**LARGE LANGUAGE MODELS AND LANGCHAIN4J**

Antonio Jurjević, Jozo Pejić, Mateo Plavčić

**Content**

[1 Artificial Intelligence (AI) 3](#_Toc200711813)

[1.1 Applications of Artificial Intelligence 3](#_Toc200711814)

[1.2 Artificial Intelligence Techniques 4](#_Toc200711815)

[1.2.1 Machine Learning (ML) 4](#_Toc200711816)

[1.2.2 Deep Learning (DL) 4](#_Toc200711817)

[1.2.3 Natural Language Processing (NLP) 4](#_Toc200711818)

[1.2.4 Computer Vision 5](#_Toc200711819)

[2 Large Language Models 5](#_Toc200711820)

[2.1 Evolution of Language Models 5](#_Toc200711821)

[2.2 Transformer Architecture 6](#_Toc200711822)

[2.3 LLM Training 6](#_Toc200711823)

[2.4 LLM Capabilities 6](#_Toc200711824)

[2.5 Limitations and Challenges of LLMs 7](#_Toc200711825)

[2.6 Popular LLMs 7](#_Toc200711826)

[3 LangChain4j 8](#_Toc200711827)

[3.1 Architecture and Design 9](#_Toc200711828)

[3.2 LLM Integration in Java 10](#_Toc200711829)

[3.3 Prompt Engineering and Templates 10](#_Toc200711830)

[3.3.1 Basic Example 11](#_Toc200711831)

[3.3.2 Multi Variable Templates 11](#_Toc200711832)

[3.3.3 Use with Chains 11](#_Toc200711833)

[3.3.4 Conditional and Dynamic Prompt Selection 12](#_Toc200711834)

[3.4 Memory and Stateful Chat in LangChain4j 13](#_Toc200711835)

[3.4.1 Short-Term Memory (In-Memory) 13](#_Toc200711836)

[3.4.2 Long-Term Memory 13](#_Toc200711837)

[3.5 Agents and Tool Use 14](#_Toc200711838)

[3.5.1 How Agents Work 14](#_Toc200711839)

[3.5.2 Tool Chaining and Decision-Making 15](#_Toc200711840)

[3.5.3 Practical Applications 15](#_Toc200711841)

[3.6 Document Retrieval and RAG (Retrieval-Augmented Generation) 15](#_Toc200711842)

[3.6.1 Architecture and Workflow 15](#_Toc200711843)

[3.6.2 Advantages of Using Langchain4j for Retrieval-Augmented Generation (RAG) 16](#_Toc200711844)

[3.6.3 Future Enhancements: Hybrid Search and Re-ranking 17](#_Toc200711845)

[3.7 LangChain4j Use Cases 17](#_Toc200711846)

[4 LangChain4j Versus Other Frameworks 18](#_Toc200711847)

[4.1 LangChain (Python) 18](#_Toc200711848)

[4.2 Haystack (Python) 18](#_Toc200711849)

[4.3 OpenAI SDKs 19](#_Toc200711850)

[4.4 Strengths of LangChain4j in the Java Ecosystem 19](#_Toc200711851)

[5 LangChain4j Integration with Ollama 20](#_Toc200711852)

[5.1 What is Ollama? 20](#_Toc200711853)

[5.2 Benefits of Using Ollama with LangChain4j 20](#_Toc200711854)

[5.3 Connecting LangChain4j with Ollama 21](#_Toc200711855)

[5.4 Use Cases for LangChain4j + Ollama 21](#_Toc200711856)

[6 Demo Project: Gitlab Student Practice Report Generation 22](#_Toc200711857)

[6.1 Fetching data from Gitlab and storing it into a database 22](#_Toc200711858)

[6.2 Student selection 23](#_Toc200711859)

[6.3 Selecting relevant data from the database 24](#_Toc200711860)

[6.3.1 Problem with wiki pages fetching 24](#_Toc200711861)

[6.4 Report generation 24](#_Toc200711862)

# 

# Artificial Intelligence (AI)

**Artificial Intelligence (AI)** is a field of computer science focused on creating systems capable of performing tasks that would typically require human intelligence. These tasks include decision-making, visual perception, speech recognition, language translation, and natural language understanding. The development of AI has moved from rule-based systems in the early days to data-driven approaches, particularly machine learning and deep learning.

One of the most significant advancements in AI in recent years is the rise of **Large Language Models (LLMs)**. These are deep learning models trained on vast amount of text data, allowing them to generate, translate, summarize, and reason over human language with impressive accuracy and coherence. LLMs such as **OpenAI’s GPT-4**, **Google’s PaLM**, and **Meta’s LLaMA** exemplify the power of transformer-based architectures.

## Applications of Artificial Intelligence

Artificial Intelligence is no longer limited to academic research or niche applications. Today, AI technologies are deeply embedded in various industries and everyday services, transforming how humans interact with machines and digital systems. Some of the most important applications are:

1. **Natural Language Processing (NLP)** – NLP is a subfield of AI focused on enabling computers to understand, generate and interpret human language. NLP is commonly used in:

* Chatbots and virtual assistants – Use LLM-s to engage in conversations, answer questions and perform tasks. The most popular examples are Siri, Alexa and ChatGPT.
* Machine translation systems – Translate text between languages using deep learning models (Google Translate).
* Text summarization and question answering – Commonly used in customer service and market analysis.

1. **Computer Vision** – Computer Vision enables machines to interpret and analyse visual data. Use cases include:

* Facial recognition – used in security and smartphone unlocking systems.
* Autonomous vehicles – rely on traffic sign recognition and object detection.
* Medical imaging – assisting doctors in recognising diseases such as cancer and pneumonia.

1. **Robotics and Automation** – AI enhances robotics by aiding in decision making scenarios and real time control. It is extensively used in industrial robots operating in a manufacturing line, medical robots assisting surgeons with operations requiring razor sharp precision.
2. **Predictive Analytics** – Predictive analytics uses historical data to forecast future events. It is commonly used in finance and healthcare sectors to predict stock market trends or asses patient risk.

## Artificial Intelligence Techniques

Artificial Intelligence uses a variety of techniques that enable machines to perceive, learn, reason, and act. These techniques can be categorized based on their learning paradigms, reasoning capabilities, and the tasks they are designed to perform.

### Machine Learning (ML)

Machine Learning is a subset of AI that focuses on building systems that learn from data as opposed to being explicitly programmed. The term *machine learning* was coined in 1959 by Arthur Samuel, an IBM employee and pioneer in the field of computer gaming and artificial intelligence. Machine learning approaches are traditionally divided into four broad categories, which correspond to learning paradigms.

1. **Supervised learning** – in supervised learning models are trained on labelled datasets, each input has a corresponding output. Common algorithms are: Linear Regression, Decision Trees and Neural Networks. It is commonly used in image classification and medical analysis.
2. **Unsupervised learning** – deals with unlabeled data, model tries to find hidden patterns or groupings in data. Common algorithms are: K-Means Clustering, Hierarchical Clustering and Principal Component Analysis. Mostly used in anomaly detection and customer segmentation in marketing.
3. **Semi-Supervised learning** – combines small amount of labeled data with a large amount of unlabeled data. Used in facial recognition systems.
4. **Reinforcement Learning** - An agent learns by interacting with an environment. It receives feedback in the form of rewards or penalties and learns to take actions that maximize long-term rewards. Commonly used in robotics and game AI.

### Deep Learning (DL)

Deep learning is a subset of Machine learning that focuses on utilizing multi-layered neural networks to perform tasks such as classification, regression and representational learning. Key architectures:

1. **Convolutional Neural Networks (CNNs)** – used for image processing.
2. **Recurrent Neural Networks (RNNs) and** **Long Short-Term Memory (LSTM)** - used for time series and natural language.
3. **Transformers** - used in language models like BERT, GPT, etc.

Deep learning is used in speech recognition (Siri, Google Assistant), Autonomous driving and Language models.

### Natural Language Processing (NLP)

Subfield of computer science that focuses on enabling machines to understand, generate and interpret human language. Natural language processing has its roots in the 1950s.[[1]](https://en.wikipedia.org/wiki/Natural_language_processing#cite_note-1) Already in 1950, Alan Turing published an article titled "Computing Machinery and Intelligence" which proposed what is now called the Turing test as a criterion of intelligence. It is used in chatbots, machine translation and document summarization.

### Computer Vision

Computer vision is a field that deals with how computers can be made to gain understanding from images or videos. It uses techniques such as: image classification, object detection, and facial recognition. It is used in security systems, autonomous vehicles and medical image analysis.

# Large Language Models

Large language models (LLMs) are cutting edge advancement in the field of artificial intelligence and natural language processing. They are designed understand, generate and manipulate human language in a way that was previously not possible. LLMs are based on deep learning, in particular the Transformer architecture which revolutionized the field of NLP.

Unlike earlier language models, which relied on smaller datasets and were trained with limited context windows, LLMs are trained on a large amount of text sourced from books, websites, and other digital documents. These models typically contain **billions (or even trillions) of parameters**.

The primary task of LLMs is to predict the next word in a sequence, but this at first glance simple task leads to emergence of sophisticated language understanding and generating ability. That means LLMs can summarize texts, write code, answer questions etc.

The key strength of LLMs is the ability to generalize. By learning from vast amount of different texts, they develop understanding of grammar, reasoning patterns, facts about the world and they can even recognise metaphors or humour.

Despite their remarkable capabilities, LLMs also come with limitations. They can produce **hallucinations**, confident but incorrect outputs and may struggle with **logical reasoning, factual accuracy,** or **bias** inherited from training data. Additionally, training and running LLMs requires significant computational resources.

## Evolution of Language Models

The development of language models has gone through several important phases. Early systems were **rule-based**, relying on hand-crafted grammar rules to process language, which were rigid and difficult to scale. These gave way to **statistical models**, such as **n-gram models**, which estimate the probability of a word given the previous few words.

Later, models like **Hidden Markov Models (HMMs)** brought probabilistic modeling into sequence prediction, often used in part-of-speech tagging and speech recognition. However, these still had limitations in capturing context.

The next step forward came with **word embeddings** such as GloVe and Word2Vec, which allowed words to be represented as dense vectors capturing semantic meaning based on usage context. This enabled more nuanced understanding of language and served as the foundation for neural-based models.

Eventually, **Recurrent Neural Networks (RNNs)** and **Long Short-Term Memory (LSTM)** networks enabled modeling longer sequences but were difficult to train on very large texts. The introduction of the **Transformer architecture** in 2017 marked a turning point, enabling efficient parallel training and improved performance on NLP tasks. All modern LLMs are built on Transformer architecture.

## Transformer Architecture

Transformer is a deep learning architecture developed by researchers at Google and is based on the multi-head attention mechanism, which was proposed in the 2017 paper **"Attention is all you need"**. Text is converted to numerical representations called tokens, and each token is converted into a vector via lookup from a word embedding table. At each layer, each token is then contextualized within the scope of the context window with other (unmasked) tokens via a parallel multi-head attention mechanism, allowing the signal for key tokens to be amplified and less important tokens to be diminished.

The original Transformer consists of two main components: the **encoder**, which processes the input data, and the **decoder**, which generates output. In models like BERT, only the encoder is used (for understanding tasks), while models like GPT use only the decoder (for generation). Full encoder-decoder models like T5 and BART are used for tasks such as translation or summarization.

Transformers scale well because they can efficiently model long-range dependencies and leverage large datasets through parallel computation, particularly on GPUs. Positional encodings are added to maintain order information since the architecture itself does not inherently model sequence position. The architecture is the backbone of modern LLMs, enabling them to capture complex language patterns and perform multiple tasks with minimal task specific tuning.

## LLM Training

Training LLMs is a complex process involving **pretraining** and sometimes **fine-tuning**. Pretraining includes feeding LLM with massive amount of data, from Wikipedia articles, textbooks to online forums. That means LLMs can learn language patterns by predicting the next token in sequence or filling in masked words.

After pretraining, a model can be **fine-tuned** on specific tasks or domains, improving performance in targeted applications.

Challenges include managing **computational costs**, ensuring **data quality**, and handling **ethical concerns** related to the training data.

## LLM Capabilities

LLMs have demonstrated a large range of capabilities that make them adaptable across various industries and tasks. One of the most common applications is **text generation**. LLMs can compose various styles of text like poems, essays and professional emails. The generated content often mimics human-like fluency and tone, making these models useful for both creative and formal writing tasks.

Another key application is **question answering**, where LLMs can answer to both factual and contextual queries. In some benchmarks, their performance has surpassed that of specialized question-answering systems, showcasing their ability to understand and retrieve relevant information from a prompt or embedded context.

LLMs also excel in **summarization**, effectively condensing large bodies of text into concise and informative summaries. This is particularly useful in contexts such as legal documentation, academic research, or news aggregation.

In the field of **translation**, LLMs have high accuracy in multilingual tasks. They are capable of translating text between many languages while preserving meaning, tone, and context.

Another important capability is **code generation** where models such as Codex and GPT-4 can write, explain, and even debug code snippets in multiple programming languages. These models assist developers by providing autocomplete suggestions, generating boilerplate code, and helping with logic implementation.

Finally, LLMs demonstrate strong performance in **few-shot and zero-shot learning**. This means they can perform new tasks with minimal or even no explicit training examples, simply by interpreting instructions or patterns provided in the input prompt. This flexibility is one of the reasons why prompt engineering has become a crucial skill in effectively deploying LLMs.

Overall, the versatility of LLMs makes them valuable across a range of domains, including healthcare, law, education, and software engineering, among many others. As these capabilities continue to evolve, LLMs are becoming integral tools in both research and industry.

## Limitations and Challenges of LLMs

Despite their impressive capabilities, LLMs come with notable limitations. One key issue is their tendency to produce "hallucinations” which means confidently generating information that is factually incorrect, due to their reliance on statistical patterns. Additionally, these models often reflect biases found in their training data, which can lead to inappropriate or skewed outputs, particularly in sensitive contexts.

Another limitation lies in their **contextual memory**. Most LLMs operate within a fixed context window, which constrains their ability to handle very long documents or sustain extended reasoning across a conversation. Moreover, the **interpretability** of LLMs remains an open challenge, as their internal workings are complex and opaque, making it difficult to explain specific outputs or decisions.

Ethical and legal concerns also persist. Questions about misuse, data privacy, and copyright infringement continue to surface, especially as these models are increasingly adopted in real-world applications. Addressing these challenges is essential to ensure that LLMs are developed and used responsibly.

## Popular LLMs

The rapid advancement of large language models has led to the development of several leading systems, each pushing the boundaries of what artificial intelligence can achieve in natural language understanding and generation. Among the most influential is **GPT-4**, developed by OpenAI. GPT-4 is a powerful general-purpose model that excels in tasks requiring reasoning, creativity, and conversational fluency. It serves as the backbone of applications like ChatGPT and is widely used in education, content creation, customer support, and more. Its ability to generate coherent and contextually appropriate responses has made it one of the most recognized LLMs globally.

Another influential model is **Claude**, developed by Anthropic. Unlike many other models, Claude is designed with a strong emphasis on alignment, safety, and ethical behaviour. Named after Claude Shannon, this model focuses on being helpful, harmless, and honest. It is built to reduce the risk of harmful outputs and is trained with a method called constitutional AI, which guides its behaviour through a set of principles. Claude is seen as a significant step forward in developing more controllable and reliable AI systems.

**PaLM**, created by Google DeepMind, represents Google’s entry into high-capacity LLMs. PaLM has been integrated into applications such as Bard and various Google Workspace tools. It is known for its broad skillset, which includes coding, translation, and even basic reasoning tasks.

Meta’s contribution to the LLM landscape is the **LLaMA** series (Large Language Model Meta AI). LLaMA models are distinct in that Meta has released them with open-weight licensing, making them widely accessible for academic and research use. The second generation, **LLaMA 2**, has been optimized for fine-tuning and deployment in more constrained environments, such as edge devices or smaller cloud instances. This openness has fueled a large community of developers and researchers to experiment with and adapt the models for a variety of tasks.

Another significant model is **Gemini**, the successor of previously mentioned PaLM. Gemini is a multimodal model, capable of processing not only text but also images and potentially other forms of input, integrating them seamlessly in a single interaction. This makes it especially powerful for tasks that require understanding and reasoning across different media types. Gemini represent Google´s move to human like AI interactions.

Each of these models, GPT-4, Claude, PaLM, LLaMA, and Gemini bring unique strengths and priorities to the AI ecosystem. While GPT-4 emphasizes general-purpose usability and broad accessibility through APIs, Claude prioritizes safety and alignment. PaLM and Gemini focus on integration with Google’s ecosystem and multimodality, and LLaMA supports openness and research flexibility.

# LangChain4j

As LLMs are becoming more and more popular and central to modern applications, many frameworks have emerged to make it easier for developers to integrate LLMs into real world systems. One such framework is **LangChain**, originally created in Python and JavaScript, designed to streamline the process of building LLM-powered applications by combining models with external data, tools, and memory. **LangChain4j** brings this powerful framework to the Java ecosystem, allowing Java developers to leverage LLMs in their own applications without switching languages.

Langchain4j is particularly significant in enterprise settings, where Java remains a dominant technology. By offering a familiar, object-oriented API, Langchain4j simplifies interaction with LLMs like OpenAI's GPT, Ollama, and others. It also supports modular development of agents, tools, memory, prompt templates, and document retrieval systems.

## Architecture and Design

LangChain4j is built with a modular architecture that mirrors the structure of the original LangChain framework, bringing powerful LLM tooling into the Java ecosystem. Its design is centered on flexibility, composability, and clean abstraction. Each module, ranging from LLM integration to memory, agents, and retrievers is designed to operate independently or in coordination with others, making it easy to build complex LLM applications with minimal code duplication. This modularity is especially beneficial for enterprise developers accustomed to frameworks like Spring, as LangChain4j uses common Java patterns such as dependency injection and fluent builders to create and connect components.

At the heart of Langchain4j are several key building blocks: **LLM interfaces**, **prompts**, **memory**, **chains**, **agents**, **tools**, and **retrievers**. Each plays a distinct role in orchestrating how a language model interacts with users, tools, and external knowledge.

The **LLM interfaces** serve as unified entry points to various language model providers. These interfaces abstract away differences between models like OpenAI's GPT-4, Hugging Face-hosted models or locally deployed models via Ollama. This allows developers to switch providers or support multiple backends with minimal changes to their codebase.

**Prompts** are central to directing the behaviour of language models. Rather than hard-coding query strings, Langchain4j uses prompt templates, which can include placeholders for dynamic variables. This approach promotes code reuse, reduces duplication, and simplifies the process of tuning prompts during development. Templates help ensure consistent interactions, especially in production-grade applications where precision is vital.

**Memory** enables applications to maintain state across multiple turns in a conversation. In stateless systems, each LLM query is handled in isolation, but memory allows for continuity, where past interactions can inform future responses. Langchain4j supports both short-term memory and long-term memory, enabling applications like chatbots, personal assistants, or report generators to maintain a coherent dialog history.

**Chains** are sequences of operations that combine language models, prompts, and memory into cohesive workflows. For example, a chain might retrieve documents, summarize them with an LLM, and then pass the summary to another chain for classification. By linking components together, developers can construct multi-step pipelines.

**Agents** in Langchain4j introduce a layer of autonomous decision-making. Rather than following a predefined sequence, agents interpret user input and dynamically choose which tools or actions to invoke, based on the model’s reasoning. This approach enables more intelligent and interactive applications, such as digital assistants that can book appointments, fetch real-time data, or analyze documents.

**Tools** are user-defined functions that an agent can call to perform specific actions. These might include web search, API calls, database lookups, or even code execution. Tools are registered with descriptive metadata so that the LLM can decide when and how to use them.

Finally, **retrievers** are components responsible for fetching relevant information from external sources which are often unstructured text documents.

Together, these components form the backbone of Langchain4j. By integrating them in a modular and reusable way, developers can build sophisticated LLM applications in Java while maintaining clean code and scalability.

## LLM Integration in Java

LangChain4j simplifies the process of integrating large language models into Java applications by providing client abstractions for a variety of LLM providers. These clients wrap the necessary HTTP calls, API keys, and request/response formatting behind easy-to-use interfaces, enabling developers to focus on functionality rather than connectivity.

One of the most commonly used integrations is the **OpenAIClient**, which allows applications to interact with OpenAI’s suite of models, including ChatGPT, GPT-4, and GPT-3.5. By specifying the model name and API key, developers can quickly begin sending prompts and receiving completions from OpenAI’s cloud-based infrastructure.

For developers seeking full control the best option is **OllamaClient** which offers a bridge to local models like Mistral and LLaMA or other models running through Ollama. This is particularly useful for teams working with sensitive data or constrained by internet access, as it allows for deploying and querying models entirely on local hardware.

Langchain4j also includes support for hosted models on platforms like Hugging Face through the **HuggingFaceClient**. This client enables seamless access to thousands of open-source models in various domains, including natural language understanding, classification, translation, and more. Developers can specify which hosted model to use by its repository ID and authenticate using Hugging Face tokens.

In addition, there is a **LocalModelClient**, designed for situations where developers expose LLMs through their own RESTful APIs. This flexible option allows integration with custom deployments.

A key strength of Langchain4j is that switching between these providers requires minimal code changes. Since each client adheres to the same interface, developers can inject different LLM implementations depending on the environment, use case, or performance constraints. This pluggable design makes Langchain4j adaptable to both experimentation and production.

## Prompt Engineering and Templates

One of the most defining features in LangChain4j is its support of **prompt templates,** which enable Java developers to construct flexible and dynamic prompts for large language models. Rather than hardcoding static text, prompt templates allow you to inject variables and structure into the prompt, improving both **reusability** and **readability** while reducing errors during development.

**Prompt engineering** is a fundamental technique when working with LLMs because the phrasing, structure, and context of a prompt significantly influence the model's output. Langchain4j embraces this concept by allowing developers to define prompts as templates with placeholders, which are filled at runtime. This approach simplifies experimentation and tuning, making it easier to iterate on prompt wording and context without rewriting large blocks of code.

### Basic Example

Here is a simple example of using a PromptTemplate in Langchain4j:

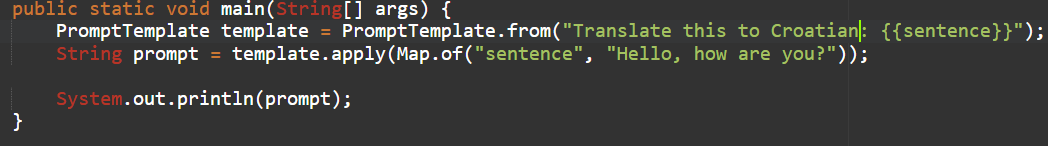


Figure 3.1 Prompt template example

In this case, {{sentence}} is a placeholder that is dynamically replaced with the provided input. This template can then be passed to an LLM for translation. Reusing the same structure with different sentences is straightforward, making this an ideal approach for localization tasks.

### Multi Variable Templates

Langchain4j supports templates with multiple variables. For example:

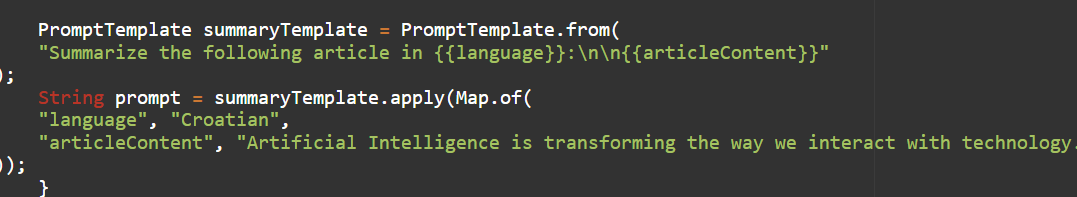


Figure 3.2 Multi variable template example

Output would be:

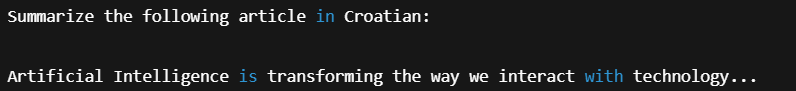


Figure 3.3 Output of the example above

This flexibility is especially useful for **multilingual applications** (e.g., translation, localization tools) and **parameter-based generation** (e.g., changing tone, length).

### Use with Chains

A **Chain** in Langchain4j is a reusable flow made up of several components:

* Prompt templates - building model input
* LLM clients – generate responses
* Memory – retain context
* Tools or retrievers – fetch information

Langchain4j chains are analogous to pipelines where each element transforms the input/output or enriches it. Basic example:

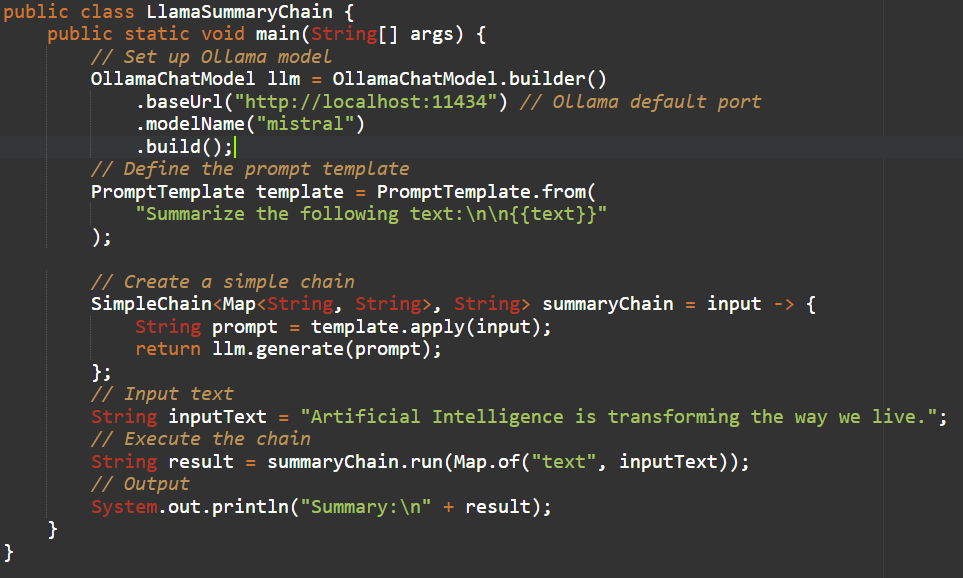


Figure 3.4 Chain example

By separating **template logic** from **model invocation**, we get easier unit testing, reusability across different chain stages, swappable components.

### Conditional and Dynamic Prompt Selection

While Langchain4j templates themselves are static, it is possible to **dynamically select** and **compose** them based on logic in your Java code. Basic example:

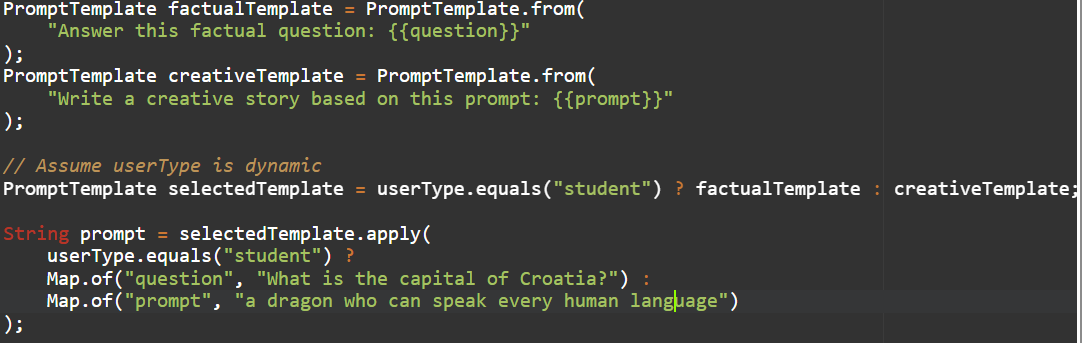


Figure 3.5 Dynamic prompt selection example

This approach is commonly used in role-based applications, content filtering and user personalization.

## Memory and Stateful Chat in LangChain4j

Langchain4j enables **stateful interactions** by integrating memory components that help track past messages, enabling coherent multi-turn conversations. This is crucial for building **AI assistants, helpdesk bots**, or **domain-specific chat agents** that need to remember user context.

There are two types of memory: short-term memory and long-term memory.

### Short-Term Memory (In-Memory)

It is suitable for per-session conversations. It lives only during the lifecycle of the application or session and it is fast and easy to use. Basic example:

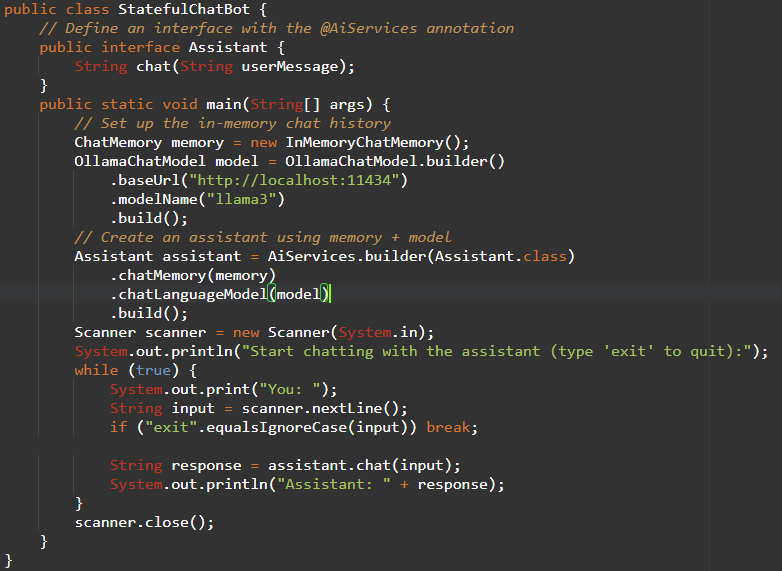


Figure 3.6 Short-term memory example

In the example above, the InMemoryChatMemory stores all previous user and assistant messages. Langchain4j automatically includes past messages in each prompt so that the LLM can generate context-aware responses. The assistant is built from an annotated interface (Assistant) using the AiServices factory.

### Long-Term Memory

Long-term memory in Langchain4j plays a critical role in creating intelligent, stateful applications that can retain knowledge across sessions and adapt to user behaviour over time. Unlike short-term memory, which is typically confined to a single interaction or session, long-term memory persists data in external systems such as databases, file systems, or vector stores. This allows applications powered by LLMs to recall relevant information from previous interactions, remember user preferences, or retrieve facts from a document corpus which makes them far more useful, especially in real-world scenarios.

Langchain4j provides a flexible interface for implementing long-term memory. Developers can create custom memory modules by persisting user messages and assistant responses in relational databases like PostgreSQL, NoSQL stores, or even flat files. This makes it possible for an LLM-powered application to "remember" user’s past inputs and responses.

Another powerful use of long-term memory is in the form of vector-based retrieval systems. Langchain4j supports integration with vector databases such as Chroma, Weaviate, Pinecone, or Qdrant, which store embedded document representations in a high-dimensional space. Developers can preprocess and index domain-specific documents into these vector stores by converting their content into numerical embeddings using models like Mistral or LLaMA via Ollama.

This architecture is often used in **Retrieval-Augmented Generation (RAG**) pipelines, where the retrieval mechanism provides context to the language model, dramatically improving reliability and reducing hallucinations.

## Agents and Tool Use

LangChain4j’s agent system is one of its most powerful features, enabling LLMs to interact not just with text, but also with external tools and systems (dynamically and autonomously). Agents act as intelligent intermediaries that use LLMs for reasoning and decision-making, and then select appropriate tools to accomplish a given task. This allows developers to build AI systems that go far beyond passive text generation and become truly interactive components of larger software ecosystems.

In the center of this system are Tool interfaces. A Tool in LangChain4j is a callable unit, method or a function that performs a defined action. It can represent an API call (e.g., weather, stock prices), a database query or any custom computation. Tools are created using a builder pattern, which makes them concise, testable, and reusable. Here's an example:

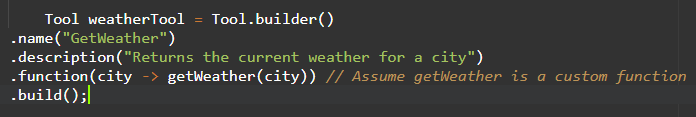


Figure 3.7 Tool example

In this snippet, getWeather(city) could be a method that connects to a weather API and returns a human-readable weather report.

### How Agents Work

Agents in Langchain4j combine a reasoning LLM (such as Mistral or LLaMA via Ollama) with a list of tools. When a user provides a prompt like "What’s the weather in Split today, and what should I wear?"*,* the agent can:

* Analyse the question using an LLM.
* Decide that it needs real-time weather information.
* Choose the GetWeather tool and call it with the parameter "Split".
* Retrieve the result and compose a full answer.

This process can also be extended to multiple tools. For example, an agent could combine a weather tool, a calendar lookup tool, and a recommendation tool to plan an outdoor activity.

### Tool Chaining and Decision-Making

LangChain4j agents can also engage in multi-step reasoning, where they chain together multiple tools. For instance, an LLM might first decide to get the weather, then based on that result, call another tool to check for local events that match the weather conditions.

Agents rely on a prompt-driven internal reasoning loop, where they keep evaluating the next best action until the task is complete. This is all abstracted nicely by Langchain4j, but developers can fine-tune the decision policy, stop conditions, and tool availability.

### Practical Applications

The agent and tool architecture is especially valuable in real-world use cases, such as:

* **Customer Support Assistants** - Agents can pull customer records, track shipments, and create support tickets via APIs.
* **Chatbots** - They can read knowledge bases, query CRMs, and send internal notifications.
* **Scientific Assistants** - Capable of running simulations, fetching data from APIs like PubMed, or analysing datasets.
* **Coding Assistants** - Agents can fetch documentation, compile code, and test snippets dynamically.

## Document Retrieval and RAG (Retrieval-Augmented Generation)

One of the most transformative advancements in enhancing the factual accuracy of LLMs is **Retrieval-Augmented Generation (RAG)**. Traditional LLMs, while powerful, are limited by the static nature of their training data and cannot access up-to-date or proprietary information. This is where Langchain4j shines by enabling developers to integrate RAG workflows within Java applications, thus combining the reasoning power of LLMs with the specificity of external knowledge sources.

At a high level, RAG works by retrieving relevant documents based on a user’s query and injecting this content into the prompt sent to the language model. This allows the model to "know" about documents it has never seen during training, leading to more accurate, grounded, and explainable outputs.

### Architecture and Workflow

Langchain4j supports a modular pipeline for implementing RAG, involving the following steps:

1. **Document Loading** - Load textual content from various file formats such as PDF, Markdown, or plain text. LangChain4j provides loaders to handle parsing and chunking of documents into manageable sections.
2. **Embedding** - Each document chunk is converted into a high-dimensional vector using an **embedding model**. Langchain4j supports multiple embedding providers including: Local models, OpenAI embeddings and Cohere embeddings.
3. **Vector Storage** - The vectors representing document chunks are stored in a **vector database**, allowing for fast similarity searches. Langchain4j supports integration with Pinecone, Redis and Qdrant.
4. **Query-Time Retrieval** - When the user asks a question, their query is embedded into the same vector space. The system then retrieves the top-k most similar documents using cosine similarity or other distance metrics.
5. **Augmented Prompt Construction** - The retrieved chunks are inserted into a prompt template and passed to the LLM for final generation.

This pipeline ensures that the model’s answer is not only coherent but also grounded in factual evidencefrom the retrieved documents.

### Advantages of Using Langchain4j for Retrieval-Augmented Generation (RAG)

LangChain4j offers significant advantages, particularly for developers and enterprises that are heavily invested in the Java ecosystem. By providing native integration with Java, it enables organizations to build advanced **Retrieval-Augmented Generation (RAG)** systems without needing to adopt new programming languages, re-architect existing systems, or rely on loosely-coupled microservices written in Python. This seamless compatibility not only reduces development overhead but also aligns with the best practices of Java-based enterprise software development, such as strong typing, robust build tools, and predictable deployment pipelines.

Another major strength of LangChain4j lies in its flexibility and modularity. The framework has been designed with a plug-and-play architecture that supports multiple vector databases (such as Weaviate, Qdrant, and Pinecone) and embedding model providers (including Hugging Face, OpenAI, and local transformers). This allows developers to tailor their technology stack according to the needs of their specific domain, whether they are working on high-throughput applications, latency-sensitive environments, or specialized data domains like law, medicine, or finance. As a result, LangChain4j is suitable for both prototyping and production deployment across a variety of industries.

In terms of document handling, LangChain4j includes fine-tuned and efficient document processing pipelines. These pipelines intelligently chunk documents based on semantic or structural patterns, filter irrelevant content, and prepare the data for indexing and retrieval. Effective chunking plays a critical role in the performance of RAG systems, as it directly affects how well the model can retrieve relevant context during inference. LangChain4j’s built-in optimizations help ensure that only the most meaningful pieces of information are retrieved, thereby improving the quality of the generated responses and the overall user experience.

Furthermore, data security and privacy are core design considerations within LangChain4j. The framework supports running vector stores and large language models locally or within private infrastructure, which is a significant benefit for enterprises concerned about data sovereignty, compliance, or handling proprietary datasets. In regulated sectors such as healthcare, finance, or government, where cloud-based solutions may pose legal or operational risks, the ability to operate LLMs and indexes fully on-premises makes LangChain4j a particularly attractive choice. This enables organizations to harness the power of generative AI while maintaining full control over their sensitive information.

In summary, LangChain4j's native Java integration, extensibility, optimized data pipelines, and strong emphasis on privacy make it a highly effective and practical solution for building intelligent, context-aware AI applications within modern Java ecosystems.

### Future Enhancements: Hybrid Search and Re-ranking

One of the most anticipated future enhancements for Langchain4j is the introduction of hybrid search capabilities. Pure vector similarity search methods, while powerful, can sometimes yield results that are semantically related but not directly relevant to the user’s query. Hybrid search addresses these limitations by combining keyword-based filtering with vector-based similarity search. This means that Langchain4j will enable developers to first filter documents using exact keyword matches or metadata constraints. After this initial filtering, a vector similarity search will be applied to the smaller, more focused subset of documents to rank them according to semantic relevance. This combination improves both precision and recall, allowing for more accurate and context-aware search results. It also supports more complex queries where both keyword presence and semantic meaning matter.

Another significant planned enhancement is the integration of re-ranking models. Initial document retrieval often relies on approximate nearest neighbour searches or keyword matches, which can result in a rough ordering of results. While this can be effective, the top-ranked documents may not always be the most relevant or helpful in addressing the query’s intent. LangChain4j plans to incorporate smaller transformer models dedicated to re-ranking the initially retrieved documents. These models will assess each document’s relevance more deeply by analysing the semantic fit between the query and the document content. The results will be reordered to prioritize those documents that are most likely to provide useful information.

Together, these enhancements will make LangChain4j-powered RAG pipelines more precise, context-aware, and suitable for enterprise applications. By combining exact keyword filtering with semantic search and refining results through re-ranking, LangChain4j will enable the development of advanced search and question-answering systems that deliver high relevance and accuracy while maintaining flexibility and security.

## LangChain4j Use Cases

LangChain4j enables a wide range of innovative applications within the Java ecosystem by providing robust tools for retrieval-augmented generation and semantic search. One prominent use case is the development of AI chatbots that possess memory and context awareness. These chatbots can maintain a conversation thread, remember past interactions, and provide responses that take into account the entire dialogue history, resulting in more natural and useful user experiences.

Langchain4j is also well-suited for creating intelligent code assistants integrated into developer tools. These assistants can analyse Java methods or code snippets and provide explanations, suggestions, or improvements, helping developers to write better code and understand complex codebases faster. This use case enhances developer productivity by embedding AI capabilities directly into their workflows.

Additionally, LangChain4j supports the creation of AI-powered report generators that synthesize information from multiple sources such as Git commits, project wikis, and databases. This allows teams to automatically generate comprehensive and up-to-date reports without manual effort, streamlining documentation and project tracking.

Beyond text-based applications, LangChain4j can serve as the foundation for multi-modal assistants when combined with external APIs that handle image, audio, or other data types. Such assistants can process and understand information across different modalities, enabling richer interactions and broader functionality, such as voice-controlled search or image-aware question answering.

Overall, LangChain4j’s flexibility and deep integration with the Java platform make it a powerful tool to build intelligent applications that leverage retrieval-augmented generation and semantic understanding across diverse domains.

# LangChain4j Versus Other Frameworks

Langchain4j is a relatively new but rapidly growing framework designed specifically for Java developers who want to leverage Retrieval-Augmented Generation (RAG) and build complex AI applications using LLMs. To understand its unique position, it is useful to compare it with other popular frameworks that serve similar purposes in different programming environments, notably LangChain for Python, Haystack, and OpenAI’s official SDKs.

## LangChain (Python)

LangChain is the original and most mature framework for constructing sophisticated pipelines that combine large language models (LLMs) with external data sources, tools, APIs, and memory components. Written entirely in Python, LangChain is tightly integrated with Python’s vibrant machine learning ecosystem, which includes libraries like Hugging Face Transformers, PyTorch, TensorFlow, and Scikit-learn. This deep integration gives LangChain immediate access to the latest advancements in LLMs, embeddings, and vector search infrastructure, making it a top choice for developers building cutting-edge AI applications.

In contrast, LangChain4j targets the Java ecosystem, which traditionally dominates enterprise backend development, large-scale applications, and systems that demand robustness, type safety, and long-term maintainability. Although Langchain4j is still evolving, it enables Java developers to build RAG solutions without switching to Python or relying on cross-language integrations. However, the Java ecosystem currently has fewer pretrained models and AI research tools directly available compared to Python, which can limit some experimentation.

## Haystack (Python)

Haystack is another powerful and mature Python framework, specifically designed for building semantic search, question-answering systems, and knowledge retrieval pipelines. Developed by deepset, Haystack has established itself as a go-to tool for natural language processing tasks that involve extracting answers from large collections of documents. It features robust support for dense vector-based retrieval using models like Sentence Transformers, as well as sparse retrieval using traditional methods like BM25, allowing developers to experiment with hybrid approaches for optimal performance. Additionally, Haystack integrates seamlessly with a wide variety of vector databases and provides built-in components for preprocessing, indexing, and querying documents, which significantly speeds up development time.

While Haystack is more specialized for search and QA tasks, LangChain4j aims to provide a more general-purpose chaining framework with flexibility for a wide range of LLM-powered applications beyond just search, such as chatbots or report generation. For Java developers, LangChain4j represents a direct way to build such pipelines without bridging to Python, though Haystack may have more mature tooling and community support in the document search niche.

## OpenAI SDKs

OpenAI provides official SDKs for various languages, including Python and JavaScript, enabling straightforward access to OpenAI’s LLM APIs. These SDKs are typically lightweight and focus on basic API interactions, such as sending prompts and receiving completions but they lack the high-level orchestration, document retrieval, memory handling, and vector search features that are critical for building Retrieval-Augmented Generation (RAG) systems or multi-step AI workflows. Developers using these SDKs often need to build significant infrastructure themselves to handle tasks like embedding generation, storage, and context-aware querying.

LangChain4j fills this gap by offering a robust Java-native framework that abstracts and automates many of these components. It provides tools for chaining model calls, processing and embedding documents, managing conversation memory, and integrating with popular vector databases for similarity search. This cohesive structure is especially valuable in enterprise Java ecosystems where modularity, maintainability, and data governance are crucial. By staying entirely within the Java stack, LangChain4j also reduces complexity, enhances performance, and supports tighter integration with existing authentication, logging, and monitoring solutions.

## Strengths of LangChain4j in the Java Ecosystem

LangChain4j offers significant benefits for Java developers by integrating AI capabilities directly into the Java ecosystem. Its design leverages the strengths of Java’s mature development environment and enterprise focus, enabling organizations to build reliable, maintainable, and secure AI-powered applications without needing to bridge to other programming languages. Key advantages of LangChain4j in the Java ecosystem include:

* **Seamless Java integration -** LangChain4j fits naturally into Java applications, taking full advantage of Java’s strong static typing, robust build tools, and widespread use in enterprise environments. This integration eliminates the need for complex setups involving Python microservices or cross-language communication. This seamless integration allows developers to incorporate advanced AI features like contextual awareness, document retrieval, and intelligent response generation directly into Java applications. Moreover, LangChain4j leverages Java’s strong static typing, Maven/Gradle build systems, and widespread deployment practices (e.g., Spring Boot, Docker) to ensure that the AI components conform to the same standards and deployment pipelines as the rest of the application.
* **Robustness and maintainability** - With Java’s comprehensive tooling and static type system, developers can create AI pipelines that are easier to maintain, debug, and scale. This robustness is crucial for large-scale production systems where reliability is paramount. Features such as compile-time checks, IDE support for refactoring, unit testing frameworks, and static code analysis tools help ensure long-term maintainability and reduce technical debt. Furthermore, LangChain4j aligns with standard Java architectural patterns, allowing teams to integrate AI components into existing CI/CD pipelines, apply consistent logging and monitoring practices, and ensure smooth team collaboration across large codebases.
* **Security and Data Privacy -** Many enterprises operate in regulated industries or handle sensitive information that must remain on-premises to comply with legal, contractual, or internal security policies. LangChain4j addresses these concerns by supporting local vector databases and on-premise deployment LLMs, eliminating the need to send data to third-party cloud providers. This architecture ensures that proprietary documents, customer records, or internal reports are processed and queried entirely within the organization’s infrastructure. Additionally, developers can implement access controls, auditing mechanisms, and encryption using familiar Java security frameworks, making it easier to meet compliance requirements such as GDPR or ISO standards while building AI-driven applications.

# LangChain4j Integration with Ollama

LangChain4j offers seamless integration with a variety of local and remote language models, one of the most notable being **Ollama**, a platform designed to run LLMs locally with ease. This integration allows Java developers to incorporate powerful open-source language models directly into their applications without relying on external APIs or cloud-based services. By combining LangChain4j’s modular LLM orchestration with Ollama’s local model execution, developers gain enhanced control over data privacy, latency, and customization, which is especially critical in enterprise settings.

## What is Ollama?

Ollama is a lightweight, developer-friendly runtime designed to run large language models like LLaMA, Mistral, and other open-source models locally on a user’s machine. It abstracts away the complexity of GPU handling and model loading, making it easy for developers to interact with LLMs using simple commands and APIs. Its focus is on providing fast, private, and customizable AI infrastructure without the need for cloud services.

## Benefits of Using Ollama with LangChain4j

Integrating Ollama with LangChain4j offers several distinct advantages:

* **Local Inference** - All LLM inference happens locally, ensuring that no data leaves the user’s environment
* **Low Latency** - Running models locally reduces response time by eliminating network delays associated with cloud APIs.
* **Cost Efficiency** - Developers avoid API usage fees and cloud costs by relying on local hardware.
* **Custom Model Support**: Ollama allows loading fine-tuned or domain-specific models, giving teams flexibility for niche applications.

## Connecting LangChain4j with Ollama

LangChain4j provides built-in support for connecting to Ollama's local server. This is done via the **OllamaLanguageModel** class, which communicates with Ollama’s REST API under the hood. Developers can instantiate this model by simply providing the base URL and model name.

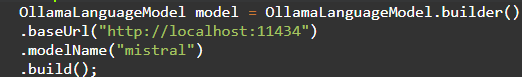


Figure 5.1 Ollama model builder

This integration allows the use of local models within LangChain4j chains and agents, enabling full offline RAG and chatbot workflows.

## Use Cases for LangChain4j + Ollama

The combination of LangChain4j and Ollama creates a powerful toolkit for building secure, efficient, and highly customizable AI-powered applications in Java environments. Because both technologies support on-premise execution, they are especially suited for scenarios where privacy, offline access, or control over the infrastructure is crucial. Below are key use cases where this integration is particularly effective:

* **Enterprise chatbots with strict privacy requirements -** Many industries, such as finance, healthcare, and government handle sensitive customer or internal data that cannot be exposed to third-party APIs or public cloud services. Using LangChain4j in conjunction with Ollama, organizations can build chatbots that understand context, retain memory, and deliver intelligent responses, all while ensuring that data never leaves their internal systems.
* **On-device assistants for field work -** In industrial, medical, or remote field operations, devices often need to operate in environments with limited or no internet connectivity. On-device assistants powered by Ollama can run locally on rugged laptops or edge devices, providing real-time support, documentation lookup, or anomaly detection. LangChain4j provides the tools to integrate such AI components into enterprise-grade Java software already in use in these domains.
* **Secure report generators** - Organizations often need to synthesize information from internal sources such as Git repositories, databases or internal files into coherent reports. When this data is confidential, using local models with LangChain4j and Ollama ensures that no proprietary data is leaked during generation.
* **Academic and research tools for local data analysis** - Universities and research institutions often work with large, locally stored datasets that cannot or should not be uploaded to cloud-based LLM providers. With LangChain4j and Ollama, researchers can build Java applications for literature review, semantic search, summarization, and natural language querying of local datasets. This is especially useful in scientific fields where the analysis must remain reproducible, confidential, or fully offline.

# Demo Project: Gitlab Student Practice Report Generation

The goal of the demo project was to make an app that generates student reports based on Gitlab activity. It fetches all relevant data from Gitlab API and stores it in PostgreSQL database. That data is then used to make a practice report for each student. For text generation we have used Llama3 model because it is reasonably fast and it has great text generation possibilities.

## Fetching data from Gitlab and storing it into a database

Firstly, a user has to make a Gitlab **PAT (Personal Access Token)** and give it full access in order to be able to use the Gitlab REST API. Gitlab API gives the option to fetch all commits, branches, issues and wiki pages from the given URL.

The API returns structured JSON responses, which are then parsed into corresponding Java objects using Jackson Object Mapper. These objects are mapped to internal model classes.



Figure 6.1 Branch model class

Data persistence is handled via a dedicated **DAO (Data Access Object)** layer that encapsulates all interaction with the **PostgreSQL** database. This design ensures modularity and separation of concerns, making the data layer reusable and testable.

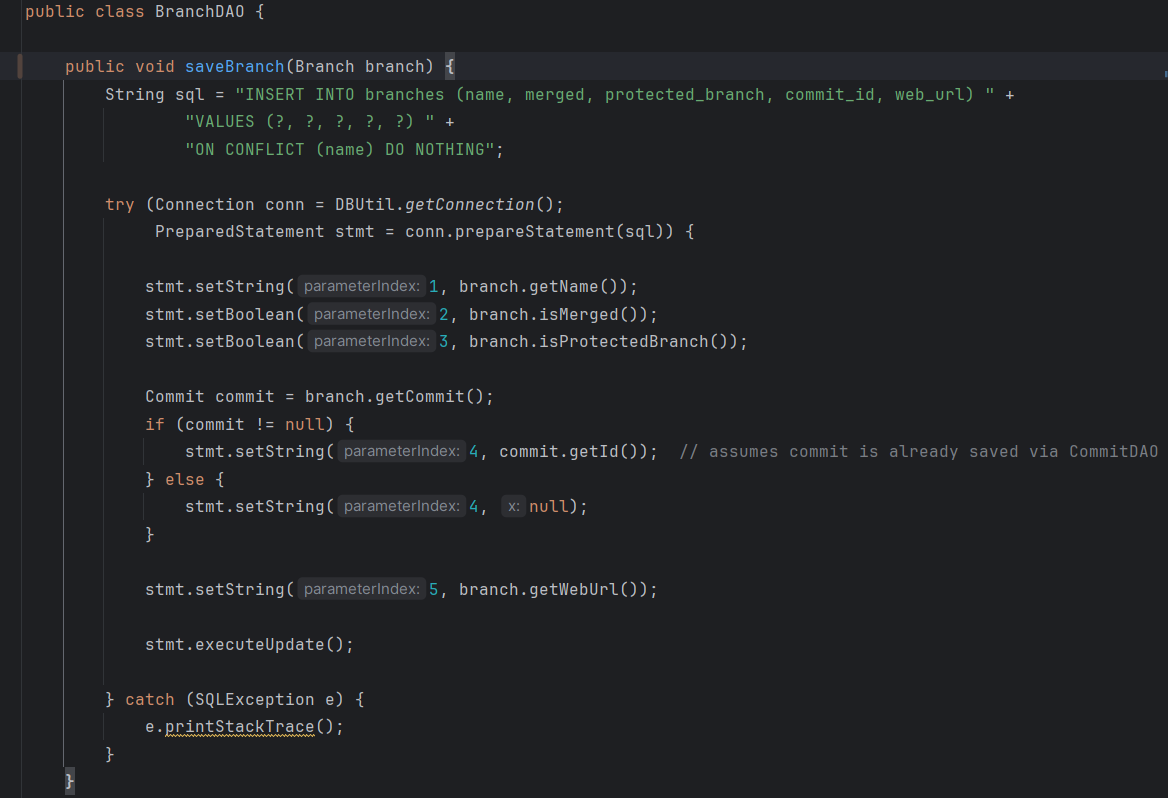


Figure 6.2 BranchDAO class

Each time the applications is launched, an update function is performed, so we always have up to date Gitlab data stored in the database.

## Student selection

To enable report generation for individual students, the application includes functionality to fetch all users from the database via GET request to api/users endpoint and present them in the user interface through a dropdown menu. The user can select one or multiple students or use a select all button.

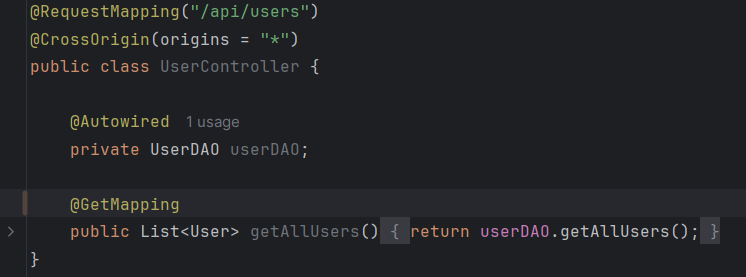


Figure 6.3 api/users endpoint

Once a student is selected, their unique GitLab username is passed to the api/reports endpoint, which uses it to filter the relevant commits, issues, or contributions from the database. This filtered data forms the basis for generating a personalized report, ensuring that each student’s contributions can be analysed individually and fairly.

## Selecting relevant data from the database

Once the user selects one or more students from a dropdown list in the frontend, the application sends a list of usernames (e.g., ["ivan.ivic", "ana.anic"]) to the backend.

On the backend, this list is used to fetch all relevant data from the PostgreSQL database for each selected user. This includes commits, issues (both open and closed), associated branches and wiki pages.

For each username, the backend uses the DAO layer to perform SQL queries on the relevant tables and extract the necessary records. These records are then used to dynamically construct a **custom prompt**, which is passed to the LLM for generating a tailored report. This allows the system to generate highly specific outputs that reflect the user's individual contributions and activity within the GitLab project.

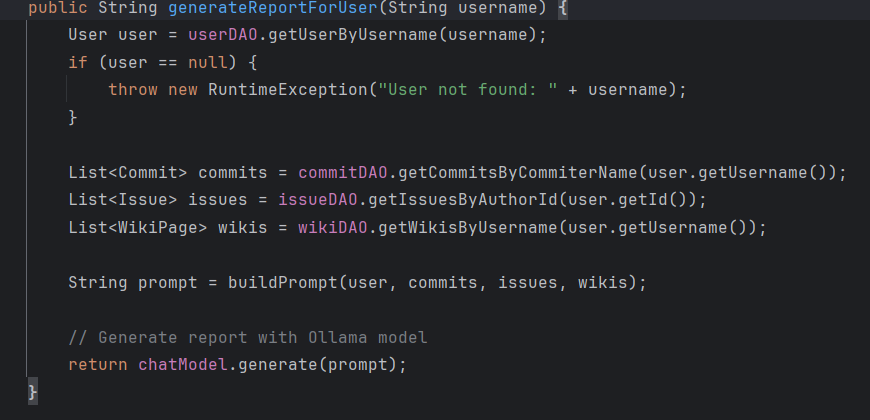


Figure 6.4 Generate report function

### Problem with wiki pages fetching

One of the challenges encountered during data fetching from the database involves the **wikis table**, which does **not contain a direct reference to user ID or username**. Unlike commits or issues, where author information is explicitly available, wiki pages fetched from GitLab do not include metadata linking them to a specific user.

To work around this limitation, we implemented a strategy based on **slug matching**. Specifically, we convert the selected username to capitalized format and then perform a SQL LIKE query against slug field in the wikis table. This approach assumes that wiki page slugs follow a naming convention similar to the username.

## Report generation

For report generation the application is using **LangChain4**j framework and **Ollama**. LangChain4j is a powerful framework that allows us to connect Java applications with LLMs. Ollama gives us the option to run LLMs locally without need for any external services like OpenAI. This greatly improves the security of the application.

In this setup, we utilize **Llama 3**, one of the latest state-of-the-art LLMs supported by Ollama. Llama 3 combines strong language understanding with efficient performance, making it ideal for generating high-quality textual reports.

The report generation process uses dynamic prompt creation with collected data. This dynamic prompt encapsulates the key details of a user’s project activity, ensuring that the LLM can generate relevant and context-aware summaries. Using Langchain4j, the system programmatically constructs these prompts, feeding them into Ollama’s Llama 3 model. The LLM then generates a coherent, well-structured report in English.

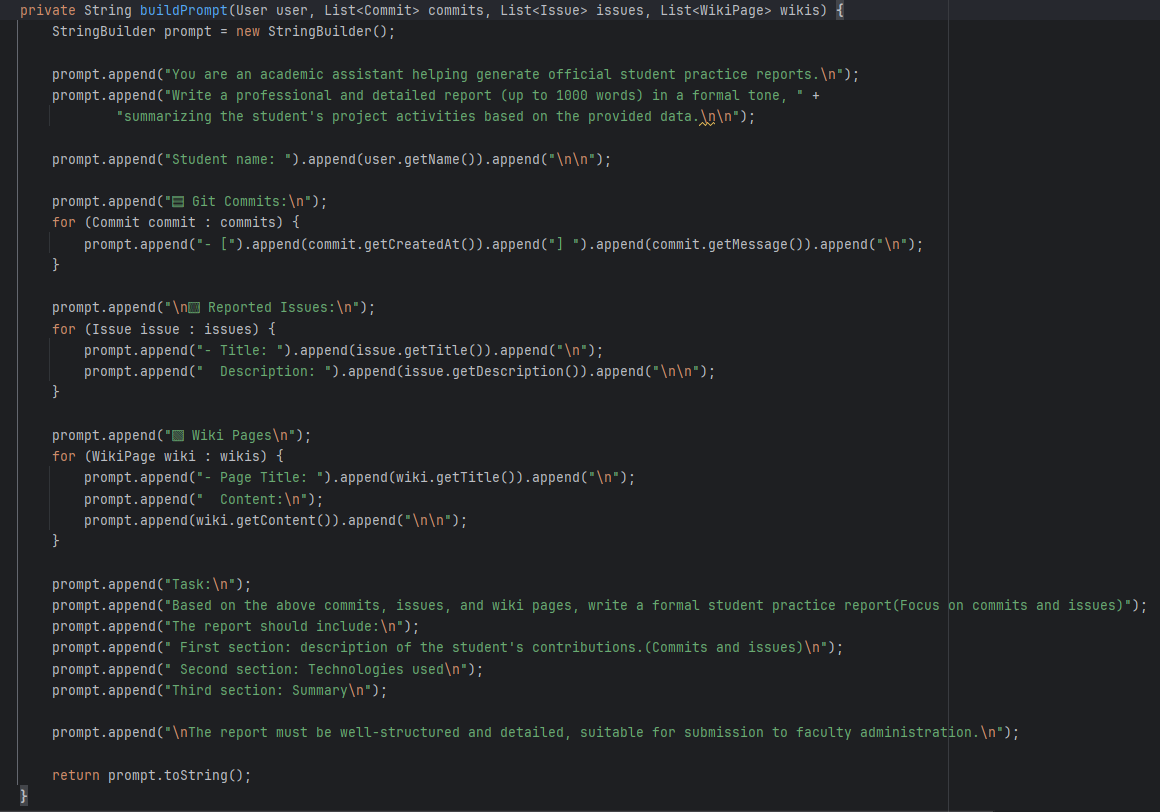


Figure 6.5 Build prompt function

## Report translation

The integration of the Google Translate API into the backend system provides seamless automatic translation capabilities, specifically supporting the Croatian language among many others. This allows generated reports, originally created in English, to be translated into Croatian automatically before being presented to the user. The reports needs to be in Croatian to be suitable for admission to the faculty.

The translation process occurs entirely on the backend to ensure a smooth and consistent user experience. Users accessing the system see only the Croatian version of the report, which removes any complexity or need for manual translation on their part.

Fallback mechanism is also present, if translation fails, the English version is displayed to the user. This ensures that users still receive the information they need, even if automatic translation is not possible at that moment. This approach balances user convenience with reliability by automating the translation process while maintaining continuity of service.

## Displaying report

The report section features a simple and user-friendly interface designed for easy navigation and readability. It supports displaying multiple reports simultaneously, allowing users to browse through various generated documents without confusion.

Each report is clearly labeled with a distinct title and accompanied by its content, providing quick identification and access. To enhance usability, the interface includes filtering options based on usernames, enabling users to quickly find reports associated with specific individuals.

This combination of clear presentation, multiple report handling, and efficient filtering creates an intuitive experience for users managing and reviewing reports.



Figure 6.6 Generated report example

# References

1. Brown, T. et al. (2020). Language Models are Few-Shot Learners. [arXiv:2005.14165](https://arxiv.org/abs/2005.14165)
2. Russell, S., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach(4th ed.). Pearson.
3. Langchain4j. (2024). Langchain4j Documentation. <https://www.langchain4j.dev>
4. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), <https://doi.org/10.1038/nature14539>
5. Vaswani, A. et al. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems. <https://arxiv.org/abs/1706.03762>
6. Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation. <https://doi.org/10.1162/neco.1997.9.8.1735>
7. Ollama. (2024). Ollama - Run Large Language Models Locally. <https://ollama.com/>
8. deepset. (2023). Haystack: Open Source Framework for NLP-powered Search & QA. <https://haystack.deepset.ai/>
9. OpenAI. (2024). OpenAI API Reference and SDKs. <https://platform.openai.com/docs/sdk>
10. Meta AI. (2023). LLaMA: Open and Efficient Foundation Language Models. <https://ai.facebook.com/blog/large-language-model-llama-meta-ai/>
11. Mistral AI. (2023). Mistral: Open-source Language Models. <https://www.mistral.ai/>