```
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*:

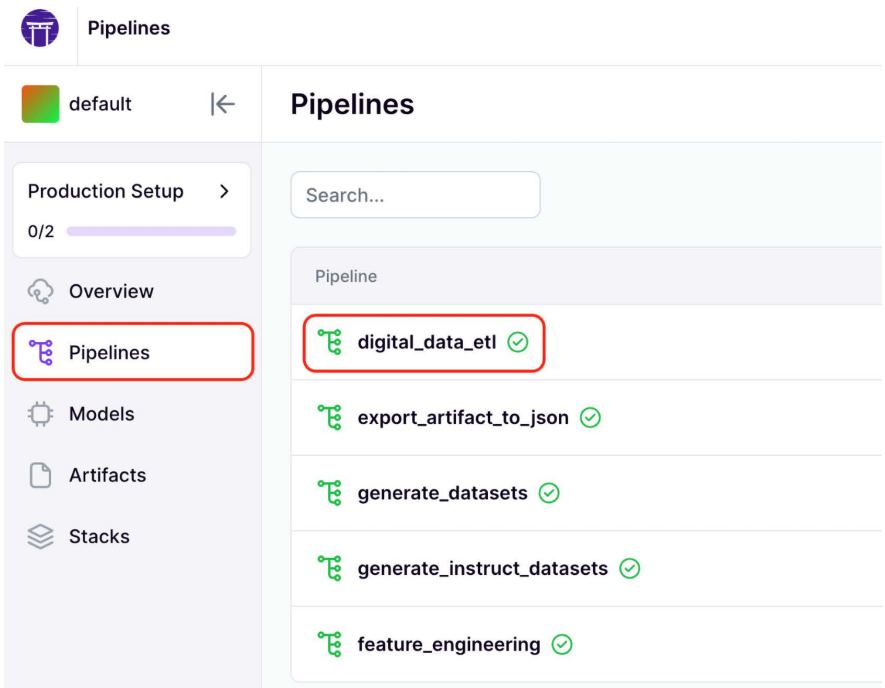

Figure 2.2: ZenML Pipelines dashboard

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*.

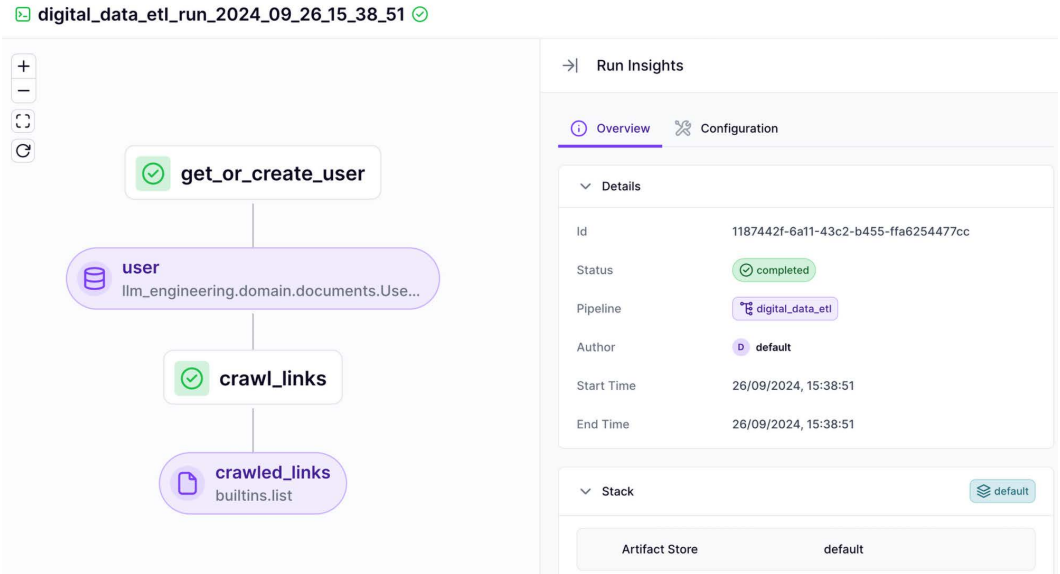

Figure 2.4: ZenML digital_data_etl pipeline run dashboard (example of a specific pipeline run)

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*.

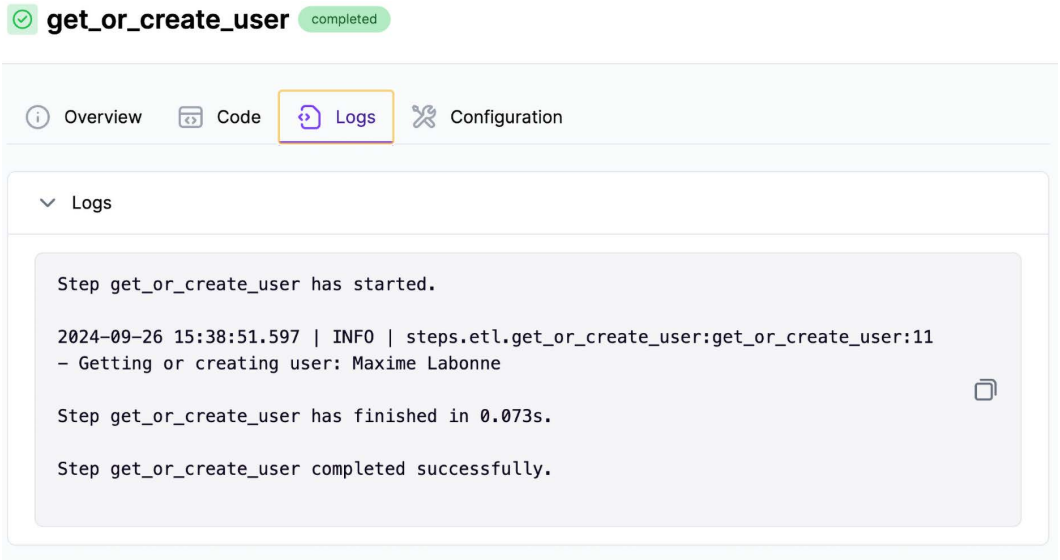

Figure 2.5: Example of insights from a specific step of the digital_data_etl pipeline run

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*:

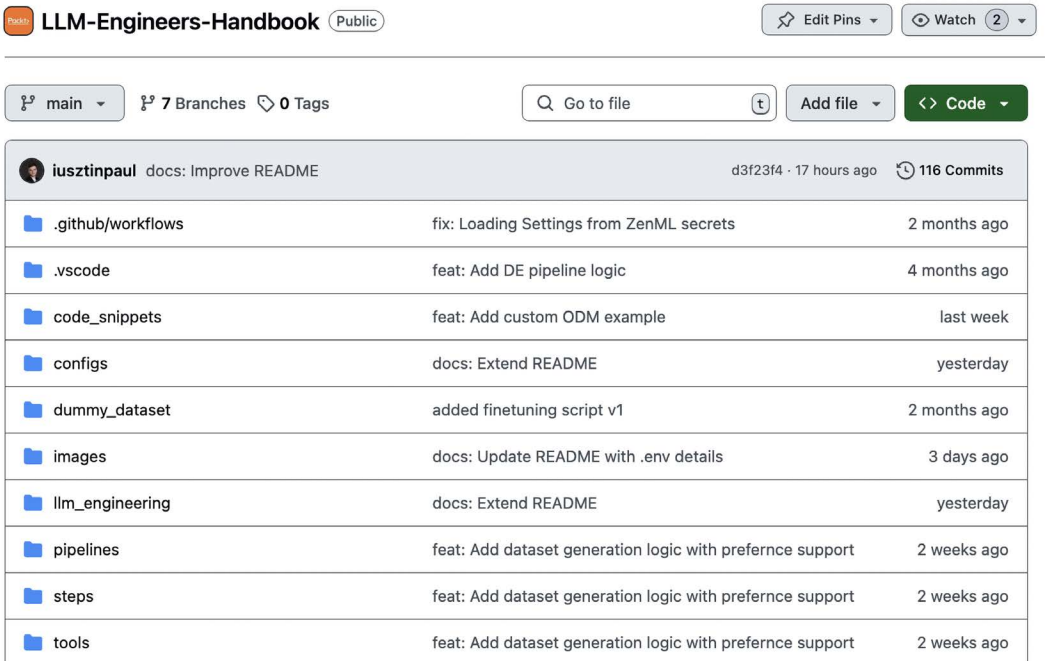

Figure 2.6: LLM-Engineers-Handbook repository folder structure

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*:

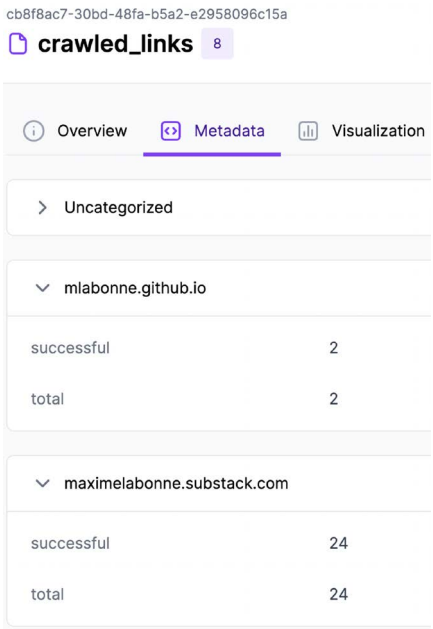

Figure 2.8: ZenML metadata example using the digital_data_etl pipeline as an example

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.

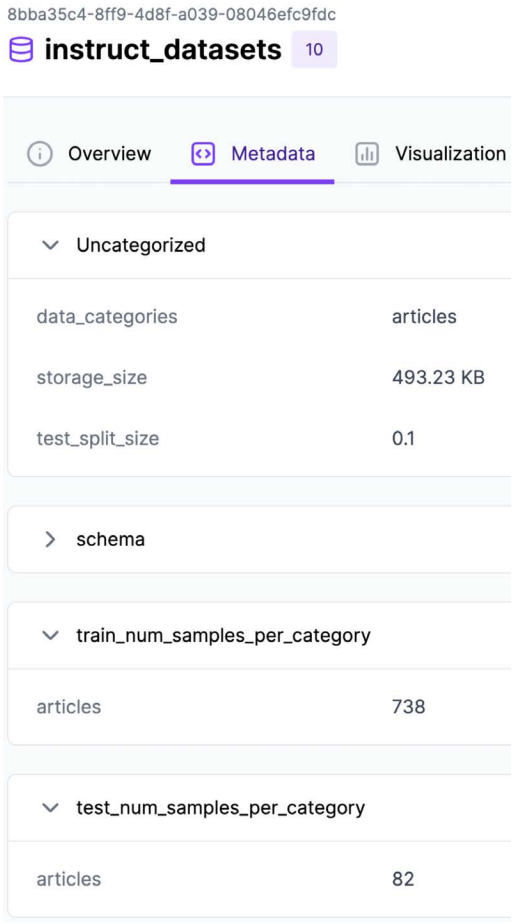

Figure 2.9: ZenML metadata example for the instruct_datasets artifact

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.

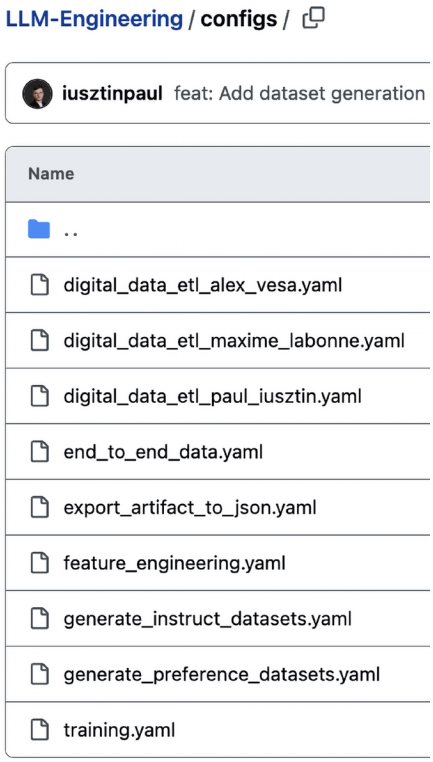

Figure 2.10: ZenML pipeline configs

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.

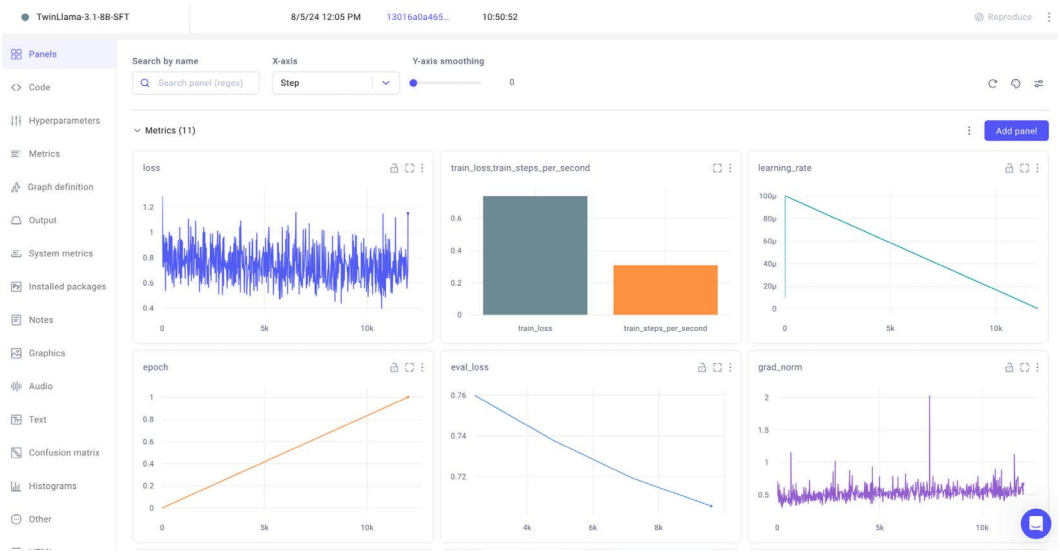

Figure 2.11: Comet ML training metrics example

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*.

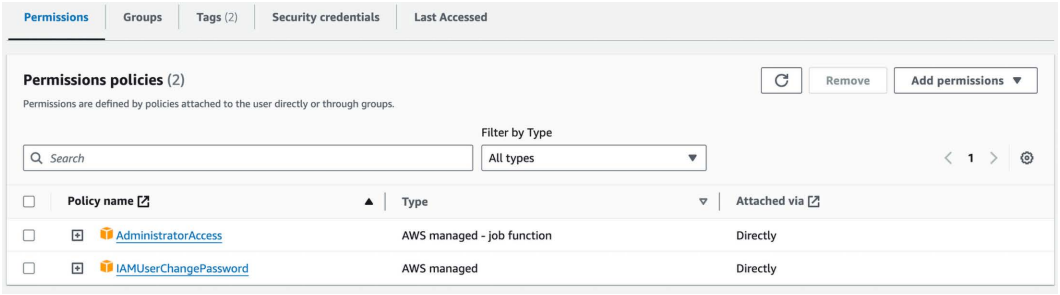

Figure 2.13: IAM user permission policies example

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.

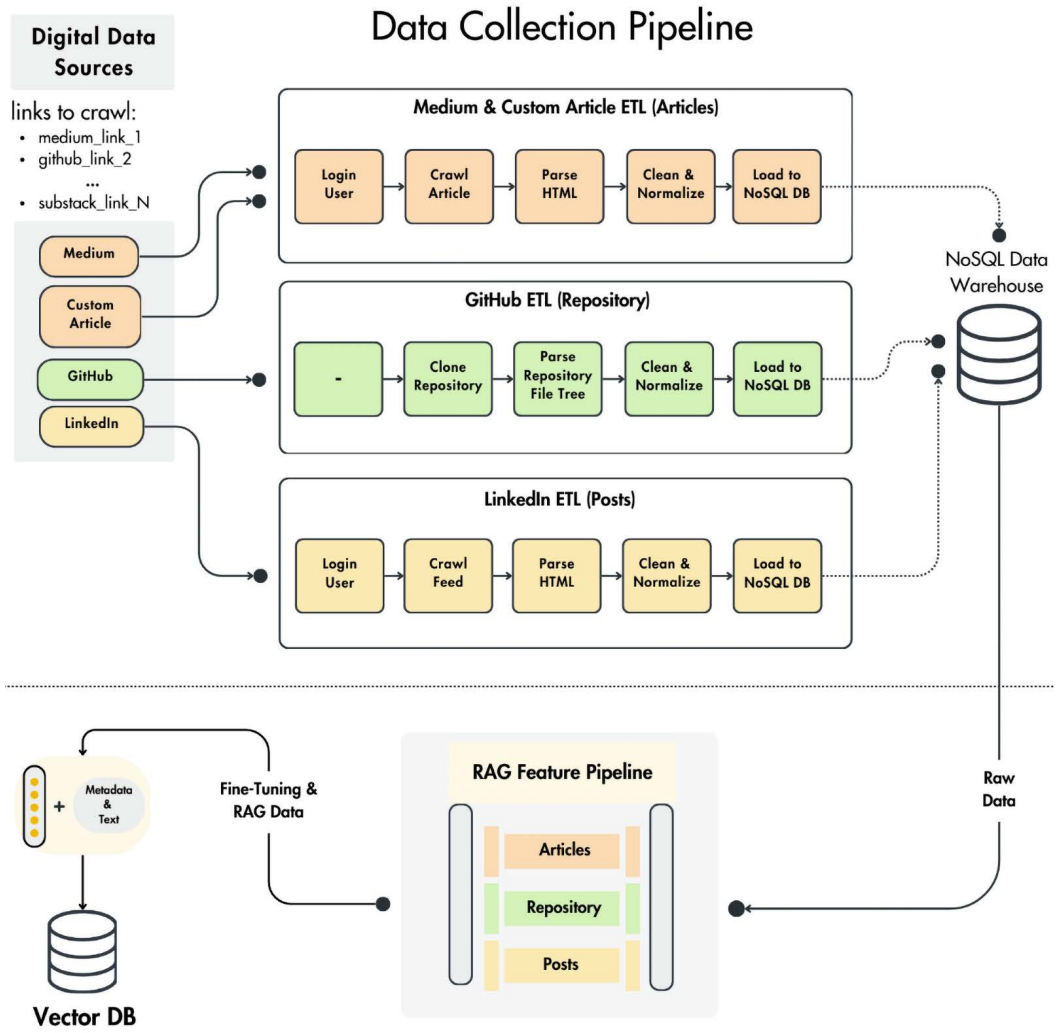

Figure 3.1: LLM Twin's data collection ETL pipeline architecture

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.

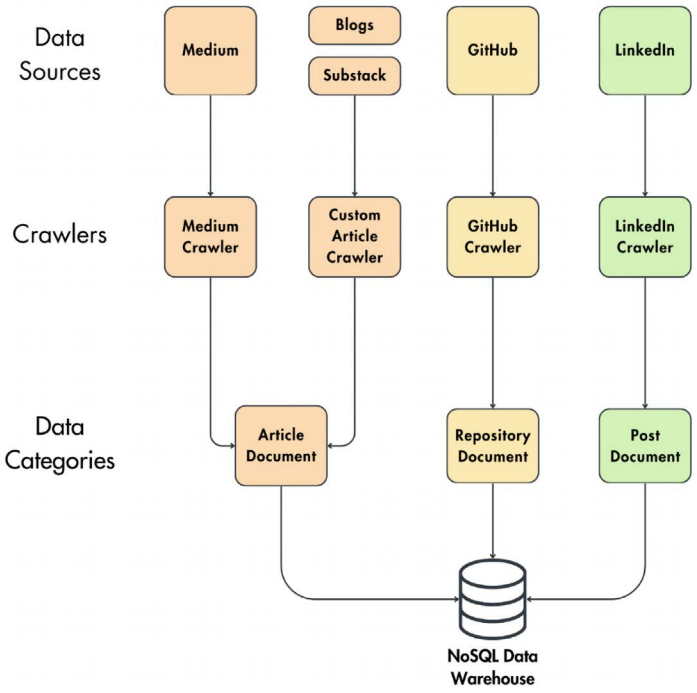

Figure 3.2: The relationship between the crawlers and the data categories

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`.

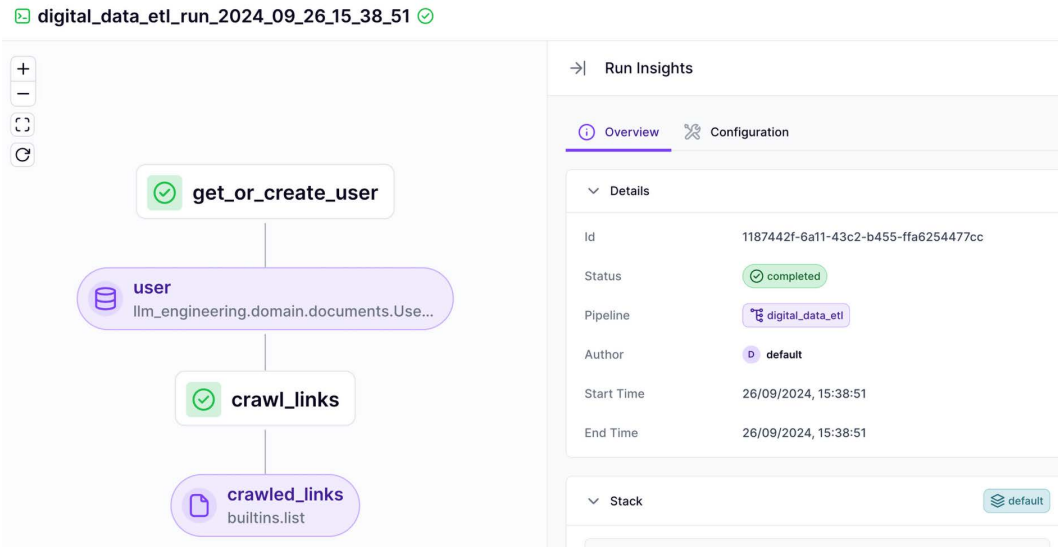

Figure 3.3: Example of a digital_data_etl pipeline run from ZenML's dashboard

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`.

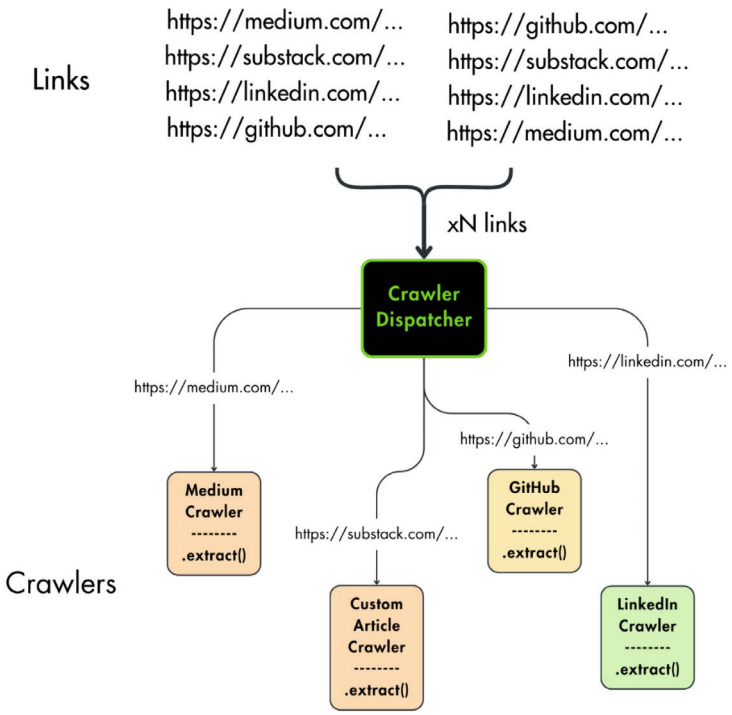

Figure 3.4: The relationship between the provided links, the CrawlerDispatcher, and the crawlers

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:
```