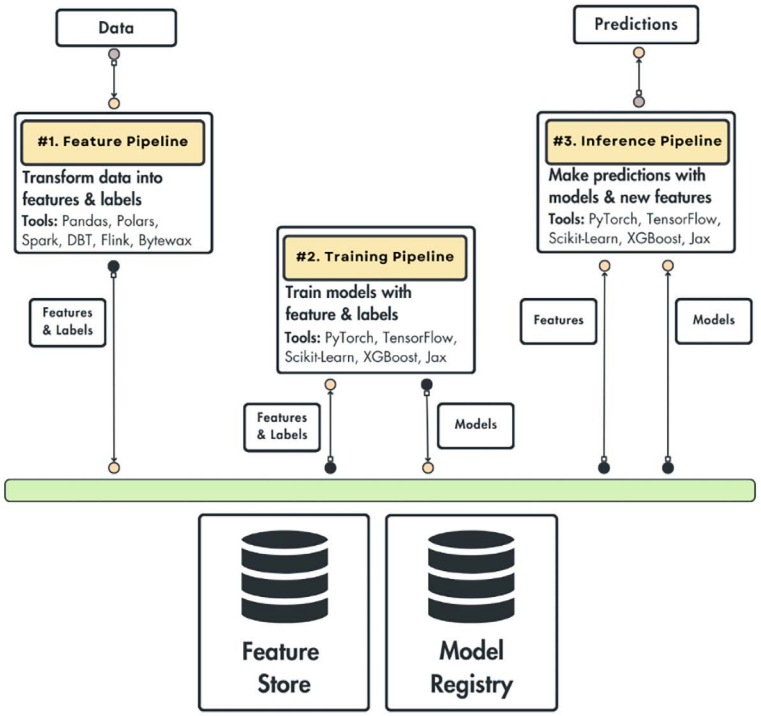

Figure 1.5: FTI pipelines architecture

Before going into the details, it is essential to understand that each pipeline is a different component that can run on a different process or hardware. Thus, each pipeline can be written using a different technology, by a different team, or scaled differently. The key idea is that the design is very flexible to the needs of your team. It acts as a mind map for structuring your architecture.

The feature pipeline

The feature pipeline takes raw data as input, processes it, and outputs the features and labels required by the model for training or inference. Instead of directly passing them to the model, the features and labels are stored inside a feature store. Its responsibility is to store, version, track, and share the features. By saving the features in a feature store, we always have a state of our features. Thus, we can easily send the features to the training and inference pipelines.

As the data is versioned, we can always ensure that the training and inference time features match. Thus, we avoid the training-serving skew problem.

The training pipeline

The training pipeline takes the features and labels from the features stored as input and outputs a train model or models. The models are stored in a model registry. Its role is similar to that of feature stores, but this time, the model is the first-class citizen. Thus, the model registry will store, version, track, and share the model with the inference pipeline.

Also, most modern model registries support a metadata store that allows you to specify essential aspects of how the model was trained. The most important are the features, labels, and their version used to train the model. Thus, we will always know what data the model was trained on.

The inference pipeline

The inference pipeline takes as input the features and labels from the feature store and the trained model from the model registry. With these two, predictions can be easily made in either batch or real-time mode.

As this is a versatile pattern, it is up to you to decide what you do with your predictions. If it's a batch system, they will probably be stored in a DB. If it's a real-time system, the predictions will be served to the client who requested them. Additionally, the features, labels, and models are versioned. We can easily upgrade or roll back the deployment of the model. For example, we will always know that model v1 uses features F1, F2, and F3, and model v2 uses F2, F3, and F4. Thus, we can quickly change the connections between the model and features.

Benefits of the FTI architecture

To conclude, the most important thing you must remember about the FTI pipelines is their interface:

- The feature pipeline takes in data and outputs the features and labels saved to the feature store.
- The training pipeline queries the features store for features and labels and outputs a model to the model registry.
- The inference pipeline uses the features from the feature store and the model from the model registry to make predictions.

It doesn't matter how complex your ML system gets, these interfaces will remain the same.

Now that we understand better how the pattern works, we want to highlight the main benefits of using this pattern:

- As you have just three components, it is intuitive to use and easy to understand.
- Each component can be written into its tech stack, so we can quickly adapt them to specific needs, such as big or streaming data. Also, it allows us to pick the best tools for the job.
- As there is a transparent interface between the three components, each one can be developed by a different team (if necessary), making the development more manageable and scalable.
- Every component can be deployed, scaled, and monitored independently.

The final thing you must understand about the FTI pattern is that the system doesn't have to contain only three pipelines. In most cases, it will include more. For example, the feature pipeline can be composed of a service that computes the features and one that validates the data. Also, the training pipeline can be composed of the training and evaluation components.

The FTI pipelines act as logical layers. Thus, it is perfectly fine for each to be complex and contain multiple services. However, what is essential is to stick to the same interface on how the FTI pipelines interact with each other through the feature store and model registries. By doing so, each FTI component can evolve differently, without knowing the details of each other and without breaking the system on new changes.

To learn more about the FTI pipeline pattern, consider reading *From MLOps to ML Systems with Feature/Training/Inference Pipelines* by Jim Dowling, CEO and co-founder of Hopsworks: `https://www.hopsworks.ai/post/mlops-to-ml-systems-with-fti-pipelines`. His article inspired this section.

Now that we understand the FTI pipeline architecture, the final step of this chapter is to see how it can be applied to the LLM Twin use case.

Designing the system architecture of the LLM Twin

In this section, we will list the concrete technical details of the LLM Twin application and understand how we can solve them by designing our LLM system using the FTI architecture. However, before diving into the pipelines, we want to highlight that we won't focus on the tooling or the tech stack at this step. We only want to define a high-level architecture of the system, which is language-, framework-, platform-, and infrastructure-agnostic at this point. We will focus on each component's scope, interface, and interconnectivity. In future chapters, we will cover the implementation details and tech stack.

Listing the technical details of the LLM Twin architecture

Until now, we defined what the LLM Twin should support from the user's point of view. Now, let's clarify the requirements of the ML system from a purely technical perspective:

- On the data side, we have to do the following:
    - Collect data from LinkedIn, Medium, Substack, and GitHub completely autonomously and on a schedule
    - Standardize the crawled data and store it in a data warehouse
    - Clean the raw data
    - Create instruct datasets for fine-tuning an LLM
    - Chunk and embed the cleaned data. Store the vectorized data into a vector DB for RAG.
- For training, we have to do the following:
    - Fine-tune LLMs of various sizes (7B, 14B, 30B, or 70B parameters)
    - Fine-tune on instruction datasets of multiple sizes
    - Switch between LLM types (for example, between Mistral, Llama, and GPT)
    - Track and compare experiments

    - Test potential production LLM candidates before deploying them
    - Automatically start the training when new instruction datasets are available.
- The inference code will have the following properties:
    - A REST API interface for clients to interact with the LLM Twin
    - Access to the vector DB in real time for RAG
    - Inference with LLMs of various sizes
    - Autoscaling based on user requests
    - Automatically deploy the LLMs that pass the evaluation step.
- The system will support the following LLMOps features:
    - Instruction dataset versioning, lineage, and reusability
    - Model versioning, lineage, and reusability
    - Experiment tracking
    - **Continuous training**, **continuous integration**, and **continuous delivery** (CT/CI/CD)
    - Prompt and system monitoring

If any technical requirement doesn't make sense now, bear with us. To avoid repetition, we will examine the details in their specific chapter.

The preceding list is quite comprehensive. We could have detailed it even more, but at this point, we want to focus on the core functionality. When implementing each component, we will look into all the little details. But for now, the fundamental question we must ask ourselves is this: How can we apply the FTI pipeline design to implement the preceding list of requirements?

How to design the LLM Twin architecture using the FTI pipeline design

We will split the system into four core components. You will ask yourself this: "Four? Why not three, as the FTI pipeline design clearly states?" That is a great question. Fortunately, the answer is simple. We must also implement the data pipeline along the three feature/training/inference pipelines. According to best practices:

- The data engineering team owns the data pipeline
- The ML engineering team owns the FTI pipelines.

Given our goal of building an MVP with a small team, we must implement the entire application. This includes defining the data collection and FTI pipelines. Tackling a problem end to end is often encountered in start-ups that can't afford dedicated teams. Thus, engineers have to wear many hats, depending on the state of the product. Nevertheless, in any scenario, knowing how an end-to-end ML system works is valuable for better understanding other people's work.

Figure 1.6 shows the LLM system architecture. The best way to understand it is to review the four components individually and explain how they work.

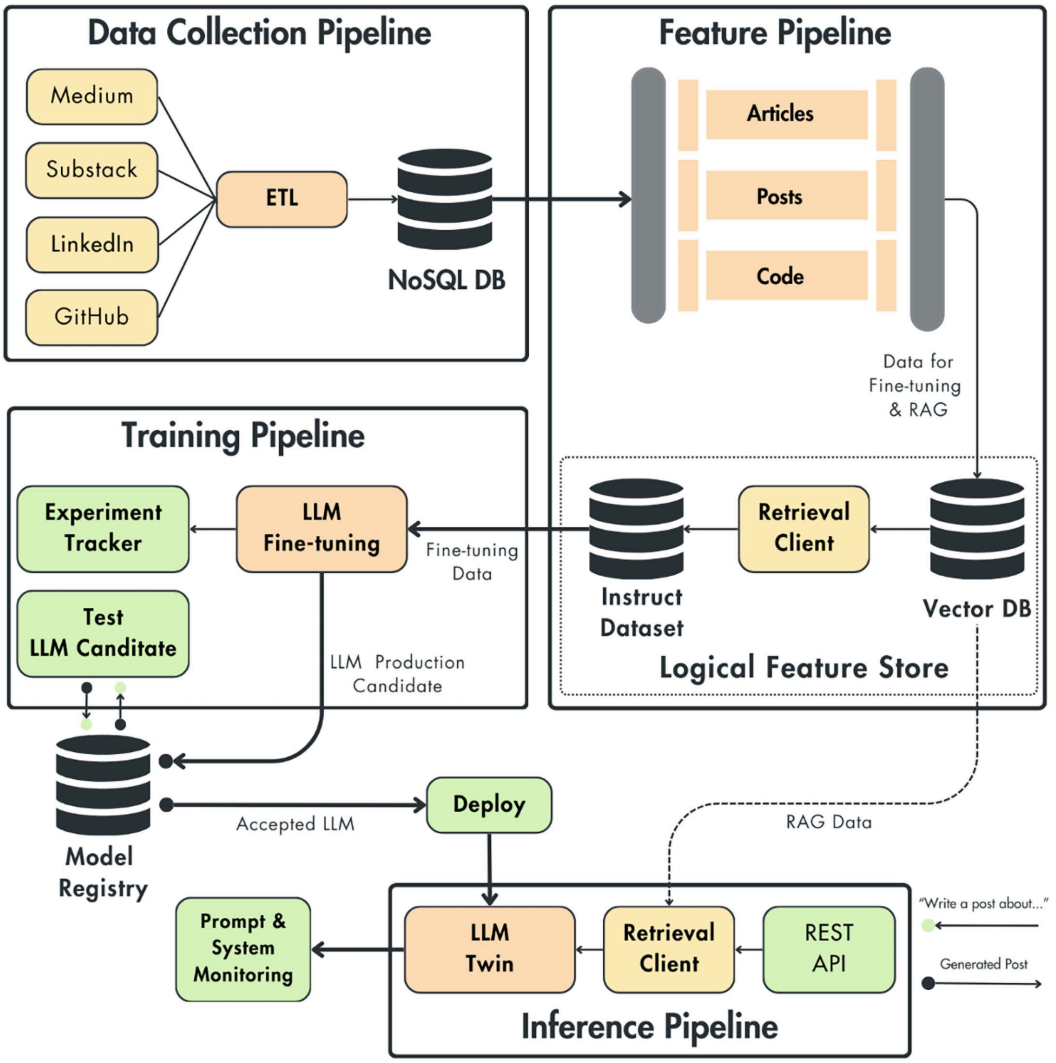

Figure 1.6: LLM Twin high-level architecture

Data collection pipeline

The data collection pipeline involves crawling your personal data from Medium, Substack, LinkedIn, and GitHub. As a data pipeline, we will use the **extract, load, transform** (**ETL**) pattern to extract data from social media platforms, standardize it, and load it into a data warehouse.

It is critical to highlight that the data collection pipeline is designed to crawl data only from your social media platform. It will not have access to other people. As an example for this book, we agreed to make our collected data available for learning purposes. Otherwise, using other people's data without their consent is not moral.

The output of this component will be a NoSQL DB, which will act as our data warehouse. As we work with text data, which is naturally unstructured, a NoSQL DB fits like a glove.

Even though a NoSQL DB, such as MongoDB, is not labeled as a data warehouse, from our point of view, it will act as one. Why? Because it stores standardized raw data gathered by various ETL pipelines that are ready to be ingested into an ML system.

The collected digital data is binned into three categories:

- Articles (Medium, Substack)
- Posts (LinkedIn)
- Code (GitHub)

We want to abstract away the platform where the data was crawled. For example, when feeding an article to the LLM, knowing it came from Medium or Substack is not essential. We can keep the source URL as metadata to give references. However, from the processing, fine-tuning, and RAG points of view, it is vital to know what type of data we ingested, as each category must be processed differently. For example, the chunking strategy between a post, article, and piece of code will look different.

Also, by grouping the data by category, not the source, we can quickly plug data from other platforms, such as X into the posts or GitLab into the code collection. As a modular system, we must attach an additional ETL in the data collection pipeline, and everything else will work without further code modifications.

Feature pipeline

The feature pipeline's role is to take raw articles, posts, and code data points from the data warehouse, process them, and load them into the feature store.

The characteristics of the FTI pattern are already present.

Here are some custom properties of the LLM Twin's feature pipeline:

- It processes three types of data differently: articles, posts, and code
- It contains three main processing steps necessary for fine-tuning and RAG: cleaning, chunking, and embedding
- It creates two snapshots of the digital data, one after cleaning (used for fine-tuning) and one after embedding (used for RAG)
- It uses a logical feature store instead of a specialized feature store

Let's zoom in on the logical feature store part a bit. As with any RAG-based system, one of the central pieces of the infrastructure is a vector DB. Instead of integrating another DB, more concretely, a specialized feature store, we used the vector DB, plus some additional logic to check all the properties of a feature store our system needs.

The vector DB doesn't offer the concept of a training dataset, but it can be used as a NoSQL DB. This means we can access data points using their ID and collection name. Thus, we can easily query the vector DB for new data points without any vector search logic. Ultimately, we will wrap the retrieved data into a versioned, tracked, and shareable artifact—more on artifacts in *Chapter 2*. For now, you must know it is an MLOps concept used to wrap data and enrich it with the properties listed before.

How will the rest of the system access the logical feature store? The training pipeline will use the instruct datasets as artifacts, and the inference pipeline will query the vector DB for additional context using vector search techniques.

For our use case, this is more than enough because of the following reasons:

- The artifacts work great for offline use cases such as training
- The vector DB is built for online access, which we require for inference.

In future chapters, however, we will explain how the three data categories (articles, posts, and code) are cleaned, chunked, and embedded.

To conclude, we take in raw article, post, or code data points, process them, and store them in a feature store to make them accessible to the training and inference pipelines. Note that trimming all the complexity away and focusing only on the interface is a perfect match with the FTI pattern. Beautiful, right?

Training pipeline

The training pipeline consumes instruct datasets from the feature store, fine-tunes an LLM with it, and stores the tuned LLM weights in a model registry. More concretely, when a new instruct dataset is available in the logical feature store, we will trigger the training pipeline, consume the artifact, and fine-tune the LLM.

In the initial stages, the data science team owns this step. They run multiple experiments to find the best model and hyperparameters for the job, either through automatic hyperparameter tuning or manually. To compare and pick the best set of hyperparameters, we will use an experiment tracker to log everything of value and compare it between experiments. Ultimately, they will pick the best hyperparameters and fine-tuned LLM and propose it as the LLM production candidate. The proposed LLM is then stored in the model registry. After the experimentation phase is over, we store and reuse the best hyperparameters found to eliminate the manual restrictions of the process. Now, we can completely automate the training process, known as continuous training.

The testing pipeline is triggered for a more detailed analysis than during fine-tuning. Before pushing the new model to production, assessing it against a stricter set of tests is critical to see that the latest candidate is better than what is currently in production. If this step passes, the model is ultimately tagged as accepted and deployed to the production inference pipeline. Even in a fully automated ML system, it is recommended to have a manual step before accepting a new production model. It is like pushing the red button before a significant action with high consequences. Thus, at this stage, an expert looks at a report generated by the testing component. If everything looks good, it approves the model, and the automation can continue.

The particularities of this component will be on LLM aspects, such as the following:

- How do you implement an LLM agnostic pipeline?
- What fine-tuning techniques should you use?
- How do you scale the fine-tuning algorithm on LLMs and datasets of various sizes?
- How do you pick an LLM production candidate from multiple experiments?
- How do you test the LLM to decide whether to push it to production or not?

By the end of this book, you will know how to answer all these questions.

One last aspect we want to clarify is **CT**. Our modular design allows us to quickly leverage an ML orchestrator to schedule and trigger different system parts. For example, we can schedule the data collection pipeline to crawl data every week.

Then, we can trigger the feature pipeline when new data is available in the data warehouse and the training pipeline when new instruction datasets are available.

Inference pipeline

The inference pipeline is the last piece of the puzzle. It is connected to the model registry and logical feature store. It loads a fine-tuned LLM from the model registry, and from the logical feature store, it accesses the vector DB for RAG. It takes in client requests through a REST API as queries. It uses the fine-tuned LLM and access to the vector DB to carry out RAG and answer the queries.

All the client queries, enriched prompts using RAG, and generated answers are sent to a prompt monitoring system to analyze, debug, and better understand the system. Based on specific requirements, the monitoring system can trigger alarms to take action either manually or automatically.

At the interface level, this component follows exactly the FTI architecture, but when zooming in, we can observe unique characteristics of an LLM and RAG system, such as the following:

- A retrieval client used to do vector searches for RAG
- Prompt templates used to map user queries and external information to LLM inputs
- Special tools for prompt monitoring

Final thoughts on the FTI design and the LLM Twin architecture

We don't have to be highly rigid about the FTI pattern. It is a tool used to clarify how to design ML systems. For example, instead of using a dedicated features store just because that is how it is done, in our system, it is easier and cheaper to use a logical feature store based on a vector DB and artifacts. What was important to focus on were the required properties a feature store provides, such as a versioned and reusable training dataset.

Ultimately, we will explain the computing requirements of each component briefly. The data collection and feature pipeline are mostly CPU-based and do not require powerful machines. The training pipeline requires powerful GPU-based machines that could load an LLM and fine-tune it. The inference pipeline is somewhere in the middle. It still needs a powerful machine but is less compute-intensive than the training step. However, it must be tested carefully, as the inference pipeline directly interfaces with the user. Thus, we want the latency to be within the required parameters for a good user experience. However, using the FTI design is not an issue. We can pick the proper computing requirements for each component.

Also, each pipeline will be scaled differently. The data and feature pipelines will be scaled horizontally based on the CPU and RAM load. The training pipeline will be scaled vertically by adding more GPUs. The inference pipeline will be scaled horizontally based on the number of client requests.

To conclude, the presented LLM architecture checks all the technical requirements listed at the beginning of the section. It processes the data as requested, and the training is modular and can be quickly adapted to different LLMs, datasets, or fine-tuning techniques. The inference pipeline supports RAG and is exposed as a REST API. On the LLMOps side, the system supports dataset and model versioning, lineage, and reusability. The system has a monitoring service, and the whole ML architecture is designed with CT/CI/CD in mind.

This concludes the high-level overview of the LLM Twin architecture.

Summary

This first chapter was critical to understanding the book's goal. As a product-oriented book that will walk you through building an end-to-end ML system, it was essential to understand the concept of an LLM Twin initially. Afterward, we walked you through what an MVP is and how to plan our LLM Twin MVP based on our available resources. Following this, we translated our concept into a practical technical solution with specific requirements. In this context, we introduced the FTI design pattern and showcased its real-world application in designing systems that are both modular and scalable. Ultimately, we successfully applied the FTI pattern to design the architecture of the LLM Twin to fit all our technical requirements.

Having a clear vision of the big picture is essential when building systems. Understanding how a single component will be integrated into the rest of the application can be very valuable when working on it. We started with a more abstract presentation of the LLM Twin architecture, focusing on each component's scope, interface, and interconnectivity.

The following chapters will explore how to implement and deploy each component. On the MLOps side, we will walk you through using a computing platform, orchestrator, model registry, artifacts, and other tools and concepts to support all MLOps best practices.

References

- Dowling, J. (2024a, July 11). *From MLOps to ML Systems with Feature/Training/Inference Pipelines. Hopsworks*. `https://www.hopsworks.ai/post/mlops-to-ml-systems-with-fti-pipelines`

- Dowling, J. (2024b, August 5). *Modularity and Composability for AI Systems with AI Pipelines and Shared Storage*. *Hopsworks*. https://www.hopsworks.ai/post/modularity-and-composability-for-ai-systems-with-ai-pipelines-and-shared-storage
- Joseph, M. (2024, August 23). *The Taxonomy for Data Transformations in AI Systems*. *Hopsworks*. https://www.hopsworks.ai/post/a-taxonomy-for-data-transformations-in-ai-systems
- *MLOps: Continuous delivery and automation pipelines in machine learning*. (2024, August 28). Google Cloud. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning
- Qwak. (2024a, June 2). *CI/CD for Machine Learning in 2024: Best Practices to build, test, and Deploy* | Infer. *Medium*. https://medium.com/infer-qwak/ci-cd-for-machine-learning-in-2024-best-practices-to-build-test-and-deploy-c4ad869824d2
- Qwak. (2024b, July 23). *5 Best Open Source Tools to build End-to-End MLOPs Pipeline* in 2024. *Medium*. https://medium.com/infer-qwak/building-an-end-to-end-mlops-pipeline-with-open-source-tools-d8bacbf4184f
- Salama, K., Kazmierczak, J., & Schut, D. (2021). *Practitioners guide to MLOPs: A framework for continuous delivery and automation of machine learning* (1st ed.) [PDF]. Google Cloud. https://services.google.com/fh/files/misc/practitioners_guide_to_mlops_whitepaper.pdf

Join our book's Discord space

Join our community's Discord space for discussions with the authors and other readers:

https://packt.link/llmeng

2

Tooling and Installation

This chapter presents all the essential tools that will be used throughout the book, especially in implementing and deploying the LLM Twin project. At this point in the book, we don't plan to present in-depth LLM, RAG, MLOps, or LLMOps concepts. We will quickly walk you through our tech stack and prerequisites to avoid repeating ourselves throughout the book on how to set up a particular tool and why we chose it. Starting with *Chapter 3*, we will begin exploring our LLM Twin use case by implementing a data collection ETL that crawls data from the internet.

In the first part of the chapter, we will present the tools within the Python ecosystem to manage multiple Python versions, create a virtual environment, and install the pinned dependencies required for our project to run. Alongside presenting these tools, we will also show how to install the `LLM-Engineers-Handbook` repository on your local machine (in case you want to try out the code yourself): `https://github.com/PacktPublishing/LLM-Engineers-Handbook`.

Next, we will explore all the MLOps and LLMOps tools we will use, starting with more generic tools, such as a model registry, and moving on to more LLM-oriented tools, such as LLM evaluation and prompt monitoring tools. We will also understand how to manage a project with multiple ML pipelines using ZenML, an orchestrator bridging the gap between ML and MLOps. Also, we will quickly explore what databases we will use for NoSQL and vector storage. We will show you how to run all these components on your local machine using Docker. Lastly, we will quickly review AWS and show you how to create an AWS user and access keys and install and configure the AWS CLI to manipulate your cloud resources programmatically. We will also explore SageMaker and why we use it to train and deploy our open-source LLMs.

If you are familiar with these tools, you can safely skip this chapter. We also explain how to install the project and set up all the necessary components in the repository's `README`. Thus, you also have the option to use that as more concise documentation if you plan to run the code while reading the book.

To sum all that up, in this chapter, we will explore the following topics:

- Python ecosystem and project installation
- MLOps and LLMOps tooling
- Databases for storing unstructured and vector data
- Preparing for AWS

By the end of this chapter, you will be aware of all the tools we will use across the book. Also, you will have learned how to install the `LLM-Engineers-Handbook` repository, set up the rest of the tools, and use them if you run the code while reading the book.

Python ecosystem and project installation

Any Python project needs three fundamental tools: the Python interpreter, dependency management, and a task execution tool. The Python interpreter executes your Python project as expected. All the code within the book is tested with Python 3.11.8. You can download the Python interpreter from here: `https://www.python.org/downloads/`. We recommend installing the exact Python version (Python 3.11.8) to run the LLM Twin project using `pyenv`, making the installation process straightforward.

Instead of installing multiple global Python versions, we recommend managing them using `pyenv`, a Python version management tool that lets you manage multiple Python versions between projects. You can install it using this link: `https://github.com/pyenv/pyenv?tab=readme-ov-file#installation`.

After you have installed `pyenv`, you can install the latest version of Python 3.11, using `pyenv`, as follows:

```
pyenv install 3.11.8
```

Now list all installed Python versions to see that it was installed correctly:

```
pyenv versions
```

You should see something like this:

```
# * system
```

```
#   3.11.8
```

To make Python 3.11.8 the default version across your entire system (whenever you open a new terminal), use the following command:

```
pyenv global 3.11.8
```

However, we aim to use Python 3.11.8 locally only in our repository. To achieve that, first, we have to clone the repository and navigate to it:

```
git clone https://github.com/PacktPublishing/LLM-Engineers-Handbook.git
cd LLM-Engineers-Handbook
```

Because we defined a `.python-version` file within the repository, `pyenv` will know to pick up the version from that file and use it locally whenever you are working within that folder. To double-check that, run the following command while you are in the repository:

```
python --version
```

It should output:

```
# Python 3.11.8
```

To create the `.python-version` file, you must run `pyenv local 3.11.8` once. Then, `pyenv` will always know to use that Python version while working within a specific directory.

Now that we have installed the correct Python version using `pyenv`, let's move on to Poetry, which we will use as our dependency and virtual environment manager.

Poetry: dependency and virtual environment management

Poetry is one of the most popular dependency and virtual environment managers within the Python ecosystem. But let's start by clarifying what a dependency manager is. In Python, a dependency manager allows you to specify, install, update, and manage external libraries or packages (dependencies) that a project relies on. For example, this is a simple Poetry requirements file that uses Python 3.11 and the `requests` and `numpy` Python packages.

```
[tool.poetry.dependencies]
python = "^3.11"
requests = "^2.25.1"
numpy = "^1.19.5"

[build-system]
```

```
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

By using Poetry to pin your dependencies, you always ensure that you install the correct version of the dependencies that your projects work with. Poetry, by default, saves all its requirements in `pyproject.toml` files, which are stored at the root of your repository, as you can see in the cloned LLM-Engineers-Handbook repository.

Another massive advantage of using Poetry is that it creates a new Python virtual environment in which it installs the specified Python version and requirements. A virtual environment allows you to isolate your project's dependencies from your global Python dependencies and other projects. By doing so, you ensure there are no version clashes between projects. For example, let's assume that Project A needs `numpy == 1.19.5`, and Project B needs `numpy == 1.26.0`. If you keep both projects in the global Python environment, that will not work, as Project B will override Project A's `numpy` installation, which will corrupt Project A and stop it from working. Using Poetry, you can isolate each project in its own Python environment with its own Python dependencies, avoiding any dependency clashes.

You can install Poetry from here: `https://python-poetry.org/docs/`. We use Poetry 1.8.3 throughout the book. Once Poetry is installed, navigate to your cloned LLM-Engineers-Handbook repository and run the following command to install all the necessary Python dependencies:

```
poetry install --without aws
```

This command knows to pick up all the dependencies from your repository that are listed in the `pyproject.toml` and `poetry.lock` files. After the installation, you can activate your Poetry environment by running `poetry shell` in your terminal or by prefixing all your CLI commands as follows: `poetry run <your command>`.

One final note on Poetry is that it locks down the exact versions of the dependency tree in the `poetry.lock` file based on the definitions added to the `project.toml` file. While the `pyproject.toml` file may specify version ranges (e.g., `requests = "^2.25.1"`), the `poetry.lock` file records the exact version (e.g., `requests = "2.25.1"`) that was installed. It also locks the versions of sub-dependencies (dependencies of your dependencies), which may not be explicitly listed in your `pyproject.toml` file. By locking all the dependencies and sub-dependencies to specific versions, the `poetry.lock` file ensures that all project installations use the same versions of each package. This consistency leads to predictable behavior, reducing the likelihood of encountering "works on my machine" issues.

Other tools similar to Poetry are Venv and Conda for creating virtual environments. Still, they lack the dependency management option. Thus, you must do it through Python's default `requirements.txt` files, which are less powerful than Poetry's `lock` files. Another option is Pipenv, which feature-wise is more like Poetry but slower, and `uv`, which is a replacement for Poetry built in Rust, making it blazing fast. `uv` has lots of potential to replace Poetry, making it worthwhile to test out: `https://github.com/astral-sh/uv`.

The final piece of the puzzle is to look at the task execution tool we used to manage all our CLI commands.

Poe the Poet: task execution tool

Poe the Poet is a plugin on top of Poetry that is used to manage and execute all the CLI commands required to interact with the project. It helps you define and run tasks within your Python project, simplifying automation and script execution. Other popular options are Makefile, Invoke, or shell scripts, but Poe the Poet eliminates the need to write separate shell scripts or Makefiles for managing project tasks, making it an elegant way to manage tasks using the same configuration file that Poetry already uses for dependencies.

When working with Poe the Poet, instead of having all your commands documented in a README file or other document, you can add them directly to your `pyproject.toml` file and execute them in the command line with an alias. For example, using Poe the Poet, we can define the following tasks in a `pyproject.toml` file:

```
[tool.poe.tasks]
test = "pytest"
format = "black ."
start = "python main.py"
```

You can then run these tasks using the poe command:

```
poetry poe test
poetry poe format
poetry poe start
```

You can install Poe the Poet as a Poetry plugin, as follows:

```
poetry self add 'poethepoet[poetry_plugin]'
```

To conclude, using a tool as a façade over all your CLI commands is necessary to run your application. It significantly simplifies the application's complexity and enhances collaboration as it acts as out-of-the-box documentation.

Assuming you have pyenv and Poetry installed, here are all the commands you need to run to clone the repository and install the dependencies and Poe the Poet as a Poetry plugin:

```
git clone https://github.com/PacktPublishing/LLM-Engineers-Handbook.gitcd
LLM-Engineers-Handbook
poetry install --without aws
poetry self add 'poethepoet[poetry_plugin]'
```

To make the project fully operational, there are still a few steps to follow, such as filling out a `.env` file with your credentials and getting tokens from OpenAI and Hugging Face. But this book isn't an installation guide, so we've moved all these details into the repository's README as they are useful only if you plan to run the repository: `https://github.com/PacktPublishing/LLM-Engineers-Handbook`.

Now that we have installed our Python project, let's present the MLOps tools we will use in the book. If you are already familiar with these tools, you can safely skip the following tooling section and move on to the *Databases for storing unstructured and vector data* section.

MLOps and LLMOps tooling

This section will quickly present all the MLOps and LLMOps tools we will use throughout the book and their role in building ML systems using MLOps best practices. At this point in the book, we don't aim to detail all the MLOps components we will use to implement the LLM Twin use case, such as model registries and orchestrators, but only provide a quick idea of what they are and how to use them. As we develop the LLM Twin project throughout the book, you will see hands-on examples of how we use all these tools. In *Chapter 11*, we will dive deeply into the theory of MLOps and LLMOps and connect all the dots. As the MLOps and LLMOps fields are highly practical, we will leave the theory of these aspects to the end, as it will be much easier to understand it after you go through the LLM Twin use case implementation.

Also, this section is not dedicated to showing you how to set up each tool. It focuses primarily on what each tool is used for and highlights the core features used throughout this book.

Still, using Docker, you can quickly run the whole infrastructure locally. If you want to run the steps within the book yourself, you can host the application locally with these three simple steps:

1. Have Docker 27.1.1 (or higher) installed.

2. Fill your `.env` file with all the necessary credentials as explained in the repository README.
3. Run `poetry poe local-infrastructure-up` to locally spin up ZenML (`http://127.0.0.1:8237/`) and the MongoDB and Qdrant databases.

You can read more details on how to run everything locally in the LLM-Engineers-Handbook repository README: `https://github.com/PacktPublishing/LLM-Engineers-Handbook`. Within the book, we will also show you how to deploy each component to the cloud.

Hugging Face: model registry

A model registry is a centralized repository that manages ML models throughout their lifecycle. It stores models along with their metadata, version history, and performance metrics, serving as a single source of truth. In MLOps, a model registry is crucial for tracking, sharing, and documenting model versions, facilitating team collaboration. Also, it is a fundamental element in the deployment process as it integrates with **continuous integration and continuous deployment (CI/CD)** pipelines.

We used Hugging Face as our model registry, as we can leverage its ecosystem to easily share our fine-tuned LLM Twin models with anyone who reads the book. Also, by following the Hugging Face model registry interface, we can easily integrate the model with all the frameworks around the LLMs ecosystem, such as Unsloth for fine-tuning and SageMaker for inference.

Our fine-tuned LLMs are available on Hugging Face at:

- **TwinLlama 3.1 8B** (after fine-tuning): `https://huggingface.co/mlabonne/TwinLlama-3.1-8B`
- **TwinLlama 3.1 8B DPO** (after preference alignment): `https://huggingface.co/mlabonne/TwinLlama-3.1-8B-DPO`

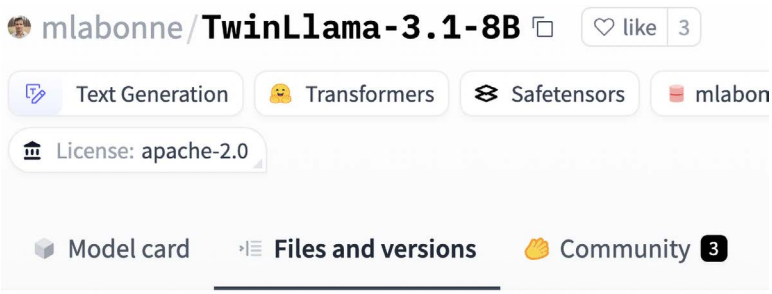

Figure 2.1: Hugging Face model registry example

For a quick demo, we have them available on Hugging Face Spaces:

- **TwinLlama 3.1 8B:** `https://huggingface.co/spaces/mlabonne/TwinLlama-3.1-8B`
- **TwinLlama 3.1 8B DPO:** `https://huggingface.co/spaces/mlabonne/TwinLlama-3.1-8B-DPO`

Most ML tools provide model registry features. For example, ZenML, Comet, and SageMaker, which we will present in future sections, also offer their own model registries. They are good options, but we picked Hugging Face solely because of its ecosystem, which provides easy shareability and integration throughout the open-source environment. Thus, you will usually select the model registry that integrates the most with your project's tooling and requirements.

ZenML: orchestrator, artifacts, and metadata

ZenML acts as the bridge between ML and MLOps. Thus, it offers multiple MLOps features that make your ML pipeline traceability, reproducibility, deployment, and maintainability easier. At its core, it is designed to create reproducible workflows in machine learning. It addresses the challenge of transitioning from exploratory research in Jupyter notebooks to a production-ready ML environment. It tackles production-based replication issues, such as versioning difficulties, reproducing experiments, organizing complex ML workflows, bridging the gap between training and deployment, and tracking metadata. Thus, ZenML's main features are orchestrating ML pipelines, storing and versioning ML pipelines as outputs, and attaching metadata to artifacts for better observability.

Instead of being another ML platform, ZenML introduced the concept of a *stack,* which allows you to run ZenML on multiple infrastructure options. A stack will enable you to connect ZenML to different cloud services, such as:

- An orchestrator and compute engine (for example, AWS SageMaker or Vertex AI)
- Remote storage (for instance, AWS S3 or Google Cloud Storage buckets)
- A container registry (for example, Docker Registry or AWS ECR)

Thus, ZenML acts as a glue that brings all your infrastructure and tools together in one place through its *stack* feature, allowing you to quickly iterate through your development processes and easily monitor your entire ML system. The beauty of this is that ZenML doesn't vendor-lock you into any cloud platform. It completely abstracts away the implementation of your Python code from the infrastructure it runs on. For example, in our LLM Twin use case, we used the AWS stack:

- SageMaker as our orchestrator and compute

- S3 as our remote storage used to store and track artifacts
- ECR as our container registry

However, the Python code contains no S3 or ECR particularities, as ZenML takes care of them. Thus, we can easily switch to other providers, such as Google Cloud Storage or Azure. For more details on ZenML *stacks*, you can start here: `https://docs.zenml.io/user-guide/production-guide/understand-stacks`.

We will focus only on the ZenML features used throughout the book, such as orchestrating, artifacts, and metadata. For more details on ZenML, check out their starter guide: `https://docs.zenml.io/user-guide/starter-guide`.

The local version of the ZenML server comes installed as a Python package. Thus, when running `poetry install`, it installs a ZenML debugging server that you can use locally. In *Chapter 11*, we will show you how to use their cloud serverless option to deploy the ML pipelines to AWS.

Orchestrator

An orchestrator is a system that automates, schedules, and coordinates all your ML pipelines. It ensures that each pipeline—such as data ingestion, preprocessing, model training, and deployment—executes in the correct order and handles dependencies efficiently. By managing these processes, an orchestrator optimizes resource utilization, handles failures gracefully, and enhances scalability, making complex ML pipelines more reliable and easier to manage.

How does ZenML work as an orchestrator? It works with **pipelines** and **steps**. A pipeline is a high-level object that contains multiple steps. A function becomes a ZenML pipeline by being decorated with `@pipeline`, and a step when decorated with `@step`. This is a standard pattern when using orchestrators: you have a high-level function, often called a pipeline, that calls multiple units/steps/tasks.

Let's explore how we can implement a ZenML pipeline with one of the ML pipelines implemented for the LLM Twin project. In the code snippet below, we defined a ZenML pipeline that queries the database for a user based on its full name and crawls all the provided links under that user:

```
from zenml import pipeline
from steps.etl import crawl_links, get_or_create_user

@pipeline
```

```
def digital_data_etl(user_full_name: str, links: list[str]) -> None:
    user = get_or_create_user(user_full_name)
    crawl_links(user=user, links=links)
```

You can run the pipeline with the following CLI command: `poetry poe run-digital-data-etl`. To visualize the pipeline run, you can go to your ZenML dashboard (at `http://127.0.0.1:8237/`) and, on the left panel, click on the **Pipelines** tab and then on the **digital_data_etl** pipeline, as illustrated in *Figure 2.2*:

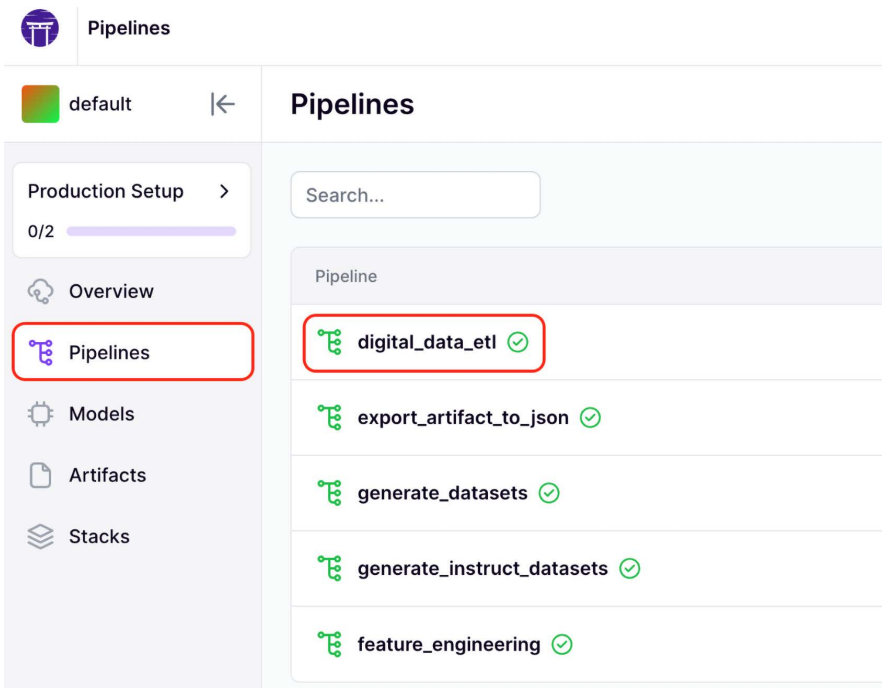

Figure 2.2: ZenML Pipelines dashboard

After clicking on the **digital_data_etl** pipeline, you can visualize all the previous and current pipeline runs, as seen in *Figure 2.3*. You can see which one succeeded, failed, or is still running. Also, you can see the stack used to run the pipeline, where the default stack is the one used to run your ML pipelines locally.