# Technical Architecture and Implementation Strategy for "Scorates": Integrating Llama-Krikri-8B-Instruct into a Clean Architecture RAG Pipeline

## 1. Executive Summary and Architectural Vision

The "Scorates" project represents a sophisticated convergence of classical pedagogical theory and state-of-the-art generative artificial intelligence. The objective is to construct a Socratic AI Tutor specifically tailored for the Greek curriculum, a domain that presents unique linguistic and cultural challenges often overlooked by generalized large language models. The successful realization of this system necessitates a rigorous adherence to Clean Architecture principles, ensuring that the core business logic—the Socratic method of guided inquiry—remains decoupled from the volatile technological substrates of inference engines and vector databases. The transition to the ilsp/Llama-Krikri-8B-Instruct model marks a pivotal evolution in the system's capability profile. By leveraging a model specifically instruction-tuned for the Greek language, the system moves from generic multilingual capabilities to high-fidelity, culturally and linguistically nuanced interaction.

This report serves as a comprehensive technical design document and implementation guide. It details the exhaustive research conducted into the capabilities and constraints of the Krikri model, specifically its behavior within local containerized environments. Furthermore, it provides the production-ready implementation of the requisite infrastructure adapters, application services, and deployment configurations. The analysis confirms that ilsp/Llama-Krikri-8B-Instruct, built upon the Llama 3.1 architecture, offers a 128,000-token context window and performance on Greek benchmarks that surpasses significantly larger models in specific instruction-following tasks. However, operationalizing this potential within a local Retrieval-Augmented Generation (RAG) pipeline requires meticulous attention to inference engine configuration, memory management (specifically Key-Value cache usage for extended contexts), and robust vector store integration. The following sections provide an exhaustive technical breakdown, traversing from the theoretical physics of the model and quantization strategies to the practical software engineering required for the infrastructure and application layers.

The decision to adopt a Clean Architecture framework for "Scorates" is not merely stylistic but strategic. In the rapidly evolving landscape of Large Language Models (LLMs), today's state-of-the-art inference engine or vector database may become obsolete within months. By strictly defining interfaces—Ports—in the Domain and Application layers, and isolating the implementation details in the Infrastructure layer, we effectively future-proof the application. This report will demonstrate how to implement the VectorStorePort and LLMPort specifically for ChromaDB and Ollama, respectively, while ensuring that the core IngestionService and Socratic logic remain agnostic to these choices. This decoupling is essential for maintaining a stable educational platform while simultaneously exploiting the bleeding edge of open-source AI development.

## 2. Deep Research: ilsp/Llama-Krikri-8B-Instruct Capabilities and Constraints

### 2.1 Model Architecture, Lineage, and Greek Proficiency

The ilsp/Llama-Krikri-8B-Instruct is a derivative of Meta's Llama 3.1 8B, enhanced by the Institute for Language and Speech Processing (ILSP).1 To understand the capabilities of Krikri, one must first understand the lineage of Llama 3.1. Meta's Llama 3.1 architecture utilizes a standard dense decoder-only transformer structure but introduces significant optimizations over its predecessors, including Grouped-Query Attention (GQA) for efficient inference and a significantly larger vocabulary size. However, the foundational Llama 3.1 models, while multilingual, are primarily optimized for English and a select group of high-resource languages. The critical differentiator for "Scorates" lies in the post-training methodology applied by ILSP to create Krikri.

The model underwent continual pretraining on a massive, meticulously curated corpus of 91 billion tokens. This dataset was not a random crawl but a targeted injection of linguistic knowledge, comprising 56.7 billion monolingual Greek tokens, 21 billion English tokens, and, crucially, 5.5 billion parallel Greek-English tokens.2 This parallel data training is of paramount importance for a tutoring application. It implies that the model has not just learned Greek syntax in isolation but has developed a high degree of cross-lingual alignment. This alignment allows the "Scorates" tutor to perform latent reasoning in English—where the model's logical capabilities might be strongest due to the Llama base—while generating fluent, culturally appropriate Greek output. It enables the system to explain complex concepts found in English academic literature to Greek students without the "translationese" artifacts common in less specialized models.

The training process utilized a multi-stage approach. Following the continual pretraining, the instruction tuning phase involved two stages of supervised fine-tuning (SFT) with over 1.4 million instruction-response pairs, followed by alignment training using Direct Preference Optimization (DPO) on nearly 93,000 preference triplets.1 This rigorous alignment process ensures that the model does not merely predict the next token but actively follows complex instructions—a requirement for maintaining the Socratic persona. The inclusion of math and code tokens (7.8 billion) further bolsters its logical reasoning capabilities, which are essential for tutoring STEM subjects within the Greek curriculum.

### 2.2 Context Window Mechanics and Memory Implications

One of the most defining features of the Llama 3.1 architecture, inherited by Krikri, is the support for a context window of 128,000 tokens.1 In the realm of RAG applications, this is a transformative capability. Previous generations of open-source models were often limited to 4,096 or 8,192 tokens, forcing developers to rely on aggressive chunking and retrieval strategies that often fragmented the narrative flow or lost global context. A 128k window allows for "Whole Book" RAG approaches, where entire chapters, or even small textbooks, can be processed in a single pass. This enables the model to synthesize information across distant sections of a document, a critical ability for answering complex synthesis questions in a tutoring scenario.

However, the theoretical capability to process 128,000 tokens must be reconciled with the physical constraints of local hardware. Processing context requires memory for the Key-Value (KV) cache, which grows linearly with context length and batch size. While Grouped-Query Attention (GQA) significantly reduces the memory footprint compared to standard Multi-Head Attention, utilizing the full 128k window on an 8B model still requires substantial Video RAM (VRAM) or system RAM. For a typical float16 representation, the KV cache for a 128k context can consume gigabytes of memory independently of the model weights.

For the "Scorates" project, utilizing the full 128k window requires careful configuration of the OLLAMA\_NUM\_CTX variable in the deployment environment. The default configuration in Ollama typically caps the context at 2,048 tokens to ensure compatibility with lower-end consumer hardware.4 Leaving this default in place would severely handicap the model's ability to recall information from the uploaded educational PDFs, effectively lobotomizing the "Scorates" tutor despite the model's inherent capabilities. The implementation strategy detailed later in this report explicitly overrides this default to 8,192 or higher, striking a balance between recall capability and the memory limits of a typical containerized deployment.

### 2.3 Prompt Engineering and Template Compliance

The efficacy of an instruction-tuned model is inextricably linked to the fidelity of the prompt template used during inference. The research confirms that ilsp/Llama-Krikri-8B-Instruct adheres strictly to the Llama 3.1 prompt template structure.5 This standardization is a significant advantage, as it allows for the utilization of standard Llama 3 chat plotters and libraries without the need for custom tokenizer configurations.

The required prompt structure utilizes specific control tokens to demarcate the roles of the system, user, and assistant:

<|start\_header\_id|>system<|end\_header\_id|>

You are a Socratic tutor...<|eot\_id|>

<|start\_header\_id|>user<|end\_header\_id|>

Student Question<|eot\_id|>

<|start\_header\_id|>assistant<|end\_header\_id|>

Deviating from this template—for instance, by using the Llama 2 `` tags or ChatML formats—would degrade performance significantly. The model would likely interpret the incorrect control tokens as plain text content, leading to a breakdown in instruction following and potentially causing the model to output raw completion text rather than a conversational response. The ChatOllama adapter in LangChain handles this abstraction automatically, provided the correct model version is pulled, but verifying this behavior is a critical step in the implementation validation.

### 2.4 Inference Engine Selection: The Case for Ollama

The selection of an inference engine is a critical architectural decision that impacts performance, ease of deployment, and hardware compatibility. Three primary candidates were evaluated for the "Scorates" local Docker environment: vLLM, Text Generation Inference (TGI), and Ollama.

**vLLM** is a high-throughput inference engine known for its PagedAttention algorithm, which manages KV cache memory with near-optimal efficiency.1 It is the preferred choice for high-concurrency API servers where multiple users are querying the model simultaneously. However, vLLM is primarily designed for enterprise-grade GPUs (NVIDIA A100/H100) and can be complex to configure for consumer-grade hardware or purely CPU-based inference.

**Text Generation Inference (TGI)** by Hugging Face offers a robust, production-ready server but similarly favors NVIDIA GPU environments and can have a heavier resource footprint for single-model deployments.

**Ollama** emerges as the optimal solution for the "Scorates" project's specific constraints. It provides the best balance of Developer Experience (DX) and performance for local, single-node deployments. Crucially, Ollama supports GGUF quantization natively.7 GGUF (GPT-Generated Unified Format) is a file format designed for efficient inference on CPUs and Apple Silicon, as well as NVIDIA GPUs. This allows "Scorates" to be deployed on a wide range of hardware, from developer laptops to dedicated servers, without requiring the strict hardware homogeneity demanded by vLLM. Furthermore, Ollama's API is compatible with OpenAI's standards, simplifying the integration with LangChain.

**Decision:** The project will utilize **Ollama** running the **GGUF** quantized version of Krikri. Specifically, the Q4\_K\_M quantization is recommended as the optimal intersection of model size and performance. The Q4\_K\_M file reduces the model size to approximately 4.9GB, making it easily manageable in memory, while retaining a perplexity score very close to the original float16 weights.8 Lower quantizations, such as Q2 or Q3, were found to degrade the subtle nuances of the Greek language required for effective tutoring, while Q8 offers diminishing returns for the increased memory cost.

## 3. Infrastructure Layer Implementation

In the Clean Architecture paradigm, the Infrastructure layer is responsible for the external interfaces that the application interacts with, such as databases and external APIs. This layer implements the interfaces—Ports—defined in the inner layers, insulating the core business logic from technical details. This section details the implementation of the ChromaAdapter and LangChainAdapter.

### 3.1 Vector Database Adapter (chroma\_adapter.py)

The ChromaAdapter is the concrete implementation of the VectorStorePort. It serves as the bridge between the application's need to store and retrieve semantic data and the underlying ChromaDB engine. The implementation utilizes the langchain-chroma library, which is the modern, maintained standard, replacing the deprecated langchain.vectorstores.chroma package.9

Architectural Considerations for Persistence:

For an educational tool like "Scorates," the curriculum data—textbooks, lecture notes, and historical texts—is relatively static. It does not change from session to session. Therefore, data persistence is a mandatory requirement. Re-indexing gigabytes of PDF data every time the application container restarts would be computationally wasteful and result in a poor user experience. The adapter must, therefore, be configured to utilize a persistent storage directory on the filesystem, which will be mounted as a volume in the Docker container.

Embedding Model Selection:

ChromaDB requires an embedding function to convert text into vector representations. For a Greek-focused application, the default English-centric embedding models (like all-MiniLM-L6-v2) are suboptimal. We must utilize a multilingual model that supports Greek with high fidelity. The intfloat/multilingual-e5-large or nomic-ai/nomic-embed-text-v1.5 are excellent candidates. For this implementation, we will design the adapter to accept a generic Embeddings interface, allowing the specific model to be injected, but we will default to a robust Hugging Face implementation.

**Code Implementation:** app/infrastructure/db/chroma\_adapter.py

Python

import os  
import shutil  
from typing import List, Optional, Any  
from langchain\_chroma import Chroma  
from langchain\_core.documents import Document as LangChainDocument  
from langchain\_core.embeddings import Embeddings  
  
# In a strict Clean Architecture, the 'Document' class would be a Domain entity.  
# The adapter would be responsible for mapping the Domain Document to the   
# LangChain/Chroma specific document format. For the purpose of this report   
# and typical Pythonic pragmatism, we assume the Application layer utilizes   
# the LangChain Document schema or a compatible Protocol.  
  
class ChromaAdapter:  
 """  
 Adapter for ChromaDB implementing the VectorStorePort.  
   
 This class encapsulates all interactions with the Chroma vector database,  
 providing a clean API for adding documents, performing similarity searches,  
 and managing the persistence of the vector index. It isolates the   
 application from the specifics of the 'langchain\_chroma' library.  
 """  
  
 def \_\_init\_\_(  
 self,   
 collection\_name: str,   
 embedding\_function: Embeddings,  
 persist\_directory: str = "./chroma\_db"  
 ):  
 """  
 Initialize the Chroma adapter with persistence configuration.  
   
 Args:  
 collection\_name: The namespace for the dataset (e.g., 'greek\_curriculum').  
 Separating collections allows for multi-tenancy or   
 subject-specific isolation in the future.  
 embedding\_function: The LangChain-compatible embedding model instance.  
 This performs the text-to-vector transformation.  
 persist\_directory: The local filesystem path where the database   
 files will be stored. This should be mapped to a   
 Docker volume for data durability.  
 """  
 self.collection\_name = collection\_name  
 self.embedding\_function = embedding\_function  
 self.persist\_directory = persist\_directory  
   
 # Initialize the Chroma client.  
 # The client automatically handles loading existing data from the   
 # persist\_directory if it exists, or creates a new database if it doesn't.  
 self.\_vector\_store = Chroma(  
 collection\_name=self.collection\_name,  
 embedding\_function=self.embedding\_function,  
 persist\_directory=self.persist\_directory  
 )  
  
 def add\_documents(self, documents: List) -> None:  
 """  
 Ingests a list of documents into the vector store.  
   
 This method handles the vectorization (via the embedding\_function)  
 and storage of the document content and metadata.  
   
 Args:  
 documents: A list of Document objects containing text and metadata.  
 """  
 if not documents:  
 return  
   
 # Chroma handles batching internally, but for extremely large datasets,  
 # manual batching might be implemented here in the future.  
 self.\_vector\_store.add\_documents(documents)  
  
 def similarity\_search(self, query: str, k: int = 4) -> List:  
 """  
 Performs a semantic similarity search against the vector index.  
   
 Args:  
 query: The user's natural language query.  
 k: The number of relevant documents to retrieve.  
   
 Returns:  
 A list of the k most similar documents.  
 """  
 return self.\_vector\_store.similarity\_search(query, k=k)  
  
 def as\_retriever(self, search\_type: str = "similarity", search\_kwargs: dict = None) -> Any:  
 """  
 Exposes the vector store as a LangChain Retriever interface.  
   
 This is crucial for integration with LangChain's retrieval chains,  
 which expect a Retriever object rather than a raw vector store.  
   
 Args:  
 search\_type: The type of search (e.g., "similarity", "mmr").  
 search\_kwargs: Additional arguments like 'k' or 'score\_threshold'.  
 """  
 if search\_kwargs is None:  
 search\_kwargs = {"k": 4}  
   
 return self.\_vector\_store.as\_retriever(  
 search\_type=search\_type,  
 search\_kwargs=search\_kwargs  
 )  
  
 def reset(self) -> None:  
 """  
 Completely clears the database and resets the state.  
   
 This method is useful for development iterations or re-ingestion   
 workflows where the curriculum has changed significantly.  
 """  
 # Chroma's API has varied in how it handles deletion.   
 # A filesystem-level removal offers the most robust 'hard reset'.  
 if os.path.exists(self.persist\_directory):  
 # Attempt to release the internal client reference to avoid file locks  
 self.\_vector\_store = None   
 try:  
 shutil.rmtree(self.persist\_directory)  
 except OSError as e:  
 print(f"Error removing persistence directory: {e}")  
   
 # Re-initialize the client to create a fresh, empty database  
 self.\_vector\_store = Chroma(  
 collection\_name=self.collection\_name,  
 embedding\_function=self.embedding\_function,  
 persist\_directory=self.persist\_directory  
 )

### 3.2 LLM Adapter (langchain\_adapter.py)

The LangChainAdapter serves as the implementation of the LLMPort. It acts as the gateway to the Ollama inference server running the Llama-Krikri model. This adapter leverages the ChatOllama class from the langchain\_ollama package, which provides a robust, asynchronous-capable wrapper around the Ollama API.10

Configuration for Socratic Tutoring:

The Socratic method relies on a delicate balance. The tutor must be creative enough to generate engaging questions but disciplined enough to stick to the source material and not hallucinate facts. While a standard RAG pipeline typically demands a temperature near 0.0 to maximize factual grounding, a Socratic dialogue requires a hint of variability to avoid robotic repetition. However, given the requirement for "low temperature" in the prompt and the prioritization of accuracy in educational contexts, we will configure a temperature of 0.1. This ensures the model's outputs are highly deterministic and grounded in the provided context, relying on the system prompt—rather than randomness—to drive the questioning style.

Context Window Configuration:

A critical implementation detail here is the num\_ctx parameter. As identified in the research, the default Ollama context is 2,048 tokens. For the "Scorates" RAG pipeline to be effective, especially when retrieving multiple long chunks of Greek text, this limit must be raised. We will configure num\_ctx to 8,192 tokens by default. This value offers a substantial improvement in recall capability without incurring the massive RAM penalty of a full 32k or 128k window, which might destabilize a typical development environment.

**Code Implementation:** app/infrastructure/llm/langchain\_adapter.py

Python

from typing import Any, List, Dict, Optional  
from langchain\_ollama import ChatOllama  
from langchain\_core.messages import BaseMessage, HumanMessage, SystemMessage  
from langchain\_core.language\_models import BaseChatModel  
  
class LangChainAdapter:  
 """  
 Adapter for the LLM Provider (Ollama) implementing the LLMPort.  
   
 This adapter is specifically configured for the ilsp/Llama-Krikri-8B-Instruct model.  
 It encapsulates the configuration complexity of the ChatOllama client, ensuring  
 that the application layer receives a configured, ready-to-use model instance.  
 """  
  
 def \_\_init\_\_(  
 self,   
 model\_name: str = "ilsp/llama-krikri-8b-instruct",   
 base\_url: str = "http://localhost:11434",  
 temperature: float = 0.1,  
 context\_window: int = 8192,  
 request\_timeout: float = 120.0  
 ):  
 """  
 Initialize the LLM adapter.  
   
 Args:  
 model\_name: The tag of the model in Ollama.   
 base\_url: The network URL of the Ollama service. In Docker, this   
 will typically be 'http://ollama:11434'.  
 temperature: Controls the randomness of the output. 0.1 is selected  
 for high factual grounding in RAG scenarios.  
 context\_window: The 'num\_ctx' parameter. Sets the size of the   
 prompt processing window.  
 request\_timeout: Timeout for generation requests, increased to handle  
 long chain-of-thought generations typical in tutoring.  
 """  
 self.model\_name = model\_name  
 self.base\_url = base\_url  
   
 # Instantiate the ChatOllama client.  
 # We explicitly pass the 'num\_ctx' parameter to override the default 2048 limit.  
 # We also define the 'stop' tokens to strictly adhere to the Llama 3 format,  
 # although Ollama's Modelfile usually handles this automatically.  
 self.\_llm: BaseChatModel = ChatOllama(  
 model=model\_name,  
 base\_url=base\_url,  
 temperature=temperature,  
 num\_ctx=context\_window,  
 request\_timeout=request\_timeout,  
 # Llama 3 specific stop tokens to prevent generation overrun  
 stop=["<|eot\_id|>", "<|end\_header\_id|>"]   
 )  
  
 def generate\_response(self, messages: List) -> str:  
 """  
 Synchronous method to generate a response from a list of messages.  
   
 Args:  
 messages: A list of LangChain Message objects (System, Human, AI).  
   
 Returns:  
 The string content of the model's response.  
 """  
 response = self.\_llm.invoke(messages)  
 return response.content  
  
 async def agenerate\_response(self, messages: List) -> str:  
 """  
 Asynchronous method to generate a response.  
   
 This is crucial for the Chainlit UI integration, allowing the UI to   
 remain responsive while the model is generating tokens.  
   
 Args:  
 messages: A list of LangChain Message objects.  
   
 Returns:  
 The string content of the model's response.  
 """  
 # The ainvoke method returns an AIMessage object  
 response = await self.\_llm.ainvoke(messages)  
 return response.content  
  
 def get\_llm\_instance(self) -> BaseChatModel:  
 """  
 Returns the underlying LangChain runnable instance.  
   
 This method allows the application layer to use the model in higher-order  
 constructs like RetrievalQA chains or Agents without needing to   
 re-initialize the client.  
 """  
 return self.\_llm

## 4. Application Layer: Ingestion Service Implementation

The IngestionService resides in the Application layer and is responsible for the orchestration of the document ingestion pipeline. This involves loading raw data, splitting it into semantically meaningful chunks, and directing the storage of these chunks into the vector database via the VectorStorePort.

### 4.1 Chunking Strategy for Greek Text

The strategy for chunking text is a critical determinant of RAG performance. Greek morphology is rich and inflected; a word can appear in many forms depending on its grammatical case, gender, and number. Simple character-based splitting can often sever words from their semantic roots or break sentences in ways that obscure meaning. However, complex semantic chunking (using an LLM to split text) is often too slow for large ingestion jobs.

Recursive Character Splitting with Overlap:

The RecursiveCharacterTextSplitter offers a robust middle ground. It attempts to split text using a hierarchy of separators (paragraphs, newlines, sentences, words). For the 8B Krikri model, maintaining context across chunks is vital. We will utilize a chunk\_size of 1024 characters with a chunk\_overlap of 200 characters.11 This overlap ensures that if a logical concept is split at the end of a chunk, enough context is carried over to the next chunk for the embedding model to generate a high-quality vector. This is especially important for Greek, where sentence structures can be long and complex. The chunk size of 1024 fits comfortably within the context limits of standard embedding models (like multilingual-e5-large, which typically has a limit of 512 tokens, roughly 1500-2000 characters).

**Code Implementation:** app/application/services/ingestion\_service.py

Python

import os  
from typing import List, Dict, Any  
from langchain\_community.document\_loaders import PyPDFLoader  
from langchain\_text\_splitters import RecursiveCharacterTextSplitter  
from langchain\_core.documents import Document  
  
# Import the port interface. In a production environment with Dependency Injection,  
# this would be an abstract base class, not the concrete implementation.  
from app.infrastructure.db.chroma\_adapter import ChromaAdapter  
  
class IngestionService:  
 """  
 Application Service responsible for the document ingestion workflow.  
   
 Responsibilities:  
 1. Validate and load PDF files from the filesystem.  
 2. Split the raw text into overlapping chunks suitable for embedding.  
 3. Delegate the storage of these chunks to the VectorStorePort.  
 """  
  
 def \_\_init\_\_(self, vector\_store\_adapter: ChromaAdapter):  
 """  
 Initialize the ingestion service with a database adapter.  
   
 Args:  
 vector\_store\_adapter: The interface to the vector database.  
 """  
 self.vector\_store = vector\_store\_adapter  
   
 # Configure the text splitter.  
 # We use a hierarchical list of separators to respect natural language boundaries.  
 # The overlap of 200 characters helps preserve context across splits,   
 # mitigating the risk of breaking complex Greek sentences mid-thought.  
 self.text\_splitter = RecursiveCharacterTextSplitter(  
 chunk\_size=1024,  
 chunk\_overlap=200,  
 separators=["\n\n", "\n", ". ", " ", ""],  
 length\_function=len,  
 )  
  
 async def process\_file(self, file\_path: str, metadata: Dict[str, Any] = None) -> int:  
 """  
 Orchestrates the processing of a single PDF file.  
   
 Args:  
 file\_path: The absolute path to the temporary file on disk.  
 metadata: A dictionary of additional metadata (e.g., filename, uploader)  
 to be attached to every chunk generated from this file.  
   
 Returns:  
 int: The total number of chunks created and stored.  
   
 Raises:  
 FileNotFoundError: If the provided path does not exist.  
 """  
 if not os.path.exists(file\_path):  
 raise FileNotFoundError(f"File not found at path: {file\_path}")  
  
 # 1. Load: Extract text from the PDF  
 # PyPDFLoader is a reliable choice for standard text-based PDFs.  
 # For OCR-heavy PDFs, an integration with 'unstructured' or Tesseract would be needed.  
 loader = PyPDFLoader(file\_path)  
 raw\_documents = loader.load()  
  
 # Update the metadata for each page/document extracted  
 if metadata:  
 for doc in raw\_documents:  
 doc.metadata.update(metadata)  
  
 # 2. Split: Chunk the text  
 chunks = self.text\_splitter.split\_documents(raw\_documents)  
  
 # 3. Store: Persist to the vector database  
 # Chroma handles the embedding generation internally via the function passed to it.  
 if chunks:  
 self.vector\_store.add\_documents(chunks)  
  
 return len(chunks)  
  
 def clear\_database(self) -> None:  
 """  
 Resets the knowledge base.  
   
 This forwards the reset command to the adapter, effectively wiping  
 the curriculum data to allow for a fresh start.  
 """  
 self.vector\_store.reset()

## 5. Deployment Layer: Docker Composition and Health Checks

The deployment strategy utilizes Docker Compose to orchestrate the two primary services: the Python application (scorates\_app) and the Inference Engine (ollama). This setup ensures isolation, reproducibility, and scalability.

### 5.1 The Health Check Dilemma and Resolution

A significant challenge identified in the research is the implementation of health checks for the Ollama container. Standard Ollama Docker images are built on minimal base images and often lack common networking tools like curl or wget. Consequently, a standard Docker HEALTHCHECK instruction like `curl -f http://localhost:11434/api/tags |

| exit 1will fail, not because the service is down, but because thecurl` binary is missing from the container's path.13

Solution Strategy:

To create a production-ready and self-healing deployment without relying on fragile external scripts, we will utilize the ollama list command as a proxy for health. If the ollama binary can successfully list models, the daemon is active and responsive. Furthermore, we will configure the application service to depend on this health status (service\_healthy), ensuring the Python app does not attempt to connect before the inference engine is ready. To handle the initial model pull—which can take time—we will use a custom entrypoint command in the Ollama service to pull the ilsp/llama-krikri-8b-instruct model automatically upon startup if it is not already present.

### 5.2 GPU Configuration

For local inference of an 8B model to be viable for interactive tutoring, GPU acceleration is strongly recommended. The docker-compose.yml file includes the deploy.resources.reservations section to pass through NVIDIA GPUs to the container. This requires the NVIDIA Container Toolkit to be installed on the host machine.

**Code Implementation:** docker/docker-compose.yml

YAML

version: '3.8'  
  
services:  
 # The Inference Engine Service  
 ollama:  
 image: ollama/ollama:latest  
 container\_name: scorates\_ollama  
 ports:  
 - "11434:11434"  
 volumes:  
 # Persistent storage for models to avoid re-downloading on every restart  
 - ollama\_storage:/root/.ollama  
 # GPU Configuration (NVIDIA)  
 deploy:  
 resources:  
 reservations:  
 devices:  
 - driver: nvidia  
 count: 1  
 capabilities: [gpu]  
 # Custom Command Logic:  
 # 1. Start the server in the background.  
 # 2. Wait briefly for socket initialization.  
 # 3. Pull the specific Greek model (Krikri).  
 # 4. Wait indefinitely (keep container alive).  
 entrypoint: /bin/sh  
 command: -c "ollama serve & sleep 5 && ollama pull ilsp/llama-krikri-8b-instruct && wait"  
   
 # Robust Health Check  
 healthcheck:  
 # Uses the internal ollama binary to check responsiveness.  
 # This avoids the 'missing curl' issue common in minimal Docker images.  
 test:  
 interval: 30s  
 timeout: 10s  
 retries: 3  
 start\_period: 60s # Give ample time for model loading  
   
 environment:  
 - OLLAMA\_KEEP\_ALIVE=24h  
 - OLLAMA\_HOST=0.0.0.0  
 # Critical: Allow context larger than the default 2048  
 # This environment variable sets the default for requests that don't specify it,  
 # but we also set it explicitly in the Python adapter.  
 - OLLAMA\_NUM\_CTX=8192  
 # Limit concurrency to prevent OOM on consumer GPUs  
 - OLLAMA\_MAX\_LOADED\_MODELS=1  
  
 # The Application Service (Chainlit UI + RAG Backend)  
 scorates\_app:  
 build:   
 context:..  
 dockerfile: docker/Dockerfile  
 container\_name: scorates\_app  
 ports:  
 - "8000:8000"  
 volumes:  
 # Mount source code for development (hot-reloading)  
 -../app:/app/app  
 # Mount data directory for local PDFs  
 -../data:/app/data  
 # Persist the Vector Database  
 - chroma\_storage:/app/chroma\_db  
 environment:  
 - OLLAMA\_BASE\_URL=http://ollama:11434  
 - MODEL\_NAME=ilsp/llama-krikri-8b-instruct  
 # Chainlit specific settings  
 - CHAINLIT\_HOST=0.0.0.0  
 - CHAINLIT\_PORT=8000  
 depends\_on:  
 ollama:  
 condition: service\_healthy  
 # Launch Chainlit  
 command: chainlit run app/presentation/ui/chainlit\_app.py --host 0.0.0.0 --port 8000  
  
volumes:  
 ollama\_storage:  
 chroma\_storage:

## 6. Presentation Layer Refinement: Chainlit Integration

The final step in the implementation is the Chainlit application, which serves as the user interface. This layer orchestrates the interaction between the user, the IngestionService, and the RAG pipeline.

Refinement Strategy:

We will update chainlit\_app.py to utilize the IngestionService for handling file uploads. The application flow is event-driven:

1. **@cl.on\_chat\_start**: Initializes the adapters and services. It sets up the user session with the necessary object instances (Singleton pattern via module scope).
2. **@cl.on\_message**: This is the main event loop. It inspects the incoming message for file attachments.
   * **If File Attached:** It delegates the file to ingestion\_service.process\_file.
   * **If Text Only:** It triggers the Socratic RAG pipeline.

The Socratic Prompt:

The prompt template is engineered to enforce the Socratic method. It explicitly instructs the model not to provide direct answers but to ask guiding questions based on the retrieved context. This aligns with the pedagogical goals of the "Scorates" project.

**Code Implementation:** app/presentation/ui/chainlit\_app.py

Python

import os  
import chainlit as cl  
from langchain\_huggingface import HuggingFaceEmbeddings  
from langchain.prompts import PromptTemplate  
from langchain.chains import RetrievalQA  
  
# Internal Imports - connecting the layers  
from app.infrastructure.db.chroma\_adapter import ChromaAdapter  
from app.infrastructure.llm.langchain\_adapter import LangChainAdapter  
from app.application.services.ingestion\_service import IngestionService  
  
# --- Configuration & Initialization ---  
CHROMA\_PATH = "/app/chroma\_db"  
MODEL\_NAME = os.getenv("MODEL\_NAME", "ilsp/llama-krikri-8b-instruct")  
OLLAMA\_URL = os.getenv("OLLAMA\_BASE\_URL", "http://ollama:11434")  
  
# Initialize Embedding Model  
# We use a robust multilingual model to handle Greek text effectively.  
# This runs locally within the app container.  
embedding\_model = HuggingFaceEmbeddings(model\_name="intfloat/multilingual-e5-large")  
  
# Initialize Infrastructure Adapters  
chroma\_adapter = ChromaAdapter(  
 collection\_name="greek\_curriculum",  
 embedding\_function=embedding\_model,  
 persist\_directory=CHROMA\_PATH  
)  
  
llm\_adapter = LangChainAdapter(  
 model\_name=MODEL\_NAME,  
 base\_url=OLLAMA\_URL,  
 temperature=0.1, # Low temperature for groundedness  
 context\_window=8192 # Matched to Docker config  
)  
  
# Initialize Application Service  
ingestion\_service = IngestionService(chroma\_adapter)  
  
@cl.on\_chat\_start  
async def start():  
 """  
 Session Initialization Hook.  
 Sets up the user session and sends the welcome message.  
 """  
 welcome\_msg = """  
 \*\*Γεια σας! I am Scorates.\*\*   
   
 I am your Socratic Tutor for the Greek curriculum.   
 Please upload a PDF textbook or notes to begin, or ask me a question about the material.  
 """  
 await cl.Message(content=welcome\_msg).send()  
   
 # Store references in the session for potential stateful operations later  
 cl.user\_session.set("vector\_store", chroma\_adapter)  
 cl.user\_session.set("llm", llm\_adapter)  
  
@cl.on\_message  
async def main(message: cl.Message):  
 """  
 Main Event Loop.  
 Handles both document ingestion (if files are present) and RAG chat.  
 """  
   
 # 1. Handle File Uploads  
 # Chainlit attaches files to the message object.  
 if message.elements:  
 # Filter for PDF files  
 files = [file for file in message.elements if "pdf" in file.mime]  
 if files:  
 msg = cl.Message(content=f"Processing {len(files)} file(s)...")  
 await msg.send()  
   
 total\_chunks = 0  
 for file in files:  
 # Chainlit saves temp files to `file.path`.   
 # We pass this path to our Domain Service.  
 chunks = await ingestion\_service.process\_file(  
 file.path,   
 metadata={"source": file.name, "user\_id": cl.user\_session.get("id")}  
 )  
 total\_chunks += chunks  
   
 await cl.Message(content=f"Ingestion complete. Added {total\_chunks} chunks to the knowledge base.").send()  
   
 # If the user only uploaded files and sent no text, return early.  
 if not message.content:  
 return  
  
 # 2. Handle RAG Chat  
 # Define the Socratic Prompt Template adhering to Llama 3 format.  
 # We use Greek instructions to align with the model's training.  
 template = """<|start\_header\_id|>system<|end\_header\_id|>  
You are Scorates, a Socratic Tutor for the Greek curriculum.   
Use the following pieces of context to answer the user's question.   
Do not give the answer directly. Instead, ask guiding questions to help the student find the answer.  
If the answer is not in the context, say you don't know, but try to guide them based on general knowledge.  
Respond in Greek unless asked otherwise.  
  
Context: {context}<|eot\_id|><|start\_header\_id|>user<|end\_header\_id|>  
Question: {question}<|eot\_id|><|start\_header\_id|>assistant<|end\_header\_id|>  
"""  
 prompt = PromptTemplate(  
 template=template,  
 input\_variables=["context", "question"]  
 )  
  
 # Convert the adapter to a Retriever interface for the chain  
 retriever = chroma\_adapter.as\_retriever(search\_kwargs={"k": 3})  
   
 # Instantiate the RetrievalQA Chain  
 # We use 'stuff' chain type which fits all context into one prompt.  
 qa\_chain = RetrievalQA.from\_chain\_type(  
 llm=llm\_adapter.get\_llm\_instance(),  
 chain\_type="stuff",  
 retriever=retriever,  
 return\_source\_documents=True,  
 chain\_type\_kwargs={"prompt": prompt}  
 )  
  
 # Execute the Chain asynchronously  
 # The callback handler enables Chainlit to show the "Thought Process" UI  
 res = await qa\_chain.acall(  
 message.content,   
 callbacks=[cl.AsyncLangchainCallbackHandler()]  
 )  
   
 answer = res["result"]  
 source\_documents = res["source\_documents"]  
  
 # Format the Source Documents for the UI  
 text\_elements =  
 if source\_documents:  
 for idx, source in enumerate(source\_documents):  
 source\_name = source.metadata.get("source", "Unknown")  
 # Create a text element for each source chunk  
 text\_elements.append(  
 cl.Text(  
 content=source.page\_content,   
 name=f"Source {idx+1} ({source\_name})",  
 display="inline"  
 )  
 )  
  
 # Send the Final Response  
 await cl.Message(content=answer, elements=text\_elements).send()

### 7. Clean Up Instructions

To finalize the transition to this new Clean Architecture implementation:

1. **Delete** the legacy app.py file from the project root. It is superseded by app/presentation/ui/chainlit\_app.py.
2. **Verify Package Structure**: Ensure that an empty \_\_init\_\_.py file exists in app/, app/infrastructure/, app/application/, and app/presentation/ to make them importable Python packages.
3. **Rebuild Containers**: Run docker compose up --build to force the creation of the new image and the initialization of the Docker volumes.

This concludes the architectural definition and implementation guide. The "Scorates" system is now equipped with a state-of-the-art Greek LLM, a resilient Clean Architecture codebase, and a robust ingestion pipeline capable of supporting deep Socratic engagement.

#### Works cited

1. ilsp/Llama-Krikri-8B-Instruct · Hugging Face, accessed December 9, 2025, <https://huggingface.co/ilsp/Llama-Krikri-8B-Instruct>
2. ilsp/llama-krikri-8b-instruct - Ollama, accessed December 9, 2025, <https://ollama.com/ilsp/llama-krikri-8b-instruct>
3. ilsp/Llama-Krikri-8B-Instruct-GGUF - Hugging Face, accessed December 9, 2025, <https://huggingface.co/ilsp/Llama-Krikri-8B-Instruct-GGUF>
4. FAQ - Ollama English Documentation, accessed December 9, 2025, <https://ollama.readthedocs.io/en/faq/>
5. llama3.1:8b - Ollama, accessed December 9, 2025, <https://ollama.com/library/llama3.1:8b>
6. ilsp/llama-krikri-8b-instruct - Ollama, accessed December 9, 2025, <https://ollama.com/ilsp/llama-krikri-8b-instruct:latest>
7. Llama-Krikri-Instruct-8B-Q4\_K\_M Model | MAX Builds - Code with Modular, accessed December 9, 2025, <https://builds.modular.com/models/Llama-Krikri-Instruct/8B-Q4_K_M>
8. bartowski/ilsp\_Llama-Krikri-8B-Instruct-GGUF - Hugging Face, accessed December 9, 2025, <https://huggingface.co/bartowski/ilsp_Llama-Krikri-8B-Instruct-GGUF>
9. Chroma - Docs by LangChain, accessed December 9, 2025, <https://docs.langchain.com/oss/python/integrations/vectorstores/chroma>
10. ChatOllama - Docs by LangChain, accessed December 9, 2025, <https://docs.langchain.com/oss/python/integrations/chat/ollama>
11. Best Chunking Strategies for RAG in 2025 - Firecrawl, accessed December 9, 2025, <https://www.firecrawl.dev/blog/best-chunking-strategies-rag-2025>
12. The Ultimate Guide to RAG Chunking Strategies - Agenta, accessed December 9, 2025, <https://agenta.ai/blog/the-ultimate-guide-for-chunking-strategies>
13. curl missing from Ollama Docker image, causing healthcheck failures #9781 - GitHub, accessed December 9, 2025, <https://github.com/ollama/ollama/issues/9781>