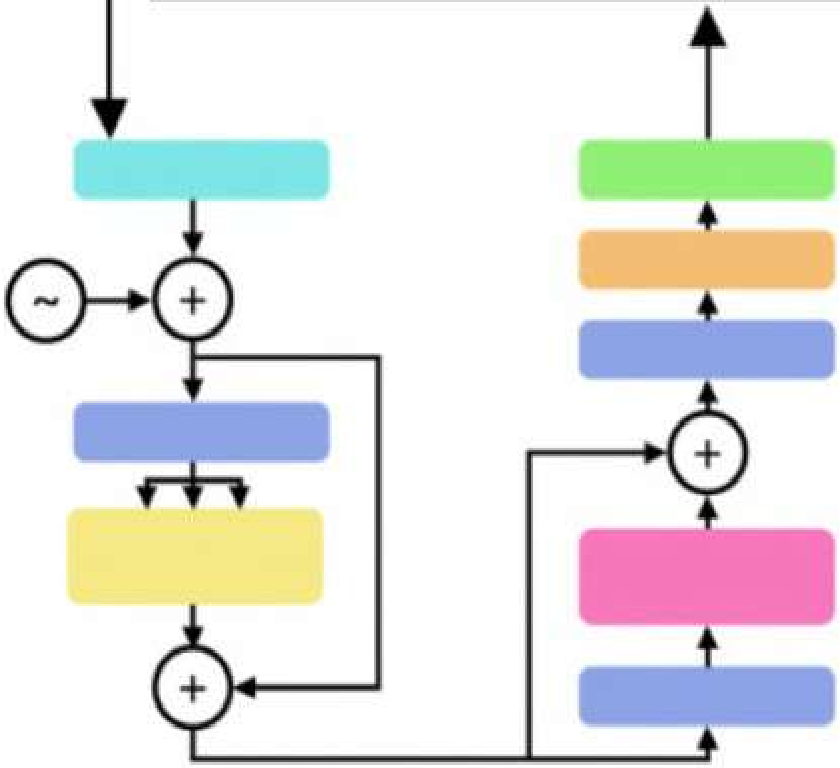
EXPERT INSIGHT

Early Access

4178 44 29483 596

LLM Engineer's Handbook

49924



Master the art of engineering Large Language Models from concept to production

Paul Iusztin | Maxime Labonne
Alex Vesa

packt

LLM Engineer’s Handbook

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Early Access Publication: LLM Engineer’s Handbook

Early Access Production Reference: B31105

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-83620-007-9

www.packt.com

Table of Contents

LLM Engineer’s Handbook: Master the art of engineering Large Language Models from concept to production

Welcome to Packt Early Access. We’re giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1 Understanding the LLM Twin Concept and its 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 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 lifecycle, 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 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 a 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, 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 4* on fine-tuning and *Chapters 8* and *9* on RAG, but for now, let's look at a few examples to intuitively understand what we stated above.

Here are some scenarios on 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 like yourself.

All the scenarios above 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 my 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 if 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:

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

What is 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 personal 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:

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:

- what data do we collect
- how we preprocess the data
- how do we feed the data into the LLM
- how do we chain multiple prompts for the desired results
- how do 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 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

A minimum viable product (or 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.

Define 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 3 people with 2 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 following features 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 DB using our digital data for RAG
- create LinkedIn posts leveraging:
- 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:
- configure your social media links and trigger the collection step
- a chat that allows you to 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 SWE & 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:

- ingest, clean and validate fresh data
- training vs. 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.

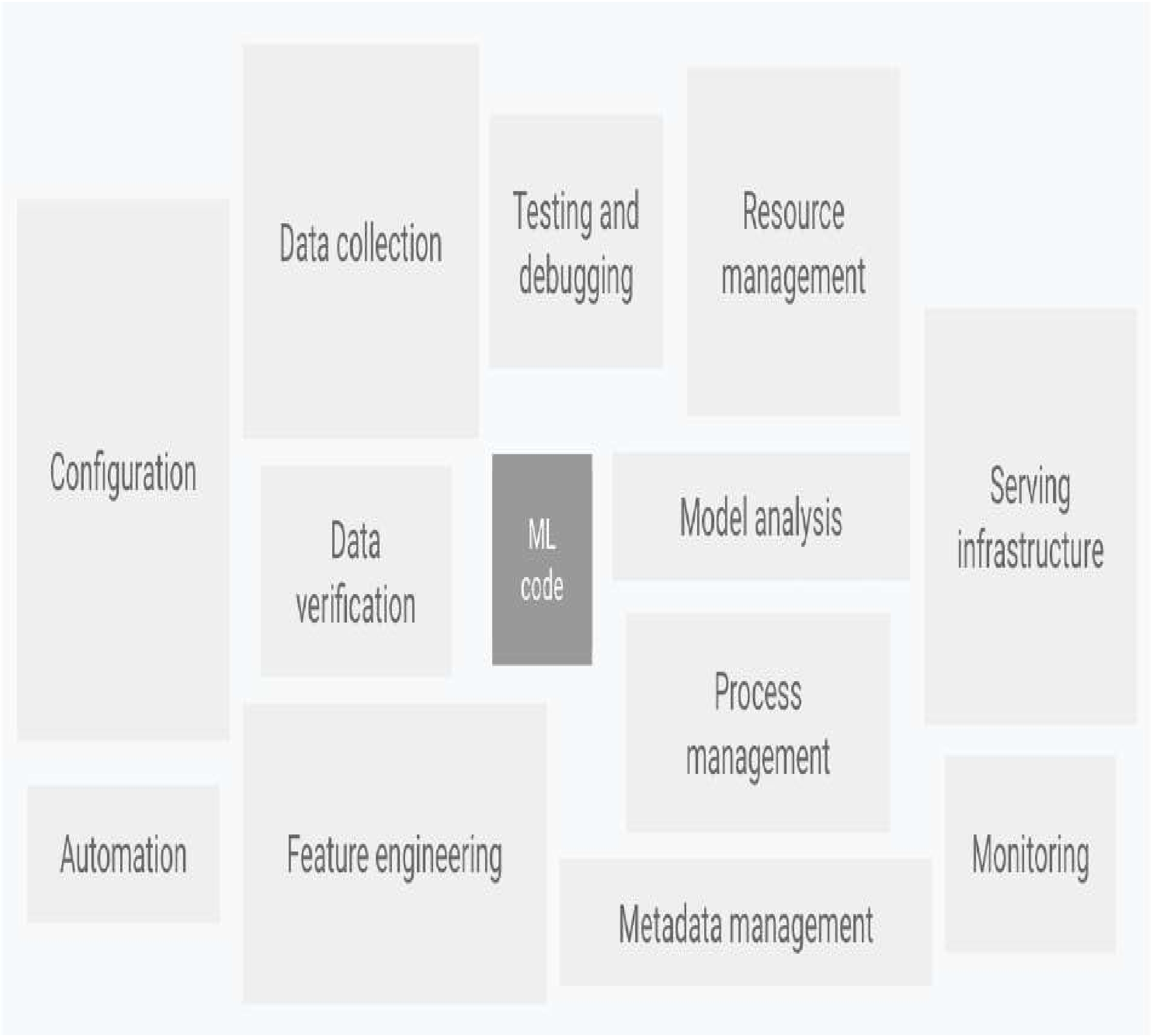


Figure 1.1: Elements for ML systems

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: "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 database, 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 three components stated above.

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.

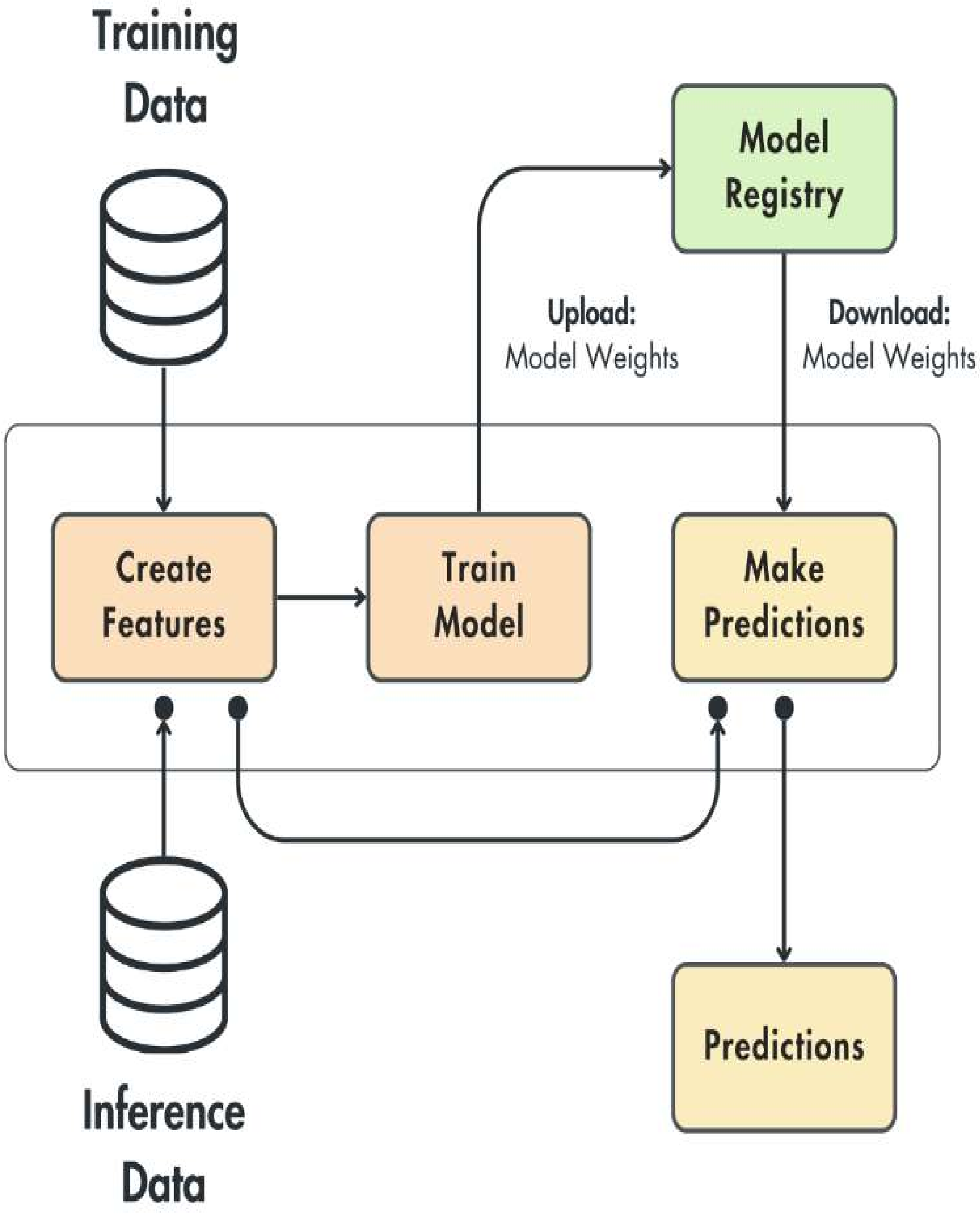


Figure 1.2: Monolithic batch pipeline architecture

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

- 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
- hard to rewrite the prediction module in a more efficient language such as C++, Java or Rust
- hard to share the work between multiple teams between the features, training, and prediction modules
- impossible to switch to a 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 much detail in *Chapters 8* and *9*.

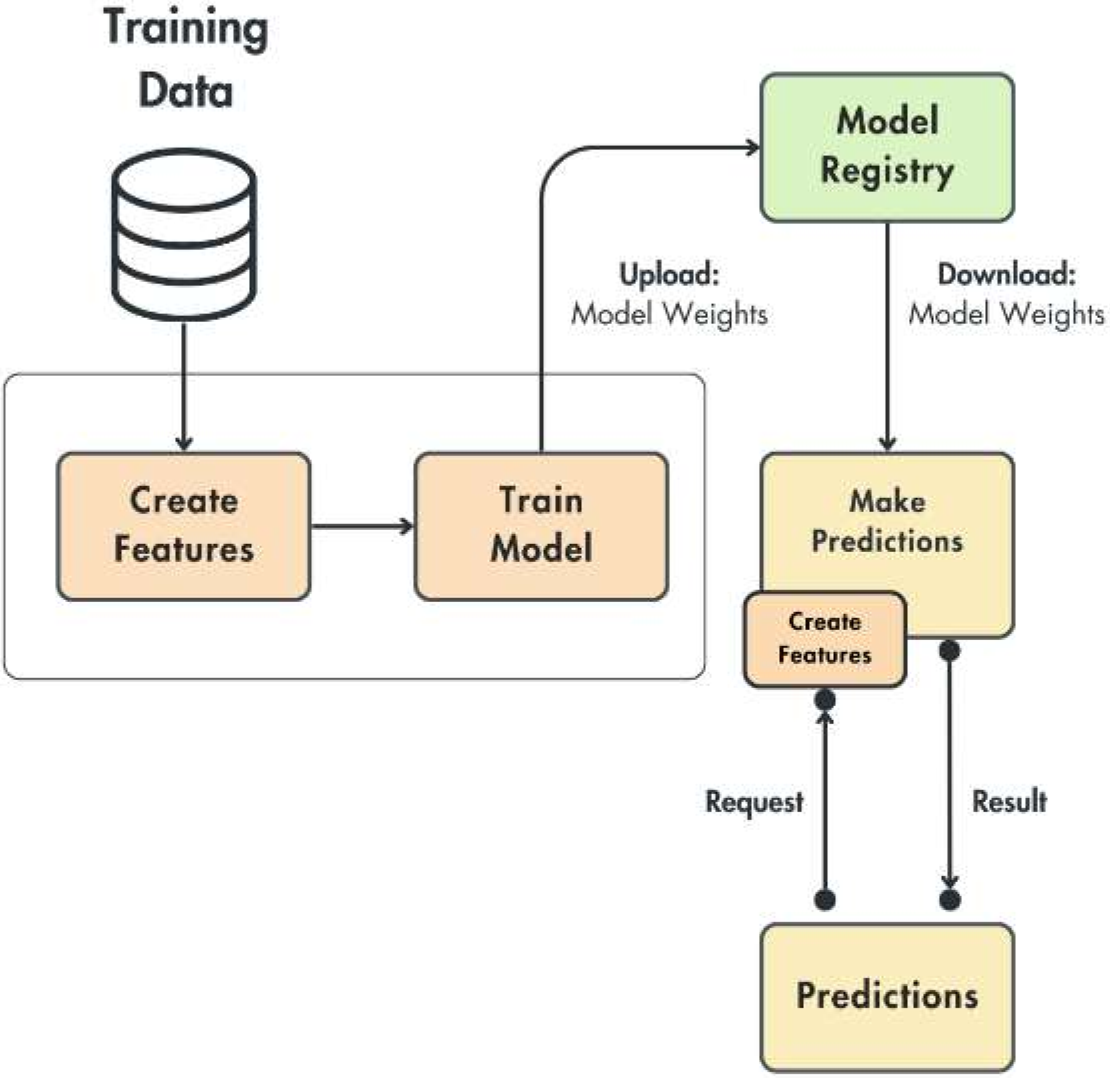
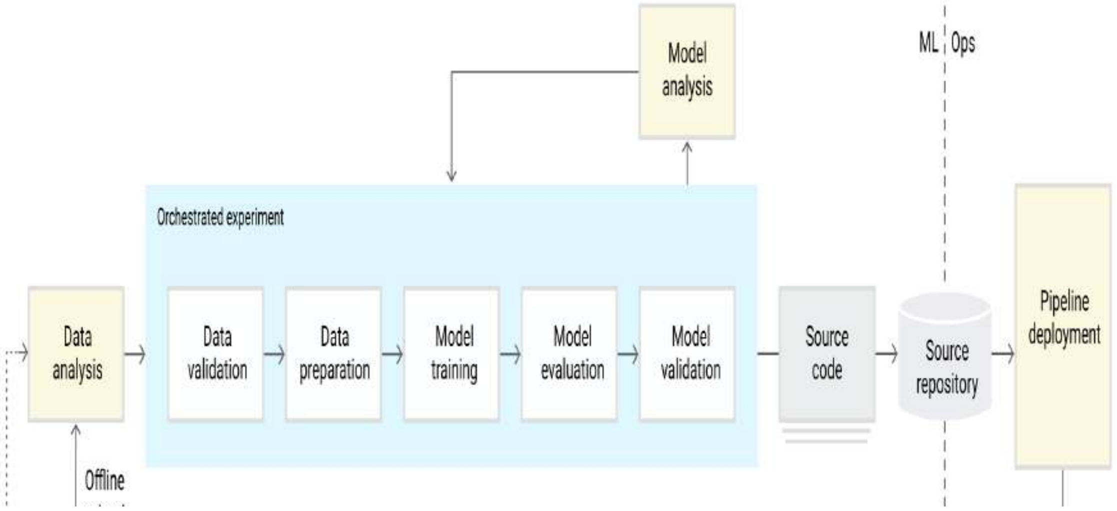


Figure 1.3: Stateless real-time architecture

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 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 (Source: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning):



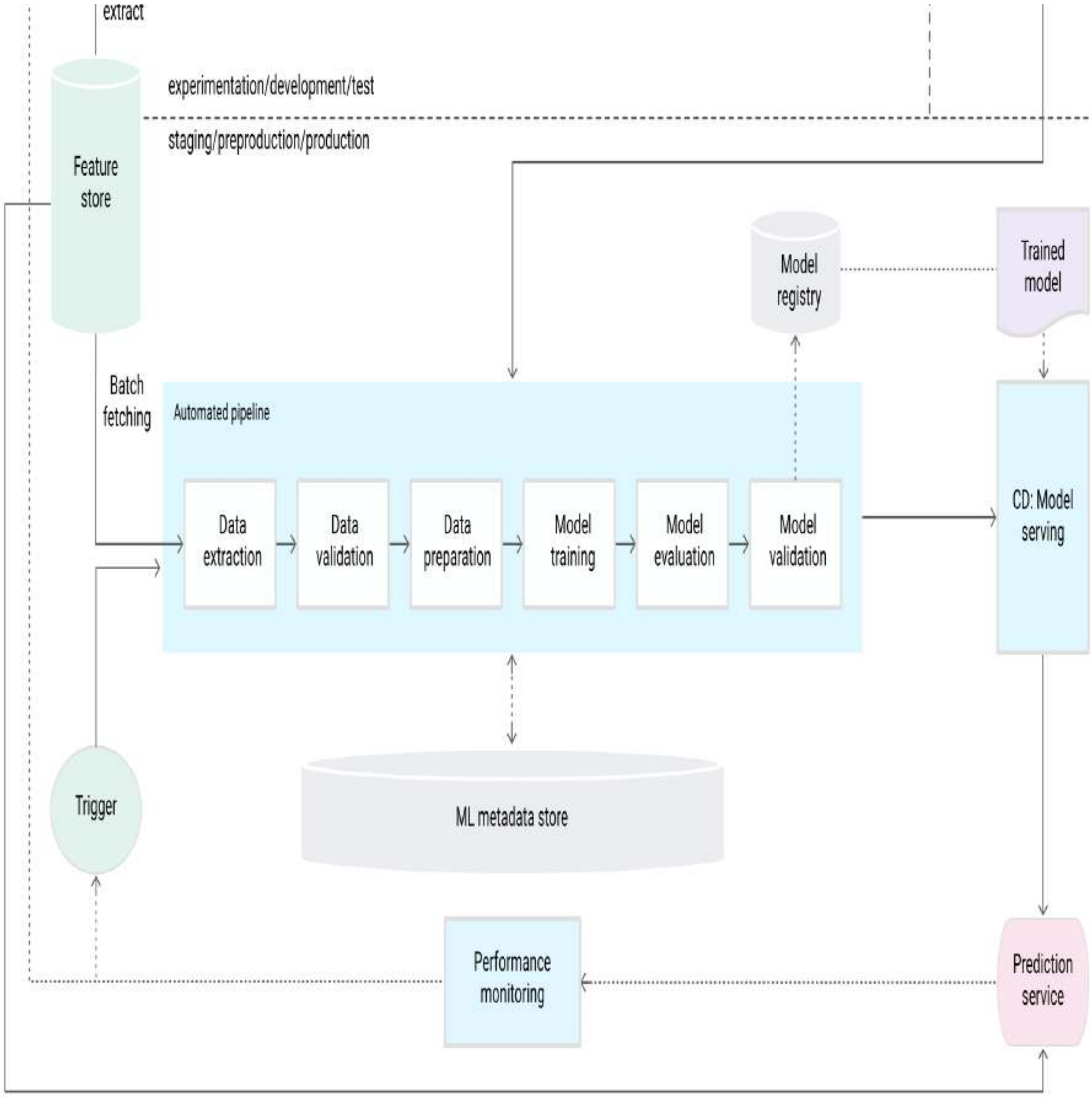


Figure 1.4: ML pipeline automation for CT

But here is where the feature/training/inference 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 (feature, training, inference) pipelines. 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 database, 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.

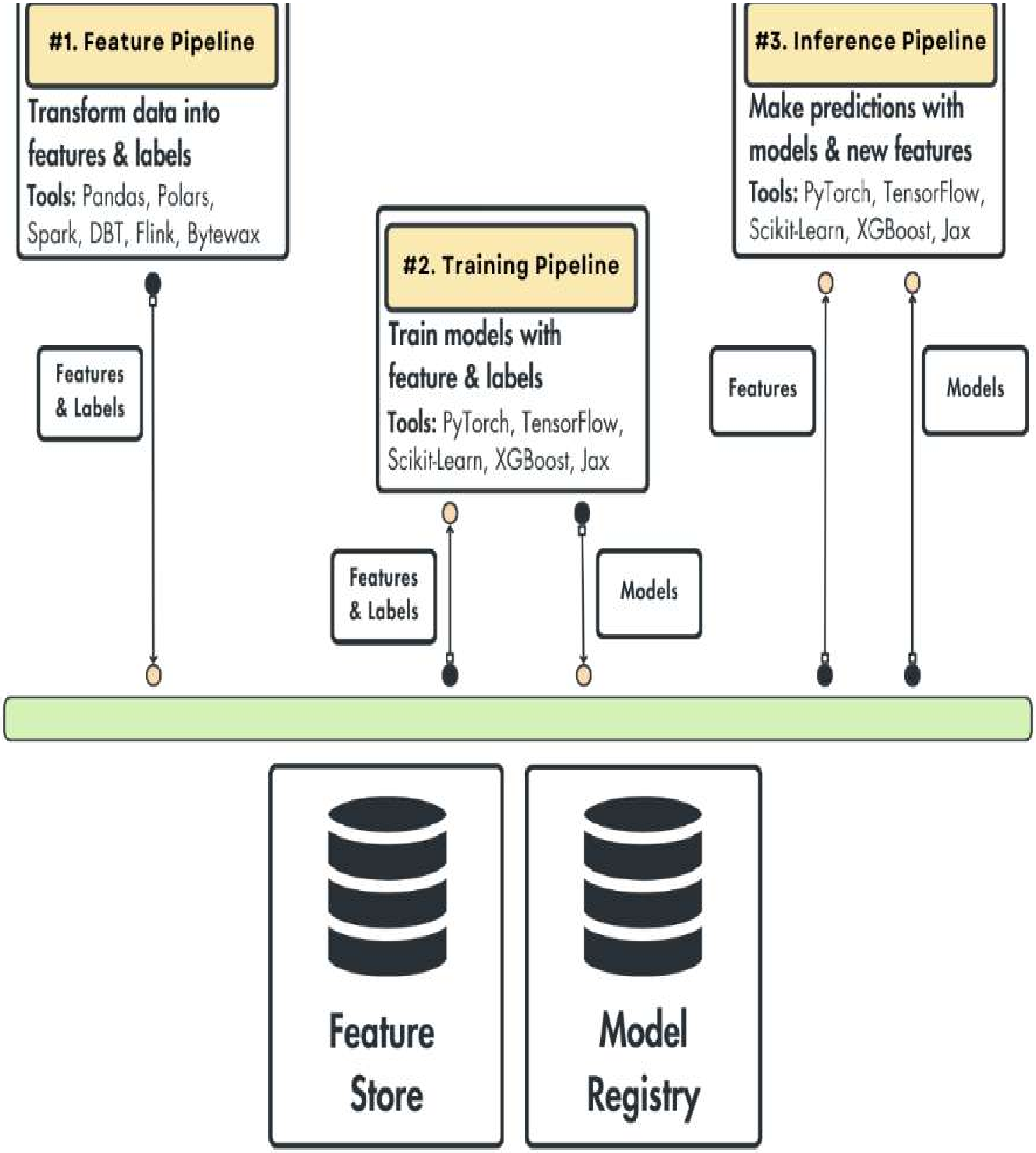

#1. Feature Pipeline
Transform data into features & labels
Tools: Pandas, Polars, Spark, DBT, Flink, Bytewax
#2. Training Pipeline
Train models with feature & labels
Tools: PyTorch, TensorFlow, Scikit-Learn, XGBoost, Jax
#3. Inference Pipeline
Make predictions with models & new features
Tools: PyTorch, TensorFlow, Scikit-Learn, XGBoost, Jax
Features & Labels
Features & Labels
Models
Features
Models
Feature Store
Model Registry

Figure 1.5: Feature/Training/Inference (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 into a feature store, we always have a state of our features. Thus, we can easily send the features to the training and inference pipeline(s).

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 store 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 & 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 database. If it's a real-time system, the predictions will be served to the client who requested them. Additionally, as the features, labels, and model 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 features & labels saved to the feature store.
- The training pipelines query the features store for features & labels and output 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 features/training/inference pipeline pattern, consider reading this article by Jim Dowling, CEO and Co-Founder at Hopsworks: From MLOps to ML Systems with Feature/Training/Inference 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:

- 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:

- 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 LLM of various sizes;
- autoscaling based on user requests;
- automatically deploy the LLMs that pass the evaluation step.

The system will support the next LLMOps features:

- instruction dataset versioning, lineage and reusability
- model versioning, lineage and reusability
- experiment tracking
- continuous training, integration and 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 list above 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: "How can we apply the FTI pipeline design to implement the list of requirements from above?"

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

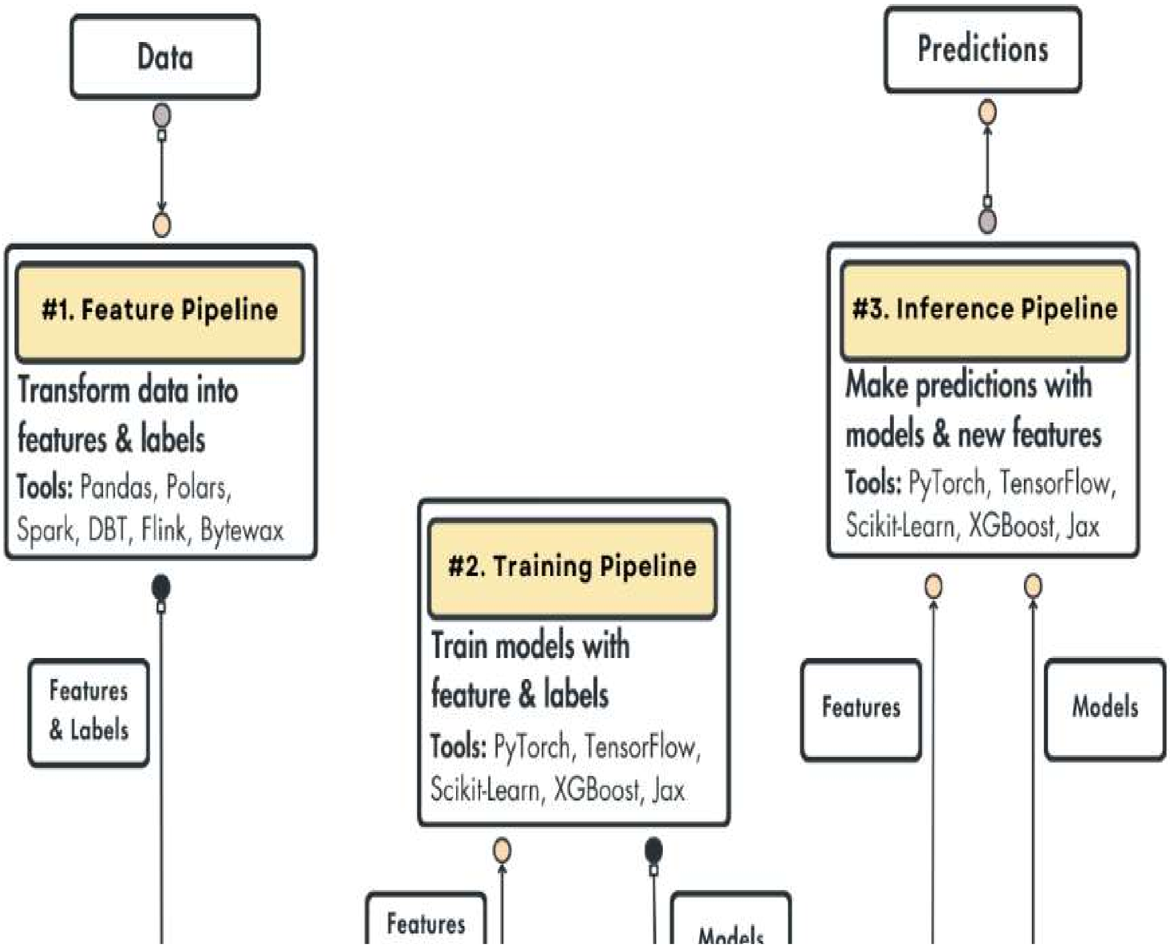
We will split the system into four core components. You will ask yourself: "Four? Why not three, as the FTI pipeline design clearly states." That is a great question. Fortunately, the answer is simple. We must also implement the data pipeline along the three feature/training/inference pipelines. According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


Data
Predictions
#1. Feature Pipeline
Transform data into features & labels
Tools: Pandas, Polars, Spark, DBT, Flink, Bytewax
#3. Inference Pipeline
Make predictions with models & new features
Tools: PyTorch, TensorFlow, Scikit-Learn, XGBoost, Jax
#2. Training Pipeline
Train models with feature & labels
Tools: PyTorch, TensorFlow, Scikit-Learn, XGBoost, Jax
Features & Labels
Features
Models
Features
Models

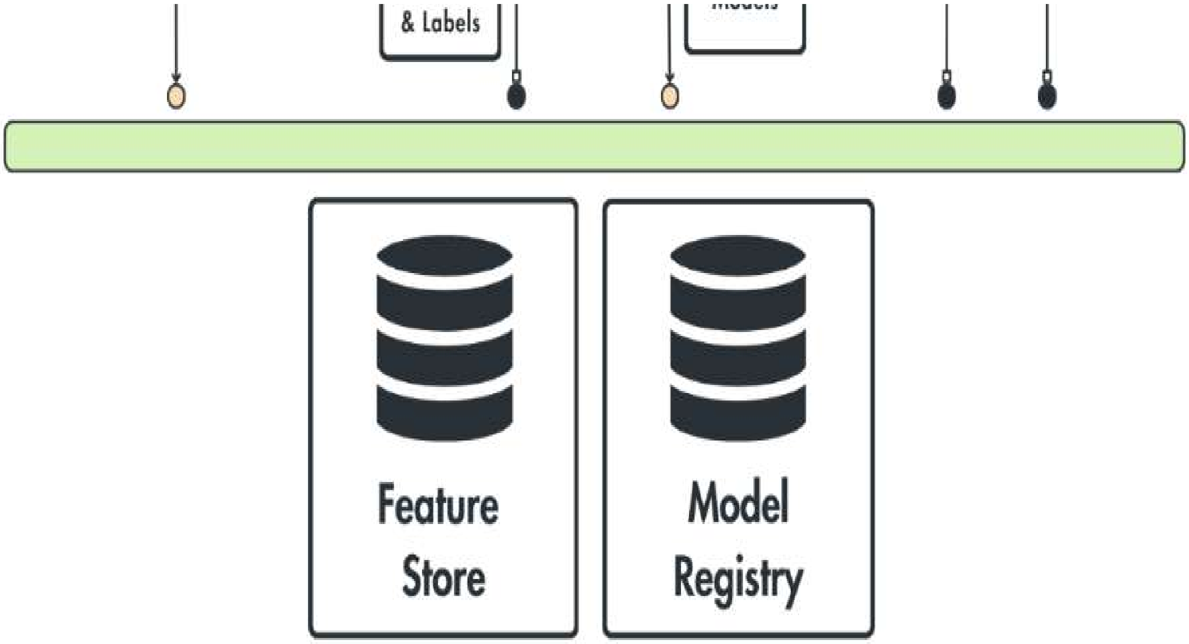


Figure 1.6: LLM twin high-level architecture

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 database, which will act as our data warehouse. As we work with text data, which is naturally unstructured, a NoSQL database fits like a glove.

Even though a NoSQL database, 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.

What 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 database. Instead of integrating another database, more concretely a specialized feature store, we used the vector database, plus some additional logic to check all the properties of a feature store our system needs.

The vector database doesn't offer the concept of a training dataset. But it can be used as a NoSQL database. Which 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 database for additional context using vector search techniques.

For our use case, this is more than enough because:

- 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, post 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:

- How do you implement an LLM agnostic pipeline?
- What fine-tuning techniques should you use?
- How to scale the fine-tuning algorithm on LLMs and datasets of various sizes?
- How to pick an LLM production candidate from multiple experiments?
- How to 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 **continuous training (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 database 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 do 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 a LLM and RAG system, such as:

- 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 database 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

The 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 a minimum viable product (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 feature/training/inference (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 including an ML platform, orchestrator, and

infrastructure as code (**IaC**) tool to support all MLOps best practices.

7 Inference Optimization

Deploying LLMs is challenging due to their significant computational and memory requirements. Efficiently running these models necessitates the use of specialized accelerators, such as GPUs or TPUs, which can parallelize operations and achieve higher throughput. While some applications, like document generation, can be processed in batches overnight, others require low latency and fast generation, such as code completion. As a result, optimizing the inference process – how these models make predictions based on input data – is critical for many practical applications. This includes reducing the time it takes to generate the first token (latency), increasing the number of tokens generated per second (throughput), and minimizing the memory footprint of LLMs.

Indeed, naive deployment approaches lead to poor hardware utilization and underwhelming throughput and latency. Fortunately, a variety of optimization techniques have emerged to dramatically speed up inference. This chapter will explore key methods like speculative decoding, model parallelism, and weight quantization, demonstrating how thoughtful implementations can achieve speedups of 2-4X or more. We will also introduce three popular inference engines (Text

Generation Inference, vLLM, and TensorRT-LLM) and compare their features in terms of inference optimization.

In this chapter, we will cover the following topics:

- Model optimization strategies
- Model parallelism
- Model quantization

By the end of this chapter, you will understand the core challenges in LLM inference, be familiar with state-of-the-art optimization techniques, including model parallelism and weight quantization.

Technical requirements

All the code examples from this chapter can be found on GitHub at [INSERT LINK]

Model optimization strategies

In this section, we will detail several optimization strategies that are commonly used to speed up inference and reduce Video Random Access Memory (VRAM) usage, such as implementing a (static) *KV cache*, *continuous batching*, *speculative decoding*, and *optimized attention mechanisms*.Most of the LLMs used nowadays, like GPT or Llama, are powered by a decoder-only Transformer architecture. The *decoder-only* architecture is designed for text generation tasks. It predicts the next word in a sequence based on preceding words, making it effective for generating contextually appropriate text continuations.

In contrast, an *encoder-only* architecture, like BERT, focuses on understanding and representing the input text with detailed embeddings. It excels in tasks that require comprehensive context understanding, such as text classification and named entity recognition. Finally, the encoder-decoder architecture, like T5, combines both functionalities. The encoder processes the input text to generate a context-rich representation, which the decoder then uses to produce the output text. This dual structure is particularly powerful for sequence-to-sequence tasks like translation and summarization, where understanding the input context and generating a relevant output are equally important.

In this chapter, we only focus on the decoder-only architecture, which dominates the LLM field.

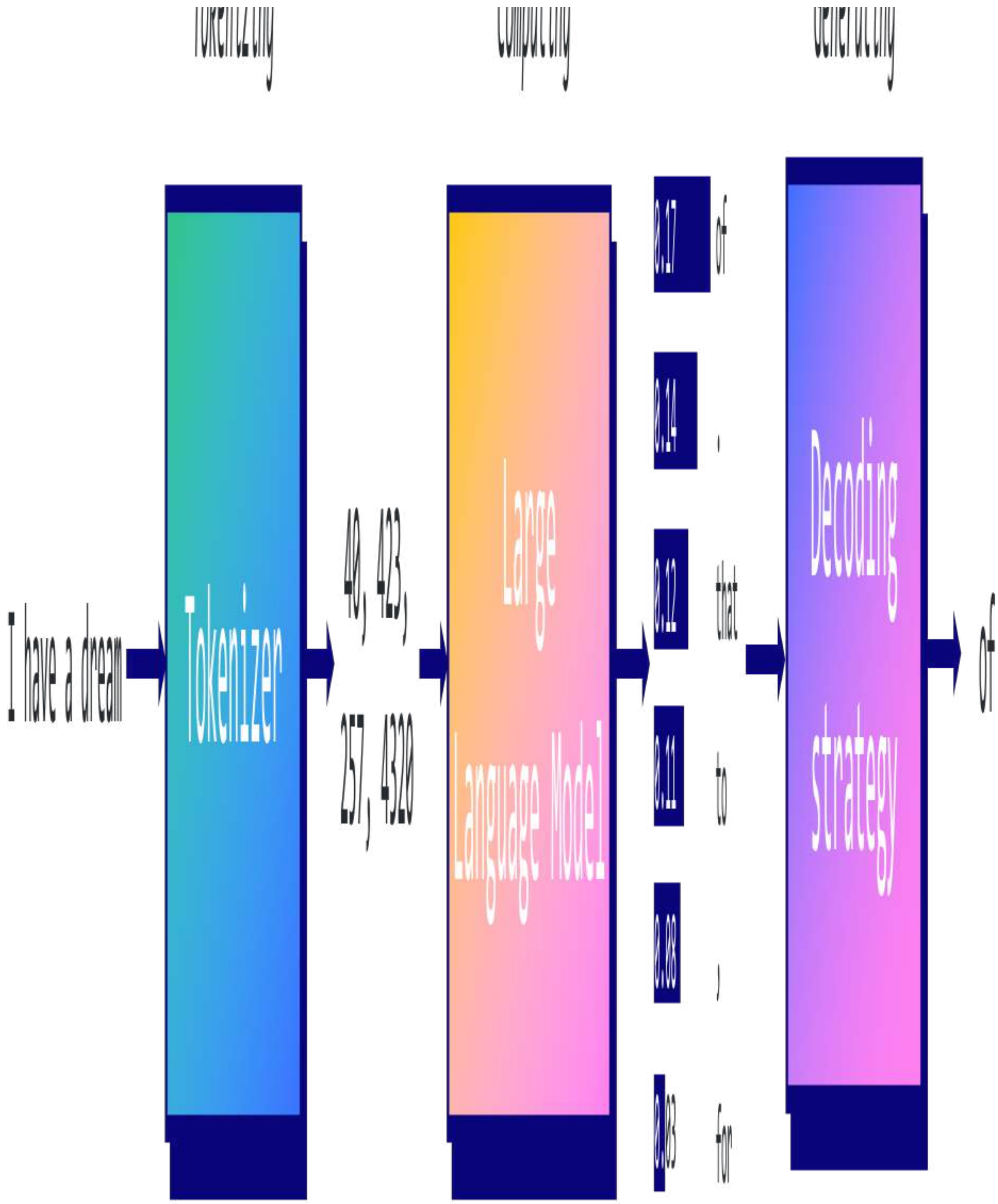


Figure 7.1 – Inference process with decoder-only models. We provide "I have a dream" as input and obtain "of" as output.

As shown in Figure 7.1, the basic inference process for a decoder-only model involves:

1. **Tokenizing** the input prompt and passing it through an embedding layer and positional encoding
2. **Computing** key and value pairs for each input token using the multi-head attention mechanism
3. **Generating** output tokens sequentially, one at a time, using the computed keys and values

While *Steps 1* and *2* are computationally expensive, they consist of highly parallelizable matrix multiplication that can achieve high hardware utilization on accelerators like GPUs and TPUs.

The real challenge is that the token generation in *Step 3* is inherently sequential – to generate the next token, you need to have generated all previous tokens. This leads to an iterative process where the output sequence is grown one token at a time, failing to leverage the parallel computing capabilities of the hardware. Addressing this bottleneck is one the core focuses of inference optimization.

KV cache

We saw that LLMs generate text token by token, which is slow because each new prediction depends on the entire previous context. For example, to predict the 100th token in a sequence, the model needs the context of tokens 1 through 99. When predicting the 101st token, it again needs the information from tokens 1 through 99, plus token 100. This repeated computation is particularly inefficient.

The **key-value** (**KV**) cache addresses this issue by storing key-value pairs produced by self-attention layers. Instead of recalculating these pairs for each new token, the model retrieves them from the cache, significantly speeding up the generation. You can see an illustration of this technique in Figure 7.2:

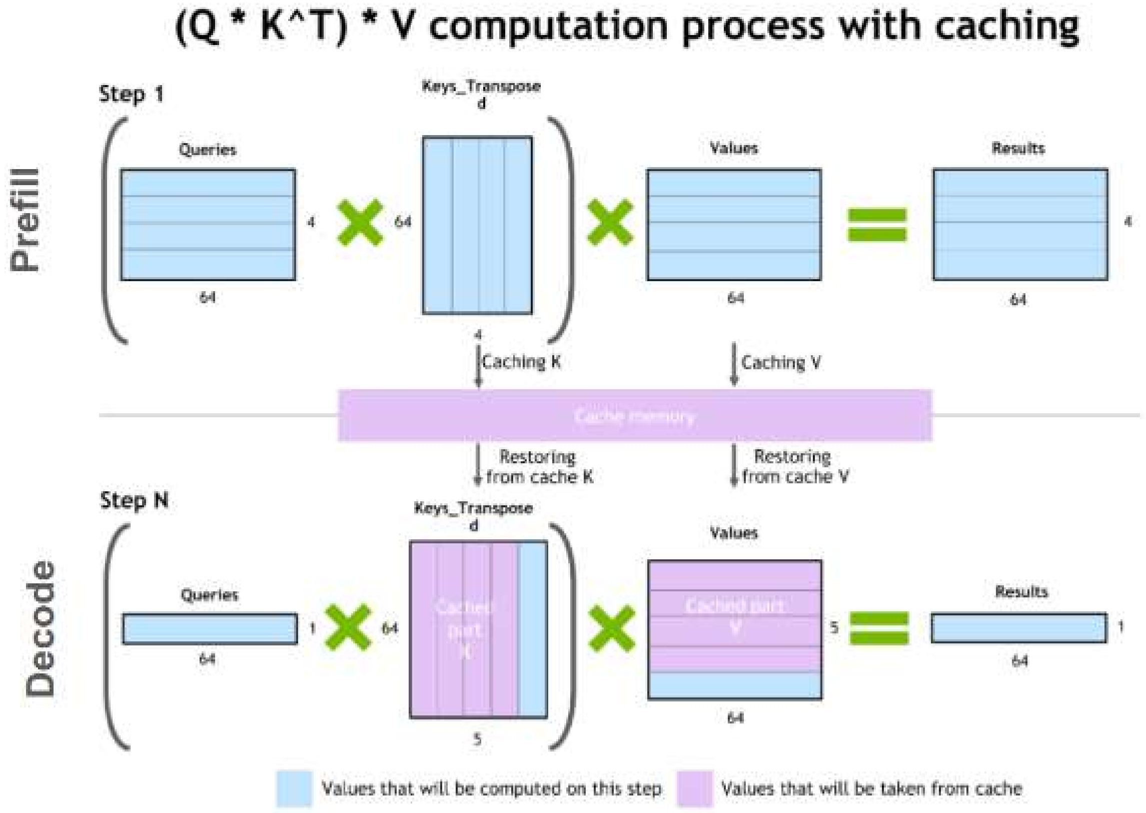


Figure 7.2 – Illustration of the KV cache.

When a new token is generated, only the key and value for that single token need to be computed and added to the cache. The KV cache is an immediate optimization that is implemented in every popular tool and library. Some implementations maintain a separate KV cache for each layer of the model.

The size of the KV cache scales with the number of tokens () and several model dimensions, like the number of layers (), the number of attention heads (), their dimension (), and the precision of the parameters in bytes ():

For a typical 7B parameter model using 16-bit precision, this exceeds 2 GB for high sequence lengths (higher than 2048 tokens). Larger models with more layers and higher embedding dimensions will see even greater memory requirements.

Since the KV cache grows with each generation step and is dynamic, it prevents taking advantage of torch.compile, a powerful optimization tool that fuses PyTorch code into fast and optimized kernels. The *static KV cache* solves this issue by pre-allocating the KV cache size to a maximum value which allows combining it with torch.compile for up to a 4x speed up in the forward pass.

To configure a model to use a static KV cache with the transformers library:

1. We import the tokenizer and the model we want to optimize.

```
import torchx
from transformers import AutoTokenizer, AutoModelForC
model_id = "google/gemma-2b-it"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id
```

1. To implement the static cache, we change the cache implementation in the model's generation config to `"static"`.

```
model.generation_config.cache_implementation = "stati
```

1. Now that our KV cache is static, we can compile the model using torch.compile.

```
compiled_model = torch.compile(model, mode="reduce-ov
```

1. We tokenize an input question "What is a 2+2?" and store it on a GPU if available (if not, we store it on the CPU)

```
device = "cuda" if torch.cuda.is_available() else "cp
inputs = tokenizer("What is 2+2?", return_tensors="pt
```

1. Let's use the `generate()` method to get the model's output and decode it with `batch_decode()` to print its answer.

```
outputs = model.generate(**inputs, do_sample=True, te
print(tokenizer.batch_decode(outputs, skip_special_to
['What is 2+2?\n\nThe answer is 4. 2+2 = 4.']
```

This returns a list containing both the input and output, correctly answering our question.

> *Note that the static cache doesn't work with all architectures. For details on which architectures are supported, check the transformers documentation.*

Efficiently managing the KV cache is essential as it can quickly exhaust available GPU memory and limit the batch sizes that can be processed. This has motivated the development of memory-efficient attention mechanisms and other techniques we will cover in the last section.

Continuous batching

Batching, or processing multiple inference requests simultaneously, is a standard approach to achieve high throughput. Larger batch sizes spread out the memory cost of the model weights and transfer more data to the GPU at once, better saturating its parallel compute capacity.

However, decoder-only models pose a particular challenge due to the high variability in input prompt lengths and desired output lengths. Some requests may have short prompts and only need a one-word answer, while others may input a lengthy context and expect a multi-paragraph response.

With traditional batching, we would have to wait for the longest request in a batch to complete before starting a new batch. This leads to under-utilization as the accelerator sits partly idle waiting for a straggling request to finish. *Continuous batching*, also known as "in-flight" batching, aims to avoid idle time by immediately feeding a new request into the batch as soon as one completes.

The batching process begins the same - by filling the batch with initial requests. But as soon as a request completes its generation, it is evicted from the batch and a new request takes its place. In this way, the accelerator is always processing a full batch, leading to maximally efficient hardware utilization. An additional consideration is the need to periodically pause the generation process to run prefill, or the embedding and encoding of waiting requests. Finding the optimal balance between generation and prefill requires some tuning of the waiting-served ratio hyperparameter.

Continuous batching is natively implemented in most inference frameworks, like Hugging Face's Text Generation Inference (TGI), vLLM, and NVIDIA TensorRT-LLM.

Speculative decoding

Another powerful optimization technique is *speculative decoding*, also called assisted generation. The key insight is that even with continuous batching, the token-by-token generation process fails to fully saturate the parallel processing capabilities of the accelerator. Speculative decoding aims to use this spare compute capacity to predict multiple tokens simultaneously, using a smaller proxy model (see Figure 7.3).

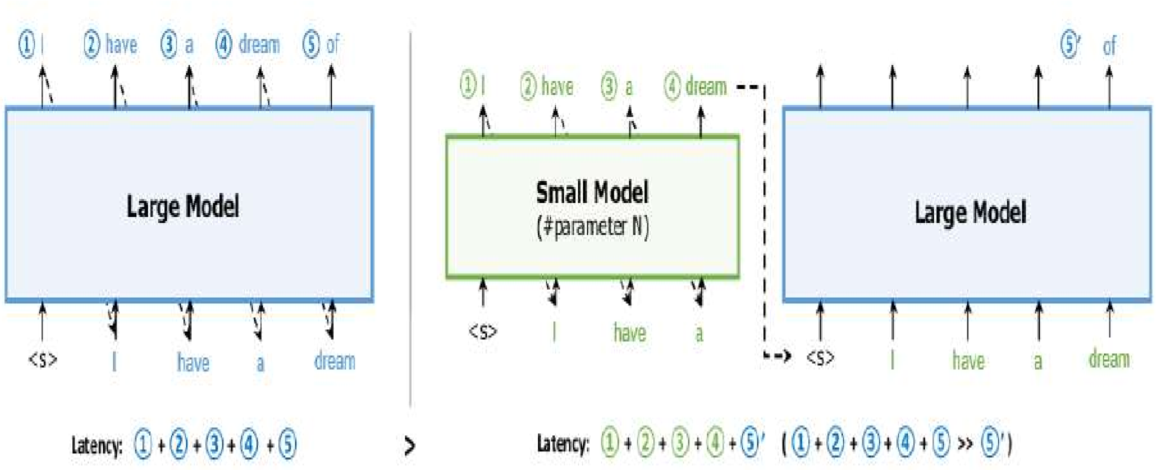


Figure 7.3 – Illustration of traditional decoding (left) and speculative decoding (right)

The general approach is:

- Apply a smaller model, like a distilled or pruned version of the main model, to predict multiple token completions in

parallel. This could be 5-10 tokens predicted in a single step.
- Feed these speculative completions into the full model to validate which predictions match what the large model would have generated.
- Retain the longest matching prefix from the speculative completions and discard any incorrect tokens.

The result is that, if the small model approximates the large model well, multiple tokens can be generated in a single step. This avoids running the expensive large model for some number of iterations. The degree of speedup depends on the quality of the small model's predictions – a 90% match could result in a 3-4X speedup.

It is crucial that both models use the same tokenizer. If this is not the case, the tokens generated by the draft model will not align with those produced by the large model, making them incompatible. Let's implement it using the transformers library. In this example, we will use two Qwen1.5 models from Alibaba Cloud: a 1.8B version as the main model, and a 0.5B version as the draft model. Note that, if you have enough VRAM, you can use much larger models like 14B, 32B, 72B, or 110B as the main model. Here, we're limited by the VRAM of the T4 GPU in Google Colab, but to get the largest speed up, the assistant model should be much smaller than the large model.

Here's a step-by-step guide to implement speculative decoding:

1. Load the tokenizer and both models.

```
import torch
from transformers import AutoTokenizer, AutoModelForC
model_id = "Qwen/Qwen1.5-1.8B-Chat"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id
draft_model = AutoModelForCausalLM.from_pretrained("Q
```

1. We then tokenize the same input and store it in the accelerator if available.

```
device = "cuda" if torch.cuda.is_available() else "cp
inputs = tokenizer("What is 2+2?", return_tensors="pt
```

1. We can now use `model.generate()` with the argument "`assistant_model`" to enable speculative decoding.

```
outputs = model.generate(**inputs, do_sample=True, as
print(tokenizer.batch_decode(outputs, skip_special_to
['What is 2+2? 2 + 2 equals 4!']
```

The speedup in this small example is not significant, but it is clearly noticeable with bigger models.

Prompt lookup decoding is a variant of speculative decoding tailored to input-grounded tasks like summarization where there is often overlap between the prompt and output. Shared n-grams are used as the LLM candidate tokens. We can enable it by using the "`prompt_lookup_num_tokens`" parameter in `model.generate()`:

```
outputs = model.generate(**inputs, prompt_lookup_num_
```

By combining static KV cache with torch.compile, implementing continuous batching, and leveraging speculative decoding techniques, LLMs can see inference speedups of 2-4x or more with no loss in quality.

Another approach to create a small proxy model consists of jointly fine-tuning a small model alongside the large model for maximum fidelity. A representative technique here is Medusa, which inserts dedicated speculation heads into the main model. A Medusa-1 approach fine-tunes these speculation heads while freezing the large model, while a Medusa-2 approach jointly fine-tunes both the speculation heads and the large model. The

Medusa method has demonstrated impressive results, enabling a 70M parameter model to closely approximate the performance of a 7B parameter model on a range of tasks.

Speculative decoding is natively supported by TGI.

Optimized attention mechanisms

The Transformer architecture is based on the attention mechanism, which scales quadratically with the number of input tokens (or sequence length). This is particularly inefficient for longer sequences, where the size of the KV cache can blow up.

Introduced by Kwon, Li, et al. (2023), *PagedAttention* addresses these memory challenges by drawing inspiration from virtual memory and paging in operating systems. It partitions the KV cache into blocks, eliminating the need for contiguous memory allocation. Each block contains the keys and values for a fixed number of tokens. During attention computation, the PagedAttention kernel efficiently fetches these blocks, regardless of their physical memory location.

This partitioning allows for near-optimal memory utilization. This is useful for batching more sequences together, which increases throughput and GPU utilization. Moreover, PagedAttention's block-based approach naturally supports memory sharing across multiple output sequences generated from the same prompt. This is particularly advantageous in parallel sampling and beam search, where the same prompt is used to generate multiple outputs. The shared memory blocks reduce redundant computations and memory usage, cutting the memory overhead by up to 55% and improving throughput by up to 2.2x according to the authors. The vLLM library received the first implementation of PagedAttention. Since then, PagedAttention has also been implemented in TGI and TensorRT-LLM.

Another popular option is *FlashAttention-2*. Developed by Tri Dao (2023), it introduces several key innovations that are designed to address the quadratic runtime and memory constraints in traditional attention. By dividing input and output matrices into smaller blocks, FlashAttention-2 ensures that these blocks can fit into the GPU's on-chip SRAM, which is much faster than high-bandwidth memory. This approach significantly reduces the frequency of data transfers between the GPU's main memory and its processing units. This is combined with online softmax, which computes the softmax function independently for each block of the attention scores matrix, rather than for the entire matrix at once. By maintaining a running maximum and a running sum of exponentials, FlashAttention-2 can calculate attention probabilities without needing to store large intermediate matrices.

Additionally, FlashAttention-2's online softmax computation allows for block-wise processing, maintaining accuracy while significantly reducing memory requirements. This is particularly important for training, where recomputation of intermediate values (instead of storing them) in the backward pass reduces memory usage from quadratic to linear in relation to sequence length.

FlashAttention-2 can easily be used with the transformers library:

1. Install the `flash-attn` library with `--no-build-isolation` so we don't install the dependencies:

```
pip install flash-attn --no-build-isolation
```

1. To use FlashAttention-2 for inference, specify `"flash_attention_2"` in the `attn_implementation` parameter when loading a model. For example, here is how to load Mistral-7B-Instruct-v0.3 with FlashAttention-2:

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.3",
    attn_implementation="flash_attention_2",
)
```

The techniques presented in this section focused on improving the model's efficiency to process tokens. In the next section, we will discuss how to distribute our model and calculations across multiple GPUs.

Model parallelism

Model parallelism allows distributing the memory and compute requirements of LLMs across multiple GPUs. This enables training and inference of models too large to fit on a single device, while also improving performance in terms of throughput (tokens per second).

There are three main approaches to model parallelism, each involving splitting the model weights and computation in different ways: *data parallelism*, *pipeline parallelism* and *tensor parallelism*. Although these approaches were originally developed for training, but we can re-use them for inference by focusing on the forward pass only.

Data parallelism

Data parallelism (DP) is the simplest type of model parallelism. It involves making copies of the model and distributing these replicas across different GPUs (see Figure 7.5). Each GPU processes a subset of the data simultaneously. During training, the gradients calculated on each GPU are averaged and used to update the model parameters, ensuring that each replica remains synchronized. This approach is particularly beneficial when the batch size is too large to fit into a single machine or when aiming to speed up the training process.

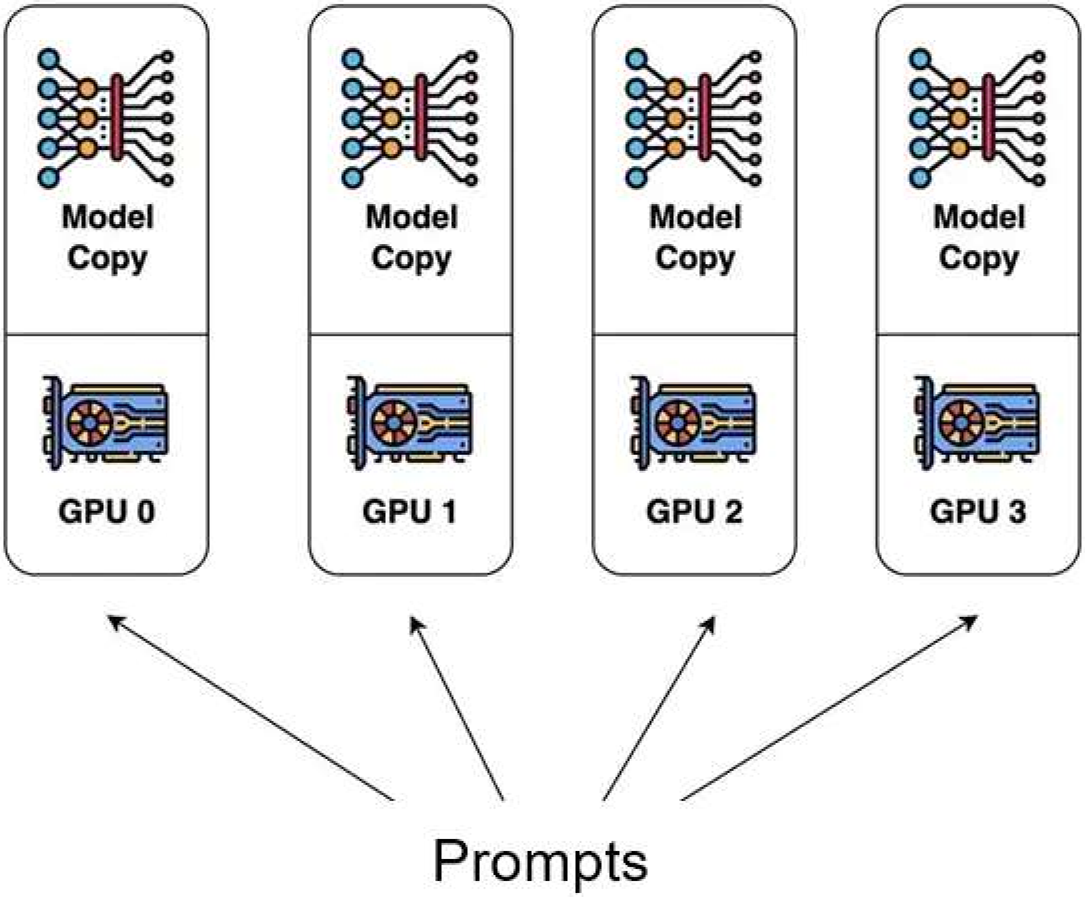


Figure 7.5 – Illustration of data parallelism with four GPUs

During inference, DP can be useful for processing concurrent requests. By distributing the workload across multiple GPUs, this approach helps reduce latency, as multiple requests can be handled simultaneously. This concurrent processing also increases throughput, since a higher number of requests can be processed at the same time.

However, the effectiveness of DP is limited by the model size and the communication overhead between GPUs. Indeed, replicating the model's parameters on each GPU is inefficient. It means this technique only works when the model is small enough to fit into a single GPU, leaving less room for input data and thus limiting the batch size. For larger models or when memory is a constraint, this can be a significant drawback.

Typically, DP is mainly used for training, while pipeline and tensor parallelism are preferred for inference.

Pipeline parallelism

Introduced by Huang et al. in the GPipe paper (2019), **pipeline parallelism** (**PP**) is a strategy for distributing the computational load of training and running large neural networks across multiple GPUs.

Unlike traditional DP, which replicates the entire model on each GPU, pipeline parallelism partitions the model's layers across different GPUs. This approach allows each GPU to handle a specific portion of the model, thereby reducing the memory burden on individual GPUs.

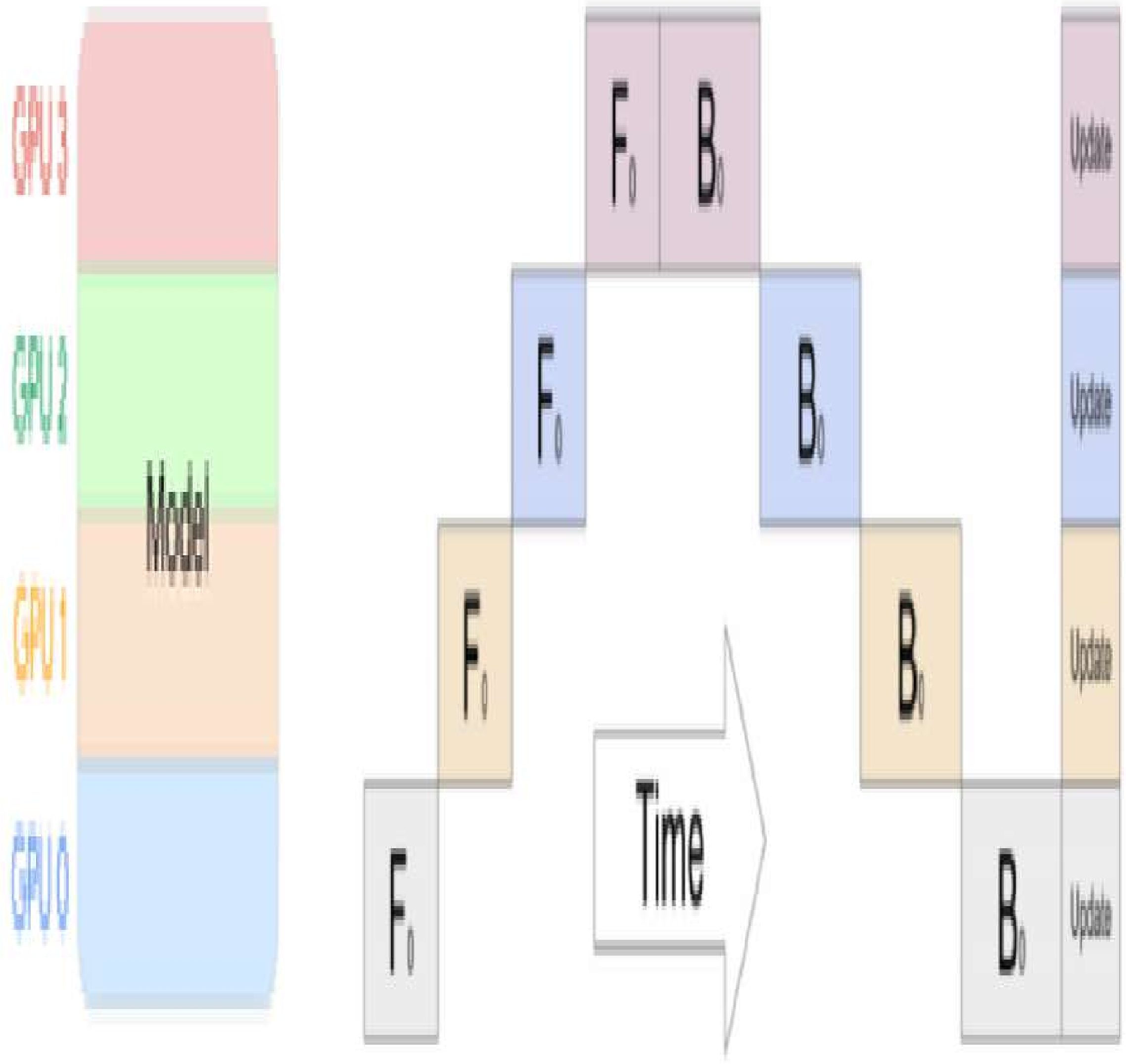


Figure 7.4 – Illustration of pipeline parallelism with four GPUs

As shown in Figure 7.4, a typical 4-way pipeline parallel split, the model is divided into four segments, with each segment assigned to a different GPU. The first 25% of the model's layers might be processed by GPU 1, the next 25% by GPU 2, and so on. During the forward pass, activations are computed and then passed along to the next GPU. For training, the backward pass follows a similar sequence in reverse, with gradients being propagated back through the GPUs. The number of GPUs is often referred to as the degree of parallelism.

The primary advantage of pipeline parallelism is its ability to significantly reduce the memory requirements per GPU. However, this approach introduces new challenges, particularly related to the sequential nature of the pipeline. One of the main issues is the occurrence of "pipeline bubbles." These bubbles arise when some GPUs are idle, waiting for activations from preceding layers. This idle time can reduce the overall efficiency of the process.

Micro-batching has been developed to mitigate the impact of pipeline bubbles. By splitting the input batch into smaller sub-batches, micro-batching ensures that GPUs remain busier, as the next sub-batch can begin processing before the previous one is fully completed.

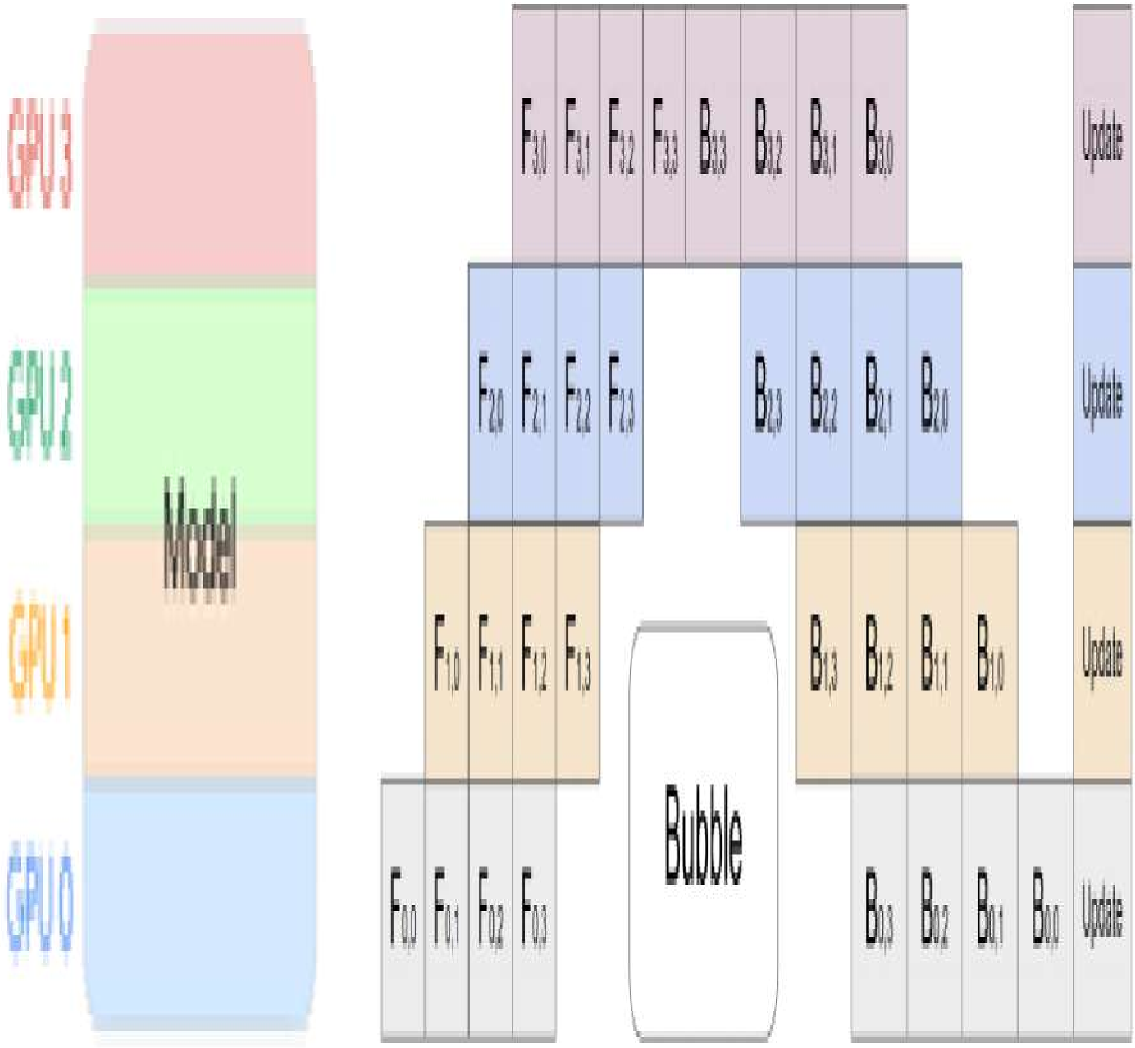


Figure 7.6 – Illustration of pipeline parallelism with micro-batching.

Figure 7.6 shows an example of pipeline parallelism with micro-batching. In this example, the pipeline has four stages (F0, F1, F2, F3), and the input batch is divided into four micro-batches. GPU 0 will process forward paths F0,0, F0,1, F0,2, and F0,3 sequentially. Once F0,0 is complete, GPU 1 can immediately start processing F1,0 and so on. After completing these forward passes, GPU 0 waits for the other GPUs to finish their respective forward computations before starting the backward paths (B0,3, B0,2, B0,1, B0,0).

Pipeline parallelism is implemented in distributed training frameworks like Megatron-LM, DeepSpeed (ZeRO) and PyTorch through the dedicated PiPPy (Pipeline Parallelism for PyTorch) library. As of this date, only certain inference frameworks like TensorRT-LLM support pipeline parallelism.

Tensor Parallelism

Introduced by Shoeby, Patwary, Puri et al. in the Megatron-LM paper (2019), tensor parallelism (TP) is another popular technique to distribute the computation of LLM layers across multiple devices. In contrast to pipeline parallelism, TP splits the weight matrices found in individual layers. This enables simultaneous computations, significantly reducing memory bottlenecks and increasing processing speed.

In TP, large matrices, such as the weight matrices in MLPs or the attention heads in self-attention layers, are partitioned across several GPUs. Each GPU holds a portion of these matrices and performs computations on its respective slice.

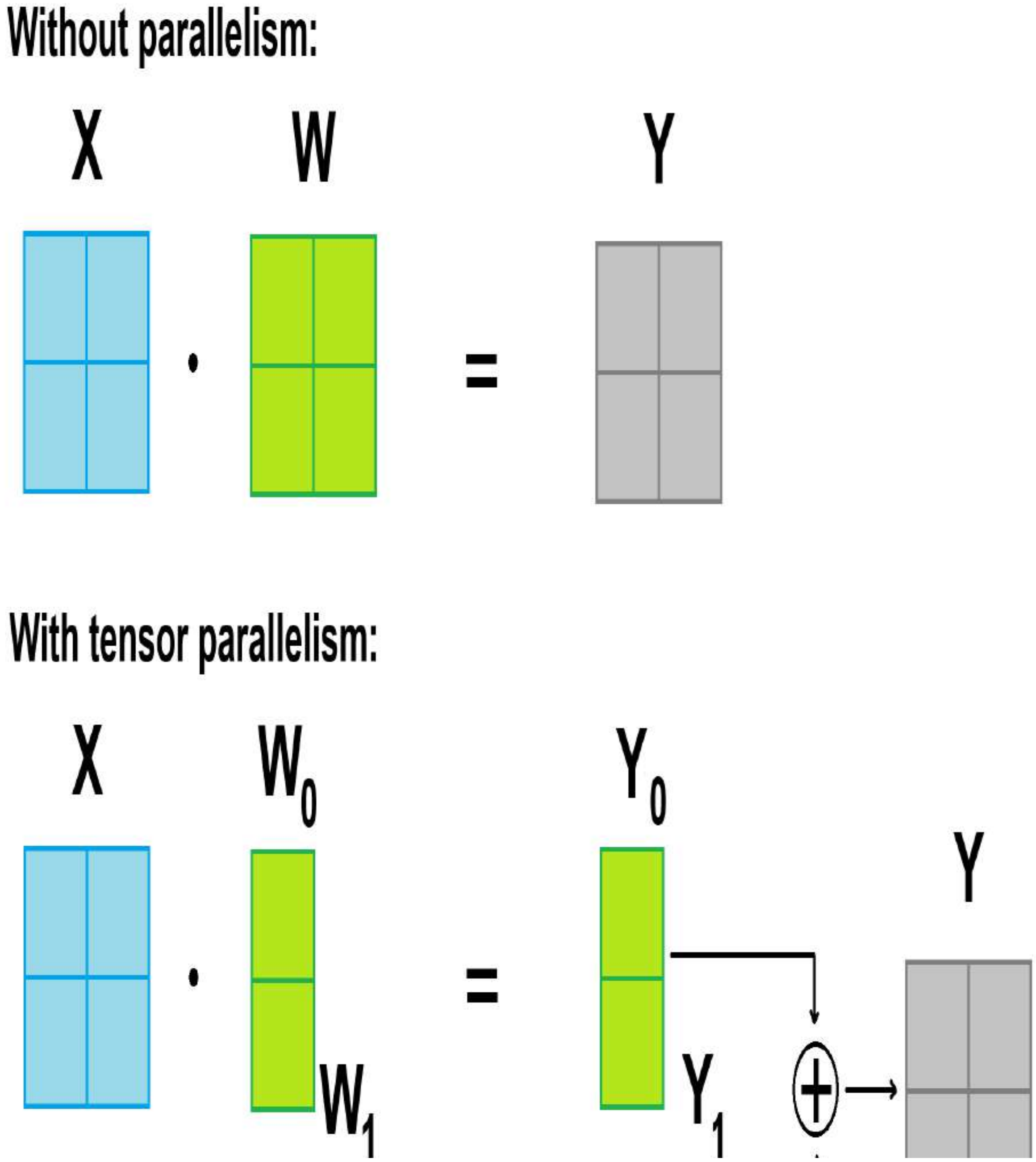


Figure 7.7 – Illustration of column-wise tensor parallelism in an MLP layer (W).

For instance, in an MLP layer, the weight matrix is divided so that each GPU processes only a subset of the weights (see Figure 7.7). The inputs are broadcast to all GPUs, which then independently compute their respective outputs. The partial results are then aggregated through an all-reduce operation, combining them to form the final output.

In the context of self-attention layers, TP is particularly efficient due to the inherent parallelism of attention heads. Each GPU can compute a subset of these heads independently, allowing the model to process large sequences more effectively. This makes TP more efficient than pipeline parallelism, which requires waiting for the completion of previous layers.

Despite its advantages, TP is not universally applicable to all layers of a neural network. Layers like LayerNorm and Dropout, which have dependencies spanning the entire input, cannot be efficiently partitioned and are typically replicated across devices instead. However, these operations can be split on the sequence dimension of the input instead (sequence parallelism). Different GPUs can compute these layers on different slices of the input sequence, avoiding replication of weights. This technique is limited to a few specific layers, but it can provide additional memory savings, especially for very large input sequence lengths.

Moreover, TP necessitates high-speed interconnects between devices to minimize communication overhead, making it impractical to implement across nodes with insufficient interconnect bandwidth.

Tensor parallelism is also implemented in distributed training frameworks like Megatron-LM, DeepSpeed (ZeRO) and PyTorch (FSDP). It is available in most inference frameworks like TGI, vLLM, and TensorRT-LLM.

Combining approaches

Data, tensor, and pipeline parallelisms are orthogonal techniques that can be combined. Figure 7.8 illustrates how a given model can be split according to each approach:

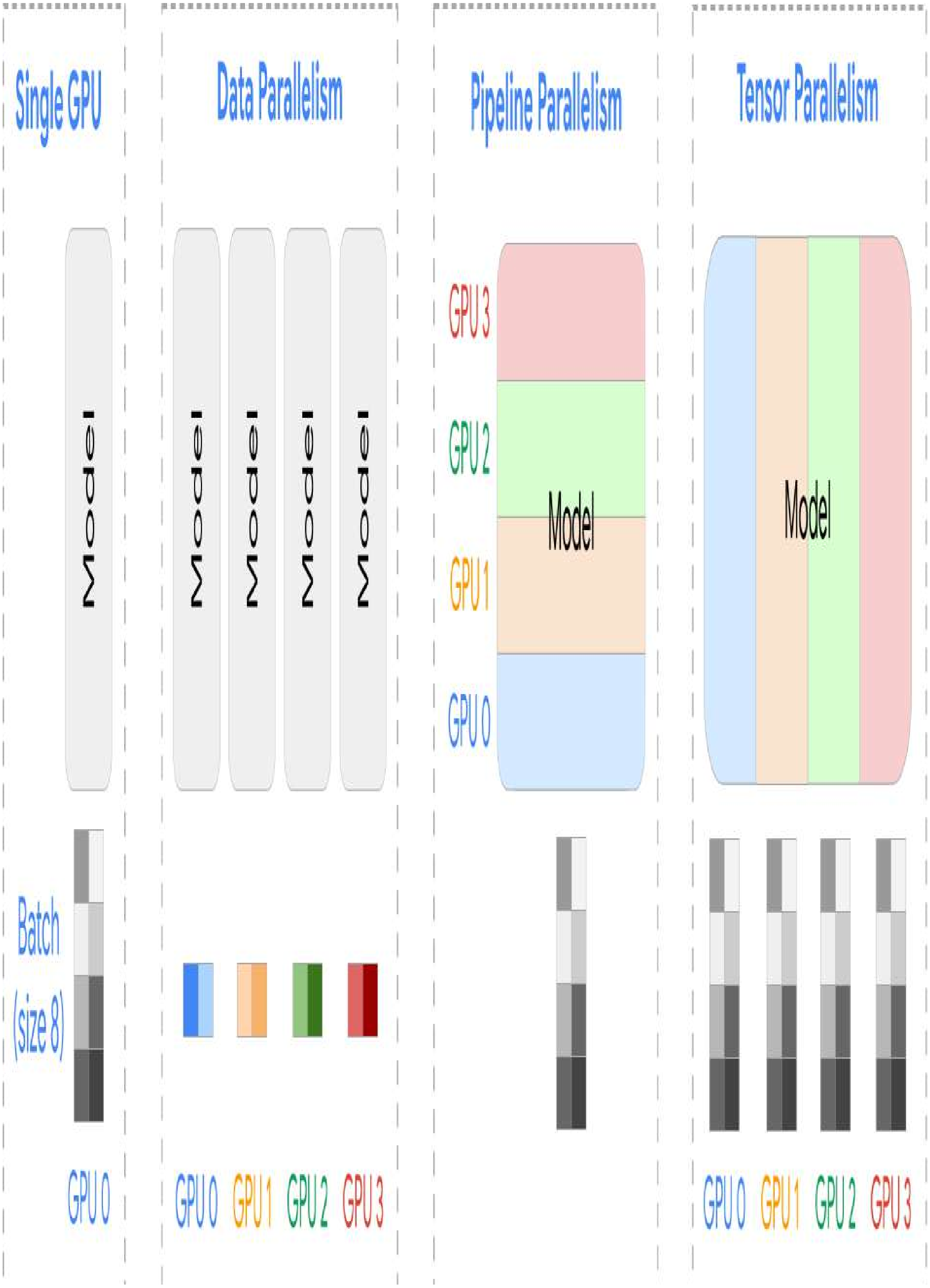

Single GPU
Model
Batch
(size 8)
GPU 0
Data Parallelism
Model
Model
Model
Model
GPU 0
GPU 1
GPU 2
GPU 3
Pipeline Parallelism
GPU 3
GPU 2
GPU 1
GPU 0
Model
Tensor Parallelism
Model
GPU 0
GPU 1
GPU 2
GPU 3

Figure 7.8 – Illustration of the different model parallelism techniques

Combining these techniques can mitigate their respective issues. Pipeline parallelism provides the greatest memory reduction, but sacrifices efficiency due to pipeline bubbles. This may be ideal if the primary constraint is fitting the model in the GPU memory. In contrast, if low latency is paramount, then prioritizing tensor parallelism and accepting a larger memory footprint may be the better tradeoff. In practice, a model may be split depth-wise into a few pipeline stages, with tensor parallelism used within each stage.

Balancing these tradeoffs and mapping a given model architecture onto available hardware accelerators is a key challenge in deploying LLMs.

Model quantization

Quantization refers to the process of representing the weights and activations of a neural network using lower precision data types. In the context of LLMs, quantization primarily focuses on reducing the precision of the model's weights and activations. By default, weights are typically stored in a 16-bit or 32-bit floating-point format (FP16 or FP32), which provides high precision but comes at the cost of increased memory usage and computational complexity. Quantization is a solution to reduce the memory footprint and accelerate the inference of LLMs.

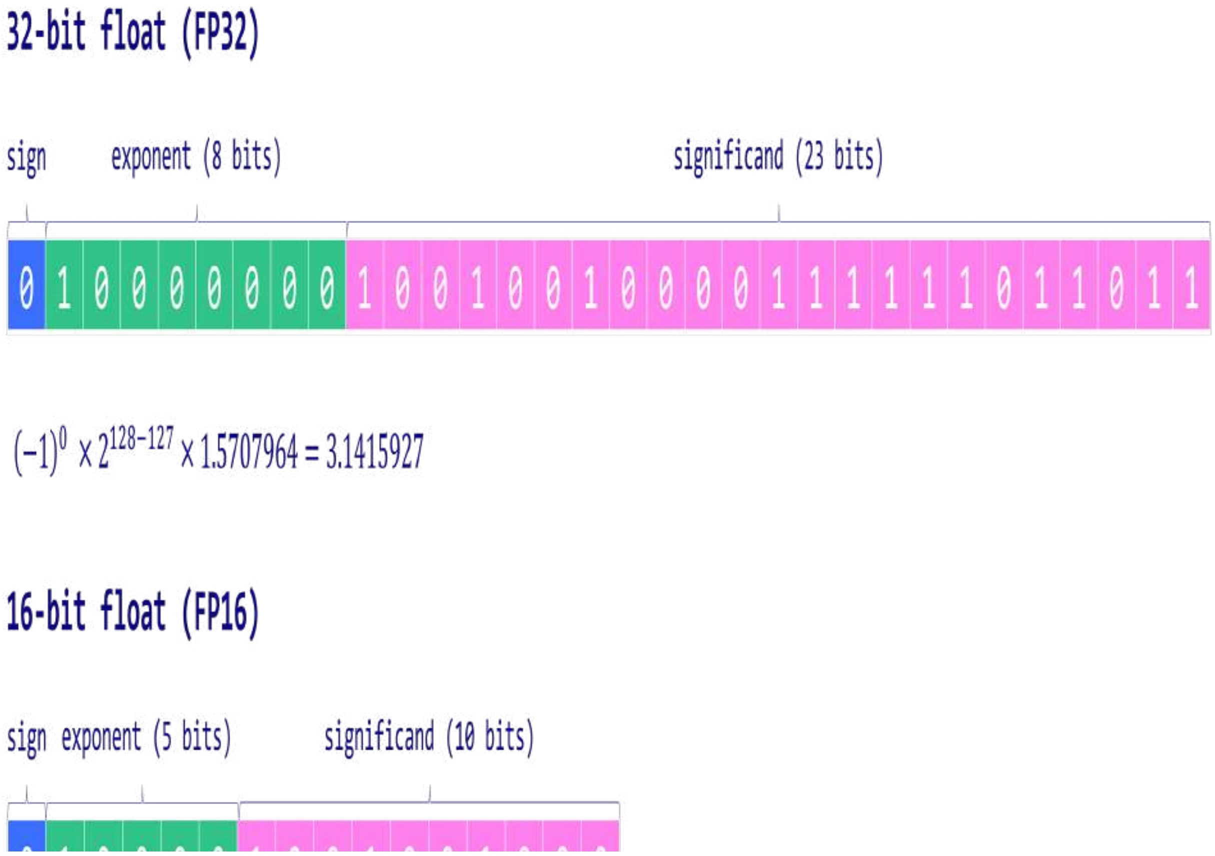
In addition to these benefits, larger models with over 30 billion parameters can outperform smaller models (7B-13B LLMs) in terms of quality when quantized to 2 or 3-bit precision. This means they can achieve superior performance while maintaining a comparable memory footprint.

In this section, we will introduce the concepts of quantization, GGUF with llama.cpp, GPTQ and EXL2, along with an overview of additional techniques. In addition to the code provided in this section, readers can refer to AutoQuant (bit.ly/autoquant) to quantize their models using a Google Colab notebook.

Introduction to quantization

There are two main approaches to weight quantization: *Post-Training Quantization* (PTQ) and *Quantization-Aware Training* (QAT). PTQ is a straightforward technique where the weights of a pre-trained model are directly converted to a lower precision format without any retraining. While PTQ is easy to implement, it may result in some performance degradation. On the other hand, QAT performs quantization during the training or fine-tuning stage, allowing the model to adapt to the lower precision weights. QAT often yields better performance compared to PTQ but requires additional computational resources and representative training data.

The choice of data type plays a crucial role in quantization. Floating-point numbers, such as *FP32*, *FP16* (half-precision), and *BF16* (brain floating-point), are commonly used in deep learning. These formats allocate a fixed number of bits to represent the *sign*, *exponent*, and *significand* (mantissa) of a number.



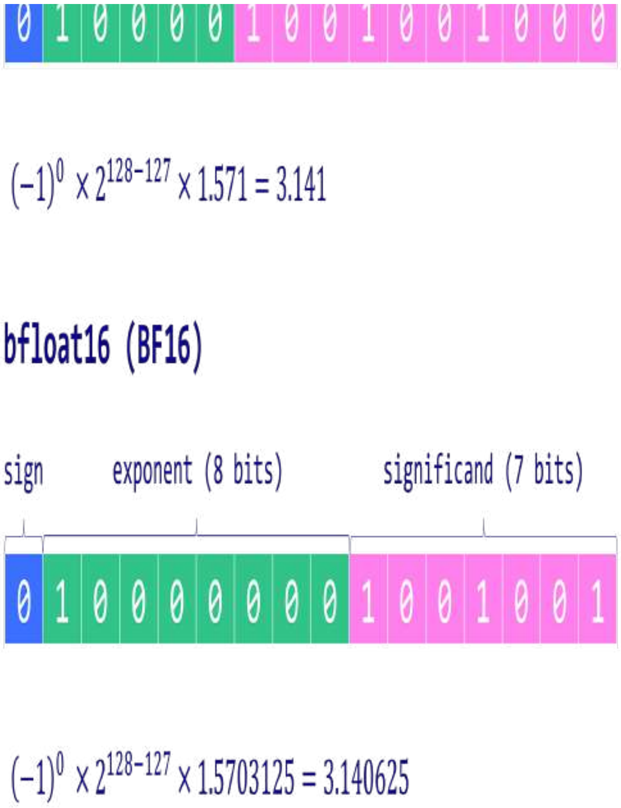
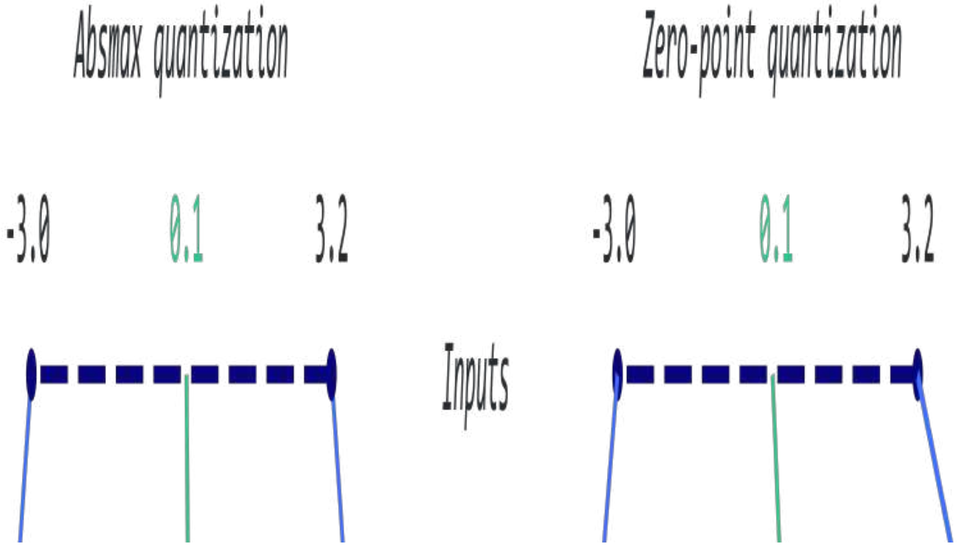


Figure 7.9 – Comparison between FP32, FP16, and BF16 formats.

A sign of 0 represents a positive number, while 1 indicates a negative number. On the other hand, the exponent controls the range that is represented (big or small). Finally, the significand controls the precision of the number (number of digits). The formula used to convert these representations into real numbers is:

The data types shown in Figure 7.7 display different tradeoffs, as illustrated with different representations of (). FP32 uses 32 bits, providing high precision but also requiring more memory. On the other hand, FP16 and BF16 use 16 bits, lowering the memory footprint at the cost of a lower precision. In general, neural networks prefer bigger range than better precision, which is why BF16 is the most popular data type when the hardware supports it. For example, NVIDIA's Ampere architecture (A100, A30, etc.) supports BF16 but previous generations like Turing (T4, T40, etc.) do not.

However, we are not restricted to these three data types. Lower-precision data types, such as INT8 (8-bit integers), can be employed for quantization, further reducing the memory footprint. Naïve quantization techniques, such as *absolute maximum (absmax) quantization* and *zero-point quantization*, can be applied to convert FP32, FP16, or BF16 weights to INT8, as illustrated in Figure 7.10:



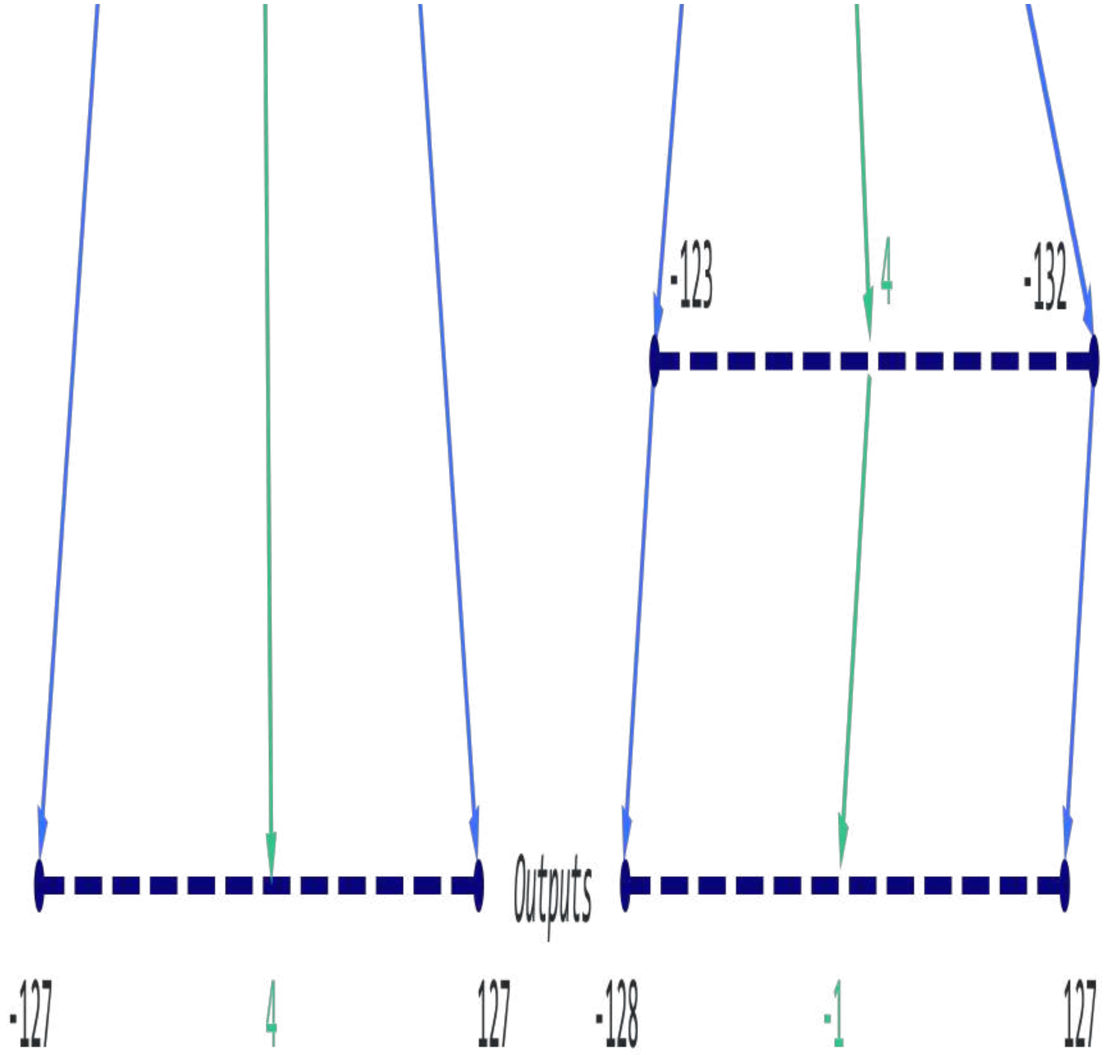


Figure 7.10 – Quantization of 0.1 in a [-3.0, 3.2] range with absmax quantization and zero-point quantization

Absmax quantization maps the original weights to the range [-127, 127] by dividing them by the absolute maximum value of and scaling them:

For example, if our absolute maximum value is 3.2 (see Figure 7.8), a weight of 0.1 would be quantized to . To dequantize it, we do the inverse operation:

This means that if we dequantize our weight, we obtain . We see a rounding error of in this example. In Python, we can implement it as follows with the PyTorch library:

```
import torch
def absmax_quantize(X):
    # Calculate scale
    scale = 127 / torch.max(torch.abs(X))
    # Quantize
    X_quant = (scale * X).round()
    return X_quant.to(torch.int8)
```

Zero-point quantization, on the other hand, considers asymmetric input distributions and maps the weights to the range [-128, 127] by introducing a zero-point offset:

Where and .

If we take the same example with a weight of 0.1, we get a scale of and zero-point value of . The weight of 0.1 would be quantized to , unlike the value of provided by absmax.

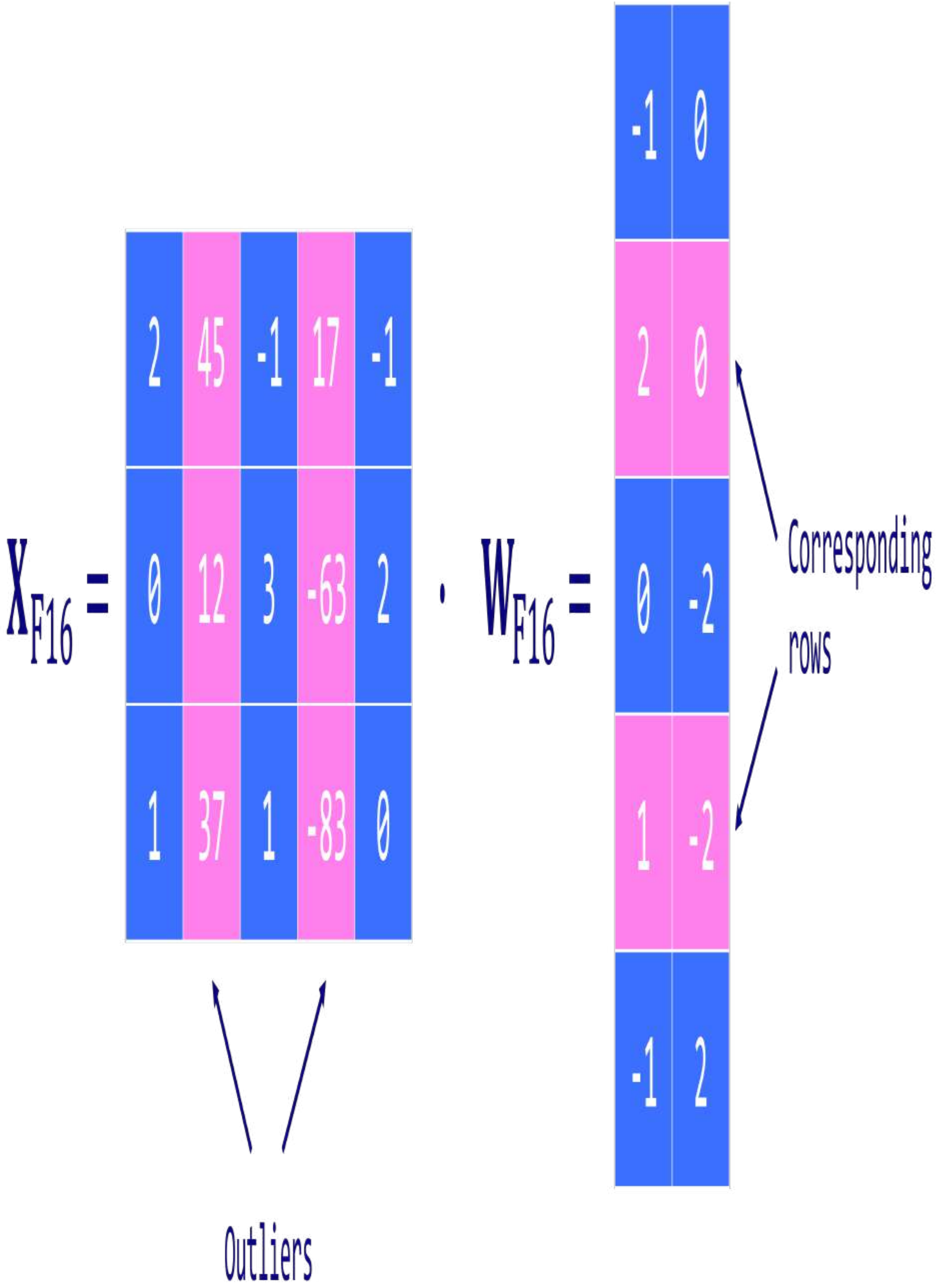
We can easily get the dequantization by applying the inverse operation:

In Python, zero-point quantization can be implemented as follows:

```
def zeropoint_quantize(X):
    # Calculate value range (denominator)
    x_range = torch.max(X) - torch.min(X)
    x_range = 1 if x_range == 0 else x_range
    # Calculate scale
    scale = 255 / x_range
    # Shift by zero-point
    zeropoint = (-scale * torch.min(X) - 128).round()
    # Scale and round the inputs
    X_quant = torch.clip((X * scale + zeropoint).roun

    return X_quant.to(torch.int8)
```

However, naïve quantization methods have limitations, particularly when dealing with *outlier features* in LLMs. Outlier features are extreme weight values (about 0.1% of total values) that can significantly impact the quantization process, leading to reduced precision for other values. Discarding these outliers is not feasible as it would degrade the model's performance. You can see an example of outliers in Figure 7.11:



$$X_{F16} = \begin{bmatrix} 2 & 45 & -1 & 17 & -1 \\ 0 & 12 & 3 & -63 & 2 \\ 1 & 37 & 1 & -83 & 0 \end{bmatrix} \cdot W_{F16} = \begin{bmatrix} -1 & 0 \\ 2 & 0 \\ 0 & -2 \\ 1 & -2 \\ -1 & 2 \end{bmatrix}$$

Figure 7.11 – Example of outliers in a weight matrix.

To address the outlier problem, more advanced quantization techniques have been proposed. One notable example is LLM.int8(), introduced by Dettmers et al. (2022). LLM.int8() employs a mixed-precision quantization scheme, where outlier features are processed using FP16, while the remaining values are quantized to INT8. This approach effectively reduces the memory footprint of LLMs by nearly 2x while minimizing performance degradation.

LLM.int8() works by performing matrix multiplication in three steps. First, it extracts columns containing outlier features from the input hidden states using a custom threshold. Second, it performs separate matrix multiplications for the outliers (in FP16) and non-outliers (in INT8) using vector-wise quantization. Finally, it dequantizes the non-outlier results and combines them with the outlier results to obtain the final output in FP16.

The effectiveness of LLM.int8() has been demonstrated empirically, showing negligible performance degradation (<1%) compared to the original FP32 models. However, it does introduce an additional computational overhead, resulting in around 20% slower inference for large models. Models can be directly loaded in 8-bit precision with the transformer library using LLM.int8() as follows:

```
from transformers import AutoModelForCausalLM
model_name = "meta-llama/Meta-Llama-3-8B-Instruct"
model = AutoModelForCausalLM.from_pretrained(model_na
```

Introduced by Dettmers et al. (2023), NF4 is a 4-bit precision format designed for QLoRA. It is also integrated into the transformers library but requires the bitsandbytes library as a dependency. To load a model in NF4 (4-bit precision), you can use the load_in_4bit parameter as follows:

```
from transformers import AutoModelForCausalLM
model_name = "meta-llama/Meta-Llama-3-8B-Instruct"
model = AutoModelForCausalLM.from_pretrained(model_na
```

Quantization with GGUF and llama.cpp

The llama.cpp project is an open-source C++ software library created by Georgi Gerganov, designed for performing inference with various LLMs. It is the most popular quantization technique, with many quantized models available on the Hugging Face Hub.

Compared to other libraries that rely on hardware-specific closed-source libraries like CUDA, llama.cpp can run on a broader range of hardware. It has gained significant popularity, particularly among users without specialized hardware, as it can operate on CPUs and Android devices. Moreover, llama.cpp can also offload layers to the GPU, accelerating inference speed. It is compatible with different inference optimization techniques, such as FlashAttention-2 and speculative decoding.

This project features its own quantization format, GGUF, designed to simplify and speed up model loading. GGUF files store tensors and metadata, supporting various formats, from 1-bit to 8-bit precision. It follows a naming convention based on the number of bits used and specific variants, such as:

- **IQ1_S, IQ1_M**: 1-bit precision, very low quality.
- **IQ2_XXS/XS/S/M, Q2_K**: 2-bit precision, generally low quality but IQ2 can be usable for large models.
- **IQ3_XXS/XS/S/M, Q3_K_S/M/L**: 3-bit precision, low quality but usable for large models.
- **IQ4_XS/NL, Q4_K_S/M, Q4_0/1**: 4-bit precision, good quality, usable for most models.
- **Q5_K_S/M, Q5_0/1**: 5-bit precision, high quality.
- **Q6_K**: 6-bit precision, very high quality.
- **Q8_0**: 8-bit precision, highest quality.

To provide a brief overview on GGUF quantization, llama.cpp groups values into blocks and rounds them to a lower precision. For instance, the legacy Q4_0 format handles 32 values per block, scaling and quantizing them based on the largest weight value in the block (). In Q4_1, the smallest Lvalue in the block is also added (). In Q4_K, weights are divided into super-blocks, containing 8 blocks with 32 values. Block scales and minimum values are also quantized in higher precision with 6 bits (). Finally, i-quants like IQ4_XS are inspired by another quantization technique called QuIP#. It ensures an even number of positive (or negative) quant signs in groups of eight and implements lattice to store their magnitude.

Here is a practical example of how to quantize a model in GGUF format. The following steps can be executed on a free T4 GPU in Google Colab:

1. Install llama.cpp and the required libraries:

```
!git clone https://github.com/ggerganov/llama.cpp
!cd llama.cpp && git pull && make clean && LLAMA_CUBL
!pip install -r llama.cpp/requirements.txt
```

1. Download the model to convert. We provide the model ID from the Hugging Face Hub. For example, "mistralai/Mistral-

7B-Instruct-v0.2":

```
MODEL_ID = "mlabonne/EvolCodeLlama-7b"
MODEL_NAME = MODEL_ID.split('/')[-1]
!git lfs install
!git clone https://huggingface.co/{MODEL_ID}
```

1. First, we convert the model into FP16. This is an intermediary artifact that will be used for every GGUF quantization type. Note that different conversion scripts exist in llama.cpp and are compatible with different models.

```
fp16 = f"{MODEL_NAME}/{MODEL_NAME.lower()}.fp16.bin"
!python llama.cpp/convert.py {MODEL_NAME} --outtype f
```

1. We select a format (here, Q4_K_M) and start the quantization. This process can take an hour on a T4 GPU:

```
METHOD = "q4_k_m"
qtype = f"{MODEL_NAME}/{MODEL_NAME.lower()}.{method.u
!./llama.cpp/quantize {fp16} {qtype} {METHOD}
```

1. Once it's done, your quantized model is ready. You can download it locally, or upload it to the Hugging Face Hub

using the following code:

```
from huggingface_hub import create_repo, HfApi
hf_token = "" # Specify your token
username = "" # Specify your username
api = HfApi()
# Create empty repo
create_repo(
    repo_id = f"{username}/{MODEL_NAME}-GGUF",
    repo_type="model",
    exist_ok=True,
    token=hf_token
)
# Upload gguf files
api.upload_folder(
    folder_path=MODEL_NAME,
    repo_id=f"{username}/{MODEL_NAME}-GGUF",
    allow_patterns=f"*.gguf",
    token=hf_token
)
```

GGUF models can be used with backends such as llama-cpp-python and frameworks like LangChain. This is useful if you want to integrate a quantized model into a broader system. You can also directly chat with the model using frontends like llama.cpp's lightweight server, LM Studio, and Text Generation

Web UI. These tools enable easy interaction with the GGUF models, providing an experience similar to ChatGPT.

Quantization with GPTQ and EXL2

While GGUF and llama.cpp offer CPU inference with GPU offloading, GPTQ and EXL2 are two quantization formats dedicated to GPUs. This makes them both faster than llama.cpp during inference. In particular, EXL2 offers the highest throughput with its dedicated library, ExLlamaV2.

GPTQ and EXL2 quants are based on the GPTQ algorithm, introduced by Frantar et al. (2023). It optimizes weight quantization for LLMs by refining the Optimal Brain Quantization (OBQ) approach to handle extensive matrices efficiently. It begins with a Cholesky decomposition of the Hessian inverse, ensuring numerical stability. Instead of quantizing weights in a strict order, GPTQ processes them in batches, updating columns and associated blocks iteratively. This method leverages lazy batch updates, reducing computational redundancy and memory bottlenecks.

While GPTQ is limited to 4-bit precision, EXL2 offers more flexibility with a highly customizable precision that can mix different quantization levels. This allows for precise bitrates between 2 and 8 bits per weight, such as 2.3, 3.5, or 6.0. It can also apply multiple quantization levels to each linear layer, prioritizing more important weights with higher bit quantization. Parameters are selected automatically by quantizing each matrix multiple times and choosing a combination that minimizes quantization error while meeting a target bitrate. In practice, this allows 70B models to run on a single 24 GB GPU with 2.55-bit precision.

The inference itself is handled by the ExLlamaV2 library, which supports both GPTQ and EXL2 models.

In the following example, let's quantize a model in the EXL2 format using ExLlamaV2. These steps can be executed on a free T4 GPU in Google Colab:

1. Install the ExLlamaV2 library from source.

```
!git clone https://github.com/turboderp/exllamav2
!pip install -e exllamav2
```

1. We download the model to quantize by cloning its repo from the Hugging Face Hub.

```
MODEL_ID = "meta-llama/Llama-2-7b-chat-hf"
MODEL_NAME = MODEL_ID.split('/')[-1]
!git lfs install
!git clone https://huggingface.co/{MODEL_ID}
```

1. Download the calibration dataset used to measure the quantization error. In this case, we will use WikiText-103, a standard calibration dataset with high-quality articles from Wikipedia:

```
!wget https://huggingface.co/datasets/wikitext/resolv
```

1. Quantize the model at a given precision (for example, 4.5).

```
!mkdir quant
!python exllamav2/convert.py \
    -i {MODEL_NAME} \
    -o quant \
    -c wikitext-test.parquet \
    -b 4.5
```

The quantized model can then be uploaded to the Hugging Face Hub as seen previously.

GPTQ and EXL2 quants are not as widely supported as GGUF. For example, frontends like LM Studio do not currently integrate them. You can use other tools instead, like oobabooga's Text Generation Web UI. It is also directly integrated into the transformers library and supported by TGI. GPTQ models are also supported in TensorRT-LLM.

While less popular than GGUF, you can find a lot of GPTQ and EXL2 models on the Hugging Face Hub.

Other quantization techniques

There is a variety of quantization techniques beyond GGUF, GPTQ, and EXL2. This subsection will briefly introduce **Activate-aware Weight Quantization** (**AWQ**) as well as extreme quantization techniques like QuIP# and HQQ.

Introduced by Lin et al. (2023), AWQ is another popular quantization algorithm. It identifies and protects the most important weights, which are determined based on activation magnitude instead of weight magnitude. This approach involves applying optimal per-channel scaling to these salient weights, without relying on backpropagation or reconstruction, ensuring that the LLM does not overfit the calibration set. While it relies on a different paradigm, AWQ is quite close to GPTQ and EXL2 versions, although slightly slower. They are well-supported by inference engines and integrated into TGI, vLLM, and TensorRT-LLM.

An interesting trend is the quantization of models into 1 or 2-bit precision. While some formats, like EXL2, allow for extreme quantization, the quality of the models often suffers significantly. However, recent algorithms like QuIP# (Quantization with Incoherence Processing) and HQQ (Half-Quadratic Quantization) have targeted this regime and offer quantization methods that better preserve the performance of the original models. This is particularly true for large models (over 30B parameters), which can end up taking less space than 7B or 13B parameter models while providing higher quality outputs. This trend is expected to continue, further optimizing these quantization methods.

To conclude this chapter, here is a table summarizing the features of the three main inference engines we covered in the previous sections:

Technique	TGI	vLLM	TensorRT-LLM

Continuous batching	✓	✓	✓
Speculative decoding	✓		
FlashAttention2	✓	✓	✓
PagedAttention	✓	✓	✓
Pipeline Parallelism			✓
Tensor parallelism	✓	✓	✓
GPTQ	✓		✓
EXL2	✓		
AWQ	✓	✓	✓

Table 7.1 – Summary of features for TGI, vLLM, and TensorRT-LLM .

Summary

In summary, inference optimization is a critical aspect of deploying LLMs effectively. This chapter explored various optimization techniques, including optimized generation methods, model parallelism, and weight quantization. Significant speedups can be achieved by leveraging techniques like predicting multiple tokens in parallel with speculative decoding, or using an optimized attention mechanism with FlashAttention-2. Additionally, we discussed how model parallelism methods, including data, pipeline, and tensor parallelism, distribute the computational load across multiple GPUs to increase throughput and reduce latency. Weight quantization, with formats like GGUF and EXL2, further reduces the memory footprint and accelerates inference, with some calculated tradeoff in output quality.

Understanding and applying these optimization strategies are essential for achieving high performance in practical applications of LLMs, such as chatbots and code completion. The choice of techniques and tools depends on specific requirements, including available hardware, desired latency, and throughput. By combining various approaches, such as continuous batching, and speculative decoding, along with advanced attention mechanisms and model parallelism, users can tailor their deployment strategies to maximize efficiency.

In the next chapter, we will cover the RAG ingestion pipeline. We will introduce vector databases and their role in efficiently storing and retrieving high-dimensional embeddings. We will also discuss embedding models, which transform documents into vectors suitable for processing by the RAG pipeline. The chapter will cover the key steps of the RAG feature pipeline, including chunking and embedding documents, ingesting these documents into a vector DB, and applying pre-retrieval optimizations to improve performance. Additionally, we will cover how to set up the necessary infrastructure programmatically using Pulumi and conclude by deploying the RAG ingestion pipeline to AWS.

References

- Hugging Face, Text Generation Inference, https://github.com/huggingface/text-generation-inference, 2022.
- W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C.H. Yu, J.E. Gonzalez, H. Zhang, I. Stoica, Efficient Memory Management for Large Language Model Serving with PagedAttention, 2023.
- Nvidia, TensorRT-LLM, https://github.com/NVIDIA/TensorRT-LLM, 2023.

- Y. Leviathan, M. Kalman, Y. Matias, Fast Inference from Transformers via Speculative Decoding, 2023.
- T. Cai, Y. Li, Z. Geng, H. Peng, J.D. Lee, D. Chen, T. Dao, Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads, 2024.
- W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C.H. Yu, J.E. Gonzalez, H. Zhang, I. Stoica, Efficient Memory Management for Large Language Model Serving with PagedAttention, 2023.
- R.Y. Aminabadi, S. Rajbhandari, M. Zhang, A.A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase, Y. He, DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale, 2022.
- Y. Huang, Y. Cheng, A. Bapna, O. Firat, M.X. Chen, D. Chen, H. Lee, J. Ngiam, Q.V. Le, Y. Wu, Z. Chen, GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, 2019.
- K. James Reed, PiPPy: Pipeline Parallelism for PyTorch, https://github.com/pytorch/PiPPy, 2022.
- M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2020.
- Verma and Vaidya, Mastering LLM Techniques: Inference Optimization, NVIDIA Developer Technical Blog,

https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/, 2023.

- T. Dettmers, M. Lewis, Y. Belkada, L. Zettlemoyer, LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale, 2022.
- G. Gerganov, llama.cpp, https://github.com/ggerganov/llama.cpp, 2023.
- E. Frantar, S. Ashkboos, T. Hoefler, D. Alistarh, GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers, 2023.
- Tuboderp, exllamav2, https://github.com/turboderp/exllamav2, 2023.
- J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, S. Han, AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration, 2024.

8 RAG Feature Pipeline

Retrieval-augmented generation (RAG) is fundamental in most generative AI applications. RAG's core responsibility is to inject custom data into the LLM to perform a given action (e.g., summarize, reformulate, extract, etc., the injected data).

You often want to use the LLM on data it wasn't trained on (e.g., private or new data). As fine-tuning an LLM is a highly costly operation, RAG is a compelling strategy that bypasses the need for constant fine-tuning to access that new data.

First, we will start with a theoretical part that focuses on the fundamentals of RAG and how it works. We will then walk you through all the components of a naïve RAG system: chunking, embedding and vector DBs. Ultimately, we will present various optimizations used for an advanced RAG system.

Secondly, we will continue exploring LLM Twin's RAG feature pipeline architecture. At this step, we will apply all the theoretical aspects we discussed at the beginning of the chapter.

Thirdly, we will go through a practical example by implementing the LLM Twin's RAG feature pipeline based on the system design described above and deploy it to AWS.

The main sections of this chapter are:

- Understanding RAG
- Digging into advanced RAG for pre-retrieval optimizations
- Exploring the LLM Twin's RAG feature pipeline architecture
- Implementing the LLM Twin's RAG feature pipeline
- Setting up the infrastructure using Pulumni
- Deploying to AWS

By the end of this chapter, you will have a clear and comprehensive understanding of what RAG is and how it is applied to our LLM Twin use case.

Understanding RAG

Retrieval-augmented generation (RAG) enhances the accuracy and reliability of generative AI models with information fetched from external sources. It is a technique complementary to the internal knowledge of the LLMs. Before going into the details let's understand what RAG stands for:

- **Retrieval:** search for relevant data
- **Augmented:** add the data as context to the prompt
- **Generation:** use the augmented prompt with an LLM for generation

Any LLM is bound to understand the data it was trained on, sometimes called parameterized knowledge. Thus, even if the LLM can perfectly answer what happened in the past, it won't have access to the newest data or any other external sources on which it wasn't trained.

Let's take the most powerful model from OpenAI as an example, which in the summer of 2024 is GPT-4o. The model is trained on data up to Oct 2023. Thus, if we ask what happened during the 2020 pandemic, it can be answered perfectly due to its parametrized knowledge. However, it will not know the answer if we ask about the 2024 soccer EURO cup results due to its bounded parametrized knowledge. Another scenario is that it will start confidently hallucinating and provide a faulty answer.

RAG overcomes these two limitations of LLMs. It provides access to external or latest data and prevents hallucinations, enhancing generative AI models' accuracy and reliability.

Why use RAG?

We briefly explained the importance of using RAG in generative AI applications above. Now, we will dig deeper into the "why". Next, we will focus on what a naïve RAG framework looks like.

For now, to get an intuition about RAG, you have to know that when using RAG, we inject the necessary information into the prompt to answer the initial user question. After, we pass the augmented prompt to the LLM for the final answer. Now the LLM will use the additional context to answer the user question.

There are two fundamental problems that RAG solves:

- Hallucinations
- Old or private information

Hallucinations

If a chatbot without RAG is asked a question about something it wasn't trained on, there are big changes that will give you a confident answer about something that isn't true. Let's take the 2024 soccer EURO Cup as an example. If the model is trained up to Oct 2023 and we ask something about the tournament, it will most likely come up with some random answer that is hard to differentiate from reality.

Even if the LLM doesn't hallucinate all the time, it raises the concern of trust in its answers. Thus, we have to ask ourselves: "When can we trust the LLM's answers?" and "How can we evaluate if the answers are correct?"

By introducing RAG, we will enforce the LLM to always answer solely based on the introduced context. The LLM will act as the reasoning engine, while the additional information added through RAG will act as the single source of truth for the generated answer.

By doing so, we can quickly evaluate if the LLM's answer is based on the external data or not.

Old information

Any LLM is trained or fine-tuned on a subset of the total world knowledge dataset. This is due to three main issues:

- **Private data:** You cannot train your model on data you don't own or have the right to use.
- **New data**: New data is generated every second. Thus, you would have to constantly train your LLM to keep up.
- **Costs:** Training or fine-tuning an LLM is an extremely costly operation. Hence, it is not feasible to do it on an hourly or daily basis.

RAG solved these issues, as you no longer have to constantly fine-tune your LLM on new data (or even private data). Directly injecting the necessary data to respond to user questions into the prompts that are fed to the LLM is enough to generate correct and valuable answers.

To conclude, RAG is key for a robust and flexible generative AI system. But how do we inject the right data into the prompt based on the user's questions? We will dig into the technical aspects of RAG in the next sections.

The vanilla RAG framework

Every RAG system is similar at its roots. We will first focus on understanding RAG in its simplest form. Later, we will gradually introduce more advanced RAG techniques to improve the system's accuracy.

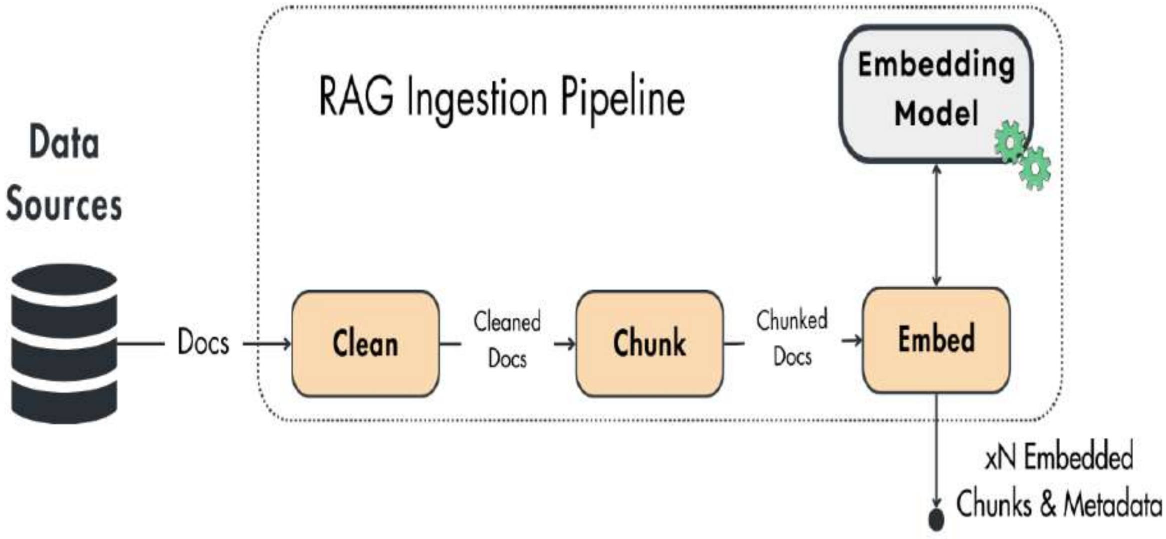
A RAG system is composed of three main modules independent of each other:

- **Ingestion pipeline:** A batch or streaming pipeline used to populate the vector DB.
- **Retrieval pipeline:** A module that queries the vector DB and retrieves relevant entries to the user's input.
- **Generation pipeline:** The layer that uses the retrieved data to augment the prompt and an LLM to generate answers.

As these three components are classes or services of their own, we will dig into each separately.

But how are these three modules connected? Here is a very simplistic overview:

- On the backend side, the ingestion pipeline runs on a schedule or constantly to populate the vector DB with external data.
- On the client side, the user asks a question.
- The question is passed to the retrieval module, which pre-processes the user's input and queries the vector DB.
- The generation pipelines use a prompt template, user input, and retrieved context to create the prompt.
- The prompt is passed to an LLM to generate the answer.
- The answer is shown to the user.



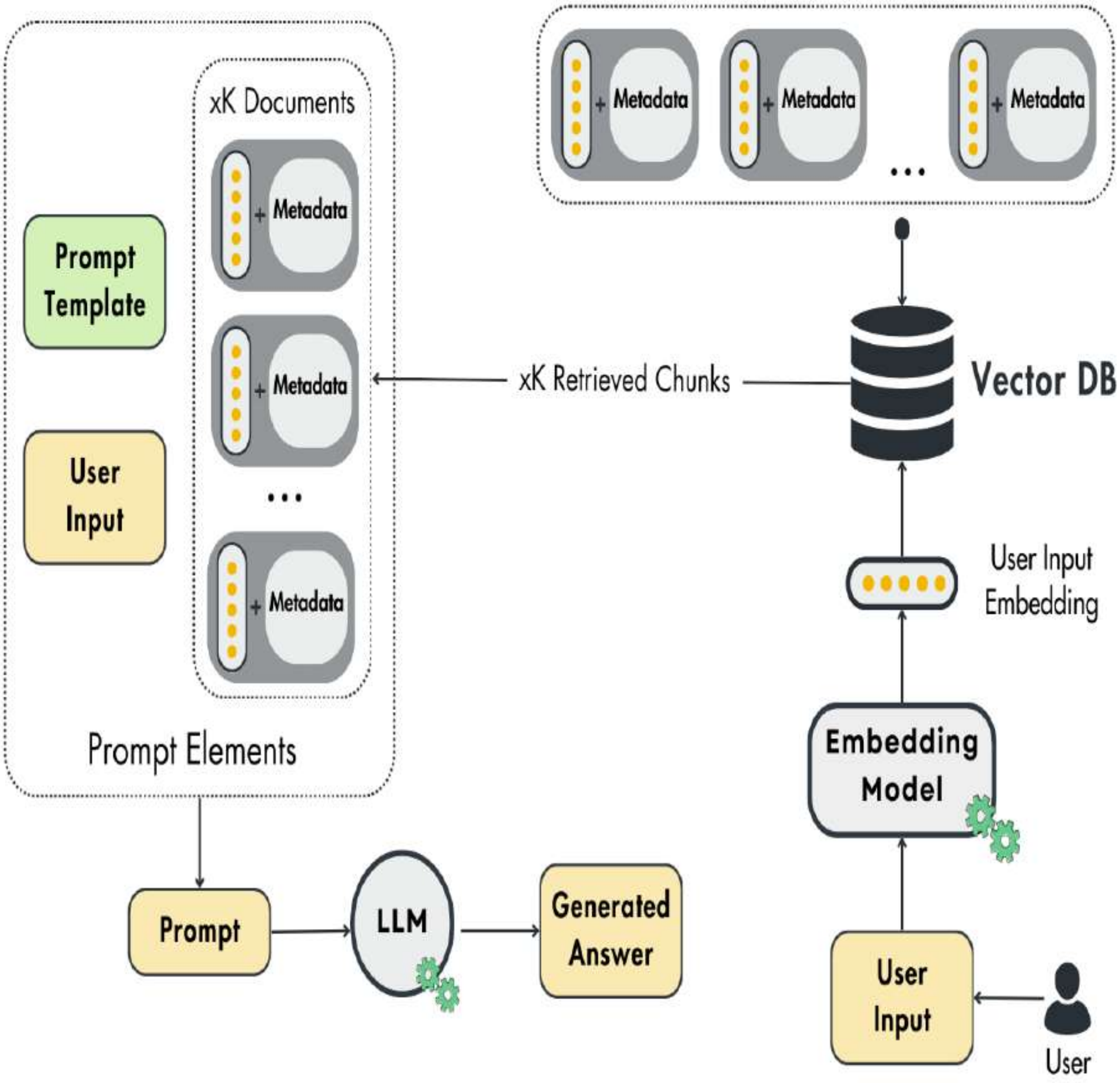


Figure 8.1 – Vanilla RAG architecture

Ingestion pipeline

The RAG ingestion pipeline extracts raw documents from various data sources (e.g., data warehouse, data lake, web pages, etc.). Then, it cleans, chunks and embeds the documents.

Ultimately, it loads the embedded chunks to a vector DB (or other similar vector storage).

Thus, the RAG ingestion pipeline is split again into the following:

- The **data extraction module** gathers all necessary data from various sources such as databases, APIs, or web pages. This module is highly dependent on your data. It can be as easy as querying your data warehouse or something more complex, such as crawling Wikipedia.
- A **cleaning layer** that standardizes and removes unwanted characters from the extracted data.
- The **chunking module** splits the cleaned documents into smaller ones. As we want to pass the document's content to an embedding model, this is necessary to ensure it doesn't exceed the model's input maximum size. Also, chunking is required to separate specific regions that are semantically related. For example, when chunking a book chapter, the most optimal way is to group similar paragraphs into the same chunk. By doing so, at the retrieval time, you will add only the essential data to the prompt.
- The **embedding component** uses an embedding model to take the chunk's content (text, images, audio, etc.) and project it into a dense vector packed with semantic value –

more on embeddings in the Embeddings models section below.

- The **loading module** takes the embedded chunks along with a metadata document. The metadata will contain essential information such as the embedded content, the URL to the source of the chunk, when the content was published on the web, etc. The embedding is used as an index to query similar chunks, while the metadata is used to access the information added to augment the prompt.

At this point, we have an RAG ingestion pipeline that takes raw documents as input, processes them, and populates a vector DB. The next step is to retrieve relevant data from the vector store correctly.

Retrieval pipeline

The retrieval components take the user's input (text, image, audio, etc.), embed it, and query the vector DB for similar vectors to the user's input.

The primary function of the retrieval step is to project the user's input into the same vector space as the embeddings used as an index in the vector DB. This allows us to find the top K's most similar entries by comparing the embeddings from the vector storage with the user's input vector. These entries then serve as content to augment the prompt that is passed to the LLM to generate the answer.

You must use a distance metric to compare two vectors, such as the Euclidean or Manhattan distance. But the most popular one is the cosine distance, which is equal to 1 minus the cosine of the angle between two vectors as follows:

It ranges from -1 to 1, with a value of -1 when vectors A and B are in opposite directions, 0 if they are orthogonal and 1 if they point in the same direction.

Most of the time, the cosine distance works well in non-linear complex vector spaces. However, it is essential to notice that choosing the proper distance between two vectors depends on your data and the embedding model you use.

One critical factor to highlight is that the user's input and embeddings must be in the same vector space. Otherwise, you cannot compute the distance between them. To do so, it is essential to pre-process the user input in the same way you processed the raw documents in the RAG ingestion pipeline. It means you must clean, chunk (if necessary), and embed the user's input using the same functions, models, and hyperparameters. Similar to how you have to pre-process the data into features in the same way between training and inference, otherwise the inference will yield inaccurate results – a phenomenon also known as the training-serving skew.

Generation pipeline

The last step of the RAG system is to take the user's input and retrieve data, pass it to an LLM and generate a valuable answer.

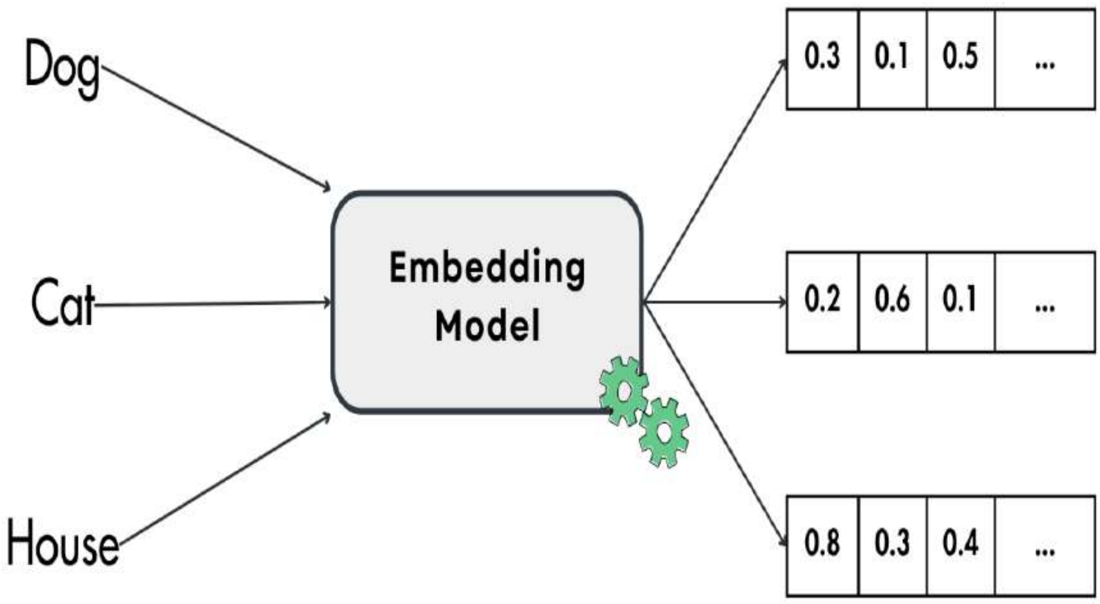
The final prompt results from a prompt template populated with the user's query and retrieved context. You might have a single or multiple prompt templates depending on your application. Usually, all the prompt engineering is done at the prompt template level.

Each prompt template and LLM should be tracked and versioned using MLOps best practices. Thus, you always know that a given answer was generated by a specific version of the LLM and prompt template(s)—more on this in Chapter 9.

What are embeddings

Imagine you're trying to teach a computer to understand the world. Embeddings are like a particular translator that turns these things into a numerical code. This code isn't random, though, because similar words or items end up with codes that are close to each other. It's like a map where words with similar meanings are clustered together.

With that in mind, a more theoretical definition is that embeddings are dense numerical representations of objects encoded as vectors in a continuous vector space, such as words, images, or items in a recommendation system. This transformation helps capture the semantic meaning and relationships between the objects. For instance, in natural language processing (NLP), embeddings translate words into vectors where semantically similar words are positioned closely together in the vector space.



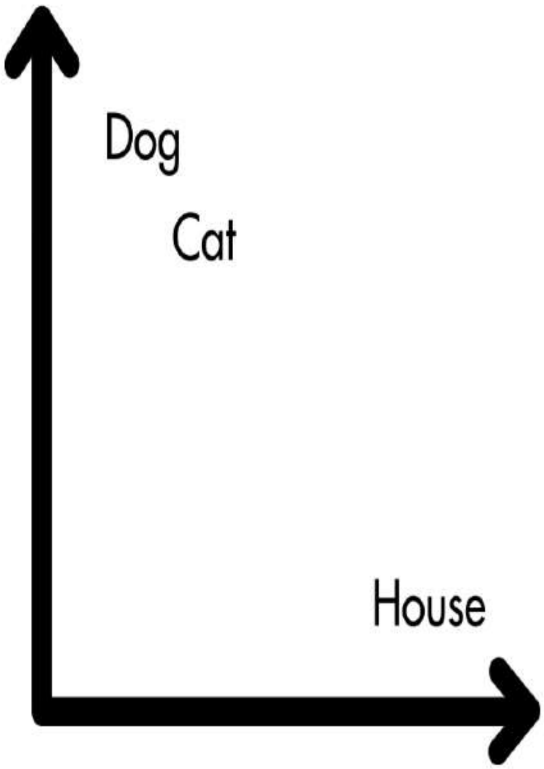


Figure 8.2 – What are embeddings

A popular method is visualizing the embeddings to understand and evaluate their geometrical relationship. As the embeddings often have more than 2 or 3 dimensions, usually between 64 and 2048, you must project them again to 2D or 3D.

For example, you can use UMAP, a dimensionality reduction method well known for keeping the geometrical properties between the points when projecting the embeddings to 2D or

3D. Another popular algorithm for dimensionality reduction when visualizing vectors is t-SNE. However, compared to UMAP, it is more stochastic and doesn't preserve the topological relationships between the points.

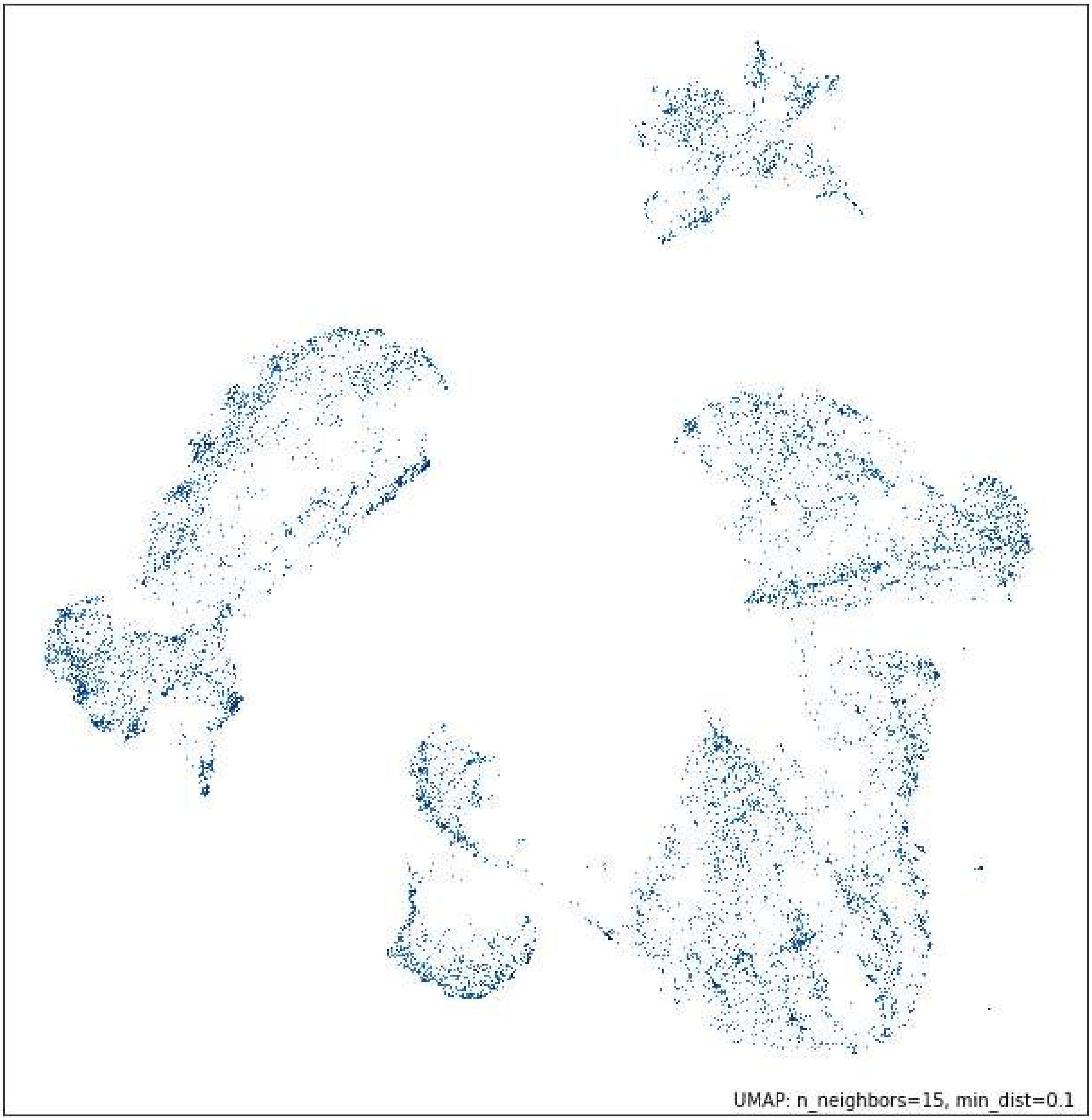


Figure 8.3 - Visualize embeddings using UMAP. Image taken from UMAP's documentation.

Why embeddings are so powerful

First, machine learning models work only with numerical values. This is not a problem when working with tabular data, as the data is often in numerical form or can easily be processed into numbers. Embeddings come in handy when we want to feed words, images or audio data into models.

For instance, when working with transformer models, you tokenize all your text input, where each token has an embedding associated with it. The beauty of this process lies in its simplicity-the input to the transformer is a sequence of embeddings, which can be easily and confidently interpreted by the dense layers of the neural network.

Based on this example, you can use embeddings to encode any categorical variable and feed it to an ML model. But why not use other simple methods, such as one hot encoding? When working with categorical variables with high cardinality, such as language vocabularies, you will suffer from the curse of dimensionality when using other classical methods. For example, if your vocabulary has 10000 tokens, then only one token will have a length of 10000 after applying one hot encoding. If the input sequence has N tokens, that will become in N * 10000 input parameters. If N >= 100, often when inputting text, the input is too large to be usable. Another issue with other classical methods that don't suffer from the curse of dimensionality, such as hashing, is that you lose the semantic relationships between the vectors.

Secondly, embedding your input reduces the size of its dimension and condenses all of its semantic meaning into a dense vector. This is an extremely popular technique when working with images, where a CNN encoder module maps the high-dimensional meaning into an embedding, which is later processed by a CNN decoder that performs the classification or regression steps.

The image below shows a typical CNN layout. Imagine tiny squares within each layer. Those are the "receptive fields." Each square feeds information to a single neuron in the previous layer. As you move through the network, two key things are happening:

Shrinking the Picture: Special "subsampling" operations make the layers smaller, focusing on essential details.

Learning Features: "Convolution" operations, on the other hand, actually increase the layer size as the network learns more complex features from the image.

Finally, a fully connected layer at the end takes all this processed information and transforms it into the final vector embedding, a numerical image representation.

Figure 8.4 – Creating embeddings from an image using a CNN. Image source.

How are embeddings created?

Embeddings are created by deep learning models that understand the context and semantics of your input and project it into a continuous vector space.

Various deep learning models can be used to create embeddings, varying by the data input type. Thus, it is fundamental to understand your data and what you need from it before picking an embedding model.

For example, when working with text data, one of the early methods used to create embeddings for your vocabulary is Word2Vec and GloVe. These are still popular methods used today for simpler applications.

Another popular method is to use encoder-only transformers, such as BERT, or other methods from its family, such as RoBERTa. These models leverage the encoder of the transformer architecture to smartly project your input into a dense vector space that can later be used as embeddings.

To quickly compute the embeddings in Python, you can conveniently leverage the Sentence Transformers Python package (also available in HuggingFace's transformer package). This tool provides a user-friendly interface, making the embedding process straightforward and efficient.

```
from sentence_transformers import SentenceTransformer
# 1. Load a pretrained Sentence Transformer model
model = SentenceTransformer("all-MiniLM-L6-v2")
# The sentences to encode
sentences = [
    "The weather is lovely today.",
    "It's so sunny outside!",
    "He drove to the stadium.",
]
# 2. Calculate embeddings by calling model.encode()
embeddings = model.encode(sentences)
print(embeddings.shape)
# [3, 384]
# 3. Calculate the embedding similarities
similarities = model.similarity(embeddings, embedding
print(similarities)
# tensor([[1.0000, 0.6660, 0.1046],
#         [0.6660, 1.0000, 0.1411],
#         [0.1046, 0.1411, 1.0000]])
```

The example is taken from Sentence Transformer's quick start documentation.

The best-performing embedding model can change with time and your specific use case. You can find particular models on the Massive Text Embedding Benchmark (MTEB) on HuggingFace. Depending on your needs, you can consider the best-performing model, the one with the best accuracy or the one with the smallest memory footprint. This decision is solely based on your requirements (e.g., accuracy, hardware, and so on). However, HuggingFace and SentenceTransformer make switching between different models straightforward. Thus, you can always experiment with various options.

When working with images, you can embed them using convolutional neural networks (CNN). Popular CNN networks are based on the ResNet architecture. However, we can't directly use image embedding techniques for audio recordings. Instead, we can create a visual representation of the audio, such as a spectrogram, and then apply image embedding models to those visuals. This allows us to capture the essence of images and sounds in a way computers can understand.

By leveraging models like CLIP, you can practically embed a piece of text and an image in the same vector space. This allows you to find similar images using a sentence as input, or the other way around, demonstrating the practicality of CLIP.

```
from sentence_transformers import SentenceTransformer
from PIL import Image
# Load CLIP model
model = SentenceTransformer("clip-ViT-B-32")
# Encode an image:
img_emb = model.encode(Image.open("two_dogs_in_snow.j
# Encode text descriptions
text_emb = model.encode(
    ["Two dogs in the snow", "A cat on a table", "A p
)
# Compute similarities
similarity_scores = model.similarity(img_emb, text_em
print(similarity_scores)
```

The example is taken from Sentence Transformer's image search documentation.

Here, we provided a small introduction to how embeddings can be computed. The realm of specific implementations is vast, but what is important to know is that embeddings can be computed for most digital data categories, such as words, sentences, documents, images, videos, graphs, etc.

It's crucial to grasp that you must use specialized models when you need to compute the distance between two different data categories, such as the distance between the vector of a sentence and of an image. These models are designed to project both data types into the same vector space, such as CLIP, ensuring accurate distance computation.

Applications of embeddings

Due to the generative AI revolution, which uses RAG, embeddings become extremely popular in information retrieval tasks, such as semantic search for text, code, images, and audio and long-term memory of agents.

But before Gen AI, embeddings were already heavily used in:

- Representing categorical variables (e.g., vocabulary tokens) that are fed to an ML model.
- Recommender system by encoding the users and items and finding their relationship.

- Clustering and outlier detection.
- Data visualization by using algorithms such as UMAP.
- Classification by using the embeddings as features.
- Zero-shot classification by comparing the embedding of each class and picking the most similar one.

The last step to fully understanding how RAG works is to examine vector DBs and how they leverage embeddings to retrieve data.

More on vector DBs

Vector databases are specialized databases designed to efficiently store, index, and retrieve vector embeddings. Traditional scalar-based databases struggle with the complexity of vector data, making vector databases crucial for tasks like real-time semantic search.

The difference between a vector index and a database

While standalone vector indices like FAISS are effective for similarity search, they lack vector databases' comprehensive data management capabilities. Vector databases support CRUD operations, metadata filtering, scalability, real-time updates, backups, ecosystem integration, and robust data security, making them more suited for production environments than standalone indices.

How does a vector database work?

Think of how you usually search a database. You type in something specific, and the system spits out the exact match. That's how traditional databases work. Vector databases are different. Instead of perfect matches, we look for the closest neighbors of the query vector. Under the hood, a vector database uses Approximate Nearest Neighbor (ANN) algorithms to find these close neighbors.

Approximate Nearest Neighbor algorithms don't return the top matches for a given search, but standard Nearest Neighbor algorithms are too slow to work in practice. Also, it is shown empirically that using only approximations of the top matches for a given input query works well enough. Thus, the trade-off between accuracy and latency ultimately favors ANN algorithms.

This is a typical workflow of a vector database:

- **Indexing Vectors:** Vectors are indexed using data structures optimized for high-dimensional data. Common indexing techniques include hierarchical navigable small world

(HNSW), random projection, product quantization (PQ), and locality-sensitive hashing (LSH).

- **Querying for Similarity**: During a search, the database queries the indexed vectors to find those most similar to the input vector. This process involves comparing vectors based on similarity measures such as cosine similarity, Euclidean distance, or dot product. Each has unique advantages and is suitable for different use cases.
- **Post-processing Results**: After identifying potential matches, the results undergo post-processing to refine accuracy. This step ensures that the most relevant vectors are returned to the user.

Vector databases can filter results based on metadata before or after the vector search. Both approaches have trade-offs in terms of performance and accuracy. The query also depends on the metadata (along with the vector index), so it contains a metadata index user for filtering operations.

Algorithms for creating the vector index

Vector DBs use various algorithms to create the vector index and manage searching data efficiently:

- **Random Projection:** Random projection reduces the dimensionality of vectors by projecting them into a lower-dimensional space using a random matrix. This technique preserves the relative distances between vectors, facilitating faster searches.
- **Product Quantization (PQ**): Product quantization compresses vectors by dividing them into smaller sub-vectors and then quantizing these sub-vectors into representative codes. This reduces memory usage and speeds up similarity searches.
- **Locality-sensitive Hashing (LSH):** Locality-sensitive hashing maps similar vectors into buckets. This method enables fast approximate nearest neighbor searches by focusing on a subset of the data, reducing the computational complexity.
- **Hierarchical Navigable Small World (HNSW):** HNSW constructs a multi-layer graph where each node represents a set of vectors. Similar nodes are connected, allowing the algorithm to navigate the graph and find the nearest neighbors efficiently.

These algorithms enable vector databases to efficiently handle complex and large-scale data, making them a perfect fit for a variety of AI and machine learning applications.

Database operations

Vector databases also share common characteristics with standard databases to ensure high performance, fault tolerance, and ease of management in production environments. Key operations include:

- **Sharding and Replication:** Data is partitioned (sharded) across multiple nodes to ensure scalability and high availability. Data replication across nodes helps maintain data integrity and availability in case of node failures.
- **Monitoring:** Continuous monitoring of database performance, including query latency and resource usage (RAM, CPU, disk), helps maintain optimal operations and identify potential issues before they impact the system.
- **Access Control:** Implementing robust access control mechanisms ensures that only authorized users can access and modify data. This includes role-based access controls and other security protocols to protect sensitive information.
- **Backups:** Regular database backups are critical for disaster recovery. Automated backup processes ensure that data can be restored to a previous state in case of corruption or loss.

An overview of advanced RAG

The vanilla RAG framework presented in the first section of this chapter doesn't address many fundamental aspects that impact the quality of the retrieval and answer generation, such as:

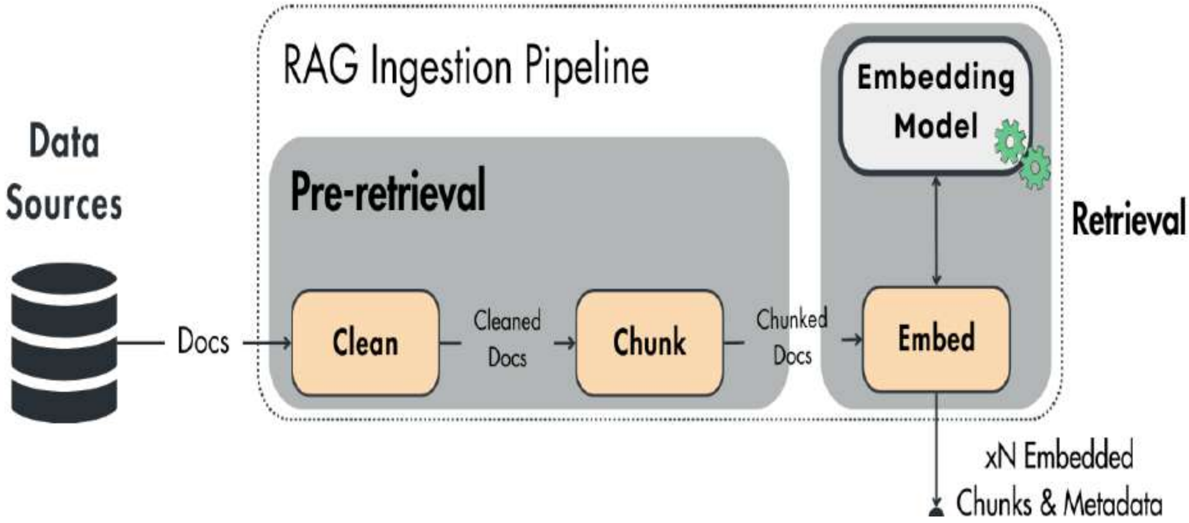
- Are the retrieved documents relevant to the user's question?
- Is the retrieved context enough to answer the user's question?
- Is there any redundant information that only adds noise to the augmented prompt?
- Does the latency of the retrieval step match our requirements?
- What do we do if we can't generate a valid answer using the retrieved information?

From the questions above, we can draw two conclusions. The first one is that we need a robust evaluation module for our RAG system that can quantify and measure the quality of the retrieved data and generate answers relative to the user's question. We will discuss this topic in more detail in Chapter 9. The second conclusion is that we must improve our RAG framework to address the retrieval limitations directly in the algorithm. These improvements are known as advanced RAG.

The vanilla RAG design can be optimized at three different stages:

- **Pre-retrieval:** focuses on how to structure and preprocess your data for data indexing optimizations as well as query optimizations
- **Retrieval:** revolves around improving the embedding models and metadata filtering to improve the vector search step
- **Post-retrieval:** mainly targets different ways to filter out noise from the retrieved documents and compress the prompt before feeding it to an LLM for answer generation

This section is not meant to be an exhaustive list of all the advanced RAG methods available. The goal is to build an intuition about what can be optimized. We will use only examples based on text data, but the principles of advanced RAG remain the same regardless of the data category. Now, let's zoom in on all three components.



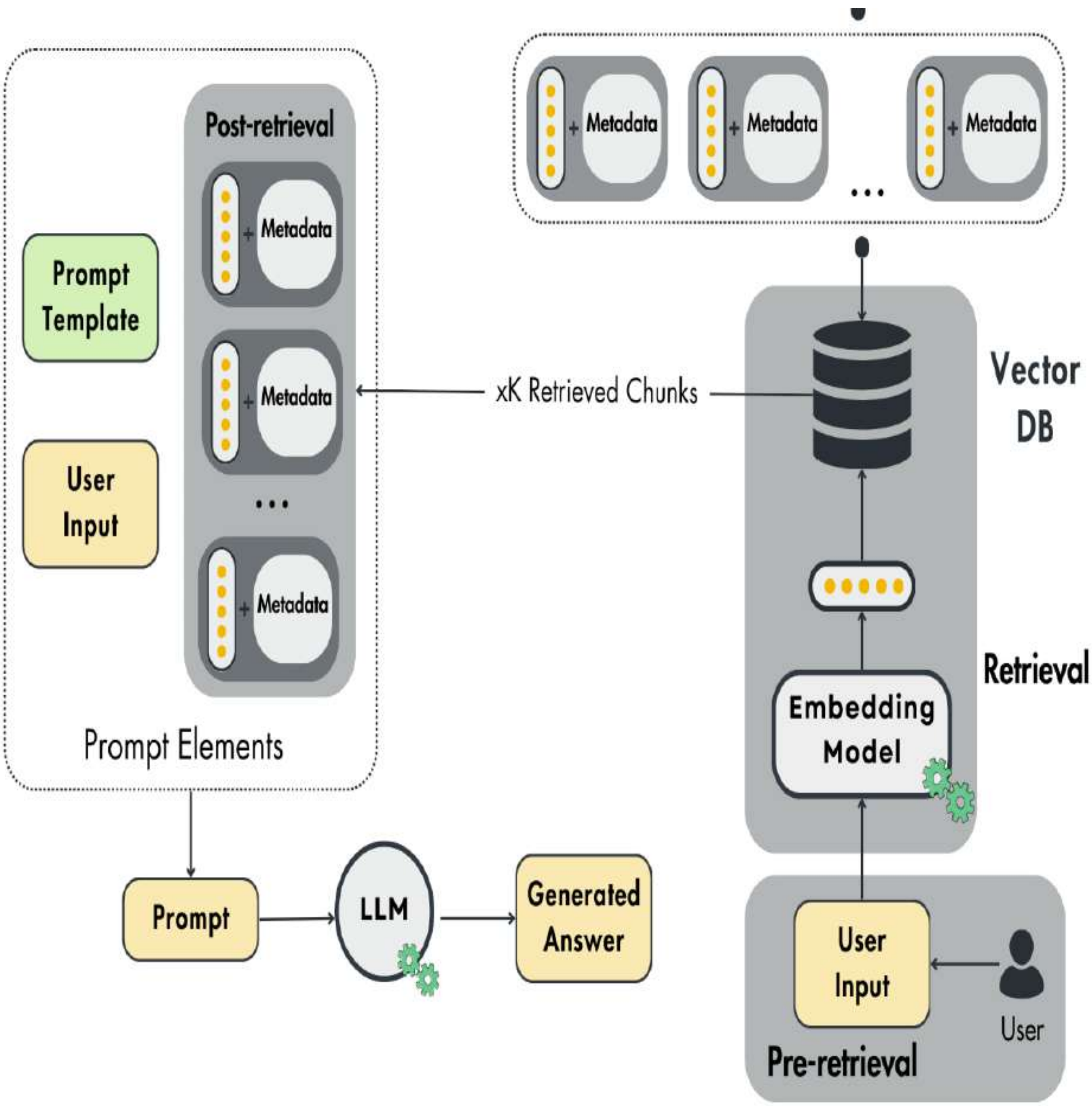


Figure 8.5 – The three stages of advanced RAG.

Pre-retrieval

The pre-retrieval steps are performed in two different ways:

- **Data indexing:** part of the RAG ingestion pipeline. It is mainly implemented within the cleaning or chunking modules to preprocess the data for better indexing.
- **Query optimization:** the algorithm is performed directly on the user's query before embedding it and retrieving the chunks from the vector DB.

As we index our data using embeddings that semantically represent the content of a chunked document, most of the **data indexing** techniques focus on better preprocessing and structuring the data to improve retrieval efficiency, such as:

- **Sliding Window:** Introduces overlap between data chunks, making searches smoother.
- **Enhancing Data Granularity:** Involves data cleaning techniques like removing irrelevant details, verifying factual accuracy, and updating outdated information. A clean and accurate dataset allows for sharper retrieval.
- **Metadata:** Adding metadata tags like dates, URLs, external IDs, or chapter markers helps filter results efficiently during retrieval.
- **Optimizing Index Structures:** It is based on different data index methods, such as various chunk sizes and multi-indexing strategies.

- **Small-to-big**: The algorithm decouples the chunks used for retrieval and the context used in the prompt for the final answer generation. The algorithm uses a small sequence of text to compute the embedding while preserving the sequence itself and a wider window around it in the metadata. Thus, using smaller chunks enhances the retrieval's accuracy, while the larger context adds more contextual information to the LLM. The intuition behind this is that if we use the whole text for computing the embedding, we might introduce too much noise, or the text could contain multiple topics, which results in a poor overall semantic representation of the embedding.

On the **query optimization** side, we can leverage techniques such as query routing, query rewriting and query expansion to refine the retrieved information for the LLM further:

- **Query routing:** Imagine a librarian directing you to the right section of the library. Query routing acts similarly, analyzing the user's question and identifying the most suitable subset of data sources within your indexed information. This ensures the LLM focuses its retrieval efforts on the most relevant data, saving time and computational resources.
- **Query rewriting:** Sometimes, the user's initial query might not perfectly align with the way your data is structured.

  Query rewriting tackles this by reformulating the question to match the indexed information better. This can involve techniques like:

  - **Paraphrasing:** Rephrasing the user's query while preserving its meaning (e.g., "What are the causes of climate change?" could be rewritten as "Factors contributing to global warming").
  - **Synonym substitution:** Replacing less common words with synonyms to broaden the search scope (e.g., " joyful" could be rewritten as "happy").
  - **Sub-queries:** For longer queries, we can break them down into multiple, shorter, and more focused sub-queries. This can help the retrieval stage identify relevant documents more precisely.

- **Hypothetical document embeddings (HyDE):** This technique involves having an LLM create a hypothetical response to the query. Then, both the original query and the LLM's response are fed into the retrieval stage.
- **Query expansion:** This approach aims to enrich the user's question by adding additional terms or concepts, resulting in different perspectives of the same initial question. For example, when searching for "disease," you can leverage synonyms and related terms associated with the original query words and also include "illnesses" or "ailments."

- **Self-query:** The core idea is to map unstructured queries into structured ones. An LLM identifies key entities, events, and relationships within the input text. These identities are used as filtering parameters to reduce the vector search space (e.g., identify cities within the query, for example, "Paris," and add it to your filter to reduce your vector search space).

Retrieval

The retrieval step can be optimized in two fundamental ways:

- **Improving the embedding models** used in the RAG ingestion pipeline to encode the chunked documents and, at inference time, transform the user's input.
- **Leveraging the database's filter and search features.** This step will be used solely at inference time when you have to retrieve the most similar chunks based on user input.

Both strategies are aligned with our ultimate goal: to enhance the vector search step by leveraging the semantic similarity between the query and the indexed data.

When improving the embedding models, you usually have to fine-tune the pre-trained embedding models to tailor them to specific jargon and nuances of your domain, especially for areas with evolving terminology or rare terms.

Instead of fine-tuning the embedding model, you can leverage Instructor models to guide the embedding generation process with an instruction/prompt aimed at your domain. Tailoring your embedding network to your data using such a model can be a good option, as fine-tuning a model consumes more computing and human resources.

```
from InstructorEmbedding import INSTRUCTOR
model = INSTRUCTOR('hkunlp/instructor-xl')
sentence = "3D ActionSLAM: wearable person tracking i
instruction = "Represent the Science title:"
embeddings = model.encode([[instruction,sentence]])
```

The example is taken from Instruct's HuggingFace documentation.

On the other side of the spectrum, here is how you can improve your retrieval by leveraging classic filter and search database features:

Hybrid search: It is a vector and keyword-based search blend. Keyword-based search excels at identifying documents containing specific keywords. When your task demands pinpoint accuracy, and the retrieved information must include exact keyword matches, hybrid search shines. Vector search, while powerful, can sometimes struggle with finding exact matches, but it excels at finding more general semantic similarities. You leverage both keyword matching and semantic similarities by combining the two methods. You have a parameter, usually called alpha, that controls the weight between the two methods. The algorithm has two independent searches, which are later normalized and unified.

Filtered vector search: This type of search leverages the metadata index to filter for specific keywords within the metadata. It differs from a hybrid search in that you retrieve the data once using only the vector index and perform the filtering step before or after the vector search to reduce your search space.

In practice, on the retrieval side, you usually start with filtered vector search or hybrid search, as they are fairly quick to implement. This approach gives you the flexibility to adjust your strategy based on performance. If the results are not as expected, you can always fine-tune your embedding model.

Post-retrieval

The post-retrieval optimizations are solely performed on the retrieved data to ensure that the LLM's performance is not compromised by issues such as limited context windows or noisy data. This is because the retrieved context can sometimes be too large or contain irrelevant information, both of which can distract the LLM.

Two popular methods performed at the post-retrieval step are:

- **Prompt compression:** eliminate unnecessary details while keeping the essence of the data.
- **Re-ranking:** Use a cross-encoder ML model to give a matching score between the user's input and every retrieved chunk. The retrieved items are sorted based on this score. Only the top N results are kept as the most relevant. As you can see in Figure 8.6, this works because the re-ranking model can find more complex relationships between the user input and some content than a simple similarity search. However, we can't apply this model at the initial retrieval step because it is costly. That is why a popular strategy is to retrieve the data using a similarity distance between the embeddings and refine the retrieved information using a re-raking model, as illustrated in Figure 8.7.

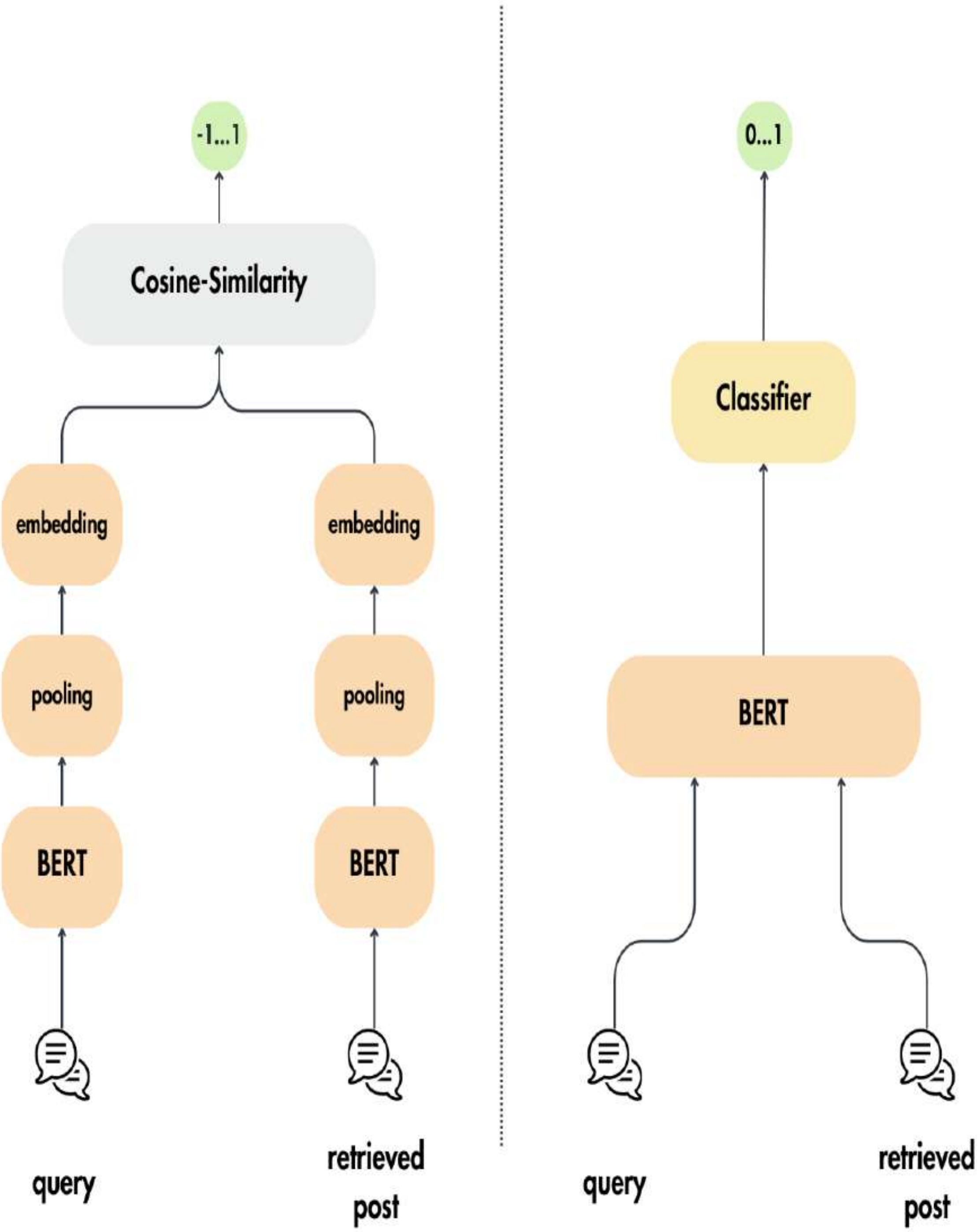
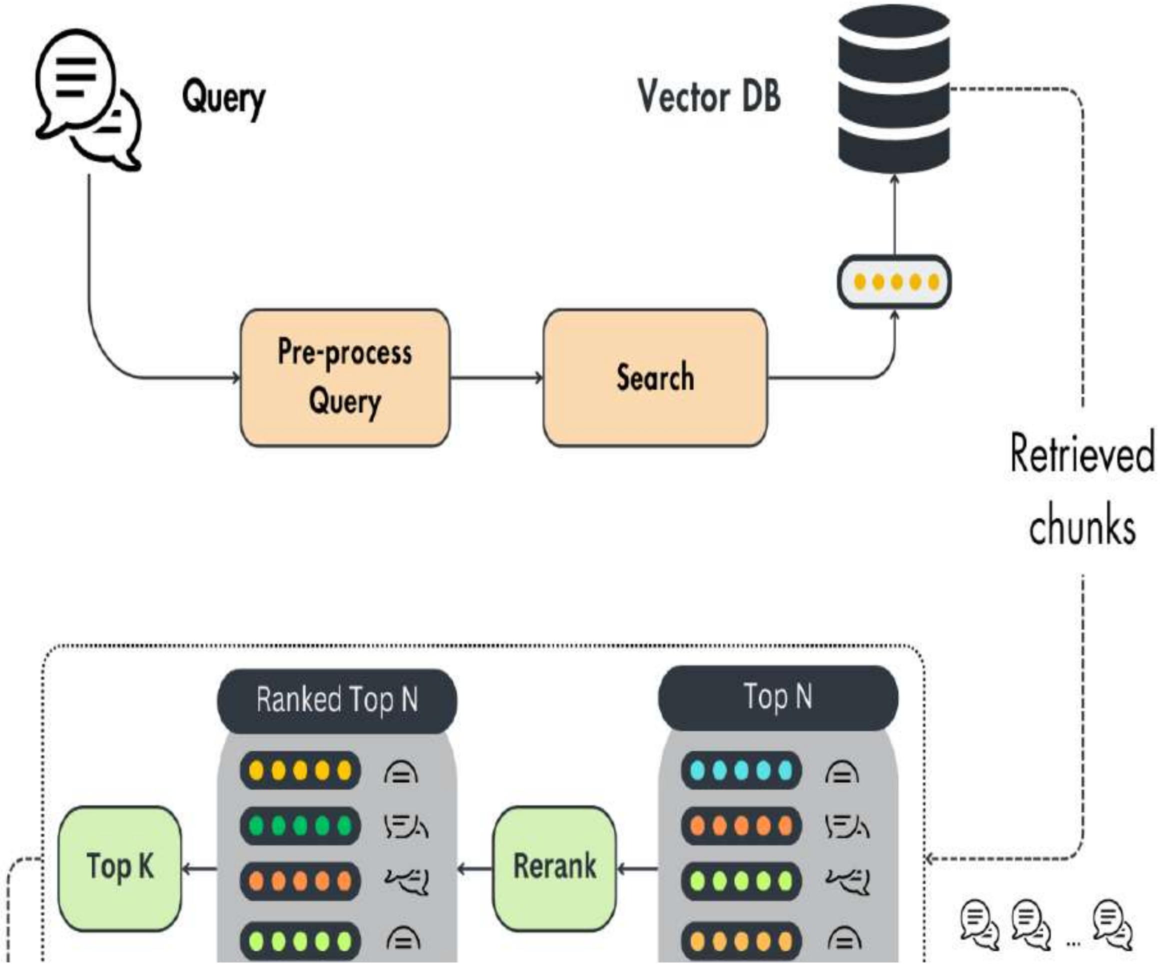


Figure 8.6 – Bi-Encoder (the standard embedding model) vs. Cross-Encoder.

The abovementioned techniques are far from an exhaustive list of all potential solutions. We used them as examples to get an intuition on what you can (and should) optimize at each step in your RAG workflow. The truth is that these techniques can vary tremendously by the type of data you work with. For example, if we work with multi-modal data such as text and images, most of the techniques from above won't work as they are designed for text only.



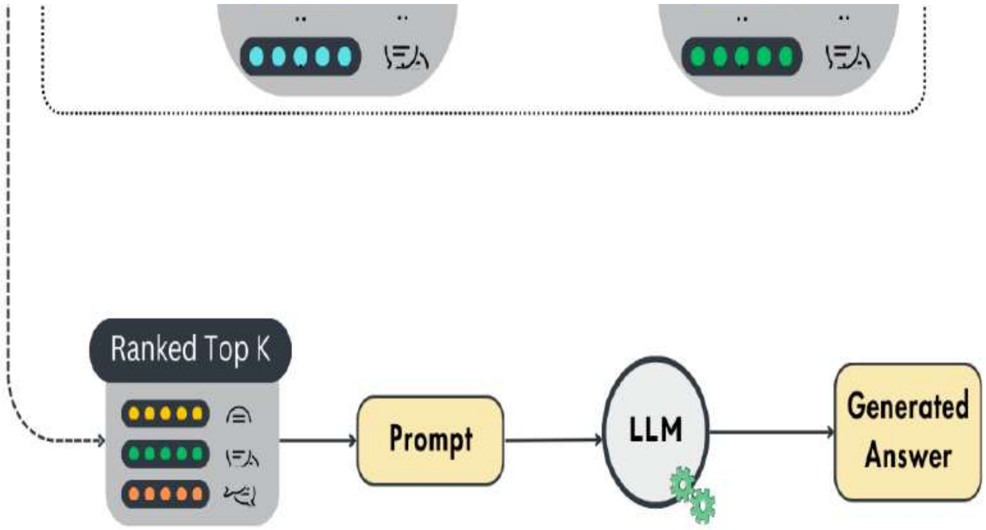


Figure 8.7 – The re-ranking algorithm.

To summarize, the primary goal of these optimizations is to enhance the RAG algorithm at three key stages: pre-retrieval, retrieval, and post-retrieval. This involves preprocessing data for improved vector indexing, adjusting user queries for more accurate searches, enhancing the embedding model, utilizing classic filtering database operations, and removing noisy data. By keeping these goals in mind, you can effectively optimize your RAG workflow for data processing and retrieval.

Exploring the LLM Twin's RAG feature pipeline architecture

Now that you have a strong intuition and understanding of RAG and its workings, we will continue exploring our particular LLM Twin use case. The goal is to provide a hands-on end-to-end example to solidify the theory presented in this chapter.

Any RAG system is split into two independent components:

- The **ingestion pipeline**, which takes in raw data, cleans, chunks, embeds and loads it into a vector database.
- The **inference pipeline**, which queries the vector database for relevant context and ultimately generates an answer by levering an LLM.

In this chapter, we will focus on implementing the RAG ingestion pipeline, and in Chapter 9, we will continue developing the inference pipeline.

With that in mind, let's have a quick refresher on the problem we are trying to solve and where we get our raw data. Remember that we are building an end-to-end ML system. Thus, all the components talk to each other through an interface (or a contract), and each pipeline has a single responsibility. In our case, we ingest raw documents, preprocess them and load them to a vector database.

The problem we are solving

As presented in the previous chapter, this book aims to show you how to build a production-ready LLM Twin backed by an end-to-end ML system.

In this chapter, we want to design a RAG feature pipeline that takes raw social media data (e.g., articles, code repositories, posts) from our MongoDB data warehouse. The text of the raw documents will be cleaned, chunked, embedded, and ultimately loaded to a feature store. As discussed in Chapter 1, we will implement a logical feature store using ZenML artifacts and a Qdrant vector database.

As we want to build a fully automated feature pipeline, we want to sync the data warehouse and logical feature store. Remember that at inference time, the context used to generate the answer is retrieved from the vector database. Thus, the speed of synchronization between the data warehouse and the feature store will directly impact the accuracy of our RAG algorithm.

Another key consideration is how to automate the feature pipeline and integrate it with the rest of our ML system. Our goal is to minimize any desynchronization between the two data storages, as this could potentially compromise the integrity of our system.

To conclude, we must design a feature pipeline that constantly syncs the data warehouse and logical feature store while processing the data accordingly.

The Feature Store

The **feature store** will be the **central access point** for all the features used within the training and inference pipelines.

The training pipeline will use the cleaned data from the feature store (stored as artifacts) to fine-tune LLMs. The inference pipeline will query the vector DB for chunked documents for RAG.

That is why we are designing a feature pipeline and not only a RAG ingestion pipeline. In practice, the feature pipeline contains multiple subcomponents, one of which is the RAG logic.

Remember that the feature pipeline is mainly used as a mind map to navigate the complexity of ML systems. It clearly states that it takes as input raw data and outputs features and optional labels, which are stored in the feature store. Thus, a good intuition is to consider that all the logic between the data warehouse and the feature store goes into the feature pipeline namespace, consisting of one or more sub-pipelines. For example, we will implement another pipeline that takes in cleaned data, processes it into instruct datasets and stores them into artifacts, which also sits under the feature pipeline umbrella as the artifacts are part of the logical feature store. Another example would be implementing a data validation pipeline on top of the raw data or computed features.

Another important observation to make is that text data, stored as strings, are not considered features if you follow the standard conventions. A feature is something that is fed directly into the model. For example, we would have to tokenize the instruct datasets or chunked documents to be considered as features. *Why?* Because the tokens are fed directly to the model and not the sentences as strings. Unfortunately, this makes the system more complex and unflexible. Thus, we will do the tokenization at runtime. But this observation is important to understand as it's a clear example that you don't have to be too rigid about the FTI architecture. You have to take it and adapt it to your own use case.

Where does the raw data come from?

As a quick reminder, all the raw documents are stored in a MongoDB data warehouse. The data warehouse is populated by the data collection ETL pipeline presented in Chapter 3. The ETL pipeline crawls various platforms such as Medium and

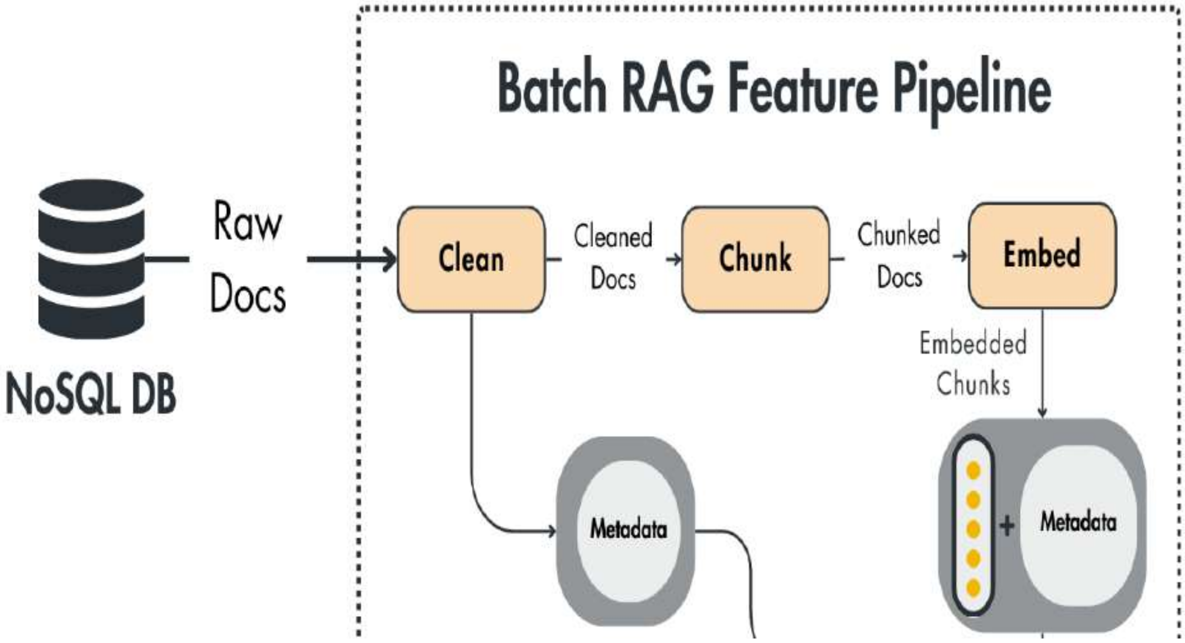
Substack, standardizes the data, and loads it into MongoDB. Check out Chapter 3 for more details on this topic.

Designing the architecture of the RAG feature pipeline

The last step is to architect and go through the design of the RAG feature pipeline of the LLM Twin application.

We will use a batch design scheduled to poll data from the MongoDB data warehouse, process it, and load it to a Qdrant vector DB. The first question to ask ourselves is, "Why a batch pipeline?"

Before answering that, let's quickly understand how a batch architecture works and behaves relative to a streaming design.



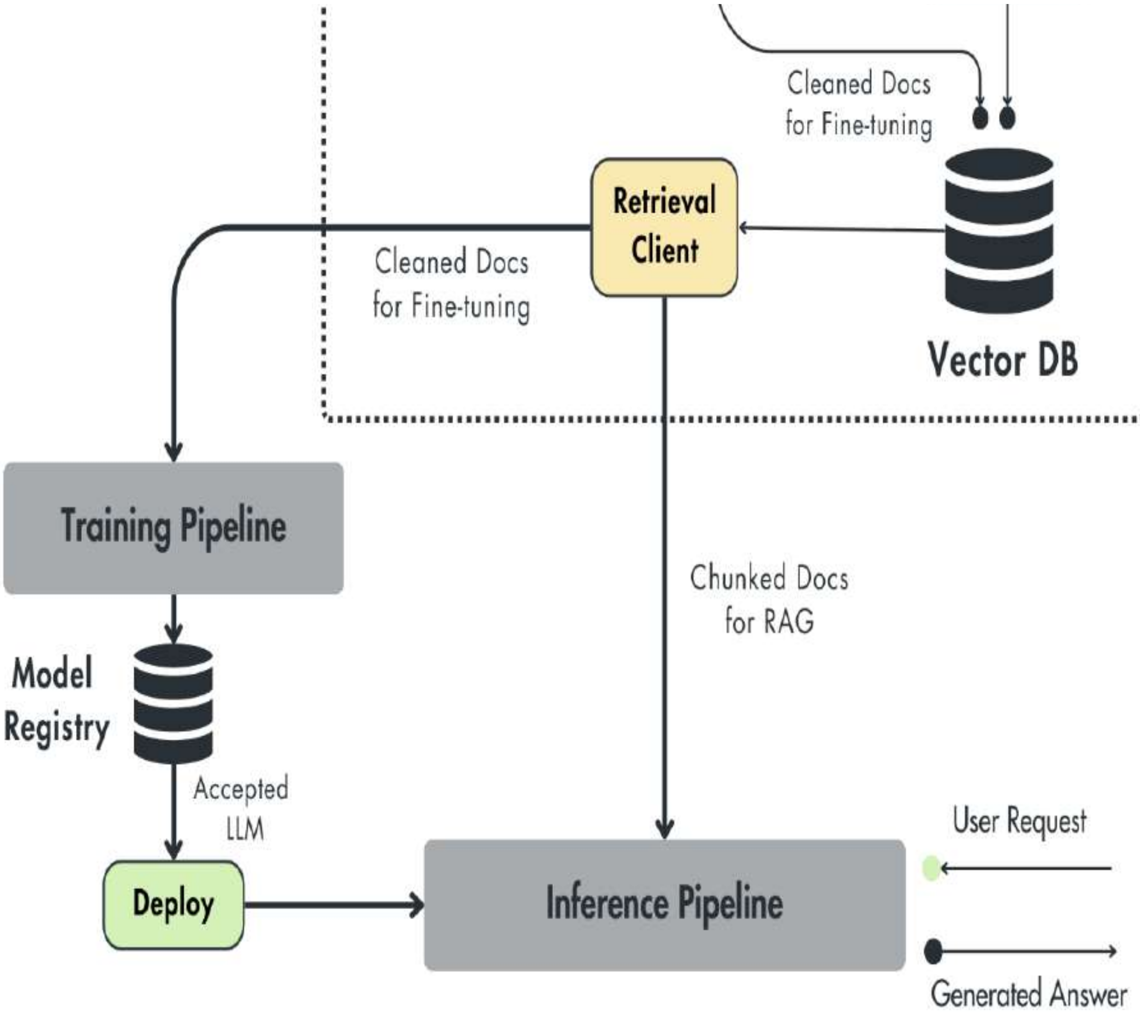


Figure 8.8 – The architecture of the LLM Twin's RAG feature pipeline.

Batch pipelines

A batch pipeline in data systems refers to a data processing method where data is collected, processed, and stored in predefined intervals and larger volumes, also known as "batches". This approach differs from real-time or streaming data processing, where data is processed continuously as it arrives. In a batch pipeline:

- **Data collection**: Data is collected from various sources and stored until sufficient amounts are accumulated for processing. This can include data from databases, logs, files, and other sources.
- **Scheduled processing**: Data processing is scheduled at regular intervals, for example, hourly or daily. During this time, the collected data is processed in bulk. This can involve data cleansing, transformation, aggregation, and other operations.
- **Data loading**: After processing, the data is loaded into the target system, such as a database, data warehouse, data lake or feature store. This processed data is then available for analysis, querying, or further processing.
- Batch pipelines are particularly useful when dealing with large volumes of data that do not require immediate processing. They offer several advantages, including:
- **Efficiency:** Batch processing can handle large volumes of data more efficiently than real-time processing, allowing for optimized resource allocation and parallel processing.
- **Complex processing:** Batch pipelines can perform complex data transformations and aggregations that might be too resource-intensive for real-time processing.

- **Simplicity:** Batch processing systems' architectures are often simpler than those of real-time systems, making them easier to implement and maintain.

Batch vs. streaming pipelines

When implementing feature pipelines, you have two main design choices: batch and streaming.

You can easily write a dedicated chapter explaining how streaming pipelines work. That is already a strong signal that streaming pipelines are more complex than batch ones. But we want to make a parenthesis comparing batch and streaming architectures to provide you with a quick intuition on why a batch pipeline is more than enough for our use case.

Aspect	Batch Pipeline	Streaming Pipeline
Processing Schedule	Processes data at regular intervals (e.g., every minute, hourly, daily).	Processes data continuously, with minimal latency.

Efficiency	Handles large volumes of data more efficiently, optimizing resource allocation and parallel processing.	Handles single data points, providing immediate insights and updates, allowing for rapid response to changes.
Processing Complexity	Capable of performing complex data transformations and aggregations.	Designed to handle high-velocity data streams with low latency.
Use Cases	Suitable for scenarios where immediate data processing is not critical. Commonly used in data warehousing,	Ideal for applications requiring real-time analytics, features, monitoring, and event-driven architectures.

	reporting, ETL processes and feature pipelines.	
System Complexity	Compared to streaming pipelines, systems are generally simpler to implement and maintain.	It is more complex to implement and maintain due to the need for low-latency processing, fault tolerance and scalability. The tooling is also more advanced and complicated.

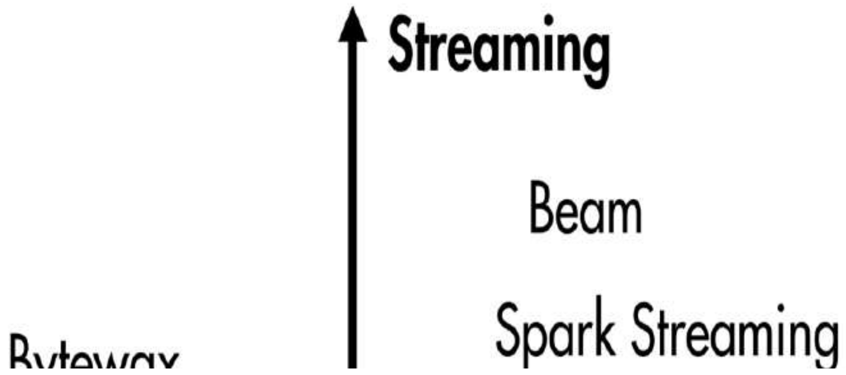
Table 8.1 – Batch vs. streaming pipelines

To conclude, we have used a batch architecture (and not a streaming one) to implement the LLM Twin's feature pipeline for the following reasons:

- **Does not require immediate data processing:** Even if syncing the data warehouse and feature store is critical for an accurate RAG system, a delay of a few minutes is acceptable. Thus, we can schedule the batch pipeline to run every minute, constantly syncing the two data storages. This technique works because the data volume is small. The whole data warehouse will have only thousands of records, not millions or billions. Hence, we can quickly iterate through them and sync the two databases.

- **Simplicity:** As stated above, implementing a streaming pipeline is x2 more complex. In the real world, you want to keep your system as simple as possible, making it easier to understand, debug and maintain. Also, simplicity usually translates to lower infrastructure and development costs.

In Figure 8.9, we compare what tools you can use based on your architecture (streaming vs. batch) and the quantity of data you have to process (small vs. big data). In our use case, we are in the smaller data and batch quadrant, where we picked a combination of vanilla Python and Gen AI tools such as LangChain, Sentence Transformers and Unstructured.



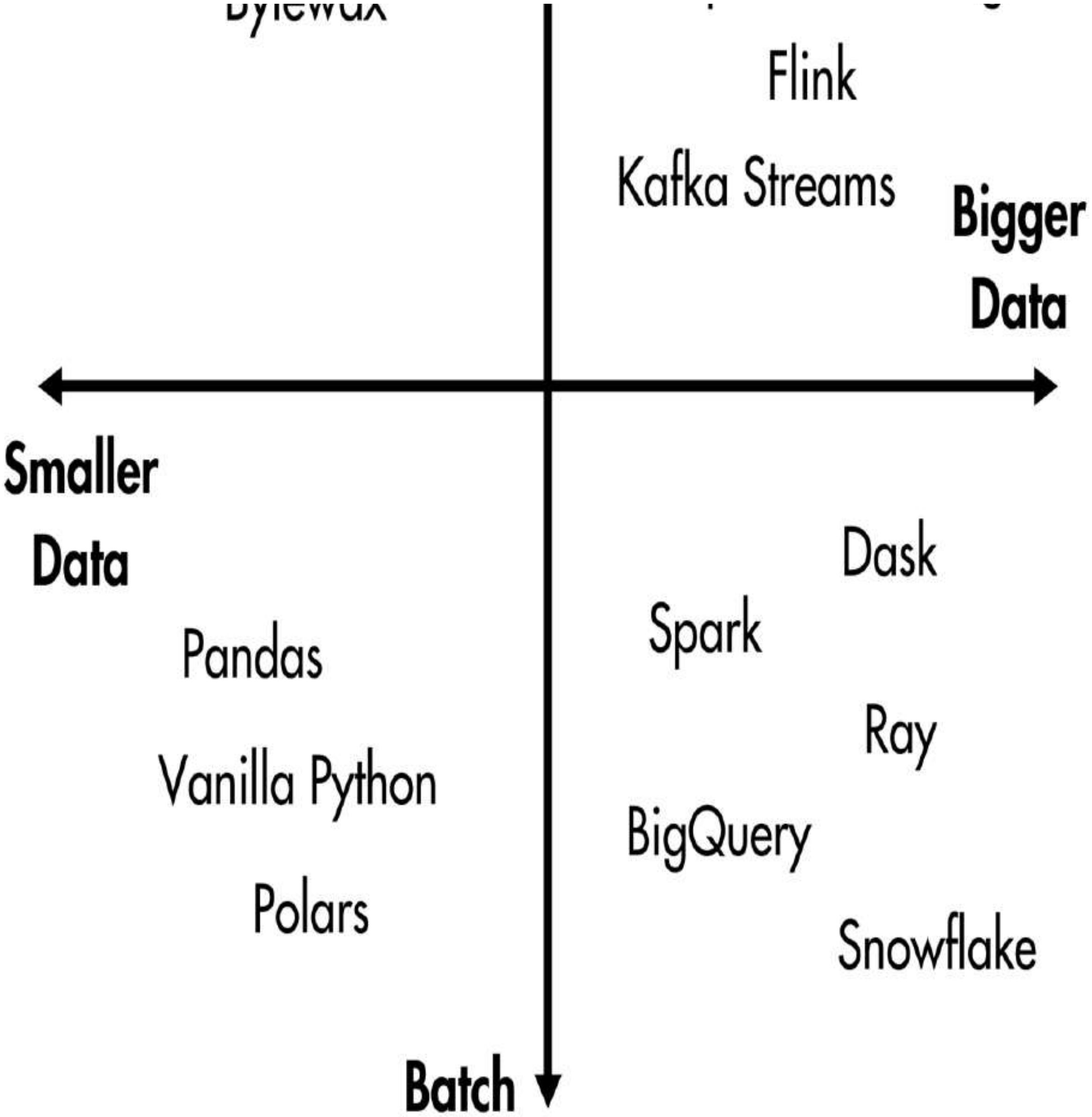


Figure 8.9 – Tools on streaming vs. batch and smaller vs. bigger data spectrum.

In the *"Change data capture (CDC): syncing the data warehouse and feature store"* section later in this chapter, we will discuss when switching from a batch architecture to a streaming one makes sense.

Core steps

Most of the RAG feature pipelines are composed of five core steps. The one implemented in the LLM Twin architecture makes no exception. Thus, you can quickly adapt this pattern for other RAG applications, but here is what the LLM Twin's RAG feature pipeline looks like:

- **Data extraction:** Extract the latest articles, code repositories, and posts from the MongoDB data warehouse. At the extraction step, you usually aggregate all the data you need for processing.
- **Cleaning:** The data from the data warehouse is standardized and partially clean, but we have to ensure that the text contains only useful information, is not duplicated and can be interpreted by the embedding model. For example, we must clean and normalize all non-ASCII characters before passing the text to the embedding model. Also, to keep the information semantically dense, we decided to replace all the URLs with placeholders and remove all emojis. The cleaning step is more art than science. Hence, after you have the first

iteration with an evaluation mechanism in place, you will probably reiterate and improve it.

- **Chunking:** You must adopt various chunking strategies based on each data category and embedding model. For example, when working with code repositories, you want the chunks broader, while when working with articles, you want them narrower or scoped at the paragraph level. Depending on your data, you must decide if you split your document based on the chapter, section, paragraph, sentence or just a fixed window size. Also, you have to ensure that the chunk size doesn't exceed the maximum input size of the embedding model. That is why you usually chunk a document based on your data structure and the maximum input size of the model.
- **Embedding:** You pass each chunk individually to an embedding model of your choice. Implementation-wise, this step is usually the simplest, as tools such as SentenceTransformers and HuggingFace provide high-level interfaces for most embedding models. As explained in the *"What are embeddings"* section of this chapter, at this step, the most critical decisions are to decide what model to use and whether to fine-tune it or not. For example, we used an *"all-mpnet-base-v2"* embedding model from *SentenceTransformers*, which is relatively tiny and runs on

most machines. However, we provide a configuration file where you can quickly configure the embedding model with something more powerful based on the SoTA when reading this book. You can quickly find other options on the Massive Text Embedding Benchmark (MTEB) on HuggingFace.

- **Data loading:** The final step combines the embedding of a chunked document and its metadata, such as author and the document ID, content, URL, platform and creation date. Ultimately, we wrap the vector and the metadata into a structure compatible with Qdrant and push it to the vector database. As we want to use Qdrant as the single source of truth for the features, we also push the cleaned documents (before chunking) to Qdrant. We can push data without vectors, as the metadata index of Qdrant behaves like a NoSQL database. Thus, pushing metadata without a vector attached to it is like using a standard NoSQL engine.

Why did we choose Qdrant as our vector DB?

Qdrant is one of the most popular, robust and feature-rich vector databases out there. For our small MVP, we could have used almost any vector database, but we wanted to pick something that is light and has a high chance of being still used in the industry in the following years.

At the time we wrote this book, Qdrant was used by big players such as X (former 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.

Other popular options are Milvus, Redis, Weaviate, Pinecone, Chroma and pgvector (a PostgreSQL plugin for vector indexes). While writing this book, I 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 can 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, which compares all the top vector databases in everything you can think about, from the license and release year to database features, embedding models and frameworks supported.

Change data capture (CDC): syncing the data warehouse and feature store

As highlighted a few times in this chapter, data is constantly changing, which can result in databases, data lakes, data warehouses and feature stores getting out of sync. Change data capture (CDC) is a strategy that allows you to optimally keep two or more data storage in sync without computing and I/O overhead. It captures any CRUD operation done on the source database and replicates it on a target database. Optionally, add preprocessing steps in between the replication.

The syncing issues also apply when building a feature pipeline. One key design choice concerns how to sync the data warehouse with the feature store to have data fresh enough for your particular use case.

In our LLM Twin use case, we chose a naïve approach out of simplicity. We implemented a batch pipeline that is triggered periodically or manually. It reads all the raw data from the data warehouse, processes it in batches and inserts new records or updates old ones from the Qdrant vector DB. This works fine when you are working with a small number of records, at the order of thousands or tens of thousands. But our naïve approach raises the following questions:

What happens if the data suddenly grows to millions of records (or higher)?

What happens if a record is deleted from the data warehouse? How is this reflected in the feature store?

What if we want to process only the new or updated items from the data warehouse and not all of them?

Fortunately, the CDC pattern can solve all of these issues. When implementing CDC, you can take multiple approaches, but all of them use either a push or pull strategy:

- **Push:** The source database is the primary driver in the push approach. It actively identifies and transmits data modifications to target systems for processing. This method ensures near-instantaneous updates at the target, but data loss can occur if target systems are inaccessible. To mitigate this, a messaging system is typically employed as a buffer.
- **Pull:** The pull method assigns a more passive role to the source database, which only records data changes. Target systems periodically request these changes and handle updates accordingly. While this approach lightens the load on the source, it introduces a delay in data propagation. A messaging system is again essential to prevent data loss during periods of target system unavailability.

In summary, the push method is ideal for applications demanding immediate data access, while the pull method is better suited for large-scale data transfers where real-time updates aren't critical.

With that in mind, there are different methods to detect changes in data. Thus, let's list the main CDC patterns that are used in the industry:

- **Timestamp-based:** The approach involves adding a modification time column to database tables, usually called LAST_MODIFIED or LAST_UPDATED. Downstream systems can query this column to identify records that have been updated since their last check. While simple to implement, this method is limited to tracking changes, not deletions, and imposes performance overhead due to the need to scan entire tables.
- **Trigger-based:** The trigger-based approach utilizes database triggers to automatically record data modifications in a separate table upon INSERT, UPDATE, or DELETE operations, often known as the event table. This method provides comprehensive change tracking but can impact the database performance due to the additional write operations involved for each event.
- **Log-based:** Databases maintain transaction logs to record all data modifications, including timestamps. Primarily used for recovery, these logs can also be leveraged to propagate changes to target systems in real-time. This approach minimizes the performance impact on the source database. As a huge advantage, it avoids additional processing overhead on the source database, captures all data changes, and requires no schema modification. But on the opposite

side, it lacks standardized log formats, leading to vendor-specific implementations.

For more details on CDC, I recommend this article from Confluent's blog: What is Change Data Capture?

With these CDC techniques in mind, we could quickly implement a pull timestamp-based strategy in our RAG feature pipeline to sync the data warehouse and feature store more optimally when the data grows. Our implementation is still pull-based but doesn't check any last updated field in the source database; it just pulls everything from the data warehouse.

However, the most popular and optimal technique in the industry is the log-based one. It doesn't add any I/O overhead to the source database, has low latency and supports all CRUD operations. The biggest downside is its development complexity, which requires a queue to capture all the CRUD events and a streaming pipeline to process them.

As this is an LLM book and not a data engineering one, we wanted to keep things simple, but it's important to know that these techniques exist, and you can always upgrade your current implementation when it doesn't fit your application requirements anymore.

Why is the data stored in two snapshots?

We store two snapshots of our data in the logical feature store:

After the data is cleaned: for fine-tuning LLMs;

After the documents are chunked and embedded: for RAG.

Why did we design it this way? Remember that the features should be accessed solely from the feature store for training and inference. Thus, this adds consistency to our design and makes it cleaner.

Also, storing the data cleaned specifically for our fine-tuning and embedding use case in the MongoDB data warehouse would have been an antipattern. The data from the warehouse is shared all across the company. Thus, processing it for a specific use case is not good practice. Imagine another summarization use case where we must clean and pre-process the data differently. We must create a new "Cleaned Data" table prefixed with the use case name. We have to repeat that for every new use case. Therefore, to avoid having a spaghetti data warehouse, the data from the data warehouse is generic and is modeled to specific applications only in downstream components, which in our case is the feature store.

Ultimately, as we mentioned in the "Core steps" section, you can leverage the metadata index of a vector DB as a NoSQL database. Based on these factors, we decided to keep the cleaned data in Qdrant, along with the chunked and embedded versions of the documents.

As a quick reminder, when operationalizing our LLM Twin system, the create instruct dataset pipeline, explained in Chapter 4, will read the cleaned documents from Qdrant, process them, and save them under a versioned ZenML artifact. The training pipeline requires a dataset and not plain documents. This is a reminder that our logical feature store comprises the Qdrant vector database for online serving and ZenML artifacts for offline training.

Orchestration

ZenML will orchestrate the batch RAG feature pipeline. Using ZenML, we can schedule it to run on a schedule, for example, every hour, or quickly manually trigger. Another option is to trigger it after the ETL data collection pipeline finishes.

By orchestrating the feature pipeline and integrating it into ZenML (or any other orchestration tool), we can operationalize the feature pipeline with the end goal of continuous training

(CT). Also, using ZenML, we can deploy all the pipelines to AWS. We will go into the details of CI/CD/CT and AWS deployment of the pipelines in Chapter 11.

Implementing the LLM Twin's RAG feature pipeline

The last step is to review the LLM Twin's RAG feature pipeline code to see how we applied everything we discussed in this chapter. We will walk you through the:

- ZenML code;
- Pydantic domain objects;
- a custom object-vector mapping (ODM) implementation;
- the cleaning, chunking, and embedding logic for all our data categories.

We will take a top-down approach. Thus, let's start with the settings class and ZenML pipeline.

Settings

We use Pydantic Settings to define a global settings class that loads sensitive or non-sensitive variables from a `.env` file. This approach also gives us all the benefits of Pydantic, such as type validation For example, if we provide for the

`QDRANT_DATABASE_PORT` variable a string instead of an integer, the program will crash. This behavior makes the whole application more deterministic and reliable.

Here is what the `Settings` class looks like with all the variables necessary to build the RAG feature pipeline:

```
from pydantic import BaseSettings
class Settings(BaseSettings):
    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"
    … # Some other settings…
    # RAG
    TEXT_EMBEDDING_MODEL_ID: str = "sentence-transfor
    RERANKING_CROSS_ENCODER_MODEL_ID: str = "cross-en
    RAG_MODEL_DEVICE: str = "cpu"
    # QdrantDB Vector DB
    USE_QDRANT_CLOUD: bool = False
    QDRANT_DATABASE_HOST: str = "localhost"
    QDRANT_DATABASE_PORT: int = 6333
    QDRANT_DATABASE_URL: str = "http://localhost:6333
    QDRANT_CLOUD_URL: str = "str"
    QDRANT_APIKEY: str | None = None
    … # More settings…
settings = Settings()
```

As stated in the internal Config class, all the variables have default values or can be overridden by providing a `.env` file.

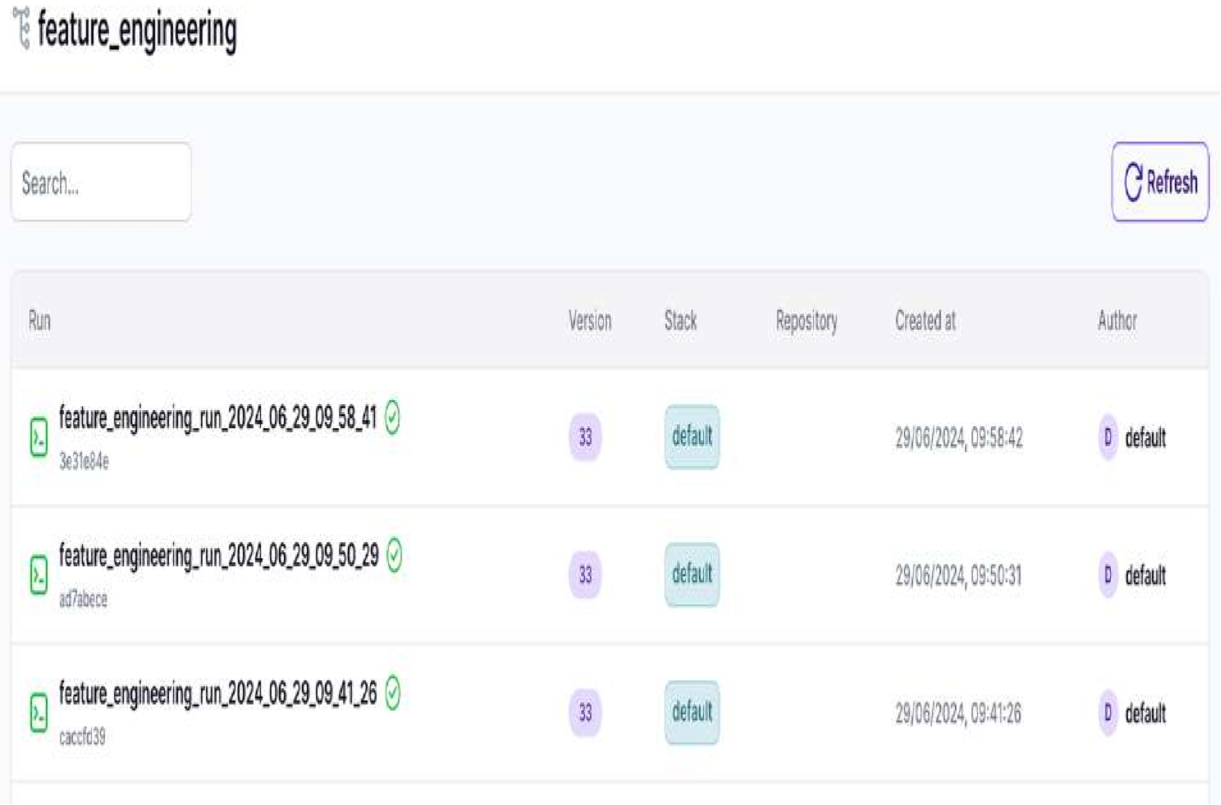
ZenML Pipeline

The ZenML pipeline is the entry point for the RAG feature engineering pipeline. It reflects the five core steps of a RAG ingestion code: extracting raw documents, cleaning, chunking, embedding, and loading them to the logical feature store (available in the GitHub repository at llm_engineering.interfaces.orchestrator.pipelines.feature_engineering.py):

```
from zenml import pipeline
from llm_engineering.interfaces.orchestrator.steps im
@pipeline
def feature_engineering(author_full_names: list[str])
    raw_documents = fe_steps.query_data_warehouse(aut
    cleaned_documents = fe_steps.clean_documents(raw_
    fe_steps.load_to_vector_db(cleaned_documents)
    embedded_documents = fe_steps.chunk_and_embed(cle
    fe_steps.load_to_vector_db(embedded_documents)
```

ZenML 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`. All the other calls with the `feature_engineering()` function are ZenML steps. This is a standard pattern when using orchestrators: you have a high-level function, often called a pipeline, that calls multiple units/steps/tasks.

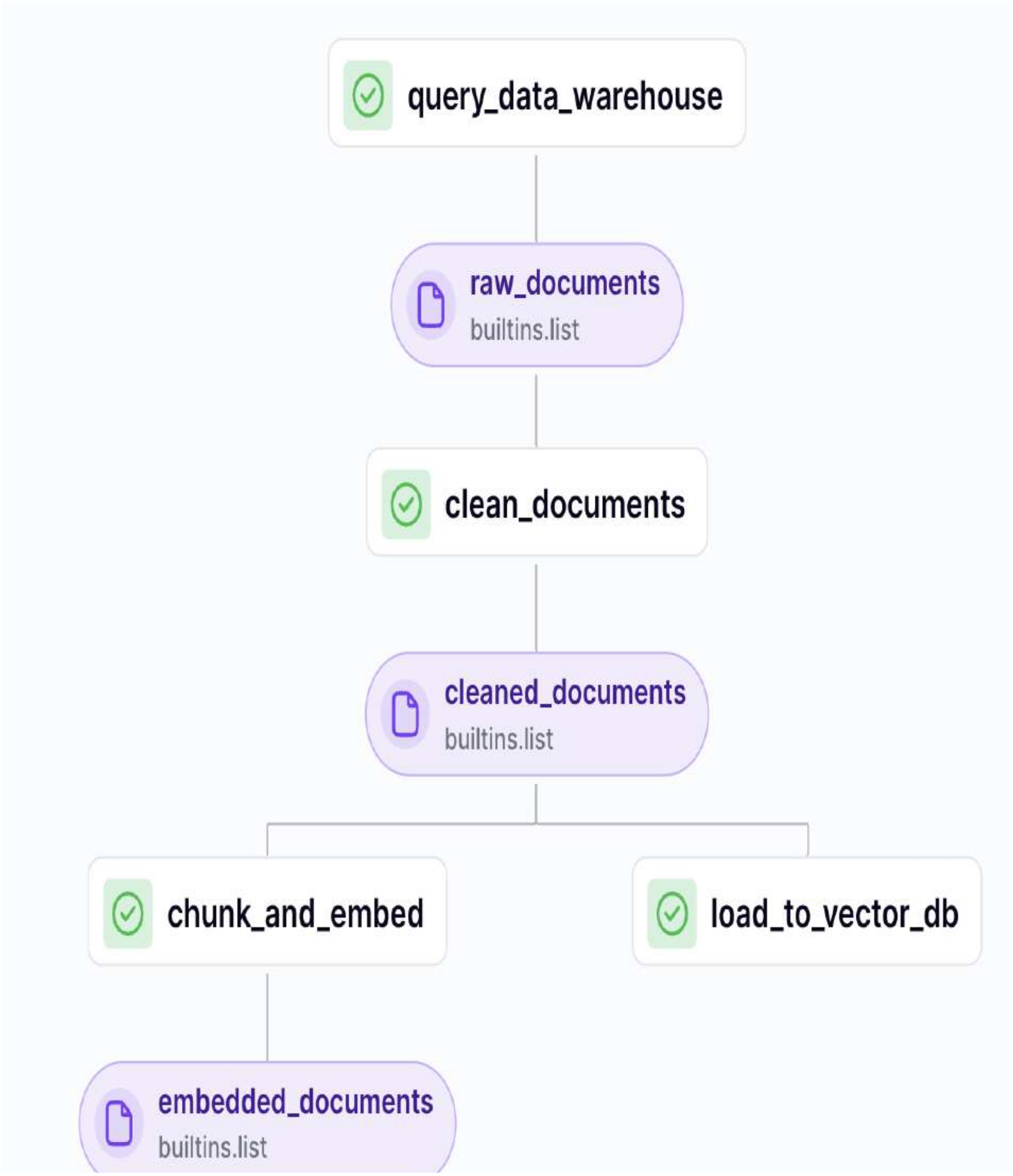


Run	Version	Stack	Repository	Created at	Author
feature_engineering_run_2024_06_29_09_58_41 ✓ 3e31e84e	33	default		29/06/2024, 09:58:42	D default
feature_engineering_run_2024_06_29_09_50_29 ✓ ad7abece	33	default		29/06/2024, 09:50:31	D default
feature_engineering_run_2024_06_29_09_41_26 ✓ caccfd39	33	default		29/06/2024, 09:41:26	D default

Figure 8.10 – Feature pipeline runs in ZenML dashboard

Figure 8.10 shows how multiple pipeline runs look in ZenML's dashboard. Figure 8.11 shows the DAG of the RAG feature pipeline, where you can follow all the pipeline steps and their output artifacts. Remember that whatever is returned from a

ZenML step is automatically saved as an artifact, stored in ZenML's artifact registry, versioned and sharable across the application.



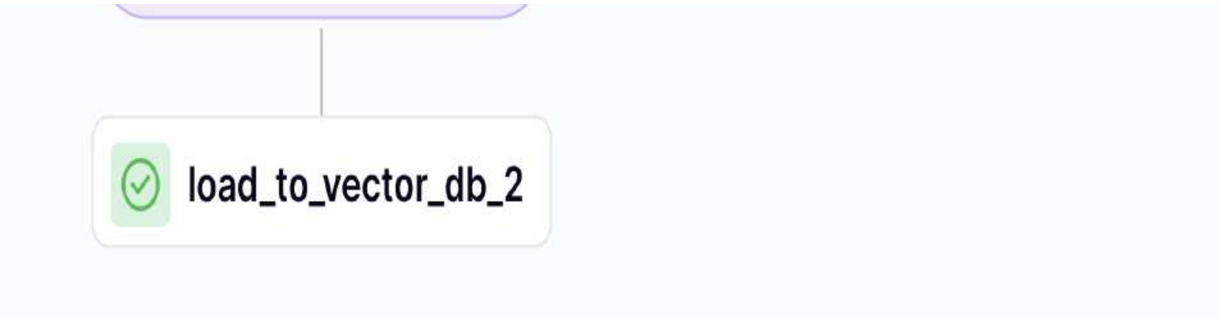


Figure 8.11 – Feature pipeline DAG in the ZenML dashboard

The final puzzle piece is understanding how to configure the RAG feature pipeline dynamically. All its available settings are exposed as function parameters. Here, we need only a list of author's names, as seen in the function's signature: `feature_engineering(author_full_names: list[str])`. We inject a YAML configuration file at runtime that contains all the necessary values based on different use cases. For example, the following config includes a list of all the authors of this book as we want to populate the feature store with data from all of us (available in the GitHub repository at configs/feature_engineering.yaml):

```
parameters:
  author_full_names:
    - Alex Vesa
    - Maxime Labonne
    - Paul Iusztin
```

The beauty of this approach is that you don't have to modify the code to configure the feature pipeline with different input values. You have to provide a different configuration file when running it as follows:

```
feature_engineering.with_options(config_path="…/featu
```

You can adapt this code to provide the config path from the CLI, allowing you to operationalize the pipeline quickly.

Before diving into the ZenML steps, let's look at how to call the RAG feature engineering pipeline. We implement a run.py file that allows us to configure all our ZenML pipelines from the CLI, which is precisely what we described above. Thus, we have to call the run.py file as a module and invoke the feature engineer pipeline:

```
python -m llm_engineering.interfaces.orchestrator.run
```

We wrapped all the necessary steps into an intuitive interface using Poe the Poet. A tool similar to Makefile that plays nicely with Poetry. Using a tool as a façade over all your CLI commands is necessary to run your application as it significantly simplifies its complexity. Think about it as out-of-the-box documentation. To run the feature pipeline using a Poe command, run:

```
poetry poe run-feature-engineering-pipeline
```

Let's move forward to the ZenML steps.

ZenML Steps

Let's take each step individually and understand what happens under the hood. Source code available GitHub at llm_engineering/interfaces/orchestrator/steps/feature_engineering.

Querying the data warehouse

The first thing to notice is that a step is a Python function decorated with `@step`, similar to how a ZenML pipeline works. The function below takes as input a list of authors' full names and performs the following core steps:

It attempts to get or create a `UserDocument` instance using the first and last names, appending this instance to the authors list. If the user doesn't exist, it throws an error.

It fetches all the raw data for the user from the data warehouse and extends the `documents` list to include these user documents.

Ultimately, it computes a descriptive metadata dictionary logged and tracked in ZenML.

```
… # other imports
from zenml import get_step_context, step
@step
def query_data_warehouse(
    author_full_names: list[str],
) -> Annotated[list, "raw_documents"]:
    documents = []
    authors = []
    for author_full_name in author_full_names:
        logger.info(f"Querying data warehouse for use
        first_name, last_name = utils.split_user_full
        logger.info(f"First name: {first_name}, Last 
        user = UserDocument.get_or_create(first_name=
        authors.append(user)
        results = fetch_all_data(user)
        user_documents = [doc for query_result in res
        documents.extend(user_documents)
    step_context = get_step_context()
    step_context.add_output_metadata(output_name="raw
    return documents
```

The fetch function leverages a thread pool that runs each query on a different thread. As we have multiple data categories, we have to make a different query for the articles, posts, and repositories as they are stored in different collections. Each query calls the data warehouse, which is bounded by the network I/O and data warehouse latency, not by the machine's CPU. Thus, by moving each query to a different thread, we can parallelize them. Ultimately, instead of adding the latency of each query as the total timing, the time to run this fetch function will be the max between all the calls.

Using threads to parallelize I/O-bounded calls is good practice in Python, as they are not locked by the Python Global Interpreter Lock (GIL). In contrast, adding each call to a different process would add too much overhead, as a process takes longer to spin off than a thread.

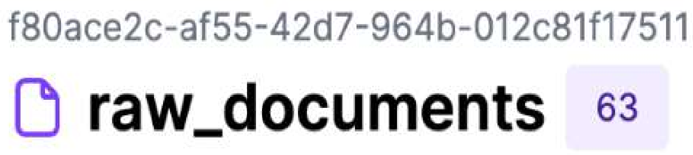
In Python, you want to parallelize things with processes only when the operations are CPU or memory-bound because the GIL affects them. Each process has a different GIL. Thus, parallelizing your computing logic, such as processing a batch of documents or images already loaded in memory, isn't affected by Python's GIL limitations.

```
def fetch_all_data(user: UserDocument) -> dict[str, l
    user_id = str(user.id)
    with ThreadPoolExecutor() as executor:
        future_to_query = {
            executor.submit(__fetch_articles, user_id
            executor.submit(__fetch_posts, user_id):
            executor.submit(__fetch_repositories, use
        }
        results = {}
        for future in as_completed(future_to_query):
            query_name = future_to_query[future]
            try:
                results[query_name] = future.result()
            except Exception:
                logger.exception(f"'{query_name}' req
                results[query_name] = []
    return results
```

The `_get_metadata()` function takes the list of queried documents and authors and counts the number of them relative to each data category:

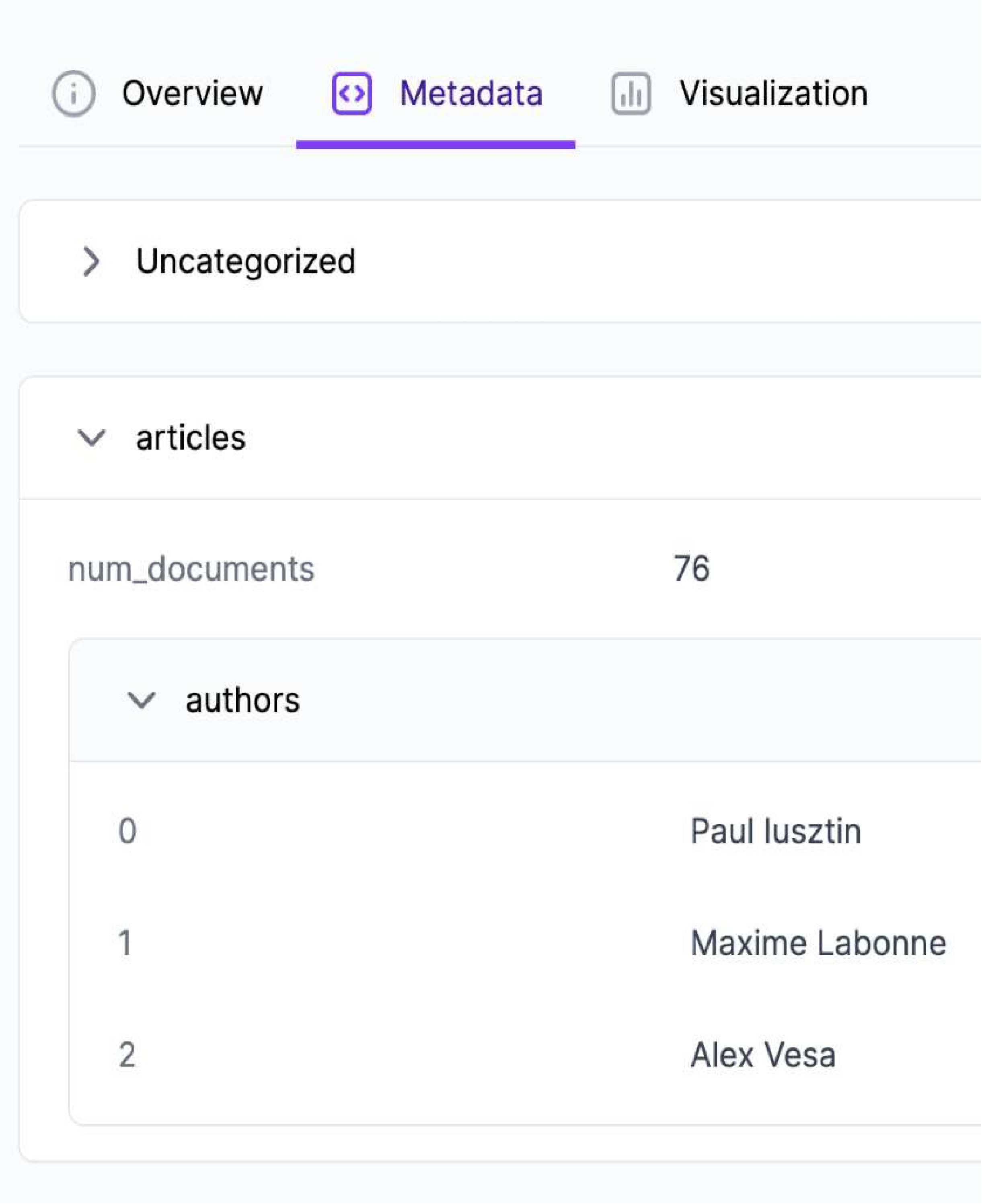
```
def _get_metadata(documents: list[Document]) -> dict:
    metadata = {
        "num_documents": len(documents),
    }
    for document in documents:
        collection = document.get_collection_name()
        if collection not in metadata:
            metadata[collection] = {}
        if "authors" not in metadata[collection]:
            metadata[collection]["authors"] = list()
        metadata[collection]["num_documents"] = metad
        metadata[collection]["authors"].append(docume
    for value in metadata.values():
        if isinstance(value, dict) and "authors" in v
            value["authors"] = list(set(value["author
    return metadata
```

We will expose this metadata in the ZenML dashboard to quickly see some statistics on the loaded data. For example, in Figure 8.12, we accessed the metadata tab of the `query_data_warehouse()` step, where you can see that within that particular run of the feature pipeline, we loaded 76 documents from three authors. This is extremely powerful for monitoring and debugging batch pipelines. You can always extend it with anything that makes sense for your use case.



f80ace2c-af55-42d7-964b-012c81f17511

raw_documents 63



num_documents	76

authors

0	Paul Iusztin
1	Maxime Labonne
2	Alex Vesa

Figure 8.12 – Metadata of the query the data warehouse ZenML step.

Cleaning the documents

In the cleaning step, we iterate through all the documents and delegate all the logic to a `CleaningDispatcher` who knows what cleaning logic to apply based on the data category. Remember that we want to apply, or the ability to apply in the future, different cleaning techniques on articles, posts, and code repositories.

```
@step
def clean_documents(
    documents: Annotated[list, "raw_documents"],
) -> Annotated[list, "cleaned_documents"]:
    cleaned_documents = []
    for document in documents:
        cleaned_document = CleaningDispatcher.dispatc
        cleaned_documents.append(cleaned_document)
    step_context = get_step_context()
    step_context.add_output_metadata(output_name="cle
    return cleaned_documents
```

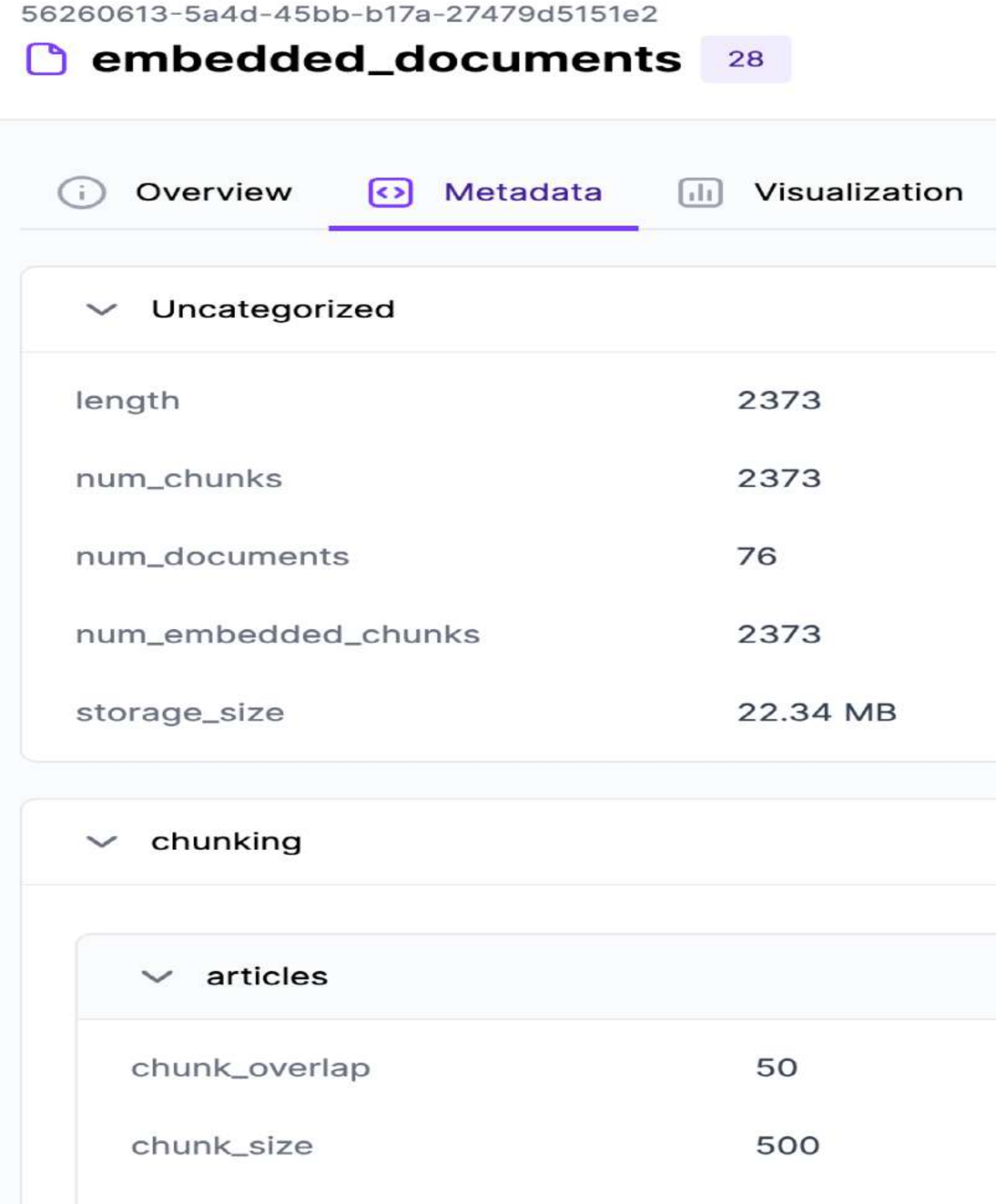
The computed metadata is similar to what we logged in the `query_data_warehouse()` step. Thus, let's move on to chunking and embedding.

Chunk and embed the cleaned documents

Similar to how we cleaned the documents, we delegate the chunking and embedding logic to a dispatcher who knows how to handle each data category. Note that the chunking dispatcher returns a list instead of a single object, which makes sense as the document is split into multiple chunks. We will dig into the dispatcher in "The dispatcher layer" section of this chapter.

```
@step
def chunk_and_embed(
    cleaned_documents: Annotated[list, "cleaned_docum
) -> Annotated[list, "embedded_documents"]:
    metadata = {"chunking": {}, "embedding": {}, "num
    embedded_chunks = []
    for document in cleaned_documents:
        chunks = ChunkingDispatcher.dispatch(document
        metadata["chunking"] = _add_chunks_metadata(c
        for batched_chunks in utils.misc.batch(chunks
            batched_embedded_chunks = EmbeddingDispat
            embedded_chunks.extend(batched_embedded_c
    … # add more stuff to the metadata dictionary
    step_context.add_output_metadata(output_name="emb
    return embedded_chunks
```

In Figure 8.13, you can see the metadata of the chunking and embedding ZenML step. For example, you can quickly understand that we transformed 76 documents into 2373 chunks or the properties we used for chunking articles, such as a `chunk_size` of 500 and a `chunk_overlap` of 50.



56260613-5a4d-45bb-b17a-27479d5151e2

embedded_documents 28

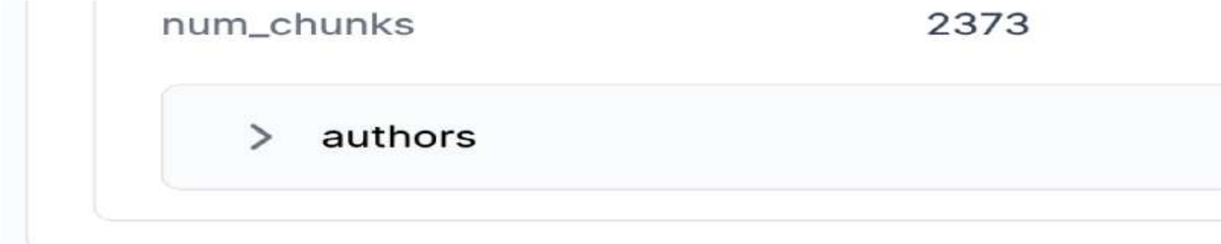
Uncategorized

length	2373
num_chunks	2373
num_documents	76
num_embedded_chunks	2373
storage_size	22.34 MB

chunking

articles

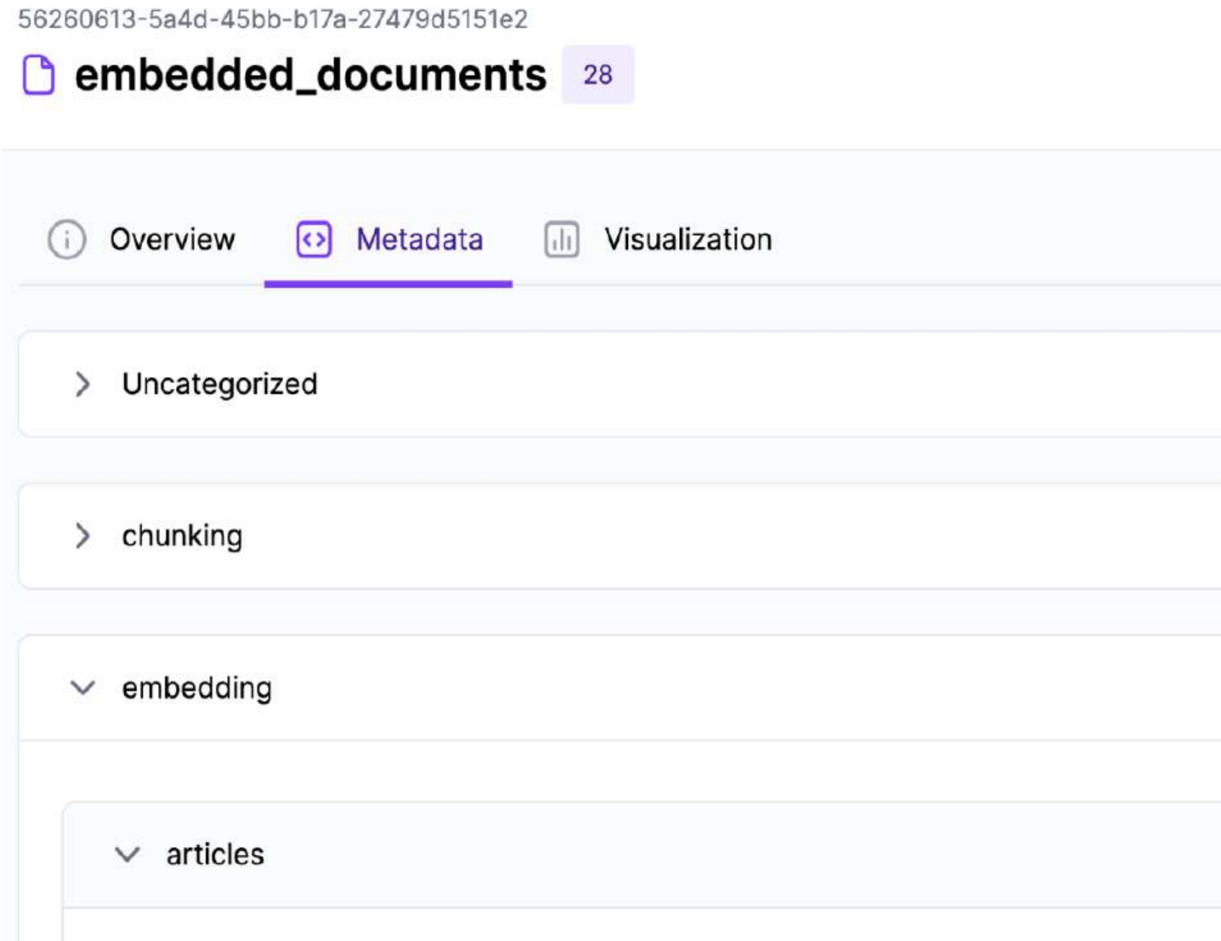
chunk_overlap	50
chunk_size	500



num_chunks 2373

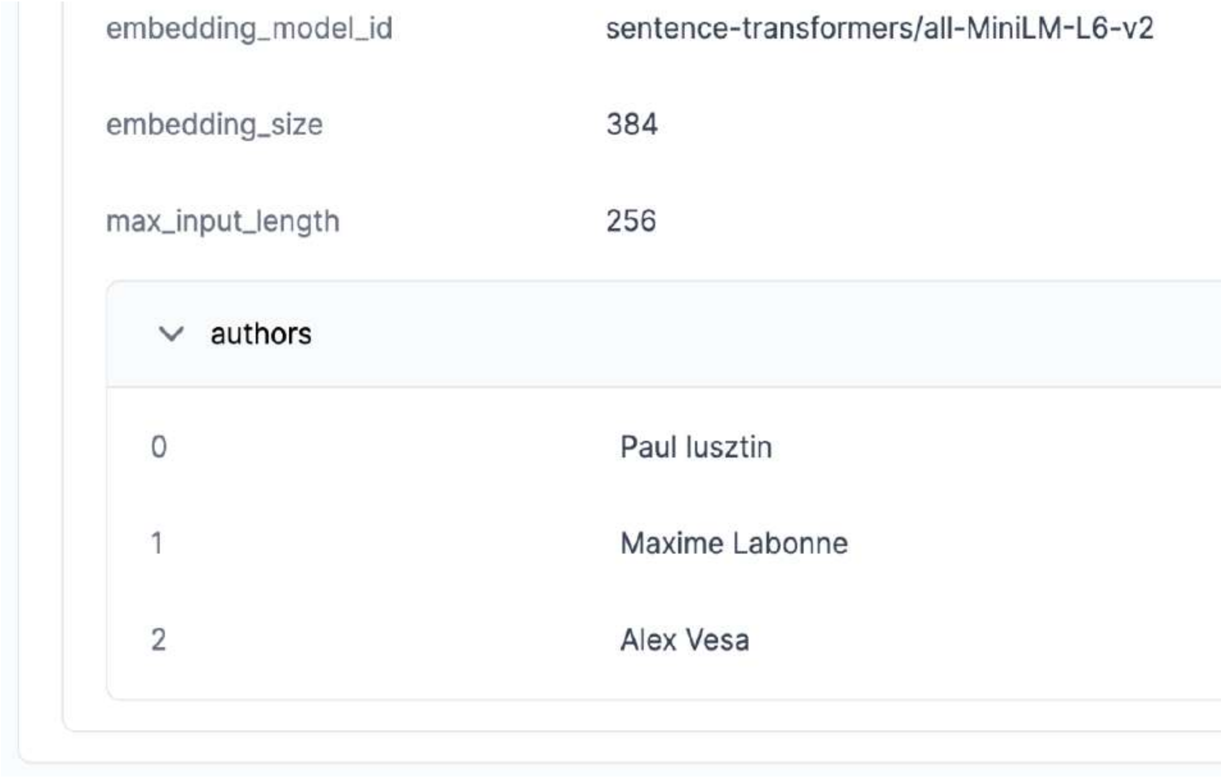
Figure 8.13 – Metadata of the embedding and chunking ZenML step, detailing the uncategorized and chunking dropdowns.

In Figure 8.14, the rest of the ZenML metadata from the embedding and chunking step details the embedding model and its properties used to compute the vectors.



56260613-5a4d-45bb-b17a-27479d5151e2

embedded_documents 28



embedding_model_id	sentence-transformers/all-MiniLM-L6-v2
embedding_size	384
max_input_length	256

authors

0	Paul Iusztin
1	Maxime Labonne
2	Alex Vesa

Figure 8.14 – Metadata of the embedding and chunking ZenML step, detailing the embedding dropdown.

As ML systems can break at any time while in production due to drifts or untreated use cases, levering the metadata section to monitor the ingested data can be a powerful tool that will save debugging days, translating to tens of thousands of dollars or more for your business.

Loading the documents to the vector database

As each article, post or code repository sits in a different collection inside the vector database, we have to group all the documents based on their data category. Then, we load each group in bulk in the Qdrant vector database.

```
@step
def load_to_vector_db(
    documents: Annotated[list, "documents"],
) -> None:
    logger.info(f"Loading {len(documents)} documents 
    grouped_documents = VectorBaseDocument.group_by_c
    for document_class, documents in grouped_document
        logger.info(f"Loading documents into {documen
        document_class.bulk_insert(documents)
```

Pydantic domain entities

Before investigating the dispatchers, we must understand the domain objects we work with. To some extent, in implementing the LLM Twin, we are following the Domain-Driven Design (DDD) principles, which state that domain entities are the core of your application. Thus, before proceeding, it's important to understand the hierarchy of domain classes we are working with.

The code for the domain entities is available on GitHub under llm_engineering/domain.

We used Pydantic to model all our domain entities. When we wrote the book, choosing Pydantic was a no-brainer, as it is the go-to Python package for writing data structures with out-of-the-box type validation. As Python is a dynamically typed language, using Pydantic for type validation at runtime makes your system order of times more robust, as you can be sure that you are always working with the right type of data.

The domain of our LLM Twin application is split into two dimensions:

- **The data category**: post, article, repository
- **The state of the data:** cleaned, chunked, embedded

We decided to create a base class for each state of the document, resulting in having the following base abstract classes:

- `class CleanedDocument(VectorBaseDocument, ABC)`
- `class Chunk(VectorBaseDocument, ABC)`
- `class EmbeddedChunk(VectorBaseDocument, ABC)`

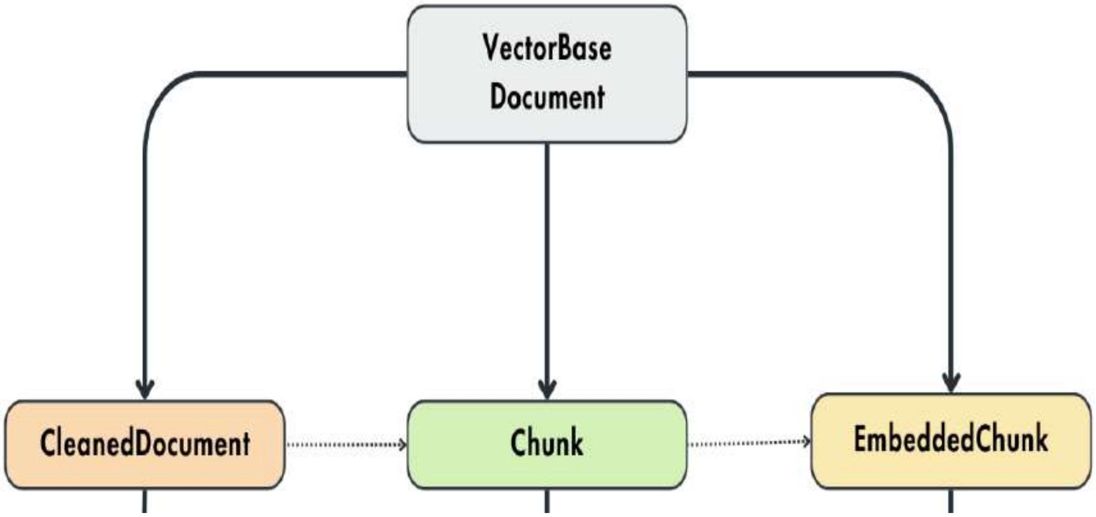
Note that all of them inherit the `VectorBaseDocument` class, which is our custom object vector mapping (OVM) implementation, which we will explain in the next section of this chapter. Also, it inherits from ABC, which makes the class abstract. Thus, you cannot initialize an object out of these classes but only inherit to inherit from them. That is why base classes are always marked as abstract.

Each base abstract class from above will have a subclass representing its data category. For example, the `CleanedDocument` class will have the following subclasses:

`class CleanedPostDocument(CleanedDocument)`

- `class CleanedArticleDocument(CleanedDocument)`
- `class CleanedRepositoryDocument(CleanedDocument)`

We will repeat the same logic for the `Chunk` and `EmbeddedChunk` base abstract classes.



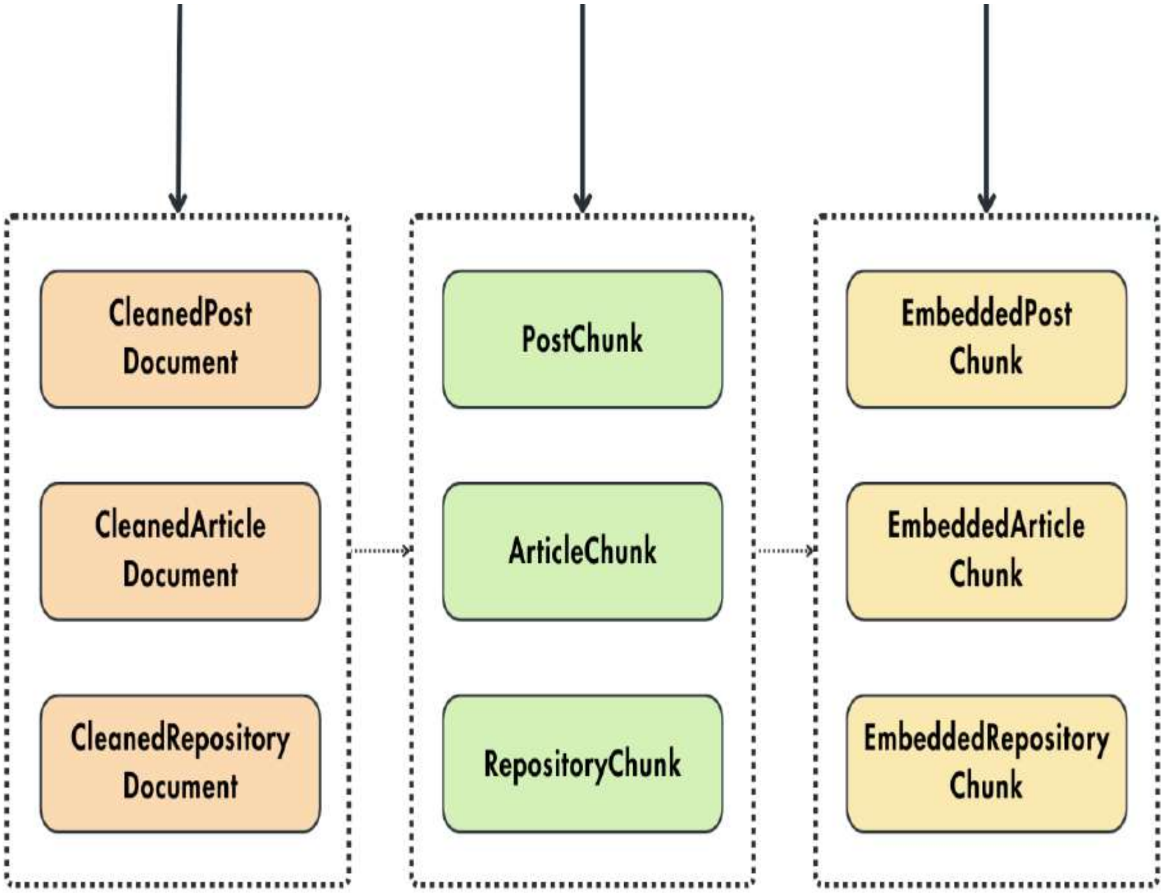


Figure 8.15 – Domain entities class hierarchy and their interaction.

We chose this design because the list of states will rarely change, and we want to extend the list of data categories. Thus, structuring the classes after the state allows us to plug another data category by inheriting these base abstract classes.

Let's see the complete code for the hierarchy of the cleaned document. All the attributes of a cleaned document will be saved within the metadata of the vector database. For example, the metadata of a cleaned article document will always contain the content, platform, author ID, author full name and link of the article.

Another fundamental aspect is the `Config` internal class, which defines the name of the collection within the vector database, the data category of the entity, and whether to leverage the vector index when creating the collection.

```
class CleanedDocument(VectorBaseDocument, ABC):
    content: str
    platform: str
    author_id: UUID4
    author_full_name: str
class CleanedPostDocument(CleanedDocument):
    image: Optional[str] = None
    class Config:
        name = "cleaned_posts"
        category = DataCategory.POSTS
        use_vector_index = False
class CleanedArticleDocument(CleanedDocument):
    link: str
    class Config:
        name = "cleaned_articles"
        category = DataCategory.ARTICLES
        use_vector_index = False
class CleanedRepositoryDocument(CleanedDocument):
    name: str
    link: str
    class Config:
        name = "cleaned_repositories"
        category = DataCategory.REPOSITORIES
        use_vector_index = False
```

To conclude this section, let's also take a look at the base abstract class of the chunk and embedded chunk:

```
class Chunk(VectorBaseDocument, ABC):
    content: str
    platform: str
    document_id: UUID4
    author_id: UUID4
    author_full_name: str
    metadata: dict = Field(default_factory=dict)
… # PostChunk, ArticleChunk, RepositoryChunk
class EmbeddedChunk(VectorBaseDocument, ABC):
    content: str
    embedding: list[float] | None
    platform: str
    document_id: UUID4
    author_id: UUID4
    author_full_name: str
    metadata: dict = Field(default_factory=dict)
… # EmbeddedPostChunk, EmbeddedArticleChunk, Embedded
```

We also defined an Enum that aggregates all our data categories in a single structure of constants:

```
class DataCategory(StrEnum):
    POSTS = "posts"
    ARTICLES = "articles"
    REPOSITORIES = "repositories"
```

The last step to fully understand how the domain objects work is to zoom into the `VectorBaseDocument` OVM class.

Object-vector mapping (OVM)

The word "Object-vector mapping (OVM)" is inspired by the "Object-relational mapping (ORM)" pattern, which is omnipresent in software engineering. We called it OVM because we work with embedding and vector DBs instead of structured data and SQL tables.

But what is an ORM in the first place? This technique lets you query and manipulate data from a database using an objectoriented paradigm. While the term often refers to a specific library type, ORM is fundamentally a conceptual approach. Instead of writing SQL or API-specific queries, you encapsulate all the complexity under an ORM class that knows how to handle all the database operations, most commonly based on CRUD. An ORM interacts with an SQL database, such as PostgreSQL or MySQL.

ost modern Python applications use ORMs when interacting with the database. Even though SQL is still a popular choice in the data world, you will rarely see raw SQL queries in Python backend components. The most popular Python ORM is SQLAlchemy. Also, with the rise of FastAPI, SQLModel is a common choice, which is actually a wrapper over SQLAlchemy that makes the integration easier with FastAPI.

For example, in the example below, using SQLAlchemy, we defined a User ORM with an ID, name, and list of addresses:

```
… # other imports
from sqlalchemy.orm import mapped_column, Mapped, rel
class User(Base):
    __tablename__ = "user_account"
    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(30))
    addresses: Mapped[List["Address"]] = relationship
        back_populates="user", cascade="all, delete-c
    )
```

Source: SQL Alchemy Quick Start Guide

Using the User ORM, we can quickly insert or query users directly from Python without writing a line of SQL. Note that an ORM usually supports all CRUD operations.

```
session = ... # initialize database session
session.add(user)
select(User).where(User.name.in_(["Paul", "Maxime", "A
```

Now, let's move to our OVM implementation. Even if our custom example is simple, it's a powerful example of how to write modular and extendable code by leveraging OOP principles.

The full implementation of the `VectorBaseDocument` class is available on GitHub under the llm_engineering/domain/base/vector.py file.

Our OVM will support CRUD operations on top of Qdrant. Based on our application's demands, we limited it only to create and read operations, but it can easily be extended to update and delete records. Let's take a look at the core of the implementation:

```
from pydantic import UUID4, BaseModel
from typing import Generic
from llm_engineering.infrastructure.db.qdrant import
T = TypeVar("T", bound="VectorBaseDocument")
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    id: UUID4 = Field(default_factory=uuid.uuid4)
    @classmethod
    def from_record(cls: Type[T], point: Record) -> T
        _id = UUID(point.id, version=4)
        payload = point.payload or {}
        attributes = {
            "id": _id,
            **payload,
        }
        if cls._has_class_attribute("embedding"):
            payload["embedding"] = point.vector or No
        return cls(**attributes)
    def to_point(self: T, **kwargs) -> PointStruct:
        exclude_unset = kwargs.pop("exclude_unset", F
        by_alias = kwargs.pop("by_alias", True)
        payload = self.dict(exclude_unset=exclude_uns
        _id = str(payload.pop("id"))
        vector = payload.pop("embedding", {})
        if vector and isinstance(vector, np.ndarray):
            vector = vector.tolist()
        return PointStruct(id=_id, vector=vector, pay
```

The `VectorBaseDocument` class inherits from Pydantic's `BaseModel` and helps us structure a single record's attributes from the vector database. Every OVM will be initialized by default with a UUID4 as its unique identifier. Using generics, more precisely by inheriting from `Generic[T]`, the signatures of all the subclasses of the `VectorBaseDocument` class will adapt to that given class. For example, the `from_record()` method of the `Chunk()` class, which inherits `VectorBaseDocument`, will return the Chunk type, which drastically helps the static analyzer and type checkers such as mypy.

The `from_record()` method adapts a data point from Qdrant's format to our internal structure based on Pydantic. On the other hand, the `to_point()` method takes the attributes of the current instance and adapts them to Qdrant's `PointStruct()` format. We will leverage these two methods for our create and read operations.

Ultimately, all operations made to Qdrant will be done through the `connection` instance, which is instantiated in the application's infrastructure layer.

The `bulk_insert()` method maps each document to a point. Then, it uses the Qdrant `connection` instance to load all the points to a given collection in Qdrant. If the insertion fails once, it tries to create the collection and do the insertion again. Often, it is good practice to split your logic into two functions. One private function contains the logic, in our case _bulk_insert(), and one public function handles all the errors and failure scenarios.

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    … # Rest of the class
    @classmethod
    def bulk_insert(cls: Type[T], documents: list["Ve
        try:
            cls._bulk_insert(documents)
        except exceptions.UnexpectedResponse:
            logger.info(
                f"Collection '{cls.get_collection_nam
            )
            cls.create_collection()
            try:
                cls._bulk_insert(documents)
            except exceptions.UnexpectedResponse:
                logger.error(f"Failed to insert docum
                return False
        return True
    @classmethod
    def _bulk_insert(cls: Type[T], documents: list["V
        points = [doc.to_point() for doc in documents
        connection.upsert(collection_name=cls.get_col
```

The collection name is inferred from the `Config` class defined in the subclasses inheriting the OVM:

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    … # Rest of the class
    @classmethod
    def get_collection_name(cls: Type[T]) -> str:
        if not hasattr(cls, "Config") or not hasattr(
            raise ImproperlyConfigured(
                "The class should define a Config cla
            )
        return cls.Config.name
```

Now, we must define a method that lets us read all the records from the vector DB (without using vector similarity search logic). The `bulk_find()` method enables us to scroll (or list) all the records from a collection. The function below scrolls the Qdrant vector DB, which returns a list of data points, which are ultimately mapped to our internal structure using the `from_record()` method.

The limit parameters control how many items we return at once, and the offset signals the ID of the point from which Qdrant starts returning records.

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    … # Rest of the class
    @classmethod
    def bulk_find(cls: Type[T], limit: int = 10, **kw
        try:
            documents, next_offset = cls._bulk_find(l
        except exceptions.UnexpectedResponse:
            logger.error(f"Failed to search documents
            documents, next_offset = [], None
        return documents, next_offset
    @classmethod
    def _bulk_find(cls: Type[T], limit: int = 10, **k
        collection_name = cls.get_collection_name()
        offset = kwargs.pop("offset", None)
        offset = str(offset) if offset else None
        records, next_offset = connection.scroll(
            collection_name=collection_name,
            limit=limit,
            with_payload=kwargs.pop("with_payload", T
            with_vectors=kwargs.pop("with_vectors", F
            offset=offset,
            **kwargs,
        )
        documents = [cls.from_record(record) for reco
        if next_offset is not None:
            next_offset = UUID(next_offset, version=4
        return documents, next_offset
```

The last piece of the puzzle is to define a method that performs a vector similarity search on a provided query embedding. Like before, we defined a public `search()` and private _`search()` method. The search is performed by Qdrant when calling the `connection.search()` function.

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    … # Rest of the class
    @classmethod
    def search(cls: Type[T], query_vector: list, limi
        try:
            documents = cls._search(query_vector=quer
        except exceptions.UnexpectedResponse:
            logger.error(f"Failed to search documents
            documents = []
        return documents
    @classmethod
    def _search(cls: Type[T], query_vector: list, lim
        collection_name = cls.get_collection_name()
        records = connection.search(
            collection_name=collection_name,
            query_vector=query_vector,
            limit=limit,
            with_payload=kwargs.pop("with_payload", T
            with_vectors=kwargs.pop("with_vectors", F
            **kwargs,
        )
        documents = [cls.from_record(record) for reco
        return documents
```

Now that we understand what our domain entities look like and how the OVM works let's move on to the dispatchers who clean, chunk, and embed the documents.

The dispatcher layer

A dispatcher inputs a document and applies dedicated handlers based on its data category (article, post, or repository). A handler can either clean, chunk or embed a document.

Let's start by zooming in on the `CleaningDispatcher`. It mainly implements a `dispatch()` method that inputs a raw document. Based on its data category, it instantiates and calls a handler that applies the cleaning logic specific to that data point.

```
class CleaningDispatcher:
    cleaning_factory = CleaningHandlerFactory()
    @classmethod
    def dispatch(cls, data_model: NoSQLBaseDocument)
        data_category = DataCategory(data_model.get_c
        handler = cls.cleaning_factory.create_handler
        clean_model = handler.clean(data_model)
        logger.info(
            "Data cleaned successfully.",
            data_category=data_category,
            cleaned_content_len=len(clean_model.conte
        )
        return clean_model
```

The key in the dispatcher logic is the `CleaningHandlerFactory()`, which instantiates a different cleaning handler based on the document's data category.

```
class CleaningHandlerFactory:
    @staticmethod
    def create_handler(data_category: DataCategory) -
        if data_category == DataCategory.POSTS:
            return PostCleaningHandler()
        elif data_category == DataCategory.ARTICLES:
            return ArticleCleaningHandler()
        elif data_category == DataCategory.REPOSITORI
            return RepositoryCleaningHandler()
        else:
            raise ValueError("Unsupported data type")
```

The dispatcher or factory classes are nothing fancy, but they offer an intuitive and simple interface for applying various operations to your documents. When manipulating documents, instead of worrying about their data category and polluting your business logic with if-else statements, you have a class dedicated to handling that. You have a single class that cleans any document, which respects the DRY (don't repeat yourself) principles from software engineering. By respecting DRY, you have a single point of failure, and the code can easily be extended. For example, if we add an extra type, we must extend only the factory class instead of multiple occurrences in the code.

The `ChunkingDispatcher` and `EmbeddingDispatcher` follow the same pattern. They use a `ChunkingHandlerFactory` and, respectively, an `EmbeddingHandlerFactory` that initializes the correct handler based on the data category of the input document. Afterward, they call the handler and return the result.

The source code of all the dispatchers and factories can be found on GitHub at llm_engineering/application/preprocessing/dispatchers.py

The factory class leverages the Abstract Factory creational pattern, which instantiates a family of classes implementing the same interface. In our case, these handlers implement the `clean()` method regardless of the handler type.

Also, the handler class family leverages the Strategy behavioral pattern used to instantiate when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.

Intuitively, in our dispatcher layer, the combination of the factory and strategy patterns works as follows:

Initially, we knew we wanted to clean the data, but as we knew the data category only at runtime, we couldn't decide on what strategy to apply.

We can write the whole code around the cleaning code and abstract away the logic under a `Handler()` interface, which will represent our strategy.

When we get a data point, we apply the abstract factory pattern and create the correct cleaning handler for its data type.

Ultimately, the dispatcher layer uses the handler and executes the right strategy.

By doing so, we:

isolate the logic for a given data category;

leverage polymorphism to avoid filling up the code with 1000x if-else statements;

make the code modular and extendable: when a new data category arrives, we must implement a new handler and modify the factory class without touching any other part of the code.

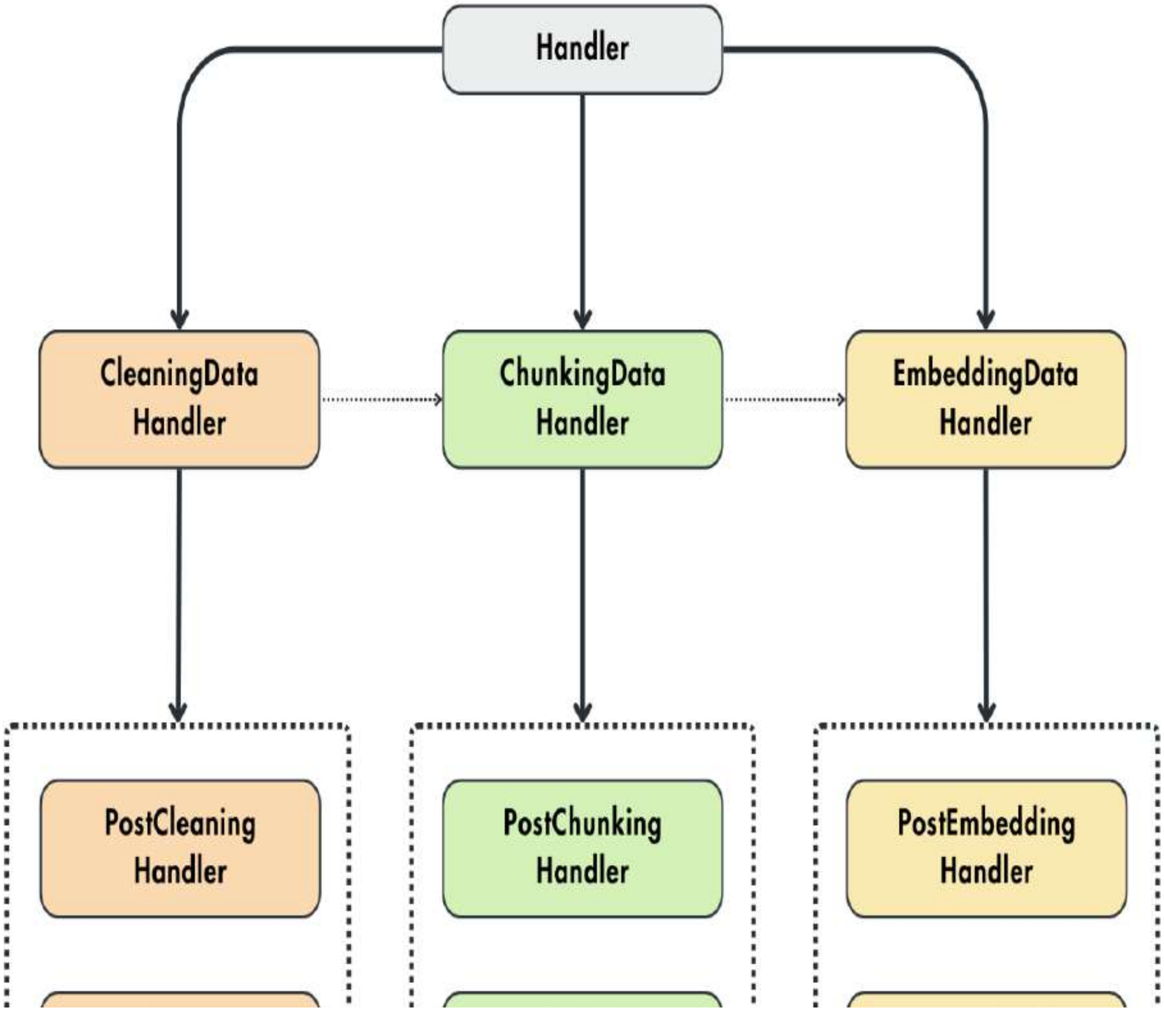
> *Until now, we have just modeled our entities and how the data flows in our application. We haven't written a single cleaning, chunking or embedding code. That is one big difference between a quick demo and a production-ready application. In a demo, you don't care about software engineering best practices and structuring your code to make it future-proof. However, writing clean, modular, and scalable code is critical for its longevity when building a real-world application.*

The last component of the RAG feature pipeline is the implementation of the cleaning, chunking and embedding handlers.

The handlers

The handler has a one-on-one structure with our domain, meaning that every entity has its own handler, as shown in Figure 8.15. Thus, we will have the following base interfaces:

- `class CleaningDataHandler()`
- `class ChunkingDataHandler()`
- `class EmbeddingDataHandler()`



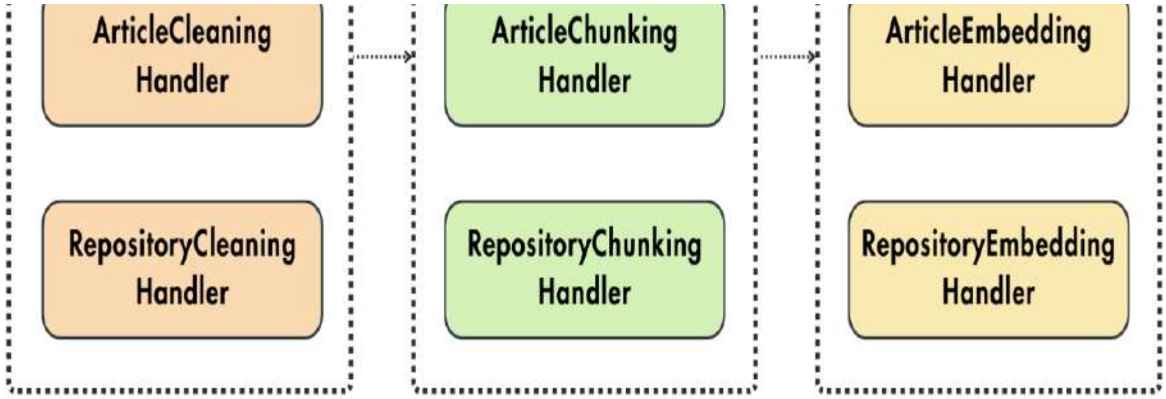

ArticleCleaning Handler
ArticleChunking Handler
ArticleEmbedding Handler
RepositoryCleaning Handler
RepositoryChunking Handler
RepositoryEmbedding Handler

Figure 8.16 – Handlers class hierarchy and their interaction.

The code for all the handlers is available on GitHub at llm_engineering/application/preprocessing. Now, let's examine each handler family and see how it is implemented.

The cleaning handlers

The `CleaningDataHandler()` strategy interface looks as follows:

```
… # Other imports.
from typing import Generic, TypeVar
DocumentT = TypeVar("DocumentT", bound=Document)
CleanedDocumentT = TypeVar("CleanedDocumentT", bound=
class CleaningDataHandler(ABC, Generic[DocumentT, Cle
    @abstractmethod
    def clean(self, data_model: DocumentT) -> Cleaned
        pass
```

Now, for every post, article and repository, we have to implement a different handler, as follows:

```
class PostCleaningHandler(CleaningDataHandler):
    def clean(self, data_model: PostDocument) -> Clea
        return CleanedPostDocument(
            id=data_model.id,
            content=clean_text(" #### ".join(data_mod
            … # Copy the rest of the parameters from 
        )
class ArticleCleaningHandler(CleaningDataHandler):
    def clean(self, data_model: ArticleDocument) -> C
        valid_content = [content for content in data_
        return CleanedArticleDocument(
            id=data_model.id,
            content=clean_text(" #### ".join(valid_co
            platform=data_model.platform,
            link=data_model.link,
            author_id=data_model.author_id,
            author_full_name=data_model.author_full_n
        )
class RepositoryCleaningHandler(CleaningDataHandler):
    def clean(self, data_model: RepositoryDocument) -
        return CleanedRepositoryDocument(
            id=data_model.id,
            content=clean_text(" #### ".join(data_mod
            … # Copy the rest of the parameters from
        )
```

The handlers input a raw document domain entity, clean the content, and return a cleaned document. All the handlers use the `clean_text()` function to clean the text. Out of simplicity, we used the same cleaning technique for all the data categories. Still, in a real-world setup, we would have to further optimize and create a different cleaning function for each data category. The strategy pattern makes this a breeze, as we swap the cleaning function in the handlers, and that's it.

The cleaning steps applied in the `clean_text()` function are the same ones discussed in Chapter 4 in the *"Creating an instruction dataset"* section. We don't want to repeat ourselves. Thus, for a refresher, check out that chapter. At this point, we mostly care about automating and integrating the whole logic into the RAG feature pipeline. Thus, after operationalizing the ML system, all the cleaned data used for fine-tuning will be accessed from the logical feature store, making it the single source of truth for accessing data.

The chunking handlers

First, let's examine the `ChunkingDataHandler()` strategy handler. We exposed the `chunk_size` and `chunk_overlap` as class properties to make them verbose and let the subclasses override them based on their data category and use case. The handler takes cleaned documents as input and returns chunk entities.

```
… # Other imports.
from typing import Generic, TypeVar
CleanedDocumentT = TypeVar("CleanedDocumentT", bound=
ChunkT = TypeVar("ChunkT", bound=Chunk)
class ChunkingDataHandler(ABC, Generic[CleanedDocumen
    @property
    def chunk_size(self) -> int:
        return 500
    @property
    def chunk_overlap(self) -> int:
        return 50
    @abstractmethod
    def chunk(self, data_model: CleanedDocumentT) ->
        pass
```

Let's understand how the `ArticleChunkingHandler()` class is implemented. The `chunk_size` and `chunk_overlap` properties allow customization, so the other handlers are incredibly similar. Thus, we won't add them here, but they are available in the GitHub repository at llm_engineering/application/preprocessing.

The handler inputs cleaned article documents and returns a list of article chunk entities. It uses the `chunk_text()` function to split the cleaned content into chunks. The chunking function is customized based on the `chunk_size` and `chunk_overlap` attributes. The `chunk_id` is computed as the MD5 hash of the chunk's content. Thus, if the two chunks have precisely the same content, they will have the same ID, and we can easily deduplicate them. Lastly, we create a list of chunk entities and return them.

```
class ArticleChunkingHandler(ChunkingDataHandler):
    def chunk(self, data_model: CleanedArticleDocumen
        data_models_list = []
        cleaned_content = data_model.content
        chunks = chunk_text(cleaned_content, chunk_si

        for chunk in chunks:
            chunk_id = hashlib.md5(chunk.encode()).he
            model = ArticleChunk(
                id=UUID(chunk_id, version=4),
                content=chunk,
                platform=data_model.platform,
                link=data_model.link,
                document_id=data_model.id,
                author_id=data_model.author_id,
                author_full_name=data_model.author_fu
                metadata={
                    "chunk_size": self.chunk_size,
                    "chunk_overlap": self.chunk_overl
                },
            )
            data_models_list.append(model)
        return data_models_list
```

The last step is to dig into the `chunk_text()` function, which is a two-step process:

It uses a `RecursiveCharacterTextSplitter()` from LangChain to split the text based on a given separator or chunk size. Using the separator, we first try to find paragraphs in the given text, but if there are no paragraphs or they are too long, we cut it at a given chunk size.

Notice that we want to ensure that the chunk doesn't exceed the maximum input length of the embedding model. Thus, we pass all the chunks created above into a `SenteceTransformersTokenTextSplitter()`, which considers the maximum input length of the model. At this point, we also apply the `chunk_overlap` logic, as we want to do it only after we validate that the chunk is small enough.

```
… # Other imports.
from langchain.text_splitter import RecursiveCharacte
from llm_engineering.application.networks import Embe
def chunk_text(text: str, chunk_size: int = 500, chun
    character_splitter = RecursiveCharacterTextSplitt
    text_split_by_characters = character_splitter.spl
    token_splitter = SentenceTransformersTokenTextSpl
        chunk_overlap=chunk_overlap,
        tokens_per_chunk=embedding_model.max_input_le
        model_name=embedding_model.model_id,
    )
    chunks_by_tokens = []
    for section in text_split_by_characters:
        chunks_by_tokens.extend(token_splitter.split_
    return chunks_by_tokens
```

To conclude, the function above returns a list of chunks that respect both the provided chunk parameters and the embedding model's max input length.

The embedding handlers

The embedding handlers differ slightly from the others as the `EmbeddingDataHandler()` interface contains most of the logic. We took this approach because when calling the embedding model, we want to batch as many samples as possible to optimize the inference process. When running the model on a GPU, the batched samples are processed independently and in parallel. Thus, by batching the chunks, we can optimize the inference process by 10x or more, depending on the batch size and hardware we use.

We implemented an `embed()` method, in case you want to run the inference on a single data point, and an `embed_batch()` method. The `embed_batch()` method takes chunked documents as input, gathers their content into a list, passes them to the embedding model and maps the results to an embedded chunk domain entity. The mapping is done through the `map_model()` abstract method, which has to be customized for every data category.

```
… # Other imports.
from typing import Generic, TypeVar, cast
from llm_engineering.application.networks import Embe
ChunkT = TypeVar("ChunkT", bound=Chunk)
EmbeddedChunkT = TypeVar("EmbeddedChunkT", bound=Embe
embedding_model = EmbeddingModelSingleton()
```

```
class EmbeddingDataHandler(ABC, Generic[ChunkT, Embed
    """
    Abstract class for all embedding data handlers.
    All data transformations logic for the embedding
    """
    def embed(self, data_model: ChunkT) -> EmbeddedCh
        return self.embed_batch([data_model])[0]
    def embed_batch(self, data_model: list[ChunkT]) -
        embedding_model_input = [data_model.content f
        embeddings = embedding_model(embedding_model_
        embedded_chunk = [
            self.map_model(data_model, cast(list[floa
            for data_model, embedding in zip(data_mod
        ]
        return embedded_chunk
    @abstractmethod
    def map_model(self, data_model: ChunkT, embedding
        pass
```

Let's look only at the implementation of the `ArticleEmbeddingHandler()`, as the other handlers are highly similar. As you can see, we only have to implement the `map_model()` method, which takes a chunk of input and computes the embeddings in batch mode. Its scope is to map this information to an `EmbeddedArticleChunk` Pydantic entity.

```
class ArticleEmbeddingHandler(EmbeddingDataHandler):
    def map_model(self, data_model: ArticleChunk, emb
        return EmbeddedArticleChunk(
            id=data_model.id,
            content=data_model.content,
            embedding=embedding,
            platform=data_model.platform,
            link=data_model.link,
            document_id=data_model.document_id,
            author_id=data_model.author_id,
            author_full_name=data_model.author_full_n
            metadata={
                "embedding_model_id": embedding_model
                "embedding_size": embedding_model.emb
                "max_input_length": embedding_model.m
            },
        )
```

The last step is to understand how the `EmbeddingModelSingleton()` works. It is a wrapper over the `SentenceTransformer()` class from Sentence Transformers that initializes the embedding model. Writing a wrapper over external packages is often good practice. Thus, when you want to change the third-party tool, you have to modify only the internal logic of the wrapper instead of the whole code base.

The `SentenceTransformer()` class is initialized with the `model_id` defined in the `Settings` class, allowing us to quickly test multiple embedding models just by changing the configuration file and not the code. That is why I am not insisting at all on what embedding model to use. This differs constantly based on your use case, data, hardware, and latency. But by writing a generic class, which can quickly be configured, you can experiment with multiple embedding models until you find the best one for you.

```
from sentence_transformers.SentenceTransformer import
from llm_engineering.settings import settings
from .base import SingletonMeta
class EmbeddingModelSingleton(metaclass=SingletonMeta
    def __init__(
        self,
        model_id: str = settings.TEXT_EMBEDDING_MODEL
        device: str = settings.RAG_MODEL_DEVICE,
        cache_dir: Optional[Path] = None,
    ) -> None:
        self._model_id = model_id
        self._device = device
        self._model = SentenceTransformer(
            self._model_id,
            device=self._device,
            cache_folder=str(cache_dir) if cache_dir
        )
```

```
            self._model.eval()
    @property
    def model_id(self) -> str:
        return self._model_id
    @cached_property
    def embedding_size(self) -> int:
        dummy_embedding = self._model.encode("")
        return dummy_embedding.shape[0]
    @property
    def max_input_length(self) -> int:
        return self._model.max_seq_length
    @property
    def tokenizer(self) -> AutoTokenizer:
        return self._model.tokenizer
    def __call__(
        self, input_text: str | list[str], to_list: b
    ) -> NDArray[np.float32] | list[float] | list[lis
        try:
            embeddings = self._model.encode(input_tex
        except Exception:
            logger.error(f"Error generating embedding
            return [] if to_list else np.array([])
        if to_list:
            embeddings = embeddings.tolist()
        return embeddings
```

The embedding model class implements the Singleton pattern, a creational design pattern that ensures a class has only one instance while providing a global access point to this instance. The `EmbeddingModelSingleton()` class inherits from the `SingletonMeta` class, which ensures that whenever an `EmbeddingModelSingleton()` is instantiated, it returns the same instance. This works well with ML models, as you load them once in memory through the Singleton pattern, and afterward, you can use them anywhere in the codebase. Otherwise, you risk loading the model in memory every time you use it or loading it multiple times, resulting in memory issues. Also, this makes it very convenient to access properties such as the `embedding_size`, where you have to make a dummy forward pass into the embedding model to find the size of its output. As a Singleton, you do this forward pass only once, and then you have it accessible all the time during the program's execution.

Summary

This chapter began with a soft introduction to RAG and why and when you should use it. We also understood how embeddings and vector databases work, representing the cornerstone of any RAG system.

Then, we looked into advanced RAG and why we need it in the first place. We built a strong intuition about what parts of the RAG can be optimized and proposed some popular advanced RAG techniques for working with textual data.

Next, we applied everything we learned about RAG to designing the architecture of LLM Twin's RAG feature pipeline. We also understood the difference between a batch and streaming pipeline and presented a short introduction to the CDC pattern, which helps sync two databases.

Ultimately, we went step-by-step into the implementation of the LLM Twin's RAG feature pipeline, where we saw how to integrate ZenML as an orchestrator, how to design the domain entities of the application and how to implement an OVM module. Also, we understood how to apply some software engineering best practices, such as the abstract factory and strategy software patterns, to implement a modular and extendable layer that applies different cleaning, chunking, and embedding techniques based on the data category of each document.

This chapter focused only on implementing the ingestion pipeline, which is just one component of a standard RAG application. In Chapter 9, we will conclude the RAG system by implementing the retrieval and generation components and integrating them into the inference pipeline.