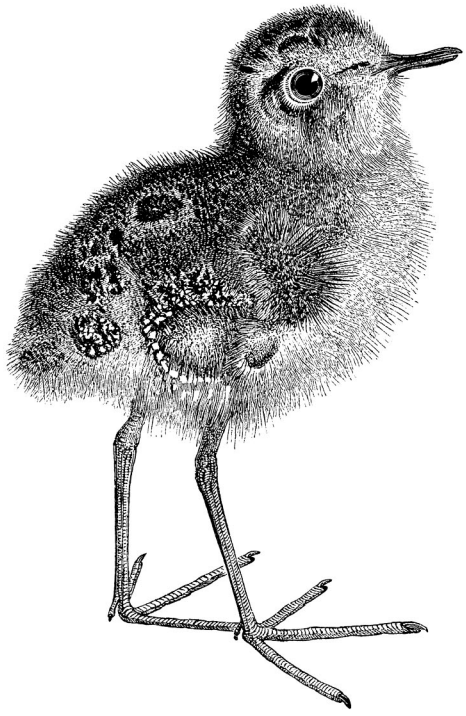
O'REILLY®

Applied AI for Enterprise Java Development

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



Early Release
RAW & UNEDITED
Compliments of
Red Hat Developer

Alex Soto Bueno,
Markus Eisele
& Natale Vinto

Launch your Developer Sandbox for Red Hat OpenShift today

`red.ht/sandb0x`

Red Hat Developer

Build here. Go anywhere.

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

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

978-1-098-17444-6

[FILL IN]

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

This chapter talks about the most prominent AI models and their unique specialties. We help you understand the available models and you'll learn how to choose the right model for your use case. We also cover model serving infrastructure—how to deploy, scale, and manage AI models in both cloud and local environments. This chapter equips you with the knowledge to serve models efficiently in production.

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 using Quarkus, a lightweight Java framework. 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.

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.

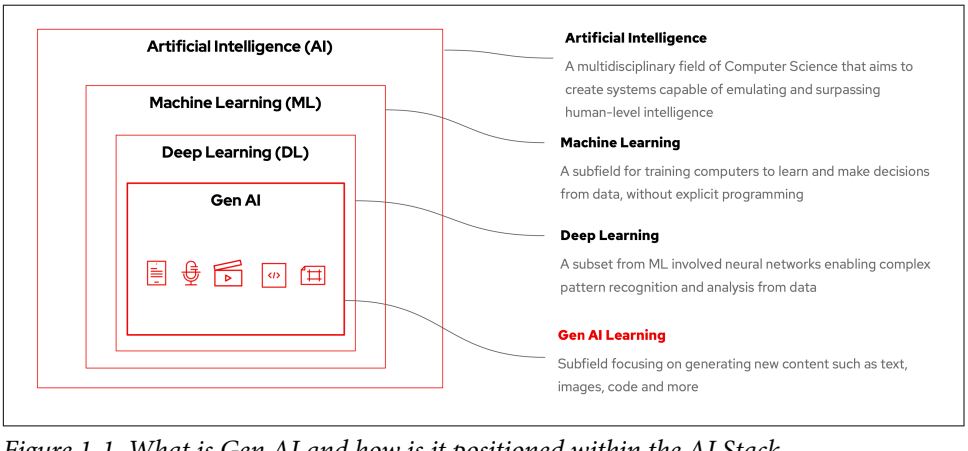


Figure 1-1. What is Gen AI and how is it positioned within the AI Stack.

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.

Looking at Machine Learning (ML) compared to Deep Learning (DL), many people assume that they're one and the same. However, while both techniques share some similarities, they also have major differences in the way they model reality. One key difference lies in their depth - ML algorithms can be shallow or deep, whereas DL specifically refers to the use of neural networks with multiple layers. This added com-

plexity 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.

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 contex-

tualized 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. 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 particular takeaway from this is that running LLMs on local developer machines is quite challenging. These models are often very large in size, requiring significant computational resources that can easily overwhelm not just the CPU, but also memory and disk space on typical development machines. This makes working with such models difficult in local environments. In later chapters, particularly Chapter 4, 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 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.

Open Source Training Data

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 open source models is the growing need for explainable and transpartent decision making by models. With recent incidents generated by corporate chat bots only being indivudual examples, it is important that companies understand how their models work and can trust in their output. As AI becomes more relevant in all areas, people want to know how decisions are made and what factors influence them. Instead of treating models like black-boxes, transparency and openness builds trust in AI systems. On top, governments and regulatory bodies are starting to require a certain level of transparency from AI-driven decision-making processes, especially in healthcare, finance, and law enforcement. Lastly there is also growing concerns about the potential for bias and unfair treatment. While there are technical approaches to explaining model behaviour and giving insights into behaviours most of the safeguards in place today aren't perfect and require a certain amount of safeguards in place. We will talk about this in chapter three.

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.

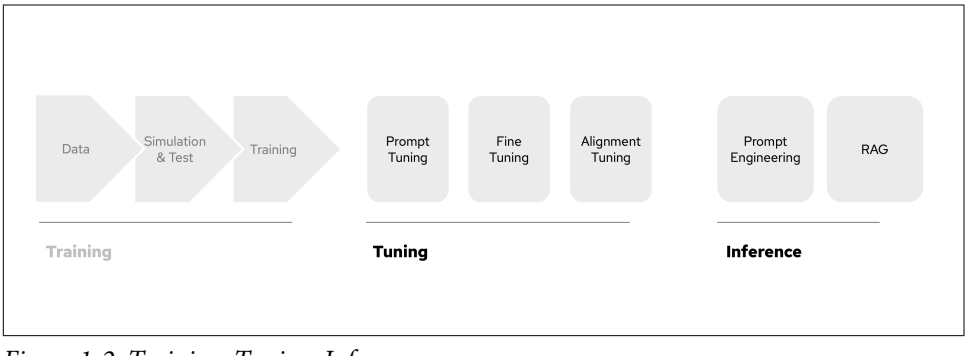


Figure 1-2. Training, Tuning, Inference.

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 hughe amounts of resources and they are generally exceptionally good at general purpose tasks. The next phase a model usually goes through is known as 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 spe-

cific 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

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.

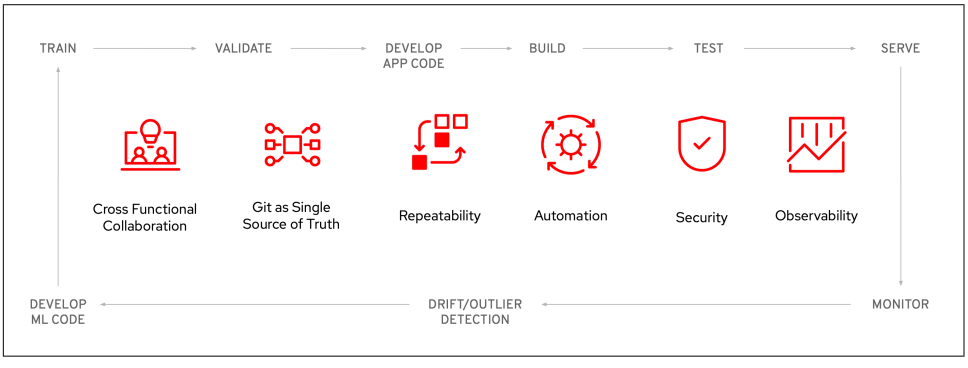


Figure 1-3. DevOps and MLOps

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.

The two approaches are ultimately deeply interwined and complement each other. To make things even more complicated, MLOps is kind of a catch-all term. There's plenty of other terms that are closely related. These include: ModelOps, LLMOps, or DataOps. They all refer to a similar set of practices in different combinations. There is no single "correct" way to combine DevOps and MLOps because the approach will depend on the organization, the teams involved, and their established practices. Some teams may prefer closer integration between data science and development, while others may want clear divisions between model development and application development. Organizational structure, team expertise, and project goals play significant roles in how the workflows are integrated.

For example, in a smaller organization, data scientists and developers might work more closely, sharing codebases and automating model deployment alongside application code. In larger organizations, teams might be more specialized, requiring

distinct processes for model management and software engineering, leading to a more modular approach to integration.

In summary, DevOps and MLOps work together by borrowing best practices from each other, but with the added layers of data management, model training, and continuous monitoring in MLOps. The right approach will depend on team collaboration, project complexity, and organizational needs.

Conclusion

In conclusion, the development and deployment of Large Language Models (LLMs) require a solid understanding of the training, tuning, and inference processes involved. As the field of MLOps continues to evolve, it is essential to recognize the key differences between DevOps and MLOps, with the latter focusing on the specific needs of machine learning model development and deployment. By acknowledging the intersection approaches required for cloud-native application- and model-development, teams can effectively collaborate across disciplines and bring AI-infused applications successfully into production.

Chapter Two will introduce you to various classifications of LLMs and unveal 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
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 2-1 shows the transformation between a JSON document and the model parameters done by the Inference API:

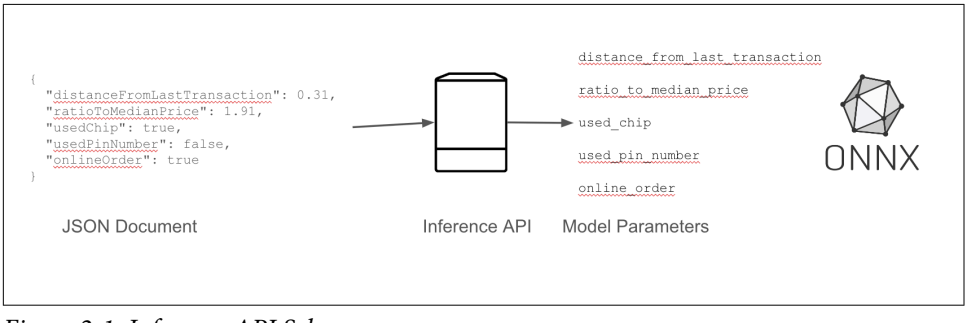


Figure 2-1. Inference API Schema

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 2-2 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:

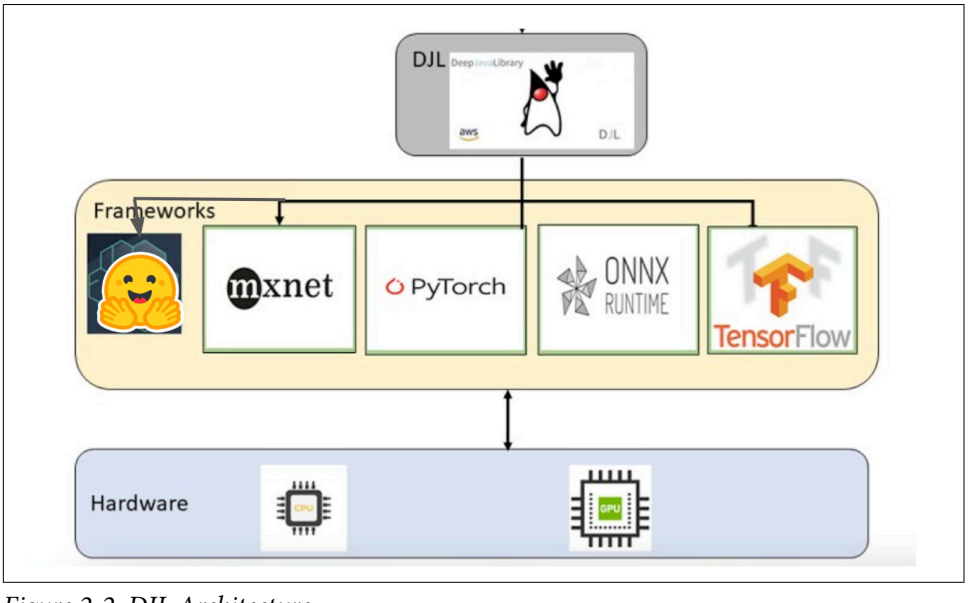


Figure 2-2. DJL Architecture

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 2-3 shows what we want to implement:

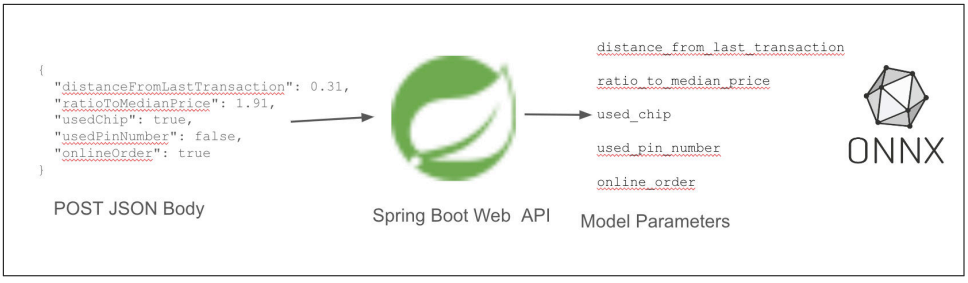


Figure 2-3. Spring Boot Rest API Schema

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 2-4 shows the Spring Initializr parameters for this example:

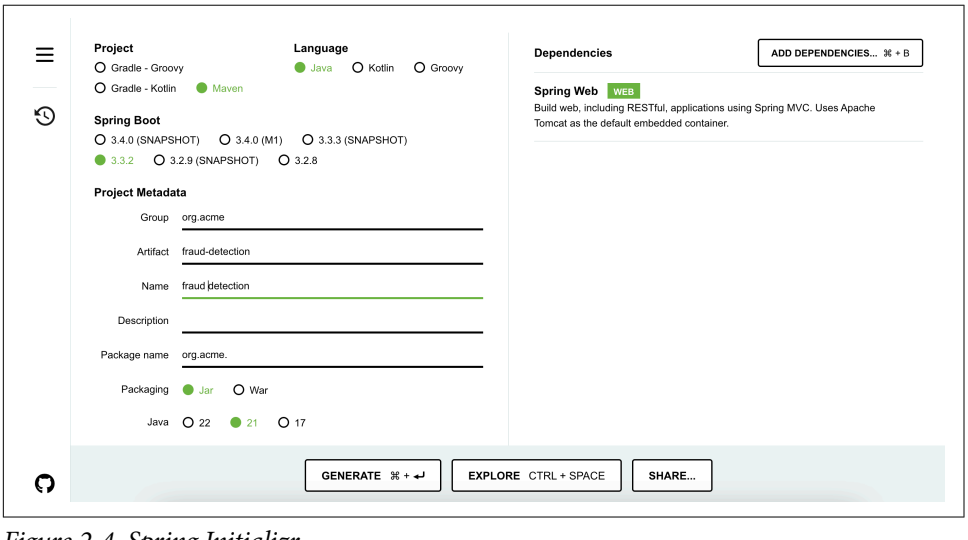


Figure 2-4. Spring Initializr

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 o 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 2-5 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.

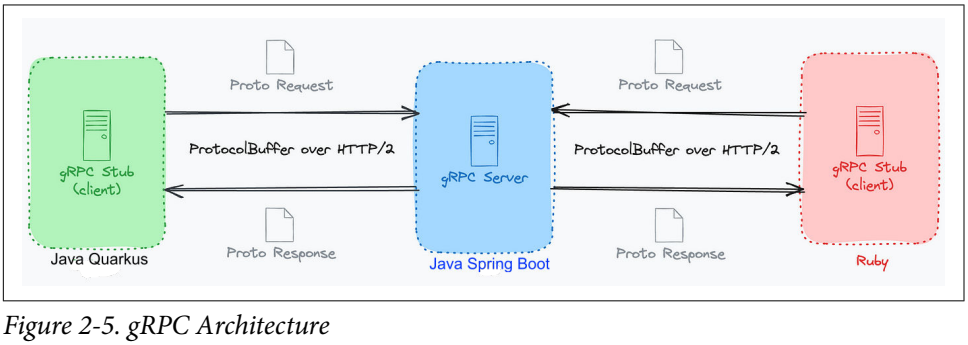


Figure 2-5. gRPC Architecture

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 2-6 summarizes the process:

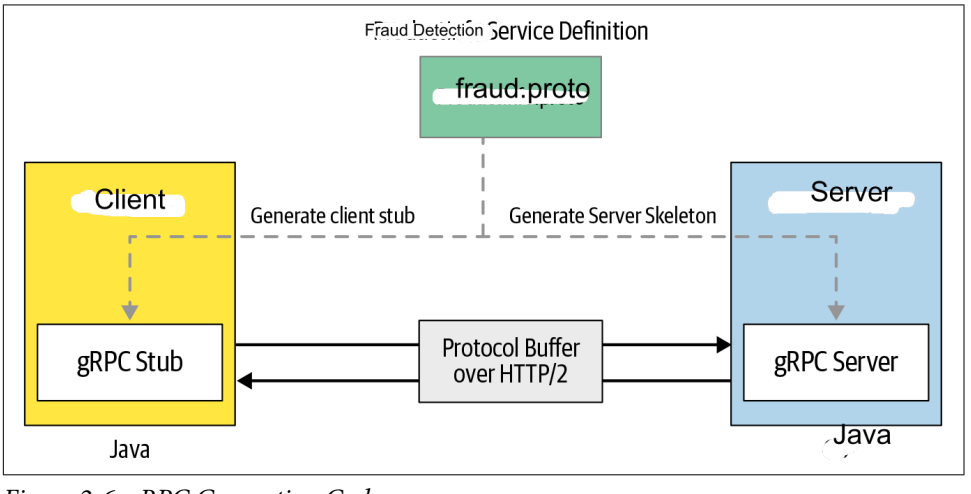


Figure 2-6. gRPC Generation Code

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.

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.