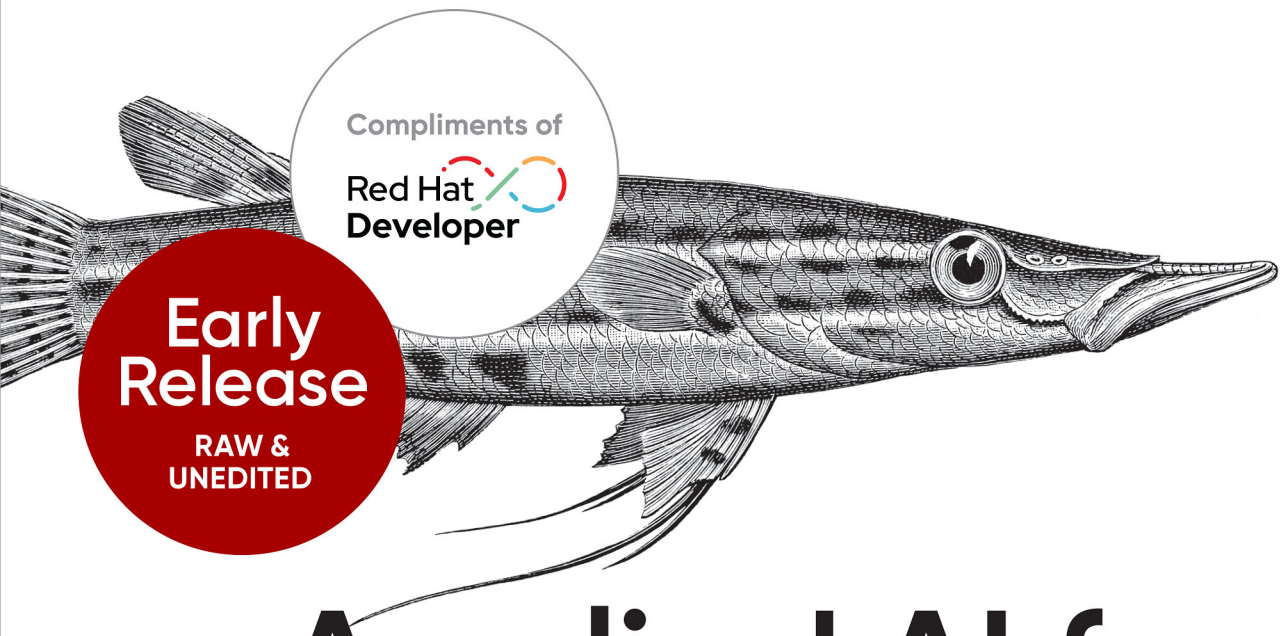
O'REILLY®

Compliments of

Red Hat Developer

Early Release

RAW & UNEDITED



Applied AI for Enterprise Java Development

Leveraging Generative AI, LLMs, and Machine Learning in the Java Enterprise

Alex Soto Bueno,
Markus Eisele & Natale Vinto

Applied AI for Enterprise Java Development

How to Successfully Leverage Generative AI, Large Language Models, and Machine Learning in the Java Enterprise

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Alex Soto Bueno, Markus Eisele, and Natale Vinto

O'REILLY®

Applied AI for Enterprise Java Development
by Alex Soto Bueno, Markus Eisele, and Natale Vinto

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Melissa Potter and Brian Guerin
Production Editor: Katherine Tozer
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2025: First Edition

Revision History for the Early Release

2024-10-30: First Release
2024-12-12: Second Release
2025-02-18: Third Release
2025-04-29: Fourth Release
2025-06-27: Fifth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098174507* for release details.

978-1-098-17444-6

[FILL IN]

[Alexandra i Ada] I was once like you are now and I know that it's not easy, to be calm when you've found something going on.
—Alex

To my children, whose future this technology will help shape. I'm grateful for the chance to share what I've learned — and hopeful for the world you'll create. —Markus

.
—Natale

Table of Contents

Brief Table of Contents (*Not Yet Final*)

Preface

Why We Wrote the Book

The demand for AI skills in the enterprise Java world is exploding, but let's face it: learning AI can be intimidating for Java developers. Many resources are too theoretical, focus heavily on data science, or rely on programming languages that are unfamiliar to enterprise environments. As seasoned programmers with years of experience in large-scale enterprise Java projects, we've faced the same challenges. When we started exploring AI and LLMs, we were frustrated by the lack of practical resources tailored to Java developers. Most materials seemed out of reach, buried under layers of Python code and abstract concepts.

That's why we wrote this book. It's the practical guide we wish we had, designed for Java developers who want to build real-world AI applications using the tools and frameworks they already know and love. Inside, you'll find clear explanations of essential AI techniques, hands-on examples, and real-world projects that will help you to integrate AI into your existing Java projects.

Who Should Read This Book

It is designed for developers who are interested in learning how to build systems that use AI and Deep Learning coupled with technologies they know and love around cloud native infrastructure and Java based applications and services. Developers like yourself, who are curious about the potential of Artificial Intelligence (AI) and specifically Deep Learning (DL) and of course Large Language Models. We do not only want to help you understand the basics but also give you the ability to apply core technologies and concepts to transform your projects into modern applications. Whether you're a seasoned developer or just starting out, this book will guide you through the process of applying AI concepts and techniques to real-world problems with concrete examples.

This book is perfect for:

- Java developers looking to expand their skill set into AI and machine learning
- IT professionals seeking to understand the practical implementation of the business value that AI promises to deliver

As the title already implies, we intend to keep this book practical and development centric. This book isn't a perfect fit but will still benefit:

- Business leaders and decision-makers. We focus on code and implementation details a lot. While the introductory chapters provides some context and introduce challenges, we will not talk a lot about business challenges.
- Data scientists and analysts. Developers could get some use out of our tuning approaches but won't need a complete overview of the data science theory behind the magic.

How the Book Is Organized

In this book, you'll gain a deeper understanding of how to apply AI techniques like machine learning (ML), natural language processing (NLP), and deep learning (DL) to solve real-world problems. Each chapter is designed to build your knowledge progressively, giving you the practical skills needed to apply AI within the Java ecosystem.

Chapter 1: The Enterprise AI Conundrum - Fundamentals of AI and Deep Learning
: We begin with the foundational concepts necessary for working on modern AI projects, focusing on the key principles of machine learning and deep learning. This chapter covers the minimal knowledge needed to collaborate effectively with Data Scientists and use AI frameworks. Think about it as building a comon taxonomy. We also provide a brief history of AI and DL, explaining their evolution and how they've shaped today's landscape. From here, we introduce how these techniques can be applied to real-world problems, touching on the importance and role of Open Source within the new world, the challenge of training data, and the side effects developers face when working with these data-driven models.

Chapter 2: The New Types of Applications - Generative AI and Language Models
: In this chapter, we explore the world of large language models. After a brief introduction to AI classifications, you'll get an overview of the most common taxonomies used to describe generative AI models. We'll dive into the mechanics of tuning models, including the differences between alignment tuning, prompt tuning, and prompt engineering. By the end of this chapter, you'll understand how to "query" models and apply different tuning strategies to get the results you need.

Chapter 3: Models: Serving, Inference, and Architectures - Architectural Concepts for AI-Infused Applications

Now that we have the basics in place, we move into the architectural aspects of AI applications. This chapter walks you through best practices for integrating AI into existing systems, focusing on modern enterprise architectures like APIs, microservices, and cloud-native applications. We'll start with a simple scenario and build out more complex solutions, adding one conceptual building block at a time.

Chapter 4: Public Models - Exploring AI Models and Model Serving Infrastructure

Chapter 4 introduces foundational concepts for AI-powered applications. It focuses on embedding vectors, vector stores, and their integration with augmented queries.. The chapter emphasizes running these capabilities locally for performance, cost, privacy, and offline requirements. This chapter lays the groundwork for hands-on implementations in subsequent chapters.

Chapter 5: Inference API - Inference and Querying AI Models with Java

We take a closer look at the process of "querying" AI models, often referred to as inference or asking a model to make a prediction. We introduce the standard APIs that allow you to perform inference and walk through practical Java examples that show how to integrate AI models seamlessly into your applications. By the end of this chapter, you'll be proficient in writing Java code that interacts with AI models to deliver real-time results.

Chapter 6: Accessing the Inference Model with Java - Building a Full Quarkus-Based AI Application

This hands-on chapter walks you through the creation of a full AI-infused application. You'll learn how to integrate a trained model into your application using both REST and gRPC protocols and explore testing strategies to ensure your AI components work as expected. By the end, you'll have your first functional AI-powered Java application.

Chapter 7: Introduction to LangChain4J

LangChain4J is a powerful library that simplifies the integration of large language models (LLMs) into Java applications. In this chapter, we introduce the core concepts of LangChain4J and explain its key abstractions.

Chapter 8: Image Processing - Stream-Based Processing for Images and Video

This chapter takes you through stream-based data processing, where you'll learn to work with complex data types like images and videos. We'll walk you through image manipulation algorithms and cover video processing techniques, including optical character recognition (OCR).

Chapter 9: Enterprise Use Cases

Chapter nine covers enterprise use cases. We'll discuss real life examples that we have seen and how they make use of either generative or predictive AI. It is a selection of experiences you can use to extend your problem solving toolbox with the help of AI.

Chapter 10: Architecture AI Patterns

In this final chapter, we shift focus from foundational concepts and basic implementations to the patterns and best practices you'll need for building AI applications that are robust, efficient, and production-ready. While the previous chapters provided clear, easy-to-follow examples, real-world AI deployments often require more sophisticated approaches to address the unique challenges that arise at scale which you will experience a selection of in this chapter.

Prerequisites and Software

While the first chapter introduces a lot of concepts that are likely not familiar to you yet, we'll dive into coding later on. For this, you need some software packages installed on your local machine. Make sure to download and install the following software:

- Java 17+ (*https://openjdk.java.net/projects/jdk/17/*)
- Maven 3.8+ (*https://maven.apache.org/download.cgi*)
- Podman Desktop v1.11.1+ (*https://podman-desktop.io/*)
- Podman Desktop AI lab extension

We are assuming that you'll run the examples from this book on your laptop and you have a solid understanding of Java already. The models we are going to work with are publicly accessible and we will help you download, install, and access them when we get to later chapters. If you have a GPU at hand, perfect. But it won't be neccessary for this book. Just make sure you have a reasonable amount of disc space available on your machine.

We would like to say thank you very much to our O'Reilly editors , for your trust and supporting us.

Also thank you very much to the book's technical reviewers: because you help us to make this book look perfect.

Alex Soto

Bit by bit, you'll end up the book, but as in movies, first the credits, I'd like to acknowledge Santa (Quan anem a jalar), Uri (Mercès pels peixos i les reines), Guiri

(Pugem en bici a Montserrat o no cal), Gavina, Gabi (thanks for the support); all my friends at the Red Hat and IBM developers team, we are the best.

Jonathan Vila, Abel Salgado, and Jordi Sola for the fantastic conversations to make the world better.

Last but certainly not least, I'd like to acknowledge Anita, "Was in the Winter, And Winter became the Spring, Who'd have believe you'd come along?"; my parents Mili and Ramon for buying my first computer; my daughters Ada and Alexandra, "sou les ninetes dels meus ulls."

Markus Eisele

Natale Vinto

CHAPTER 1
The Enterprise AI Conundrum

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

Artificial Intelligence (AI) has rapidly become an essential part of modern enterprise systems. We witness how it is reshaping industries and transforming the way businesses operate. And this includes the way Developers work with code. However, understanding the landscape of AI and its various classifications can be overwhelming, especially when trying to identify how it fits into the enterprise Java ecosystem and existing applications. In this chapter, we aim to provide you with a foundation by introducing the core concepts, methodologies, and terminologies that are critical to building AI-infused applications.

While the focus of this chapter is on setting the stage, it is not just about abstract definitions or acronyms. The upcoming sections will cover:

A Technical Perspective All the Way to Generative AI While large language models (LLMs) are getting most of the attention today, the field of artificial intelligence has a much longer history. Understanding how AI has developed over time is important when deciding how to use it in your projects. AI is not just about the latest trends

—it's about recognizing which technologies are reliable and ready for real-world applications. By learning about AI's background and how different approaches have evolved, you will be able to separate what is just hype from what is actually useful in your daily work. This will help you make smarter decisions when it comes to choosing AI solutions for your enterprise projects.

Open-Source Models and Training Data AI is only as good as the data it learns from. High-quality, relevant, and well-organized data is crucial to building AI systems that produce accurate and reliable results. In this chapter, you'll learn why using open-source models and data is a great advantage for your AI projects. The open-source community shares tools and resources that help everyone, including smaller companies, access the latest advancements in AI.

Ethical and Sustainability Considerations As AI becomes more common in business, it's important to think about the ethical and environmental impacts of using these technologies. Building AI systems that respect privacy, avoid bias, and are transparent in how they make decisions is becoming more and more important. And training large models requires significant computing power, which has an environmental impact. We'll introduce some of the key ethical principles you should keep in mind when building AI systems, along with the importance of designing AI in ways that are environmentally friendly.

The Lifecycle of LLMs and Ways to Influence Their Behavior If you've used AI chatbots or other tools that respond to your questions, you've interacted with large language models (LLMs). But these models don't just work by magic—they follow a lifecycle, from training to fine-tuning for specific tasks. In this chapter, we'll explain how LLMs are created and how you can influence their behavior. You'll learn the very basics about prompt tuning, prompt engineering, and alignment tuning, which are ways to guide a model's responses. By understanding how these models work, you'll be able to select the right technique for your projects.

DevOps vs. MLOps As AI becomes part of everyday software development, it's important to understand how traditional DevOps practices interact with machine learning operations (MLOps). DevOps focuses on the efficient development and deployment of software, while MLOps applies similar principles to the development and deployment of AI models. These two areas are increasingly connected, and development teams need to understand how they complement each other. We'll briefly outline the key similarities and differences between DevOps and MLOps, and show how both are necessary and interconnected to successfully deliver AI-powered applications.

Fundamental Terms AI comes with a lot of technical terms and abbreviations, and it can be easy to get lost in all the jargon. Throughout this book, we will introduce you to important AI terms in simple, clear language. From LLMs to MLOps, we'll explain everything in a way that's easy to understand and relevant to your projects. You'll also find a glossary at the end of the book that you can refer to whenever you need a

quick reminder. Understanding these basic terms will help you communicate with AI specialists and apply these concepts in your own Java development projects.

By the end of this chapter, you'll have a clearer understanding of the AI landscape and the fundamental principles. Let's begin by learning some basics and setting the stage for your journey into enterprise-level AI development.

Understanding the AI Landscape: A Technical Perspective all the way to Gen AI

Gen AI employs neural networks and deep learning algorithms to identify patterns within existing data, generating original content as a result. By analyzing large volumes of data, Gen AI algorithms synthesize knowledge to create novel text, images, audio, video, and other forms of output. The history of AI spans decades, marked by progress, occasional setbacks, and periodic breakthroughs. The individual disciplines and specializations can be thought of as a nested box system as shown in Figure 1-1. Foundational ideas in AI date back to the early 20th century, while classical AI emerged in the 1950s and gained traction in the following decades. Machine learning (ML) is a comparably new discipline which was created in the 1980s, involving training computer algorithms to learn patterns and make predictions based on data. The popularity of neural networks during this period was inspired by the structure and functioning of the human brain.

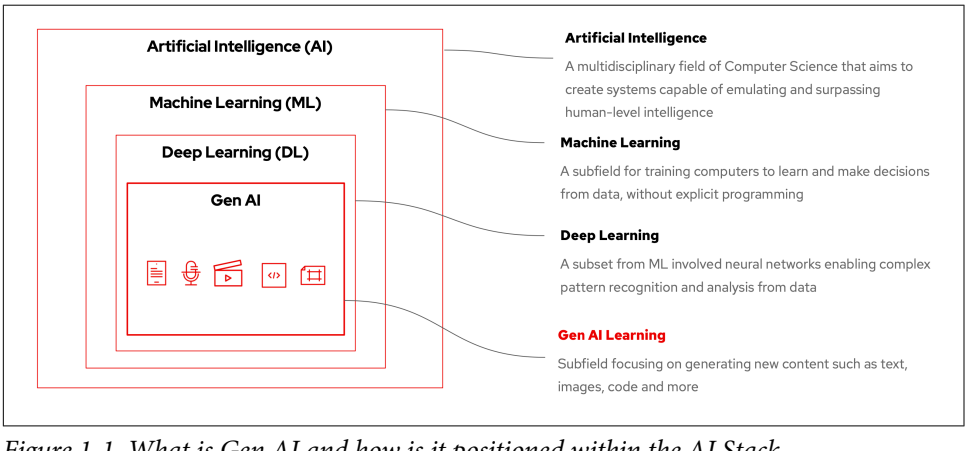


Figure 1-1. What is Gen AI and how is it positioned within the AI Stack.

What initially sounds like individual disciplines can be summarized under the general term Artificial Intelligence (AI). And AI itself is a multidisciplinary field within Computer Science that boldly strives to create systems capable of emulating and surpassing human-level intelligence. While traditional AI can be looked at as a mostly rule-based system the next evolution step is ML, which we'll dig into next.

Machine Learning (ML): The Foundation of today's AI

ML is the foundation of today's AI technology. It was the first approach that allowed computers to learn from data without the need to be explicitly programmed for every task. Instead of following predefined rules, ML algorithms can analyze patterns and relationships within large sets of data. This enables them to make decisions, classify objects, or predict outcomes based on what they've learned. The key idea behind ML is that it focuses on finding relationships between input data (features) and the results we want to predict (targets). This makes ML incredibly versatile, as it can be applied to a wide range of tasks, from recognizing images to predicting trends in data.

Machine Learning has far-reaching implications across various industries and domains. One prominent applications is Image Classification, where ML algorithms can be trained to identify objects, scenes, and actions from visual data. For instance, self-driving cars rely on image classification to detect pedestrians, roads, and obstacles.

Another application is Natural Language Processing (NLP), which enables computers to comprehend, generate, and process human language. NLP has numerous practical uses, such as chatbots that can engage in conversation, sentiment analysis for customer feedback, and machine translation for real-time language interpretation. Speech Recognition is another significant application of ML, allowing devices to transcribe spoken words into text. This technology has changed the way we interact with devices. It's early iterations brought us voice assistants like Siri, Google Assistant, and Alexa. Finally, Predictive Analytics uses ML to analyze data and forecast future outcomes. For example, healthcare providers use predictive analytics to identify high-risk patients and prevent complications, while financial institutions utilize this technology to predict stock market trends and make informed investment decisions.

Deep Learning: A Powerful Tool in the AI Arsenal

While it may have seemed like everyone was just interested in talking about LLMs, the basic theories of Machine Learning still made real progress in recent years. ML's progress was followed by Deep Learning (DL) which added another evolution to the artifical intelligence toolbox. As a subset of ML, DL involves the use of neural networks to analyze and learn from data, leveraging their unique ability to learn hierarchical representations of complex patterns. This allows DL algorithms to perform at tasks that require understanding and decision-making, such as image recognition, object detection, and segmentation in computer vision applications.

Many people assume that Machine Learning (ML) and Deep Learning (DL) are the same, but that is not quite accurate. Deep Learning is actually a subset of Machine Learning that focuses on models built with deep neural networks which are networks composed of many layers. While traditional ML algorithms can include decision trees, linear models, or shallow neural networks, DL specifically refers to architec-

tures that learn hierarchical representations through multiple layers of abstraction. This added complexity gives DL its unique ability to learn complex patterns and relationships in data. But what about the complexity itself? In most cases, DL algorithms are indeed more complex and with that computationally more expensive than ML algorithms. This is because they require larger amounts of data to train and validate models, whereas ML can often work with smaller datasets. And yet, despite these differences, both ML and DL have a wide range of applications across various fields - from image classification and speech recognition to predictive analytics and game playing AI. The key difference lies in their suitability for specific tasks: while ML is well-suited for more straightforward pattern recognition, DL shines when it comes to complex problems that require hierarchical representations of data. Machine Learning encompasses a broader range of techniques and algorithms, while Deep Learning specifically focuses on the use of neural networks to analyze and learn from data.

Generative AI: The Future of Content Generation

The advances in deep learning have laid the groundwork for Generative AI. Generative AI is all about generating new content, such as text, images, code, and more. This area has received the most attention in recent years mainly because of it's impressive demos and results around text generation and live chats. Generative AI is considered both a distinct research discipline and an application of deep learning (DL) techniques to create new behaviors. As a distinct research discipline, Gen AI integrates a wide range of techniques and approaches that focus on generating original content, such as text, images, audio, or videos. Researchers in this field explore various new methods for training models to generate coherent, realistic, and often creative outputs that get very close to perfectly mimic human-like behavior.

While generative AI has captured much of the recent attention, it is important to understand how it differs from predictive AI. Both fall under the umbrella of machine learning, but they serve different purposes. Predictive AI is focused on estimating outcomes based on historical data, while generative AI creates entirely new content based on learned patterns. The table below highlights some of the main distinctions:

Predictive AI	Generative AI
Makes predictions or classifications based on existing data	Generates new content that resembles training data
Examples: churn prediction, fraud detection, product recommendation	Examples: text generation, image synthesis, code completion
Outputs labels, probabilities, or numerical values	Outputs structured or unstructured content such as text or images
Often uses models like decision trees, logistic regression, or shallow neural networks	Relies on large-scale models such as transformers and diffusion models

Predictive AI	Generative AI
Evaluated based on accuracy or error rate	Evaluated based on creativity, coherence, or usefulness of output

Understanding this distinction is useful because it shapes how you integrate AI into your applications. Predictive AI often plays a supporting role in decision-making, while generative AI can directly shape user experiences through interaction and content creation.

At its center, Gen AI uses neural networks, enriching them with specialized architectures to further improve the results DL can already achive. For instance, convolutional neural networks (CNNs) are used for image synthesis, where complex patterns and textures are learned from unbelievably large datasets. This allows generative AI to produce almost photorealistic images that are closer to being indistinguishable from real-world counterparts than ever before. Similarly, recurrent neural networks (RNNs) are employed for language modeling, enabling GenAI to generate coherent and grammatically correct text. Think about it as a Siri 2.0. With the addition of transformer architectures for for text generation generative AI can efficently process sequential data and respond in almost real time. In particular the transformer architecture has changed the field of NLP and Large Language Models by introducing a more efficient and effective architecture for sequenzing tasks. The core innovation is the self-attention mechanism, which allows the model to capture specific parts of the input sequence simultaneously, enabling the model to capture long-range dependencies and context information. This is enhanced by an encoder-decoder architecture, where the encoder processes the input sequence and generates a contextualized representation, and the decoder generates the output sequence based on this representation.

Beyond neural networks, Gen AI also leverages generative adversarial networks (GANs) to create new data samples. GANs consist of two components: a generator network that produces new data samples and a discriminator network that evaluates the generated samples.

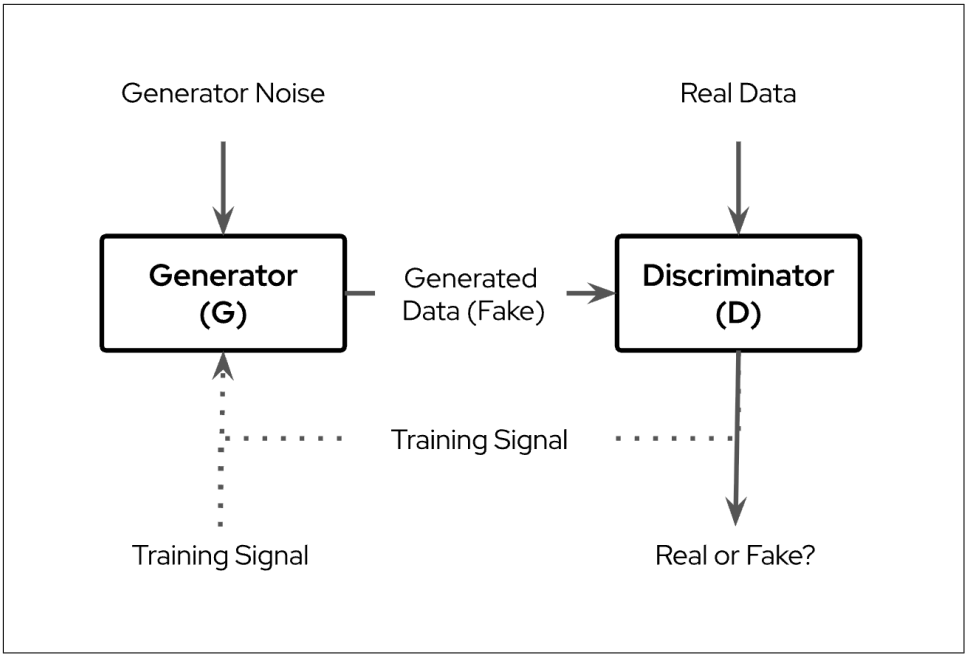


Figure 1-2. A GAN consists of two competing neural networks. The Generator tries to create data that looks real, while the Discriminator learns to tell real from fake. Over time, both networks improve, resulting in highly realistic outputs from the Generator.

This approach ensures that the generated data is not only realistic but also diverse and meaningful. Variational autoencoders (VAEs) are another type of DL model used by GenAI for image and audio generation. VAEs learn to compress and reconstruct data. This capability enables applications that generate high-quality audio samples simulating real-world sounds or even produce images that blend the styles of different artists. By combining DL techniques with new data chunking and transforming approaches, gen AI pushed applications a lot closer to being able to produce human like content.

Despite the advenanccements in research, the ongoing developments of more sophisticated computing hardware also significantly contributed to the visibility of generative AI. Namely Floating-Point Units (FPUs), Graphics Processing Unit (GPUs), and Tensor Processing Units (TPUs). A FPU excels at tasks like multiplying matrices, using specific math functions, and normalizing data. Matrix multiplication is a fundamental part of neural network calculations, and FPUs are designed to do this super fast. They also efficiently handle activation functions like sigmoid, tanh, and ReLU, which enables the execution of complex neural networks. Additionally, FPUs can perform normalization operations like batch normalization, helping to stabilize the learning process.

GPUs, originally designed for rendering graphics, have evolved into specialized processors that excel in machine learning tasks due to their unique architecture. By leveraging multiple cores they can process multiple tasks simultaneously, GPUs enable parallel processing capabilities that are particularly well-suited for handling large amounts of data. TPUs are custom-built ASICs (Application-Specific Integrated Circuits) specifically designed for accelerating machine learning and deep learning computations, particularly matrix multiplications and other deep learning operations. The speed and efficiency gains provided by FPUs, GPUs, and TPUs have a direct impact on the overall performance of machine learning models. Both for training but also for querying them.

One important consideration for developers is the practical challenge of running large language models (LLMs) on local machines. While local inference provides advantages such as better privacy, offline access, and faster iteration without relying on external APIs, these benefits come with trade-offs. LLMs are often large and can consume significant CPU, memory, and disk resources. This can make local experimentation difficult, especially on standard development laptops or desktops. However, recent advances in model quantization, containerized runtimes, and tools like Ollama or llama.cpp have made it more practical to run smaller or optimized models locally. These tools allow developers to explore and prototype with LLMs without requiring specialized hardware, though some setup and tuning may still be necessary depending on the use case. In later chapters, particularly ???, we will dive into model classification and explore strategies to overcome this issue. One such approach is model quantization, a technique that reduces the size and complexity of models by lowering the precision of the numbers used in calculations, without sacrificing too much accuracy. By quantizing models, you can reduce their memory footprint and computational load, making them more suitable for local testing and development, while still keeping them close enough to the performance you'd expect in production.

Open-Source Models and Training Data

One very important piece of the AI ecosystem is open source models. What you know and love from source code and libraries is something less comon in the world of machine learning but has been gaining a lot more attention lately.

Why Open Source is an Important Driver for Gen AI

A simplified view of AI models breaks them down into two main parts. First, there's a collection of mathematical functions, often called "layers," that are designed to solve specific problems. These layers process data and make predictions based on the input they receive. The second part involves adjusting these functions to work well with the training data. This adjustment happens through a process called "backpropagation,"

which helps the model find the best values for its functions. These values, known as "weights," are what allow the model to make accurate predictions. Once a model is trained, it consists of two main parts: the mathematical functions (the neural network itself) and the weights, which are the learned values that allow the model to make accurate predictions. Both the functions and the weights can be shared or published, much like source code in a traditional software project. However, sharing the training data (weights and functions) is less common, as it is often proprietary or sensitive. As you might imagine, open sourcing of the necessary amounts of data to train the most capable models out there is something not every vendor would want to do. Not only because it might cost the competetive advantage there are also speculations about the proper attribution and usage rights on some of the largest models out there. For the purpose of this book, we do use Open Source Models only. Not only because of the mostly hidden usage restrictions or legal limitations but also because we, the authors, believe that Open Source is an essential part of software development and the open source community is a great place to learn.

The Hidden Cost of Bad Data: Understanding Model Behavior Through Training Inputs

As you may have guessed, the training data is the ultimate factor that makes a model capable of generating specific features. If you train a model on legal paperwork, it will not be able to generate a good enough model for sport predictions. The domain and context of the training data is crucial for the success of a model. We'll talk about picking the right model for certain requirements and the selection process in chapter two, but note that it is generally important to understand the impact of data quality for training the models. Low-quality data can lead to a range of problems, including reduced accuracy, increased error rates, overfitting, underfitting, and biased outputs. Overfitting happens when a model learns the specific details of the training data so well that it fails to generalize to new, unseen data. This means that the model will perform very poorly on test or validation data, which is drawn from the same population as the training data but was not used during training.

In contrast to that, an underfitted model is like trying to fit a square peg into a round hole - it just doesn't match up with the true nature of the data. As a result, the model fails to accurately predict or classify new, unseen data. In this context, data that refers to information that is messy or contains errors is called "Noisy". It is making it harder for AI models to learn accurately. For example, if you're training a model to recognize images of cats, "noisy" data might include blurry pictures, mislabelled images, or photos that aren't even cats. This kind of incorrect or irrelevant data can confuse the model, leading it to make mistakes or give inaccurate results. In addition, data that is inconsistent, like missing values or using different formats for the same kind of information, can also cause problems. If the model doesn't have clean, reliable data to learn from, its performance will suffer, resulting in poor or biased predictions.

For instance, if an AI model is trained on data that includes biased or stereotypical information, it can end up making unfair decisions based on those biases, which could negatively impact people or groups.

You can mitigate these risks by prioritizing data quality from the beginning. This involves collecting high-quality data from the right sources, cleaning and preprocessing the data to remove noise, outliers, and inconsistencies, validating the data to ensure it meets required standards, and regularly updating and refining the model using new, high-quality data. And you may have guessed already, that this is something that developers should only rarely do but absolutely need to be aware of. Especially if they observe that their models are not performing as expected. A very simple example why this is relevant to you can be JSON processing for what is called "function calling" or "agent integration". While we will talk about this in Chapter 10 in more detail, you need to know that a model that has not been trained on JSON data, will not be able to generate it. This is a very common problem that developers face.

Adding Company specific Data to LLMs

Beyond the field of general purpose skills for large language models, there is also a growing need for task specific optimizations in certain applications. These can range from small scale edge scenarios with highly optimized models to larger scale enterprise level solutions. The most powerful feature for business applications is to add company specific data to the model. This allows it to learn more about the context of the problem at hand, which in turn improves its performance. What sounds like a job comparable to a database update is indeed more complex. There are different approaches to this which provide different benefits. We will look at training techniques that can be used for this later in chapter two when we talk about the classification of LLMs and will talk about architectural approaches in chapter three. For now it is essential to keep in mind, that there is no serious business application possible without proper integration of business relevant data into the AI-infused applications.

Explainable and Transparent AI Decisions

Another advantage of using open source models is that they can support the growing need for transparency in how AI systems are built and used. With access to the model architecture and sometimes even the training data, teams can better understand what the model has learned and how it might behave. This kind of openness can help companies build trust in the tools they use, especially in areas like healthcare, finance, and law enforcement where the impact of decisions can be serious.

However, it is important to understand that transparency is not the same as explainability. Just because a model is open source does not mean that it is easy to explain

how it arrived at a specific answer. The process that leads to a model's output is often complex and still very hard to trace, even for experts. Explainable AI is a separate field that works on this problem, but current techniques are still limited and do not always give clear answers.

Auditability is another related idea. This means being able to look back and understand how a model made a decision. It is not something that comes automatically with open source models. To achieve auditability, you need proper logging, input tracking, and clear processes for validating results. These are things developers and architects need to plan for.

There are also growing concerns about bias and unfair treatment in AI systems. Being able to review training data and model behavior is helpful, but it is not enough on its own. It still takes additional safeguards, validation steps, and human oversight to reduce risk and ensure fair use. We will come back to these topics in Chapter Three when we look at how to build responsible and trustworthy AI systems.

Ethical and Sustainability Considerations

While explainability of results is one part of the challenge, there are also a lot of ethical considerations. The most important thing to remember is that AI models are defined by the underlaying training data. This means that AI systems will always be biased towards the training data. This doesn't seem to carry a risk on first sight but there is a lot of potential for bias. For example, a model trained on racist comments might be biased towards white people. A model trained on political comments might be biased towards democrats or republicans. And these are just two obvious examples. AI models will reflect and reinforce societal biases present in the data they are trained on. The Unesco released reccomendations on AI ethics. This is a good starting point for understanding the potential biases that models might have.

But there are other thoughts that need to be taken into account when working with AI-infused applications. Energy consumption of large model deployments is dramatic, it is our duty as software architects and developers to pay close attention when executing and measureing sustainability of these systems. While there is a growing movement to direct AI usage towards good uses, towards the sustainable development goals, for example, it is important to address the sustainability of developing and using AI systems. A study by Strubell et al. illustrated that the process of training a single, deep learning, NLP model on GPUs can lead to approx. 300 tons of carbon dioxide emissions. This is roughly the equivalent of five cars over their lifetime. Other studies looked at Google's AlphaGo Zero, which generated almost 100 tons of CO2 over 40 days of training which is the equivalent of 1000 hours of air travel. In a time of global warming and commitment to reducing carbon emissions, it is essential to ask the question about whether using algorythms for simplistic tasks is really worth the cost.

The lifecycle of LLMs and ways to influence their behaviour

Now that you know a bit more about the history of AI and the major components of LLMs and how they are build, let's take a deeper look at the lifecycle of LLMs and how we can influence their behaviour as outlined in Figure 1-2.

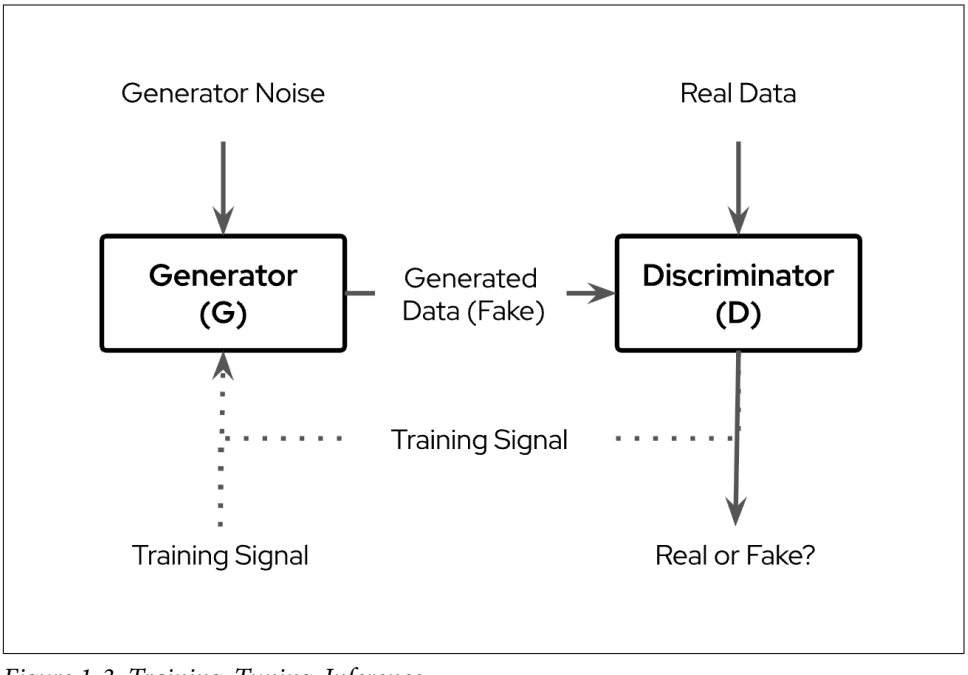


Figure 1-3. Training, Tuning, Inference.

You've already heard about training data, so it should come as no surprise that at the heart of the lifecycle lies something called the training phase. This is where LLMs are fed unbelievable amounts of data to learn from and adapt to. Once an LLM has been trained it is somewhat a general purpose model. Usually, those models are also refered to as foundation models. In particularly if we look at very large models like Llama3 example, their execution requires huge amounts of resources and they are generally exceptionally good at general purpose tasks. The next phase a model usually goes through is known as fine-tuning. This is where we adjust the model's parameters to optimize its performance on specific tasks or datasets. Through the process of hyperparameter tuning, model architects can fine-tune models for greater accuracy, efficiency, and scalability. This is generally called "hyperparameter optimization" and includes techniques like: grid search, random search, and Bayesian methods. We do not dive deeper into this in this book as both training and traditional fine-tuning are more a Data Scientist's realm. You can learn more about this in Natural Language

Processing with Transformers, Revised Edition. However we do cover two very specific and more developer relevant tuning methods in chapter Two. Most importantly prompt tuning and alignment tuning with InstructLab.

The last and probably most well known part of the lifecycle is inference, which is another word for "querying" a model. The word "inference" itself comes from the French word "inférence". In the context of LLMs, "inference" referres to the process of drawing conclusions from observations or premises. Which is a much more accurate description to what a model actually delivers. There are several ways to "query" a model and they can affect the quality and accuracy of the results, so it's important to understand the different approaches. One key aspect is how you structure your query, this is where prompt engineering comes into play. Prompt engineering involves crafting the input or question in a way that guides the model toward providing the most useful and relevant response. Another important concept is data enrichment, which refers to enhancing the data the model has access to during its processing. One powerful technique for this is RAG (Retrieval-Augmented Generation), where the model combines its internal knowledge with external, up-to-date information retrieved from a database or document source. In chapter Three, we will explore these techniques in more detail.

For now it is important to remember that models undergo a lifecycle within software projects. They are not static and should not be treated as such. While inferencing a model does not change model behavior in any way, models are only knowledable to the so called "cut of date" for their training data. If new information occurs or existing model "knowledge" needs to be changed, the weights ultimately will have to be adjusted. Either fine-tuned or re-trained. While this initially sounds like a responsiblity for a Data Science Team, it is not always possible to draw straight lines between the ultimate responsibilities of Data Science Team and the actual application developers. This book does draw a clear line though, as we do not cover training at all. We do however look in more detail into tuning techniques and inferencing architectures. But how do these teams work together in practice?

MLOps vs DevOps (and the Rise of AIOps and GenAIOps)

Two important terms have been coined during the last few years when we look at the way modern software-development and production setting is happening. The first is DevOps, a term coined in 2009 by Patrick Debois to refer to "development" and "operations". The second is Machine Learning Operations or MLOps initialy used by David Aronchick, in 2017. MLOps is a derived term and basically describes the application of DevOps principles to the Machine Learning field. The most ovious difference is the central artifact they are grouped around. The DevOps team is focused on business applications and the MLOps team is more focused on Machine

Learning models. Both describe the process of developing an artifact and making it ready for consumption in production.

DevOps and MLOps share many similarities, as both are focused on streamlining and automating workflows to ensure continuous integration (CI), continuous delivery (CD), and reliable deployment in production environments. Figure 1-3 describes one possible combination of DevOps and MLOps.

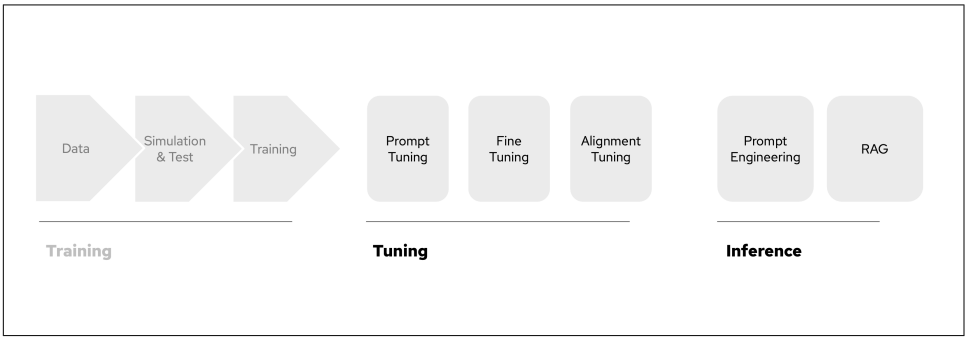


Figure 1-4. DevOps and MLOps

The shared practices, such as cross-functional collaboration, using Git as a single source of truth, repeatability, automation, security, and observability are at the core. Both DevOps and MLOps rely on collaboration between developers, data scientists, and operations teams to ensure that code, models, and configurations are well-coordinated. Automation and repeatability are emphasized for building, testing, and deploying both applications and models, ensuring consistent and reliable results. However, MLOps introduces additional layers, such as model training and data management, which are distinct from typical DevOps pipelines. The need to constantly monitor models for drift and ensure their performance over time adds complexity to MLOps, but both processes share a focus on security and observability to maintain trust and transparency in production systems.

As MLOps has matured, a broader set of terms has emerged, often used interchangeably or with overlapping meaning. These include ModelOps (focusing on model lifecycle management in a more general sense), LLMOps (specialized for large language model operations), and DataOps (emphasizing the reliability and automation of data pipelines). These terms reflect the increasing specialization in managing AI components at scale.

Adding to the landscape are AIOps and GenAIOps, which are also relevant in this context.

AIOps, short for Artificial Intelligence for IT Operations, refers to the use of machine learning and analytics to automate and enhance IT operational tasks. AIOps platforms ingest and analyze data from logs, metrics, and traces, helping operations

teams detect anomalies, predict outages, and reduce alert fatigue. While not focused on model deployment like MLOps, AIOps represents an important application of AI within production environments and often complements DevOps practices by improving infrastructure visibility and response times.

GenAIOps, or Generative AI Operations, is an emerging concept that adapts MLOps principles to the operational needs of generative AI systems, such as large language models. These systems pose new challenges, including prompt versioning, input/output validation, fine-tuning management, context construction, and guardrails. GenAIOps focuses on ensuring that generative AI components are governed, tested, deployed, and monitored just like any other critical application service, often using tools and workflows that DevOps and MLOps teams are already familiar with.

In practice, these disciplines are deeply intertwined and evolve based on organizational needs. Some teams may favor tight integration between software engineers and data scientists, using shared pipelines and infrastructure. Others may separate concerns between model development and production deployment, adopting modular and more controlled integration strategies.

In summary, while DevOps and MLOps share a common foundation, MLOps adds data and model-specific concerns. AIOps and GenAIOps further expand the operational scope of AI in production, targeting infrastructure optimization and generative model management, respectively. There is no single correct setup—each organization will find its balance based on its structure, expertise, and risk profile.

Conclusion

In this chapter, we explored the broader context of AI adoption in enterprise environments and what it means for developers. We began by examining the rise of generative AI and how it differs from traditional predictive models, introducing the core capabilities and limitations of modern large language models. We clarified the differences between predictive and generative approaches, highlighted why the distinction matters, and set the stage for thinking critically about when and how to use each.

We discussed the importance of data quality in shaping model behavior and explained why the source, structure, and cleanliness of training data can significantly affect outcomes. Concepts like overfitting, underfitting, and noise were introduced not just as theoretical ideas but as real challenges developers may encounter—even if they are not the ones training models themselves. The role of developers in selecting, integrating, and troubleshooting model behavior was emphasized throughout.

We also covered key terms that have emerged around AI operations, including DevOps, MLOps, and the newer areas of AIOps and GenAIOps. These helped place AI adoption within a familiar engineering context, showing how AI workflows can

fit into existing development and deployment practices. Understanding these terms is essential for developers who want to work effectively across teams and navigate evolving responsibilities.

Finally, we reflected on the value of open source in enterprise AI. While open source models support transparency and offer advantages in control and flexibility, we also clarified that transparency alone does not equal explainability. Concepts like auditability, bias mitigation, and regulatory compliance require their own tools and practices. Developers play a key role in implementing safeguards, validating outputs, and ensuring models behave responsibly in production.

The goal of this chapter was not to dive deep into technical implementation but to give you a grounded understanding of the enterprise AI landscape, the vocabulary that surrounds it, and the architectural concerns that will shape your work as a developer. In the chapters that follow, we will build on this foundation—starting with how to choose the right models for specific tasks and what it takes to make them usable in real-world software systems.

Chapter 2 will introduce you to various classifications of LLMs and unveil more of their inner workings. We'll provide an overview of the most common taxonomies used to describe these models. We will also dive into the mechanics of tuning these models, breaking down the differences between alignment tuning, prompt tuning, and prompt engineering.

CHAPTER 2
The New Types of Applications

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

Java developers have spent decades refining best practices for building scalable, maintainable, and performant applications. From enterprise web services to cloud-native microservices, the language and its ecosystem have been shaped by the needs of real-world applications. Now, with generative AI (GenAI) and other AI-infused capabilities, new types of applications are becoming more prominent and require additional knowledge, architecture, and tooling.

We hope that you already understand that GenAI is not a radical break from past advancements but rather an evolution of AI research in deep learning combined with the foundations of software engineering. Just as Java developers have adapted to the shift from monoliths to microservices and from imperative to reactive programming, they now face the challenge of integrating AI models into their applications in a way that aligns with the principles they already know: modularity, scalability, testability, and maintainability.

To effectively use AI in Java applications, an understanding of the fundamental components that make these systems work is not only helpful but necessary. Because of the complexity and novelty of some of these components, we decided to break it down and peel layers back individually over chapters. In this chapter, we will break down the key aspects of AI integration:

Understanding Large Language Models
: LLMs are a special class of AI models trained on vast amounts of text data to perform natural language processing tasks. We will explore how they generate responses, their limitations, and introduce you to more relevant details to be able to classify models and use the right models for your requirements.

Understanding Model Types
: Not all AI models are created equal. While generative models like large language models (LLMs) and diffusion models capture the most attention, they are just one part of the AI landscape. We will explore different types of models, including classifiers, embeddings, and retrieval-augmented generation (RAG), and how they map to real-world application needs.

Supporting Technologies
: AI models do not run in isolation; they rely on a rich ecosystem of tools and frameworks. From vector databases that store and retrieve knowledge efficiently to APIs that expose models as services, understanding the AI stack is crucial for Java developers who want to build applications that are both powerful and maintainable.

Teaching Models New Tricks
: Unlike traditional software, AI-infused applications improve through different means: fine-tuning, prompt engineering, retrieval augmentation, and reinforcement learning. We'll discuss these techniques and their trade-offs, particularly in enterprise environments where control and customization are key.

It sounds like a lot of ground to cover but we promise to cut down where possible and equip you with the most basic knowledge.

Understanding Large Language Models

As a Java developer, you might be used to working with structured data, type-safe environments, and explicit control over program execution. Large Language Models (LLMs) operate in a completely different way. Instead of executing pre-defined instructions like a Java method, they generate responses probabilistically based on learned patterns. You can think of an LLM as a powerful autocomplete function on steroids—one that doesn't just predict the next character but understands the broader context of entire conversations.

If you've ever worked with compilers, you know that source code is transformed into an intermediate representation before execution. Similarly, LLMs don't process raw text directly; instead, they convert it into numerical representations that make computations efficient. You can compare this to Java bytecode—while human-readable Java code is structured and understandable, it's the compiled bytecode that the JVM actually executes. In an LLM, tokenization plays a similar role: it translates human language into a numerical format that the model can work with.

Another useful comparison is how Java Virtual Machines (JVMs) manage Just-In-Time (JIT) compilation. A JIT compiler dynamically optimizes code at runtime based on execution patterns. Similarly, LLMs adjust their text generation dynamically, predicting words based on probability distributions instead of following a hardcoded set of rules. This probabilistic nature allows them to be flexible and creative but also means they can sometimes produce unexpected or incomplete results. Now, let's break down their components, starting with the key elements.

Key Elements of a Large Language Model

LLMs rely on several foundational elements that define their effectiveness and applicability. While their training data-model is important there are other elements that also play their roles. For instance, attention mechanisms allow models to weigh the importance of different words in a sequence, while tokenization strategies determine how efficiently input is processed. Additionally, factors like context length, memory constraints, and computational efficiency decide how well an LLM can handle complex prompts and interactions. A base understanding of these core components is necessary to successfully integrate these new features into applications because they influence performance, scalability, and overall user experience.

How LLMs Generate Responses

At a high level, LLMs process text input (Natural Language Understanding, NLU) and generate meaningful responses (Natural Language Generation, NLG). There are different acronyms and terms you need to know in order to understand model use-cases and decide which model to use for your specific application.

Natural Language Understanding (NLU)
: focuses on interpreting and analyzing human input. It is meant for tasks like intent recognition, entity extraction, text classification, and sentiment analysis. This is conceptually similar to how Java applications parse JSON or XML, extracting key data for business logic. If you are building an AI-powered search or recommendation system, an encoder-based model (e.g., BERT) optimized for NLU is typically good fit.

Natural Language Generation (NLG)
: is responsible for constructing meaningful and coherent responses. This is useful for chatbots, report generation, and text summarization. Conceptually, this mirrors Java's templating engines (e.g., Thymeleaf, Freemarker) that dynamically generate output based on structured input. Decoder-based models (e.g., GPT) are more suited for these tasks.

Tokenization
: Before processing, LLMs break input and output into smaller chunks (tokens), similar to how Java tokenizes strings using StringTokenizer or regex. Token limits affect how much context a model can "remember" in a single request. When outputting tokens the model adds a degree of randomness injecting a non-deterministic behaviour. This degree of randomness is added to simulate the process of creative thinking and it can be tuned using a model parameter called temperature.

Self-Attention Mechanism Mechanisms (Transformer Architecture)
: Large Language Models (LLMs) built on transformers use a self-attention mechanism to determine which words (tokens) matter most within a sentence. Instead of treating each word equally, the model assigns higher importance—or "attention"—to key words based on their relevance to the overall meaning. You can think of it as similar to dependency resolution in Java build tools: when Maven or Gradle resolves conflicting library versions, it actively selects the most critical dependency based on scopes or position in the dependency tree. Just as certain dependencies take precedence in Java builds, specific tokens receive more attention within transformer models.

Context Windows
: The context window is analogous to a buffer size or a stack frame in Java. Just as a method in Java has a limited stack frame size to store local variables, an LLM has a fixed memory space to store input and output tokens. For example, an LLM with a 4K-token context window (such as GPT-3.5) can process roughly 3,000 words at a time before discarding older tokens. Larger models (e.g., GPT-4 Turbo, Claude 3 Opus) support 128K+ tokens, which allow for much longer interactions without losing past context.

You've read about the basic terms now but there is more to know about how models work. The most important part is the underlaying model architecture. You don't need to remember all of this for now. We just don't want you to be surprised when we use certain descriptions later on. Treat the following overview as a place to revisit when you stumble over something later in the book.

Model Architectures

Just as Java frameworks are designed for specific workloads (Quarkus for microservices, Lucene for search, and Jackson for JSON processing) different types of AI models are optimized for specific use cases. LLMs generally fall into three categories: encoder-only, decoder-only, and encoder-decoder models, each with unique characteristics.

Encoder-only models. Such as BERT, RoBERTa, and E5, are designed for understanding text rather than generating it. These models process entire inputs at once, extracting semantic meaning and relationships between words. They are widely used in retrieval-augmented generation (RAG) pipelines, where their ability to generate vector embeddings enables semantic search in vector databases like Weaviate, Pinecone, and FAISS. By converting text into numerical representations, these models enhance enterprise search by retrieving relevant documents based on meaning rather than keywords. You can integrate encoder models with traditional Lucene-based search engines to combine lexical and semantic retrieval techniques, improving the accuracy and relevance of search results.

Beyond search, encoder models are also valuable in classification tasks, such as intent recognition for chatbots, spam detection, and fraud analysis. They enable named entity recognition (NER) and information extraction, making them useful in document processing applications where structured data must be extracted from legal, financial, or compliance-related texts. In recommendation systems, these models generate embeddings that help match users with relevant articles, documentation, or products, improving personalization. Security applications also benefit from encoder models, as they can classify logs and detect anomalies in system monitoring and fraud prevention workflows.

Decoder-only models. Such as GPT, LLaMA, and Mistral, focus on text generation. Unlike encoders, which analyze entire inputs at once, decoders generate text one token at a time, predicting the next word based on prior context. This makes them ideal for chatbots and conversational AI, where dynamic, context-aware responses are necessary. Java applications integrating AI-powered customer support can use decoder models to generate replies, assist agents with suggested responses, and provide automated insights. In software development, decoder models are widely used in code generation and auto-completion, helping developers by predicting Java code snippets, completing function calls, and even explaining complex code in natural language. Java-based enterprise applications can also leverage these models for report generation and content automation, creating summaries, legal documents, and personalized customer communications. In text rewriting and summarization, decoder models can be applied to simplify, paraphrase, or expand content dynamically, enhancing content creation workflows.

Encoder-decoder models. Such as T5, BART, and FLAN-T5, combine the strengths of both architectures, making them particularly effective for structured input-to-output transformations. Unlike decoder-only models that generate text sequentially, encoder-decoder models first process input using an encoder, then generate structured output using a decoder. This design is well-suited for machine translation, enabling Java applications to support multilingual users by translating UI elements, emails, and user-generated content in real time. Documentation localization is another practical use case, allowing businesses to translate software manuals and API documentation efficiently. In text summarization, these models extract key information from large documents, such as legal contracts, financial reports, or monitoring logs, making complex information easier to review. Java developers working with knowledge management systems can use encoder-decoder models to refine, paraphrase, and restructure content, ensuring clarity and consistency in enterprise communications.

Recent advancements in LLM architectures focus on improving efficiency without sacrificing performance. Techniques such as Mixture of Experts (MoE), used in models like GPT-4.5 and Gemini 2.5, selectively activate only a portion of the model parameters during inference, reducing computational overhead while maintaining high accuracy. This approach is conceptually similar to lazy-loading mechanisms in Java frameworks, where resources are only loaded when needed. Quantization and model distillation allow developers to deploy smaller, resource-efficient versions of large models without significant loss of accuracy, much like JVM optimizations that improve runtime performance. Emerging memory-efficient techniques, such as flash attention and sparse computation, further reduce hardware costs, akin to Java's use of memory-mapped files for optimizing performance in high-throughput applications.

Selecting the right model depends on the specific needs of an application. Java developers integrating semantic search in a RAG pipeline will benefit most from encoder-only models like BERT or E5. Applications requiring chat-based interactions, code suggestions, or dynamic content generation are best suited for decoder-only models such as GPT or LLaMA. For tasks involving machine translation, structured document transformation, and summarization, encoder-decoder models like T5 or FLAN-T5 provide the best results. Understanding these architectures allows developers to make informed decisions, balancing accuracy, efficiency, and cost while integrating AI into enterprise Java applications.

Size and Complexity

Large Language Models (LLMs) come in a variety of sizes, typically measured by the number of parameters. Parameters are basically the internal numerical values that define how well a model can predict and generate text. Just as a Java developer carefully selects the right database, caching strategy, or framework to balance per-

formance and scalability, choosing the right LLM size ensures efficient inference, cost-effectiveness, and deployment feasibility.

Smaller models, generally in the 7 billion to 13 billion parameter range (e.g., Mistral 7B, TinyLlama), are optimized for local execution and require minimal computational resources. These models are well-suited for applications that need low-latency responses, such as edge AI, embedded systems, or lightweight chatbot applications. Running such a model locally is comparable to using an embedded database like SQLite—it is efficient, self-contained, and practical for single-user workloads.

Medium-sized models, ranging from 30 billion to 65 billion parameters (e.g., LLaMA 65B), provide better contextual awareness and accuracy but demand higher memory and GPU resources. They are ideal for server-side deployment in enterprise applications, powering AI-driven customer service bots, enterprise search, document summarization, and intelligent automation tools. Their infrastructure footprint is similar to managing a Redis caching layer or a lightweight microservice cluster, where performance optimization is essential to avoid excessive resource consumption.

What's in a Name - Model Naming

Model names often include suffixes that signal their intended use case or level of optimization. "Base" models are unmodified foundation models trained on large-scale data sets without specific fine-tuning. "Instruct" or "Chat" variants are adapted for interactive conversation tasks, making them ideal for chatbot development. "Code" models are fine-tuned on programming languages, making them useful for code completion, bug fixing, and AI-assisted software development. Other common suffixes like "QA" (question-answering) and "RAG-ready" (Retrieval-Augmented Generation optimized) indicate models specifically tuned for enterprise knowledge retrieval and document-based AI workflows.

At the highest tier, large-scale models exceeding 175 billion parameters, such as GPT-4, Claude 3, and Gemini Ultra, require specialized hardware and distributed inference. These models deliver superior contextual reasoning, multi-turn conversation capabilities, and complex problem-solving. However, the infrastructure demands are immense, requiring cloud-based inference solutions due to their size and energy consumption. Using these models is akin to operating a distributed system like Apache Kafka or Elasticsearch, where scalability and resource allocation are primary concerns. Most Java developers interacting with large models will do so via cloud APIs, integrating them into applications without the need for direct infrastructure management.

Wait: What does "7 Billion Parameters" even mean here?. When we say that Mistral 7B has 7 billion parameters, we are referring to the total number of trainable weights that

define the model's behavior. These parameters are stored in tensors. These parameters define how a model processes input data and generates output, similar to how Java developers configure class variables and constants that dictate an application's behavior. In mathematical terms, an LLM is essentially a giant function with billions of parameters, and these parameters exist as multi-dimensional tensors. A simple analogy would be how Java handles matrices using multi-dimensional arrays. Suppose we have a Java program for image processing that uses a 3D array to represent an RGB image:

```
int[][][] image = new int[256][256][3]; // A 256x256 image with 3 color channels
```

In deep learning, tensors work similarly but at a much larger scale. A single LLM layer could have weight tensors shaped like [12288, 4096], meaning it has 12,288 input features and 4,096 output features. This is much like a huge adjacency matrix, where each weight value determines how one input influences an output. Working with pre-trained LLMs means working with tensor weights stored in formats like Safetensors or GGUF (more on this later). These formats efficiently load precomputed parameters into memory, similar to how Java loads compiled bytecode into the JVM for execution.

And tensors come in different precisions. While a model's parameter count defines its capacity for reasoning, contextual depth, and overall accuracy, the tensor type determines how efficiently those parameters are stored, loaded, and processed. The higher the precision of the tensor type, the more memory and computational power is required per parameter. On the opposite, lower-precision tensors allow for compression and faster execution, enabling larger models to run on smaller hardware. They come in full-precision (FP32 or FP16) and in quantized (INT8, INT4) versions. For large-scale models exceeding 175B parameters, full-precision inference is only available on massivly distributed systems. Think of how a distributed databases partition and processes large datasets. For smaller or local deployments INT8 or INT4 quantization reduces memory footprint and still keeps functional accuracy.

Optimizing Model Size with Quantization and Compression. Going from one precision to another is something you can think of as optimizing JVM memory usage and garbage collection settings to improve application performance. For LLMs the techniques used are called quantization and compression. Quantization reduces the precision of model weights, typically from 32-bit floating point (FP32) to 16-bit (FP16), 8-bit (INT8), or even 4-bit (INT4) representations.

Compression techniques such as weight pruning and distillation further reduce model size. Weight pruning removes less critical parameters, effectively shrinking the model while maintaining most of its predictive capabilities. Distillation, on the other hand, involves training a smaller "student" model to mimic a larger "teacher" model, capturing its behavior while being far more efficient. Think of it as something

similar to JIT optimizations in the JVM or the use of compressed indexes in search engines, where efficiency is achieved without sacrificing too much accuracy.

In summary, understanding how parameters and the derived precision helps optimize performance and hardware requirements:

Memory Considerations
: The 7B parameters must be loaded into GPU VRAM or RAM. Using FP16 tensors instead of FP32 reduces memory usage by half.

Inference Speed
: Larger models require more tensor computations per token generated. Using quantized INT8 or INT4 models reduces processing time at the cost of slight accuracy loss.

Context Windows
: More context (longer input prompts) means more activations, increasing VRAM usage. A 4K token context consumes significantly more memory than a 1K token context.

Deploying and Models

So far, we have explored the inner workings of transformer models to give you a foundational understanding of how they function and what the common terms and acronyms mean in this domain. The world of creating, training, and serving these models is heavily centered around Python, with very little involvement from Java. While there are exceptions, such as TensorFlow for Java, most tooling and frameworks are designed with Python in mind.

Deploying a LLM involves several steps. First, the model must be exported in a format compatible with an inference engine, which handles loading the model weights, optimizing execution, and managing resources like GPU memory. Unlike traditional Java applications AI models are packaged in formats such as ONNX, GGUF, or Safetensors. Each is designed for different execution environments. Choosing an inference engine determines how efficiently the model runs, what hardware it supports, and how well it integrates with existing applications. While Java Developers typically do not make these choices, they will have to help formulate the non functional requirements that can help with making the right choice as it directly affects factors such as latency, throughput. All needing to align with your application's requirements.

In modern cloud-native architectures, inference engines are typically accessed as services deployed either on-premise, in the cloud, or within containerized environments. Java applications interact with these services through REST APIs or gRPC to send input data and receive model predictions. This aligns with scalable, service-based architectures, where AI models are deployed as independent services that can be load-balanced and auto-scaled like other application components. Many cloud

hosted offerings such as OpenAI, Hugging Face, and cloud inference endpoints from AWS, Azure, and Google Cloud, expose standardized APIs that allow Java applications to integrate seamlessly without needing direct model deployment. More about Inference APIs and selected providers in chapter five.

For self-hosted or on-device models, Java can leverage native bindings via JNI (Java Native Interface) or JNA (Java Native Access) to directly invoke inference engines like llama.cpp or ONNX Runtime without needing external services. For scenarios requiring low-latency, on-device inference, Java applications can integrate with frameworks like Deep Java Library (DJL), which provides high-level APIs to load and execute models directly on supported hardware.

With all that in mind, let's look at some of the most popular inference engines for LLM deployment, their capabilities, and how you can access them through Java.

- vLLM + vLLM is an inference engine optimized for high-throughput, low-latency LLM serving. It features PagedAttention, an efficient memory management technique that significantly improves batch processing, streaming token generation, and GPU memory efficiency. + Java Integration: OpenAI-compatible API server
- TensorRT + NVIDIA's TensorRT is an SDK for running LLMs on NVIDIA GPUs. It offers inference, graph optimizations, quantization support (FP8, INT8, INT4), and more. + Java Integration: TensorRT uses the Triton Inference Server which offers several client libraries and examples of how to use those libraries.
- ONNX Runtime + ONNX Runtime provides optimized inference for models converted into the ONNX format, enabling cross-platform execution on CPU, GPU, and specialized AI accelerators. + Java Integration: Native Java bindings
- llama.cpp + llama.cpp is an inference engine designed to run quantized models (GGUF format) on standard hardware without requiring a GPU. It is one of the most common options for self-hosting an LLM on a local machine or deploying it on edge devices. + Java Integration: Can be accessed via JNI (Java Native Interface) bindings or a REST API wrapper.
- OpenVINO + is an inference engine designed by Intel to optimize AI workloads on Intel specific processors. + Java Integration: JNI bindings and REST API wrapper.
- RamaLama + RamaLama tool facilitates local management and serving of AI Models from OCI images. + Java Integration: uses llama.cpp REST API endpoints.
- Ollama + Ollama Server and various tools to run models on local hardware. + Java Integration: Either native Java Client library or REST API endpoints.

When running models and inference engines locally, containerization simplifies packaging the runtime, libraries, and optimizations into a single environment. Some inference engines already provide pre-built containers. Tools like Podman further streamline management of these containers, including the ability to pull model images or create custom containers for your specific hardware. Podman Desktop provides a user interface for easily spinning up and testing these AI services. We will take a closer look at how to use Podman Desktop in chapter ???.

But there is even more. While the technology advances quickly there are more ways to access models through specialized ways, as outlined in the list below. And it does not look like the options will be slowing down anytime soon.

- Cloud-Native Serving + Cloud platforms like AWS, Google Cloud, and Azure offer fully managed model serving solutions. You can deploy models through their respective marketplaces or tooling, often with automatic scaling and built-in monitoring. This reduces operational overhead but may introduce vendor lock-in.
- Edge AI + Deploying models at the edge—on IoT devices or local gateways—reduces latency and network usage. Frameworks geared for edge AI often include optimizations for low-power hardware, making it viable for real-time or mission-critical scenarios in remote locations.
- Model Registry + Model registries help you store and organize versions of your trained models. Popular services like Hugging Face Model Hub allow you to discover, share, or fine-tune community models, aiding reproducibility and easy updates.
- Knative Serving + Knative Serving is a serverless framework for Kubernetes. It automates scaling, deployment, and versioning for containerized workloads, which can simplify hosting AI inference services alongside other cloud-native applications in a unified environment.

You now have a good overeview about the model inner workings and how they can be served. We are now pivoting to the more delicate tweaks that can be made to models.

Key Hyperparameters for Model Inference

We've already talked about model parameters parameters, but there's more we should discuss: So called hyperparameters help by optimizing inference speed, response quality, and memory efficiency. While parameters (weights) define the model's learned knowledge, hyperparameters control inference behavior, allowing developers to fine-tune the creativity, accuracy, and efficiency of model responses. Many of these can be changed via API calls or Java APIs. You should experiment with hyper-parameter tuning to achieve the best results for your use cases. It's a great way to

change between precise, deterministic output or creative, open-ended responses. The following list contains the most comon Hyperparameters:

- Temperature: Controls the randomness of text generation
  - — Low values (0.2–0.5) result in deterministic, factual responses.
    - — Temperature = 0.2 - "Java garbage collection manages memory automatically."
  - — High values (0.7–1.2) push more creative, diverse outputs.
    - — Temperature = 1.0 - "Java's garbage collection is like an unseen janitor, tidying memory dynamically."
- Top-k Sampling: Limits the number of token choices to the top-k most probable tokens.
  - — A lower k results in more deterministic responses, while a higher k adds variability.
- Top-p Sampling (or Nucleus Sampling): Chooses tokens from the top p% of probability mass.
  - — Helps generate more natural-sounding responses by adjusting sampling dynamically.
- Repetition Penalty: Penalizes or reduces the probability of generating tokens that have recently appeared.
  - — Encourages the model to generate more diverse and non-repetitive output.
- Context Length: Defines how many tokens the model remembers in a single request.
  - — Short context (4K tokens) - Fast, but forgets earlier parts of a conversation.
  - — Long context (128K tokens) - Better recall, higher computational cost.

Make sure to check your API documentation to confirm which hyperparameters are supported, if any.

Model Tuning - Beyond tweaking the output

We've talked about models, adapters and tuning so far, but we've tried to avoid overloading you with knowledge that isn't directly applicable for working with models. However, sometimes just using an existing model and slightly tweaking its inputs and outputs isn't enough and you can't find a specific model for your use-case. This is when you have to look into other ways to create a specific adaptation or even create a new model. We want to make sure you understand the various ways to influence model behaviour in terms of complexity and invasiveness, to give you a better understanding of what you can probably do yourself and when you need help

from a data scientist. We do not cover all the details here, as most of them are clear Data Scientist specialties, but want to mention them for completeness. You can learn more about this in the excellent O'Reilly book AI Engineering by Chip Huyen. Tuning in the traditional sense changes the models weights and adapts a pre-trained model to a specific need. But there are many ways to change model behaviour without changing the pre-trained model but still changing the inner workings of the model. This approach, known as model adapters, allows the base model to retain its general knowledge while the adapter layers adds task-specific knowledge on top.

Common adapter techniques include:

LoRA (Low-Rank Adaptation)
: LoRA inserts small trainable layers into existing transformer weights rather than modifying the entire model.

PEFT (Parameter-Efficient Fine-Tuning)
: PEFT encompasses various adapter techniques, including LoRA, Prefix Tuning, and Adapter Layers, to fine-tune models efficiently while keeping most parameters unchanged.

Prefix Tuning & Prompt Tuning
: These methods add trainable prefixes to input prompts rather than modifying model weights, allowing task-specific customization closer to the model. Think of them as system-prompts that are build in.

Adapter models can be integrated via different techniques in inference engines and effectively layered on top of existing models. Think of it as additional layers of a container. Adapters are comonly referenced in the model name and their documentation explains for what application they are adapted for.

When the data scientist community talks about fine-tuning they can refer to different things with different complexities and cost implications. The following image gives you an overview about the different approaches and we rank them by effort and their usefulnes for certain scenarios shown in Figure 2-1.

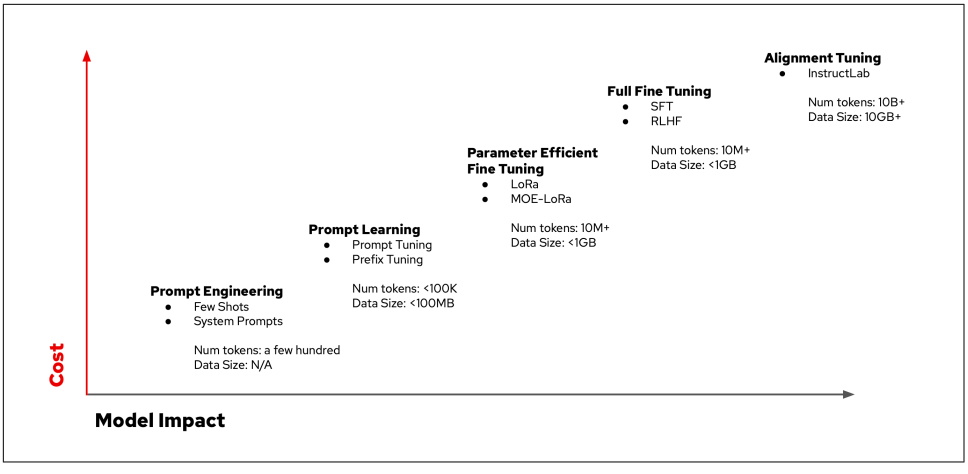


Figure 2-1. Common Tuning Techniques applied to LLMs

Let's take a look at each of these in more detail.

Prompt Tuning or Engineering. Prompt tuning is like optimizing SQL queries or tweaking configuration files, where adjustments to inputs improve overall performance without modifying the underlying system. It differs from prompt engineering, which focuses more on crafting better prompts without systematic experimentation. Prompt tuning systematically refines input patterns and embeddings so the model produces more desirable outputs. Applications or automated systems typically implement it using structured templates that modify prompts based on user interactions. This allows developers to guide model behavior with minimal overhead, making it an accessible and cost-effective approach to improving AI responses. However, unlike fine-tuning, prompt tuning does not alter the model's internal parameters, which means its effectiveness is constrained by the base model's existing capabilities.

Prompt Learning. Prompt learning extends prompt tuning by training a model with structured input-output prompt pairs. This enables the model to refine its responses based on structured examples rather than simple trial and error. This is typically done through fine-tuning methods where labeled examples guide the model's learning, helping it adjust to specific patterns or desired behaviors. One effective approach is using Low-Rank Adaptation (LoRA), which allows adjustments to be made without retraining the entire model, making it more efficient and resource-friendly. This method is mainly used in applications that require predefined response structures. Good examples are enforcement of compliance in AI-generated text, maintaining consistency in customer support interactions, or applying business logic guardrails. It can roughly be compared to writing test-driven development (TDD) tests where expected inputs and outputs help refine software behavior iteratively, ensuring

predictable and improved model performance over time. Unlike using pre-trained adapters, this will have to be executed by Data Scientists.

Parameter Efficient Fine Tuning (PEFT and LoRA). As discussed earlier, model adapters allow developers to modify specific behaviors without retraining the entire model, making this approach much more accessible.

Full Fine Tuning. Full fine-tuning means adjusting all model weights by retraining it on domain-specific data, requiring specialized knowledge and significant computational resources. This process is akin to recompiling an entire application with a new framework version instead of just upgrading a single dependency. Unlike lighter tuning methods, full fine-tuning demands expertise in machine learning, access to high-performance hardware, and a well-prepared dataset to ensure optimal results. Because of these complexities, it is not typically accessible to traditional developers and instead is done by a dedicated Data Science team.

Alignment Tuning. Alignment tuning adjusts a model's outputs to ensure compliance with ethical guidelines, safety regulations, and industry standards, making it essential for responsible AI deployment. This process involves modifying the model's decision-making process to align with predefined rules, much like defining security policies and implementing Role-Based Access Control (RBAC) to enforce access restrictions across a system. Similar to enforcing API rate limits, deploying security patches, or establishing governance policies at an enterprise level, alignment tuning ensures the AI operates within acceptable boundaries, mitigating risks associated with unintended behavior or biased decision-making.

The InstructLab project takes a novel approach to alignment tuning by using synthetic data generation and reinforcement learning from human feedback (RLHF) to refine AI behavior. InstructLab generates curated datasets that help models learn ethical reasoning, industry-specific regulations, and business logic while ensuring knowledge consistency across different applications. This approach allows developers to integrate AI systems that are adaptable to compliance needs without requiring full retraining, reducing costs and effort while maintaining safety and reliability.

Table 2-1 provides a complete overview of the tuning methods with their respective resource and cost implications and an indication of their advantages and disadvantages.

Table 2-1. Overview of Tuning Methods

Method	Effort	Resources	Cost	Skills Required	Pros	Cons
Prompt Tuning or Engineering	Low	Minimal	Low	Developers	• No extra infrastructure • Immediate, easy changes • Ideal for minor tweaks	• Limited deeper control • Trial and error needed
Prompt Learning	Medium	Moderate	Medium	Data Scientists	• Better reliability without parameter changes • More effective than prompt tuning	• Needs labeled data • Limited by model constraints
Parameter-Efficient Fine-Tuning (PEFT, LoRA)	Medium	Moderate	Medium	Developers, Data Scientists	• Lower training costs than full fine-tuning • Runs on standard GPUs • Adapts model without losing original knowledge	• Requires data prep and updates • Gains vary by task complexity
Full Fine-Tuning	Very High	Extensive	Very High	Data Scientists	• Maximum behavior control • Ideal for proprietary or highly specific tasks	• High compute/storage costs • Requires advanced ML expertise
Alignment Tuning	High	High	High	Developers	• Ensures ethical AI • Essential for regulated industries	• Complex implementation • Dedicated AI infrastructure needed • High ongoing costs

Understanding Tool Use and Function Calling

LLMs can go beyond simple text generation by interacting with external systems, APIs, and tools to enhance their capabilities. When a model claims "tool use" or "function calling," it means it has been specifically designed or fine-tuned to interpret, generate, and execute structured function calls rather than just responding with raw text. These features make LLMs more actionable and useful when it comes to retrieving data from APIs, running database queries, performing calculations, or triggering automated workflows.

"Tool use" refers to an LLM's ability to decide when and how to use external resources to complete a task. Instead of answering a question directly, the model can invoke a predefined tool, retrieve the necessary information, and incorporate it into its response. One way LLMs enable "tool use" is through system prompts, where models rely on predefined instructions embedded in system messages to determine when and how to call external tools. These prompts guide the model's behavior,

helping it recognize when an API call or external retrieval is needed instead of a direct response. An example system prompt could be the following:

```
In this environment you have access to a set of tools you can use to answer\n
 the user's question.
{{ FORMATTING INSTRUCTIONS }}
Lists and objects should use JSON format. The output is expected to be valid XML.
Here are the functions available in JSONSchema format:
{{ TOOL DEFINITIONS IN JSON SCHEMA }}
{{ USER SYSTEM PROMPT }}
{{ TOOL CONFIGURATION }}
```

By carefully crafting system prompts, you can direct the model to make decisions on when to respond with which answer in a structured way. This is just one way of prompting. You'll learn more about prompting in the section "Prompts for Developers".

For more advanced tool use, custom model adapters can be used to add a fine-tuned layer to existing models. This approach requires additional training or a specific adapter model but allows you to adapt a model to a specific use case or even domain. Instead of relying on general-purpose instructions, fine-tuned adapters improve a model's ability to detect when external functions should be invoked and generate precise API requests that align with a given service's requirements. We will talk about the architectural aspects of tools and function calling in chapter ???.

Choosing the Right LLM for Your Application

Categorizing LLMs is challenging due to their diverse capabilities, architectures, and applications. When selecting one for your project, it helps to categorize available options based on common attributes. Similar to how you evaluate every other service or library you use in your applications, you want to consider both functional and non-functional requirements. Utilizing everything we discussed so far, this section includes a common set of categories and attributes that might help you to decide which kind of model you need for your application. However, keep in mind there are additional attributes you might consider, and the final choice often depends on evaluating results—a task typically handled by data scientists. Please find a deeper understanding of model evaluation in Chip Huyen's O'Reilly book AI Engineering, 3. Evaluation Methodology.

Model Type

Choosing the right LLM type depends on how you plan to use it, balancing specificity, efficiency, and adaptability. We can group models by how they generate, retrieve, or understand text and other inputs.

Text Generation models perform well for open-ended tasks, such as chatbots, automated documentation, and summarization. They may need tuning to align with

business requirements. Instruction-tuned chat models specialize in conversational interfaces, making them suitable for customer support and AI-driven assistants; they respond to structured prompts with refined contextual understanding.

If a use case requires external knowledge, Retrieval-Augmented Generation (RAG) models integrate dynamic data sources for more accurate, domain-specific answers. Embedding models focus on semantic search, classification, and similarity matching to enhance AI-driven search and recommendation systems. Multimodal models process images and text for tasks like optical character recognition (OCR) or image-based question answering. Code generation models target developer productivity, assisting with automated refactoring and AI-assisted coding. Function and tool-calling models interact with enterprise systems to automate workflows or trigger specific API actions.

When deciding among these options, consider whether you need freeform text generation, structured responses, external knowledge, or specialized features such as coding or multimodal capabilities. This is the primary decision making category for you.

Model Size & Efficiency

Model size influences cost, accuracy, and latency. Small models (≤7B parameters) fit edge or on-premises deployments where low latency and limited resources matter most. Medium-sized models (7B–30B parameters) balance efficiency and performance, making them a balanced choice without too much infrastructure requirements. Large models (≥30B parameters) offer advanced reasoning but demand substantial compute resources.

Decide whether to prioritize lower cost, higher performance, or compatibility with existing hardware. In many scenarios, smaller or quantized models can provide results close to those of larger models, reducing hardware investments without losing essential functionality.

Deployment Approaches

The deployment strategy that you choose affects scalability, data security, and operational complexity. Consider that:

- API-based or 3rd party hosted models are straightforward to integrate, with almost no infrastructure overhead. They scale easily but may raise concerns about vendor lock-in, latency, and ongoing usage fees.
- Self-hosted models provide more control over data and can reduce inference costs when scaled out. However, they require managing GPUs or other specialized hardware and handling ongoing optimizations. This approach suits enter-

prises with strict compliance needs or those aiming to minimize reliance on external providers.

- Edge or local deployment offers low-latency, offline operations. They work well for mobile or IoT devices and also developer machines. But they face constraints due reduced model size and complexity.

Your choice depends on ease of integration, security requirements, and cost constraints. If you are handling sensitive data you want to use self-hosted or hybrid approaches. When you want to quickly deploy and scale you may opt for API- or 3rd party based models.

Supported Precision & Hardware Optimization

You already learned how numeric precisions, affects speed and memory usage. Full precision (FP32, BF16, FP16) delivers the highest accuracy. Quantized models (INT8, INT4) reduce memory demands and provide more speed. Furthermore the hardware choice influences the available precision options. For example CUDA and TensorRT based inference optimizes performance for NVIDIA GPUs, whereas ONNX, OpenVINO, and CoreML open deployment possibilities on Intel or Apple Silicon. Evaluate whether you need specialized accelerators or if general-purpose hardware will suffice.

Ethical Considerations & Bias

Bias and ethical risks arise when training on broad datasets. Mitigation strategies help ensure fairness and align with regulations. Some enterprise models implement built-in bias filtering, whereas open-source models may require extra oversight to manage potentially harmful outputs.

Regulatory compliance is also vital, particularly when handling personally identifiable information (PII). Content filtering features in proprietary models can help address these concerns, while open-source implementations demand custom safeguards. Transparency matters as well; models with open weights enable deeper scrutiny of training data and decision-making. Strike a balance between ethical obligations, operational constraints, and responsible AI practices.

Community & Documentation Support

Success with LLMs often relies on a robust developer ecosystem, solid documentation, and community support. Widely adopted open-source projects tend to offer extensive forums, software development kits (SDKs), and established best practices.

Enterprises that prefer vendor-backed services can look for solutions with service-level agreements (SLAs) and direct support. Comprehensive documentation, libraries, and frameworks—especially those with Java-friendly APIs—streamline the

integration process. When deciding, consider both the reliability of the model and the ecosystem's maturity to ensure a smoother rollout.

Closed vs. Open Source

LLMs can be categorized by their licensing models, which influence accessibility, customization options, and long-term sustainability. The choice between closed-source and open-source models carries significant consequences for enterprises, especially with respect to control, cost, and flexibility.

Closed-source models are proprietary solutions often provided through cloud-hosted APIs or software products. They typically feature specific capabilities and benefit from ongoing updates from the vendor. However, they can limit visibility into the underlying mechanisms, introduce vendor lock-in, and raise potential data privacy issues. Similar to using a proprietary Java framework, where you gain enterprise-level support but relinquish detailed control over the implementation.

Open-source models offer transparency, community-driven development, and the freedom to self-host and customize. Organizations with strict data governance often prefer these models because they maintain full authority over deployment and fine-tuning. Yet, open-source solutions generally require more in-house engineering for maintenance and optimization. This is comparable to open-source Java frameworks, which grant flexibility but demand internal expertise.

Enterprises must weigh their need for control, compliance, and cost-efficiency when deciding which approach to adopt. Closed-source offerings may provide an out-of-the-box experience, while open-source alternatives allow greater adaptability and independence from vendor constraints.

Example Categorization

Table 2-2 shows an example matrix that helps you weigh these attributes based on your project's priorities. The matrix includes potential considerations for each attribute and how you might rate them for different use cases (for instance, on a scale of Low, Medium, High).

Table 2-2. Decision Making Matrix

Attribute	Decision Factors	Example Rating
Model Type	General vs. domain focus Flexibility vs. specialization	Low/Med/High
Model Size & Efficiency	Resource consumption (CPU/GPU/Memory) Response time requirements	Low/Med/High
Deployment Modality	Data privacy needs Infrastructure control vs. convenience	Low/Med/High

Attribute	Decision Factors	Example Rating
Supported Precision & Hardware	Need for high throughput Hardware availability (GPUs vs. CPUs, etc.)	Low/Med/High
Ethical Considerations & Bias	User trust Regulations and compliance	Low/Med/High
Community & Documentation Support	Maturity of ecosystem Availability of tutorials / community expertise	Low/Med/High
Closed vs. Open Source	Proprietary and Transparency Data and Model Ownership and Flexibility	Low/Med/High
Function & Tool Calling	Application integration requirements Real-time data or external services	Low/Med/High

Let's walk through how to Use the Matrix:

+Define your primary goals (e.g., text classification, code completion, or domain-specific question answering). Then note whether you need a broad or niche solution. .Prioritize the Attributes +Determine which attributes matter most. For instance, if data security is paramount, you might score "Deployment Modality" and "Ethical Considerations" as "High." .Assign Ratings +Rate each attribute based on how critical it is to your project. A "Low" rating indicates it is less important, while "High" signals a crucial requirement. .Evaluate Trade-offs +Review high-rated attributes to see if they conflict. For example, you might want robust tool calling and low resource usage, but a smaller model may not offer extensive integration options. .Choose a Model Category After weighing the trade-offs, you can see which LLM category (general, specialized, large, small, etc.) or deployment approach (cloud, on-premises) best meets your needs.

Foundation Models or Expert Models - Where are we headed?

After examining different ways to categorize models, it's helpful to look ahead and consider how these categories might evolve in the future. One important distinction that is clearly emerging is between Foundation Models (FMs) and Expert Models (EMs). A Foundation Model in the context of Large Language Models refers to a type of pre-trained model that serves as the basis for a variety of specialized applications. Much like the foundation of a skyscraper supports structures of varying complexity, FMs provide a general-purpose framework that can be fine-tuned or adapted for specific tasks. These models are trained on vast datasets. Think of it like almost all the available public text, code, images, and other data sources—to learn broad linguistic, factual, and contextual representations. While FMs are designed to be general-purpose, many real-world applications benefit from Expert Models, which are way smaller and optimized for specific domains. They often outperform general-purpose FMs in their niche areas by focusing on task-specific accuracy and efficiency. In practice, organizations often deploy ensembles of expert models

rather than relying on a single FM. By combining domain-specialized models with a general-purpose FM, companies can achieve higher precision in critical applications while still leveraging the broad knowledge embedded in the foundational model.

Industry Perspectives: Large vs. Small Models vs. Task Oriented vs. Domain Specific

Researchers and practitioners continue to debate the trade-offs between large and small models. Initially, many viewed bigger models (measured in billions or even trillions of parameters) as the path forward. They seem to demonstrate better generalization and language capabilities. Scaling and using large models comes at a significant cost at every stage of the model lifecycle. As a result, there has been a shift toward smaller, task-optimized models that perform well with significantly less computational demands. Techniques such as distillation, pruning, and quantization help compress large models while keeping the desired model capabilities. Open-source models like Mistral 7B and LLaMA 2 13B are examples of this trend. They offer good performance at a fraction of the size of models like GPT-4 or Gemini.

Some industry experts argue that small, specialized models working in concert will outperform monolithic large models in specific applications. This is where model chaining and hybrid architectures come into play.

Mixture of Experts (MoE), Multi-Modal Models, Model Chaining, et al

A growing trend in AI is moving beyond purely text-based models to multi-modal models that integrate text, images, audio, and video. Those allow users to interact with AI in more natural ways. These models expand the traditional foundation model paradigm by enabling use-cases that combine inputs. Another evolving concept is model chaining, where multiple specialized models collaborate dynamically instead of relying on a single monolithic FM. Instead of deploying a general-purpose model to handle all tasks, task-specific expert models (e.g., a summarization model, a retrieval-augmented generation (RAG) model, or a reasoning engine) work together to achieve better accuracy and efficiency. This aligns with the shift toward retrieval-augmented generation (RAG) pipelines, where smaller models retrieve relevant documents before generating responses, reducing the need for massive parameter counts. Instead of chaining complete models, the MoE approach works model internally. Neural sub-networks represent multiple "experts" within a larger neural network, and a router selectively activates only those experts best suited to handle the input. Many systems already broadly use this approach.

DeepSeek and the Future of Model Architectures

Innovations like DeepSeek introduce hybrid model architectures that combine traditional neural networks with new reasoning and retrieval mechanisms. These approaches try to enhance the efficiency of FMs by focusing on modular, interpretable, and adaptable architectures rather than expensive scaling. Techniques such as

adaptive model scaling, task-specific adapters, and memory-augmented transformers push the boundaries of what FMs can achieve. Data Science moves fast and new models and further approaches appear fast. This is surely also based on the perceived competition in a very active field.

Now that we've covered the technical details of large language models, let's focus on practical application for developers. The upcoming section gives you an overview of how to write effective prompts.

Prompts for Developers - Why Prompts Matter in AI-Infused Applications

Prompts are the primary mechanism for interacting with LLMs. They define how an AI system responds, influencing the quality, relevance, and reliability of generated content. For Java developers building AI-infused applications, understanding prompt design is one of the most important skills. A well-structured prompt can reduce hallucinations, improve consistency, and optimize performance without requiring fine-tuning of the model. There are many different recommendations out there on how to write effective prompts and which techniques to use. An example is the OpenAI Prompt Engineering Guide or the book Prompt Engineering for Generative AI. Consider this a brief overview and the beginning of your learning journey.

Types of Prompts

Prompts differ based on their source and how they guide the model. Key types include:

User Prompts: Direct input from the user

User prompts are the raw input provided by end users. These are typically unstructured and need preprocessing or context enrichment to ensure accurate responses.

```
String userPrompt = "What is the capital of France?";
```

Handling user prompts effectively requires input sanitization, intent recognition, and context enhancement. We will get to this in more detail in the next chapter.

System Prompts: Instructions that guide model behavior

System prompts define how the model behaves in a session. These are often set at the start of an interaction and remain hidden from the user. They can be used to establish tone, enforce constraints, or guide the model's responses.

```
String systemPrompt = "You are a helpful AI assistant\n
                       that provides concise and factual responses.";
```

System prompts help in defining boundaries of the LLM within applications. They can also be used to enforce certain outputs or contain tool-calling instructions.

Contextual Prompts: Pre-populated or dynamically generated inputs

Contextual prompts include background information, past interactions, or domain-specific knowledge added to the prompt to improve responses. These can be dynamically generated based on user history or external data. This is also a very effective way to inject memory into conversations. We'll cover more on the architectural aspects of this in the next chapter.

```
String context = "User previously asked about European capitals.";
String fullPrompt = context + " " + userPrompt;
```

Contextual prompts enhance the relevance of responses, particularly in multi-turn conversations and function as de-facto memory, helping LLMs to keep conversational cohesivness.

Principles of Writing Effective Prompts

Specificity and structure are essential for effective prompt engineering. By being precise and organizing your prompts logically, you can significantly improve the quality and relevance of the responses you receive from LLMs. Investing time in crafting well-structured and specific prompts is a crucial step in getting the most out of these powerful tools. Here are two examples for a too specific and a too vague prompt:

```
String vaguePrompt = "Tell me about Java."; // Too broad
String specificPrompt = "Explain Java's garbage collection \n
                         mechanisms in one paragraph."; // Much better
```

The vaguePrompt is so open-ended that the LLM could respond with anything related to Java. Maybe its history, its uses, its syntax, etc. The specificPrompt, on the other hand, clearly states what information is needed and even specifies the desired length (one paragraph).

Crafting good prompts for Large Language Models (LLMs) is important for getting the responses you want. Common mistakes can make this difficult. One problem is being too vague, which leads to unclear or general results. Giving the model too much information can also confuse it. It's also important to keep prompt length in mind, as very long prompts may be cut off. Finally, you should test and change your prompts to make them better.

Prompting Techniques

Different prompting techniques offer structured ways to interact with models, ranging from direct instructions and examples to more complex methods that enhance reasoning or incorporate external knowledge.

Zero-shot prompting: Asking without context

Zero-shot prompting is where you ask the model to do something without giving it any examples. The model uses what it learned during training to understand and complete the request. It's like asking someone who knows a lot about a subject a question without giving them any background. Zero-shot prompts rely entirely on the model's pre-trained knowledge. A simple example looks like this:

```
String zeroShotPrompt = "Define polymorphism in object-oriented programming.";
```

While zero-shot prompting can work well, it has limits. Accuracy can vary with task complexity. Ambiguous prompts can be misinterpreted. It may have trouble with completely new tasks. Zero-shot prompting is good for quickly testing an LLM's abilities. It's a good starting point, but other methods might be needed for more complex tasks.

Few-shot prompting: Providing examples to guide responses

With few-shot prompting you provide a few examples of the task you want the model to perform. These examples demonstrate the desired input-output relationship and help steer the model towards generating the correct type of response. It's like showing someone a couple of examples of how to do something before asking them to do it themselves. Working with examples in Java can look like this:

```
String fewShotPrompt = "Translate the following phrases to Spanish:\n\n" +
                       "English: Hello\n" +
                       "Spanish: Hola\n\n" +        // Example 1
                       "English: Good morning\n" +
                       "Spanish: Buenos días\n\n" + // Example 2
                       "English: How are you?\n" +
                       "Spanish: ";                 // The LLM completes this
```

In this example, you're giving the LLM two examples of English-Spanish translations. This helps the model understand that you want a Spanish translation for "How are you?". It's more likely to give a correct translation ("¿Cómo estás?") than if you had used a zero-shot prompt. By seeing a few examples, the LLM can better understand the pattern or rule you want it to follow. It can generalize from these examples and apply the learned pattern to new, unseen inputs. This is particularly helpful for tasks where the desired output format is specific or where the task is slightly ambiguous.

Chain-of-Thought (CoT) prompting: Encouraging step-by-step reasoning

Chain-of-Thought (CoT) prompting is intended to help LLMs perform complex reasoning by explicitly generating a series of intermediate steps, or a "chain of thought," before arriving at a final answer. It's like asking someone to "show their work" on a math problem. Instead of just getting the final result, you see the step-by-step reasoning that led to it. This works great for word problems like the following:

```
Problem: Roger has 5 tennis balls. He buys 2 more cans of tennis balls.
Each can has 3 tennis balls. How many tennis balls does he have now?

Chain of Thought:
Roger started with 5 tennis balls.
He bought 2 cans of 3 tennis balls each, so he bought 2 * 3 = 6 tennis balls.
In total, he has 5 + 6 = 11 tennis balls.

Answer: 11
```

You're not just giving the LLM the problem and asking for the answer. You're showing it how to solve the problem by breaking it down into steps. If you then give the LLM a similar problem, it's more likely to generate its own chain of thought and arrive at the correct answer. LLMs have learned a lot about reasoning from the massive datasets they were trained on. However, they don't always explicitly use this reasoning ability when answering questions. CoT prompting encourages them to activate and utilize their reasoning capabilities by providing examples of how to think step-by-step.

Self-Consistency: Improving accuracy by generating multiple responses

Self-consistency is an approach to improve the accuracy of LLM responses, particularly on reasoning tasks. The core idea is simple: instead of relying on a single generated response, you generate multiple responses and then select the most consistent one. This leverages the idea that while an LLM might make occasional errors in its reasoning, the correct answer is more likely to appear consistently across multiple attempts. LLMs are probabilistic models. They don't always produce the same output for the same input. Sometimes they make mistakes, especially on complex reasoning tasks. However, the assumption behind self-consistency is that the correct answer is more likely to be generated repeatedly, even if the LLM makes occasional errors. By generating multiple responses, you increase the chances of capturing the correct answer and filtering out the incorrect ones.

Instruction Prompting: Directing the model explicitly

Instruction prompting is a straightforward approach. It involves giving a model explicit instructions about what you want it to do. Instead of relying on implicit cues or indirect suggestions, you directly tell the LLM what task to perform and what kind

of output you expect. It's like giving someone very clear and specific directions. A very explict prompt looks like the following:

```
TRANSLATE: English to Spanish
TEXT: "Hello, world!"
```

Instruction prompting is often used in conjunction with other prompting approaches. You might use instruction prompting within a few-shot learning setup, where the examples also include clear instructions. It can also be combined with chain-of-thought prompting by instructing the model to "show its reasoning step-by-step" before providing the final answer.

Retrieval-Augmented Generation (RAG): Enhancing prompts with external data

RAG is basically a version of Contextual Prompts. Instead of any text context, RAG use external sources, like databases or documents, to get more current or specific information before answering a question. Technically this is more an architectural approach than a prompting technique. We will look at how this works in more detail in ??? and unravel the complexity step by step.

Advanced Strategies

Building on fundamental techniques, this section covers advanced prompting strategies for creating more dynamic, reliable, and optimized interactions with language models. You will see many of these general examples again in chapters 6 and 7.

Dynamic Prompt Construction: Combining static and generated inputs

This approach involves building prompts on the fly by combining fixed text with information generated by other parts of the system. For example, you might have a template prompt for summarizing a product, but the specific product details (name, description, price) are pulled from a database and inserted into the prompt before sending it to the model. This allows for flexible and context-aware prompts. You can use Java's String features to dynamically add content like in the following example:

```
String productTemplate = "Summarize product:\n
                         Name: {productName}\n
                         Description: {productDescription}";

String productName = "Wireless Headphones";
String productDescription = "Noise-canceling, Bluetooth 5.0";

String dynamicPrompt = productTemplate.replace("{productName}", productName)
                                      .replace("{productDescription}", productDescription),
                                       String.valueOf(productPrice);

// The 'dynamicPrompt' can be sent to the model.
```

Use code to build prompts dynamically from templates and variable data for increased flexibility and context awareness.

Using Prompt Chaining to Maintain Context

If you need to maintain context across multiple interactions with a model you can use prompt chaining. Instead of treating each prompt in isolation, you link them together. The output of one prompt becomes part of the input for the next. This is useful for multi-step tasks, like building a story or answering complex questions that require multiple pieces of information. Think about a coversation where you still remeber the beginning and not just reply to the next question. Picture this as a very small prompt to prompt memory like in the following:

```
String initialPrompt = "List the ingredients in a Margherita pizza.";
String firstResponse = getLLMResponse(initialPrompt); // Hypothetical LLM call

String followUpPrompt = "How do I make a pizza with these ingredients:\n"
                        + firstResponse;
String finalResponse = getLLMResponse(followUpPrompt); // Hypothetical LLM call
```

Prompt chaining enables multi-step problem-solving or a very small conversational memory by incorporating previous model responses into subsequent prompts.

Guardrails and Validations for Safer Outputs

You'll find a lot of definitions for the terms guardrails and validations. They basically all mean the same. Ensuring that a models's output is safe and reliable. Guardrails might involve filtering or rejecting outputs that contain harmful content. Validations might check if the output conforms to a certain format or logic. For example, if you ask a model to generate code, you might validate that the code compiles correctly. We will look into more specific implementations in chapter 7. You can build them manually into your code as in the below example or use library features like in Quarkus or like LangChain4J provides for certain models.

```
String llmR = getLLMResponse("Write a short story."); // Hypothetical LLM call

// Simple guardrail: Check for harmful content
if (llmR.contains("violent") || llmR.contains("hate")) {
    System.out.println("Response flagged for inappropriate content.");
} else {
    // Validation (example: check length)
    if (llmR.length() > 500) {
      System.out.println("Response too long. Truncating.");
      llmR = llmR.substring(0, 500);
    }
    System.out.println(llmR);
}
```

Apply guardrails and validations to model outputs to enforce safety standards and verify conformance with expectations.

Leveraging APIs for Prompt Customization

Model providers often offer APIs that let you customize the prompting process. These APIs might allow you to set hyperparameter that control the model behavior, or they might provide tools for managing and organizing your prompts. Their ability depends on the API used and the functionality exposed.

Optimizing for Performance vs. Cost

Generating longer responses or making many API calls adds up in cost for usage or resources. Therefore, it's important to optimize for both performance (getting good results) and cost (minimizing expenses). This might involve using shorter prompts, caching common responses, or choosing a less expensive LLM for less critical tasks.

Debugging Prompts: Troubleshooting Poor Responses

"Debugging" prompts involves figuring out why the LLM gave a bad response and then revising the prompt to fix the problem. This often requires careful analysis of the prompt and the LLM's output to pinpoint the issue. It's like debugging code, but instead of code, you're debugging your questions.

Mastering prompting techniques gives us direct control over model interaction, but AI infused applications require more technical components. Let's look at the supporting technologies you usually find, such as embeddings, vector databases, caches, agents, and frameworks that facilitate more complex solutions.

Supporting Technologies

LLMs require a full-stack ecosystem beyond just model inference. Technologies like vector databases, caching, orchestration frameworks, function calling, and security layers are necessary for a production ready application.

Vector Databases & Embedding Models

Vector databases and embedding models are key elements for system architectures of AI systems. Especially when working with lots of unstructured information and complex searches. Embedding models transform text into vectors, which are lists of numbers that capture the text's meaning. Similar texts have vectors that are close together, enabling vector searches based on meaning. Vector databases then store and quickly search these vectors, using specialized indexing to find the vectors closest to a search query. This allows for fast retrieval of relevant documents, even in huge datasets. In a RAG setup, a user's question is converted into a vector and used to query the

vector database. The database returns similar document vectors, and the corresponding documents are retrieved. These documents are combined with the user's question to create a prompt for the LLM, which then generates a more informed response. Essentially, embedding models provide semantic understanding, and vector databases provide efficient search, working together to help getting specific information out of models without changing their weights. If you can not wait to see some more code behind how this actually works, turn the pages forward to chapter ???.

Caching & Performance Optimization

Caching involves storing the results of LLM requests so that identical or similar requests can be served directly from the cache, avoiding repeated calls to the model and saving both time and resources. Key considerations for caching include determining the appropriate cache key for identifying similar requests, establishing a cache invalidation strategy to handle outdated or changed data, selecting a suitable storage mechanism (in-memory, local files, or dedicated services) and managing cache size.

Beyond caching, other performance optimization techniques are also helpful. We've already talked about prompt optimizations but there's also the possibility to batch requests in single calls to reduce overhead. Asynchronous or stream based requests lets applications continue on other tasks while waiting for model responses. Continuous monitoring and profiling helps to identify performance bottlenecks as with any other traditional system. So it is really important to implement a suitable monitoring solution from the very beginning of your project.

AI Agent Frameworks

Models alone are not enough to build intelligent applications. You need tools to integrate these models into workflows, interact with external systems, and handle structured decision-making. AI agent frameworks promise to bridge this gap by managing tool execution, memory, and reasoning. All in one place. The term "agent" is used differently across various discussions today. At a basic level, an agent can be:

Most real-world implementations today focus on tool invocation, while full agent-based architectures are still developing.

LangChain4J for example offers a structured way to integrate AI-driven tools into applications, using declarative tool definitions together with prompt management and structured responses, enriched by context handling and memory management. IBM's Bee Agent Framework takes a different approach, focusing on multi-agent workflows. It also contains a notion of distributed agents with explicit task execution and planning as well as cusomized integrations.Bee is in early development but presents an alternative to single-agent models by allowing multiple agents to work together.

Model Context Protocol (MCP)

The Model Context Protocol (MCP) emerged as a very early standard and defines how applications provide contextual information to AI models. MCP is probably going to replace traditional tool/function calling with a structured, session-based approach that separates context, intent, and execution. Instead of issuing one-shot function calls packed into prompts, the model operates within defined contexts, expresses goals as intents, and interacts with typed resources through clear protocols. This design enables lifecycle management, state awareness, and consistent behavior across runtimes. Unlike tightly coupled, tool-specific implementations, MCP promotes model-agnostic interoperability, better debugging, and long-lived, goal-driven interactions—ideal for building robust, agentic systems.

API Integration

API integration is another really imporant part in modern system architectures. It makes AI models accessible and manageable in production environments. While models provide the intelligence, APIs handle communication, security, monitoring, and performance optimization. You can find traditional API management solutions with specific offerings enhanced for model access but also features integrated into AI platforms or even model registries. The main responsibilities of API management frameworks are:

- Authentication & Authorization: Using OAuth, JWT, or API keys to restrict access.
- Role-Based Access Control (RBAC): Grants permissions based on user roles (e.g., developers vs. inference consumers).
- Audit Logs: Tracks API requests for monitoring and compliance.
- Rate limiting: Preventing excessive API usage.
- Load balancing: Distributing requests across multiple instances.
- Observability - Latency detection: Measuring response times.
- Observability - Request volume monitoring: Identifying trends and potential scaling needs.
- Caching: Reducing redundant API calls for efficiency.

Beyond handling requests, API integration also serves as the first line of defense for securing model access and enforcing usage policies. But API gateways alone aren't enough—deeper, layered security is required to protect models, data, and operations end to end.

Model Security, Compliance & Access Control

While API management usually is the outermost layer of an AI-infused system, there are more elements concerned when it comes to further security considerations. Security, compliance, and access control are layered across different parts of an AI system. From model storage and access control to runtime monitoring and compliance enforcement. This adds more complexity to those applications ultimately. What makes it particularly challenging is the heterogenity in infrastructure, runtimes and even languages. While we do know how to build robust cloud-native applications, the age of AI-infused applications just started and outside of some very few offerings, there hardly is any cohesive all-in-one platform available as of today.

To manage these risks effectively, different components of an AI platform require their own security measures. The following areas highlight where targeted controls and best practices are essential and need to be implemented.

Model Storage and Registry Security
: A model registry stores and tracks different versions of models, ensuring governance and traceability. Security in this layer includes:

  - Encryption – Protects stored models from unauthorized access.
  - Integrity Checks – Uses hashing or digital signatures to ensure the model has not been tampered with.
  - Access Controls – Limits who can upload, modify, or retrieve models.

Compliance and Governance
: AI models must comply with regulations like GDPR, HIPAA, and SOC 2, depending on their use case. Compliance involves:

  - Data Anonymization – Ensures personal data used in training does not expose sensitive information.
  - Audit Trails – Logs all model training, versioning, and inference requests for traceability.
  - Bias & Fairness Audits – Implements tools like IBM AI Fairness 360 to detect biases in models.

Runtime Security and Model Monitoring
: Once deployed, models need continuous monitoring for security threats, performance degradation, or adversarial attacks. This includes:

  - Input Validation – Prevents injection attacks and malformed inputs that could crash the model.
  - Drift Detection – Alerts when input data distribution changes significantly.
  - Rate Limiting – Controls excessive API requests to prevent abuse.

Conclusion

AI integration in Java applications builds on established principles rather than replacing them. Just as we adapted to cloud-native architectures and reactive programming, it is now working with AI models that requires new tools and concepts while maintaining modularity, scalability, and maintainability.

This chapter covered the foundational elements of AI integration—understanding large language models, architectural choices, and supporting technologies. We explored different model types, deployment strategies, and the trade-offs between open and closed-source solutions. We also introduced retrieval-augmented generation (RAG), function calling, and tuning techniques that help integrate AI into enterprise systems effectively.

The challenge for Java developers is not only in understanding AI models but in applying them in ways that align with existing software practices. Whether optimizing inference, selecting deployment methods, or refining model responses, the ability to integrate AI effectively will shape future applications.

The next chapter will focus on architecting AI-powered applications, ensuring these capabilities fit seamlessly into enterprise systems.

CHAPTER 3
Inference API

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

You've already expanded your knowledge about AI, and the many types of models. Moreover, you deployed these models locally (if possible) and test them with some queries. But when it is time to use models, you need to expose them properly, follow your organization's best practices, and provide developers with an easy way to consume the model.

An Inference API helps solve these problems, making models accessible to all developers.

This chapter will explore how to expose an AI/ML model using an Inference API in Java.

What is an Inference API?

An Inference API allows developers to send data (in any protocol, such as HTTP, gRPC, Kafka, etc.) to a server with a machine learning model deployed and receive the predictions or classifications as a result.

Practically, every time you access cloud models like *OpenAI* or *Gemini* or models deployed locally using *ollama*, you do so through their Inference API.

Even though it is common these days to use big models trained by big corporations like Google, IBM, or Meta, mostly for LLM purposes, you might need to use small custom-trained models to solve one specific problem for your business.

Usually, these models are developed by your organization's data scientists, and you must develop some code to infer them.

Let's take a look at the following example:

Suppose you are working for a bank, and data scientists have trained a custom model to detect whether a credit card transaction can be considered fraud.

The model is in `onnx` format with six input parameters and one output parameter of type `float`.

As input parameters:

`distance_from_last_transaction`
: The distance from the last transaction that happened. For example, 0.3111400080477545.

`ratio_to_median_price`
: Ratio of purchased price transaction to median purchase price. For example, 1.9459399775518593.

`used_chip`
: Is the transaction through the chip. 1.0 if `true, `0.0` if false.

`used_pin_number`
: Is the transaction that happened by using a PIN number. 1.0 if `true`, 0.0 if `false`.

`online_order`
: Is the transaction an online order. 1.0 if `true`, 0.0 if `false`.

And the output parameter:

`prediction`
: The probability the transaction is fraudulent. For example, 0.9625362.

A few things you might notice here are:

- Everything is a float, even when referring to a boolean like in the `used_chip` field.
- The output is a probability, but from the business point of view, you want to know if there has been fraud.
- Developers prefer using classes instead of individual parameters.

This is a typical use case for creating an Inference API for the model to add an abstraction layer that makes consuming the model easier.

The Figure 3-1 shows the transformation between a JSON document and the model parameters done by the Inference API:

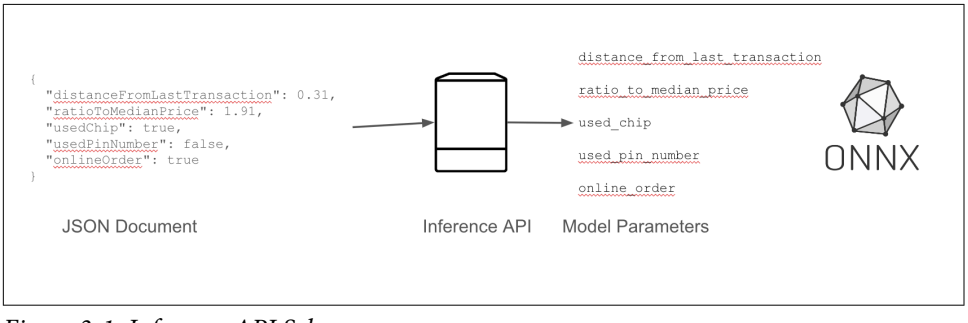


Figure 3-1. Inference API Schema

The advantages of having an Inference API are:

- The models are easily scalable. The model has a standard API, and because of the stateless nature of models, you can scale up and down as any other application of your portfolio.
- The models are easy to integrate with any service as they offer a well-known API (REST, Kafka, gRPC, …)
- It offers an abstraction layer to add features like security, monitoring, logging, …

Now that we understand why having an Inference API for exposing a model is important let's explore some examples of Inference APIs.

Examples of Inference APIs

Open (and not Open) Source tools offer an inference API to consume models from any application. In most cases, the model is exposed using a REST API with a documented format. The application only needs a REST Client to interact with the model.

Nowadays, there are two popular Inference APIs that might become the de facto API in the LLM space. We already discussed them in the previous chapter: one is OpenAI, and the other is Ollama.

Let's explore each of these APIs briefly. The idea is not to provide full documentation of these APIs but to give you concrete examples of Infernece APIs so that in case you develop one, you can get some ideas from them.

OpenAI

OpenAI offers different Inference APIs, such as *chat completions*, *embeddings*, *image*, *image manipulation*, or *fine tuning*.

To interact with those models, create an HTTP request including the following parts:

- The HTTP method used to communicate with the API is `POST`.
- OpenAI uses a Bearer token to authenticate requests to the model.
- Hence, any call must have an HTTP header named `Authorization` with the value `Bearer $OPENAI_API_KEY.`
- The body content of the request is a JSON document.

In the case of *chat completions*, two fields are mandatory: the `model` to use and the `messages` to send to complete.

An example of body content sending a simple question is shown in the following snippet:

```
{
    "model": "gpt-4o", ❶
    "messages": [ ❷
      {
        "role": "system", ❸
        "content": "You are a helpful assistant."
      },
      {
        "role": "user", ❹
        "content": "What is the Capital of Japan?"
      }
    ],
    "temperature": 0.2 ❺
  }
```

❶ Model to use

❷ Messages sent to the model with the role

❸ Role `system` allows you to specify the way the model answers questions

❹ Role `user` is the question

❺ Temperature value defaults to 1.

And the response contains multiple fields, the most important one `choices` offering the responses calculated by the model:

```
{
  "id": "chatcmpl-123",
...
  "choices": [{ ❶
    "index": 0,
    "message": {
      "role": "assistant", ❷
      "content": "\n\nThe capital of Japan is Tokyo.", ❸
    },
    "logprobs": null,
    "finish_reason": "stop"
  }],
...
}
```

❶ A list of chat completion choices.

❷ The role of the author of this message.

❸ The response of the message

In the case of *embeddings*, `model` and `input` fields are required:

```
{
    "input": "This is a cat", ❶
    "model": "text-embedding-ada-002" ❷
}
```

❶ String to vectorize

❷ Model to use

The response contains an array of floats in the `data` field containing the vector data:

```
{
  "object": "list",
  "data": [
    {
      "object": "embedding",
      "embedding": [ ❶
        0.0023064255,
        -0.009327292,
        .... (1536 floats total for ada-002)
```

```
        -0.0028842222,
      ],
      "index": 0
    }
  ],
...
}
```

❶ The vector data

These are two examples of OpenAI Inference API, but you can find the documentation at *https://platform.openai.com/docs/overview*.

Ollama

Ollama provides an Inference API to access *LLM* models that are running in ollama.

Ollama has taken a significant step forward by making itself compatible with the OpenAI Chat Completions API, making it possible to use more tooling and applications with Ollama. This effectively means interacting with models running in ollama for *chat completions* can be done either with OpenAI API or with ollama API.

It uses the `POST` HTTP method, and the body content of the request is a JSON document, requiring two fields, `model` and `prompt`:

```
{
  "model": "llama3", ❶
  "prompt": "Why is the sky blue?", ❷
  "stream": false ❸
}
```

❶ Name of the model to send the request

❷ Message sent to the model

❸ The response is returned as a single response object rather than a stream

The response is:

```
{
  "model": "llama3",
...
  "response": "The sky is blue because it is the color of the sky.", ❶
  "done": true,
...
}
```

❶ The generated response

In a similar way to OpenAI, llama provides an API for calculating embeddings. The request format is quite similar, requiring the `model`, and `input` fields:

```
{
  "model": "all-minilm",
  "input": ["Why is the sky blue?"]
}
```

The response is a list of embeddings:

```
{
  "model": "all-minilm",
  "embeddings": [[
    0.010071029, -0.0017594862, 0.05007221, 0.04692972, 0.054916814,
    0.008599704, 0.105441414, -0.025878139, 0.12958129, 0.031952348
  ]]
}
```

These are two examples of ollama Inference API, but you can find the documentation at *https://github.com/ollama/ollama/blob/main/docs/api.md*

In these sections, we discussed why an Inference API is important and explored some existing ones, mostly for LLM models.

Next, let's get back to our fraud detection model introduced at the beginning of this chapter. Let's discuss how to implement an Inference API for the model and, even more importantly, how to do it in Java.

In the next section, we'll develop an Inference API in Java, deploy it, and send some queries to validate its correct behavior.

Deplying Inference Models in Java

Deep Java Library (or *DJL*) is an open-source Java project created by Amazon to develop, create, train, test and infer machine learning and deep learning models natively in Java.

DJL provides a set of APIs abstracting the complexity involved in developing Deep learning models, providing a unified way for training and inferencing for most popular AI/ML frameworks like Apache MxNet, PyTorch, Tensorflow, ONNX formats, or even the popular HuggingFace AutoTokenizer and Pipeline.

DJL contains a high-level abstraction layer that connects to the corresponding AI/ML model to use, making a change on the runtime almost transparent from the Java application layer.

You can configure DJL to use CPU or GPU; both modes are supported based on the hardware configuration.

A model is just a file/s. DJL will load the model and offer a programmatic way to interact with it. The model can be trained using DJL or any other training tool (Python `sk-learn`) as long as it saves the model into a supported file format by DJL.

The Figure 3-2 shows an overview of the DJL architecture. The bottom layer shows the integration between DJL and CPU/GPU, the middle layer are native libraries to run the models, and tese layers are controlled using plain Java:

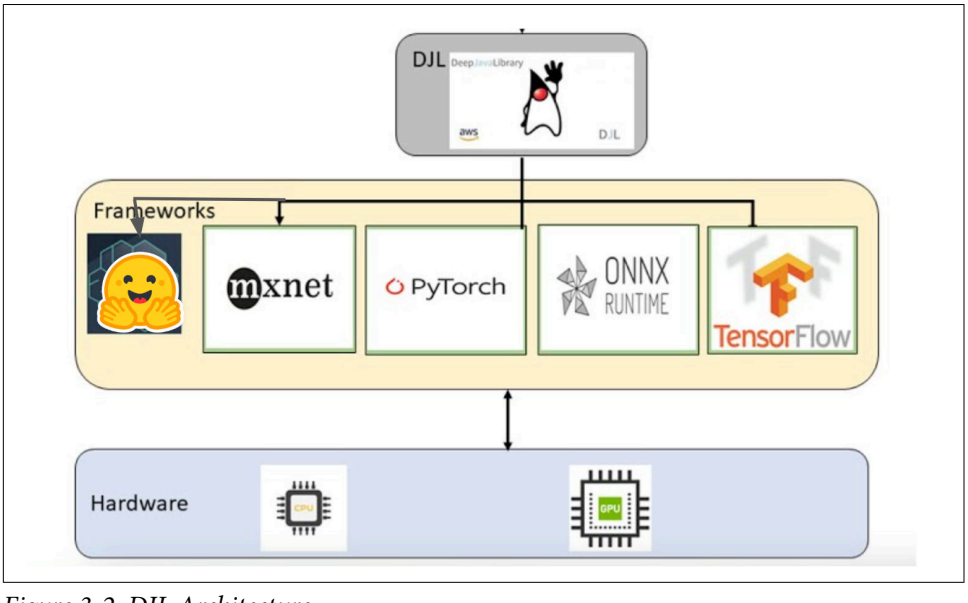


Figure 3-2. DJL Architecture

Even though DJL provides a layer of abstraction, you still need to have a basic understanding of machine learning common concepts.

Inferencing models with DJL

The best way to understand DJL for inferencing models is to develop an example. Let's develop a Java application using DJL to create an Inference API to expose the `onnx` fraud detection model described previously.

Let's use Spring Boot to create a REST endpoint to infer the model. The Figure 3-3 shows what we want to implement:

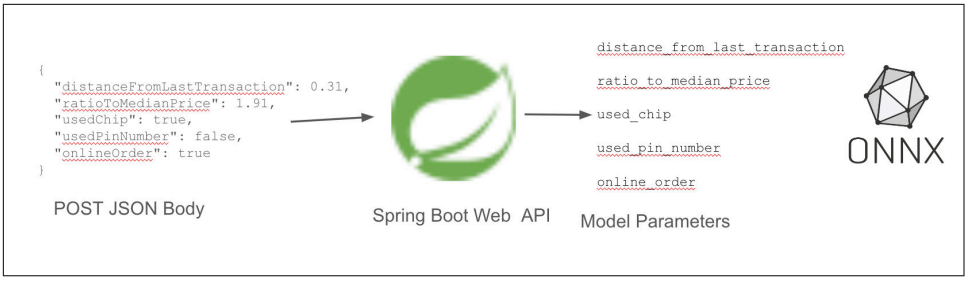


Figure 3-3. Spring Boot Rest API Schema

First, generate a simple Spring Boot application with Spring Web dependency. You can use Spring Initializr (*https://start.spring.io/*) to scaffold the project or start from scratch. The name of the project is `fraud-detection`, and add the `Spring Web` dependency.

The Figure 3-4 shows the Spring Initializr parameters for this example:

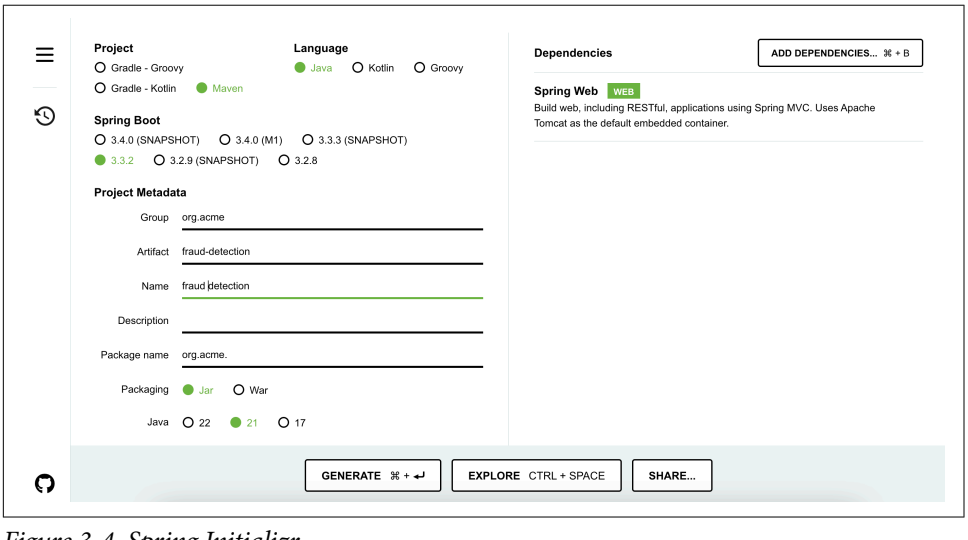


Figure 3-4. Spring Initializr

With the basic layout of the project, let's work through the details, starting with adding the DJL dependencies.

Dependencies

DJL offers multiple dependencies depending on the AI/ML framework used. DJL project provides a Bill of Materials (BOM) dependency to manage the versions of the project's dependencies, offering a centralized location to define and update these versions.

Add the BOM dependency (in the `dependencyManagement` section) in the `pom.xml` file of the project:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>ai.djl</groupId>
            <artifactId>bom</artifactId>
            <version>0.29.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Since the model is in `onnx` format, add the following dependency containing the ONNX engine: `onnxruntime-engine`:

```
<dependency> ❶
    <groupId>ai.djl.onnxruntime</groupId>
    <artifactId>onnxruntime-engine</artifactId>
</dependency>
```

❶ No version is required as it is inherited from BOM

The next step is creating two Java records, one representing the request and another representing the response.

POJOs

The request is a simple Java record with all the transaction details.

```
public record TransactionDetails(String txId,
            float distanceFromLastTransaction,
            float ratioToMedianPrice, boolean usedChip,
            boolean usedPinNumber, boolean onlineOrder) {}
```

The response is also a Java record returning a boolean setting if the transaction is fraudulent.

```
public record FraudResponse(String txId, boolean fraud) {
}
```

The next step is configuring and loading the model into memory.

Loading the model

We'll use two classes to configure and load the fraud detection model: `ai.djl.repository.zoo.Criteria` and `ai.djl.repository.zoo.ZooModel`. Let's look at each of those in more detail:

`Criteria`
: This class configures the location and interaction with the model. Criteria support loading models from multiple storages (local, S3, HDFS, URL) or implementing own protocol (FTP, JDBC, …). Moreover, you configure the transformation from Java parameters to model parameters and viceversa.

`ZooModel`
: The ModelZoo API offers a standardized method for loading models while abstracting from the engine. Its declarative approach provides excellent flexibility for testing and deploying the model.

Create a Spring Boot Configuration class to instantiate these classes. A Spring Boot Configuration class needs to be annotated with `@org.springframework.context.annotation.Configuration`.

```
@Configuration
public class ModelConfiguration {
}
```

Then, create two methods, one instantiating a `Criteria` and the other one a `ZooModel`.

The first method creates a `Criteria` object with the following parameters:

- The location of the model file, in this case, the model is stored at classpath.
- The data type that developers send to the model, for this example, the Java record created previously with all the transaction information.
- The data type returned by the model, a boolean indicating whether the given transaction is fraudulent.
- The transformer to adapt the data types from Java code (`TransactionDetails`, `Boolean`) to the model parameters (`ai.djl.ndarray.NDList`).
- The engine of the model.

```
@Bean
public Criteria<TransactionDetails, Boolean> criteria() { ❶

    String modelLocation = Thread.currentThread()
 .getContextClassLoader()
 .getResource("model.onnx").toExternalForm(); ❷

    return Criteria.builder()
 .setTypes(TransactionDetails.class, Boolean.class) ❸
 .optModelUrls(modelLocation) ❹
 .optTranslator(new TransactionTransformer(THRESHOLD)) ❺
 .optEngine("OnnxRuntime") ❻
 .build();
}
```

❶ The Criteria object is parametrized with the input and output types

❷ Gets the location of the model within the classpath

❸ Sets the types

❹ The Model location

❺ Instantiates the Transformer to adapt the parameter.

❻ The Runtime. This is especially useful when more than one engine is present in the classpath.

The second method creates the `ZooModel` instance from the `Criteria` object created in the previous method:

```
@Bean
public ZooModel<TransactionDetails, Boolean> model(
 @Qualifier("criteria") Criteria<TransactionDetails, Boolean> criteria) ❶
 throws Exception {
    return criteria.loadModel(); ❷
}
```

❶ Criteria object is injected

❷ Calls the method to load the model

One piece is missing from the previous implementation, and it is the `Transaction Transformer` class code.

Transformer

The transformer is a class implementing the `ai.djl.translate.NoBatchifyTranslator` to adapt the model's input and output parameters to Java business classes.

The model input and output classes are of type `ai.djl.ndarray.NDList`, which represents a list of arrays of floats.

For the fraud model, the *input* is an array in which the first position is the `distanceFromLastTransaction` parameter value, the second position is the value of `ratioToMedianPrice`, and so on. For the *output*, it is an array of one position with the probability of fraud.

The transformer has the responsibility to have this knowledge and adapt it according to the model.

Let's implement one transformer for this use case:

```
public class TransactionTransformer
        implements NoBatchifyTranslator<TransactionDetails, Boolean> { ❶

    private final float threshold; ❷

    public TransactionTransformer(float threshold) {
        this.threshold = threshold;
}

@Override
    public NDList processInput(TranslatorContext ctx, TransactionDetails input) ❸
            throws Exception {
        NDArray array = ctx.getNDManager().create(toFloatRepresentation(input),
            new Shape(1, 5)); ❹
        return new NDList(array);
}

    private static  float[] toFloatRepresentation(TransactionDetails td) {
        return new float[] {
            td.distanceFromLastTransaction(),
            td.ratioToMedianPrice(),
            booleanAsFloat(td.usedChip()),
            booleanAsFloat(td.usedPinNumber()),
            booleanAsFloat(td.onlineOrder())
};
}

    private static float booleanAsFloat(boolean flag) {
        return flag ? 1.0f : 0.0f;
}

@Override
    public Boolean processOutput(TranslatorContext ctx, NDList list)  ❺
            throws Exception {
        NDArray result = list.getFirst();
        float prediction = result.toFloatArray()[0];
        System.out.println("Prediction: " + prediction);

        return prediction > threshold; ❻
}

}
```

❶ Interface with types to transform

❷ Parameter set to decide when fraud is considered

❸ Transforming business inputs to model inputs

❹ Shape is the size of the array (5 parameters)

❺ Process the output of the model

❻ Calculates if the probability of fraud is beyond the threshold or not

With the model in memory, it is time to query it with some data.

Predict

The model is accessed through the `ai.djl.inference.Predictor` interface. The predictor is the main class that orchestrates the inference process.

The predictor is **not thread-safe**, so performing predictions in parallel requires one instance for each thread. There are multiple ways to handle this problem. One option is creating the `Predictor` instance per request. Another option is to create a pool of `Predictor` instances so threads can access them.

Moreover, it is **very important** to close the predictor when it is no longer required to free memory.

Our advice here is to measure the performance of creating the `Predictor` instance per request and then decide whether it is acceptable or use the first or the second option.

To implement per-request strategy in Spring Boot, return a `java.util.function.Supplier` instance, so you have control over when the object is created and closed.

```
@Bean
public Supplier<Predictor<TransactionDetails, Boolean>> ❶
predictorProvider(ZooModel<TransactionDetails, Boolean> model) { ❷
    return model::newPredictor; ❸
}
```

❶ Returns a `Supplier` instance of the parametrized `Predictor`

❷ `ZooModel` created previously is injected

❸ Creates the `Supplier`

The last thing to do is expose the model through a REST API.

REST Controller

To create a REST API in Spring Boot, annotate a class with `@org.springframework.web.bind.annotation.RestController`.

Moreover, since the request to detect fraud should go through the `POST` HTTP Method, annotate the method with the' @org.springframework.web.bind.annotation.PostMapping` annotation.

The `Predictor` supplier instance is injected using the `@jakarta.annotation.Resource` annotation.

```
@RestController
public class FraudDetectionInferenceController {

 @Resource
    private Supplier<Predictor<TransactionDetails, Boolean>> predictorSupplier; ❶

 @PostMapping("/inference")
    FraudResponse detectFraud(@RequestBody TransactionDetails transactionDetails)
        throws TranslateException {
        try (var p = predictorSupplier.get()) { ❷ ❸
            boolean fraud = p.predict(transactionDetails); ❹
            return new FraudResponse(transactionDetails.txId(), fraud); ❺
 }
 }
}
```

❶ Injects the supplier

❷ Creates a new instance of the `Predictor`

❸ `Predictor` implements `Autoclosable`, so try-with-resources is used

❹ Makes the call to the model

❺ Builds the response

The service is ready to start and expose the model.

Testing the example

Go to the terminal window, move to the application folder, and start the service by calling the following command:

```
./mvnw clean spring-boot:run
```

Then send two requests to the service, one with no fraud parameters and another one with fraud parameters:

```
// None Fraud Transaction

curl -X POST localhost:8080/inference \
     -H 'Content-type:application/json' \
     -d '{"txId": "1234",
 "distanceFromLastTransaction": 0.3111400080477545,
 "ratioToMedianPrice": 1.9459399775518593,
 "usedChip": true,
 "usedPinNumber": true,
 "onlineOrder": false}'
```

```
// Fraud Transaction

curl -X POST localhost:8080/inference \
     -H 'Content-type:application/json' \
     -d '{"txId": "5678",
 "distanceFromLastTransaction": 0.3111400080477545,
 "ratioToMedianPrice": 1.9459399775518593,
 "usedChip": true,
 "usedPinNumber": false,
 "onlineOrder": false}'
```

And the output of both requests are:

```
{"txId":"1234","fraud":false}
{"txId":"5678","fraud":true}
```

Moreover, if you inspect the Spring Boot console logs, you'll see the calculated probability of fraud done by the model.

```
Prediction: 0.4939952
Prediction: 0.9625362
```

Now, you've successfully run an Inference API exposing a model using only Java.

Let's take a look of what's happening under the hood when the application starts the DJL framework:

Under the hood

JAR file doesn't bundle the AI/ML engine for size reasons. In this example, if the JAR contained the ONNX runtime, it should contain all ONNX runtime for all the supported plaforms. For example, ONNX runtime for Operating Systems like Linux or MacOS and all possible hardware, such as ARM or x86 architectures.

To avoid this problem, when we start an application using DJL, it automatically downloads the model engine for the running architecture.

DJL uses cache directories to store model engine-specific native files; they are downloaded only once. By default, cache directories are located in the `.djl.ai` directory under the current user's home directory.

You can change this, by setting the `DJL_CACHE_DIR` system property or environment variable. Adjusting this variable will alter the storage location for both model and engine native files.

DJL does not automatically clean obsolete cache in the current version. Users can manually remove unused models or native engine files.

If you plan to containerize the application, we recommend bundling the engine inside the container to avoid downloading the model every time the container is started. Furthermore, start-up time is improved.

One of the best features of the *DJL* framework is its flexibility in not requiring a specific protocol for model inferencing. You can opt for the Kafka protocol if you have an event-driven system or the *gRPC* protocol for high-performance communication.

Let's see how the current example changes when using *gRPC*.

Inferencing Models with gRPC

gRPC is an open-source API framework following the Remote Procedure Call (RPC) model. Although the *RPC* model is general, *gRPC* serves as a particular implementation. *gRPC* employs Protocol Buffers and HTTP/2 for data transmission.

gRPC is only the protocol definition; every language and frameworks have an implementation of both the main elements of a *gRPC* application, the *gRPC Server* and the *gRPC Stub* .

gRPC Server
: It is the server part of the application, where you define the endpoint and implement the business logic.

gRPC Stub
: It is the client part of the application, the code that makes remote calls to the server part.

The Figure 3-5 provides a high-level overview of a *gRPC* architecture of an application. You see a *gRPC Service* implemented in Java, and two clients connecting to this service (one in Java and the other one in Ruby) using Protocol Buffer format.

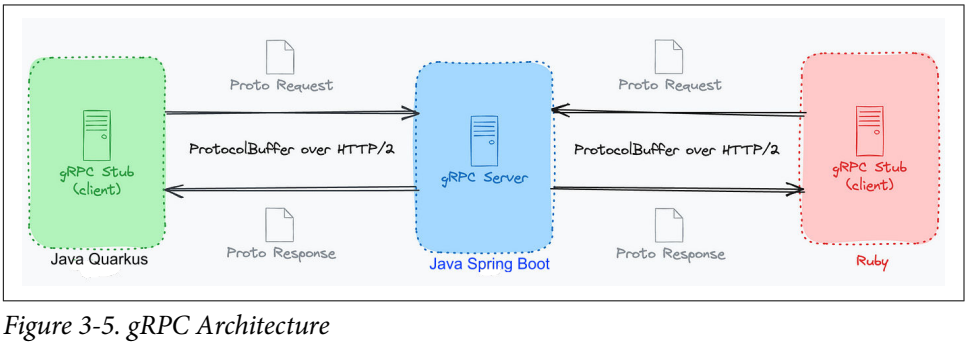


Figure 3-5. gRPC Architecture

gRPC offers advantages over REST when implementing high-performance systems, with high data loads, or when you need real-time applications. In most cases, *gRPC* is used for internal systems communications, for example, between internal services in a microservices architecture. Our intention here is not to go deep in *gRPC*, but to show you the versatility of inferencing models with Java.

Throughout the book, you'll see more ways of doing this, but for now let's transform the Fraud Detection example into a *gRPC* application.

Protocol Buffers

The initial step in using protocol buffers is to define the structure for the data you want to serialize, along with the services, specifying the RPC method parameters and return types as protocol buffer messages. This information is defined in a `.proto` file used as the Interface Definition Language (*IDL*).

Let's implement the *gRPC Server* in the Spring Boot project.

Create a `fraud.proto` file in `src/main/proto` with the following content expressing the Fraud Detection contract.

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.acme.stub"; ❶

package fraud;


service FraudDetection { ❷
 rpc Predict (TxDetails) returns (FraudRes) {} ❸
}

message TxDetails { ❹
 string tx_id = 1; ❺
 float distance_from_last_transaction = 2;
 float ratio_to_median_price = 3;
 bool used_chip = 4;
 bool used_pin_number = 5;
 bool online_order = 6;
}

message FraudRes {
 string tx_id = 1;
 bool fraud = 2;
}
```

❶ Defines package where classes are going to be materialized

❷ Defines Service name

❸ Defines method signature

❹ Defines the data transferred

❺ Integer is the order of the field

With the contract API created, use a *gRPC* compiler to scaffold all the required classes for implementing the server side.

The Figure 3-6 summarizes the process:

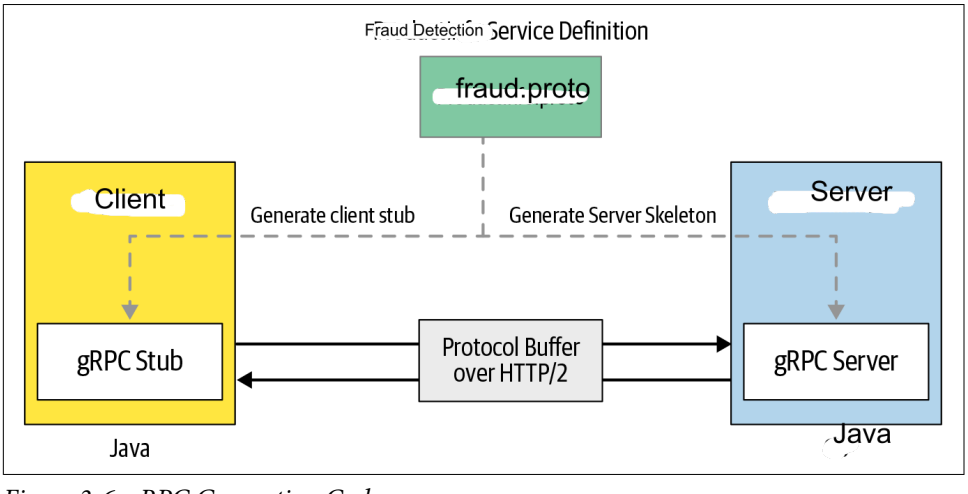


Figure 3-6. gRPC Generation Code

Let's create the *gRPC server* reusing the Spring Boot project but implementing now the Inference API for the Fraud Detection model using gRPC Protocol Buffers.

Implementing the gRPC Server

To implement the server part, open the `pom.xml` file and add dependencies for coding the *gRPC server* using Spring Boot ecosystem. Add the Maven extension and plugin to automatically read the `src/main/proto/fraud.proto` file and generate the required stubs and skeletons classes.

These generated classes are the data messages (`TxDetails` and `FraudRes`) and the base classes containing the logic for running the *gRPC server*.

Add the following dependencies:

```
<dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-protobuf</artifactId>
        <version>1.62.2</version>
```

```
</dependency>

<dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-stub</artifactId>
        <version>1.62.2</version>
</dependency>

<dependency>
        <groupId>net.devh</groupId>
        <artifactId>grpc-server-spring-boot-starter</artifactId>
        <version>3.1.0.RELEASE</version>
</dependency>

<dependency>
        <groupId>javax.annotation</groupId>
        <artifactId>javax.annotation-api</artifactId>
        <version>1.3.2</version>
        <scope>provided</scope>
        <optional>true</optional>
</dependency>

<build>
    ...
        <extensions>
                <extension> ❶
                        <groupId>kr.motd.maven</groupId>
                        <artifactId>os-maven-plugin</artifactId>
                        <version>1.7.1</version>
                </extension>
        </extensions>
    ...
        <plugins>
                <plugin> ❷
           <groupId>org.xolstice.maven.plugins</groupId>
                        <artifactId>protobuf-maven-plugin</artifactId>
                        <version>0.6.1</version>
                        <configuration> ❸
                                <protocArtifact>
          com.google.protobuf:protoc:3.25.1:exe:${os.detected.classifier}
                                </protocArtifact>
                                <pluginId>grpc-java</pluginId>
                                <pluginArtifact>
          io.grpc:protoc-gen-grpc-java:3.25.1:exe:${os.detected.classifier}
                                </pluginArtifact>
                        </configuration>
                        <executions> ❹
                                <execution>
                                        <id>protobuf-compile</id>
                                        <goals>
                                                <goal>compile</goal>
                                                <goal>test-compile</goal>
                                        </goals>
```

```
                </execution>
                <execution>
                    <id>protobuf-compile-custom</id>
                    <goals>
                        <goal>compile-custom</goal>
                        <goal>test-compile-custom</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        ...
    </plugins>
```

❶ Adds an extension that gets OS information and stores it as system properties

❷ Registers plugin to compile the protobuf file

❸ Configures the plugin using properties set by the `os-maven-plugin` extension to download the correct version of `protobuf` compiler

❹ Links the plugin lifecycle to the Maven `compile` lifecycle

At this point, every time you compile the project through Maven, the `protobuf-maven-plugin` generates the required *gRPC* classes from `.proto` file. These classes are generated at `target/generated-sources/protobuf` directory, and automatically added to the classpath and also packaged in the final JAR file.

Some IDEs don't recognize these directories as source code, giving you compilation errors. To avoid these problems, register these directories as source directories in IDE configuration or using Maven.

In a terminal window, run the following command to generate these classes:

```
./mvnw clean compile
```

The final step is to implement the business logic executed by the *gRPC server*.

Generated classes are packaged in the package defined at the `java_package` option defined in the `fraud.proto` file; in this case, it is `org.acme.stub`.

To implement the service, create a new class annotated with `@net.devh.boot.grpc.server.service.GrpcService` and extend the base class `org.acme.stub.FraudDetectionGrpc.FraudDetectionImplBase` generated previously by protobuf plugin which contains all the code for binding the service.

```
@GrpcService
public class FraudDetectionInferenceGrpcController
```

```
    extends org.acme.stub.FraudDetectionGrpc.FraudDetectionImplBase { ❶
}
```

❶ Base class name is the servicename defined in the `fraud.proto`

Since the project uses the Spring Boot framework, you can inject dependencies using the `@Autowired` or `@Resource` annotations.

Inject of the `ai.djl.inference.Predictor` class as done in the REST Controller to access the model:

```
@Resource
private Supplier<Predictor<org.acme.TransactionDetails, Boolean>>
 predictorSupplier;
```

Finally, implement the `rpc` method defined in `fraud.proto` file under `FraudDetection` service. This method is the remote method invoked when the *gRPC client* makes the request to the Inference API.

Because of the streaming nature of *gRPC*, the response is sent using a reactive call through the `io.grpc.stub.StreamObserver` class.

```
@Override
public void predict(TxDetails request,
        StreamObserver<FraudResponse> responseObserver) { ❶

    org.acme.TransactionDetails td = ❷
        new org.acme.TransactionDetails(
            request.getTxId(),
            request.getDistanceFromLastTransaction(),
            request.getRatioToMedianPrice(),
            request.getUsedChip(),
            request.getUsedPinNumber(),
            request.getOnlineOrder()
 );

    try (var p = predictorSupplier.get()) { ❸

        boolean fraud = p.predict(td);

        FraudRes fraudResponse = FraudRes.newBuilder()
 .setTxId(td.txId())
 .setFraud(fraud).build(); ❹

        responseObserver.onNext(fraudResponse); ❺
        responseObserver.onCompleted(); ❻
 } catch (TranslateException e) {
            throw new RuntimeException(e);
 }
}
```

❶ RPC Method receives input parameters and the `StreamObserver` instance to send the output result.

❷ Transforms the *gRPC* messages to DJL classes.

❸ Gets the predictor as we did in the REST Controller

❹ Creates the *gRPC* message for the output

❺ Sends the result

❻ Finishes the stream for the current request

Both REST and *gRPC* implementations can coexist in the same project. Start the service with the `spring-boot:run` goal to notice that both endpoints are available:

```
./mvnw clean spring-boot:run

o.s.b.w.embedded.tomcat.TomcatWebServer: Tomcat started on port 8080
                                            (http) with context path '/'
n.d.b.g.s.s.GrpcServerLifecycle: gRPC Server started,
                                    listening on address: *, port: 9090
```

Sending requests to a *gRPC server* is not as easy as with REST; you can use tools like `grpc-client-cli` (*https://github.com/vadimi/grpc-client-cli*), but in the following chapter, you'll learn how to access both implementations from Java.

Next Steps

You've completed the first step in inferring models in Java. DJL has more advanced features, such as training models, automatic download of popular models (*resnet*, *yolo*, …), image manipulation utilities, or transformers.

This chapter's example was simple, but depending on the model, things might be more complicated, especially when images are involved.

In later chapters, we'll explore more complex examples of inferencing models using DJL and show you other useful enterprise use cases and models.

In the next chapter, you'll learn how to consume the Inference APIs defined in this chapter before diving deep into DJL.

CHAPTER 4

Accessing the Inference Model with Java

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

In the previous chapter, you learned to develop and expose a model that produces data using an Inference API. That chapter covered half of the development; you only learned how to expose the model, but how about consuming this model from another service? Now it is time to cover the other half, which involves writing the code to consume the API.

In this chapter, we'll complete the previous example, you'll create Java clients to consume the Fraud Inference APIs to detect if a given transaction can be considered fraudulent or not.

We'll show you writting clients for Spring Boot and Quarkus using both REST and gRPC clients.

Connecting to an Inference API with Quarkus

Quarkus provides two methods for implementing REST Clients:

- The Jakarta REST Client is the standard Jakarta EE approach for interacting with RESTful services.
- The MicroProfile REST Client provides a type-safe approach to invoke RESTful services over HTTP using as much of the Jakarta RESTful Web Services spec as possible. The REST client is defined as a Java interface, making it type-safe and providing the network configuration using Jakarta RESTful Web Services annotations.

In this section, you'll develop a Quarkus service consuming the Fraud Detection model using the MicroProfile REST Client.

Architecture

Let's create a Quarkus service sending requests to the Fraud Service Inference API developed in the previous chapter.

This service contains a list of all transactions done and exposes an endpoint to validate whether a given transaction ID can be considered fraudulent.

The Figure 4-1 shows the architecture of what you'll be implementing in this chapter.

Quarkus service receives an incoming request to validate whether a given transaction is fraudulent. The service gets the transaction information from the database and sends the data to the fraudulent service to validate whether the transaction is fraudulent. Finally, the result is stored in the database and returned to the caller.

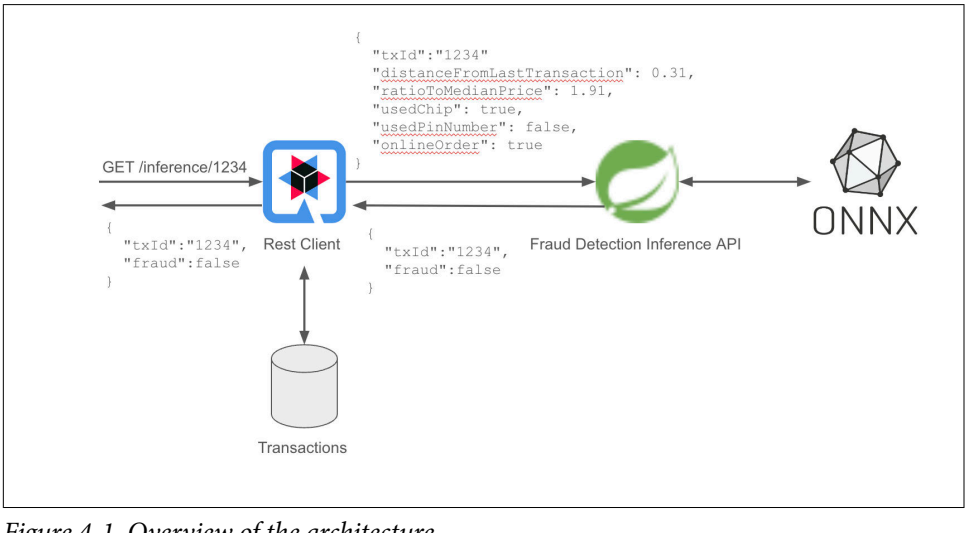


Figure 4-1. Overview of the architecture

Let's remember the document format returned by the inference API, as it is important to implement it correctly on the client side.

The Fraud Inference API

The Fraud Inference API developed in the previous chapter uses the HTTP POST method, exposing the `/inference` endpoint and JSON documents as body requests and responses.

An example of body content could be:

```
{
    "txId": "5678",
    "distanceFromLastTransaction": 0.3111400080477545,
    "ratioToMedianPrice": 1.9459399775518593,
    "usedChip": true,
    "usedPinNumber": false,
    "onlineOrder": false
}
```

And a response:

```
{
    "txId":"5678",
    "fraud":true
}
```

Let's scaffold a Quarkus project to implement the consumer part.

Creating the Quarkus project

First, generate a simple Quarkus application with REST Jackson and REST Client Jackson dependencies. You can use Code Quarkus to scaffold the project or start from scratch.

With the basic layout of the project, let's write the REST Client using the MicroProfile REST Client spec.

REST Client interface

Create the `org.acme.FraudDetectionService` interface to interact with the Inference API.

In this interface, you define the following information:

- The connection information using JakartaEE annotations (`@jakarta.ws.rs.Path` for the endpoint and `@jakarta.ws.rs.POST` for the HTTP Method) .
- The classes used for body content and response.

- Annotate the class as a REST Client with the `@org.eclipse.microprofile.rest.client.inject.RegisterRestClient` annotation and set the client's name.

```
@Path("/inference") ❶
@RegisterRestClient(configKey = "fraud-model") ❷
public interface FraudDetectionService {

    @POST ❸
    FraudResponse isFraud(TransactionDetails transactionDetails); ❹ ❺
}
```

❶ Remote Path to connect

❷ Sets the interface as REST client

❸ The request uses the `POST` HTTP method

❹ `TransactionDetails` is serialized to JSON as body message

❺ `FraudResponse` is serialized to JSON as response

Host to connect is set in the `application.properties` file with the `quarkus.rest-client.<configKey>.url` property. Open the `src/main/resources/application.properties` file and add the following line:

```
quarkus.rest-client.fraud-model.url=http://localhost:8080 ❶ ❷
```

❶ `configKey` value was set to `fraud-model` in the `RegisterRestClient` annotation

❷ The inference API is deployed locally

With a few lines, we've developed the REST Client and it's ready for use. The next step is creating the REST endpoint.

REST Resource

The next step is to create the REST endpoint, which will call the REST client created earlier. The endpoint is set up to handle requests using the `GET` HTTP method, and it is implemented with the `@jakarta.ws.rs.GET annotation. The transaction ID is passed as a path parameter using the `@jakarta.ws.rs.PathParam annotation.

To use the REST client, you should inject the interface using the `@org.eclipse.microprofile.rest.client.inject.RestClient annotation.

Please create a class called `TransactionResource` with the following content:

```
@Path("/fraud")
public class TransactionResource {

    // ....

    @RestClient ❶
    FraudDetectionService fraudDetectionService;

    @GET
    @Path("/{txId}") ❷
    public FraudResponse detectFraud(@PathParam("txId") String txId) { ❸

        final TransactionDetails transaction = findTransactionById(txId);

        final FraudResponse fraudResponse = fraudDetectionService.isFraud(transaction); ❹
        markTransactionFraud(fraudResponse.txId(), fraudResponse.fraud());

        return fraudResponse;
    }

    // ....

}
```

❶ Interface is injected

❷ Defines the path param

❸ Injects the path param value as method parameter

❹ Executes the remote call to Inference API

The service is ready to start using the inference model.

Set the `quarkus.http.port=8000` property in the `application.properties` file to start this service in port `8000` so it doesn't collide with the Spring Boot port.

Testing the example

To test the example, you need to start the Spring Boot service developed in the previous chapter and the Quarkus service developed in this chapter.

In one terminal window, navigate to the Fraud Detection Inference directory and start the Spring Boot service by running the following command:

```
./mvnw clean spring-boot:run
```

In another terminal window, start the Quarkus service running the following command:

```
./mvnw quarkus:dev
```

With both services running, send the following request to the `TransactionResource` endpoint:

```
curl localhost:8000/fraud/1234

{"txId":"1234","fraud":false}
```

You consumed an Inference API using Quarkus; in the next section, we'll see implementing the same consumer using Spring Boot.

Connecting to an inference API with Spring Boot WebClient

Let's implement a REST client but at this time using Spring Boot WebFlux classes.

`WebClient` is an interface serving as the primary entry point for executing web requests, replacing the traditional `RestTemplate` classes. Furthermore, this new client is a reactive, non-blocking solution that operates over the HTTP/1.1 protocol, but it is suitable for synchronous operations.

Adding WebClient Dependency

We can use `WebClient` with synchronous and asynchronous operations, but the client is under reactive dependencies.

Add the following dependency if the project is not already a WebFlux service.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

With the dependency registered, let's implement the code to make REST calls.

Using the WebClient

To call remote REST services, instantiate the `org.springframework.web.reactive.function.client.WebClient` interface as a class attribute. Then, we'll use this interface to create the request call and retrieve the result.

```
private final WebClient webClient;

public TransactionController() {
    webClient = WebClient.create("http://localhost:8080"); ❶
}
```

```
@GetMapping("/fraud/{txId}")
FraudResponse detectFraud(@org.springframework.web.bind.annotation.PathVariable
                          String txId) {

    final TransactionDetails transaction = findTransactionById(txId);

    final ResponseEntity<FraudResponse> fraudResponseResponseEntity = webClient.post() ❷
        .uri("/inference") ❸
        .body(Mono.just(transaction), TransactionDetails.class) ❹
        .retrieve() ❺
        .toEntity(FraudResponse.class)
        .block(); ❻

    return fraudResponseResponseEntity.getBody();
}
```

❶ Creates and configures the `WebClient`

❷ Instantiates a new instance to execute a `POST`

❸ Sets the path part

❹ Body content

❺ Executes the call

❻ Transforms the async call to sync

So far, you have used both frameworks to consume an Inference API with two different approaches: declarative and programmatic. You can integrate any Java (REST) HTTP client without issues.

Let's now implement the same logic for consuming a model but using gRPC protocol instead of REST.

Connecting to the Inference API with Quarkus gRPC client

Let's build a *gRPC client* with Quarkus to access the Fraud Detection model exposed as the *gRPC server* built in the previous chapter.

As you did when implementing the server-side part, you need to generate the *gRPC Stub* from the protobuf file.

Quarkus only requires you to register `quarkus-grpc` and `quarkus-rest` extensions.

Adding gRPC Dependencies

Open `pom.xml` file of the Fraud Client project and under the `dependencies` section add the following dependency:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-grpc</artifactId>
</dependency>
```

You should already have the `quarkus-rest` dependency registered as you are reusing the project.

With the dependency registered, let's implement the code to make gRPC calls.

Implementing the gRPC Client

Create the `fraud.proto` file under `src/main/proto` directory with the following content:

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.acme.stub";

package fraud;


service FraudDetection {
 rpc Predict (TxDetails) returns (FraudRes) {}
}

message TxDetails {
 string tx_id = 1;
 float distance_from_last_transaction = 2;
 float ratio_to_median_price = 3;
 bool used_chip = 4;
 bool used_pin_number = 5;
 bool online_order = 6;
}

message FraudRes {
 string tx_id = 1;
 bool fraud = 2;
}
```

It is the same file created in the server-side project, so you can copy it or put it in a shared project and add the project as an external dependency.

With this setup, you can place the protobuf file in the `src/main/proto` directory. The `quarkus-maven-plugin` (already present in any Quarkus project) will then generate Java files from the proto files.

Under the hood, the `quarkus-maven-plugin` fetches a compatible version of `protoc` from Maven repositories based on your OS and CPU architecture.

At this point, every time you compile the project through Maven, the `quarkus-maven-plugin` generates the required *gRPC* classes from the `.proto` file. These classes are generated at the `target/generated-sources/grpc` directory, automatically added to the classpath, and packaged in the final JAR file.

Some IDEs don't recognize these directories as source code, giving you compilation errors. To avoid these problems, register these directories as source directories in IDE configuration or using Maven.

In a terminal window, run the following command to generate these classes:

```
./mvnw clean compile
```

The final step is sending requests using the *gRPC client*, which uses the classes generated in the previous step.

Inject the generated service interface with the name `org.acme.stub.FraudDetection` into the `TransactionResource` class using the `@io.quarkus.grpc.GrpcClient` annotation.

```
@GrpcClient("fraud") ❶
FraudDetection fraud;
```

❶ Inject the service and configure its name.

Quarkus provides a runtime implementation for the interface that is similar to the REST Client.

When implementing the server-side part, you have seen that *gRPC* applications are inherently reactive. Quarkus uses the project *Mutiny* to implement reactive applications, similar to Spring Boot WebFlux or ReactiveX, and it integrates smoothly with *gRPC*.

Mutiny uses the `io.smallrye.mutiny.Uni` class to represent a lazy asynchronous operation that generates a single item. Since the Fraud Detection service returns a single result (fraud or not), the `Uni` class is used as the return type by the *gRPC client*.

Let's implement a new endpoint to verify if a transaction is fraudulent, but using *gRPC* instead of REST.

```
 @GET
@Path("/grpc/{txId}")
public Uni<FraudResponse> detectFraudGrpcClient( ❶
        @PathParam("txId") String txId) {

    final TransactionDetails tx = findTransactionById(txId);

    final TxDetails txDetails = TxDetails.newBuilder() ❷
        .setTxId(txId)
        .setDistanceFromLastTransaction(tx.distanceFromLastTransaction())
        .setRatioToMedianPrice(tx.ratioToMedianPrice())
        .setOnlineOrder(tx.onlineOrder())
        .setUsedChip(tx.usedChip())
        .setUsedPinNumber(tx.usedPinNumber())
        .build();

    final Uni<FraudRes> predicted = fraud.predict(txDetails); ❸
    return predicted
        .onItem() ❹
        .transform(fr -> new FraudResponse(fr.getTxId(), fr.getFraud())); ❺
}
```

❶ Reactive endpoint, not necessary to block for the result

❷ *gRPC* input message

❸ Makes the remote call

❹ For the message item returned by the service

❺ Transforms the *gRPC* message to required output type

The last step before running the example is configuring the remote service location in the `application.properties` file:

```
quarkus.grpc.clients.fraud.host=localhost ❶
quarkus.grpc.clients.fraud.port=9090
```

❶ `fraud` is the name used in the `@GrpcClient` annotation

These are all the steps required for using a *gRPC client* in a Quarkus application.

Going Beyond

So far, we have looked at using Inference APIs as REST or gRPC using standard Java libraries that were not specifically designed for AI/ML. This approach works well for cases where the model is stateless and can be used for a single purpose, such as detecting fraud or calculating embeddings.

However, when using large language models like Llama3, OpenAI, and Mistral, a plain REST client might not be sufficient to meet all the requirements. For instance:

- Models are stateless, but in some scenarios, it's crucial to know what was asked before in order to generate a correct answer. Generic clients do not have memory features.
- Using RAG is not directly supported by clients.
- There is no agent support.
- You need to implement the specific Inference API for each model you use.

For these reasons, there are some projects in the Java ecosystem to address these limitations. The most popular one is **LangChain4J**.

In the next chapter, we'll introduce you the LangChain4J project and discuss how to use it when interacting with LLM models.

CHAPTER 5
LangChain4j

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

The previous chapter introduced how to consume AI models as an inference model API. It works for simple problems, but when you develop more complicated solutions where AI is heavily involved, you need more features than a simple request/response. In this chapter, we'll introduce LangChain4j, a framework for simplifying the integration of AI/LLM capabilities into Java applications providing high-level capabilities like memory, or data augmentation.

We'll cover examples using plain Java and its integration with Quarkus and Spring Boot so you can get a full picture of its use in different Java projects.

In this chapter, you'll learn Langchain4j from the basics to advanced scenarios using prompting, memory, data augmentation, image processing, but will save RAG for the next chapter.

What is LangChain4j?

LangChain4j is a Java implementation inspired by the popular Python LangChain framework. It helps developers build applications that integrate with LLMs. LangChain4j provides tools and abstractions to simplify the integration of LLMs into Java-based applications, enabling functionalities like natural language processing, text generation, question-answer, and more.

As the LangChain framework, LangChain4J offers some features to simplify the development of applications that integrate with LLMs.

Let's dig into some of these key features.

Unified APIs

As yu saw in Chapters 3 and 4, LLMs offer different APIs to access them. For example, the API to access the OpenAI ChatGPT model might differ from the one to access a HuggingFace model or the models embedded into the Java Virtual Machine using projects like *JLama* or *llama3*.

The same with embedding models or vector stores like *PGVector* or *Chroma*, which use proprietary protocols to communicate with the server.

LangChain4j provides a unified API, so you can easily switch between them without rewriting your code (or with minimal changes). It is like the well-known Java Persistence API (JPA) but for abstracting for models instead of databases.

In terms of language models, LangChain4j offers integration with more than 15 models, such as Anthropic, Google AI Gemini, HuggingFace, OpenAI, JLama, Mistral AI, Ollama, or Qwen.

The Figure 5-1 shows the relationship between a Java application, LangChain4j, and the models.

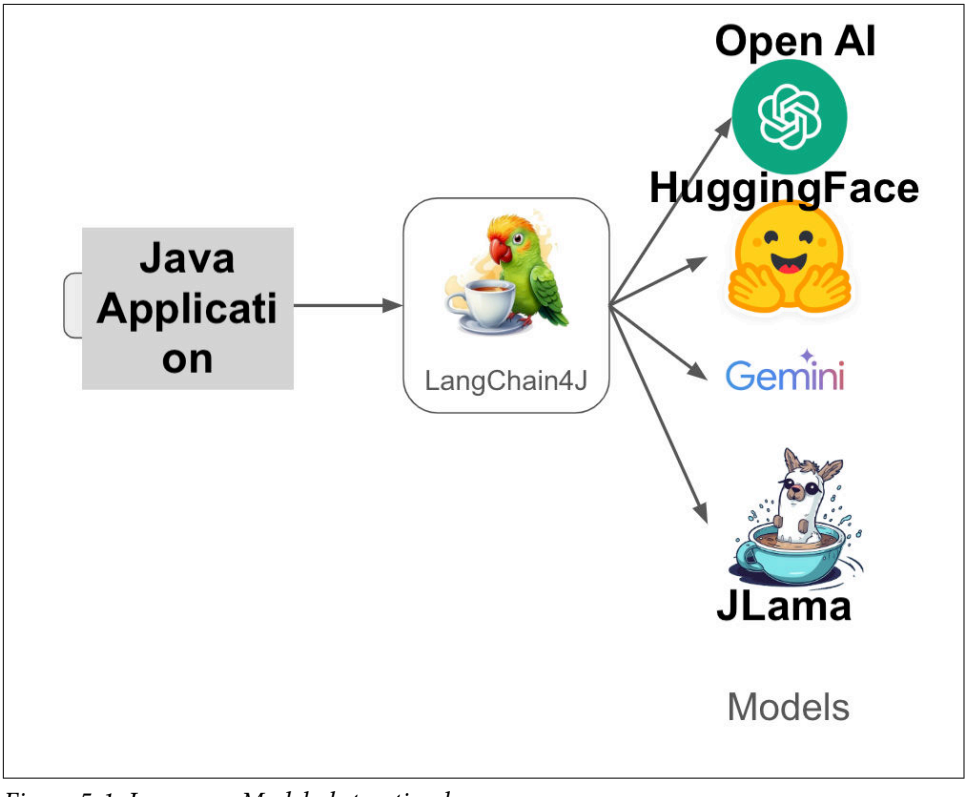


Figure 5-1. Language Model abstraction layer

Every supported model should implement `dev.langchain4j.model.chat.ChatLanguageModel` in order to be LangChain4j compliant.

For example, if you want to use the OpenAI model, you should add the following dependency into your classpath:

```
<dependency>
    <groupId>dev.langchain4j</groupId>
    <artifactId>langchain4j-open-ai</artifactId>
    <version>....</version>
</dependency>
```

And instantiate the `ChatModel` interface using the `dev.langchain4j.model.openai.OpenAiChatModel` class:

```
static ChatModel model = OpenAiChatModel.builder() ❶
 .apiKey(OPENAI_API_KEY) ❷
 .modelName(GPT_4_O_MINI) ❸
 .build();
```

❶ Instantiates the object for interacting with OpenAI

❷ Sets the API key to access to the model

❸ Uses *GPT-4o mini* model

The `ChatModel` interface contains methods to send requests to the model and get the response. Typically, you use one of the multiple overloads of `chat` method:

```
String output = model.chat("Who is Lionel Messi");
```

Don't worry if this isn't quite clear yet. Right now we are just giving you a basic introduction to the LangChain4j API, but we'll provide you with multiple full running examples later in this chapter.

LangChain4j supports more than 20 embedding models (mostly used to calculate vectors), such as *ONNX* models, OpenAI, HuggingFace, Cohere, LocalAI, etc.

It also integrates with over 20 vector databases, including Chroma, DuckDB, ElasticSearch, PGVector, Neo4J, and Redis.

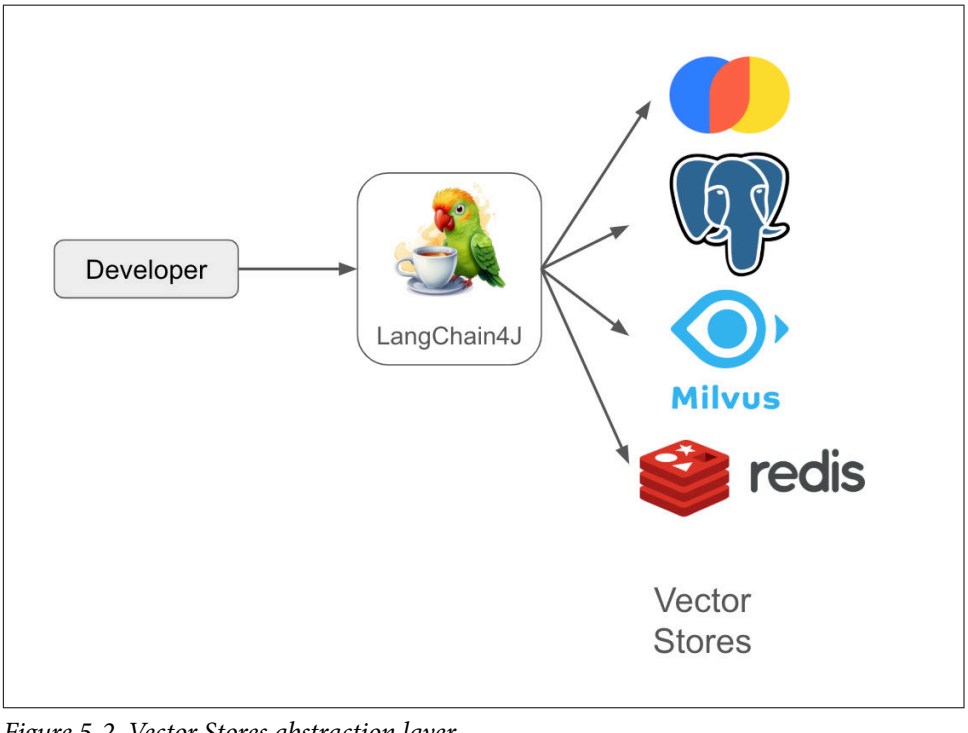


Figure 5-2. Vector Stores abstraction layer

Prompt Templates

LangChain4j provides tools for creating and managing prompt templates, making structuring inputs for the LLMs easier.

A prompt is the input text provided to the model to generate a response. The prompt's quality, clarity, and structure significantly influence the model's output. An example of an input prompt might be:

```
Please give me a list of all Studio Ghibli movies.

The output should be in a JSON array format, with the movie's title
in the name field and the release date in the year field.
For example:

[
 {
 "name": "My Neighbor Totoro",
 "year": 1988
 }
]
```

In the above prompt, you describe what you want and how the model should provide the information.

Moreover, input prompts have two roles: *system message* and *user message*.

A **system message** is a special type of message used to set the behavior or context for the model in a conversation. It is typically the first part of a message in a conversation and guides the model's behavior, tone, or role throughout the conversation. For example, you could provide the following system message: "You are a helpful assistant who speaks like a pirate." This would make the model generate the output as a pirate.

A **user message** is the text or query that the user wants the model to respond to. For example, "What is the weather today? " assuming the model is able to respond to this question and using the previous system message, a possible response in the style of a pirate could be: "Arrr, matey! The skies be clear, and the sun be shinin' bright today!"

In LangChain4j, there are two ways to set the prompt: declaratively or programmatically.

Declaratively, you could do this using either `@dev.langchain4j.service.UserMessage` or `@dev.langchain4j.service.SystemMessage`, or programmatically by instantiating the `dev.langchain4j.model.input.Prompt` class.

A Prompt can contain placeholders resolved at runtime to make them more dynamic; for example, it is valid to set the prompt message as `What is the capital of {{country}}` and provide the `country` value at runtime.

If you use Quarkus, a prompt can also be a *Qute* expression. *Qute* is a templating engine explicitly developed for Quarkus.

LangChain4j has also *Structured Prompts*. The idea is to resolve the prompt placeholders from a Java object. You annotate the Java class holding the values with the `@dev.langchain4j.model.input.structured.StructuredPrompt`. The annotation defines the prompt template with placeholders referring to the fields of the class.

The following prompt uses a Structured Prompt to request the model to generate a story about a given topic where a list of characters should appear:

```
@StructuredPrompt({ ❶
    """
    Create a story about {{story}} where appear the
     following characters {{characters}}.
     """, ❷
})
class CreateStoryPrompt {

    String story; ❸
    List<String> characters; ❹
}
```

❶ Annotates the class to define the prompt

❷ Defines the prompt with the placeholders

❸ Defines a variable to set the *story* placeholder

❹ Defines a list of Strings to set the *characters* placeholder. LangChain4J will serialize the list into placeholders.

The prompt `ChatModel` method requires an object of type `dev.langchain4j.model.input.Prompt`. To transform the annotated class to a `Prompt` instance, use the `dev.langchain4j.model.input.structured.StructuredPromptProcessor` utility class:

```
CreateStoryPrompt createStoryPrompt = new CreateStoryPrompt(); ❶
createStoryPrompt.story = "A forrest where there are animals";
createStoryPrompt.characters = List.of("Snow White", "Grumpy Dwarf");

Prompt prompt = StructuredPromptProcessor.toPrompt(createStoryPrompt); ❷
AiMessage aiMessage = model.chat(prompt.toUserMessage()) ❸
 .content(); ❹

String response = aiMessage.text(); ❺
```

❶ Creates the Java object and fills it with the required values

❷ Transforms the Java instance to a `Prompt` object

❸ Creates the prompt as a *user message* role

❹ Using `Prompt`, the model returns an object of type `dev.langchain4j.data.message.AiMessage` containing the textual response

❺ Gets the response

Prompting is a key subject in LangChain4J, and we'll go through plenty of examples later in this chapter.

Structured Outputs

Most modern LLMs support generating outputs in a structured format, typically JSON. You've already seen this in the ??? section.

LangChain4J can automatically process these outputs and map them to Java objects. It supports the following return types:

`java.lang.String`
: It is the textual response in the String object.

`dev.langchain4j.data.message.AiMessage`
: The message can contain either a textual response or a request to execute one/multiple tool(s).

`Any custom POJO`
: If the output is in JSON format, then LangChain4J automatically unmarshal the JSON output to the specified object. Moreover, you use this feature to extract information from unstructured text to a class, using field names to extract the information and fill it in the object.

Any `Enum` *or* `List<Enum>` *or* `Set<Enum>`
: LangChain4J maps the output as an Enum. For example, if the output is a string containing only *Positive* and the `Sentiment`, JAVA enum has an entry named *POSITIVE*, this instance is automatically created.

`boolean/Boolean`
: Matches a *yes*/*no*, *true*/*false*, … to a boolean.

`byte/short/int/BigInteger/long/float/double/BigDecimal`
: Transforms to any of these numerical objects.

`Date/LocalDate/LocalTime/LocalDateTime`
: Transforms to any of these date/time objects.

`List<String>/Set<String>`
: To get the answer in a list of bullet points.

`Map<K, V>`
: Map representation of the output.

`dev.langchain4j.service.Result<T>`
: It contains the LLM response, and additional information associated with it, such as `dev.langchain4j.model.output.TokenUsage`, `dev.langchain4j.model.output.FinishReason`, `dev.langchain4j.rag.content.Content`'s retrieved during RAG.

Let's take a look an example of extracting an unstructured text into a Plain Old Java Object (POJO).

Suppose you need to extract information about a bank transaction. The text you'd sent as the prompt is: "Extract information about a transaction from My name is Alex; I completed a transaction on July 4th, 2023 from my account with IBAN 123456789 in the amount of $25.5"

You could fill the following Java record class if you set the return type to `TransactionInfo`:

```
record TransactionInfo(String fullName, String iban,
                LocalDate transactionDatet, double amount) {
}
```

You'll develop this example later in the chapter, with all the required classes, as there is still a piece of code that needs to be covered.

Memory

LangChain4j includes memory management features, allowing applications to maintain context across multiple interactions with an LLM.

LLMs have no state; every request has no context of previously asked questions and their responses. This might work in some use cases, but if you need to maintain a conversation about the model, for example, implementing a chatbot to resolve issues with a reservation, stateless conversations are not an option.

To implement state to a model, you append the previous conversation taken with the model with the current request. In this way, the model has the context of previous conversations, you pass the *memory* to the model, so it can *remember* the conversation, and produce a response based on that.

Maintaining and managing messages manually is cumbersome, LangChain4j offers a `dev.langchain4j.memory.ChatMemory` interface and multiple implementations to automate storing, managing, and appending the *memory* to the user message.

Moreover, LangChain4j implements some eviction strategies to remove messages from storage automatically.

Eviction is important because it impacts the application's performance and cost. Each token incurs a **cost**, meaning longer conversations become increasingly expensive. Also, the more tokens sent to the LLM, the **longer** it takes to generate a response. Evicting excess messages can improve processing speed.

In addition, LLMs have a maximum token limit they can process in a single interaction. If a conversation exceeds this limit, specific messages, typically the oldest ones, need to be removed.

LangChain4J provides two strategies at the time of writing this book:

Sliding Window
: Keep the *N* most recent messages and remove older ones that exceed the limit. It is implemented in `dev.langchain4j.memory.chat.MessageWindowChatMemory` class.

Token Sliding Window
: This window retains the *N* most recent tokens, evicting older messages as needed. Messages are indivisible. It is implemented in `dev.langchain4j.memory.chat.TokenWindowChatMemory` class.

In the following snippet, we show you how to use the memory feature to record the interaction between a user and the model:

```
ChatMemory chatMemory = MessageWindowChatMemory.builder() ❶
 .maxMessages(20) ❷
 .build();

UserMessage userMessage1 = userMessage(
                "How do I write a REST endpoint in Java using Quarkus? "); ❸
chatMemory.add(userMessage1); ❹

final Response<AiMessage> response1 = model.chat(chatMemory.messages());❺
chatMemory.add(response1.content()); ❻

UserMessage userMessage2 = userMessage(
                "Create a test of the first point? " +
                "Be short, 15 lines of code maximum."); ❼
chatMemory.add(userMessage2);
model.chat(chatMemory.messages()); ❽
```

❶ Creates an in-memory message memory instance

❷ Only handle 20 messages

❸ Defines the first prompt

❹ Adds the prompt into memory

❺ Sends the memory to the model

❻ Adds the response into the memory

❼ Defines the second prompt

❽ Sends the first prompt, the response from the first prompt, and the second prompt

In this example, the model generates a Quarkus test even though you never refer to Quarkus in the second prompt. You are sending all the memory to the model, so when you refer to *the first point*, the model knows you are referring to the previous prompt.

The `ChatMemory` object is created per user. This means you should provide an instance of this object for every user. You don't want to share your memories with everybody after all.

As we mentioned, LangChain4j stores messages in an in-memory instance, but the API is open to providing any other implementation. LangChain4j provides some out-of-the-box implementations, but you can provide one by implementing the `dev.langchain4j.store.memory.chat.ChatMemoryStore` interface.

To inject a `ChatMemoryStore` into the `ChatMemory`, use the `chatMemoryStore()` method to override the default in-memory storage with a persistence storage. In the following example, you use a Redis to store the chat messages:

```
RedisChatMemoryStore redisStore = new RedisChatMemoryStore(...); ❶

ChatMemory chatMemory = MessageWindowChatMemory.builder()
 .id("abcd") ❷
 .maxMessages(10)
 .chatMemoryStore(redisStore) ❸
 .build();
```

❶ Creates the Redis Store with the required connection parameters

❷ Sets an ID to identify this object

❸ Injects the Memory Store instance

The interface is the same, but now messages are stored inside a Redis instance instead of in memory store.

Data Augmentation

A model can only give answers with the data you (or someone else) trained on. For example, if you trained the model in 2024 and asked about any event that happened during 2025, the model might respond with something like *I don't know* or hallucinate and provide you with some invented response.

But the same can happen when you ask for live information, for example, "What is the weather today in Berlin?" Obviously, it cannot provide an answer.

Data augmentation is a method for retrieving and incorporating relevant information into the prompt before sending it to the model. By doing this, the model receives pertinent context with the prompt message, generating responses based on correct information.

In the Figure 5-3, you see how an application gets the weather information from a REST Endpoint, appends it to the prompt, and sends it to the model to generate the weather forecast.

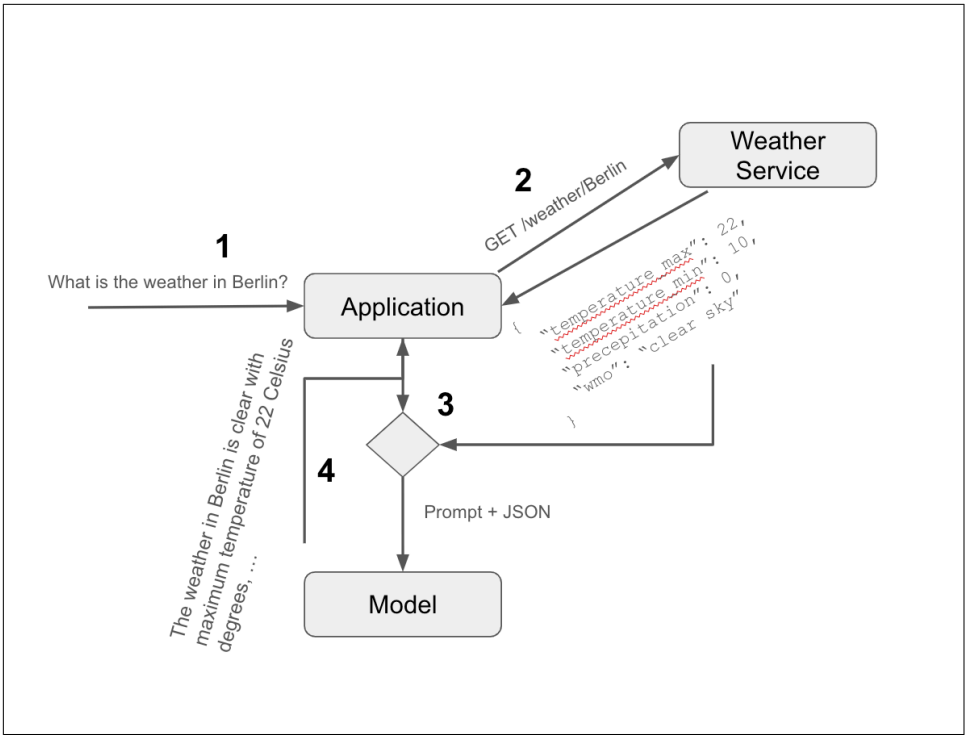


Figure 5-3. Data Augmentation

The steps illustrated in the figure are:

1. First, the user sends a chat message asking about the *weather in Berlin*, and the application extracts the city name.
2. The second step is to send a request to a weather system to get data about Berlin's weather in JSON format.
3. Third is appending the *system message*, the *user message*, and the JSON data and sending it to the model.
4. Fourth, with all this information, the model generates a response using a narrative weather forecast, and returns to the application.

LangChain4j provides the `dev.langchain4j.rag.content.retriever.ContentRetriever` interface to implement the data augmentation logic (in our example getting the JSON document from weather service). LangChain4j has out-of-the-box implementations for enriching data from a web search engine or a vector store.

Let's implement the `ContentRetriever` class for getting weather information from a service and transform the output to be consumed by the model:

```
public class WeatherContentRetriever implements ContentRetriever {

    @Override
    public List<Content> retrieve(Query query) { ❶

        String city = query.text(); ❷

        JsonObject json = getWeatherFromCity(city); ❸
        return List.of(
                Content.from(json.toString()) ❹
        );
    }

}
```

❶ `dev.langchain4j.rag.query.Query` provides the city (*Berlin*, *Barcelona*, …)

❷ Gets the text from the query

❸ Makes the REST call

❹ `dev.langchain4j.rag.content.Content` with the response as text. Optionally, you add metadata too

Finally, wrap the `ContentRetriever` implementation into `dev.langchain4j.rag.DefaultRetrievalAugmentor` class.

We'll explore this class deeply later in the chapter. For now, keep in mind that data can be augmented from multiple sources, and depending on the type of query. For example, if a user asks about the weather, the query is routed to a weather service retriever, while if he asks about flights between two cities, the query is routed to a web search engine retriever.

The below code wraps the weather augmentator into the `DefaultRetrievalAugmentor` class:

```
UserMessage userMessage = UserMessage.from("Berlin"); ❶
Metadata metadata = Metadata.from(userMessage, null, null);

DefaultRetrievalAugmentor rAugmentor = DefaultRetrievalAugmentor.builder()
 .contentRetriever(weatherContentRetriever)
 .build(); ❷

AugmentationRequest aRequest = new AugmentationRequest(
 userMessage, Metadata metadata);

AugmentationResult aResult = rAugmentor.augment(aRequest); ❸

model.chat(aResult.chatMessage()); ❹
```

❶ Creates the prompt as a user message

❷ Instantiates the Retrieval Augmentator, injecting the weather content retriever

❸ Augmentates the original prompt with the weather information

❹ Sends the new user message (original prompt + weather info)

You can augment the prompt from any data source, such as a remote service, a database, or a web search engine, but the most common use case is from a vector store.

In the following chapter, we'll explore using Data Augmentation (or Retrieval-Augmented Generation, or RAG) for vector stores.

Tools

LangChain4j supports calling functions from the model to the service as long as the model supports too. I know this sounds wild, but there are some use cases where the model itself is unable to perform a task or where you prefer to run it in the service. In these situations, you can configure the model to send a call to a function defined in the service to get the required information before proceeding to the generation part.

Consider the following prompt: "*Write a poem about {{topic}}. The poem should be {{lines}} lines long. Then send this poem by email.*".

The first part of the prompt is easier for an LLM, but a model is unable to send an email. To solve this problem, you could use *tooling* to send the prompt with the signature of a method invoked by the model in the described circumstances.

The model invokes the function, but the service side executes it; the model side will not execute anything.

Figure 5-4 shows the workflow of the summary of a function calling with the example of sending an email:

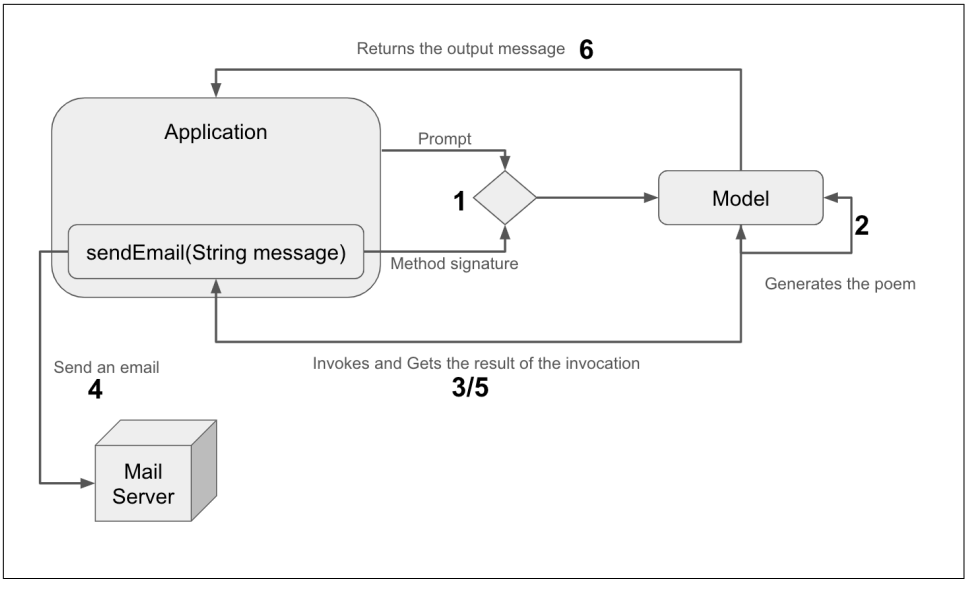


Figure 5-4. Function Calling

1. First, the prompt and the **method signature** are appended to the same request and sent to the model. This way, the model knows what methods can be called. In addition to the method signature, a description of when (or in which cases) this method should be invoked is also sent.
2. Second, the model processes the first part of the user message, *write a poem about…*, until it arrives at the part of *send this poem by email*.
3. Third, since the model doesn't know how to send an email, it checks the list of functions you provided to see if there is any function calling for this, and if so, it invokes it.
4. Fourth, the email method is executed, and the application sends the email with the poem.
5. Fifth, it returns the result, saying it was possible to send an email.
6. Sixth, get the application result, saying if it was possible to process all the instructions.

This is a simple example, but you can extend it to other use cases, such as executing Database queries or Math operations.

Two things are required in terms of code: the first one is to annotate the exposed function with the `dev.langchain4j.agent.tool.Tool` annotation. The second one, append the class signature to the prompt using the `dev.langchain4j.agent.tool.ToolSpecifications` helper class:

```
class EmailService {

 @Tool("send given content by email") ❶
    public void sendAnEmail(String content) { ❷
        sendEmail(
            "origin@quarkus.io",
            "sendMeALetter@quarkus.io",
            "A poem for you",
 content
 );
 }
}

List<ToolSpecification> toolSpecifications = ToolSpecifications
 .toolSpecificationsFrom(EmailService.class); ❸

String prompt = "Write a poem about....send this poem by email.";

ChatRequest chatRequest = ChatRequest.builder()
 .messages(UserMessage.from(prompt))
 .toolSpecifications(toolSpecifications)
 .build(); ❹

ChatResponse chatResponse = model.chat(chatRequest); ❺
```

❶ Describes when the model should invoke the function

❷ Parameter is the generated content from the model

❸ Creates the signature method to send to model

❹ Creates the request object appending the prompt with the method signature

❺ Sends the request to the model

Then, you'll need to manually execute the required tool(s) using the details provided in the ToolExecutionRequest(s), and send the tool execution results back to the LLM.

In Chapter 9, we'll cover a complete example invoking tools in this way, but for now, we'll show you an easier way to do it.

So far, you've seen LangChain4J's low-level API, which uses components like the `UserMessage`, `ChatMemory`, or `ToolSpecifications` classes. However, LangChain4J also offers high-level APIs that declaratively hide complexity and boilerplate while maintaining flexibility.

High Level API

Using a low-level API gives significant flexibility and complete control over the usage of the API. Still, you are not focusing on business logic but writing code unrelated to the business. LangChain4j implements the **AI Services** concept, simplifying the development of LLM applications while hiding the complexities seen in the previous sections.

We will use this chapter and other chapters on the high-level API

Let's reimplement some of the previous examples but using the high-level API that AI Services provide.

The most important element in **AI Services** is defining an interface and optionally annotating it to configure it with different options, such as user message, system message, memory store, etc. Then, proxy it using the `dev.langchain4j.service.AiServices`, which implements this interface for you using the low-level api.

Prompting

Let's create a simple `Trivia` interface that responds to questions about the capitals of countries.

```
interface Trivia {
 @SystemMessage("Return the capital of given country.
 Respond only the city") ❶
 @UserMessage("What is the capital of {{country}}") ❷
    String question(@V("country") String country); ❸
}

Trivia trivia = AiServices.create(Trivia.class, model); ❹

trivia.question("Japan"); ❺
```

❶ Sets a system message for this method

❷ Sets the user message with a temple placeholder

❸ `dev.langchain4j.service.V` makes the variable a prompt template variable

❹ Proxies the interface to create an instance, injecting the configured model

❺ Invokes the AI service

Everything is configured declaratively; instantiating objects such as `UserMessage`, `UserMessage`, `Prompt`, etc., is not required. The same happens with structured outputs. The proxy created around the `Trivia` interface transforms the annotations into internal calls to the low-level API.

Figure 5-5 shows the relationship between the interface and the proxy. The developer uses the interface, and under the hood, LangChain4j redirects the calls to a Java proxy implementing all the logic required to do the call to the model.

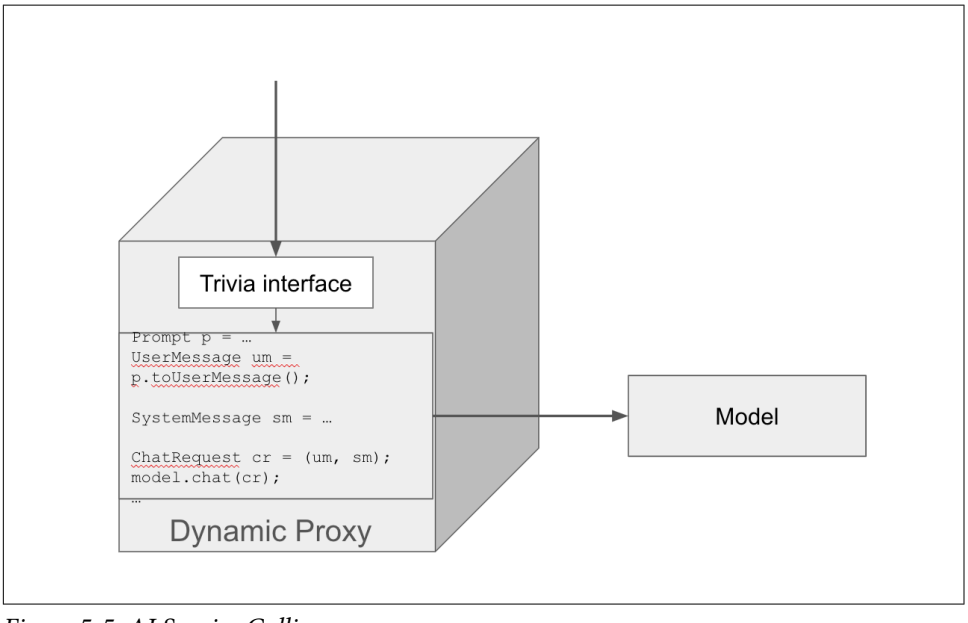


Figure 5-5. AI Service Calling

Similarly, the `AiServices` class has a builder to inject the model, memory, data augmentation, or tools.

Memory

In addition to instantiating a proxy around an interface using the `create` method, AI Services implements the `builder` pattern to pass configuration elements, such as the chat memory store.

The following example shows the injection of an in-memory chat memory store instance with eviction:

```
Chat chat = AiServices.builder(Chat.class) ❶
  .chatModel(model) ❷
  .chatMemory(MessageWindowChatMemory.withMaxMessages(10)) ❸
  .build();
```

❶ Sets the interface

❷ Injects the model to use

❸ Injects the chat memory store

Remember that you are using the same `ChatMemory` instance across all service invocations. If your system has multiple users (and probably it has), that behavior might not be desirable, as one user's "memory" data would affect the context of another user.

For this reason, LangChain4j has the `dev.langchain4j.service.MemoryId` annotation to assign a memory space to an ID (or user ID). The value of a method parameter marked with `@MemoryId` will be used to locate the conversation associated with that specific user. You can think of it as a *Map*, where the key is the user ID (whatever you decide is a user ID, *session id*, *username*, *websocket id*), and as a value, the *List* with the messages belonging to the conversation for that user.

When you use the `@MemoryId` annotation, you must also configure the `ChatMemoryProvider` for the AI service. Let's see an example:

```
interface Assistant {
    String chat(@MemoryId int memoryId, ❶
    @UserMessage String userMessage); ❷
}

Assistant assistant = AiServices.builder(Assistant.class)
    .chatModel(model)
    .chatMemoryProvider(memoryId -> MessageWindowChatMemory
                                        .withMaxMessages(10)) ❸
    .build();

System.out.println(assistant.chat(1, "Hello, my name is Ada")); ❹
System.out.println(assistant.chat(2, "Hello, my name is Alexandra")); ❺
System.out.println(assistant.chat(1, "What is my name?")); ❻
System.out.println(assistant.chat(2, "What is my name?")); ❼
```

❶ Sets the memory ID as an integer

❷ Sets the whole `UserMessage` as parameter

❸ Configures the `ChatMemoryProvider`

❹ Stores the question and the response to key 1 of the memory

❺ Stores the question and the response to key 2 of the memory

❻ The model returns Ada as the context sent to the model is the conversation with id 1

❼ The model returns Alexandra

As this example illustrates, the memory feature is incredibly important when developing chat bots.

Data Augmentation & Tooling

The builder supports setting the retrieval augmentor (or the content retriever) instance as well as tools:

```
Chat chat = AiServices.builder(Chat.class)
 .chatModel(model)
 .contentRetriever(new WeatherContentRetriever()) ❶
 .tools(new EmailService()) ❷
 .build();
```

❶ Sets the `ContentRetriever` and creates the `DefaultRetrievalAugmentor` instance automatically

❷ Sets the tool instance to append to the context

The builder provides the `retrievalAugmentor` method for cases requiring a custom `RetrievalAugmentator`.

This section has simply given you a small taste of some of the capabilities offered by LangChain4j. In the rest of this chapter, we'll explore some of these concepts further, providing real executable examples against different models and showing the integration between LangChain4j and popular Java frameworks like Quarkus or Spring Boot.

LangChain4j with plain Java

Let's write some runnable examples step-by-step using LangChain4j.

Extracting Information From Unstructured Text.

In the first example, we'll use the OpenAI model to extract unstructured text into a Java object using the AI Service approach as shown in "Structured Outputs" on page 111.

First, add the `dev.langchain4j:langchain4j` dependency to use the high-level API and the `dev.langchain4j:langchain4j-open-ai` to use OpenAI as a model in your build tool.

Then, create a class to hold the information from the unstructured text. This class contains the fields to fill with the `dev.langchain4j.model.output.structured.Description` annotation. If you don't set the annotation, the model will do its best to match the required information using the field name. However, to be more

precise, we recommend using the annotation to explain exactly the purpose of the field so the model has more information to decide which information to put there.

```
public record TransactionInfo
 (
 @Description("full name") String name, ❶
 @Description("IBAN value") String iban,
 @Description("Date of the transaction") LocalDate transactionDate,
 @Description("Amount in dollars of the transaction")  double amount
 ) { }
```

❶ Annotates the field to give more context on the purpose of the field

Create a new class representing the AI Service:

```
public interface Transaction {
 @UserMessage("Extract information about a transaction from {{it}}") ❶
    TransactionInfo extract(String message); ❷
}
```

❶ `it` refers to the only parameter, and using the `@V` annotation is unnecessary.

❷ The method returns the POJO object with the information filled from the `message` field.

Finally, the `main` method, the `OpenAiChatModel` instance, is created, proxying the `Transaction` AI Service, and the method is called.

```
ChatModel model = OpenAiChatModel.builder()
 .apiKey("demo")
 .modelName(GPT_4_O_MINI)
 .build(); ❶

Transaction tx = AiServices.builder(Transaction.class)
 .chatModel(model)
 .build(); ❷

TransactionInfo transactionInfo =
    tx.extract("My name is Alex; "
        + "I did a transaction on July 4th, 2023 from my account "
            + "with IBAN 123456789 of $25.5"); ❸

System.out.println(transactionInfo);
```

❶ Generates the `ChatLanguageModel` to connect with OpenAI

❷ Creates the AI Service from the `Transaction` interface

❸ Extracts the information from the text and fills the `TransactionInfo` class

If you run the method, you'll get the following output:

```
TransactionInfo[name=Alex, iban=123456789, transactionDate=2023-07-04, amount=25.5]
```

About demo key

In this example, you are using the `demo` key. The LangChain4j community provides this key to temporarily use for demo purposes, as not all OpenAI capabilities and models are supported. It has a usage quota.

We encourage you to use your OpenAI API key for better and faster results. Also, the key might be not available all the time, might change, or you might need a different configuration to use it.

Moreover, you should set the URL to *http://langchain4j.dev/demo/openai/v1* using the `baseUrl` method located at `OpenAiChatModelBuilder` class.

All requests to the OpenAI API are routed through the LangChain4j proxy, which inserts the actual API key before sending your request to the OpenAI API. The proxy doesn't collect, store, or use your data in any manner.

Let's jump to example — text classification.

Text classification

One use case where LLMs perform well is classifying text. For example, triaging claims in customer service to assign the correct priority, categorizing software exceptions to get an overview of the most common errors in production, and so on.

However, in this example, you'll see an implementation of a system that automatically labels an issue opened in bug tracking.

Figure 5-6 shows the *label* concept in GitHub interface:

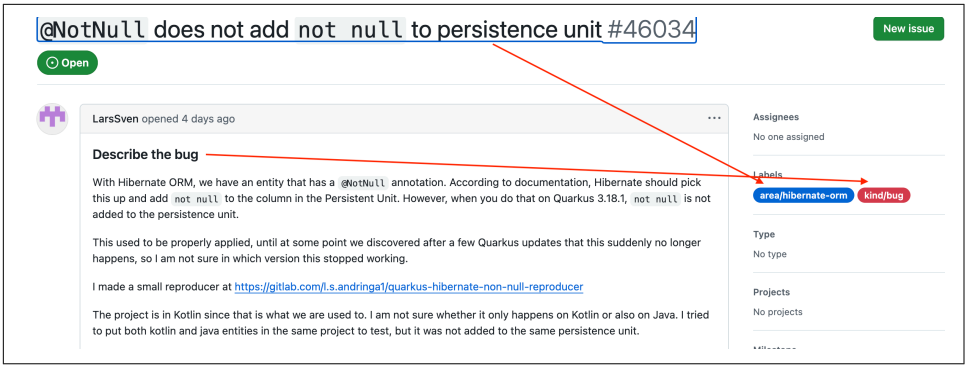


Figure 5-6. GitHub Issues

We'll not cover the bug tracker integration, only the AI part for categorizing issues from a set of possible labels.

For this example, instead of the Open AI model, we'll use the Google Gemini model.

First, add the `dev.langchain4j:langchain4j` dependency to use the high-level API and the `dev.langchain4j:langchain4j-google-ai-gemini` dependency to use Google Gemini as a model in your build tool.

Then, create an enum and a class representing the issue category:

```
public enum Label {
 PERSISTENCE, UI, EVENT, GENERIC ❶
}
```

❶ `@Description` can be used too, but for this example, it is not necessary

Even though LangChain4J works perfectly with `Enum` classes, create another class containing the `Label` enum. In this simple case, an enum might be enough, but you might need to store more information in the real world, so let's make it closer to reality.

```
public record IssueClassification(Label category){}
```

The AI Service is now a bit more complicated than before. Set a `@SystemMessage` explaining the purpose of the service. Moreover, set a `@UserMessage` explaining not only the possible category values and what the user can expect but also providing some examples to help the model classify the inputs correctly. This technique is known as *Few-shot prompting*.

```
@SystemMessage("""
 You are bot in charge of categorizing issues from a bug tracker.
 """) ❶
public interface LabelDetector {
 @UserMessage("""
```

```
    Analyze the provided issue and categorize into one of the categories.

    The issues opened are for Java projects
    so you can expect some Java acronyms,
    use them to categorize the issues as well.

    The possible values for a category must be
    PERSISTENCE, UI, EVENT or GENERIC.

    In case of not knowing how to categorize use the GENERIC label.

    Some examples of you might find:

    INPUT: Entity is not persisted
    OUTPUT: PERSISTENCE

    INPUT: JPA is failing to configure entities
    OUTPUT: PERSISTENCE

    INPUT: The element is not visible in the web
    OUTPUT: UI

    INPUT: The event is sent but never received
    OUTPUT: EVENT

    INPUT: Kafka streaming is failing in some circumstances
    OUTPUT: EVENT

    INPUT: java.lang.NullPointerException in a request
    OUTPUT: GENERIC

    INPUT: {{issueTitle}}
    OUTPUT:
    """) ❷
        IssueClassification categorizeIssue(@V("issueTitle") String issueTitle); ❸
}
```

❶ Defines a system message for any method of the AI Service

❷ Describes the categories, sets some examples, and even sets the programming language used in the project

❸ The issue title is set as a parameter to the prompt

Regarding long prompts, `@UserMessage` and `@SystemMessage` support loading prompts from a file: `@UserMessage(fromResource = "/prompt-template.txt")`. LangChain4j uses the `getResourceAsStream` method from the AI Service class (i.e., `LabelDetector.class`) to load the file.

Finally, configure the `ChatLanguageModel` object using Gemini with a valid API Key. Let's use JSON mode to force the LLM to respond with a valid JSON. JSON mode is supported by Gemini, OpenAI, Azure OpenAI, Mistral, and Ollama.

```
ChatLanguageModel model = GoogleAiGeminiChatModel.builder() ❶
 .apiKey(System.getProperty("API_KEY")) ❷
 .modelName("gemini-1.5-flash")
 .responseFormat(ResponseFormat.builder() ❸
 .type(ResponseFormatType.JSON)
 .jsonSchema(JsonSchemas.jsonSchemaFrom(IssueClassification.class) ❹
 .get())
 .build())
 .build();

LabelDetector labelDetector = AiServices.builder(LabelDetector.class)
 .chatModel(model)
 .build();

IssueClassification label1 = labelDetector
 .categorizeIssue("When storing a user in the database, "
        + "it throws an exception");

System.out.println(label1);

IssueClassification label2 = labelDetector
 .categorizeIssue("JDBC connection exception thrown"); ❺

System.out.println(label2);

IssueClassification label3 = labelDetector
 .categorizeIssue("Math operation fails when divide by 0");

System.out.println(label3);
```

❶ With this change, LangChain4j uses another model

❷ Sets Gemini API key

❸ Configures the output as JSON

❹ Configures the JSON Schema to respond

❺ Persistence label even though no persistence word found

Running the example produces the following output:

```
IssueClassification[category=PERSISTENCE]
IssueClassification[category=PERSISTENCE]
IssueClassification[category=GENERIC]
```

Notice the second one is categorized as a *persistence* issue (correctly) even though the title (*JDBC connection exception thrown*) does not mention persistence. This is because the model knows the issues are about Java, and *JDBC* stands for Java DataBase Connectivity in Java, so it classifies as a *persistence* issue.

There are other ways to classify text, like using embedding vectors or even classifying it when you don't specify the number of possible classifications. You'll see these other approaches in the following chapter.

The last example we'll cover in this section is about image generation.

Image generation and description

LangChain4j supports models that can generate images from a text description, as well as describing in text what the model sees in an image.

In this example, you'll use Gemini to describe what it sees in the image. Gemini is a multimodal model that accepts pictures, videos, audio, or PDF files as input and outputs text.

There is no need to add any new dependencies, as the previous ones are enough to run the example.

Let's ask Gemini to describe the picture in the Figure 5-7 generated using Dall·E 3 model:

Figure 5-7. mage of capybaras lounging on a beach

For this example, you create a `UserMessage` containing two different kinds of elements: text represented by the `dev.langchain4j.data.message.TextContent` class and a picture represented by the `dev.langchain4j.data.message.ImageContent` class. You must set the image as a `String` encoded in **Base64**.

```
ChatLanguageModel gemini = GoogleAiGeminiChatModel.builder()
  .apiKey(System.getProperty("API_KEY"))
  .modelName("gemini-1.5-flash")
  .build();

final String base64Img = readImageInBase64(is);
final ImageContent imageContent = ImageContent.from(base64Img, "image/png"); ❶

final TextContent question = TextContent.from(
```

```
            "What do you see in the image?"); ❷

    final UserMessage userMessage = UserMessage.from(question, imageContent); ❸

    final ChatResponse chatResponse = gemini.chat(userMessage); ❹

    System.out.println(chatResponse.aiMessage().text());
```

❶ Reads the image and creates the image content

❷ Creates a text content with the question about the image

❸ Combines both contents into a single message

❹ Sends the question with the content to the model

The output provided should be something like:

```
The image shows a cartoon scene of capybaras enjoying a day at the beach.
Specifically:

* **Capybaras:** Numerous capybaras are the central focus,
                 lounging on colorful beach towels, ...
* **Beach Setting:** The background depicts a sunny beach
                 with turquoise water. ...
* **Summery Vibe:** The overall style is bright, cheerful,
                 and cartoonish, evoking a summer vacation feel ....
...
```

This example shows the capabilities of using some models to describe an image.

Integration between LangChain4J and models is easy. It only requires adding a dependency and instantiating the correct object for each model, such as `GoogleAiGeminiChatModel` for Google Gemini, `OpenAiChatModel` for OpenAI, or `MistralAiChatModel` for Mistral AI.

However, LangChain4j also integrates with popular Java frameworks like Spring Boot or Quarkus. In the following sections, we'll explore these integrations with more examples.

Spring Boot Integration

LangChain4j provides a Spring Boot starter for integrating LangChain4j into Spring Boot applications. This integration lets developers do different things like:

- Autowiring the `ChatLanguageModel` automatically. There is no need to code the creation of the object; it is configured from the Spring Boot configuration

mechanism (i.e., `application.properties`) and created during the Spring Boot lifecycle.

- Automatic creation of AI Services. There is no need to use the `AiServices` class. Annotate the interface with a special Spring Boot annotation, and the interface is automatically proxied and injected into the context.

Moreover, if the Spring Application context contains certain elements, these element instances are automatically injected into the AI service. This is especially useful when configuring the AI Service with **Memory**, a **tool** to invoke, or **data augmentation**.

These elements are:

- `dev.langchain4j.memory.ChatMemory` and `dev.langchain4j.memory.chat.ChatMemoryProvider` instances for configuring the model with memory.
- `dev.langchain4j.rag.content.retriever.ContentRetriever` and `dev.langchain4j.ragRetrievalAugmentor` instances for configuring the model to use Dat augmentation.
- All methods of any Spring `@Component` or `@Service` class annotated with `@Tool` are registered as a function callable from the model.

When multiple components of the same type exist in the application context, the application will fail to start. To resolve this, utilize the explicit wiring mode.

Let's implement a service that triages comments between positive and negative. This example can be used in multiple situations, such as categorizing customer claims to prioritize them or monitoring social media messages. Of course, while we'll only categorize positive and negative here, you could also classify it into other categories like harassment, hate, etc.

Small language models can also achieve sentiment analysis. It is not necessary to use an LLM, but with an LLM, you can influence/add new categories at any time.

Let's start our example by showing the integration between LangChain4j and Spring Boot.

Spring Boot Dependencies

Since we are going to develop a Spring Boot web project, the first dependency is `org.springframework.boot:spring-boot-starter-web`.

The second one is the LangChain4j starter dependency. There is one starter dependency for each of the supported models. The naming convention is `langchain4j-{integration-name}-spring-boot-starter` where `integration-name` can be any of the models such as `open-ai`, `anthropic`, `ollama`, …

In this example, you'll use the `open-ai` model, hence registering the following dependency: `dev.langchain4j:langchain4j-open-ai-spring-boot-starter`. Moreover, you should add the `dev.langchain4j:langchain4j-spring-boot-starter` dependency to get support for declarative AI services in Spring Boot.

Defining the AI Service

Spring Boot integration lets you define AI Services by annotating the interface with `dev.langchain4j.service.spring.AiService`. This automates the creation of the LangChain4j proxy declaratively, so there is no need to explicitly call the `AiServices.builder()` method.

Create the `TriageService` interface to define the system and user message that will triage the input text between positive, neutral, or negative sentiment.

```
@AiService ❶
public interface TriageService {

    ❷
    @SystemMessage("""
    Analyze the sentiment of the text below.
    Respond only with one word to describe the sentiment.
    """)
    @UserMessage("""
    Your task is to process the review delimited by ---.

    The possible sentiment values are 'POSITIVE' for positive sentiment
    or 'NEGATIVE' for negative sentiment, or 'NEUTRAL'
    if you cannot detect any sentiment


    Some examples:
    - INPUT: This is fantastic news!
    OUTPUT: POSITIVE

    - INPUT: Pi is roughly equal to 3.14
    OUTPUT: NEUTRAL

    - INPUT: I really disliked the pizza.
```

```
        Who would use pineapples as a pizza topping?
        OUTPUT: NEGATIVE

        ---
        {{review}}
        ---
        """)
        Evaluation triage(String review); ❸

    }
```

❶ Uses the specific Spring Boot annotation to mark the interface's AI service

❷ Uses LangChain4j annotations because it is after all a LangChain4j AI service

❸ It is not necessary to annotate with `@V` in SpringBoot integration if you name the variable as the placeholder

The `Evaluation` class contains the sentiment analysis as enum.

```
public enum Evaluation {
 POSITIVE,
 NEGATIVE,
 NEUTRAL
}
```

Open the `application.properties` file to configure the Open AI parameters to the Spring Boot application.

```
langchain4j.open-ai.chat-model.api-key=<your_key>
langchain4j.open-ai.chat-model.model-name=gpt-4o
```

The last thing before running the example is injecting the `TriageService` into a REST controller.

REST Controller

Create a Spring REST controller with two endpoints, one using the injected `dev.langchain4j.model.chat.ChatModel` class, so using the low-level API of LangChain4j, and another endpoint using the `TriageService` AI Service developed previously:

```
@RestController
public class TriageServiceController {

    @Autowired
    TriageService triageService; ❶

    @Autowired
    ChatModel chatLanguageModel; ❷

    @GetMapping("/capital")
```

```
    public String capital() {
        return chatLanguageModel
            .chat("What is the capital of Madagascar?");
    }

    @GetMapping("/triage")
    public Map<String, Evaluation> chat() {

        Map<String, Evaluation> result = new HashMap<>();

        String claim = "I love the service you offer";

        Evaluation triage = triageService.triage(claim);

        System.out.println(claim);
        System.out.println(triage);

        result.put(claim, triage);

        claim = "I couldn't resolve my problem, "
            + "I need to wait for 2 hours to be attended and no solution yet,"
            + "the service is horrible. I hate your bank";

        Evaluation triage2 = triageService.triage(claim);

        System.out.println(claim);
        System.out.println(triage2);

        result.put(claim, triage2);

        return result;
    }
}
```

❶ Injects the AI Service

❷ Injects the low-level API

If you run the example, you'll get the following output in the service terminal:

```
I love the service you offer
POSITIVE
I couldn't resolve my problem, I needed to wait for 2 hours to be attended
 to and no solution yet, the service is horrible. I hate your bank
NEGATIVE
```

It is not hard to validate that the LLM is correctly categorizing the sentiment of the statement. As an exercise, you can develop an example for categorizing texts between toxic, hate, obscene, threat, or insult.

Spring AI (*https://spring.io/projects/spring-ai*) is another framework created by the Spring community for using Spring and LLM models. It uses a different approach specifically designed to work only with the Spring ecosystem.

You'll see some other examples of Spring Boot integration throughout the book, but now, you should be able to develop AI apps with Spring Boot without any problem.

Quarkus Integration

Similar to Spring Boot integration, the Quarkus LangChain4j extension integrates LangChain4j with the Quarkus ecosystem.

It offers a declarative approach to interacting with diverse LLMs like Ollama, Open AI, HuggingFace, etc. It also offers ways to bind LangChain4j elements like `dev.langchain4j.memory.ChatMemory` or `dev.langchain4j.rag.content.retriever.ContentRetriever` into the declared model.

In addition to this smooth integration with LangChain4j, the extension also implements some extra features:

- Integration with different **Document Stores** like Redis.
- **Guardrails**, mechanisms that let you validate the input/output of the LLM to ensure it meets your expectations. Usually, this is used for detecting hallucinations of the model or to validate the user input following some rules.
- **Response Augmenter** to extend the response from the LLM. Usually, you use this to add information on how the model generated the response. For example, in the weather example, you could append the resource who provided the data. To implement a response augmenter, implement the `io.quarkiverse.langchain4j.response.AiResponseAugmenter` interface and make it a Contexts and Dependency Injection (CDI) bean (i.e. using `@ApplicationScoped` annotation); finally, annotate the AI method to augment its response with `io.quarkiverse.langchain4j.response.ResponseAugmenter` indicating the augmenter implementation class name. It also provides tools for **Testing AI-infused** applications based on evaluation and scoring outputs, such as validating that an output is semantically similar to an expected output.
- Integrates to **WebSockets** to make integrating chat-bot-like applications easier.
- Gets some Quarkus features out of the box, like fault tolerance or OpenTelemetry features.

Let's implement a similar sentiment analysis example seen before, but to add some extra, you'll implement it as a chatbot with some new possible sentiments.

Quarkus Dependencies

You are going to develop a Quarkus web application with web sockets.

Quarkus supports static web pages out of the box by copying these resources to the `META-INF/resources` directory.

You'll use websockets to implement the interactions between the UI and the backend, add the following dependency `io.quarkus:quarkus-websockets-next` to provide the classes to develop web sockets applications with Quarkus.

The second one is the LangChain4j dependency. There is one dependency for each of the supported models. The naming convention is `quarkus-langchain4j-{integration-name}` where `integration-name` can be any of the models such as `openai`, `anthropic`, `ollama`, …

In this example, you'll use the `openai` model, registering the following dependency: `io.quarkiverse.langchain4j:quarkus-langchain4j-openai`.

Frontend

The front end is a simple HTML page with some JavaScript to create and communicate with the backend using WebSockets and a minimal HTML form. You can see the frontend of our chatbot in Figure 5-8:

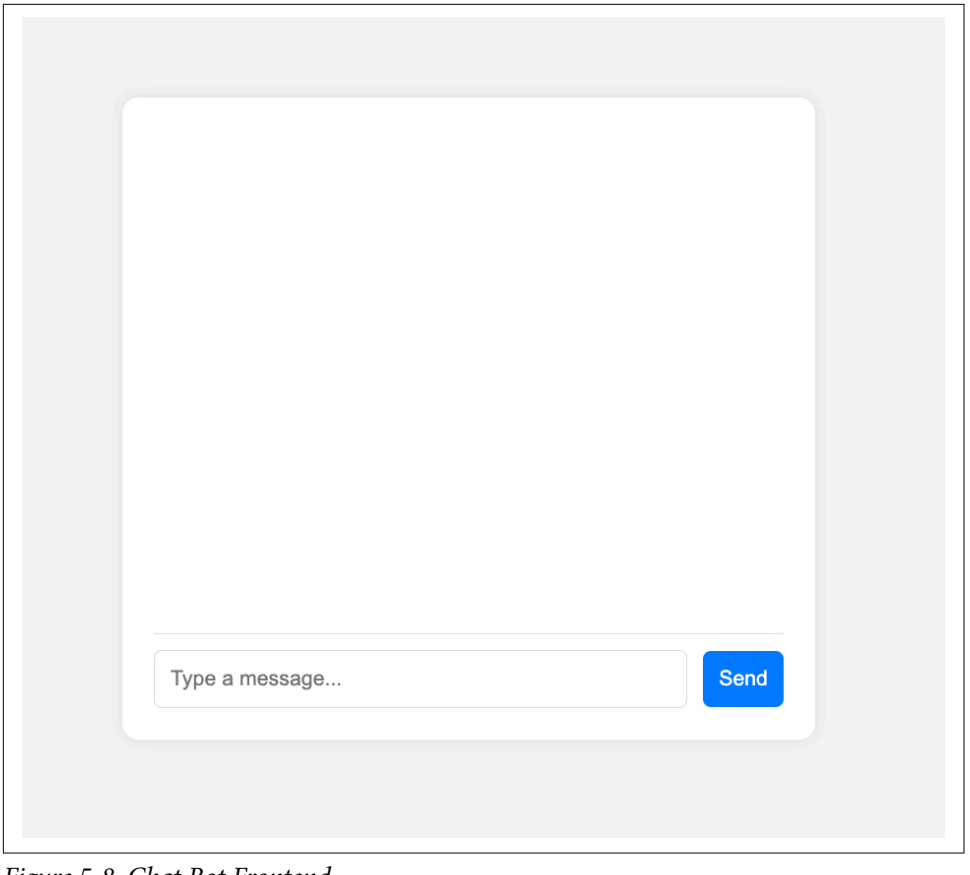


Figure 5-8. Chat Bot Frontend

Create a new file named `index.html` in the `src/main/resources/META-INF/resources` directory. Make any required directory, as Quarkus doesn't usually create the `META-INF` directory.

The web page (skipping the CSS part) should be:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>WebSocket Chatbot</title>
    <style>

    </style>
</head>
<body>
<div class="chat-container"> ❶
```

```
    <div class="chat-box" id="chat-box"></div>
    <div class="input-container">
        <input type="text" id="message" placeholder="Type a message...">
        <div class="spinner" id="spinner"></div>
        <button onclick="sendMessage()">Send</button>
    </div>
</div>

<script>
    const ws = new WebSocket('ws://localhost:8080/chat'); ❷
    const chatBox = document.getElementById('chat-box');
    const messageInput = document.getElementById('message');
    const spinner = document.getElementById('spinner');

    ws.onmessage = function(event) { ❸
        appendMessage('Bot: ' + event.data, 'left');
        spinner.style.display = 'none';
    };

    function sendMessage() { ❹
        const message = messageInput.value.trim();
        if (message === '') return;

        appendMessage('You: ' + message, 'right');
        messageInput.value = '';
        spinner.style.display = 'inline-block';

        ws.send(message);
    }

    function appendMessage(text, alignment) {
        const msgDiv = document.createElement('div');
        msgDiv.textContent = text;
        msgDiv.style.textAlign = alignment;
        chatBox.appendChild(msgDiv);
        chatBox.scrollTop = chatBox.scrollHeight;
    }
</script>
</body>
</html>
```

❶ It is the container containing the chat UI

❷ Connects to localhost using a WebSocket

❸ Shows the received message from the backend (LLM) using the WebSocket connection

❹ Sends the message to the backend using the WebSocket. Moreover, updates the chat UI

With the frontend developed, it is time to dig into the backend and, more specifically, the AI service.

Defining the AI Service

Quarkus integration lets you define AI Services by annotating the interface with the `io.quarkiverse.langchain4j.RegisterAiService`. This automates the creation of the LangChain4j proxy declaratively, so there is no need to explicitly call the `AiServices.builder()` method.

Create the `SentimentAnalysis` interface to define the system and user message that triages the input text between positive, neutral, negative, harassment, or insulting sentiments.

A Java enum defines the possible sentiments to detect:

```
public enum Evaluation {
 POSITIVE, NEGATIVE, NEUTRAL, HARASSMENT, INSULT
}
```

Moreover, the Quarkus extension lets you write prompt expressions in the Qute template engine, making them more dynamic and versatile. For this example, you are using the Qute expression in the prompt to serialize the list of sentiments into plain text, using the `for` operator:

```
    @RegisterAiService ❶
    @SystemMessage("""
    Analyze the sentiment of the text below.
    Respond only with one word to describe the sentiment.
    """)
    ❷
    @UserMessage("""

    Your task is to process the review delimited by ---.

    The possible sentiment values are:
    {#for s in sentiments}
    {s.name()}
    {/for}

    ---
    {review}
    ---
    """)
    Evaluation triage(List<Evaluation> sentiments, ❸
        String review); ❹
}
```

❶ Registers the interface as an AI Service

❷ Sets the prompt with Qute expressions to serialize the sentiments to detect

❸ The list of sentiments name is matched to the Qute expression

❹ The text to categorize

To configure the model, open the `application.properties` file, add the `api-key`, and in this exercise, you'll enable the logging of the requests sent to the Open AI service.

```
❶
quarkus.langchain4j.openai.api-key=demo
quarkus.langchain4j.openai.base-url=http://langchain4j.dev/demo/openai/v1
❷
quarkus.langchain4j.openai.log-requests=true
```

❶ Sets the key to authenticate against the model

❷ Configures LangChain4j to log requests to the console

The final task is developing the WebSocket part.

WebSockets

To implement WebSockets, you'll use the WebSockets Next Quarkus extension. This extension is a new implementation of the WebSocket API that is more efficient and easier to use.

For this example, annotate a class with the `io.quarkus.websockets.next.WebSocket` annotation to specify the exposed endpoint, and annotate a method with `io.quarkus.websockets.next.OnTextMessage` which is invoked automatically every time the frontend sends a new message through the opened WebSocket.

In addition to this, use the already known `jakarta.inject.Inject` annotation to inject the AI Service (`SentimentAnalysis`):

```
@WebSocket(path = "/chat") ❶
public class WebSocketChatBot {

    @Inject ❷
    SentimentAnalysis sentimentAnalysis;

    @OnTextMessage ❸
    public String onMessage(String message) {
        Evaluation evaluation = sentimentAnalysis.triage(
            List.of(Evaluation.values()), ❹
 message);
        return evaluation.name(); ❺
```

```
    }
}
```

❶ Configures the WebSocket endpoint to `/chat`

❷ Injects the AI Service

❸ This method is called for every message

❹ Creates a list with all possible sentiments

❺ Returns the result

Start the application and send some comments using the chatbot. For example, "*this is the worst service I have ever seen*" results as "*NEGATIVE*," while "you are a stupid making this comment_" is categorized as "*INSULT*."

If you open the application at `localhost:8080`, you'll see the Figure 5-9.

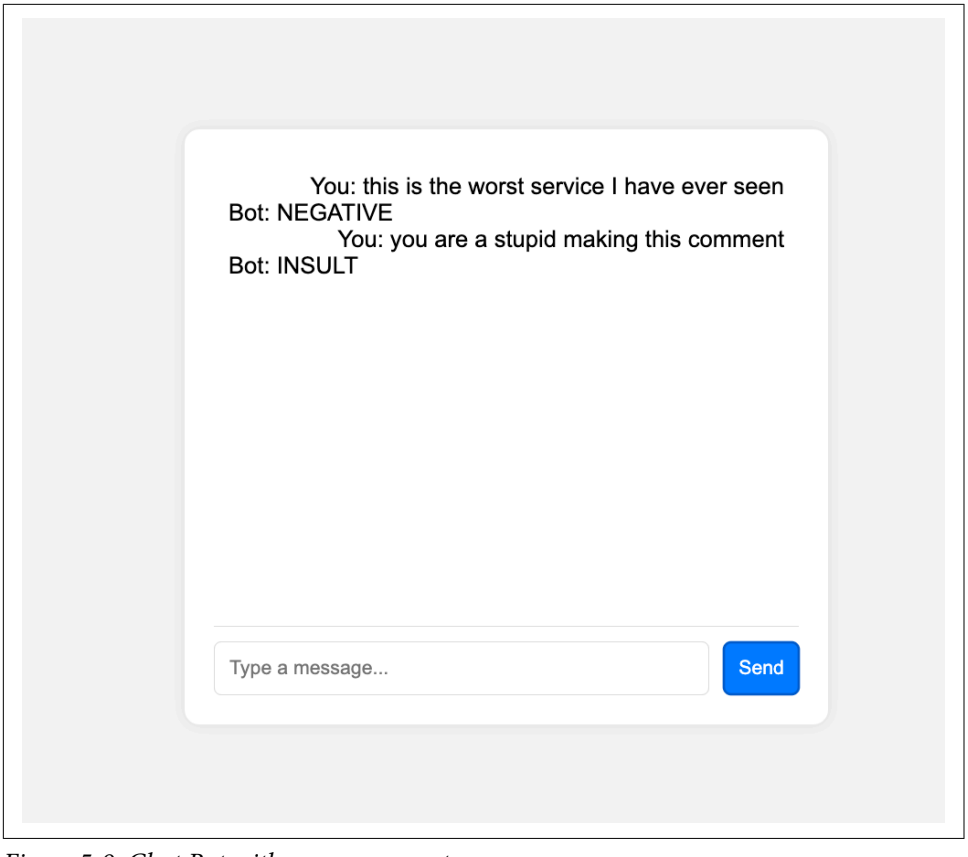


Figure 5-9. Chat Bot with some comments

Inspect the server console in the terminal to verify what the application sends to the model. The body content is interesting, as you can see the content of the system and user messages and the serialization of the list of sentiments into a text.

```
...
- body: {
 "model" : "gpt-4o-mini",
 "messages" : [ {
 "role" : "system",
 "content" : "Analyze the sentiment of the text below...
 }, {
 "role" : "user",
 "content" : "Your task is to process the review delimited by --
 The possible sentiment values are:\n
 POSITIVE\n
 NEGATIVE\n
 NEUTRAL\n
 HARRASMENT\n
 INSULT\n\n
```

```
---\n
this is the worst service I have ever seen\n
---
...
} ],
"temperature" : 1.0,
"top_p" : 1.0,
"presence_penalty" : 0.0,
"frequency_penalty" : 0.0
}
```

The approach to integrating LangChain4j with Quarkus is very similar to Spring Boot. You use different annotations but with the same result.

These integrations enhance the AI services adding extra capabilities, but everything supported in native LangChain4j AI services, is supported in the integrations too.

Let's look at another example of transforming an image into text using LLMs as optical character recognition (OCRs).

Optical Character Recognition (OCR)

OCR is the process of converting different types of documents—such as PDFs, or images, into digital text.

Let's create another example in Quarkus. In this example, you'll develop a service to extract the snippet code from a screenshot using the Open AI model again.

In Figure 5-10, you can see the flow of the example:

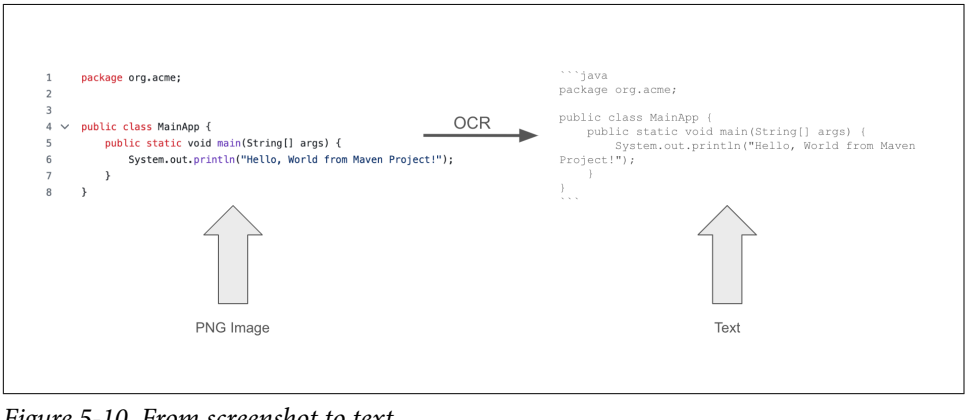


Figure 5-10. From screenshot to text

The example is similar to the one where we described what was in the image (in that casesome capybaras), but now, we'll use the OCR process to extract some source code from an image. In the left part, you see a source code captured as a screenshot, while

in the right part, shows the result after applying the OCR process, where the code is as text (formatted within a Markdown block).

You'll use the low-level API of LangChain4j to implement this example.

```
@Path("/extract")
public class ScreenShotResource {

    @Inject ❶
    ChatModel chatLanguageModel;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String extract() {

        UserMessage userMessage = UserMessage.from(
            TextContent.from( ❷
                "This is image was reported on a GitHub issue." +
                "If this is a snippet of Java code, please respond " +
                "with only the Java code. " +
                "If the lines are numbered, removes them from the output." +
                "If it is not Java code, respond with 'NOT AN IMAGE'"),
            ImageContent.from(
                URI.create("https://i.postimg.cc/fL6x1MK9/screenshot.png") ❸
        )
        );

        ChatResponse response = chatLanguageModel.chat(userMessage);
        return response.aiMessage().text(); ❹

    }
}
```

❶ Injects the LangChain4j low-level API object

❷ Creates a text object describing what to do with the image

❸ Open AI model doesn't support sending images encoded in Base64

❹ Returns the snippet in Markdown format

Calling the endpoint results in the following text:

````
```java
package org.acme;

public class MainApp {
 public static void main(String[] args) {
 System.out.println("Hello, World from Maven Project!");
 }
````
````

```
}
```

You can rewrite the previous example using the high-level API too. Quarkus integration provides the `io.quarkiverse.langchain4j.ImageUrl` annotation to automatically create an `ImageContent` from a URL.

The same example could be rewritten to:

```
@RegisterAiService
public interface CodeExtractor {

    @UserMessage("""
            This is image was reported on a GitHub issue.
            If this is a snippet of Java code, please respond
            with only the Java code.
            If the lines are numbered, removes them from the output.
            If it is not Java code, respond with 'NOT AN IMAGE'
            """)
    String extract(@ImageUrl URI image); ❶

}
```

❶ The URI paramter is automatically translated to an `ImageContent` object

The `demo` key might not work with image processing. Use your API key in `application.properties` or set it using the `QUARKUS_LANGCHAIN4J_OPENAI_API_KEY` environment variable. Quarkus supports a `.env` file with all environment variables set there, which will be loaded at boot time.

Now, you've got an idea of LangChain4j and a basic knowledge of its integration with two of the most popular Java frameworks.

For the rest of the chapter, you'll explore how to integrate memory and tools with the frameworks.

Tools

Spring Boot and Quarkus integrations provide support for the *Tools* feature. Both are based on the `@Tool` annotation, like in the LangChain4j project, but they differ in how you register the tools.

While Spring Boot integration scans the classpath searching for classes annotated with `@Component` or `@Service` and having methods annotated with `@Tool`, Quarkus requires you explicitly register the class either using the `@RegisterAiService` annotation setting the `tools` attribute or using `io.quarkiverse.langchain4j.ToolBox`

annotation to register a tool for a specific method instead of globally for all methods of the AI service.

Let's develop an example using tools. We'll use this application multiple times throughout the book to build a chatbot step-by-step. We'll show you some challenges you might find while developing a chatbot and how to solve them. Of course, we'll also develop other examples in the book, but we'll return to this example multiple times.

The application you'll develop is a chatbot for a theme park. A customer of the park will ask questions about the theme park like:

- What is the best ride according to the user rating?
- What is the waiting time for the X ride?
- Which rides can I go if my height is X?
- Can you provide me with information about X's ride?

Although you'll develop some REST endpoints for the application, you'll also provide a chatbot UI interface to make testing it easier.

The application will use the OpenAI ChatGPT model, Quarkus with LangChain4j as the Java framework, two storages, PostgreSQL (for storing ride information) and Redis (for storing ride waiting times, among other things you'll see later), and more elements that we'll explore during the following chapters.

About Quarkus Dev Services

Quarkus Dev Services is a feature of Quarkus designed to simplify application development by automatically provisioning and configuring the services required by your application during the **development** phase. This feature is useful when the application depends on external services like databases, message brokers, and so on.

Moreover, Dev Services minimize the need for configuration because it automatically detects the services and configures the application properly.

Quarkus Dev Services leverages containers to spin up services in an isolated way. To run them, you need a container engine like Docker (Desktop) or Podman (Desktop).

The lifecycle of Quarkus Dev Service works, in summary, as follows:

1. Start the application in *dev mode* by running `./mvnw quarkus:dev`

2. Quarkus verifies if there is any dependency that registers a Dev Service. For example, if you add a PostgreSQL JDBC driver, Quarkus detects an external dependency on a PostgreSQL database.
3. Then, check if a container engine is running, and if so, spin up the PostgreSQL database container in the engine.
4. Finally, configure the application to connect to this container instance.

Notice that this works with any other database (MySQL, MariaDB, MongoDB, etc.), messaging systems (Kafka, Apache Artemis, etc.), or distributed caches (Redis, etc.).

Quarkus Dev Services is a gem for developers as they can focus on coding and not deploying external dependencies.

Now that you know what the application looks like, let's start development.

Dependencies

Create a Quarkus project with the following dependencies:

`io.quarkus:quarkus-rest-jackson`
: To create a REST endpoint.

`io.quarkus:quarkus-hibernate-orm-panache` *and* `io.quarkus:quarkus-jdbc-postgresql`
: For the persistence part with PostgreSQL.

`io.quarkus:quarkus-redis-client`
: To interact with Redis instance.

`io.quarkiverse.langchain4j:quarkus-langchain4j-openai`
: LangChain4j with OpenAI integration.

Thanks to Quarkus Dev Services, when running the application in dev mode, Quarkus will start a Redis and a PostgreSQL container.

Rides Persistence

The next step is creating the persistence part to store the rating of all rides.

Create a JPA entity named `Ride`, annotated with `jakarta.persistence.Entity` with three fields: an id, a name, and the rating:

```
@Entity ❶
public class Ride {

 @jakarta.persistence.Id
 @jakarta.persistence.GeneratedValue
```

```
    private Long id; ❷

    public String name;
    public double rating;
}
```

❶ Sets the class as entity

❷ Auto-increment id

Then create a repository class to manage the persistence operations. This class implements the `io.quarkus.hibernate.orm.panache.PanacheRepository` interface, inheriting some static fields to execute basic persistence operations such as insert, delete, and find, … You'll also implement a query that selects the first ride with the maximum rating, and project the result to a Data Transfer Object to avoid exposing the internal fields like `id` to the model.

```
public record RideRecord(String name, double rating) {
}

@ApplicationScoped
public class RideRepository implements PanacheRepository<Ride> {

    @Inject
    org.jboss.logging.Logger logger; ❶

    @Tool("get the best ride") ❷
    @Transactional
    public RideRecord getTheBestRideByRatings() {

        logger.info("Get The Best Ride query");

        return findAll(Sort.descending("rating")) ❸
            .project(RideRecord.class) ❹
            .firstResult();
    }

}
```

❶ Injects a logger

❷ Annotates the query as a tool for the model

❸ Finds all rides ordered by rating

❹ Projects the result extracting only the `name` and the `rating` fields

It is a good practice to log tools to validate when the model invokes them. For the sake of simplicity, we keep the query simple, calling the `findAll` method, which is inherited from the interface.

Waiting Times service

Next we'll need to implement is the application code that stores the waiting times for each ride. In this case, you use Redis, which stores the ride name as a key and the waiting time as a value. The application calculates the waiting times randomly at boot-up time.

Quarkus uses, in this case, `io.quarkus.redis.datasource.RedisDataSource` and `io.quarkus.redis.datasource.value.ValueCommands` classes to communicate with Redis to insert and consult waiting times:

```
@ApplicationScoped
public class WaitingTime {

    @Inject
    DurationGenerator durationGenerator; ❶

    @Inject
    Logger logger;

    private final ValueCommands<String, Long> timeCommands; ❷

    public WaitingTime(RedisDataSource ds) { ❸
        this.timeCommands = ds.value(Long.class);
    }

    public void setRandomWaitingTime(String attraction) {
        this.setWaitingTime(attraction,
            this.durationGenerator.randomDurantion()); ❹
    }

    public void setWaitingTime(String attraction, long waitingTime) {
        this.timeCommands.set(attraction, waitingTime);
    }

    @Tool("get the waiting time for the given ride name") ❺
    public long getWaitingTime(String attraction) { ❻

        logger.infof("Gets waiting time for %s", attraction);

        return this.timeCommands.get(attraction); ❼
    }

}
```

❶ A class to generate random numbers

❷ Instance to set and get values in a key/value fashion

❸ Connection to Redis

❹ Sets the waiting time for a ride

❺ Annotates the method as a tool for the model

❻ The model fills the attraction value

❼ Returns the waiting time

Before running the example, the last piece implement is the AI Service to send the user chat request to the model.

AI Service

This AI Service is similar to those we created earlier in the chapter, but you use the `io.quarkiverse.langchain4j.ToolBox` annotation to register tool classes.

```
@RegisterAiService
public interface ThemeParkChatBot {

    @SystemMessage("""
    You are an assistant for answering questions about the theme park.

    These questions can only be related to theme park.
    Examples of these questions can be:

    - Can you describe a given ride?
    - What is the minimum height to enter to a ride?
    - What rides can I access with my height?
    - What is the best ride at the moment?
    - What is the waiting time for a given ride?

    If questions are not about theme park or you don't know the answer,
    you should always return "I don't know".
    Don't give information that is wrong
    """)
    @UserMessage("""
    The theme park user has the following question: {question}

    The answer must be max 2 lines.
    """)
    @ToolBox({RideRepository.class, WaitingTime.class}) ❶
    String chat(String question);

}
```

❶ Registers both tools only for this method

And the configuration file? That's the beauty of Quarkus Dev Services. You only need to add the `quarkus.langchain4j.openai.api-key` property with the OpenAI API key; The Quarkus Dev Services automatically configures the rest.

To test this code, let's write a REST endpoint to interact with the model and populate some data with the data stores.

REST Endpoint

The REST endpoint executes the ride population in the PostgreSQL database and the waiting times in Redis at startup. Then, it implements two methods with some predefined queries, so you can validate that the output is correct:

```
@Path("/ride")
public class RideResource {

    @Inject
    RideRepository rideRepository;

    @Inject
    WaitingTime waitingTime;

    @io.quarkus.runtime.Startup ❶
    @jakarta.transaction.Transactional ❷
    public void populateData() {
        insertRides();
    }

    private void insertRides() {
        Ride r1 = new Ride();
        r1.name = "Oncharted. My Penitence";
        r1.rating = 5.0;

        rideRepository.persist(r1);

        waitingTime.setRandomWaitingTime(r1.name);


        Ride r2 = new Ride();
        r2.name = "Dragon Fun";
        r2.rating = 4.9;

        rideRepository.persist(r2);

        waitingTime.setRandomWaitingTime(r2.name);
}

    @Inject
    ThemeParkChatBot themeParkChatBot;
```

```
    @GET
    @Path("/chat/best")
    public String askForTheBest() { ❸
        return this.themeParkChatBot
            .chat("What is the best ride at the moment?");
    }

    @GET
    @Path("/chat/waiting")
    public String askForWaitingTime() { ❹
        return this.themeParkChatBot
            .chat("What is the waiting time for Dragon Fun ride?");
    }
}
```

❶ Executes this code when the application is up but not ready to receive requests yet

❷ Makes the method transactional

❸ Gets the best ride

❹ Gets the waiting time for a specific ride

To execute the application in dev mode, go to the terminal and run the following command from the root directory of the application:

```
./mvnw quarkus:dev
```

After some seconds, Quarkus starts two containers (PostgreSQL and Redis), populates some data to the stores, enables dev mode, and opens the REST connections. The application is up and running and prepared to receive requests. In another terminal, run some requests, use `curl` or any other HTTP client, and check the result:

```
curl localhost:8080/ride/chat/best
The best ride at the moment is "Oncharted. My Penitence," with a rating of 5.0.%

curl localhost:8080/ride/chat/waiting
The waiting time for the Dragon Fun ride is 64 minutes
```

With a small portion of code, you have an impressive application that can respond to some questions about the theme park. Of course, there is much more to do, which we'll dig into in later chapters, but it is a good start.

But suppose the following conversation:

```
Me: What is the best ride at the moment?
Bot: The best ride at the moment is "Oncharted. My Penitence," with a rating of 5.0
Me: What is the waiting time for that?
```

It seems to be a reasonable conversation flow, but what would be the answer to the latest question? You may have guessed correctly: *I don't know* because remember the models are stateless, so it has no clue of what *that* is referring to in the current context.

You'll fix this problem later in this chapter.

Logging

It is important to understand what is going on under the covers, how the application and the model interact, what transformations are performed, and so on, especially when you are starting to develop AI applications.

For this reason, we recommend you enable logging in `application.properties`:

```
❶
quarkus.hibernate-orm.log.sql=true

❷
quarkus.langchain4j.log-requests=true
quarkus.langchain4j.log-responses=true
```

❶ Logs the executed SQL statements

❷ Logs the requests and responses between the app and the model

We'll show you in the last chapter of the book how to implement observability to monitor the application correctly in the production phase.

You are well versed in *Tooling* now, and hopefully you understand how to implement and use it and the advantages it offers. Let's walk through the *Tooling* low-level API, as this understanding will help you use *Tooling* for more advanced use cases where you need some dynamism.

Dynamic Tooling

If you have enabled the logging in the previous example, you've seen all the JSON objects exchanged between the application and the OpenAI model.

One of the message contains the tools the model can invoke if it is necessary to complete the task. These tools are registered in the `tools` section of the document, where you define each method that can be called inside a `function` object. Then, it is sent to the model.

A function informs the model what it does and what input arguments it expects; it sends the method's signature. The function object contains the following fields: The

`name` represents the method name to call, and the `description` offers the details on when to invoke the function. The model will try to get this information from the method name if not provided. Finally, in the `parameters` section, a subdocument following the JSON schema defines the function's **input** parameters.

Let's take a look at the registration of the two tools used in the theme park example for the **OpenAI API**:

```
...
"tools" : [ { ❶

    // RideRecord getTheBestRideByRatings(); function
    "type" : "function", ❷
    "function" : {
      "name" : "getTheBestRideByRatings", ❸
      "description" : "get the best ride", ❹
      "parameters" : { ❺
        "type" : "object",
        "properties" : { },
        "required" : [ ]
}
}
}, {

    // long getWaitingTime(String attraction); function
    "type" : "function",
    "function" : {
      "name" : "getWaitingTime",
      "description" : "get the waiting time for the given ride name",
      "parameters" : {
        "type" : "object",
        "properties" : {
          "attraction" : { ❻
            "type" : "string"
}
},
        "required" : [ "attraction" ] ❼
}
}
} ]
...
```

❶ Tools section

❷ Defines the input as function

❸ Sets the name of the function

❹ Description of the function for defining when to call it

❺ Input parameters of the function, in this case no parameters

❻ Defines the first parameter

❼ Makes it mandatory

LangChain4j transforms the methods annotated with `@Tool` to this JSON subdocument transparently to the developer.

However, as a developer, LangChain4j lets you define this JSON part programmatically, providing the `dev.langchain4j.agent.tool.ToolSpecification` class to create the JSON part and some JSON utilities like `dev.langchain4j.model.chat.request.json.JsonObjectSchema` to generate the `parameters` section. Let's see how to define the tool's specification for the `getTheBestRideByRatings` method:

```
ToolSpecification toolSpecification = ToolSpecification.builder() ❶
 .name("getTheBestRideByRatings")
 .description("get the best ride")
 .parameters(
                JsonObjectSchema.builder()
 .properties(Map.of()) ❷
 .build()
 )
.build();
```

❶ This object is a direct translation from the API to create the `function` subdocument

❷ Since the method has no parameters, set empty properties

This is the first step for defining tools; the second step is implementing the `dev.langchain4j.service.tool.ToolExecutor` interface. This implementation is responsible for executing the function and acts as a bridge between LangChain4j and the call of the tool method.

This interface receives the function parameters to call the function, the memory ID (more on this in the following section —for now you can just ignore this), and returns the result of the call as String. As with the `ToolSpecification`, LangChain4j provides helper methods to deal with JSON documents, like `dev.ai4j.openai4j.Json.fromJson` and `dev.ai4j.openai4j.Json.toJson` static methods to convert from/to JSON to Java objects and vice versa.

For the `getTheBestRideByRatings` example, the implementation could be as follows:

```
ToolExecutor toolExecutor = (toolExecutionRequest, memoryId) -> {
    Map<String, Object> arguments =
```

```
            fromJson(toolExecutionRequest.arguments(), Map.class); ❶

        // String ride = arguments.get("attraction").toString(); ❷

        RideRecord rr = rideRepository.getTheBestRideByRatings(); ❸
        return toJson(rr); ❹
    };
```

❶ Model sends the function arguments as JSON; you convert them to a `Map` where the key is the name of the parameter and the value of its value. In this case, the `Map` is empty.

❷ In the case of the `getWaitingTime` function call, you'd get the attraction parameter getting it from the `Map`

❸ Makes the real call to the function

❹ Returns the result as a String JSON

The third and final step is to register to the chat model, the tuple `ToolSpecification` and `ToolExecutor`, so LangChain4j knows when and what to execute in each case:

```
ThemeParkChatBot assistant = AiServices.builder(ThemeParkChatBot.class)
 .chatModel(chatLanguageModel)
 .tools(Map.of(toolSpecification, toolExecutor)) ❶
 .build();
```

❶ Registers when to call the function (key part of the Map) and what to call (value part of the the Map)

Obviously, this method is a bit complicated and cumbersome, but it opens the door to dynamically selecting which tool to execute for each invocation based on, for example, the user message.

You can set up a `dev.langchain4j.service.tool.ToolProvider` that is automatically triggered whenever the AI service is called. It will supply the tools to be included in the current request to the LLM.

We'll want to add the `getTheBestRideByRatings` tool only when the user's message contains the word *weather*. To register the `ToolProvider`, use the `toolProvider` method instead of the `tools` method because for dynamic tools you need to use atools provider and register it:

```
ToolProvider toolProvider = (toolProviderRequest) -> {
    if (toolProviderRequest.userMessage()
 .singleText().contains("weather")) { ❶
        return ToolProviderResult.builder()
 .add(toolSpecification, toolExecutor)
```

```
        .build(); ❷
    } else {
            return null; ❸
    }
};

ThemeParkChatBot assistant = AiServices.builder(ThemeParkChatBot.class)
    .chatModel(model)
    .toolProvider(toolProvider)
    .build();
```

❶ Validates if the user message contains *weather* string

❷ If contains it creates the tuple for the model

❸ If not, then returns no tool

Note that, the condition can be anything. For example, in the case of a query to an external system, you could validate that the system is reachable and, if not, provide an alternative.

You can also do something similar with the Quarkus integration. To support dynamic selection in Quarkus, implement two interfaces and annotate them as `@ApplicationScoped`.

The first interface is the `ToolProvider` interface, the same as the interface as in plain LangChain4j, but annotated with `@ApplicationScoped`:

```
@ApplicationScoped
public class WeatherToolProvider implements ToolProvider {

    @Override
    public ToolProviderResult provideTools(ToolProviderRequest request) {
        ...
        return ToolProviderResult.builder()....
    }

}
```

The second interface is `java.util.function.Supplier<ToolProvider>`, which only returns the instance of the `ToolProvider`:

```
@ApplicationScoped
public class WeatherToolProviderSupplier implements Supplier<ToolProvider> {
    @Inject
    WeatherToolProvider weatherProvider;

    @Override
    public ToolProvider get() {
        return weatherProvider;
```

```
    }
}
```

The final step is registering the `WeatherToolProviderSupplier` using the `toolProviderSupplier` property of the `RegisterAiService` annotation:

```
@RegisterAiService(toolProviderSupplier = WeatherToolProviderSupplier.class)
public interface ThemeParkChatBot {
}
```

Dynamic tooling is not necessary in most simple cases, but it is important to consider it when things become more complex.

Final notes about tooling

We've dug pretty deeply into *Tools* in this section, we have a few final notes to add here.

LLMs use the function parameter name to decide which value the user message sends to the function. But if you need to describe the parameter, you can use the `dev.langchain4j.agent.tool.P` annotation.

```
@Tool("get the waiting time for the given ride name")
public long getWaitingTime(@P("The attraction or ride name")
                           String attraction)
```

Moreover, if the function parameter is not a simple type like `int`, `double`, `String`, … but a class, you can use the `@Description` annotation to describe the class and the fields as you did in the structured output example.

Last but not least, we recommend using a class that aggregates all tooling calls or a subset of calls categorized by usage. Populating your code with `@Tool` annotations might cause you to lose control of what is called from the model, especially when you have multiple AI services. For example:

```
@ApplicationScoped
public class RidesTool {

    @Inject ❶
    RideRepository rideRepository;

    @Inject
    WaitingTime waitingTime;

    @Tool("get the best ride")
    public RideRecord getTheBestRideByRatings() {
        return rideRepository.getTheBestRideByRatings(); ❷
    }


    @Tool("get the waiting time for the given ride name")
```

```
    public long getWaitingTime(String rideName) {
        return waitingTime.getWaitingTime(rideName);
    }

}
```

❶ Injects the classes containing the logic

❷ Redirects the call to the logic

This class acts as a *Facade* or *Adapter* of tooling calls. Obviously, the methods of this class could log calls, provide security, or adapt the model's input/output parameters.

You are almost at the end of this chapter, but before wrapping up, let's look at how to fix the problem of our chatbot's lack of memory, which makes user interaction unnatural.

Memory

Memory is an important feature when implementing applications that are not stateless. This is especially true for chatbots, where you might need to keep the "history" of the messages to offer a more real-world experience.

Remember when we exposed the following conversation to our theme park chatbot:

```
Me: What is the best ride at the moment?
Bot: The best ride at the moment is "Oncharted. My Penitence," with a rating of 5.0
Me: What is the waiting time for that?
```

We said that the bot's answer would be *I don't know* because it does not understand the context of what you were asking about. As you learned at the beginning of the chapter, the solution to this problem is using the *Memory* feature, but Quarkus works slightly differently as it offers this feature out-of-the-box.

Quarkus extension automatically registers any instance of `dev.langchain4j.store.memory.chat.ChatMemoryStore` bean in the AI Service in the same way as Spring Boot integration does, but Quarkus LangChain4j extension registers an in-memory chat memory store by default if the application doesn't provide its own. The in-mem storage is a `java.util.Map` for storing messages and is transient, meaning data does not persist across application restarts.

In summary, with the Quarkus extension, you get all your interactions with memory by default if you don't configure it differently. To avoid this automatic behavior, so no memory storage is used, configure the AI Service as follows: `@RegisterAiService(chatMemoryProviderSupplier = RegisterAiService.NoChatMemoryProviderSupplier.class)`

The Figure 5-11 summarizes the flow:

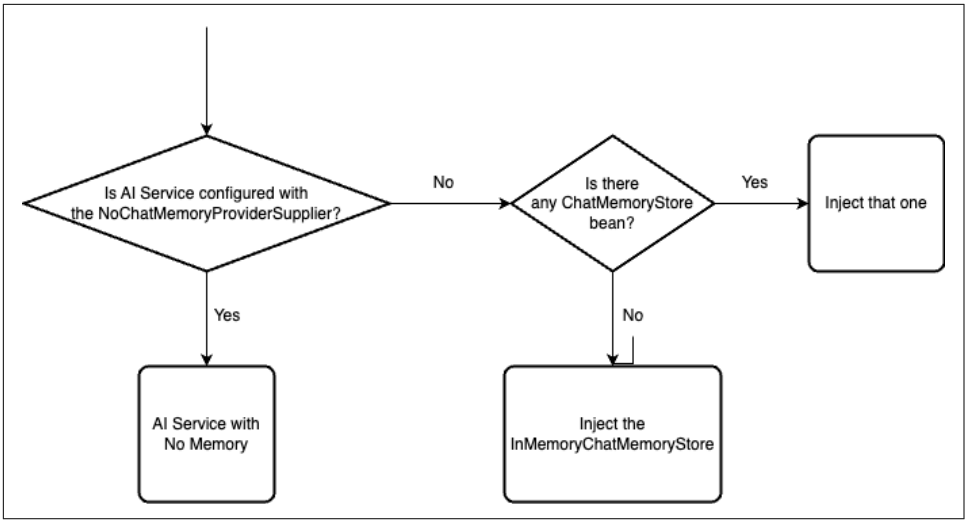


Figure 5-11. Configuring the Context Memory

Quarkus checks if there is the `RegisterAiService.NoChatMemoryProviderSupplier.class` bean, if so, then no memory is used. If it is not found, then if you provides any memory bean then it uses it, finally, otherwise creates a default one using an in-memory bean implementation.

The logical question you might wonder is: *If memory is always enabled, what happens with physical memory consumption?*

And the answer to this question is: *Quarkus has mechanisms to control it*.

The first mechanism is an instance of `dev.langchain4j.memory.chat.MessageWindowChatMemory`, which by default has a window size of ten. The application only stores 10 messages, and when you reach this number, it deletes the older ones. You can change this value by setting the `quarkus.langchain4j.chat-memory.memory-window.max-messages` property in the `application.properties` file to any other value.

Second, memory is wiped out when the AI Service leaves the CDI scope. If not specified, the AI Service scope is `RequestScoped`.

About CDI Scopes

CDI scopes define the application's lifecycle and visibility of beans (managed objects).

There are multiple defined scopes, though you can also implement a custom scope. Table 5-1 summarizes the CDI scopes available in Quarkus.

Table 5-1. CDI Scopes

Scope	Lifecycle	Use Case
`@RequestScoped`	Single HTTP request	Request-specific data
`@SessionScoped`	HTTP session	User-specific data (e.g., shopping cart)
`@ApplicationScoped`	Entire application lifecycle	Shared resources
`@Dependent`	Tied to the injecting object	Tightly coupled beans
@Singleton	Like `@ApplicationScoped` except that no client proxy is used.	Stateless services or utilities

So, depending on the annotation, the bean's lifecycle (creation and destruction) will be longer or shorter.

If you don't annotate an AI service interface with any scope annotation, it will be `@RequestScoped` by default, which means that every request wipes out that user's memory. Let's look at two examples to understand how default behavior works:

```
@GET
@Path()
public String askForWaitingTime() {
    return this.themeParkChatBot.chat("What is the waiting time for Dragon Fun ride?");
}

@GET
@Path()
public String askForSpecificWaitingTime() {
    return this.themeParkChatBot.chat("What is the waiting time for that?");
}
```

Since they are two different HTTP requests, the Quarkus extension cleans the memory after each method, so even though the same user made the requests, the response for the second call would be *I don't know*.

But what will happen if you execute this:

```
@GET
@Path()
public String askForBoth() {
    this.themeParkChatBot.chat("What is the waiting time for Dragon Fun ride?");
    return this.themeParkChatBot.chat("What is the waiting time for that?");
}
```

Then, it would return the correct answer *Waiting time for Dragon Fun is 20 minutes*. This is because the interaction with the AI service happened within the same HTTP request, so the memory is kept, and the model receives the context of the previous conversations.

Let's improve the theme park example. You'll reuse the chatbot frontend used in the first Quarkus-LangChain4j integration example with WebSockets and, of course, add some memory to the conversation. To make it more real, you'll use a Redis instance as a memory store, which makes this part more scalable, durable, and monitorable.

Dependencies

Add the `io.quarkus:quarkus-websockets-next` dependency to the project to support the web sockets communication. Add the `io.quarkiverse.langchain4j:quarkus-langchain4j-memory-store-redis` dependency to register the Redis memory store. This dependency adds the `io.quarkiverse.langchain4j.memorystore.RedisChatMemoryStore` class to the context, so the AI Service uses it as memory without further configuration.

Moreover, copy the `index.html` file from the previous project to `src/main/resources/META-INF/resources`.

Changes to code

The next step is to annotate the `ThemeParkChatBot` AI Service with `SessionScope` so that memory is kept during all conversations and not just wiped out after each request. Moreover, to keep multiple users using the chatbot, you should also change the `chat` method to add the `@MemoryId` field.

```
@RegisterAiService
@SessionScoped ❶
public interface ThemeParkChatBot {
    String chat(@MemoryId int userId, String question);
}
```

❶ Attaches the AI Service lifecycle to `SessionScoped`

But don't copy the changes yet because while these would be required if you were using LangChain4j standalone, the Quarkus extension simplifies the creation of chatbots when used together with the `quarkus-websockets-next` extension.

And you can simplify the previous AI service definition to:

```
@RegisterAiService
@SessionScoped ❶
public interface ThemeParkChatBot {
    String chat(String question); ❷
}
```

❶ The AI service is tied to the scope of the WebSocket endpoint. Quarkus cleans the chat memory when the WebSocket connection is closed

❷ There is no `@MemoryId` field used in the AI service

If there is no `@MemoryId` annotation does this mean that the chatbot is not multiuser? No, it is multiuser, but Quarkus will automatically use the WebSocket *connection ID* as the memory ID, freeing the developer to deal with the annotation. This ensures that each WebSocket session has its chat memory.

To execute the application in dev mode, go to the terminal and run the following command from the root directory of the application:

```
./mvnw quarkus:dev
```

Quarkus dev mode starts all dependencies (PostgreSQL and Redis) again. After a few seconds, open the browser again at localhost:8080 and interact with the chatbot as follows:

```
Me: What is the best ride at the moment?
Bot: The best ride at the moment is "Oncharted. My Penitence," with a rating of 5.0
Me: What is the waiting time for that?
Bot: The waiting time for "Oncharted. My Penitence" is 24 minutes.
```

Figure 5-12 shows the frontend with the interaction.

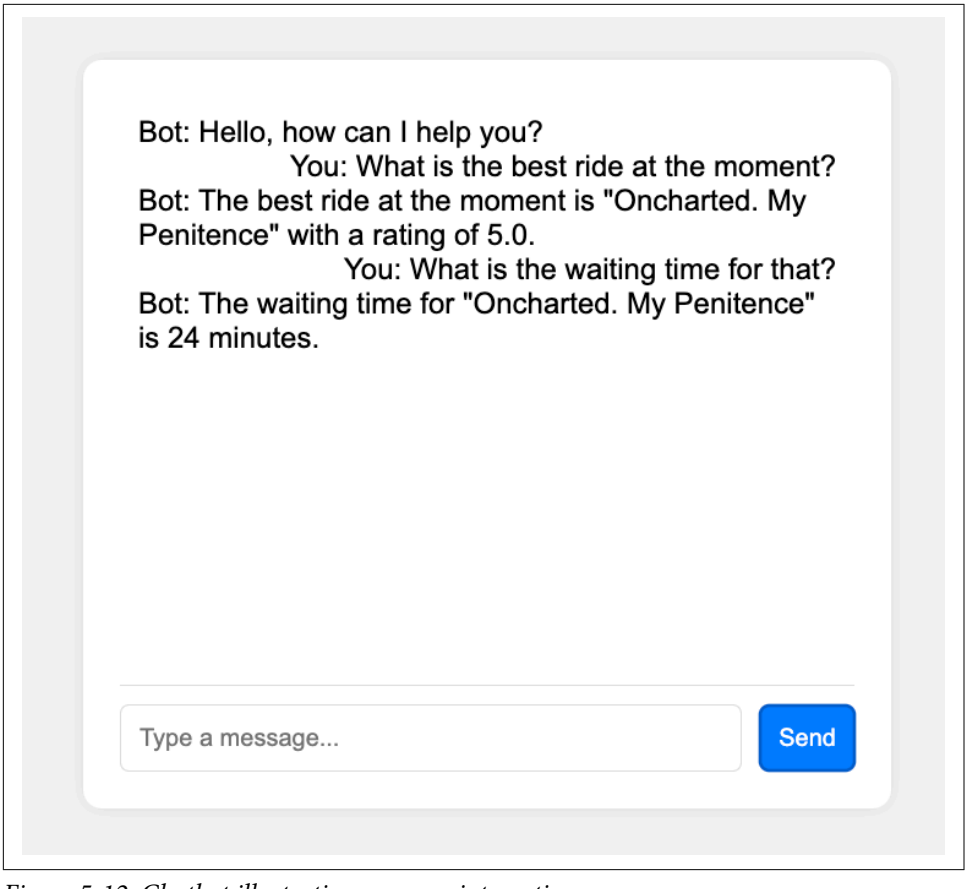


Figure 5-12. Chatbot illustrating memory integration

Moreover, you can visualize the Redis content using any Redis visualizer like Redis Insight.

Next steps

In this chapter, you learned about the LangChain4j project and how it helps develop Java applications that interact with LLMs.

As you've seen in this chapter, you can use LangChain4j for image processing, to implement a chatbot, or to categorize any kind of text, either analyzing the sentiment of the text (positive, neutral, negative) or, for example, categorizing it to detect harassment, insults, hate speech, etc.

Moreover, we discussed two important topics, *memory* and *tools*, which are two of the three key points (the third is RAG which we'll show you in the following chapter) to why to use LangChain4j is important to simplify the development of AI applications.

By now, you should have a good understanding of LangChain4j, when and how to use it, and its integration with the most popular enterprise Java frameworks, Quarkus and Spring Boot.

However, there are pieces of LangChain4j that we've got not covered yet, such as:

- Embedding Vectors.
- Retrieval-Augmented Generation (RAG).
- Semantic examples to use when LLM is not an option.

In the following chapter, we'll take what you know about LangChain4j and build on that we'll focus a lot on RAG, as it is the third key aspect of LangChain4j you should know to develop enterprise AI applications.

Let's take your LangChain4j knowledge to the next level.

CHAPTER 6
Image Processing

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

The previous chapters covered the basics of integrating Java with AI/ML projects. You also learned how to load and infer models in Java using DJL and consume them with LangChain4J and LangGraph4j. For the remaining part of this book, we will build upon this knowledge to implement more advanced use cases closer to what you might encounter in a real project.

One common use of AI in projects involves image processing for classification or information extraction tasks. The input image can be a single photo provided by a user or a stream of images from a device like a camera.

Here are some examples of image processing use cases:

- Detecting objects or people, such as in a security surveillance system.
- Classifying images by content, such as categorizing products.
- Extracting information from documents, like ID cards or passports.

- Reading vehicle license plates, for example, in speed cameras.

One common aspect on all these use cases is the need to prepare the image before it is processed by the model. This can involve tasks such as resizing the image to meet the model's input size requirements, squaring the image for central cropping, or applying other advanced algorithms like Gaussian filtering or the Canny algorithm to aid the model in image detection, classification, or processing.

This chapter does not discuss image processing algorithms but instead provides a basic understanding of when and how they can be applied in Java. After completing this chapter, you'll be able to effectively use these image-processing algorithms for some use cases provided by data engineers or vision experts.

Bur prior to get into image processing let's understand **what an image is**.

First, it's important to understand how an image is stored in memory to comprehend how image processing operates.

An image is made up of pixels, with each pixel representing a point in the image. The total number of pixels depends on the image's dimensions (width and height).

Each pixel contains information about that point, including color, opacity, and other attributes based on the image format. For instance:

- In a **grayscale** image, a pixel is an integer between 0 and 255, where 0 represents black and 255 represents white.
- In an **RGB** (Red, Green, Blue) image, a pixel is represented by a group of three integers for the red, green, and blue color components. For example, the values 255, 0, and 255 produce pink.
- In an **RGBA** (Red, Green, Blue, Alpha) image, a pixel is represented by four integers, including RGB and opacity.

In a scenario of a 4x4 image (16 pixels in total) in RGB format, the image comprises a 3-dimensional matrix (one for each color) of integers ranging from 0 to 255.

Figure 6-1 illustrates this decomposition:

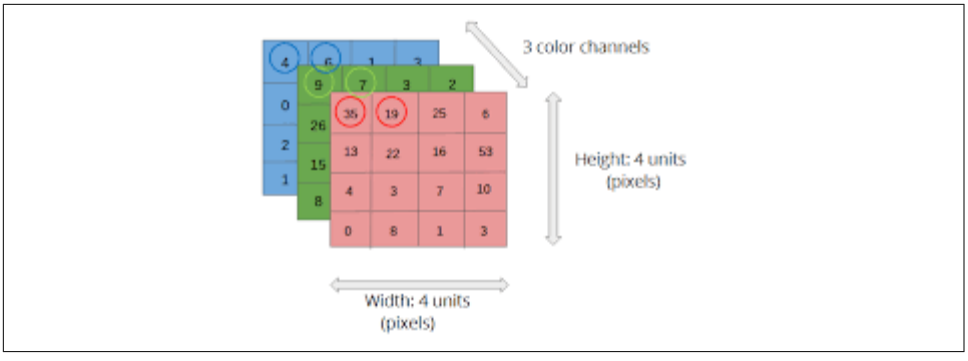


Figure 6-1. Image decomposition

Image processing is applying changes to the matrix at the pixel level, for example, changing a value close to zero to a strict zero.

Let's explore how to do image processing in Java.

OpenCV

Open Source Computer Vision Library *https://opencv.org/* (OpenCV) is a C++ library written under Apache License 2 for programming real-time computer vision algorithms and image manipulation. The library implements over 2500 algorithms.

OpenCV supports GPU acceleration, making it perfect for large or real-time image processing images.

The main operations supported by OpenCV for image processing are:

Image Acquisition
: Obtain images by loading them from a disk or capturing them from a camera. This operation can include resizing, color conversion, cropping, and other adjustments.

Image Enhancement
: Modify image levels, such as brightness and contrast, to improve visual quality.

Image Restoration
: Correct defects that degrade an image, such as noise or motion blur.

Color Image Processing
: Adjust colors through color balancing, color correction, or auto-white balance.

Morphological Processing
: Analyze shapes within images to extract useful information using algorithms like dilation and erosion.

Segmentation
: Divide an image into multiple regions for detailed scene analysis.

Even though OpenCV is written in \C, there is a Java binding project named OpenPnP OpenCV(*https://github.com/openpnp/opencv*) that uses the Java Native Interface (JNI) to load and use OpenCV natively in Java. The classes and method names used in the Java binding are similar (if not the same) as those in the OpenCV \C project, facilitating the adoption of the Java library.

To get started with OpenPnP OpenCV Java, from this point we'll refer to it simply as OpenCV, register the following dependency on your build tool:

```
<dependency>
    <groupId>org.openpnp</groupId>
    <artifactId>opencv</artifactId>
    <version>4.9.0-0</version>
</dependency>
```

You can start using OpenCV for Java, as the JAR file bundles the OpenCV native library for most platforms and architectures.

Initializing the Library

With the library present at the classpath, load the native library into memory. This is done in two ways in OpenCV:

Manual Installation

The first involves manually installing the OpenCV native library on the system and then calling `System.loadLibrary(org.opencv.core.Core.NATIVE_LIBRARY_NAME)`, usually in a `static` block.

Bundled Installation

The second one is called the `nu.pattern.OpenCV.loadLocally()` method.

This call will attempt to load the library exactly once per class loader. Initially, it will try to load from the local installation, which is equivalent to the previous method.

If this attempt fails, the loader will copy the binary from its dependency *JAR* file to a temporary directory and add that directory to `java.library.path`.

The library will remove these temporary files during a clean shutdown.

In the book, we advocate for the bundled installation to get started as no extra steps are required, like installing a library to your system by calling `OpenCV.loadLocally()`.

With the library loaded, you can start using OpenCV classes.

The library has multiple classes as a point of entry; the most important ones are `org.opencv.imgproc.Imgproc` and `org.opencv.imgcodecs.Imgcodecs` because they contain the main methods and constants to do image processing.

Let's explore the basic operations for loading and saving images.

Load and Save Images

To load an image, OpenCV offers the `org.opencv.imgcodecs.Imgcodecs.imread` method.

```
protected org.opencv.core.Mat loadImage(Path image) { ❶
    return Imgcodecs.imread( ❷
        image.toAbsolutePath().toString()
 );
}
```

❶ `imread` returns the image as a matrix representation

❷ The image location is a `String`

And the equivalent method for saving an image:

```
protected void saveImage(Mat mat, Path path) { ❶
    Imgcodecs.imwrite(
        path.toAbsolutePath().toString(), ❷
 mat);
}
```

❶ Materializes the given matrix to the path

❷ The destination location is a `String`

This API is useful when transforming an image file to an image matrix or materializing a matrix to an image file. However, in some cases, the source or destination of the photo is not a matrix but a `byte[]`. For these cases, OpenCV has the `org.opencv.core.MatOfByte` class.

The following method shows how to transform a `byte[]/java.io.InputStream` to an `org.opencv.core.Mat`:

```
private Mat fromStream(InputStream is) throws IOException {
    final byte[] bytes = toByteArray(is); ❶

    return Imgcodecs.imdecode( ❷
        new MatOfByte(bytes),  ❸
        Imgcodecs.IMREAD_UNCHANGED
```

```
    );
}
```

❶ Reads theInputStream

❷ Use the `imdecode` method to decode from bytes to an image matrix

❸ Creates a matrix from the `byte[]`

Similarly, you can transform an image matrix to a `byte[]`:

```
private InputStream toStream(Mat mat) {

    MatOfByte output = new MatOfByte();
    Imgcodecs.imencode(".jpg", mat, output); ❶

    return new ByteArrayInputStream(
        output.toArray() ❷
);
}
```

❶ Encode image matrix into a `MatOfByte` object

❷ Gets the content as `byte[]`

Now that you know how to load and save images in OpenCV, let's explore actual image processing with basic transformations.

Basic Transformations

It's not uncommon for AI models that use images as input parameters to require some image processing as a precondition for analyzing the image. This process can affect the image size, requiring you to crop or resize images, or the number of color layers, requiring you to remove the alpha channel or transform to a greyscale.

To Greyscale

To convert an image to any color space, use the `Imgproc.cvtColor` method. The typical conversion is to greyscale, running the following method:

```
private Mat toGreyScale(Mat original) { ❶

    Mat greyscale = new Mat(); ❷
    Imgproc.cvtColor(original, greyscale, Imgproc.COLOR_RGB2GRAY); ❸

    return greyscale;
}
```

❶ Original *RGB* photo

❷ Creates the object to store the conversion

❸ Converts to greyscale

Other possible conversions can be: `COLOR_BGR2HLS` to convert from RGB to HLS (Hue, Lightness, Saturation), `COLOR_RGBA2GRAY` to convert from RGBA to greyscale, or `COLOR_GRAY2RGB` to convert grayscale to RGB, to mention some of them. All constants starting from `COLOR_` in the `Ìmgproc` class refer to color conversions.

Resize

To resize an image, use the `Imgproc.resize` command, setting the new size of the image (or the resize ratio) and the interpolation method.

```
private Mat resize(Mat original, double ratio) {

    Mat resized = new Mat(); ❶
    Imgproc.resize(original, resized,
                   new Size(), ❷
 ratio, ❸
 ratio, ❹
                   Imgproc.INTER_LINEAR ❺
 );

    return resized;
}
```

❶ Creates the object to store the resized image

❷ Output image size, if not set, uses the ratio

❸ Scale factor along the horizontal axis

❹ Scale factor along the vertical axis

❺ Interpolation method

Other possible interpolation methods include `INTER_CUBIC` for cubic interpolation, or `INTER_LANCZOS4` for Lanczos interpolation method.

Sometimes, it is not possible to resize an image without deforming it. For example, the model requires a 1:1 aspect ratio, while the input image is in the 16:9 ratio. Resizing is an option, but at the cost of deforming the image. Another option is to crop the image, focusing on the important part of the image. There are many different algorithms for finding the important part using vision algorithms, but in most cases, a center crop of the image with the required aspect ratio works correctly.

Crop

Let's crop the center of an image into a square, using the `org.opencv.core.Rect` to define the valid rectangle of an image.

To implement this crop, you need to play a bit with math to calculate the exact coordinates to find the starting cropping point, as the crop size is already set. Let's take an overview of the steps required to calculate the starting point:

1. Calculate the center of the image
2. Determine the starting point for cropping
3. Ensure the cropped image is within image boundaries
4. Crop the image with the crop size defined from the starting point.

Figure 6-2 shows each of these points in a photo of 1008 x 756px with a crop size of 400px:

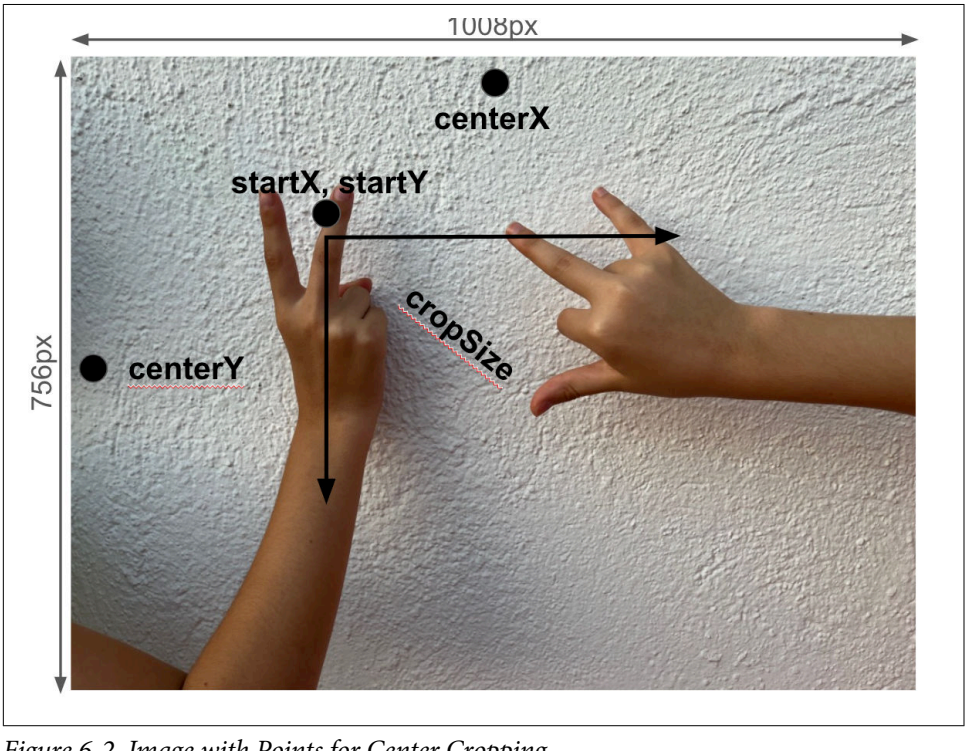


Figure 6-2. Image with Points for Center Cropping

The following snippet shows the implementation of the center cropping algorithm using OpenCV:

```
private Mat centerCrop(Mat original, int cropSize) {

    ❶
    int centerX = original.cols() / 2;
    int centerY = original.rows() / 2;

    ❷
    int startX = centerX - (cropSize / 2);
    int startY = centerY - (cropSize / 2);

    ❸
 startX = Math.max(0, startX);
 startY = Math.max(0, startY);

    int cropWidth = Math.min(cropSize, original.cols() - startX);
    int cropHeight = Math.min(cropSize, original.rows() - startY);

    Rect r = new Rect(startX, startY, cropWidth, cropHeight); ❹

    return new Mat(original, r); ❺
}
```

❶ Calculate the center of the image

❷ Calculate the top-left corner of the crop area

❸ Ensure the crop area is within the image boundaries

❹ Generates a rectangle with the valid section of the image

❺ Generates a new image matrix with only the part limited by the rectangle

The image processed with the cropping algorithm results in the following output:

Figure 6-3. Center Cropped Image

At this point you're familiar with basic image manipulation algorithms. In the next section, you'll see how to overlay elements in an image, such as another image, rectangles, or text.

Overlying

When implementing AI/ML models involving an image, the model usually either returns a string representing the categorization of the image (e.g., boots, sandals, shoes, slippers) or a list of coordinates of what the model is detecting within the image (e.g., cat, dogs, human, etc.).

In this latter use case, drawing rectangles with labels in the image is useful to show the viewer what and where the model has detected the points of interest.

Figure 6-4 shows an example of an output image with an overlay showing a detected hand:

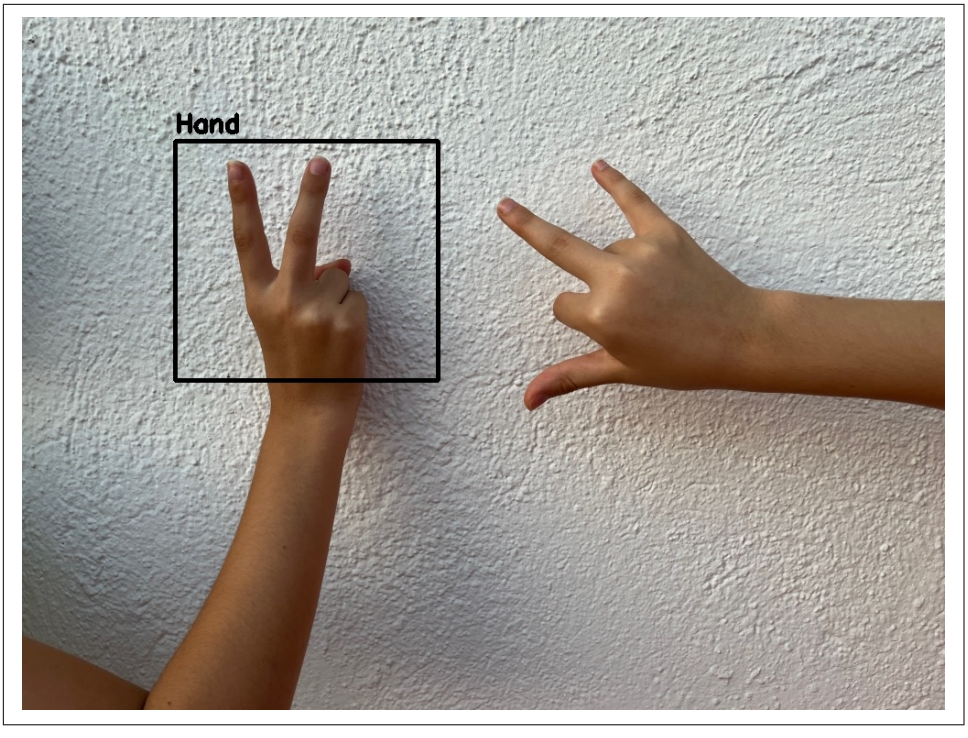


Figure 6-4. Image with the detected element

Let's explore how to overlying elements in an image using Open CV:

Draw Boundaries

OpenCV provides two methods for drawing rectangles or overlaying texts on an image: `Imgproc.rectangle` and `Imgproc.putText`.

Let's implement a method to draw the boundaries of an object. The algorithm checks the length of the text to adapt the width of the given rectangle in case the text is more significant than the rectangle:

```
protected Mat drawRectangleWithText(Mat original, Rect rectangle,
        Scalar color, String text) {

    final double fontScale = 0.9d; ❶
    final int fontThickness = 3;
    final int rectangleThickness = 3;
    final int font =  Imgproc.FONT_HERSHEY_SIMPLEX;

    Mat destination = original.clone(); ❷

    final Size textSize = Imgproc.getTextSize(text, font, fontScale,
```

```
fontThickness, null); ❸

        if (textSize.width > rectangle.width) { ❹
            rectangle.width = (int) textSize.width;
}

        Imgproc.rectangle(destination, rectangle, color,
                rectangleThickness); ❺
        Imgproc.putText(destination, text,
                            new Point(rectangle.x, rectangle.y - 10), ❻
font, fontScale, color, fontThickness); ❼

        return destination;
}
```

❶ Defines default values for font scale, font thickness, font and rectangle thickness

❷ Copies the original image to not modify it

❸ Gets the size of the text when materialized in the image

❹ Checks if the label width is bigger than the rectangle width

❺ Draws the rectangle

❻ Moves the text coordinates 10 pixels above the rectangle not to overlap

❼ Embeds the text into the given point

Another option is not drawing only the rectangle's border (or any other shape) but filling it with some color, optionally with some transparency.

In the following example, you'll create a rectangle filled with green color and a transparent layer so the main image is partially visible. This is done using the `org.opencv.core.Core.addWeighted` method.

```
protected Mat fillRectangle(Mat src, Rect rect, Scalar color, double alpha) {

    final Mat overlay = src.clone(); ❶
    Imgproc.rectangle(overlay, rect, color, ❷
                      -1); ❸

    Mat output = new Mat(); ❹
    Core.addWeighted(overlay, alpha, src, 1 - alpha, 0, output); ❺

    return output;
}
```

❶ Creates a copy of the original image

❷ Creates a rectangle

❸ Fills the rectangle with the color

❹ Output matrix

❺ Blends the images with transparency

Figure 6-5 shows the result:

Figure 6-5. Image with rectangle

In addition to drawing lines, rectangles, polygons, or circles, you can also overlay a transparent image onto another image. For example, this might be useful to hide any detected element, such as the face of a minor or sensitive data.

Overlap Images

Let's implement a method that overlays a foreground image onto a background image at a specified position.

It handles transparency by blending the pixel values of the foreground and background images based on the foreground image's alpha channel (opacity).

The steps followed by the algorithm are:

1. Convert the background and foreground images to the *RGBA* color space so they can handle the transparency.
2. Copies the pixel values from the foreground to the background image only if the location of the foreground pixel is not outside the background boundaries.
3. For each channel, the foreground and the background pixel values are blended based on the opacity value.
4. The composed image is returned as an image matrix.

```
private Mat overlayImage(Mat backgroun, Mat foregroun,
                         Point location) throws IOException {

    Mat bg = new Mat();
    Mat fg = new Mat();

    Imgproc.cvtColor(backgroun, bg, Imgproc.COLOR_RGB2RGBA); ❶
    Imgproc.cvtColor(foregroun, fg, Imgproc.COLOR_RGB2RGBA);

    for (int y = (int) Math.max(location.y , 0); y < bg.rows(); ++y) { ❷

        int fY = (int) (y - location.y);

        if(fY >= fg.rows()) ❸
            break;

        for (int x = (int) Math.max(location.x, 0); x < bg.cols(); ++x) {
            int fX = (int) (x - location.x);
            if(fX >= fg.cols()){
                    break;
}

            double opacity;
            double[] finalPixelValue = new double[4];

opacity = fg.get(fY , fX)[2]; ❹

            ❺
finalPixelValue[0] = bg.get(y, x)[0];
finalPixelValue[1] = bg.get(y, x)[1];
finalPixelValue[2] = bg.get(y, x)[2];
finalPixelValue[3] = bg.get(y, x)[3];

            for(int c = 0;  c < bg.channels(); ++c){
                if(opacity > 0){
                    double foregroundPx =  fg.get(fY, fX)[c];
                    double backgroundPx =  bg.get(y, x)[c];

                    float fOpacity = (float) (opacity / 255);
```

```
finalPixelValue[c] = ((backgroundPx * ( 1.0 - fOpacity))
                                        + (foregroundPx * fOpacity)); ❻
                    if(c==3){
finalPixelValue[c] = fg.get(fY,fX)[3];
}
}
}
            bg.put(y, x, finalPixelValue); ❼
}
}

    return bg;
}
```

❶ Convert background and foreground images to RGBA

❷ Iterate through each pixel of the background image starting from the specified location

❸ Pixel is not out of bounds

❹ Get the alpha value (opacity) of the current foreground pixel

❺ Get the initial pixel values from the background

❻ Blend the foreground and background pixel values based on the opacity

❼ Update the background image with the blended pixel values

Next, let's use image processing algorithms like binarization, gaussian blur, or canny to change the image.

Image Processing

Let's explore some algorithms that change the image content, for example, to remove/blur background objects (making them less obvious), reduce noise to increase accuracy in image analysis, or correct image perspective to provide a more calibrated view for processing.

Gaussian Blur

Gaussian Blur is the process of blurring an image using a Gaussian function. Gaussian Blur is used in image processing for multiple purposes:

Noise reduction
: Smooth the variations in pixel values; it helps remove small-scale noise.

Scale-Space Representation
: Generate multiple blurred versions of the image. It is used in multi-scale analysis and feature detection at various scales.

Preprocessing for Edge Detection
: Blur helps obtain cleaner and more accurate edge maps when used to detect edges of objects.

Reducing Aliasing
: Blur helps prevent aliasing.

To apply Gaussian blur with OpenCV, use the `Imgproc.GaussianBlur` method.

```
Imgproc.GaussianBlur(mat, blurredMat, ❶
                         new Size(7, 7), ❷
                         1 ❸
 );
```

❶ Input and Output matrix

❷ Kernel size for blurring

❸ Gaussian kernel standard deviation in X direction (sigmaX)

Applying the Gaussian blur algorithm on the hands image results in the Figure 6-6:

Figure 6-6. Gaussian Blur

After blurring, let's see how to apply a binarization process to an image.

Binarization

Binarization is the process of iterating through all pixels and setting a value of 0 (black) or 1 (white) if the pixel value is smaller than a defined threshold.

It is useful to segment an image into foreground and background regions by separating relevant elements from the background.

There are a lot of different binarization algorithms like `THRESH_BINARY`, `ADAPTIVE_THRESH_MEAN_C` selecting the threshold for a pixel based on a small region around it, or `THRESH_OTSU` for Otsu's Binarization algorithm where the algorithm finds the optimal threshold automatically.

The next code shows applying binarization process alone and with Oysu:

```
Imgproc.threshold(src, binary, ❶
                  100, ❷
                  255, ❸
                  Imgproc.THRESH_BINARY); ❹
```

```
Mat grey = toGreyScale(src); ❺
Imgproc.threshold(src, binary,
                  0, ❻
                  255,
                  Imgproc.THRESH_BINARY + Imgproc.THRESH_OTSU ❼
 );
```

❶ Input and Output matrix

❷ Threshold value

❸ Maximum value to use

❹ Thresholding type

❺ Otsu algorithm requires image in greyscale

❻ Values are ignored as the Otsu algorithm automatically calculates the point

❼ Otsu's threshold algorithm

Applying the previous binarization algorithms on the hands image results in the Figure 6-7.

Image 1
: The original image with no transformation.

Image 2
: The image after applying binary threshold.

Image 3
: The image after applying Otsu threshold.

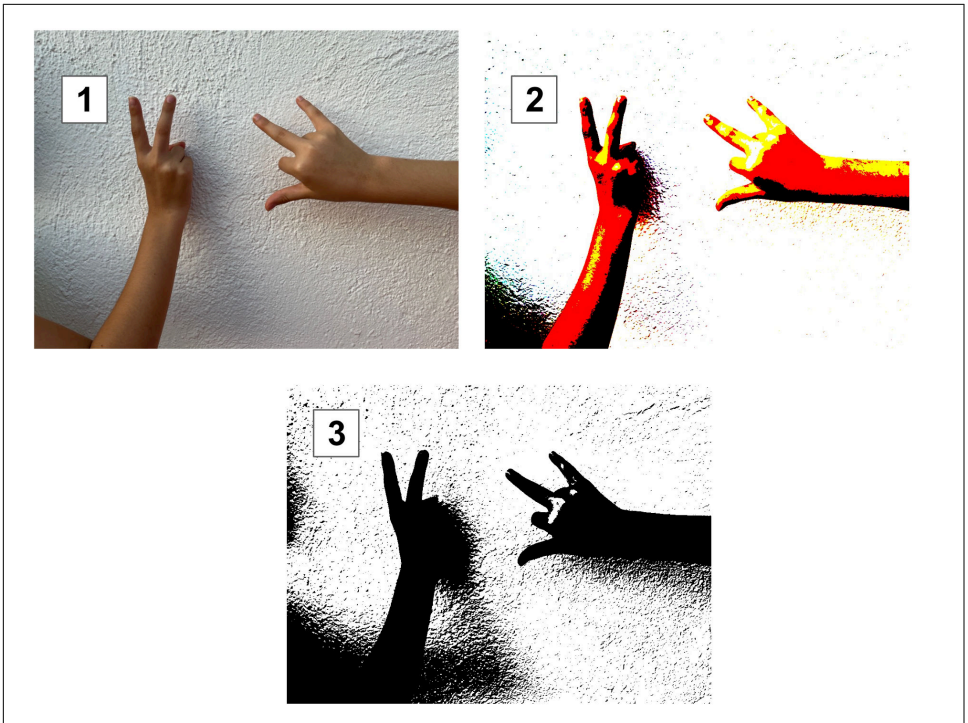


Figure 6-7. Binarization Image

Another important algorithm used in low light situations is noise reduction to improve the quality of the image.

Next part you'll learn how to apply noise reduction to an image.

Noise Reduction

Besides blurring an image to reduce noise, OpenCV implements image-denoising algorithms for greyscale and color images. The class implementing denoising algorithms is `org.opencv.photo.Photo`, which also implements other algorithms for photo manipulation, such as texture flattening, illumination change, detail enhancement, or pencil sketch.

Let's take a look at that in action:

```
Photo.fastNlMeansDenoising(src, dst,
                           10); //  ❶
```

❶ Filter strength. Big values perfectly remove noise but also remove image details, while smaller values preserve details but also preserve some noise.

So far, you executed these algorithms as a single unit: you apply the algorithm and the image changes. The last algorithms we'll show you in this section are the combination of multiple image processing algorithms to change the image.

Edge Detection

Edge detection is a crucial image processing technique for identifying and locating the boundaries or edges of objects within an image.

Some of the use cases are correcting the perspective of an image for future processing, detecting different areas present in an image (segmentation), extracting the foreground of an image, or identifying a concrete part of an image.

There are multiple algorithm combinations to implement edge detection in an image, and we'll show you the most common one:

```
protected Mat edgeProcessing(Mat mat) {
    final Mat processed = new Mat();

    Imgproc.GaussianBlur(mat, processed, new Size(7, 7), 1); ❶

    Imgproc.cvtColor(processed, processed, Imgproc.COLOR_RGB2GRAY); ❷

    Imgproc.Canny(processed, processed, 200, 25); ❸

    Imgproc.dilate(processed, processed, ❹
                new Mat(), ❺
                new Point(-1, -1), ❻
                1); ❼

    return processed;
}
```

❶ Blurs using a Gaussian filter

❷ Transforms image to greyscale

❸ Finds edges using *Canny* edge detection algorithm.

❹ Dilates the image to add pixels to the boundary of the input image, making the object more visible and filling small gaps in the image

❺ Structuring element used for dilation; matrix of 3x3 is used when empty

❻ Position of the anchor, in this case, the center of the image

❼ Number of times dilation is applied.

Applying the previous algorithm results in Figure 6-8.

Image 1
: It is the original image with no transformation.

Image 2
: The image after applying gaussian filter.

Image 3
: The image after applying Canny.

Image 4
: The Image after dilate the image.

Figure 6-8. Transformation Image

The interesting step here is the dilate step, which thickens the borders to make them easy to detect or process.

Besides processing the image, OpenCV has the `Imgproc.findContours` method to detect all the image contours and store them as points into a `List`:

```
List<MatOfPoint> allContours = new ArrayList<>();
    Imgproc.findContours(edges, allContours, ❶ ❷
                         new org.opencv.core.Mat(), ❸
```

```
                                Imgproc.RETR_TREE, ❹
                                Imgproc.CHAIN_APPROX_NONE);❺
```

❶ Input matrix

❷ List where contours are stored

❸ Optional output vector containing information about the image topology

❹ Contour retrieval mode, in this case, retrieves all of the contours and reconstructs a full hierarchy of nested contours

❺ Contour approximation method, in this case, all points are used

The previous method returns a list of all detected contours. Each `MatOfPoint` object contains a list of all the points that define a contour. A representation of one of these objects might be `[{433.0, 257.0}, {268.0, 1655.0}, {1271.0, 1823.0}, {1372.0, 274.0}]`, joining all the points with a line would be a contour of an element detected in the image.

The problem is what's happen if the algorithm detects more than one contour, how to distinguish the contour of the required object from contours of other objects.

One way to solve this problem is filtering the results by setting some certain steps:

1. Get only the `MatOfPoint` objects that cover the most significant area, calling the `Imgproc.contourArea` method, and remove the rest.
2. Approximate the resulting polygon with another polygon with fewer vertices using the `Imgproc.approxPolyDP` method. It uses the Douglas-Peucker algorithm. With this change, the shape is smoothed, closer to human-eye reality.
3. Remove all polygons with less than 4 corners.
4. Limit the result to a certain number of elements. Depending on the domain might be 1, or more.

These steps are executed in the following code:

```
final List<MatOfPoint> matOfPoints = allContours.stream() ❶
 .sorted((o1, o2) -> ❷
 (int) (Imgproc.contourArea(o2, false) -
                        Imgproc.contourArea(o1, false)))
 .map(cnt -> {
            MatOfPoint2f points2d = new MatOfPoint2f(cnt.toArray());
            final double peri = Imgproc.arcLength(points2d, true);

            MatOfPoint2f approx = new MatOfPoint2f(); ❸
            Imgproc.approxPolyDP(points2d, approx, 0.02 * peri, true); ❹
```

```
            return approx;
        })
        .filter(approx -> approx.total() >= 4) ❺
        .map(mt2f -> {
            MatOfPoint approxf1 = new MatOfPoint();
            mt2f.convertTo(approxf1, CvType.CV_32S); ❻
            return approxf1;
        })
        .limit(1) / ❼
        .toList();
```

❶ Iterates over all detected contours

❷ Sort areas in decrease order

❸ `approxPolyDP` method returns points in the `float` type

❹ Approximates the polygon

❺ Filters the detected contours to have at least 4 corners

❻ Transforms the points from `float` to `int`

❼ Limits to one result

At this point, there is only one element in the list containing the list of points conforming to the detected element, for this example, the card present in the photo.

To draw the contours to the original image use the `Imgproc.drawContours` method:

```
Mat copyOfOriginalImage = originalImage.clone(); ❶
Imgproc.drawContours(copyOfOriginalImage, matOfPoints, ❷
                     -1, ❸
 GREEN, ❹
                     5); ❺
```

❶ Copy the original image to keep it original with no modifications

❷ Draws the contours detected in the previous step in the given image

❸ Draw all detected contours

❹ `Scalar` representing green

❺ Thickness of the lines

The final image is shown in the Figure 6-9:

Figure 6-9. Card with contour

These steps are important not only for detecting elements in a photograph but also for correcting the perspective of an image. Next, let's see using Open CV to correct the image prespective.

Perspective Correction

In some use cases, such as photographed documents, the document within the image might be distorted, making it difficult to extract the information enclosed in it.

For this reason, when working with photographed documents like passports, card IDs, card licenses, etc., where we might have control over how the image is taken, it is important to include a perspective correction as a preprocessing step.

If you look closely at the previous image, the card borders are not parallel with the image borders, so let's see how to fix the image's perspective.

OpenCV has the `Imgproc.warpPerspective` to apply a perspective correction to a photo. This perspective transformation is applied by having a reference point or element to correct. This transformation can fix perspective for some aspects of the photo while distorting others; what is essential here is detecting which element needs a correction.

The steps to follow are:

1. Edge detecting the element within the image
2. Find the contour of the image
3. Map the contour points to the desired locations (for example, using L2 form)
4. Compute the perspective transformation matrix calling the `Imgproc.getPerspectiveTransform` method.
5. Apply the perspective transformation matrix to the input image calling the `Imgproc.warpPerspective` method.

Figure 6-10 shows the detection of the four corner points and the translation to correct the perspective:

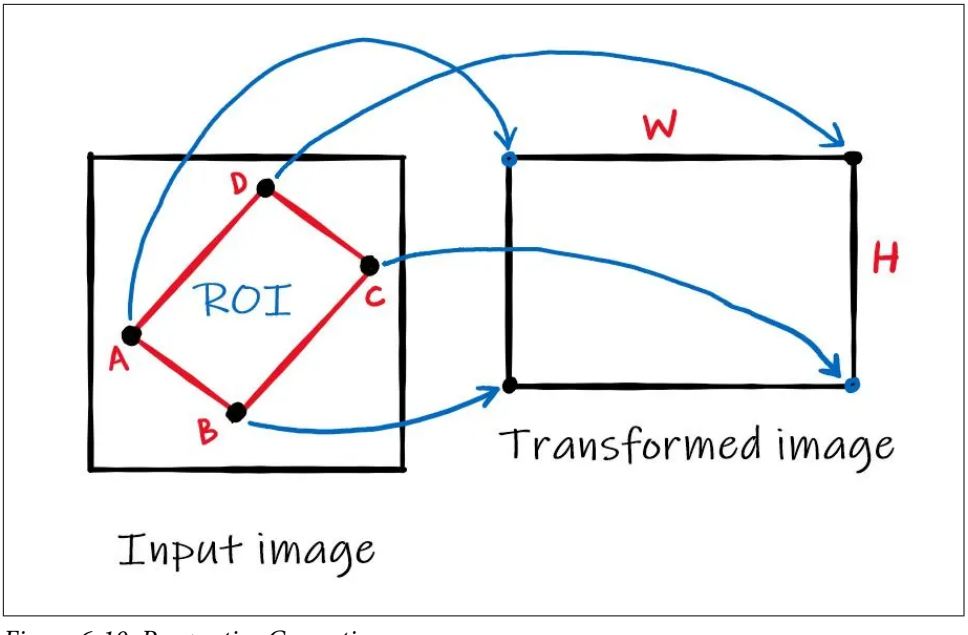


Figure 6-10. Perspective Correction

Let's do the perspective correction of the previous card image. We'll show you the algorithm after calling the `edgeProcessing` and finding and filtering the contours to one with four corners.

```
protected Mat correctingPerspective(Mat img) {

    Mat imgCopy = this.edgeProcessing(img);

    final Optional<MatOfPoint> matOfPoints = allContours.stream()
 ...
 .filter(approx -> approx.total() == 4)
 .findFirst(); ❶

    final MatOfPoint2f approxCorners = matOfPoints.get(); ❷

    MatOfPoint2f corners = arrange(approxCorners); ❸
    MatOfPoint2f destCorners  = getDestinationPoints(corners); ❹

    final Mat perspectiveTransform = Imgproc
 .getPerspectiveTransform(corners, destCorners); ❺
    org.opencv.core.Mat dst = new org.opencv.core.Mat();
    Imgproc.warpPerspective(img, dst, perspectiveTransform, img.size()); ❻

    return dst;
}
```

❶ Edge Detection steps

❷ Gets the calculated corner points

❸ Order of points in the `MatOfPoint2f` needs to be adjusted for the `getPerspectiveTransform` method

❹ Calculate the destination points using the L2 norm

❺ Compute the perspective transformation matrix to move image from original to the destination corners

❻ Apply the perspective transformation matrix

There are two methods not explained yet. One arranges the points in the correct order to be consumed by `getPerspectiveTransform`:

```
private MatOfPoint2f arrange(MatOfPoint2f approxCorners){

    Point[] pts = approxCorners.toArray();
    return new MatOfPoint2f(pts[0], pts[3], pts[1], pts[2]); ❶
}
```

❶ Rearrange the list of points to new positions

The other method is `getDestinationPoints`, which calculates the destination points of each corner to correct the image's distortion.

In this case, we use the L2 norm (or Euclidean norm), which gives the distance from the origin to the point, to translate the original (yet distorted) coordinates to new coordinates where the image element is not distorted.

Figure 6-11 shows the L2 norm formula:

$$\|\boldsymbol{x}\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}.$$

Figure 6-11. L2 Norm formula

The Figure 6-12 helps you to visualize this transformation.

The cross markers (+) are the original corner points of the element. You can see that they form a not-perfect rectangle, but they fit perfectly to the element,so it is distorted.

The star markers (*) are the points calculated using the Euclidean norm as the element's final coordinates.

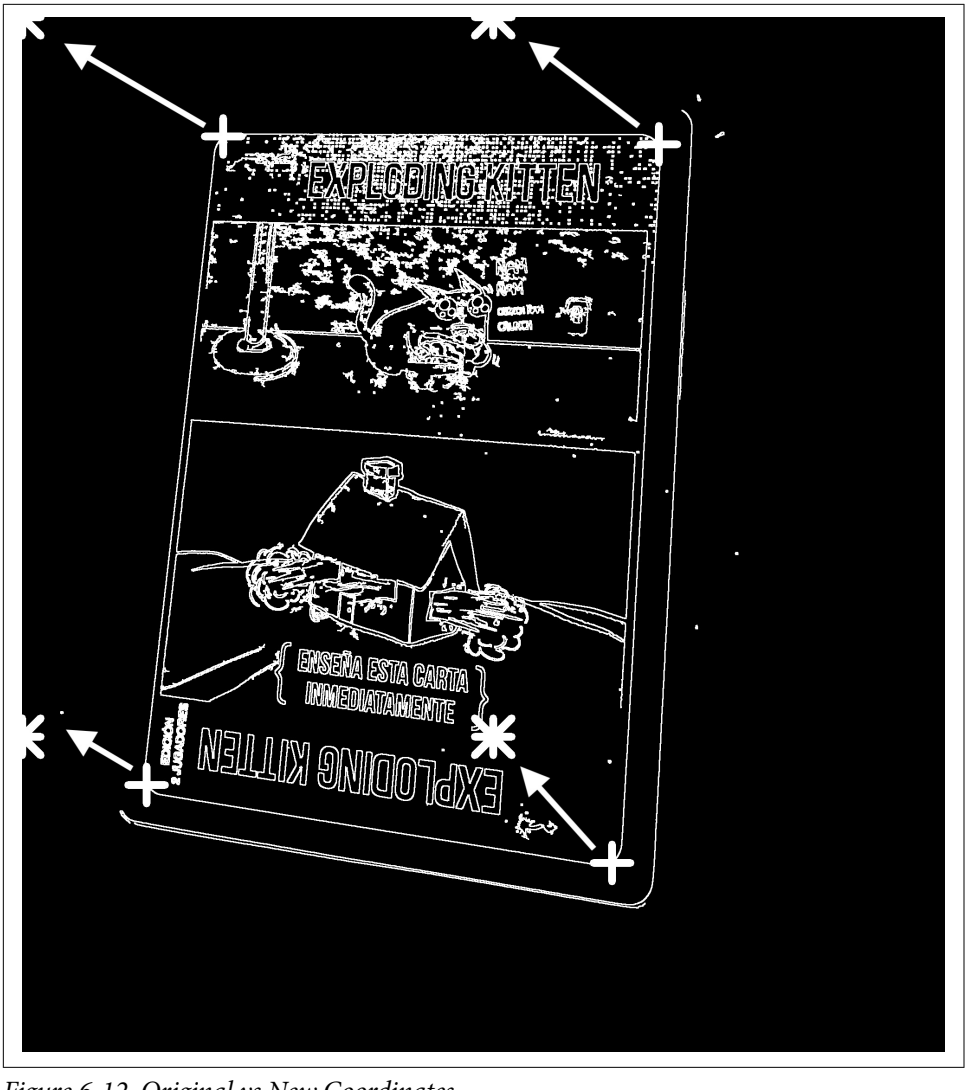


Figure 6-12. Original vs New Coordinates

The code to calculate the new coordinates is shown in the following snippet:

```
    private double calculateL2(Point p1, Point p2) {

        double x1 = p1.x;
        double x2 = p2.x;
        double y1 = p1.y;
        double y2 = p2.y;

        double xDiff = Math.pow((x1 - x2), 2);
        double yDiff = Math.pow((y1 - y2), 2);
```

```
        return Math.sqrt(xDiff + yDiff);
    }

    private MatOfPoint2f getDestinationPoints(MatOfPoint2f approxCorners) {
        Point[] pts = approxCorners.toArray();

        double w1 = calculateL2(pts[0], pts[1]);
        double w2 = calculateL2(pts[2], pts[3]);
        double width = Math.max(w1, w2);

        double h1 = calculateL2(pts[0], pts[2]);
        double h2 = calculateL2(pts[1], pts[3]);
        double height = Math.max(h1, h2);

        Point p0 = new Point(0,0);
        Point p1 = new Point(width -1,0);
        Point p2 = new Point(0, height -1);
        Point p3 = new Point(width -1, height -1);

        return new MatOfPoint2f(p0, p1, p2, p3);
    }
```

Figure 6-13 shows the original card image without distortion after applying the `correctingPerspective` method. See how the card lines parallel the image so no distortion is appreciated.

Figure 6-13. Image with no distortion

Let's see another use case related to image processing in this chapter. The following section uses the OpenCV library to read barcodes or QR codes from images.

Reading QR/BarCode

OpenCV library implements two classes for QR or Barcode recognition.

OpenCV implements several algorithms to recognize *codes*, all implicitly called when using the `org.opencv.objdetect.GraphicalCodeDetector` class.

These algorithms are grouped into three categories:

Initialization
: Constructs barcode detector.

Detect
: Detects graphical code in an image and returns the quadrangle containing the *code*. This step is important as the *code* can be in any part of the image, not a specific position.

Decode
: Reads the contents of the barcode. It returns a UTF8-encoded `String` or an empty string if the code cannot be decoded. As a pre-step, it locally binarizes the image to simplify the process.

Let's explore using OpenCV for reading QR and Bar Codes:

Barcodes

The barcode's content is decoded by matching it with various barcode encoding methods. Currently, *EAN-8*, *EAN-13*, *UPC-A* and *UPC-E* standards are supported.

The class for recognizing barcodes is `org.opencv.objdetect.BarcodeDetector`, implementing a method named `detectAndDecode`, calling both detection and decoder parts, so all process is executed with a single call.

Given the barcode shown at Figure 6-14:

Figure 6-14. Image with a barcode

The following code shows getting the barcode as a `String` from the previous image:

```
protected String readBarcode(Mat img) {
    BarcodeDetector barcodeDetector = new BarcodeDetector(); ❶
    return barcodeDetector.detectAndDecode(img); ❷
}
```

❶ Initialize the class

❷ Executes the detect and decode algorithms

Scanning a barcode is not a difficult, and in similar way, you scan a QR code.

QR Code

OpenCV provides the `org.opencv.objdetect.QRCodeDetector` class for scanning QR codes. In this case, you call the overloaded `detectAndDecode` method where the second argument is an output `Map` of vertices of the found graphical code quadrangle.

```
QRCodeDetector qrCodeDetector = new QRCodeDetector(); ❶
Mat ouput = new Mat(); ❷
String qr = qrCodeDetector.detectAndDecode(img, output); ❸
```

❶ Initialize the class

❷ Defines output `Mat`

❸ Executes the detect and decode algorithms

Draw the detected marks on the image:

```
for (int i = 0; i< output.cols(); i++) { ❶
    Point p = new Point(pointsMat.get(0, i)); ❷
    Imgproc.drawMarker(img, p, OpenCVMain.GREEN,
                           Imgproc.MARKER_CROSS, 5, 10) ❸
}
```

❶ It is a 4x1 matrix

❷ Each column contains the coordinates of one point

❸ Draws markers

After this brief and practical introduction to image processing, let's move on how to process images when they are a stream of images (e.g., a video or webcam).

Stream Processing

OpenCV provides some classes for reading, extracting information, and manipulating videos. These videos can be a file, a sequence of images, a live video from (network) webcam, or any capturing device that is addressable by a *URL* or *GStreamer* form.

The main class to manipulate videos, or get information like Frames Per Second or the size of the video is `org.opencv.videoio.VideoCapture`. Moreover, this class implements a method to read each frame as a matrix (`Mat` object) to process it, as shown in the previous sections of this chapter.

The `org.opencv.videoio.VideoWriter` provides a method to store the processed videos in several formats, the most accepted, the `mpg4` format.

```
Let's dig into how to utilize that.
```

Processing Videos

Let's develop a method that reads a video file and applies the binarization process to all frames to finally store the processed video:

```
protected void processVideo(Path src, Path dst) {
    VideoCapture capture = new VideoCapture(); ❶

    if (!capture.isOpened()) {
        capture.open(src.toAbsolutePath().toString()); ❷
 }

    double frmCount = capture.get(Videoio.CAP_PROP_FRAME_COUNT); ❸
    System.out.println("Frame Count: " + frmCount);

    double fps = capture.get(Videoio.CAP_PROP_FPS); ❹
    Size size = new Size(capture.get(Videoio.CAP_PROP_FRAME_WIDTH),
                         capture.get(Videoio CAP_PROP_FRAME_HEIGHT)); ❺

    VideoWriter writer = new VideoWriter(dst.toAbsolutePath().toString(),
            VideoWriter.fourcc('a', 'v', 'c', '1'),
 fps, size, true); ❻

    Mat img = new Mat();
    while (true) {
        capture.read(img); ❼

        if (img.empty())
            break; ❽

        writer.write(this.binaryBinarization(img)); ❾ ❿

 }

    capture.release(); ⓫
    writer.release();
}
```

❶ Instantiates the main class for video capturing

❷ Load the file

❸ Gets the number of frames

❹ Gets the Frame Per Seconds

❺ Gets the dimensions of the video

❻ Creates the class to write video to disk

❼ Reads a frame and decodes as a `Mat`

❽ If there are no more frames, skip the loop

❾ Process the matrix

❿ Writes the processed matrix to the output stream

⓫ Closes the streams and writes the content

With these few lines of code, you process "offline" videos; in the next section, you'll explore processing video in real time.

Processing WebCam

Let's implement a simple method of capturing a snapshot from the computer camera:

```
protected Mat takeSnapshot() {

    VideoCapture capture = new VideoCapture(0); ❶

    Mat image = new Mat();

    try {
        TimeUnit.SECONDS.sleep(1); ❷
 } catch (InterruptedException e) {
        throw new RuntimeException(e);
 }

    capture.read(image); ❸
    capture.release();

    return image;
}
```

❶ id of the video capturing device to open. The default one is `0`

❷ Wait till the device is ready

❸ Captures the image

We suggest you use a library like Awaitility to implement waits. For the sake of simplicity, we leave it as a `sleep` call.

Capturing from a camera is similar to making a video, but the class is configured to the device's location instead of a file.

You've now gained a good understanding of the Open CV project's capabilities for manipulating images and videos, detecting objects, and reading Barcodes and QR codes in Java. But before finishing this chapter, we have some final words about Open CV, its integration with DJL, and a Java alternative to Open CV.

OpenCV and Java

So far, you've probably noticed that even though OpenCV is well integrated with Java, the API mimics the C/C++ programming language. For example:

- Using parameters for output inherited by the pass with reference (pointers) in C++.
- Using integers instead of `enums` for configuration constants or parameter names.
- Not using exceptions to indicate errors, only booleans for setting if the operation succeeds or not, or returning an empty value (matrix with 0, blank strings) as none successful operation.
- There is a need to call the `release` method to close the object and free resources. In Java, you could use `try-with-resources`.
- Unit class is a matrix instead of an image.

When using OpenCV, we recommend creating a Java wrapper around the library, addressing some of these "problems" and implementing them in Java. You can do this by:

- Using `return` statements instead of objects as reference. In case of multiple return values, create a Java `record`.
- Configuring the integer value by putting constants in an `enum` with a field instead of an integer constant.
- Making classes implement `AutoClosable` to release the resources.

Other options for image processing exist, such as BoofCV, which offers capabilities similar to those of OpenCV. For this book, we advocated for OpenCV because of its tight integration with the *DJL* library.

DJL creates a layer of abstraction around OpenCV, fixing some of the problems mentioned in the previous section. Moreover, it implements some image processing algorithms, such as drawing boundaries in an image, so you don't need to implement them yourself.

To use this integration, register the following dependency:

```
<dependency>
    <groupId>ai.djl.opencv</groupId>
    <artifactId>opencv</artifactId>
    <version>0.29.0</version>
</dependency>
```

To load an image, use any of the methods provided by the `ai.djl.modality.cv.BufferedImageFactory` class. You can load an image from various sources like URLs, local files, or input streams.

When loading an image, it returns a class of type `ai.djl.modality.cv.Image`, which provides a suite of image manipulation functions to pre- and post-process the images and save the final result.

Let's try cropping an image to get only the left half of the image using this integration:

```
Image img = BufferedImageFactory.getInstance().fromFile(pic); ❶

int width = img.getWidth(); ❷
int height = img.getHeight();

Image croppedImg = img.getSubImage(0, 0, width / 2, height); ❸

croppedImg.save(
    new FileOutputStream("target/lresizedHands.jpg"),
        "jpg"); ❹
```

❶ Reads image from a file

❷ Gets information about the size of the image

❸ Crops the image, creating a new copy

❹ Stores the image to disk

Any of these methods can throw an exception in case of an error instead of returning an empty response or a null.

The `Image` class has the `getWrappedImage` method to get the underlying representation of the image in OpenCV object (usually a `Mat`).

We'll utilize this library deeply in the following chapter.

With a good understanding of image and video processing, let's move forward to the last section of this chapter, where we'll use Optical Character Recognition (or OCR) to transform an image with text into machine-encoded text.

OCR

Optical Character Recognition (*OCR*) is a group of algorithms that convert documents, such as scanned paper documents, *PDFs*, or images taken by a digital camera, into text data. This is useful for processing text content (for example, extracting important information or summarizing the text) or storing text in a database to make it searchable.

OCR process usually has three phases to detect characters:

Pre-Processing
: Apply some image processing algorithms to improve character recognition. These algorithms usually adjust the perspective of the image, binarize, reduce noise, or perform layout analysis to identify columns of text.

Text recognition
: This phase detects text areas, recognizing and converting individual characters into digital text.

Post-Processing
: The accuracy of the process increases if, after the detection of words, these words are matched against a dictionary (this could be a generalistic dictionary or more technical one for a specific field) to detect which words are valid within the document. Also, this process can be more complex, not just detecting exact word matches, but also similar words. For example, "Regional Cooperation" is more common in English than "Regional Cupertino."

Multiple *OCR* libraries exist, but the Tesseract library is one of the most used and accurate.

Tesseract (*https://github.com/tesseract-ocr/tesseract*) is an optical character recognition engine released as an open-source project under the Apache license. It can recognize more than 116 languages, process right-to-left text, and perform layout analysis.

The library is written in C and C++, and, like the OpenCV library, a Java wrapper surrounds it to make calls to the library transparently from Java classes.

To get started with *Tesseract Java*, which, we'll refer to it as *Tesseract* from here on, register the following dependency on your build tool:

```
<dependency>
    <groupId>org.bytedeco</groupId>
    <artifactId>tesseract-platform</artifactId>
```

```
    <version>5.3.4-1.5.10</version>
</dependency>
```

One important thing to do before using *Tesseract* is to download `tessdata` files. These files are trained models for each supported language and you can download them from Tesseract GitHub (*https://github.com/tesseract-ocr/tessdata*). For example, the file in English is named `eng.traineddata`.

Download the file and store it at `src/main/resources/eng.traineddata`.

Now, let's develop a simple application that scans and extracts PDF text.

The main class interacting with Tesseract is `org.bytedeco.tesseract.TessBaseAPI`. This class is an interface layer on top of the Tesseract instance to make calls for initializing the API, setting the content to scan, or getting the text from the given image.

The first step is to instantiate the class and call the `init` method to initialize the *OCR* engine, setting the path of the folder with all `tessdata` files and the language name to load for the current instance.

For our example:

```
static { ❶
 api = new TessBaseAPI();
    if (api.Init("src/main/resources", "eng") != 0) {
        throw new RuntimeException("Could not initialize Tesseract.");
 }
}
```

❶ Executes this method only once as it takes a lot of time

After the class initialization, you can start scanning and processing images. The input parameter must be of type `org.bytedeco.leptonica.PIX` which is the image to scan. To load an image into the `PIX` object, use the `org.bytedeco.leptonica.global.leptonica.pixRead` static method.

Finally, use the `SetImage` method to set the `PIX` and `GetUTF8Text` to get the text representation of the image.

`TessBaseAPI` is not thread-safe. For this reason, it is really important to protect access to the read and scan methods with any of the Java synchronization methods to avoid concurrent processing.

Next snippet reads an image with a text, and returns the text as a `String` object.

```
private static final ReentrantLock reentrantLock = new ReentrantLock();
```

```
String content = "";

try (PIX image = pixRead(imagePath.toFile().getAbsolutePath())) { ❶ ❷

    reentrantLock.lock(); ❸
    BytePointer bytePointer;
    try {
        api.SetImage(image); ❹
        bytePointer = api.GetUTF8Text(); ❺
        content = bytePointer.getString(); ❻

    } finally {
        if (bytePointer != null) {
            bytePointer.close();
        }
        reentrantLock.unlock();
    }

} catch (Exception e) {
}
```

❶ Using try-with-resources for automatic resource management

❷ Loads image from file location

❸ Lock the code that uses the shared resource

❹ Sets the image to Tesseract

❺ Gets the text scanned in the image as a pointer

❻ Pointer to String

The last execution step is closing down Tesseract and freeing up all memory. This step should be executed only when you no longer need to use Tesseract.

```
public static void cleanup() {
    api.End();  ❶
}
```

❶ Clean Up Resources

These are all the topics we cover in this book about image processing; in further chapters, you'll see examples of using the image processing algorithms applied to AI/ML models.

Next steps

In this chapter, you learned the basics of image/video processing. You'll typically need to apply these algorithms when AI models use images as input parameters.

Image processing is a vast topic that would require a book on its own, but we think a brief introduction of the most common used algorithms to get a good understanding of image processing in Java is important for the topic of this book.

Because of the direct relationship between Open CV and Open CV Java, you can translate any tutorial, video, book, or examples written in the Open CV \C++ version to Java.

By now you understand AI/ML and it's integration with Java, as well as how to infer models in Java using *DJL*, and consume these models with Java clients (REST or gRPC) or with LangChain4j. Moreover, this chapter showed how to preprocess images to adapt them to be suitable input parameters for a model.

However, there are pieces we haven't covered yet such as:

- Model Context Protocol (*MCP*)
- Streaming models
- Security and guards

In the book's last chapter, we'll cover these important topics, which in our opinion don't fit in any of the previous chapters.

About the Authors

Markus Eisele is a technical marketing manager in the Red Hat Application Developer Business Unit. He has been working with Java EE servers from different vendors for more than 14 years and gives presentations on his favorite topics at international Java conferences. He is a Java Champion, former Java EE Expert Group member, and founder of Germany's number-one Java conference, JavaLand. He is excited to educate developers about how microservices architectures can integrate and complement existing platforms, as well as how to successfully build resilient applications with Java and containers. He is also the author of *Modern Java EE Design Patterns* and *Developing Reactive Microservices* (O'Reilly). You can follow more frequent updates on Twitter and connect with him on LinkedIn.

Alex Soto Bueno is a director of developer experience at Red Hat. He is passionate about the Java world, software automation, and he believes in the open source software model. Alex is the coauthor of Testing Java Microservices (Manning), Quar- kus Cookbook (O'Reilly), and the forthcoming Kubernetes Secrets Management (Manning), and is a contributor to several open source projects. A Java Champion since 2017, he is also an international speaker and teacher at Salle URL University. You can follow more frequent updates on his *Twitter feed* and connect with him on *LinkedIn*.

Natale Vinto is a software engineer with more than 10 years of expertise on IT and ICT technologies and a consolidated background on telecommunications and Linux operating systems. As a solution architect with a Java development background, he spent some years as EMEA specialist solution architect for OpenShift at Red Hat. Today, Natale is a developer advocate for OpenShift at Red Hat, helping people within communities and customers have success with their Kubernetes and cloud native strategy. You can follow more frequent updates on Twitter and connect with him on LinkedIn.