The *LLM Engineer's Handbook* serves as an invaluable resource for anyone seeking a hands-on understanding of LLMs. Through practical examples and a comprehensive exploration of the LLM Twin project, the author effectively demystifies the complexities of building and deploying production-level LLM applications.

One of the book's standout features is its use of the LLM Twin project as a running example. This AI character, designed to emulate the writing style of a specific individual, provides a tangible illustration of how LLMs can be applied in real-world scenarios.

The author skillfully guides readers through the essential tools and technologies required for LLM development, including Hugging Face, ZenML, Comet, Opik, MongoDB, and Qdrant. Each tool is explained in detail, making it easy for readers to understand their functions and how they can be integrated into an LLM pipeline.

LLM Engineer's Handbook also covers a wide range of topics related to LLM development, such as data collection, fine-tuning, evaluation, inference optimization, and MLOps. Notably, the chapters on supervised fine-tuning, preference alignment, and **Retrieval Augmented Generation (RAG)** provide in-depth insights into these critical aspects of LLM development.

A particular strength of this book lies in its focus on practical implementation. The author excels at providing concrete examples and guidance on how to optimize inference pipelines and deploy LLMs effectively. This makes the book a valuable resource for both researchers and practitioners.

This book is highly recommended for anyone interested in learning about LLMs and their practical applications. By providing a comprehensive overview of the tools, techniques, and best practices involved in LLM development, the authors have created a valuable resource that will undoubtedly be a reference for many LLM Engineers

Antonio Gulli

Senior Director, Google

Contributors

About the authors

Paul Iusztin is a senior ML and MLOps engineer with over seven years of experience building GenAI, Computer Vision and MLOps solutions. His latest contribution was at Metaphysic, where he served as one of their core engineers in taking large neural networks to production. He previously worked at CoreAI, Everseen, and Continental. He is the Founder of Decoding ML, an educational channel on production-grade ML that provides posts, articles, and open-source courses to help others build real-world ML systems.

Maxime Labonne is the Head of Post-Training at Liquid AI. He holds a PhD. in ML from the Polytechnic Institute of Paris and is recognized as a Google Developer Expert in AI/ML. As an active blogger, he has made significant contributions to the open-source community, including the LLM Course on GitHub, tools such as LLM AutoEval, and several state-of-the-art models like NeuralDaredevil. He is the author of the best-selling book *Hands-On Graph Neural Networks Using Python*, published by Packt.

I want to thank my family and partner. Your unwavering support and patience made this book possible.

About the reviewer

Rany ElHousieny is an AI solutions architect and AI engineering manager with over two decades of experience in AI, NLP, and ML. Throughout his career, he has focused on the development and deployment of AI models, authoring multiple articles on AI systems architecture and ethical AI deployment. He has led groundbreaking projects at companies like Microsoft, where he spearheaded advancements in NLP and the Language Understanding Intelligent Service (LUIS). Currently, he plays a pivotal role at Clearwater Analytics, driving innovation in GenAI and AI-driven financial and investment management solutions.

I would like to thank Clearwater Analytics for providing a supportive and learning environment that fosters growth and innovation. The vision of our leaders, always staying ahead with the latest technologies, has been a constant source of inspiration. Their commitment to AI advancements made my experience of reviewing this book insightful and enriching. Special thanks to my family for their ongoing encouragement throughout this journey.

Join our book's Discord space

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

https://packt.link/llmeng

Table of Contents

Preface

The field of **LLM** engineering has rapidly emerged as a critical area in artificial intelligence and machine learning. As LLMs continue to revolutionize natural language processing and generation, the demand for professionals who can effectively implement, optimize, and deploy these models in real-world scenarios has grown exponentially. LLM engineering encompasses a wide range of disciplines, from data preparation and model fine-tuning to inference optimization and production deployment, requiring a unique blend of software engineering, machine learning expertise, and domain knowledge.

Machine Learning Operations (**MLOps**) plays a crucial role in the successful implementation of LLMs in production environments. MLOps extends the principles of DevOps to machine learning projects, focusing on automating and streamlining the entire ML lifecycle. For LLMs, MLOps is particularly important due to the complexity and scale of these models. It addresses challenges such as managing large datasets, handling model versioning, ensuring reproducibility, and maintaining model performance over time. By incorporating MLOps practices, LLM projects can achieve greater efficiency, reliability, and scalability, ultimately leading to more successful and impactful deployments.

The LLM Engineer's Handbook is a comprehensive guide to applying best practices to the new field of LLM engineering. Throughout the chapters, readers will find simplified key concepts, practical techniques, and experts tips for every stage of the LLM lifecycle. The book covers topics such as data engineering, supervised fine-tuning, model evaluation, inference optimization, and **Retrieval-Augmented Generation** (**RAG**) pipeline development.

To illustrate these concepts in action, an end-to-end project called the LLM Twin will be developed throughout the book., with the goal of imitating someone's writing style and personality. This use case will demonstrate how to build a minimum viable product to solve a specific problem, using various aspects of LLM engineering and MLOps.

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.

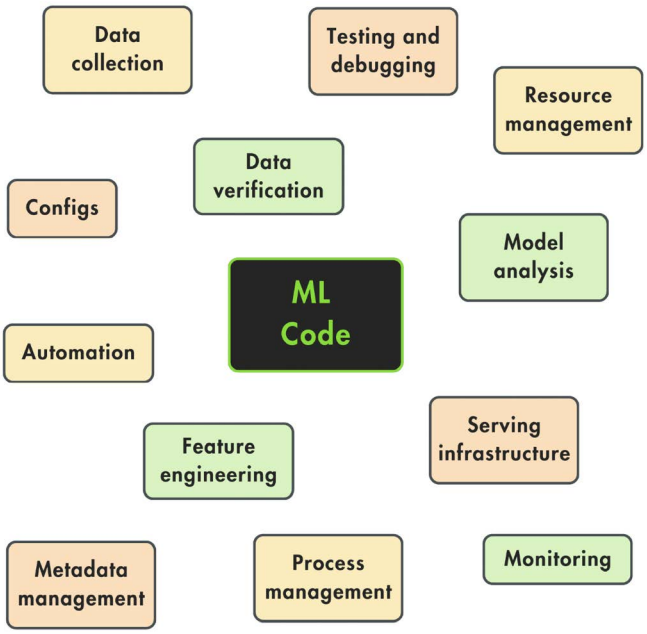

Figure 1.1: Common elements from an ML system

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.

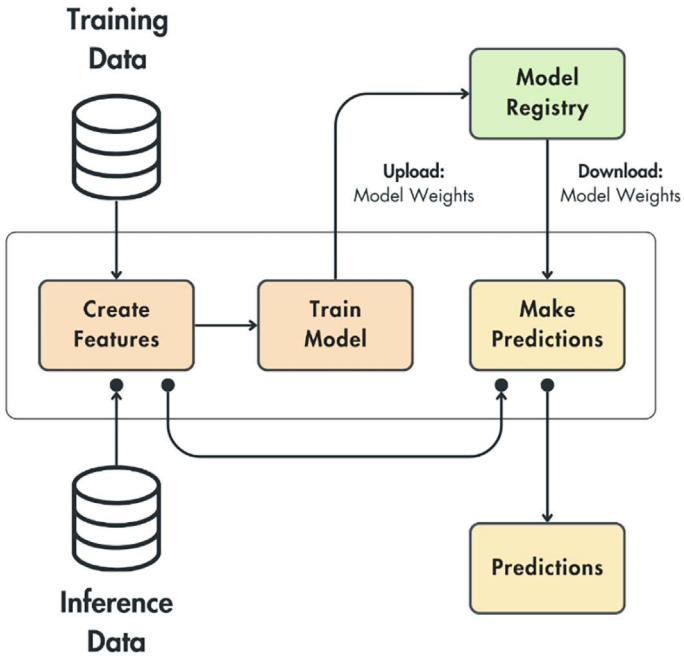

Figure 1.2: Monolithic batch pipeline architecture

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*.

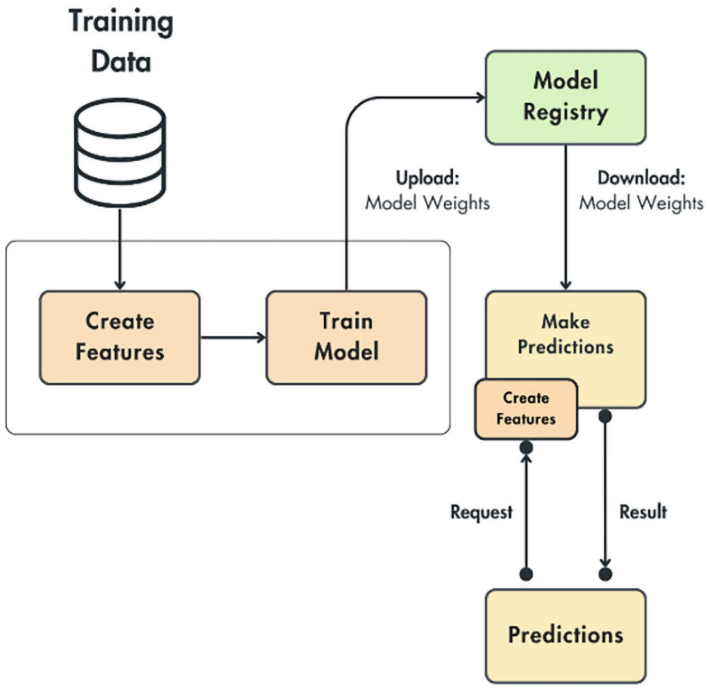

Figure 1.3: Stateless real-time architecture

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:

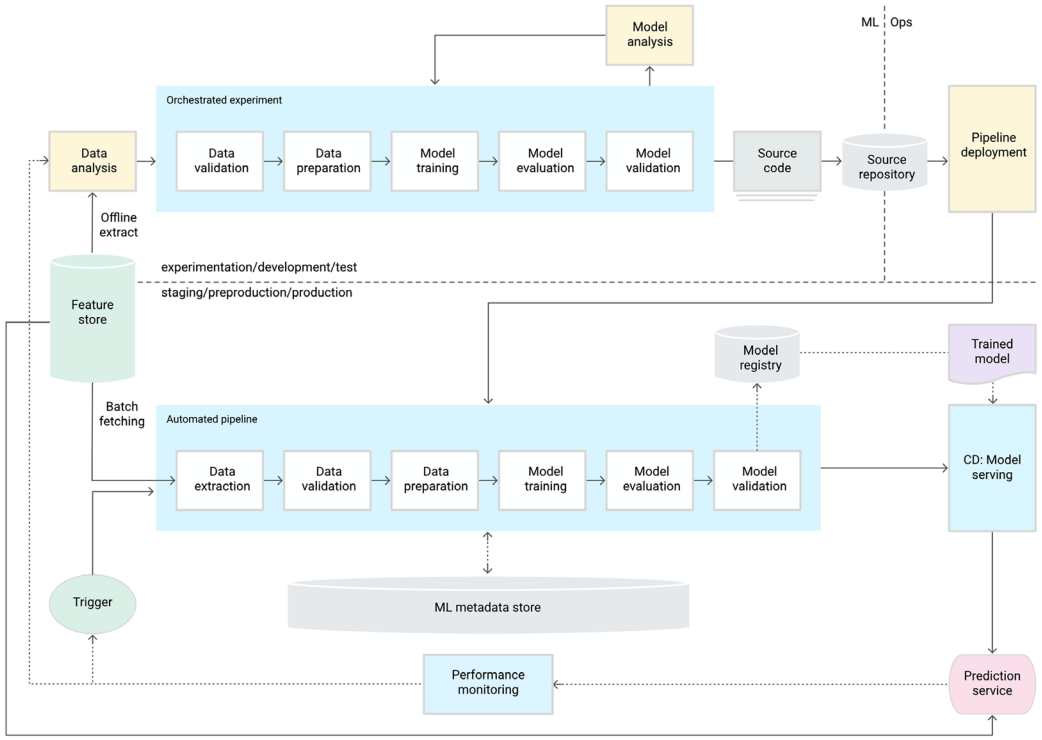

Figure 1.4: ML pipeline automation for CT (source: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning)

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.

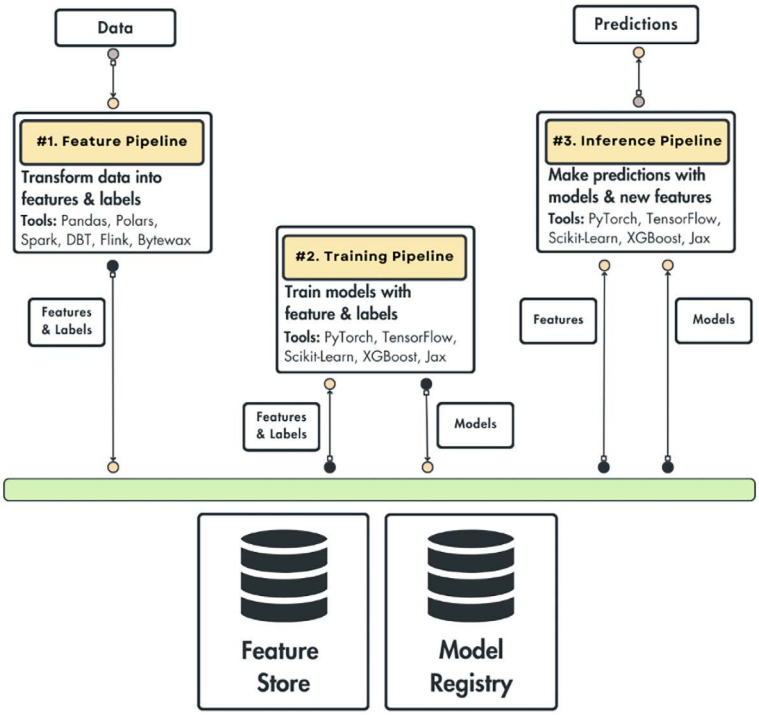

Figure 1.5: FTI pipelines architecture

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.

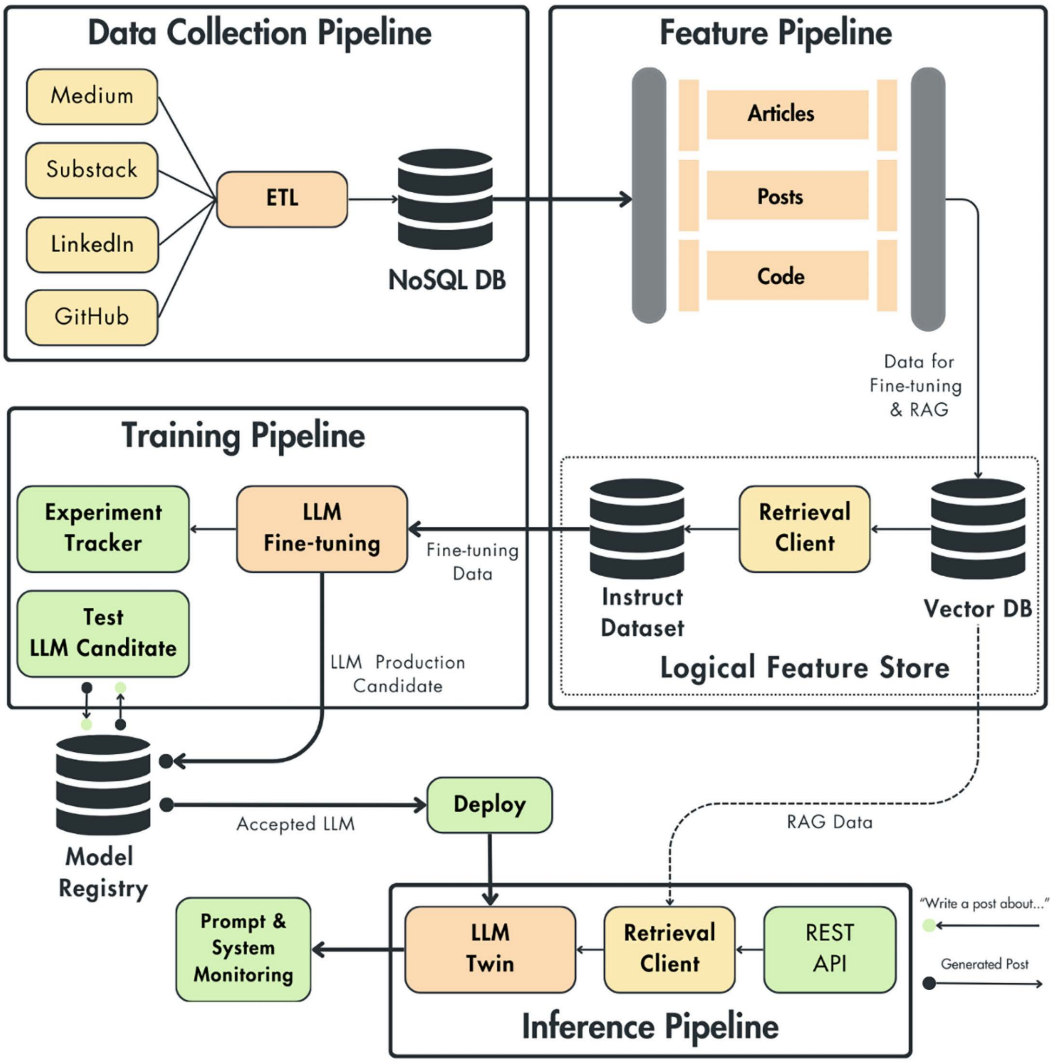

Figure 1.6: LLM Twin high-level architecture

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`

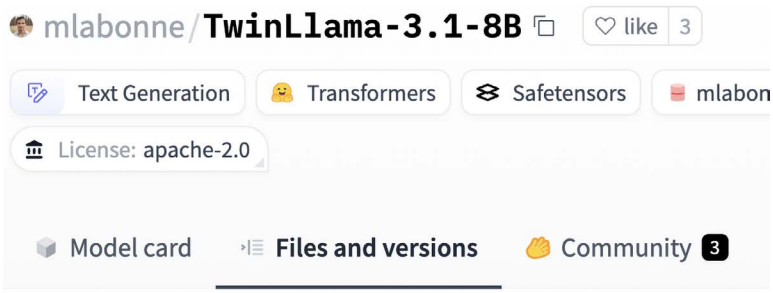

Figure 2.1: Hugging Face model registry example

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*:

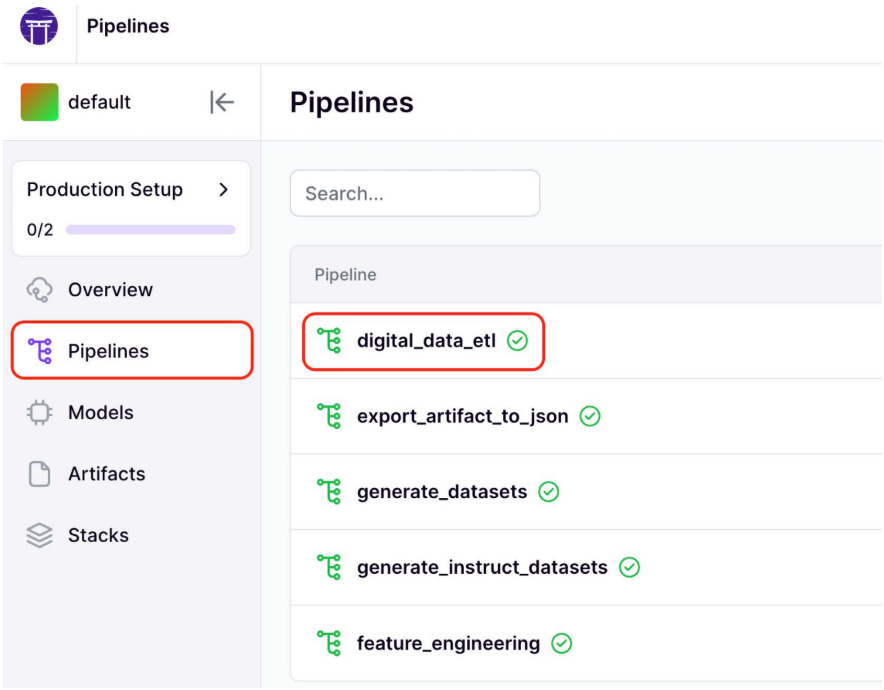

Figure 2.2: ZenML Pipelines dashboard

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.

Figure 2.3: ZenML digital_data_etl pipeline dashboard. Example of a specific pipeline

Now, after clicking on the latest **digital_data_etl** pipeline run (or any other run that succeeded or is still running), we can visualize the pipeline's steps, outputs, and insights, as illustrated in *Figure 2.4*. This structure is often called a **directed acyclic graph (DAG)**. More on DAGs in *Chapter 11*.

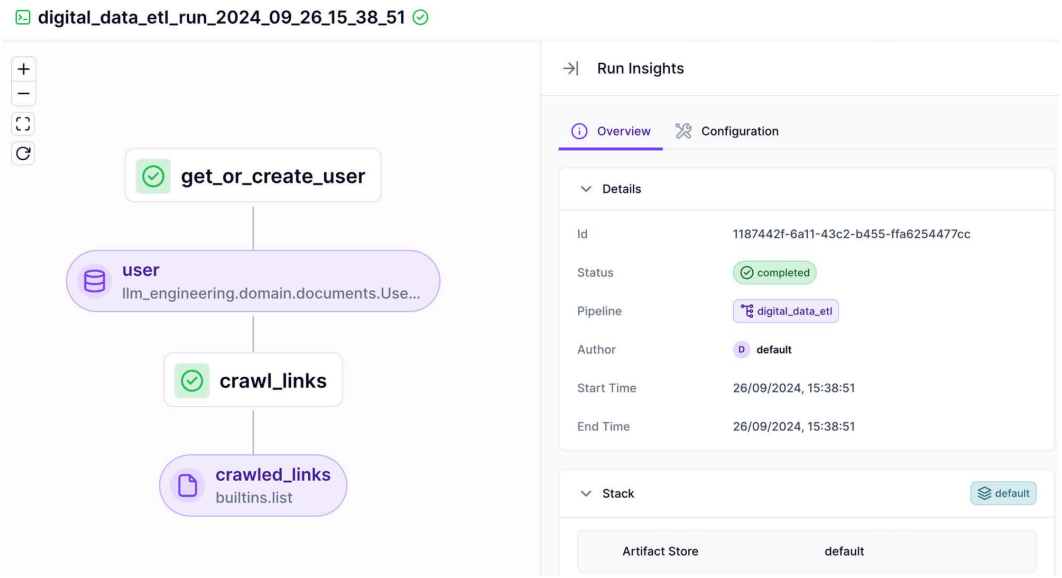

Figure 2.4: ZenML digital_data_etl pipeline run dashboard (example of a specific pipeline run)

By clicking on a specific step, you can get more insights into its code and configuration. It even aggregates the logs output by that specific step to avoid switching between tools, as shown in *Figure 2.5*.

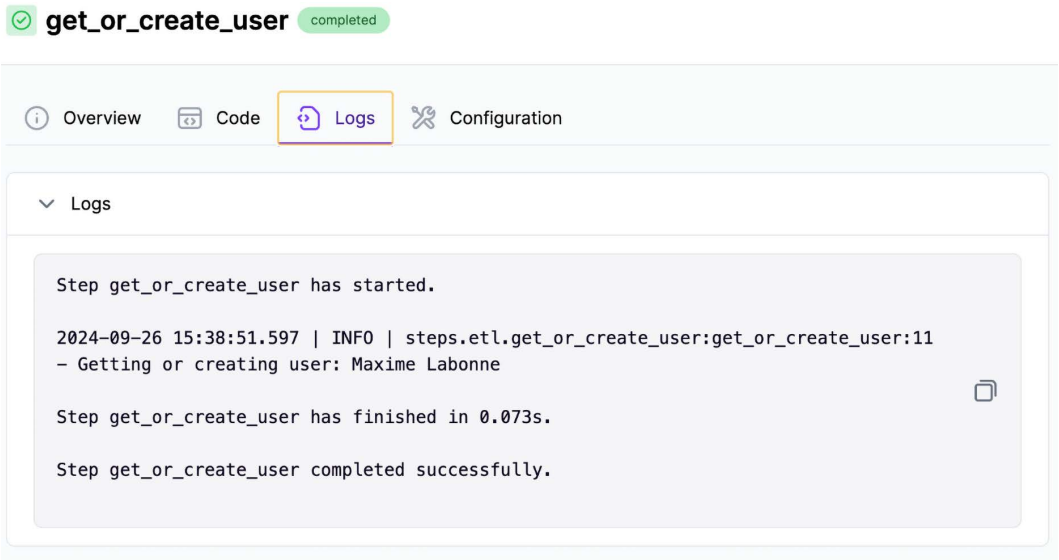

Figure 2.5: Example of insights from a specific step of the digital_data_etl pipeline run

Now that we understand how to define a ZenML pipeline and how to look it up in the dashboard, let's quickly look at how to define a ZenML step. In the code snippet below, we defined the `get_or_create_user()` step, which works just like a normal Python function but is decorated with `@step`. We won't go into the details of the logic, as we will cover the ETL logic in *Chapter 3*. For now, we will focus only on the ZenML functionality.

```
from loguru import logger
from typing_extensions import Annotated
from zenml import get_step_context, step

from llm_engineering.application import utils
```

```
from llm_engineering.domain.documents import UserDocument


@step
def get_or_create_user(user_full_name: str) -> Annotated[UserDocument,
"user"]:
    logger.info(f"Getting or creating user: {user_full_name}")

    first_name, last_name = utils.split_user_full_name(user_full_name)

    user = UserDocument.get_or_create(first_name=first_name, last_
name=last_name)


    return user
```

Within a ZenML step, you can define any Python logic your use case needs. In this simple example, we are just creating or retrieving a user, but we could replace that code with anything, starting from data collection to feature engineering and training. What is essential to notice is that to integrate ZenML with your code, you have to write modular code, where each function does just one thing. The modularity of your code makes it easy to decorate your functions with `@step` and then glue multiple steps together within a main function decorated with `@pipeline`. One design choice that will impact your application is deciding the granularity of each step, as each will run as a different unit on a different machine when deployed in the cloud.

To decouple our code from ZenML, we encapsulated all the application and domain logic into the `llm_engineering` Python module. We also defined the `pipelines` and `steps` folders, where we defined our ZenML logic. Within the `steps` module, we only used what we needed from the `llm_engineering` Python module (similar to how you use a Python package). In the `pipelines` module, we only aggregated ZenML steps to glue them into the final pipeline. Using this design, we can easily swap ZenML with another orchestrator or use our application logic in other use cases, such as a REST API. We only have to replace the ZenML code without touching the `llm_engineering` module where all our logic resides.

This folder structure is reflected at the root of the LLM-Engineers-Handbook repository, as illustrated in *Figure 2.6*:

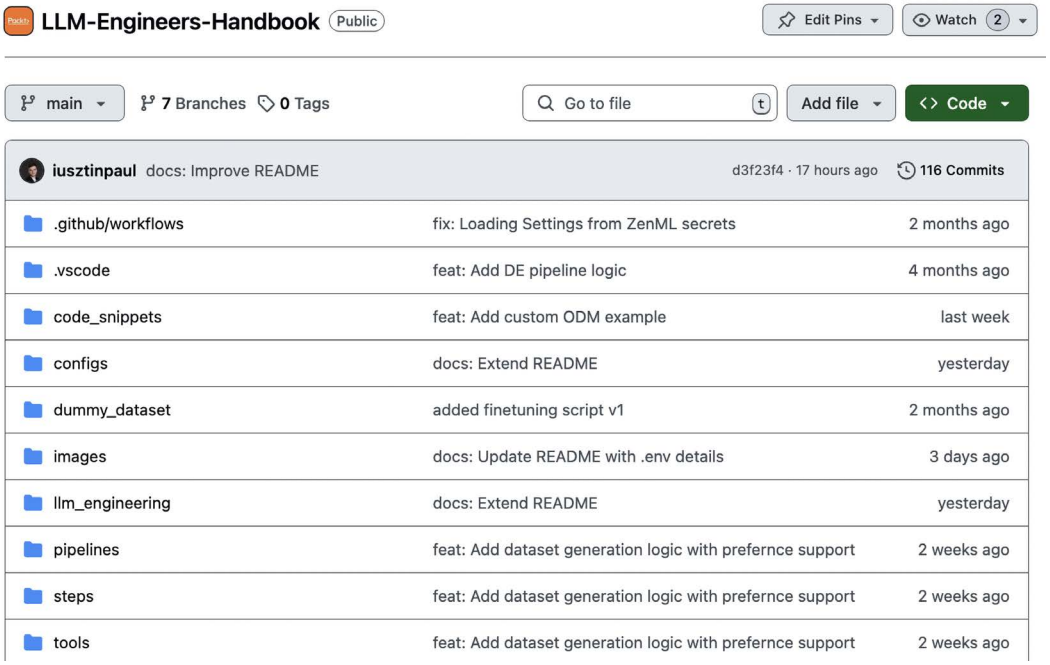

Figure 2.6: LLM-Engineers-Handbook repository folder structure

One last thing to consider when writing ZenML steps is that if you return a value, it should be serializable. ZenML can serialize most objects that can be reduced to primitive data types, but there are a few exceptions. For example, we used UUID types as IDs throughout the code, which aren't natively supported by ZenML. Thus, we had to extend ZenML's materializer to support UUIDs. We raised this issue to ZenML. Hence, in future ZenML versions, UUIDs will be supported, but it was an excellent example of the serialization aspect of transforming function outputs in artifacts.

Artifacts and metadata

As mentioned in the previous section, ZenML transforms any step output into an artifact. First, let's quickly understand what an artifact is. In MLOps, an **artifact** is any file(s) produced during the machine learning lifecycle, such as datasets, trained models, checkpoints, or logs. Artifacts are crucial for reproducing experiments and deploying models. We can transform anything into an artifact. For example, the model registry is a particular use case for an artifact. Thus, artifacts have these unique properties: they are versioned, sharable, and have metadata attached to them to understand what's inside quickly. For example, when wrapping your dataset with an artifact, you can add to its metadata the size of the dataset, the train-test split ratio, the size, types of labels, and anything else useful to understand what's inside the dataset without actually downloading it.

Let's circle back to our **digital_data_etl** pipeline example, where we had as a step output an artifact, the crawled links, which are an artifact, as seen in *Figure 2.7*

Figure 2.7: ZenML artifact example using the digital_data_etl pipeline as an example

By clicking on the `crawled_links` artifact and navigating to the **Metadata** tab, we can quickly see all the domains we crawled for a particular author, the number of links we crawled for each domain, and how many were successful, as illustrated in *Figure 2.8*:

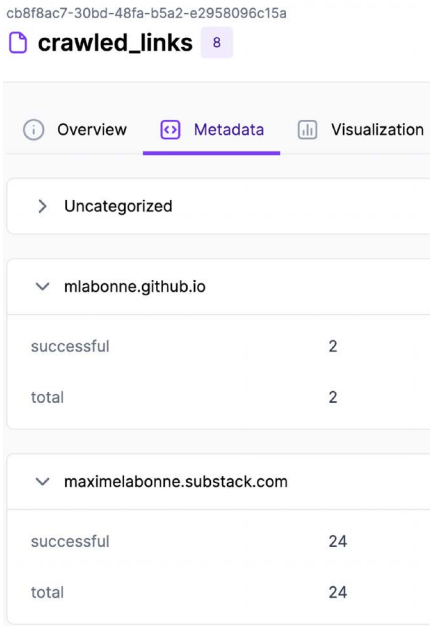

Figure 2.8: ZenML metadata example using the digital_data_etl pipeline as an example

A more interesting example of an artifact and its metadata is the generated dataset artifact. In *Figure 2.9*, we can visualize the metadata of the `instruct_datasets` artifact, which was automatically generated and will be used to fine-tune the LLM Twin model. More details on the `instruction datasets` are in *Chapter 5*. For now, we want to highlight that within the dataset's metadata, we have precomputed a lot of helpful information about it, such as how many data categories it contains, its storage size, and the number of samples per training and testing split.

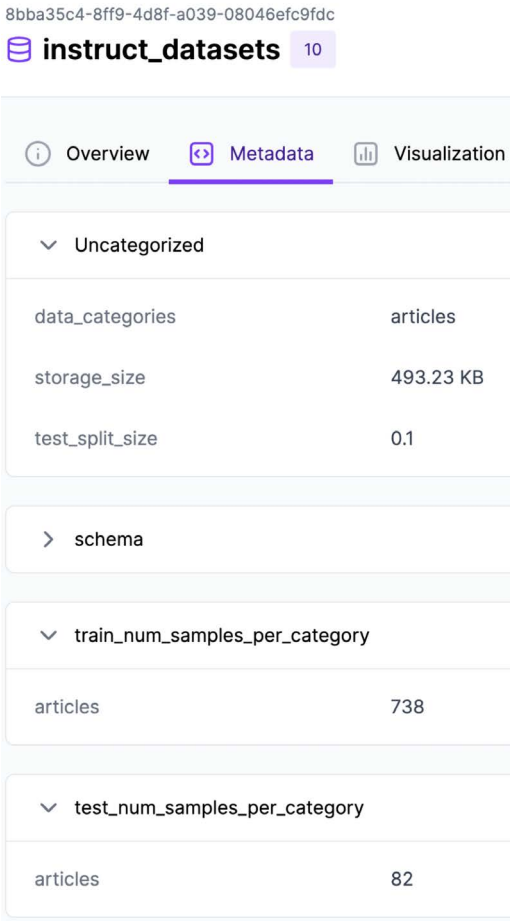

Figure 2.9: ZenML metadata example for the instruct_datasets artifact

The metadata is manually added to the artifact, as shown in the code snippet below. Thus, you can precompute and attach to the artifact's metadata anything you consider helpful for dataset discovery across your business and projects:

```
… # More imports
from zenml import ArtifactConfig, get_step_context, step


@step
def generate_intruction_dataset(
    prompts: Annotated[dict[DataCategory,
list[GenerateDatasetSamplesPrompt]], "prompts"]) -> Annotated[
```

```
    InstructTrainTestSplit,
    ArtifactConfig(
        name="instruct_datasets",
        tags=["dataset", "instruct", "cleaned"],
    ),
]:
    datasets = … # Generate datasets

    step_context = get_step_context()
    step_context.add_output_metadata(output_name="instruct_datasets",
metadata=_get_metadata_instruct_dataset(datasets))

    return datasets

def _get_metadata_instruct_dataset(datasets: InstructTrainTestSplit) ->
dict[str, Any]:
    instruct_dataset_categories = list(datasets.train.keys())
    train_num_samples = {
        category: instruct_dataset.num_samples for category, instruct_
dataset in datasets.train.items()
    }
    test_num_samples = {category: instruct_dataset.num_samples for
category, instruct_dataset in datasets.test.items()}

    return {
        "data_categories": instruct_dataset_categories,
        "test_split_size": datasets.test_split_size,
        "train_num_samples_per_category": train_num_samples,
        "test_num_samples_per_category": test_num_samples,
    }
```

Also, you can easily download and access a specific version of the dataset using its **Universally Unique Identifier (UUID)**, which you can find using the ZenML dashboard or CLI:

```
from zenml.client import Client

artifact = Client().get_artifact_version('8bba35c4-8ff9-4d8f-a039-
08046efc9fdc')
loaded_artifact = artifact.load()
```

The last step in exploring ZenML is understanding how to run and configure a ZenML pipeline.

How to run and configure a ZenML pipeline

All the ZenML pipelines can be called from the `run.py` file, accessed at `tools/run.py` in our GitHub repository. Within the `run.py` file, we implemented a simple CLI that allows you to specify what pipeline to run. For example, to call the `digital_data_etl` pipeline to crawl Maxime's content, you have to run:

```
python -m tools.run --run-etl --no-cache --etl-config-filename digital_
data_etl_maxime_labonne.yaml
```

Or, to crawl Paul's content, you can run:

```
python -m tools.run --run-etl --no-cache --etl-config-filename digital_
data_etl_paul_iusztin.yaml
```

As explained when introducing Poe the Poet, all our CLI commands used to interact with the project will be executed through Poe to simplify and standardize the project. Thus, we encapsulated these Python calls under the following poe CLI commands:

```
poetry poe run-digital-data-etl-maxime
poetry poe run-digital-data-etl-paul
```

We only change the ETL config file name when scraping content for different people. ZenML allows us to inject specific configuration files at runtime as follows:

```
config_path = root_dir / "configs" / etl_config_filename
assert config_path.exists(), f"Config file not found: { config_path }"
run_args_etl = {
    "config_path": config_path,
    "run_name": f"digital_data_etl_run_{dt.now().
strftime('%Y_%m_%d_%H_%M_%S')}"
}
 digital_data_etl.with_options()(**run_args_etl)
```

In the config file, we specify all the parameters that will input the pipeline as parameters. For example, the `configs/digital_data_etl_maxime_labonne.yaml` configuration file looks as follows:

```
parameters:
  user_full_name: Maxime Labonne # [First Name(s)] [Last Name]
  links:
    # Personal Blog
```

```
    - https://mlabonne.github.io/blog/posts/2024-07-29_Finetune_Llama31.
html
    - https://mlabonne.github.io/blog/posts/2024-07-15_The_Rise_of_
Agentic_Data_Generation.html
    # Substack
    - https://maximelabonne.substack.com/p/uncensor-any-llm-with-
abliteration-d30148b7d43e
    … # More links
```

Where the `digital_data_etl` function signature looks like this:

```
@pipeline
def digital_data_etl(user_full_name: str, links: list[str]) -> str:
```

This approach allows us to configure each pipeline at runtime without modifying the code. We can also clearly track the inputs for all our pipelines, ensuring reproducibility. As seen in *Figure 2.10*, we have one or more configs for each pipeline.

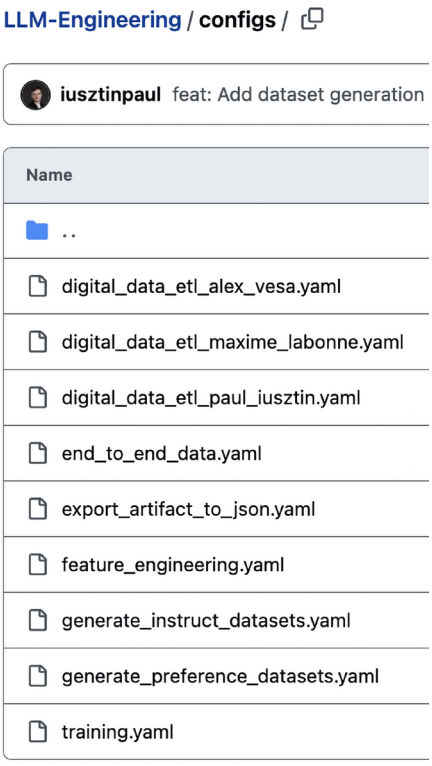

Figure 2.10: ZenML pipeline configs

Other popular orchestrators similar to ZenML that we've personally tested and consider powerful are Airflow, Prefect, Metaflow, and Dagster. Also, if you are a heavy user of Kubernetes, you can opt for Agro Workflows or Kubeflow, the latter of which works only on top of Kubernetes. We still consider ZenML the best trade-off between ease of use, features, and costs. Also, none of these tools offer the stack feature that is offered by ZenML, which allows it to avoid vendor-locking you in to any cloud ecosystem.

In *Chapter 11*, we will explore in more depth how to leverage an orchestrator to implement MLOps best practices. But now that we understand ZenML, what it is helpful for, and how to use it, let's move on to the experiment tracker.

Comet ML: experiment tracker

Training ML models is an entirely iterative and experimental process. Unlike traditional software development, it involves running multiple parallel experiments, comparing them based on pre-defined metrics, and deciding which one should advance to production. An experiment tracking tool allows you to log all the necessary information, such as metrics and visual representations of your model predictions, to compare all your experiments and quickly select the best model. Our LLM project is no exception.

As illustrated in *Figure 2.11*, we used Comet to track metrics such as training and evaluation loss or the value of the gradient norm across all our experiments.

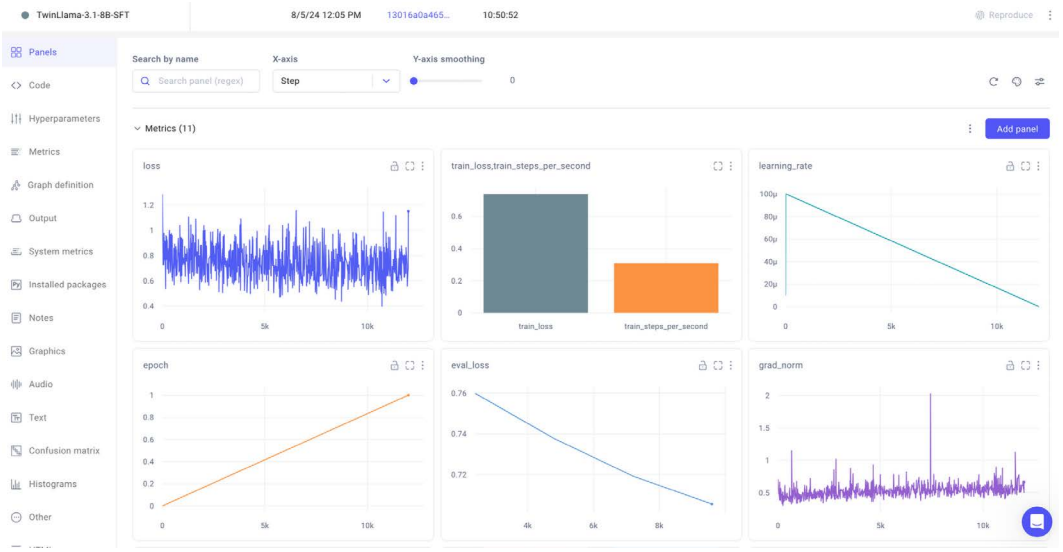

Figure 2.11: Comet ML training metrics example

Using an experiment tracker, you can go beyond training and evaluation metrics and log your training hyperparameters to track different configurations between experiments.

It also logs out-of-the-box system metrics such as GPU, CPU, or memory utilization to give you a clear picture of what resources you need during training and where potential bottlenecks slow down your training, as seen in *Figure 2.12*.

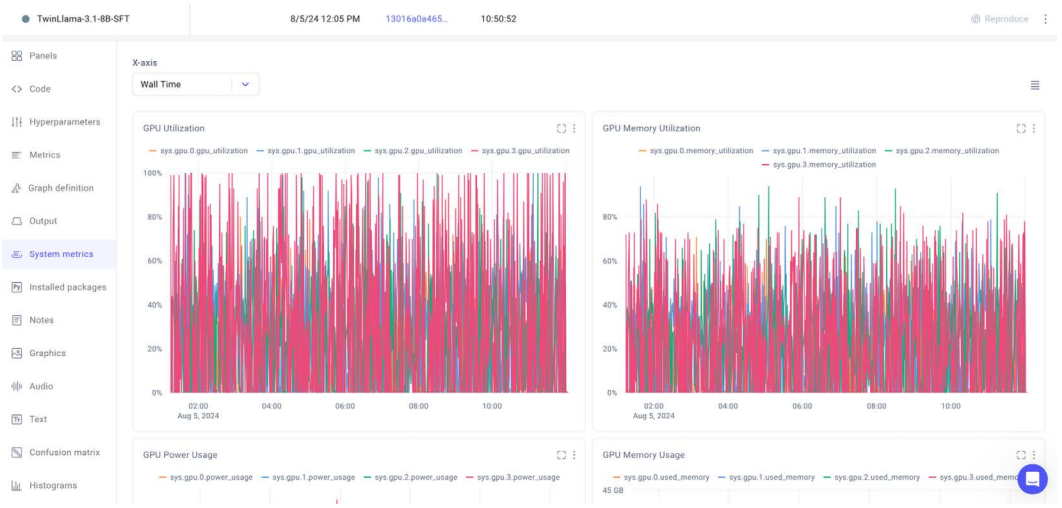

Figure 2.12: Comet ML system metrics example

You don't have to set up Comet locally. We will use their online version for free without any constraints throughout this book. Also, if you want to look more in-depth into the Comet ML experiment tracker, we made the training experiments tracked with Comet ML public while fine-tuning our LLM Twin models. You can access them here: `https://www.comet.com/mlabonne/llm-twin-training/view/new/panels`.

Other popular experiment trackers are W&B, MLflow, and Neptune. We've worked with all of them and can state that they all have mostly the same features, but Comet ML differentiates itself through its ease of use and intuitive interface. Let's move on to the final piece of the MLOps puzzle: Opik for prompt monitoring.

Opik: prompt monitoring

You cannot use standard tools and techniques when logging and monitoring prompts. The reason for this is complicated. We will dig into it in *Chapter 11*. However, to quickly give you some understanding, you cannot use standard logging tools as prompts are complex and unstructured chains.

When interacting with an LLM application, you chain multiple input prompts and the generated output into a trace, where one prompt depends on previous prompts.

Thus, instead of plain text logs, you need an intuitive way to group these traces into a specialized dashboard that makes debugging and monitoring traces of prompts easier.

We used Opik, an open-source tool made by Comet, as our prompt monitoring tool because it follows Comet's philosophy of simplicity and ease of use, which is currently relatively rare in the LLM landscape. Other options offering similar features are Langfuse (open source, `https://langfuse.com`), Galileo (not open source, `rungalileo.io`), and LangSmith (not open source, `https://www.langchain.com/langsmith`), but we found their solutions more cumbersome to use and implement. Opik, along with its serverless option, also provides a free open-source version that you have complete control over. You can read more on Opik at `https://github.com/comet-ml/opik`.

Databases for storing unstructured and vector data

We also want to present the NoSQL and vector databases we will use within our examples. When working locally, they are already integrated through Docker. Thus, when running `poetry poe local-infrastructure-up`, as instructed a few sections above, local images of Docker for both databases will be pulled and run on your machine. Also, when deploying the project, we will show you how to use their serverless option and integrate it with the rest of the LLM Twin project.

MongoDB: NoSQL database

MongoDB is one of today's most popular, robust, fast, and feature-rich NoSQL databases. It integrates well with most cloud ecosystems, such as AWS, Google Cloud, Azure, and Databricks. Thus, using MongoDB as our NoSQL database was a no-brainer.

When we wrote this book, MongoDB was used by big players such as Novo Nordisk, Delivery Hero, Okta, and Volvo. This widespread adoption suggests that MongoDB will remain a leading NoSQL database for a long time.

We use MongoDB as a NoSQL database to store the raw data we collect from the internet before processing it and pushing it into the vector database. As we work with unstructured text data, the flexibility of the NoSQL database fits like a charm.

Qdrant: vector database

Qdrant (`https://qdrant.tech/`) is one of the most popular, robust, and feature-rich vector databases. We could have used almost any vector database for our small MVP, but we wanted to pick something light and likely to be used in the industry for many years to come.

We will use Qdrant to store the data from MongoDB after it's processed and transformed for GenAI usability.

Qdrant is used by big players such as X (formerly Twitter), Disney, Microsoft, Discord, and Johnson & Johnson. Thus, it is highly probable that Qdrant will remain in the vector database game for a long time.

While writing the book, other popular options were Milvus, Redis, Weaviate, Pinecone, Chroma, and pgvector (a PostgreSQL plugin for vector indexes). We found that Qdrant offers the best trade-off between RPS, latency, and index time, making it a solid choice for many generative AI applications.

Comparing all the vector databases in detail could be a chapter in itself. We don't want to do that here. Still, if curious, you can check the *Vector DB Comparison* resource from Superlinked at `https://superlinked.com/vector-db-comparison`, which compares all the top vector databases in terms of everything you can think about, from the license and release year to database features, embedding models, and frameworks supported.

Preparing for AWS

This last part of the chapter will focus on setting up an AWS account (if you don't already have one), an AWS access key, and the CLI. Also, we will look into what SageMaker is and why we use it.

We picked AWS as our cloud provider because it's the most popular out there and the cloud in which we (the writers) have the most experience. The reality is that other big cloud providers, such as GCP or Azure, offer similar services. Thus, depending on your specific application, there is always a trade-off between development time (in which you have the most experience), features, and costs. But for our MVP, AWS, it's the perfect option as it provides robust features for everything we need, such as S3 (object storage), ECR (container registry), and SageMaker (compute for training and inference).

Setting up an AWS account, an access key, and the CLI

As AWS could change its UI/UX, the best way to instruct you on how to create an AWS account is by redirecting you to their official tutorial: `https://docs.aws.amazon.com/accounts/latest/reference/manage-acct-creating.html`.

After successfully creating an AWS account, you can access the AWS console at `http://console.aws.amazon.com`. Select **Sign in using root user email** (found under the **Sign in** button), then enter your account's email address and password.

Next, we must generate access keys to access AWS programmatically. The best option to do so is first to create an IAM user with administrative access as described in this AWS official tutorial: `https://docs.aws.amazon.com/streams/latest/dev/setting-up.html`

For production accounts, it is best practice to grant permissions with a policy of least privilege, giving each user only the permissions they require to perform their role. However, to simplify the setup of our test account, we will use the `AdministratorAccess` managed policy, which gives our user full access, as explained in the tutorial above and illustrated in *Figure 2.13*.

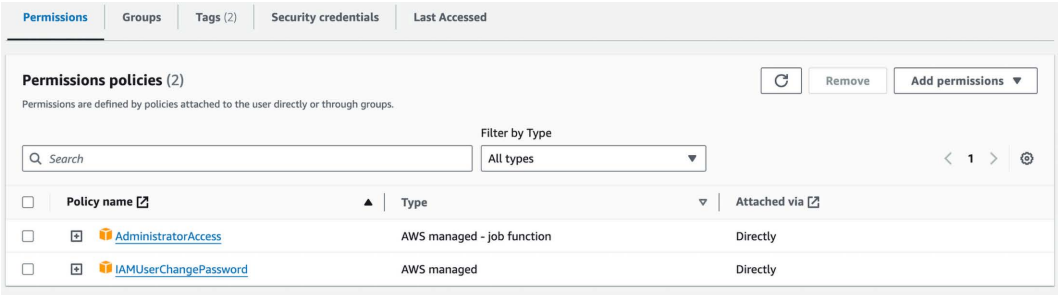

Figure 2.13: IAM user permission policies example

Next, you have to create an access key for the IAM user you just created using the following tutorial: `https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html`.

The access keys will look as follows:

```
aws_access_key_id = <your_access_key_id>
aws_secret_access_key = <your_secret_access_key>
```

Just be careful to store them somewhere safe, as you won't be able to access them after you create them. Also, be cautious with who you share them, as they could be used to access your AWS account and manipulate various AWS resources.

The last step is to install the AWS CLI and configure it with your newly created access keys. You can install the AWS CLI using the following link: `https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html`.

After installing the AWS CLI, you can configure it by running `aws configure`. Here is an example of our AWS configuration:

```
[default]
aws_access_key_id = ************
aws_secret_access_key = ************
```

```
region = eu-central-1
output = json
```

For more details on how to configure the AWS CLI, check out the following tutorial: `https://docs.aws.amazon.com/cli/v1/userguide/cli-configure-files.html`.

Also, to configure the project with your AWS credentials, you must fill in the following variables within your `.env` file:

```
AWS_REGION="eu-central-1" # Change it with your AWS region. By default, we
use "eu-central-1".
AWS_ACCESS_KEY="<your_aws_access_key>"
AWS_SECRET_KEY="<your_aws_secret_key>"
```

An important note about costs associated with hands-on tasks in this book

All the cloud services used across the book stick to their freemium option, except AWS. Thus, if you use a personal AWS account, you will be responsible for AWS costs as you follow along in this book. While some services may fall under AWS Free Tier usage, others will not. Thus, you are responsible for checking your billing console regularly.

Most of the costs will come when testing SageMaker for training and inference. Based on our tests, the AWS costs can vary between $50 and $100 using the specifications provided in this book and repository.

See the AWS documentation on setting up billing alarms to monitor your costs at `https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html`.

SageMaker: training and inference compute

The last topic of this chapter is understanding SageMaker and why we decided to use it. SageMaker is an ML platform used to train and deploy ML models. An official definition is as follows: AWS SageMaker is a fully managed machine learning service by AWS that enables developers and data scientists to build, train, and deploy machine learning models at scale. It simplifies the process by handling the underlying infrastructure, allowing users to focus on developing high-quality models efficiently.

We will use SageMaker to fine-tune and operationalize our training pipeline on clusters of GPUs and to deploy our custom LLM Twin model as a REST API that can be accessed in real time from anywhere in the world.

Why AWS SageMaker?

We must also discuss why we chose AWS SageMaker over simpler and more cost-effective options, such as AWS Bedrock. First, let's explain Bedrock and its benefits.

Amazon Bedrock is a serverless solution for deploying LLMs. Serverless means that there are no servers or infrastructure to manage. It provides pre-trained models, which you can access directly through API calls. When we wrote this book, they provided support only for Mistral, Flan, Llama 2, and Llama 3 (quite a limited list of options). You can send input data and receive predictions from the models without managing the underlying infrastructure or software. This approach significantly reduces the complexity and time required to integrate AI capabilities into applications, making it more accessible to developers with limited machine learning expertise. However, this ease of integration comes at the cost of limited customization options, as you're restricted to the pre-trained models and APIs provided by Amazon Bedrock. In terms of pricing, Bedrock uses a simple pricing model based on the number of API calls. This straightforward pricing structure makes it more efficient to estimate and control costs.

Meanwhile, SageMaker provides a comprehensive platform for building, training, and deploying machine learning models. It allows you to customize your ML processes entirely or even use the platform for research. That's why SageMaker is mainly used by data scientists and machine learning experts who know how to program, understand machine learning concepts, and are comfortable working with cloud platforms such as AWS. SageMaker is a double-edged sword regarding costs, following a pay-as-you-go pricing model similar to most AWS services. This means you have to pay for the usage of computing resources, storage, and any other services required to build your applications.

In contrast to Bedrock, even if the SageMaker endpoint is not used, you will still pay for the deployed resources on AWS, such as online EC2 instances. Thus, you have to design autoscaling systems that delete unused resources. To conclude, Bedrock offers an out-of-the-box solution that allows you to quickly deploy an API endpoint powered by one of the available foundation models. Meanwhile, SageMaker is a multi-functional platform enabling you to customize your ML logic fully.

So why did we choose SageMaker over Bedrock? Bedrock would have been an excellent solution for quickly prototyping something, but this is a book on LLM engineering, and our goal is to dig into all the engineering aspects that Bedrock tries to mask away. Thus, we chose SageMaker because of its high level of customizability, allowing us to show you all the engineering required to deploy a model.

In reality, even SageMaker isn't fully customizable. If you want complete control over your deployment, use EKS, AWS's Kubernetes self-managed service. In this case, you have direct access to the virtual machines, allowing you to fully customize how you build your ML pipelines, how they interact, and how you manage your resources. You could do the same thing with AWS ECS, AWS's version of Kubernetes. Using EKS or ECS, you could also reduce the costs, as these services cost considerably less.

To conclude, SageMaker strikes a balance between complete control and customization and a fully managed service that hides all the engineering complexity behind the scenes. This balance ensures that you have the control you need while also benefiting from the managed service's convenience.

Summary

In this chapter, we reviewed the core tools used across the book. First, we understood how to install the correct version of Python that supports our repository. Then, we looked over how to create a virtual environment and install all the dependencies using Poetry. Finally, we understood how to use a task execution tool like Poe the Poet to aggregate all the commands required to run the application.

The next step was to review all the tools used to ensure MLOps best practices, such as a model registry to share our models, an experiment tracker to manage our training experiments, an orchestrator to manage all our ML pipelines and artifacts, and metadata to manage all our files and datasets. We also understood what type of databases we need to implement the LLM Twin use case. Finally, we explored the process of setting up an AWS account, generating an access key, and configuring the AWS CLI for programmatic access to the AWS cloud. We also gained a deep understanding of AWS SageMaker and the reasons behind choosing it to build our LLM Twin application.

In the next chapter, we will explore the implementation of the LLM Twin project by starting with the data collection ETL that scrapes posts, articles, and repositories from the internet and stores them in a data warehouse.

References

- Acsany, P. (2024, February 19). *Dependency Management With Python Poetry*. https://realpython.com/dependency-management-python-poetry/
- Comet.ml. (n.d.). *comet-ml/opik: Open-source end-to-end LLM Development Platform*. GitHub. https://github.com/comet-ml/opik
- Czakon, J. (2024, September 25). *ML Experiment Tracking: What It Is, Why It Matters, and How to Implement It*. neptune.ai. https://neptune.ai/blog/ml-experiment-tracking
- Hopsworks. (n.d.). *ML Artifacts (ML Assets)?* Hopsworks. https://www.hopsworks.ai/dictionary/ml-artifacts
- *Introduction | Documentation | Poetry – Python dependency management and packaging made easy*. (n.d.). https://python-poetry.org/docs
- Jones, L. (2024, March 21). *Managing Multiple Python Versions With pyenv*. https://realpython.com/intro-to-pyenv/
- Kaewsanmua, K. (2024, January 3). *Best Machine Learning Workflow and Pipeline Orchestration Tools*. neptune.ai. https://neptune.ai/blog/best-workflow-and-pipeline-orchestration-tools
- MongoDB. (n.d.). *What is NoSQL?* NoSQL databases explained. https://www.mongodb.com/resources/basics/databases/nosql-explained
- Nat-N. (n.d.). *nat-n/poethepoet: A task runner that works well with poetry.* GitHub. https://github.com/nat-n/poethepoet
- Oladele, S. (2024, August 29). *ML Model Registry: The Ultimate Guide*. neptune.ai. https://neptune.ai/blog/ml-model-registry
- Schwaber-Cohen, R. (n.d.). *What is a Vector Database & How Does it Work? Use Cases + Examples*. Pinecone. https://www.pinecone.io/learn/vector-database/
- *Starter guide | ZenML Documentation*. (n.d.). https://docs.zenml.io/user-guide/starter-guide
- *Vector DB Comparison*. (n.d.). https://superlinked.com/vector-db-comparison

Join our book's Discord space

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

`https://packt.link/llmeng`

3

Data Engineering

This chapter will begin exploring the LLM Twin project in more depth. We will learn how to design and implement the data collection pipeline to gather the raw data we will use in all our LLM use cases, such as fine-tuning or inference. As this is not a book on data engineering, we will keep this chapter short and focus only on what is strictly necessary to collect the required raw data. Starting with *Chapter 4*, we will concentrate on LLMs and GenAI, exploring its theory and concrete implementation details.

When working on toy projects or doing research, you usually have a static dataset with which you work. But in our LLM Twin use case, we want to mimic a real-world scenario where we must gather and curate the data ourselves. Thus, implementing our data pipeline will connect the dots regarding how an end-to-end ML project works. This chapter will explore how to design and implement an **Extract, Transform, Load** (**ETL**) pipeline that crawls multiple social platforms, such as Medium, Substack, or GitHub, and aggregates the gathered data into a MongoDB data warehouse. We will show you how to implement various crawling methods, standardize the data, and load it into a data warehouse.

We will begin by designing the LLM Twin's data collection pipeline and explaining the architecture of the ETL pipeline. Afterward, we will move directly to implementing the pipeline, starting with ZenML, which will orchestrate the entire process. We will investigate the crawler implementation and understand how to implement a dispatcher layer that instantiates the right crawler class based on the domain of the provided link while following software best practices. Next, we will learn how to implement each crawler individually. Also, we will show you how to implement a data layer on top of MongoDB to structure all our documents and interact with the database.

Finally, we will explore how to run the data collection pipeline using ZenML and query the collected data from MongoDB.

Thus, in this chapter, we will study the following topics:

- Designing the LLM Twin's data collection pipeline
- Implementing the LLM Twin's data collection pipeline
- Gathering raw data into the data warehouse

By the end of this chapter, you will know how to design and implement an ETL pipeline to extract, transform, and load raw data ready to be ingested into the ML application.

Designing the LLM Twin's data collection pipeline

Before digging into the implementation, we must understand the LLM Twin's data collection ETL architecture, illustrated in *Figure 3.1*. We must explore what platforms we will crawl to extract data from and how we will design our data structures and processes. However, the first step is understanding how our data collection pipeline maps to an ETL process.

An ETL pipeline involves three fundamental steps:

1. We **extract** data from various sources. We will crawl data from platforms like Medium, Substack, and GitHub to gather raw data.
2. We **transform** this data by cleaning and standardizing it into a consistent format suitable for storage and analysis.
3. We **load** the transformed data into a data warehouse or database.

For our project, we use MongoDB as our NoSQL data warehouse. Although this is not a standard approach, we will explain the reasoning behind this choice shortly.

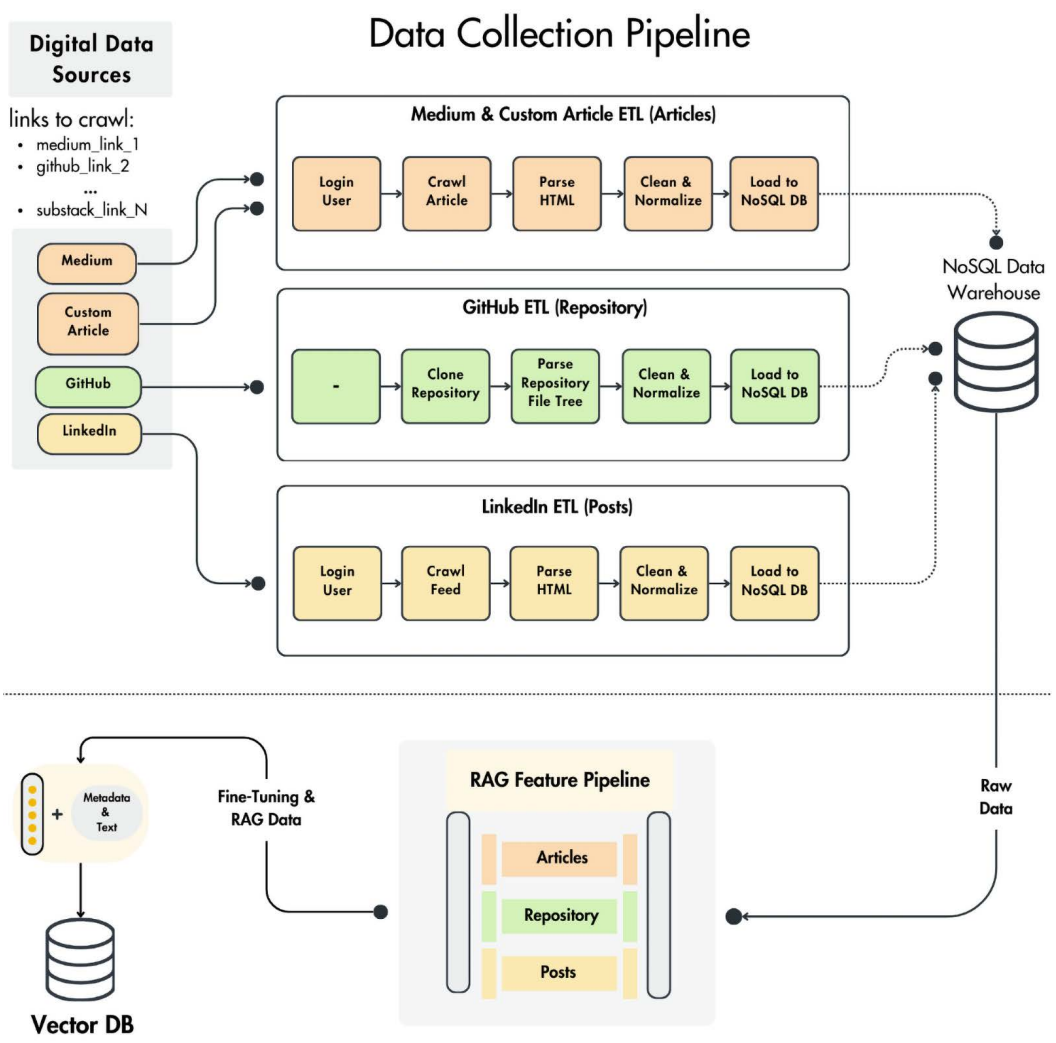

Figure 3.1: LLM Twin's data collection ETL pipeline architecture

We want to design an ETL pipeline that inputs a user and a list of links as input. Afterward, it crawls each link individually, standardizes the collected content, and saves it under that specific author in a MongoDB data warehouse.

Hence, the signature of the data collection pipeline will look as follows:

- **Input:** A list of links and their associated user (the author)
- **Output:** A list of raw documents stored in the NoSQL data warehouse

We will use `user` and `author` interchangeably, as in most scenarios across the ETL pipeline, a user is the author of the extracted content. However, within the data warehouse, we have only a user collection.

The ETL pipeline will detect the domain of each link, based on which it will call a specialized crawler. We implemented four different crawlers for three different data categories, as seen in *Figure 3.2*. First, we will explore the three fundamental data categories we will work with across the book. All our collected documents can be boiled down to an article, repository (or code), and post. It doesn't matter where the data comes from. We are primarily interested in the document's format. In most scenarios, we will have to process these data categories differently. Thus, we created a different domain entity for each, where each entity will have its class and collection in MongoDB. As we save the source URL within the document's metadata, we will still know its source and can reference it in our GenAI use cases.

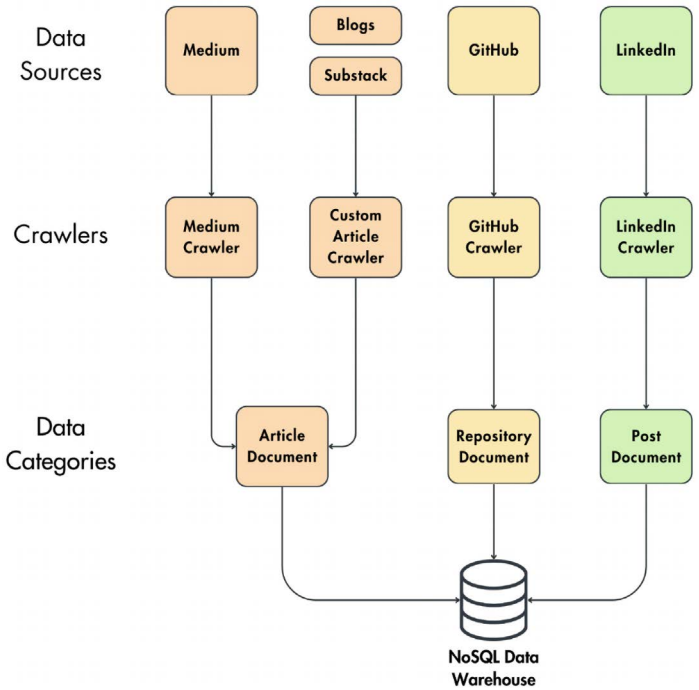

Figure 3.2: The relationship between the crawlers and the data categories

Our codebase supports four different crawlers:

- **Medium crawler**: Used to collect data from Medium. It outputs an article document. It logs in to Medium and crawls the HTML of the article's link. Then, it extracts, cleans, and normalizes the text from the HTML and loads the standardized text of the article into the NoSQL data warehouse.
- **Custom article crawler**: It performs similar steps to the Medium crawler but is a more generic implementation for collecting articles from various sites. Thus, as it doesn't implement any particularities of any platform, it doesn't perform the login step and blindly gathers all the HTML from a particular link. This is enough for articles freely available online, which you can find on Substack and people's blogs. We will use this crawler as a safety net when the link's domain isn't associated with the other supported crawlers. For example, when providing a Substack link, it will default to the custom article crawler, but when providing a Medium URL, it will use the Medium crawler.
- **GitHub crawler**: This collects data from GitHub. It outputs a repository document. It clones the repository, parses the repository file tree, cleans and normalizes the files, and loads them to the database.
- **LinkedIn crawler**: This is used to collect data from LinkedIn. It outputs multiple post documents. It logs in to LinkedIn, navigates to the user's feed, and crawls all the user's latest posts. For each post, it extracts its HTML, cleans and normalizes it, and loads it to MongoDB.

In the next section, we will examine each crawler's implementation in detail. For now, note that each crawler accesses a specific platform or site in a particular way and extracts HTML from it. Afterward, all the crawlers parse the HTML, extract the text from it, and clean and normalize it so it can be stored in the data warehouse under the same interface.

By reducing all the collected data to three data categories and not creating a new data category for every new data source, we can easily extend this architecture to multiple data sources with minimal effort. For example, if we want to start collecting data from X, we only have to implement a new crawler that outputs a post document, and that's it. The rest of the code will remain untouched. Otherwise, if we introduced the source dimension in the class and document structure, we would have to add code to all downstream layers to support any new data source. For example, we would have to implement a new document class for each new source and adapt the feature pipeline to support it.

For our proof of concept, crawling a few hundred documents is enough, but if we want to scale it to a real-world product, we would probably need more data sources to crawl from. LLMs are data-hungry. Thus, you need thousands of documents for ideal results instead of just a few hundred. But in many projects, it's an excellent strategy to implement an end-to-end project version that isn't the most accurate and iterate through it later. Thus, by using this architecture, you can easily add more data sources in future iterations to gather a larger dataset. More on LLM fine-tuning and dataset size will be covered in the next chapter.

How is the ETL process connected to the feature pipeline? The feature pipeline ingests the raw data from the MongoDB data warehouse, cleans it further, processes it into features, and stores it in the Qdrant vector DB to make it accessible for the LLM training and inference pipelines. *Chapter 4* provides more information on the feature pipeline. The ETL process is independent of the feature pipeline. The two pipelines communicate with each other strictly through the MongoDB data warehouse. Thus, the data collection pipeline can write data for MongoDB, and the feature pipeline can read from it independently and on different schedules.

Why did we use MongoDB as a data warehouse? Using a transactional database, such as MongoDB, as a data warehouse is uncommon. However, in our use case, we are working with small amounts of data, which MongoDB can handle. Even if we plan to compute statistics on top of our MongoDB collections, it will work fine at the scale of our LLM Twin's data (hundreds of documents). We picked MongoDB to store our raw data primarily because of the nature of our unstructured data: text crawled from the internet. By mainly working with unstructured text, selecting a NoSQL database that doesn't enforce a schema made our development easier and faster. Also, MongoDB is stable and easy to use. Their Python SDK is intuitive. They provide a Docker image that works out of the box locally and a cloud freemium tier that is perfect for proofs of concept, such as the LLM Twin. Thus, we can freely work with it locally and in the cloud. However, when working with big data (millions of documents or more), using a dedicated data warehouse such as Snowflake or BigQuery will be ideal.

Now that we've understood the architecture of the LLM Twin's data collection pipeline, let's move on to its implementation.

Implementing the LLM Twin's data collection pipeline

As we presented in *Chapter 2*, the entry point to each pipeline from our LLM Twin project is a ZenML pipeline, which can be configured at runtime through YAML files and run through the ZenML ecosystem. Thus, let's start by looking into the ZenML `digital_data_etl` pipeline. You'll notice that this is the same pipeline we used as an example in *Chapter 2* to illustrate ZenML. But this time, we will dig deeper into the implementation, explaining how the data collection works behind the scenes. After understanding how the pipeline works, we will explore the implementation of each crawler used to collect data from various sites and the MongoDB documents used to store and query data from the data warehouse.

ZenML pipeline and steps

In the code snippet below, we can see the implementation of the ZenML `digital_data_etl` pipeline, which inputs the user's full name and a list of links that will be crawled under that user (considered the author of the content extracted from those links). Within the function, we call two steps. In the first one, we look up the user in the database based on its full name. Then, we loop through all the links and crawl each independently. The pipeline's implementation is available in our repository at `pipelines/digital_data_etl.py`.

```
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]) -> str:
    user = get_or_create_user(user_full_name)
    last_step = crawl_links(user=user, links=links)

    return last_step.invocation_id
```

Figure 3.3 shows a run of the `digital_data_etl` pipeline on the ZenML dashboard. The next phase is to explore the `get_or_create_user` and `crawl_links` ZenML steps individually. The step implementation is available in our repository at `steps/etl`.

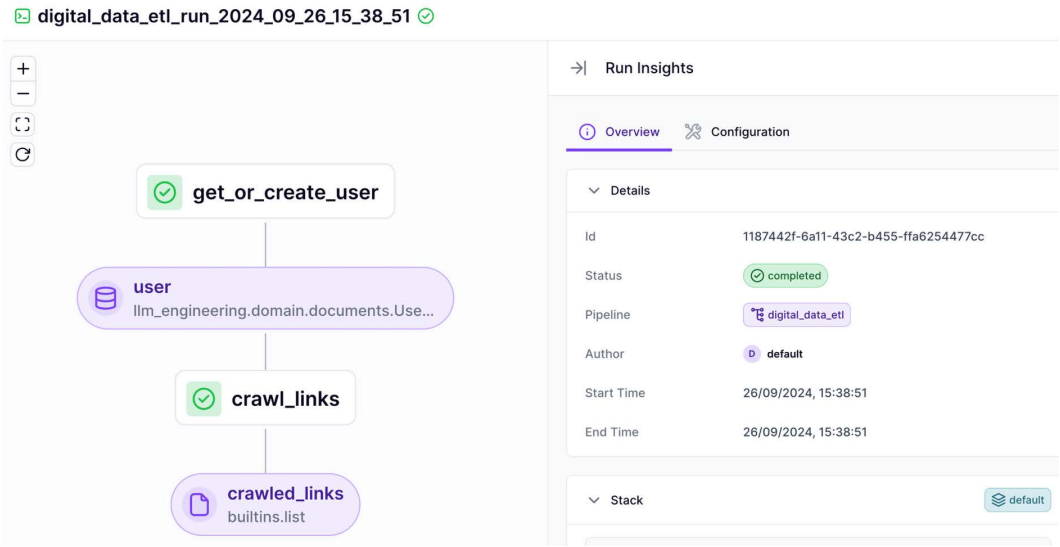

Figure 3.3: Example of a digital_data_etl pipeline run from ZenML's dashboard

We will start with the `get_or_create_user` ZenML step. We begin by importing the necessary modules and functions used throughout the script.

```
from loguru import logger
from typing_extensions import Annotated
from zenml import get_step_context, step

from llm_engineering.application import utils
from llm_engineering.domain.documents import UserDocument
```

Next, we define the function's signature, which takes a user's full name as input and retrieves an existing user or creates a new one in the MongoDB database if it doesn't exist:

```
@step
def get_or_create_user(user_full_name: str) -> Annotated[UserDocument,
"user"]:
```

Using a utility function, we split the full name into first and last names. Then, we attempt to retrieve the user from the database or create a new one if it doesn't exist. We also retrieve the current step context and add metadata about the user to the output, which will be reflected in the metadata of the `user` ZenML output artifact:

```
    logger.info(f"Getting or creating user: {user_full_name}")

    first_name, last_name = utils.split_user_full_name(user_full_name)

    user = UserDocument.get_or_create(first_name=first_name, last_
name=last_name)

    step_context = get_step_context()
    step_context.add_output_metadata(output_name="user", metadata=_get_
metadata(user_full_name, user))

    return user
```

Additionally, we define a helper function called `_get_metadata()`, which builds a dictionary containing the query parameters and the retrieved user information, which will be added as metadata to the user artifact:

```
def _get_metadata(user_full_name: str, user: UserDocument) -> dict:
    return {
        "query": {
            "user_full_name": user_full_name,
        },
        "retrieved": {
            "user_id": str(user.id),
            "first_name": user.first_name,
            "last_name": user.last_name,
        },
    }
```

We will move on to the `crawl_links` ZenML step, which collects the data from the provided links. The code begins by importing essential modules and libraries for web crawling:

```
from urllib.parse import urlparse

from loguru import logger
```

```
from tqdm import tqdm
from typing_extensions import Annotated
from zenml import get_step_context, step

from llm_engineering.application.crawlers.dispatcher import
CrawlerDispatcher
from llm_engineering.domain.documents import UserDocument
```

Following the imports, the main function inputs a list of links written by a specific author. Within this function, a crawler dispatcher is initialized and configured to handle specific domains such as LinkedIn, Medium, and GitHub:

```
@step
def crawl_links(user: UserDocument, links: list[str]) ->
Annotated[list[str], "crawled_links"]:
    dispatcher = CrawlerDispatcher.build().register_linkedin().register_
medium().register_github()

    logger.info(f"Starting to crawl {len(links)} link(s).")
```

The function initializes variables to store the output metadata and count successful crawls. It then iterates over each link. It attempts to crawl and extract data for each link, updating the count of successful crawls and accumulating metadata about each URL:

```
    metadata = {}
    successfull_crawls = 0
    for link in tqdm(links):
        successfull_crawl, crawled_domain = _crawl_link(dispatcher, link,
user)
        successfull_crawls += successfull_crawl

        metadata = _add_to_metadata(metadata, crawled_domain, successfull_
crawl)
```

After processing all links, the function attaches the accumulated metadata to the output artifact:

```
    step_context = get_step_context()
    step_context.add_output_metadata(output_name="crawled_links",
metadata=metadata)

    logger.info(f"Successfully crawled {successfull_crawls} / {len(links)}
```

```
links.")

    return links
```

The code includes a helper function that attempts to extract information from each link using the appropriate crawler based on the link's domain. It handles any exceptions that may occur during extraction and returns a tuple indicating the crawl's success and the link's domain:

```
def _crawl_link(dispatcher: CrawlerDispatcher, link: str, user:
UserDocument) -> tuple[bool, str]:
    crawler = dispatcher.get_crawler(link)
    crawler_domain = urlparse(link).netloc

    try:
        crawler.extract(link=link, user=user)

        return (True, crawler_domain)
    except Exception as e:
        logger.error(f"An error occurred while crawling: {e!s}")

        return (False, crawler_domain)
```

Another helper function is provided to update the metadata dictionary with the results of each crawl:

```
def _add_to_metadata(metadata: dict, domain: str, successfull_crawl: bool)
-> dict:
    if domain not in metadata:
        metadata[domain] = {}
    metadata[domain]["successful"] = metadata.get(domain, {}).
get("successful", 0) + successfull_crawl
    metadata[domain]["total"] = metadata.get(domain, {}).get("total", 0) +
1

    return metadata
```

As seen in the abovementioned `_crawl_link()` function, the `CrawlerDispatcher` class knows what crawler to initialize based on each link's domain. The logic is then abstracted away under the crawler's `extract()` method. Let's zoom in on the `CrawlerDispatcher` class to understand how this works fully.

The dispatcher: How do you instantiate the right crawler?

The entry point to our crawling logic is the `CrawlerDispatcher` class. As illustrated in *Figure 3.4*, the dispatcher acts as the intermediate layer between the provided links and the crawlers. It knows what crawler to associate with each URL.

The `CrawlerDispatcher` class knows how to extract the domain of each link and initialize the proper crawler that collects the data from that site. For example, if it detects the `https://medium.com` domain when providing a link to an article, it will build an instance of the `MediumCrawler` used to crawl that particular platform. With that in mind, let's explore the implementation of the `CrawlerDispatcher` class.

All the crawling logic is available in the GitHub repository at `llm_engineering/application/crawlers`.

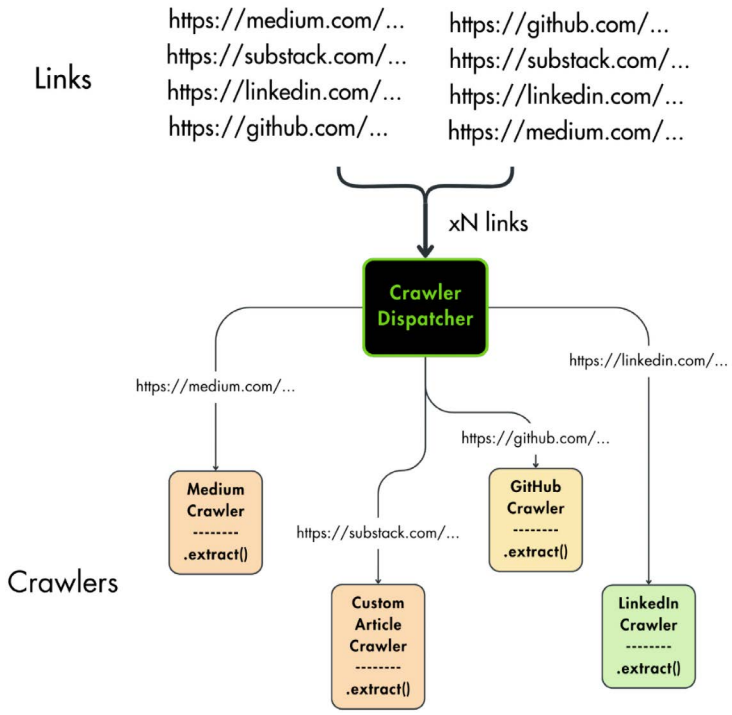

Figure 3.4: The relationship between the provided links, the CrawlerDispatcher, and the crawlers

We begin by importing the necessary Python modules for URL handling and regex, along with importing our crawler classes:

```
import re
from urllib.parse import urlparse

from loguru import logger

from .base import BaseCrawler
from .custom_article import CustomArticleCrawler
from .github import GithubCrawler
from .linkedin import LinkedInCrawler
from .medium import MediumCrawler
```

The `CrawlerDispatcher` class is defined to manage and dispatch appropriate crawler instances based on given URLs and their domains. Its constructor initializes a registry to store the registered crawlers.

```
class CrawlerDispatcher:
    def __init__(self) -> None:
        self._crawlers = {}
```

As we are using the builder creational pattern to instantiate and configure the dispatcher, we define a `build()` class method that returns an instance of the dispatcher:

```
    @classmethod
    def build(cls) -> "CrawlerDispatcher":
        dispatcher = cls()

        return dispatcher
```

The dispatcher includes methods to register crawlers for specific platforms like Medium, LinkedIn, and GitHub. These methods use a generic `register()` method under the hood to add each crawler to the registry. By returning self, we follow the builder creational pattern (more on the builder pattern: `https://refactoring.guru/design-patterns/builder`). We can chain multiple `register_*()` methods when instantiating the dispatcher as follows: `CrawlerDispatcher.build().register_linkedin().register_medium()`.

```
    def register_medium(self) -> "CrawlerDispatcher":
        self.register("https://medium.com", MediumCrawler)
```

```
        return self

    def register_linkedin(self) -> "CrawlerDispatcher":
        self.register("https://linkedin.com", LinkedInCrawler)

        return self

    def register_github(self) -> "CrawlerDispatcher":
        self.register("https://github.com", GithubCrawler)

        return self
```

The generic `register()` method normalizes each domain to ensure its format is consistent before it's added as a key to the `self._crawlers` registry of the dispatcher. This is a critical step, as we will use the key of the dictionary as the domain pattern to match future links with a crawler:

```
    def register(self, domain: str, crawler: type[BaseCrawler]) -> None:
        parsed_domain = urlparse(domain)
        domain = parsed_domain.netloc

        self._crawlers[r"https://(www\.)?{}/*".format(re.escape(domain))]
= crawler
```

Finally, the `get_crawler()` method determines the appropriate crawler for a given URL by matching it against the registered domains. If no match is found, it logs a warning and defaults to using the `CustomArticleCrawler`.

```
    def get_crawler(self, url: str) -> BaseCrawler:
        for pattern, crawler in self._crawlers.items():
            if re.match(pattern, url):
                return crawler()
        else:
            logger.warning(f"No crawler found for {url}. Defaulting to
CustomArticleCrawler.")

            return CustomArticleCrawler()
```

The next step in understanding how the data collection pipeline works is analyzing each crawler individually.

The crawlers

Before exploring each crawler's implementation, we must present their base class, which defines a unified interface for all the crawlers. As shown in *Figure 3.4*, we can implement the dispatcher layer because each crawler follows the same signature. Each class implements the `extract()` method, allowing us to leverage OOP techniques such as polymorphism, where we can work with abstract objects without knowing their concrete subclass. For example, in the `_crawl_link()` function from the ZenML steps, we had the following code:

```
crawler = dispatcher.get_crawler(link)
crawler.extract(link=link, user=user)
```

Note how we called the `extract()` method without caring about what specific type of crawler we instantiated. To conclude, working with abstract interfaces ensures core reusability and ease of extension.

Base classes

Now, let's explore the `BaseCrawler` interface, which can be found in the repository at `https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/crawlers/base.py`.

```
from abc import ABC, abstractmethod

class BaseCrawler(ABC):
    model: type[NoSQLBaseDocument]

    @abstractmethod
    def extract(self, link: str, **kwargs) -> None: ...
```

As mentioned above, the interface defines an `extract()` method that takes as input a link. Also, it defines a model attribute at the class level that represents the data category document type used to save the extracted data into the MongoDB data warehouse. Doing so allows us to customize each subclass with different data categories while preserving the same attributes at the class level. We will soon explore the `NoSQLBaseDocument` class when digging into the document entities.

We also extend the `BaseCrawler` class with a `BaseSeleniumCrawler` class, which implements reusable functionality that uses Selenium to crawl various sites, such as Medium or LinkedIn. **Selenium** is a tool for automating web browsers. It's used to interact with web pages programmatically (like logging into LinkedIn, navigating through profiles, etc.).

Selenium can programmatically control various browsers such as Chrome, Firefox, or Brave. For these specific platforms, we need Selenium to manipulate the browser programmatically to log in and scroll through the newsfeed or article before being able to extract the entire HTML. For other sites, where we don't have to go through the login step or can directly load the whole page, we can extract the HTML from a particular URL using more straightforward methods than Selenium.

For the Selenium-based crawlers to work, you must install Chrome on your machine (or a Chromium-based browser such as Brave).

The code begins by setting up the necessary imports and configurations for web crawling using Selenium and the ChromeDriver initializer. The `chromedriver_autoinstaller` ensures that the appropriate version of ChromeDriver is installed and added to the system path, maintaining compatibility with the installed version of your Google Chrome browser (or other Chromium-based browser). Selenium will use the ChromeDriver to communicate with the browser and open a headless session, where we can programmatically manipulate the browser to access various URLs, click on specific elements, such as buttons, or scroll through the newsfeed. Using the `chromedriver_autoinstaller`, we ensure we always have the correct ChromeDriver version installed that matches our machine's Chrome browser version.

```
import time
from tempfile import mkdtemp

import chromedriver_autoinstaller
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

from llm_engineering.domain.documents import NoSQLBaseDocument

# Check if the current version of chromedriver exists
# and if it doesn't exist, download it automatically,
# then add chromedriver to path
chromedriver_autoinstaller.install()
```

Next, we define the `BaseSeleniumCrawler` class for use cases where we need Selenium to collect the data, such as collecting data from Medium or LinkedIn.

Its constructor initializes various Chrome options to optimize performance, enhance security, and ensure a headless browsing environment. These options disable unnecessary features like GPU rendering, extensions, and notifications, which can interfere with automated browsing. These are standard configurations when crawling in headless mode:

```
class BaseSeleniumCrawler(BaseCrawler, ABC):
    def __init__(self, scroll_limit: int = 5) -> None:
        options = webdriver.ChromeOptions()

        options.add_argument("--no-sandbox")
        options.add_argument("--headless=new")
        options.add_argument("--disable-dev-shm-usage")
        options.add_argument("--log-level=3")
        options.add_argument("--disable-popup-blocking")
        options.add_argument("--disable-notifications")
        options.add_argument("--disable-extensions")
        options.add_argument("--disable-background-networking")
        options.add_argument("--ignore-certificate-errors")
        options.add_argument(f"--user-data-dir={mkdtemp()}")
        options.add_argument(f"--data-path={mkdtemp()}")
        options.add_argument(f"--disk-cache-dir={mkdtemp()}")
        options.add_argument("--remote-debugging-port=9226")
```

After configuring the Chrome options, the code allows subclasses to set any additional driver options by calling the `set_extra_driver_options()` method. It then initializes the scroll limit and creates a new instance of the Chrome driver with the specified options:

```
        self.set_extra_driver_options(options)

        self.scroll_limit = scroll_limit
        self.driver = webdriver.Chrome(
            options=options,
        )
```

The `BaseSeleniumCrawler` class includes placeholder methods for `set_extra_driver_options()` and `login()`, which subclasses can override to provide specific functionality. This ensures modularity, as every platform has a different login page with a different HTML structure:

```
    def set_extra_driver_options(self, options: Options) -> None:
```

```
        pass

    def login(self) -> None:
        pass
```

Finally, the `scroll_page()` method implements a scrolling mechanism to navigate through pages, such as LinkedIn, up to a specified scroll limit. It scrolls to the bottom of the page, waits for new content to load, and repeats the process until it reaches the end of the page or the scroll limit is exceeded. This method is essential for feeds where the content appears as the user scrolls:

```
    def scroll_page(self) -> None:
        """Scroll through the LinkedIn page based on the scroll limit."""
        current_scroll = 0
        last_height = self.driver.execute_script("return document.body.
scrollHeight")
        while True:
            self.driver.execute_script("window.scrollTo(0, document.body.
scrollHeight);")
            time.sleep(5)
            new_height = self.driver.execute_script("return document.body.
scrollHeight")
            if new_height == last_height or (self.scroll_limit and
current_scroll >= self.scroll_limit):
                break
            last_height = new_height
            current_scroll += 1
```

We've understood what the base classes of our crawlers look like. Next, we will look into the implementation of the following specific crawlers:

- `GitHubCrawler(BaseCrawler)`
- `CustomArticleCrawler(BaseCrawler)`
- `MediumCrawler(BaseSeleniumCrawler)`

You can find the implementation of the above crawlers in the GitHub repository at `https://github.com/PacktPublishing/LLM-Engineers-Handbook/tree/main/llm_engineering/application/crawlers`.

GitHubCrawler class

The `GithubCrawler` class is designed to scrape GitHub repositories, extending the functionality of the `BaseCrawler`. We don't have to log in to GitHub through the browser, as we can leverage Git's clone functionality. Thus, we don't have to leverage any Selenium functionality. Upon initialization, it sets up a list of patterns to ignore standard files and directories found in GitHub repositories, such as `.git`, `.toml`, `.lock`, and `.png`, ensuring that unnecessary files are excluded from the scraping process:

```
class GithubCrawler(BaseCrawler):
    model = RepositoryDocument

    def __init__(self, ignore=(".git", ".toml", ".lock", ".png")) -> None:
        super().__init__()
        self._ignore = ignore
```

Next, we implement the `extract()` method, where the crawler first checks if the repository has already been processed and stored in the database. If it exists, it exits the method to prevent storing duplicates:

```
def extract(self, link: str, **kwargs) -> None:
    old_model = self.model.find(link=link)
    if old_model is not None:
        logger.info(f"Repository already exists in the database: {link}")

        return
```

If the repository is new, the crawler extracts the repository name from the link. Then, it creates a temporary directory to clone the repository to ensure that the cloned repository is cleaned up from the local disk after it's processed:

```
    logger.info(f"Starting scrapping GitHub repository: {link}")

    repo_name = link.rstrip("/").split("/")[-1]

    local_temp = tempfile.mkdtemp()
```

Within a try block, the crawler changes the current working directory to the `temporary` directory and executes the `git clone` command in a different process:

```
    try:
```

```
        os.chdir(local_temp)
        subprocess.run(["git", "clone", link])
```

After successfully cloning the repository, the crawler constructs the path to the cloned repository. It initializes an empty dictionary used to aggregate the content of the files in a standardized way. It walks through the directory tree, skipping over any directories or files that match the ignore patterns. For each relevant file, it reads the content, removes any spaces, and stores it in the dictionary with the file path as the key:

```
        repo_path = os.path.join(local_temp, os.listdir(local_temp)[0])  #
        tree = {}
        for root, _, files in os.walk(repo_path):
            dir = root.replace(repo_path, "").lstrip("/")
            if dir.startswith(self._ignore):
                continue

            for file in files:
                if file.endswith(self._ignore):
                    continue
                file_path = os.path.join(dir, file)
                with open(os.path.join(root, file), "r", errors="ignore")
as f:
                    tree[file_path] = f.read().replace(" ", "")
```

It then creates a new instance of the `RepositoryDocument` model, populating it with the repository content, name, link, platform information, and author details. The instance is then saved to MongoDB:

```
        user = kwargs["user"]
        instance = self.model(
            content=tree,
            name=repo_name,
            link=link,
            platform="github",
            author_id=user.id,
            author_full_name=user.full_name,
        )
        instance.save()
```

Finally, whether the scraping succeeds or an exception occurs, the crawler ensures that the temporary directory is removed to clean up any resources used during the process:

```
    except Exception:
        raise
    finally:
        shutil.rmtree(local_temp)

    logger.info(f"Finished scrapping GitHub repository: {link}")
```

CustomArticleCrawler class

The `CustomArticleCrawler` class takes a different approach to collecting data from the internet. It leverages the `AsyncHtmlLoader` class to read the entire HTML from a link and the `Html2TextTransformer` class to extract the text from that HTML. Both classes are made available by the `langchain_community` Python package, as seen below, where we import all the necessary Python modules:

```
from urllib.parse import urlparse

from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_community.document_transformers.html2text import
Html2TextTransformer
from loguru import logger

from llm_engineering.domain.documents import ArticleDocument

from .base import BaseCrawler
```

Next, we define the `CustomArticleCrawler` class, which inherits from `BaseCrawler`. As before, we don't need to log in or use the scrolling functionality provided by Selenium. In the `extract` method, we first check if the article exists in the database to avoid duplicating content:

```
class CustomArticleCrawler(BaseCrawler):
    model = ArticleDocument

    def extract(self, link: str, **kwargs) -> None:
        old_model = self.model.find(link=link)
        if old_model is not None:
```

```
            logger.info(f"Article already exists in the database: {link}")

            return
```

If the article doesn't exist, we proceed to scrape it. We use the `AsyncHtmlLoader` class to load the HTML from the provided link. After, we transform it into plain text using the `Html2TextTransformer` class, which returns a list of documents. We are only interested in the first document. As we delegate the whole logic to these two classes, we don't control how the content is extracted and parsed. That's why we used this class as a fallback system for domains where we don't have anything custom implemented. These two classes follow the LangChain paradigm, which provides high-level functionality that works decently in most scenarios. It is fast to implement but hard to customize. That is one of the reasons why many developers avoid using LangChain in production use cases:

```
        logger.info(f"Starting scrapping article: {link}")

        loader = AsyncHtmlLoader([link])
        docs = loader.load()

        html2text = Html2TextTransformer()
        docs_transformed = html2text.transform_documents(docs)
        doc_transformed = docs_transformed[0]
```

We get the page content from the extracted document, plus relevant metadata such as the `title`, `subtitle`, `content`, and `language`:

```
        content = {
            "Title": doc_transformed.metadata.get("title"),
            "Subtitle": doc_transformed.metadata.get("description"),
            "Content": doc_transformed.page_content,
            "language": doc_transformed.metadata.get("language"),
        }
```

Next, we parse the URL to determine the platform (or domain) from which the article was scraped:

```
        parsed_url = urlparse(link)
        platform = parsed_url.netloc
```

We then create a new instance of the article model, populating it with the extracted content. Finally, we save this instance to the MongoDB data warehouse:

```
        user = kwargs["user"]
```

Weight decay

Weight decay works by adding a penalty for large weights to the loss function, encouraging the model to learn simpler, more generalizable features. This helps the model avoid relying too heavily on any single input feature, which can improve its performance on unseen data. Typically, weight decay values range from 0.01 to 0.1, with 0.01 being a common starting point. For example, if you're using the AdamW optimizer, you might set the weight decay to 0.01.

While weight decay can be beneficial, setting it too high can impede learning by making it difficult for the model to capture important patterns in the data. Conversely, setting it too low may not provide sufficient regularization. The optimal weight decay value often depends on the specific model architecture and dataset, so it's generally a good practice to experiment with different values.

Gradient checkpointing

Gradient checkpointing is a technique that reduces memory consumption during training by storing only a subset of intermediate activations generated in the forward pass. In standard training procedures, all intermediate activations are retained in memory to facilitate gradient calculation during the backward pass. However, for very deep networks like LLMs, this approach can quickly become impractical due to hardware limitations, especially on GPUs with limited memory capacity.

Gradient checkpointing addresses this challenge by selectively saving activations at specific layers within the network. For layers where activations are not saved, they are recomputed during the backward pass as needed for gradient computation. This approach creates a trade-off between computation time and memory usage. While it significantly reduces memory requirements, it may increase overall computation time due to the need to recalculate some activations.

Other parameters and techniques exist but play a minor role compared to those previously discussed. In the next section, we will explore how to select and tune these parameters using a concrete example.

Fine-tuning in practice

Let's now fine-tune an open-source model on our custom dataset. In this section, we will show an example that implements LoRA and QLoRA for efficiency. Depending on the hardware you have available, you can select the technique that best corresponds to your configuration.

There are many efficient open-weight models we can leverage for task or domain-specific use cases. To select the most relevant LLM, we need to consider three main parameters: