

Sync Lab S.r.l.

Progetto di stage

Andrea Cecchin



Attività di
prototipazione

Indice

1	Introduzione	2
1.1	Scopo del documento	2
1.2	Riferimenti informativi	2
2	Analisi delle tecnologie	4
3	Retrieval-Augmented Generation	8
3.1	Ingestion	8
3.2	Retrieval	9
3.3	Generation	10
4	Progettazione	12
5	Utilizzo	13
5.1	Lettura del database	13
5.2	Inserimento documento	14
5.3	Rimozione documento	14
5.4	Interrogazione chatbot	15
6	Modelli	16
6.1	Phi-3 mini	16
6.2	GPT 3.5 Turbo	16
6.3	Gemini 1.5 Pro	16
6.4	PaLM 2 Bison	17

Introduzione

1.1 Scopo del documento

Il seguente documento intende esporre le attività, svolte durante il periodo di stage relativo allo studio delle tecnologie per la Retrieval-Augmented Generation in locale, di realizzazione di un prototipo integrante le tecnologie per la RAG in un backend basato su Java Spring.

1.2 Riferimenti informativi

- Sito web di Ollama:
<https://ollama.com>
- Scheda tecnica di Phi-3 su Ollama:
<https://ollama.com/library/phi3>
- Paper ufficiale di Phi-3:
<https://arxiv.org/abs/2404.14219>
- Paper ufficiale di Gemini 1.5:
<https://arxiv.org/pdf/2403.05530>
- Paper ufficiale di PaLM 2:
<https://arxiv.org/pdf/2305.10403>
- Sito web di Open AI (modello GPT 3.5):
<https://platform.openai.com/docs/models/gpt-3-5-turbo>
- Scheda tecnica di All-MiniLM-L6-v2:
https://www.sbert.net/docs/pretrained_models.html
- Sito web di LangChain4J:
<https://docs.langchain4j.dev>
- Documentazione di LangChain4J:
<https://docs.langchain4j.dev/apidocs/index.html>

- Repository Github di LangChain4J:
<https://github.com/langchain4j/langchain4j>
- Sito web di Spring:
<https://spring.io>
- Sito web di Postman:
<https://www.postman.com>

Analisi delle tecnologie

Phi-3: Nella realizzazione del prodotto di questo protoripo, è stato necessario individuare un Large Language Model con cui interagire in locale. Per interfacciarsi ai modelli, è stato utilizzato il framework Ollama, che permette di accedere ai modelli scaricati in locale in modo semplice ed intuitivo.

In totale, sono stati testati 4 diversi Large Language Model: Llama2:7b (Meta), Phi-2.7b (Microsoft), Phi-3.8b (Microsoft) e Gemma:2b (Google).

Dopo una prima valutazione delle performance del modello nel computer utilizzato per i test, è stato individuato in Phi-3 il modello locale che unisce maggiormente leggerezza del modello con performance soddisfacenti.

LangChain4J: Per interagire con i modelli nelle task di Retrieval-Augmented Generation, è stata utilizzata la libreria LangChain4J.

Questa libreria è un progetto open-source portato avanti dalla community della nota libreria per l'AI LangChain, punto di riferimento principale per tutti gli interessati all'integrazione di modelli di linguaggio in applicazioni scritte in Python e JavaScript.

Una possibile alternativa, ma non scelta, è Spring AI, modulo del noto framework per la programmazione in Java. L'obiettivo del protoripo prevedeva l'integrazione di tecnologie RAG in un backend Spring, quindi ad un primo sguardo la scelta più ovvia sarebbe stata adottare questo modulo di Spring, senza ricorrere a un'altra libreria.

Tuttavia, LangChain4J è sembrata la scelta migliore per una serie di motivi:

- Spring AI è nata nel tentativo di emulare LangChain. LangChain4J, pur non essendo un progetto ufficiale di LangChain, condivide la stessa identica community, che con un progetto totalmente open-source sta portando tutte le medesime funzionalità della famosa libreria Python e JavaScript anche su Java.

Scegliere la prima significherebbe scartare la tecnologia che emula e che è alla sua base. È quindi più sensato optare per la seconda libreria, considerato anche la maggior popolarità e utilizzo rispetto Spring AI.

- Il numero di features è notevolmente maggiore nella libreria LanChain4J, la quale offre un maggior numero di integrazioni con le tecnologie necessarie alla RAG.

Possiamo vedere queste differenze nelle seguenti tabelle:

	LangChain4J	Spring AI
Amazon Bedrock	✓	✓
Anthropic	✓	✗
Azure OpenAI	✓	✓
ChatGLM	✓	✗
DashScope	✓	✗
Google Vertex AI Gemini	✓	✓
Google Vertex AI PaLM	✓	✓
HuggingFace	✓	✓
LocalAI	✓	✗
MistralAI	✓	✓
Ollama	✓	✓
OpenAI	✓	✓
HuggingFace	✓	✓
Qianfan	✓	✗
Zhipu AI	✓	✗

Tabella 1: Large Language Model supportati per l'integrazione

Fonti: <https://docs.langchain4j.dev/category/integrations>
<https://spring.io/projects/spring-ai>

	LangChain4J	Spring AI
Astra DB	✓	✗
Azure AI Serach	✓	✓
Azure CosmosDB	✓	✗
Cassandra	✓	✗

	LangChain4J	Spring AI
Chroma	✓	✓
Elasticsearch	✓	✗
Infinispan	✓	✗
Milvus	✓	✓
MongoDB	✓	✗
Neo4j	✓	✓
OpenSearch	✓	✗
PGVectore	✓	✓
Pinecone	✓	✓
Qdrant	✓	✓
Redis	✓	✓
Vearch	✓	✗
Vespa	✓	✗
Weaviate	✓	✓

Tabella 2: Vector Database supportati per l'integrazione

Fonti: <https://docs.langchain4j.dev/category/integrations>
<https://spring.io/projects/spring-ai>

	LangChain4J	Spring AI
Amazon Bedrock	✓	✓
Azure OpenAI	✓	✓
DashScope	✓	✗
Google Vertex AI Gemini	✓	✓
HuggingFace	✓	✗

	LangChain4J	Spring AI
LocalAI	✓	✗
MistralAI	✓	✓
Nomic	✓	✗
Ollama	✓	✓
ONNX	✓	✓
OpenAI	✓	✗
PostgresML	✗	✓
Qianfan	✓	✗
Zhipu AI	✓	✗

Tabella 3: Embedding Model supportati per l'integrazione

Fonti: <https://docs.langchain4j.dev/category/integrations>
<https://spring.io/projects/spring-ai>

All-MiniLM-L6-v2: È il modello utilizzato per la creazione degli embedding. La decisione di questo modello si basa sulla sue performance in relazione alle risorse che richiede: nonostante l'intero modello pesi solo 40MB, è in grado di elaborare grandi moli di vettori in pochi secondi, mostrando ottime performance nel riconoscimento degli embedding rilevanti per la RAG.

Gemini 1.5 Pro: È il modello utilizzato per la versione on-premise non locale del prototipo: il suo impiego era requisito obbligatorio.

Bison: È un ulteriore modello di Google utilizzato per la versione on-premise non locale del prototipo. Fa parte della famiglia di modelli PaLM 2.

GPT 3.5 Turbo: È un ulteriore modello utilizzato per la versione on-premise non locale del prototipo: il suo impiego era requisito desiderabile.

Spring: È il framework Java utilizzato per lo sviluppo del prototipo: il suo impiego era requisito obbligatorio.

Postman: È l'applicazione utilizzata per testare le API con cui utilizzare i servizi del prototipo, implementati secondo il Controller-Servcie-Repository pattern.

Retrieval-Augmented Generation

La funzionalità di Retrieval-Augmented Generation resa possibile dal prototipo è suddivisibile in tre sotto funzionalità: ingestion, retrieval e generation.

3.1 Ingestion

La fase di ingestion è relativa all’elaborazione delle fonti da cui ottenere le informazioni aggiuntive da fornire al modello nella generazione di risposte.

Una volta ottenuto un documento, il testo in esso contenuto viene estratto e successivamente suddiviso in paragrafi.

```
Document document = loadDocument(
    toPath('PATH_DEL_DOCUMENTO'),
    new ApachePdfBoxDocumentParser()
);

DocumentSplitter splitter = new DocumentByParagraphSplitter(
    700,
    100
);

List<TextSegment> segments = splitter.split(document);
```

Una volta ottenuti i segmenti del testo, essi vengono embeddizzati attraverso l’embedding model All-MiniLM-L6-v2, e successivamente memorizzati all’interno del vector store (in memory, consigliato dal team di LangChain4J per prototipi con un modesto numero di documenti da testare) in coppia con il testo originale.

```
EmbeddingModel embeddingModel = new AllMiniLmL6V2EmbeddingModel();

List<Embedding> embeddings =
    embeddingModel.embedAll(segments).content();

embeddingStore.addAll(embeddings, segments);
```

In questo modo, nella fase di retrieval delle informazioni utili, utilizzeremo gli embedding per trovare i pezzi di testo più rilevanti per rispondere alla domanda, e andremo a recuperare il testo relativo.



Figura 1: Schema illustrativo dell'ingestion

3.2 Retrieval

La fase di recupero delle informazioni comincia con l'invio al modello della domanda. La domanda viene embeddizzata con lo stesso modello utilizzato in precedenza per l'ingestion dei documenti, e viene utilizzato il vettore ottenuto per ricercare le informazioni in database con gli embedding più semanticamente simili alla domanda. Il calcolo della semantic similarity avviene tramite similarità del coseno, ovvero calcolando il coseno dell'angolo presente tra i vettori della domanda e del testo in esame.

In questa fase, dopo aver calcolato la similarità con tutti i vettori presenti nel vector store, sono individuati e recuperati i 5 segmenti di testo con gli embedding più simili alla domanda, purché essi abbiano un relevant score di almeno 0.7 (corrispondente ad una cosine similarity dello 0.5).

Questi saranno poi necessari per la fase di generazione della risposta.

```
Embedding questionEmbedding =
    embeddingModel.embed(question).content();
int maxResults = 5;
double minScore = 0.7;
List<EmbeddingMatch<TextSegment>> relevantEmbeddings =
    embeddingStore.findRelevant(
        embedding, maxResults, minScore
    );
```

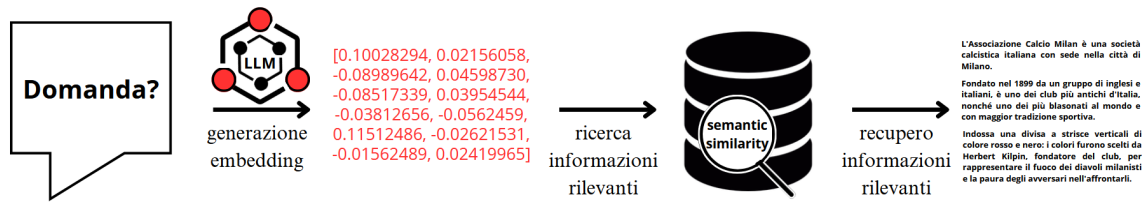


Figura 2: Schema illustrativo della retrieval

3.3 Generation

La fase di generation consiste nel sottoporre al modello la richiesta finale, contenente la domanda posta, le informazioni utili recuperate durante la retrieval e le indicazioni di come rispondere.

Tutto questo è possibile creando un apposito prompt, poi inviato al modello per la generazione della risposta.

```
PromptTemplate promptTemplate = PromptTemplate.from(
    "Answer to the following question, based ONLY
    on the context i'll give you.
    \n\n
    Question:---\n
    {{question}}\n---
    \n\n
    Context:---\n
    {{information}}\n--- end context ---
    \n\n
    If you have no useful information, answer with
    'I can't provide any answer.'. Don't use general
    knowledge to give info outside the context.\n");

String information = relevantEmbeddings.stream()
    .map(match -> match.embedded().text())
    .collect(joining("\n--\n"));

Map<String, Object> variables = new HashMap<>();
variables.put("question", question);
variables.put("information", information);

Prompt prompt = promptTemplate.apply(variables);
```

Come è possibile vedere dal frammento di codice, al nostro modello viene chiesto di rispondere solo quando riceve delle informazioni utili. Questo significa che se poniamo una domanda la cui risposta non è presente in alcun documento, non sarà identificato alcun segmento di testo con una semantic similarity sufficiente ad essere classificato come rilevante. In questo modo, nel nostro prompt inviato al modello il contesto sarà vuoto, spingendo il modello a non rispondere alla nostra domanda, anche se sa la risposta.

Ottenuto il prompt finale, non resta che passarlo al modello e richiedere la generazione della risposta.

```
// esempio di utilizzo del modello locale phi3 con ollama
ChatLanguageModel model = OllamaChatModel.builder()
    .baseUrl("http://localhost:11434")
    .modelName("phi3")
    .temperature(0.2)
    .timeout(Duration.ofMinutes(1))
    .build();

AiMessage answer =
    model.generate(prompt.toUserMessage()).content();
```

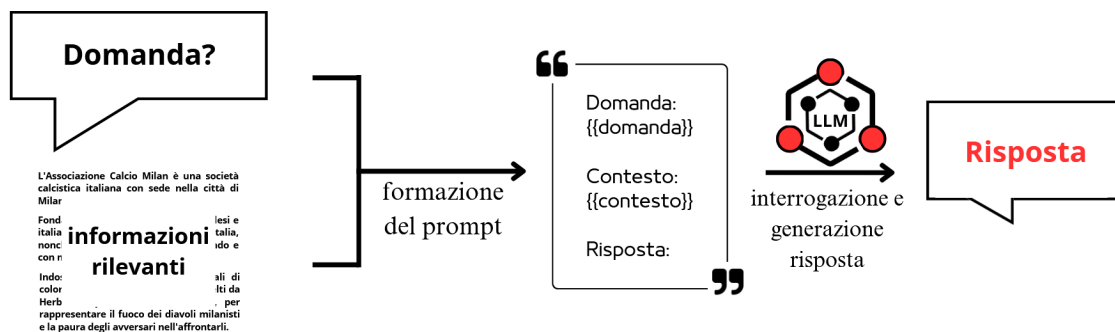


Figura 3: Schema illustrativo della generation

Progettazione

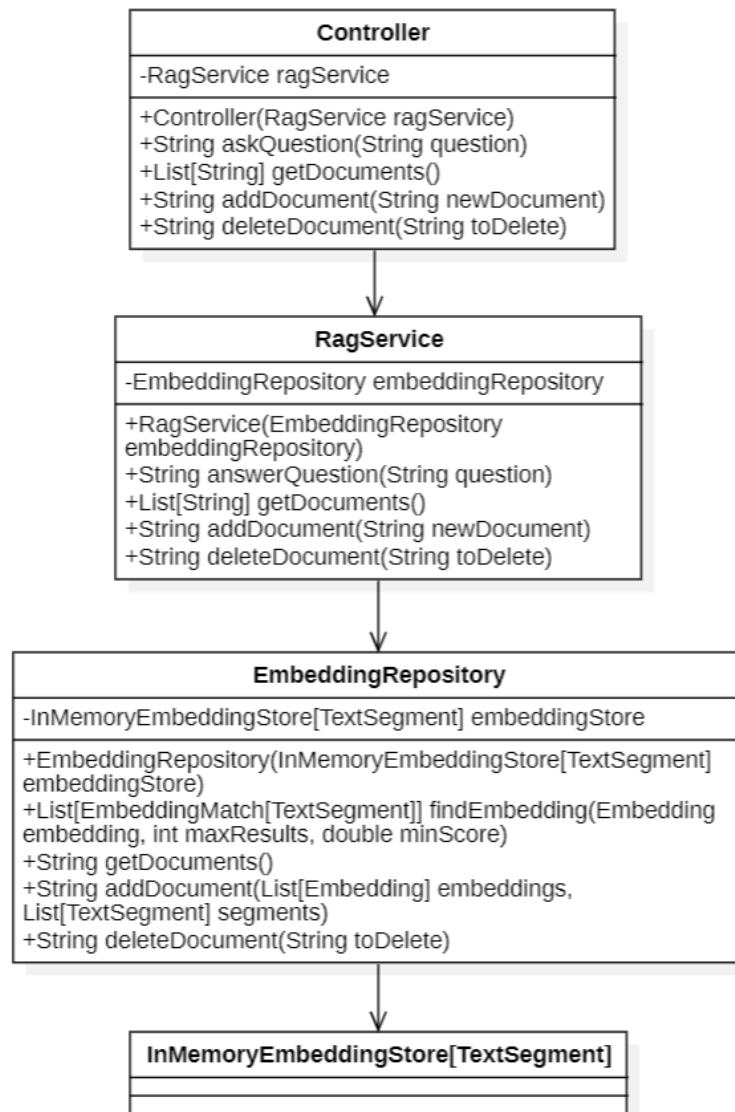


Figura 4: UML delle classi dei prototipi

Utilizzo

Per poter testare i prototipi, è necessario prima di tutto avviare l'applicazione Spring Boot dell'intero progetto.

Una volta avviata, sarà possibile testare tutte e quattro le funzionalità del prototipo, necessarie alla Retrieval-Augmented Generation: lettura del database (per conoscere quali documenti sono stati processati, con lo store degli embedding relativi), inserimento di un nuovo documento, eliminazione di un documento e interrogazione del modello.

Per tutti queste funzionalità, utilizzeremo Postman per testare le API e raggiungere i Controller REST dell'applicazione.

5.1 Lettura del database

Endpoint: localhost:8080/api/documents;

Metodo HTTP: GET;

Corpo della richiesta: vuoto;

Output atteso: Lista di stringhe dei nome dei documenti in database.

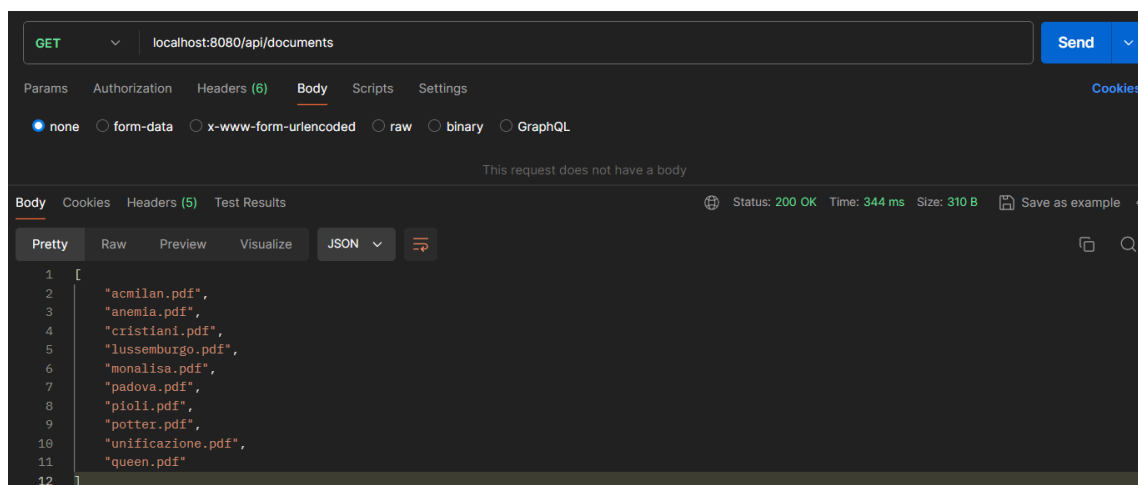


Figura 5: Esempio chiamata GET in /api/documents

5.2 Inserimento documento

Endpoint: localhost:8080/api/documents;

Metodo HTTP: POST;

Corpo della richiesta: Stringa del nome del documento da aggiungere, nel formato "nomedocumento.estensione";

Output atteso: Stringa di conferma inserimento;

NOTA BENE: In questo prototipo, per poter inserire un documento, ovvero effettuare su di esso l'ingestion, deve trovarsi all'interno del percorso "src/main/java/resources/documents/".

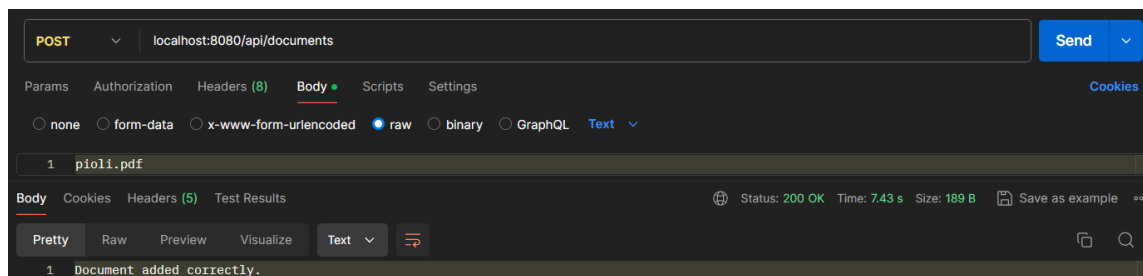


Figura 6: Esempio chiamata POST in /api/documents

5.3 Rimozione documento

Endpoint: localhost:8080/api/documents;

Metodo HTTP: DELETE;

Corpo della richiesta: Stringa del nome del documento da rimuovere, nel formato "nomedocumento.estensione";

Output atteso: Stringa di conferma rimozione;

NOTA BENE: In questo prototipo, nel rimuovere un documento, ovvero i suoi embedding, dal database vettoriale, al termine dell'operazione il file continuerà a trovarsi all'interno del percorso "src/main/java/resources/documents/", per poter essere reinserito in seguito.

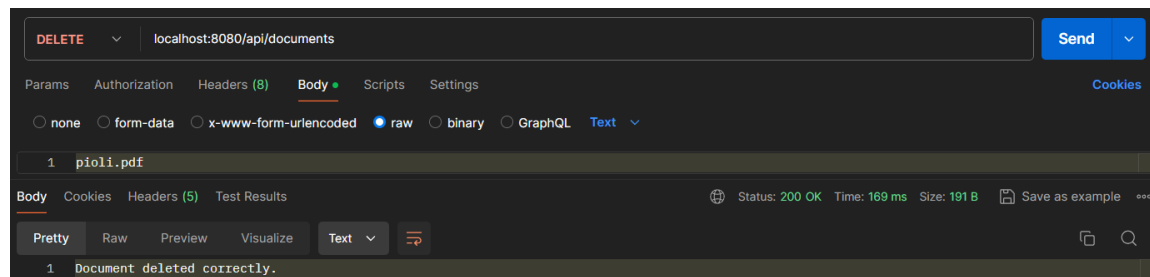


Figura 7: Esempio chiamata DELETE in /api/documents

5.4 Interrogazione chatbot

Endpoint: localhost:8080/api/ask;

Metodo HTTP: GET;

Corpo della richiesta: Stringa della domanda;

Output atteso: Stringa della risposta.

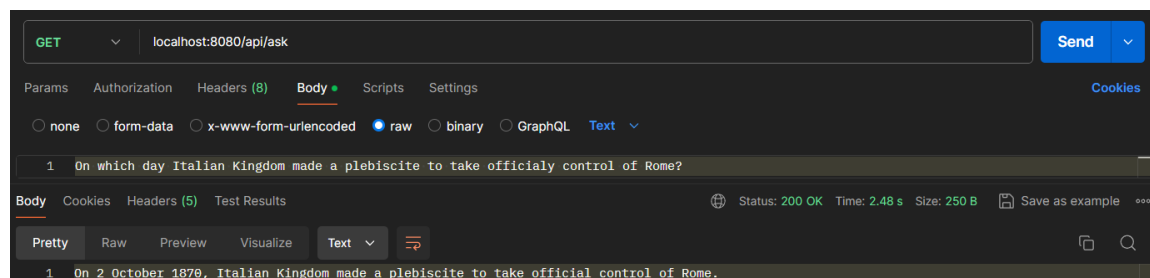


Figura 8: Esempio chiamata GET in /api/ask

Modelli

I prototipi prodotti si differenziano tra loro per il modello utilizzato nella fase di generation.

Questa differenza è visibile nel momento in cui è necessario istanziare il modello nel RagService.

6.1 Phi-3 mini

```
ChatLanguageModel model = OllamaChatModel.builder()
    .baseUrl("http://localhost:11434")
    .modelName("phi3")
    .temperature(0.2)
    .timeout(Duration.ofMinutes(1))
    .build();
```

6.2 GPT 3.5 Turbo

```
ChatLanguageModel model = OpenAiChatModel.builder()
    .apiKey("demo")
    .modelName(GPT_3_5_TURBO)
    .temperature(0.2)
    .build();
```

6.3 Gemini 1.5 Pro

```
ChatLanguageModel model = VertexAiGeminiChatModel.builder()
    .project(PROJECT-NAME)
    .location("us-central1")
    .modelName("gemini-pro")
    .temperature(0.4F)
    .build();
```

6.4 PaLM 2 Bison

```
ChatLanguageModel model = VertexAiChatModel.builder()  
    .project(PROJECT_NAME)  
    .location("us-central1")  
    .modelName("chat-bison")  
    .publisher("google")  
    .endpoint("us-central1-aiplatform.googleapis.com:443")  
    .temperature(0.5)  
    .build();
```