



Lenovo Validated Design: Hybrid AI 201 with Open- Source RAG

Version 1.0

Lower AI infrastructure costs
Optimize inference on CPU-
based infrastructure

Accelerate AI deployment
Reduce implementation
time through a validated
architecture

Simplify AI operations
Unify key components in
an integrated open-source
RAG stack

Maintain flexibility and control
Adopt an open, modular
foundation that evolves with
AI needs

Erik Robert
Abed Islam
Eric Page



Table of Contents

Introduction	1
Executive Summary	1
Intended Audience	1
Challenges and Opportunities	2
Business Challenges	2
Strategic Opportunities	2
Solution Overview	4
Solution Workflow	4
Ingestion pipeline (executed once per document)	5
Query pipeline (executed per user question)	5
Efficient AI Inference with CPU Acceleration.....	6
Solution Components	7
Hardware and Software	7
Lenovo SR630 V4.....	8
Deployment Overview.....	9
Hardware Platform	9
Compute Allocation	9
Orchestration	10
Deployment Procedure	11
Deployment Prerequisites.....	11
Deployment Workflow	11
Deployment Summary	14
Performance Validation	15
Solution Summary	20
Appendix A: Lenovo Bill of materials (BOM).....	21

Appendix B: Deployment Procedure - Details	22
1.0 Deployment Procedure	22
1.1 Install Kubernetes	22
1.2 Deploy OpenSearch.....	22
1.3 Deploy Docling.....	24
1.4 Deploy OVMS (Embedding and Reranking)	26
1.5 Launch vLLM	29
1.6 Deploy Open WebUI	29
2.0 Post-Deployment Validation	32
2.1 Service Health Checks.....	32
2.2 Functional Validation	32
Appendix C: RAG Pipeline Service Reference Implementation.....	33
1.0 Purpose and Scope	33
2.0 Service Source: rag_pipeline_service.py	33
3.0 Container Image	43
4.0 Kubernetes Manifest	44
5.0 Open WebUI Configuration	47
6.0 Document Ingestion and Management	48
7.0 Pipeline Feature Traceability.....	49
Appendix D: Abbreviations	50
Resources.....	51
Document history.....	52
Trademarks and special notices	53

Introduction

Executive Summary

Enterprise AI is moving rapidly from experimentation to production, but many organizations face high infrastructure costs, software licensing expenses, and operational complexity. Organizations need a cost-effective platform that enables them to deploy, manage, and scale AI workloads while maintaining control over their infrastructure and data.

The Lenovo Hybrid AI Platform 201 for Open-Source RAG combines Lenovo ThinkSystem infrastructure powered by Intel Xeon 6 processors with Intel Advanced Matrix Extensions, Ubuntu, Canonical Kubernetes, and open-source AI software. The validated architecture supports enterprise AI inference and Retrieval-Augmented Generation workloads while simplifying deployment and reducing infrastructure and software costs.

Designed around CPU-optimized inference, the platform enables organizations to deploy applicable AI workloads without dedicated GPU infrastructure. This approach improves hardware utilization, lowers total cost of ownership, and provides a practical foundation that can scale as workload requirements evolve.

The platform's open and modular architecture gives organizations the flexibility to adopt compatible AI models, frameworks, and tools without relying on a tightly coupled proprietary software stack. It supports use cases including AI assistants, enterprise search, knowledge management, and document-grounded question answering.

Intended Audience

This document is intended for business and technical stakeholders evaluating, designing, deploying, or operating enterprise AI and Retrieval-Augmented Generation solutions. The audience includes CIOs, CTOs, Chief Data and AI Officers, and business leaders responsible for AI strategy and investment, as well as enterprise architects, AI and machine learning engineers, platform and DevOps teams, and IT operations teams responsible for implementation and management. It is also relevant to independent software vendors and system integrators building or deploying RAG applications on Lenovo infrastructure.

Challenges and Opportunities

Business Challenges

As organizations move AI initiatives from experimentation into production, they must balance performance, cost, operational complexity, and long-term flexibility. Deploying enterprise AI requires more than selecting a foundation model; organizations must integrate infrastructure, orchestration, inference, retrieval, data processing, and user-facing services into a reliable operating environment.

Organizations commonly face the following challenges:

AI Deployment Costs

Enterprise AI environments can require significant investment in specialized infrastructure, software licensing, and ongoing platform management. Organizations need an approach that supports useful AI workloads while keeping deployment and operating costs manageable.

Platform Complexity

Enterprise RAG environments combine operating systems, container orchestration, inference engines, embedding and reranking models, vector databases, document-processing tools, and user interfaces. Integrating, configuring, and maintaining these components can require specialized expertise and create additional operational overhead.

Lengthy Deployment and Validation Cycles

Organizations often spend substantial time selecting components, designing infrastructure, validating interoperability, and tuning performance before an AI application can be deployed. Without a validated architecture, integration issues and repeated testing can delay production readiness and increase project risk.

Scaling with Predictable Performance

As the number of users, documents, and concurrent requests grow, organizations must scale infrastructure while maintaining acceptable response times and controlling costs. Without clear performance data and sizing guidance, teams may overprovision resources or deploy environments that cannot meet workload demand.

Enterprise Readiness

Production AI environments must provide secure access to enterprise data, reliable operations, consistent software management, and appropriate governance controls. Organizations must also maintain visibility into platform performance and ensure that AI services can be managed within existing IT and compliance practices.

Strategic Opportunities

The Lenovo Hybrid AI Platform 201 addresses these challenges through a validated, open architecture designed for enterprise AI inference and Retrieval-Augmented Generation (RAG) workloads.

Simplify AI Deployment and Operations

Ubuntu, Canonical Kubernetes, and validated open-source RAG components provide an integrated foundation for deploying and managing AI services. A documented configuration reduces the effort required to select, integrate, and validate individual components.

Enable Cost-efficient CPU-Based AI

Intel Xeon 6 processors with Intel Advanced Matrix Extensions support efficient CPU-based inference for applicable enterprise AI workloads. This enables organizations to deploy RAG applications without requiring dedicated GPU infrastructure while maintaining predictable performance within the validated operating range.

Maintain an Open and Flexible AI Foundation

The modular architecture allows organizations to adopt compatible models, frameworks, and tools as requirements evolve without depending on a tightly coupled proprietary AI platform.

Reduce Deployment Risk with a Validated Architecture

The Lenovo Hybrid AI Platform 201 provides a tested configuration with documented deployment procedures and measured performance characteristics. This gives organizations a repeatable starting point that reduces integration uncertainty and supports more informed workload sizing and implementation planning.

Solution Overview

The Lenovo Hybrid AI Platform 201 combines Lenovo compute and storage infrastructure with Ubuntu, Canonical Kubernetes, and open-source AI software to support enterprise inference and Retrieval-Augmented Generation workloads. The solution provides a flexible foundation for deploying and managing AI applications while allowing organizations to maintain control over their data and infrastructure.

In the validated implementation, users upload source documents that are extracted, divided into searchable chunks, and indexed in OpenSearch. When a user submits a question, the system retrieves and reranks the most relevant content before providing it to the language model as context. This approach grounds responses in enterprise source material, reduces the risk of hallucination, and supports traceability to the underlying documents.

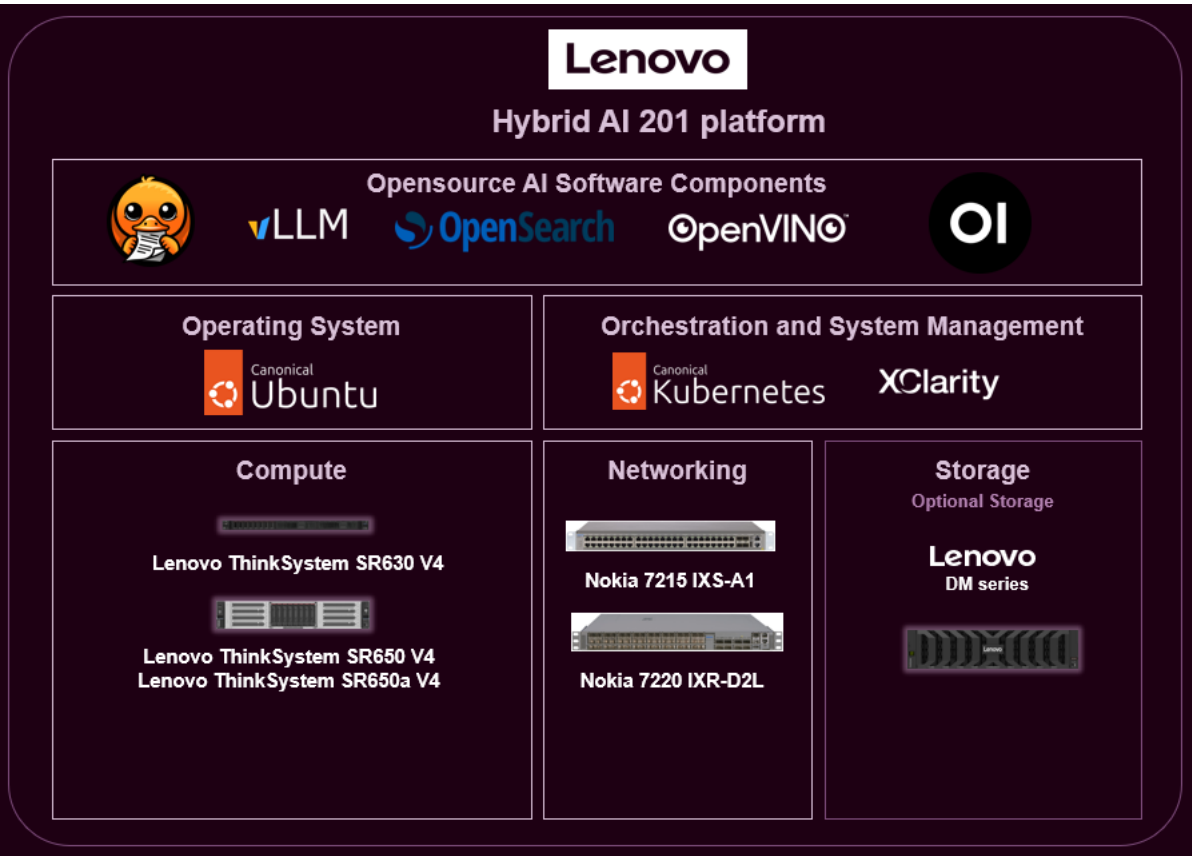


Figure 1. Hybrid AI 201 Platform with opensource software components

Solution Workflow

The system comprises two distinct pipelines: a one-time document ingestion pipeline, and a per-query retrieval-and-generation pipeline, each described below.

Ingestion pipeline (executed once per document)

1	Document Upload A source document is submitted through the front-end interface or the ingestion API.
2	Structure-Aware Extraction and Chunking Docling performs OCR and layout extraction and splits the document using a structure-aware chunker that preserves tables and section boundaries, avoiding the common failure mode in which a table is split mid-row by naive character-based chunking.
3	Embedding Each chunk is converted to a dense vector representation by the embedding model, served via OpenVINO Model Server (OVMS).
4	Indexing Chunks, vectors, and source metadata are written to OpenSearch and tagged with their originating document to support scoped retrieval at query time.

Query pipeline (executed per user question)

1	Query Embedding The user's question is embedded using the same model used at ingestion time, so that both live in the same vector space.
2	Hybrid Retrieval OpenSearch is queried by two methods in parallel — dense k-NN vector search and sparse BM25 keyword search — with results combined by Reciprocal Rank Fusion and, where a specific document is in scope, filtered to that source to prevent cross-document contamination.
3	Reranking Candidate chunks are re-scored for relevance by a dedicated cross-encoder reranker, also served via OVMS, and the top-ranked chunks are retained as final context.

4	Generation The selected context and the original question are assembled into a prompt and submitted to the LLM, served via vLLM, which streams back a grounded answer.
---	---

Efficient AI Inference with CPU Acceleration

This solution uses Intel Xeon 6 processors with Intel Advanced Matrix Extensions (Intel AMX) to provide efficient, scalable AI inference on industry-standard CPU infrastructure. Key benefits include:

- **Cost-efficient AI inference**
Supports AI inference on CPU-based infrastructure, helping organizations optimize existing infrastructure investments and reduce reliance on specialized accelerators.
- **Improved performance per watt**
Accelerates matrix-intensive AI operations to improve throughput and energy efficiency across supported inference workloads.
- **Simplified development and deployment**
Supports widely used AI frameworks and optimization tools, helping teams deploy compatible models without extensive application changes.

Outcome: A scalable, CPU-based inference platform designed to lower total cost of ownership while simplifying enterprise AI deployment and operations.

Solution Components

Hardware and Software

The Hybrid AI 201 software layer is built around an open-source foundation, giving organizations greater visibility into the components running in their environment and reducing dependence on tightly coupled proprietary AI platforms. At its core, the solution pairs vLLM, a high-performance inference engine, with Intel OpenVINO Model Server to run embedding and reranking directly on the Intel Xeon 6 processors in the Lenovo ThinkSystem SR630 V4, delivering enterprise-grade AI performance without a GPU.

OpenSearch provides fast hybrid retrieval across an organization's indexed documents, while Docling uses structure-aware processing to help preserve tables, headings, and document organization as source content is transformed into a searchable knowledge base. Open WebUI gives users a clean, familiar interface for uploading documents and interacting with their enterprise data.

Most services run as containerized workloads on Canonical Kubernetes, while vLLM runs directly on the host with NUMA-aware CPU binding to optimize inference performance across both processor sockets. Together, these components deliver a complete, validated, and extensible RAG foundation that accelerates deployment, simplifies operations, and provides the flexibility to evolve as AI requirements grow.

Table 1. Software Components

Layer	Component	Role
LLM inference	vLLM — Ministral-3-14B-Instruct (BF16, unquantized)	Generates the final answer; tensor-parallel across both CPU sockets
Embedding	OpenVINO Model Server — Qwen3-Embedding-0.6B	Converts text to vectors for both ingestion and query time
Reranking	OpenVINO Model Server — bge-reranker-base	Re-scores retrieved candidates for final relevance ranking
Vector database	OpenSearch	Stores chunk vectors and text; serves hybrid k-NN + BM25 retrieval
Document ingestion	Docling (asynchronous, structure-aware hybrid chunking)	OCR, layout extraction, and table-preserving chunking
Container orchestration	Kubernetes (Canonical distribution), containerd runtime	Deploys and manages every service; no Docker dependency
User interface	Open WebUI	Chat interface with a document Knowledge Base

Lenovo SR630 V4



Figure 2 - Lenovo ThinkSystem SR630 V4

The Lenovo ThinkSystem SR630 V4 is a dense, dual-socket rack server built on Intel Xeon 6 processors, offering high core counts and strong memory bandwidth in a compact 1U form factor. Its balance of compute density and efficiency makes it well suited to CPU-based AI inference workloads, giving customers a proven, enterprise-grade foundation for the Hybrid AI 201 platform without the cost, power, and supply constraints of GPU-based infrastructure.

If customers plan on upgrading to GPU-based infrastructure down the road, Lenovo offers multiple 2U server variants share the same Intel Xeon 6 CPU family:



Figure 3 - Lenovo ThinkSystem SR650 V4



Figure 4 – Lenovo ThinkSystem SR650a V4

For more information about these servers, visit:

<https://lenovopress.lenovo.com/lp1971-thinksystem-sr630-v4-server>

<https://lenovopress.lenovo.com/lp2127-thinksystem-sr650-v4-server>

<https://lenovopress.lenovo.com/lp2128-thinksystem-sr650a-v4-server>

Deployment Overview

The Lenovo Hybrid AI Platform 201 uses a modular deployment approach that combines validated Lenovo infrastructure with Ubuntu, Canonical Kubernetes, and an open-source AI software stack. This approach provides a flexible, repeatable foundation for deploying and managing enterprise AI inference and RAG workloads.

Hardware Platform

This solution is positioned for deployment on the following platform:

Table 2. Hardware Components

Specification	Value
Server	Lenovo ThinkSystem SR630 V4
Processors	2 × Intel Xeon 6737P, 32 cores per socket
Total physical cores	64
System memory	256 GB
Storage	1 TB SSD
GPUs	None. Inference is CPU-only throughout.

Latency validation reported in our results was performed on a test platform with an equivalent 64-core allocation (2 × Intel Xeon 6787P, 86 cores per socket, with utilisation constrained to 64 cores). The Xeon 6737P and 6787P are both members of the Intel Xeon 6 P-core (Granite Rapids) family; per-core clock speed and cache differ between the two parts, and we recommend confirming results on the specific target SKU before final sizing.

Compute Allocation

The platform's 64 physical cores are allocated across services as shown in Table 4. This allocation reflects a deliberate engineering decision, not a default: embedding and reranking were sized generously after per-stage instrumentation identified OVMS as the binding constraint on latency under load.

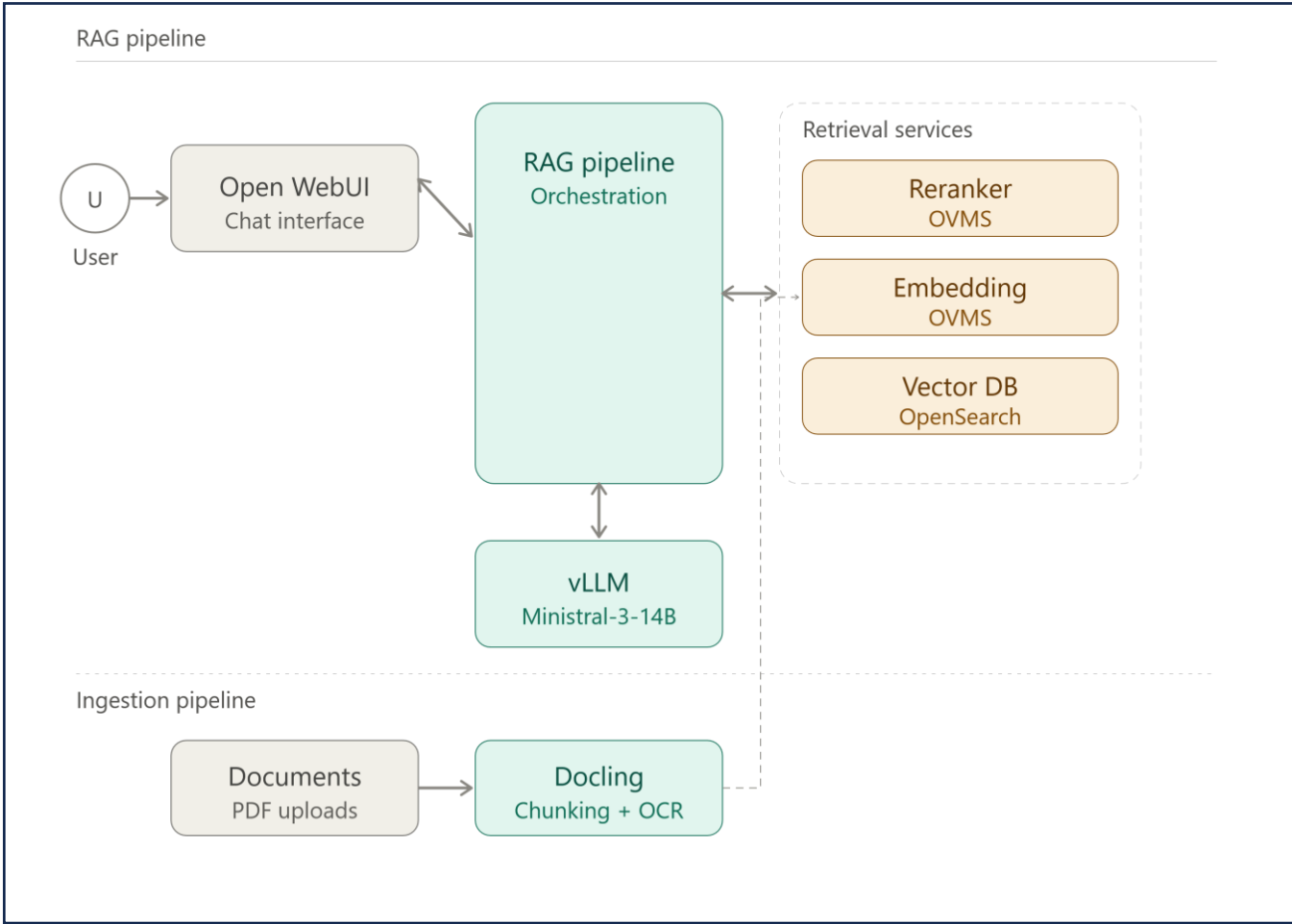
Table 3. Compute allocation across the platform's 64 physical cores

Service	Cores	Share of platform
vLLM (LLM inference)	36	56%

OpenVINO Model Server (embedding and reranking)	16	25%
Docling (ingestion)	4	6%
OpenSearch	2	3%
RAG Pipeline Service	2	3%
Front end (Open WebUI)	1	2%
Reserved headroom (OS and Kubernetes overhead)	3	5%

Orchestration

All services run as containerized workloads on a single-node Canonical Kubernetes cluster using the containerd runtime, with no Docker dependency. Images are built directly with nerdctl and loaded into the cluster's local image store, which avoids an external registry round-trip. The LLM serving process is pinned to explicit CPU core ranges using NUMA-aware host tooling, which keeps memory access local to each socket and avoids the cross-socket latency penalty that naive scheduling would otherwise introduce.



Deployment Procedure

The Lenovo Hybrid AI Platform 201 uses a modular deployment approach that combines Ubuntu Server, Canonical Kubernetes, and open-source AI components on validated Lenovo ThinkSystem infrastructure. The deployment establishes the services required for document ingestion, retrieval, reranking, and CPU-based language model inference.

The validated test configuration uses a single-node Canonical Kubernetes cluster for most containerized services, while vLLM runs directly on the host to enable NUMA-aware CPU allocation. This configuration provides predictable resource allocation across the RAG pipeline while maintaining a flexible architecture that can be adapted as workload requirements evolve.

Deployment Prerequisites

Before deploying the software stack, verify that the target system meets the validated hardware and software requirements described in this document. The validated configuration uses a Lenovo ThinkSystem SR630 V4 with two Intel Xeon 6737P processors, 256 GB of memory, and CPU-only inference.

The deployment requires Ubuntu Server, Canonical Kubernetes with the containerd runtime, network connectivity to obtain the required software images and models, and sufficient persistent storage for OpenSearch indexes and AI model files.

The CPU topology of the target server should also be confirmed before deployment. NUMA placement and CPU affinity are important considerations for vLLM because the validated configuration distributes inference processing across both processor sockets.

Deployment Workflow

The solution is deployed in the following sequence:

1. Deploy Canonical Kubernetes

Install and initialize Canonical Kubernetes and verify that the cluster is operational. Create a dedicated namespace for the RAG services. Kubernetes provides the orchestration environment for OpenSearch, Docling, OpenVINO Model Server, and Open WebUI.

2. Deploy OpenSearch

Deploy OpenSearch with persistent storage to provide the vector and lexical search capabilities used by the RAG pipeline. OpenSearch stores document chunks, vector embeddings, and associated metadata and supports both k-NN vector search and BM25 keyword search.

Configure appropriate authentication credentials and allocate the validated CPU and memory resources before making the service available to other RAG components.

3. Deploy Docling

Deploy Docling as the document-ingestion service. Docling performs document extraction, OCR, layout processing, and structure-aware chunking before content is submitted for embedding and indexing.

The validated configuration allocates four CPU cores to Docling. Because document ingestion occurs asynchronously and independently of interactive inference, resources can be adjusted based on ingestion volume and document complexity.

4. Deploy OpenVINO Model Server

Deploy OpenVINO Model Server (OVMS) to host the embedding and reranking models. In the validated configuration, OVMS serves the Qwen3 embedding model and the BGE reranker.

Allocate the validated CPU and memory resources and configure the model repository. The validated configuration assigns 16 CPU cores to OVMS because testing identified embedding and reranking as significant contributors to retrieval latency under concurrent workloads.

5. Deploy vLLM

Deploy vLLM to serve the language model used for response generation. Unlike the other solution components, vLLM runs directly on the host rather than as a Kubernetes workload. This enables explicit NUMA-aware CPU binding and helps maintain local memory access across the two processor sockets.

The validated implementation uses Ministral-3-14B-Instruct with BF16 inference and allocates 36 physical CPU cores to vLLM. CPU affinity and NUMA configuration should be adjusted to match the topology of the target server.

6. Deploy Open WebUI

Deploy Open WebUI as the user-facing interface for the RAG solution. Configure Open WebUI to communicate with vLLM for language-model inference, OVMS for embeddings, OpenSearch for document retrieval, and Docling for document processing.

Use Kubernetes service names rather than individual pod IP addresses when connecting Kubernetes-hosted services so that service connectivity remains consistent if pods are restarted or rescheduled.

Configure document chunk size, overlap, retrieval parameters, authentication, and service credentials according to the validated configuration and customer requirements.

Post-Deployment Validation

After deployment, verify the health and connectivity of each component before making the environment available to users.

Confirm that all Kubernetes-hosted services are running successfully and that vLLM is responding to inference requests. Verify connectivity between Open WebUI, OpenSearch, Docling, OVMS, and vLLM.

Perform an end-to-end functional test by uploading a representative document through Open WebUI. Confirm that the document is successfully processed and indexed, then submit a question whose answer is contained in the document. Verify that the system retrieves the appropriate source content and generates an accurate, grounded response. The current LVD uses a table-containing document for this validation and checks that the response returns the correct value from the table.

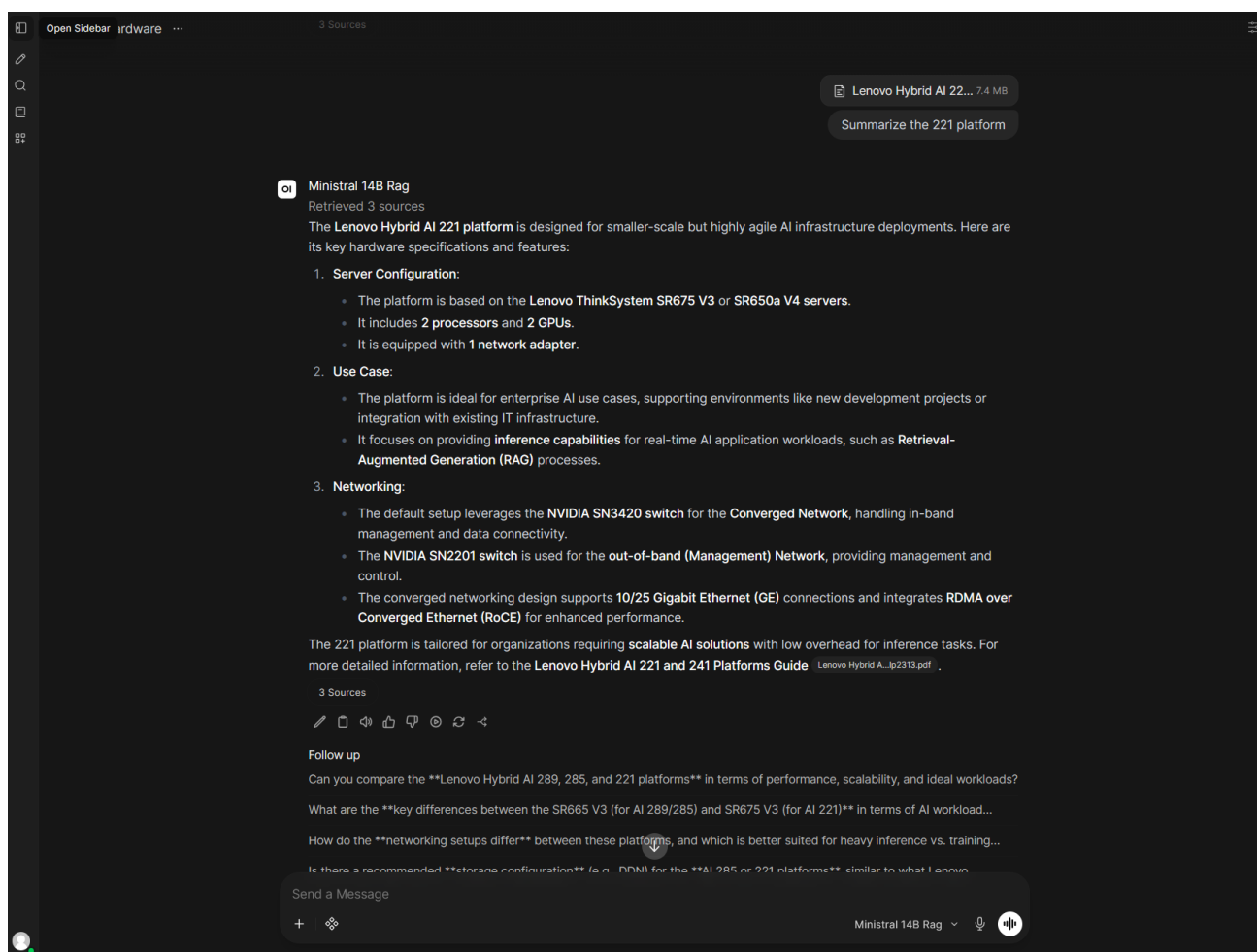


Figure 6 – Successful Open WebUI deployment, document ingestion, and response.

Key Deployment Considerations

The validated architecture incorporates several deployment considerations that should be preserved when

adapting the solution:

- **CPU and NUMA optimization:** Maintain appropriate CPU affinity for vLLM to minimize cross-socket memory access and support predictable inference performance.
- **Persistent data:** Use persistent storage for OpenSearch indexes and model repositories so that data and models survive workload restarts.
- **Service discovery:** Use Kubernetes service names for communication between containerized components rather than dynamically assigned pod addresses.
- **Resource isolation:** Allocate CPU and memory resources deliberately across inference, embedding, reranking, ingestion, and retrieval services to reduce contention.
- **Security:** Replace default credentials, protect model and data endpoints, and apply enterprise access-control and network-security policies before production use.
- **Sizing:** Treat the validated configuration as a tested reference point. Production sizing should account for model size, token lengths, concurrent users, document volumes, and required response times.

Deployment Summary

The validated deployment demonstrates a repeatable approach for implementing Lenovo Hybrid AI Platform 201 with Open-Source RAG on CPU-based Lenovo ThinkSystem infrastructure. The configuration combines Canonical Kubernetes for containerized RAG services with host-based, NUMA-aware vLLM inference and defined CPU resource allocation across inference, embedding, reranking, retrieval, and ingestion services.

The result is a functional end-to-end RAG environment supporting document ingestion, hybrid retrieval, reranking, and grounded language-model responses. The validated configuration provides a tested reference point rather than a universal production sizing recommendation; resource requirements should be adjusted based on model size, document volume, concurrency, token lengths, and response-time requirements.

The following section evaluates the performance of this configuration under increasing concurrent load and provides practical guidance for sizing and horizontal scaling.

Performance Validation

Validation Objective

Performance testing was conducted to determine whether the Lenovo Hybrid AI Platform 201 can deliver responsive enterprise RAG and LLM inference using Intel Xeon 6 CPU infrastructure. Testing focused on end-user response latency, concurrent request handling, and the latency contribution of the major RAG pipeline stages.

Validated Configuration

Table 4. Components and Validated Configuration

Component	Validated configuration
LLM	Ministral-3-14B-Instruct, BF16
LLM serving	vLLM
Embedding	Qwen3-Embedding-0.6B / OpenVINO Model Server (OVMS)
Reranking	bge-reranker-base / OpenVINO Model Server (OVMS)
Retrieval	OpenSearch hybrid vector + BM25
Compute	2 × Xeon 6787P CPUs*
Accelerator	CPU only; no GPU
Dataset	84 PDF documents embedded into approximately 67,000 vector chunks

The results in this section were measured through the RAG Pipeline Service described in Appendix C, the service that Open WebUI connects to as its model endpoint. Each request passed through query embedding, hybrid k-NN and BM25 retrieval, Reciprocal Rank Fusion, document-scoped filtering, cross-encoder reranking, and generation. The Open WebUI configuration in Appendix B uses Open WebUI's built-in retrieval, which does not enable these features by default; deploy the service in Appendix C to reproduce the validated behavior. Across 4,944 timed requests, 3 (0.06%) ended with an incomplete streaming response, all at 64 or more concurrent requests; no errors occurred within the recommended operating range.

***Validation notes:** Performance testing used an Intel Xeon 6787P-based system constrained to the same 64-core allocation used by the reference configuration. Results should therefore be treated as representative of the architecture rather than guaranteed performance for the Xeon 6737P production configuration. Final sizing should be validated on the intended server configuration.

Functional Validation Context

Before performance benchmarking, the deployed environment was functionally validated by confirming service health, document ingestion, table-based retrieval, grounded response generation, and source citation. The performance tests below focus on how that validated workflow behaves as concurrent load increases.

Performance Validation Criteria

Testing covered multiple prompt and response lengths ranging from 128-token input/output workloads through longer 256-input/1024-output workloads.

Table 5. Validation Criteria

Metric	Validation target	What it measures
Time to First Token (TTFT)	<5 seconds	How quickly the user begins receiving a response
Time per Output Token (TPOT)	<100 ms/token	How quickly the response is generated once streaming begins

Performance Validation Results

Table 6. Validation Results

Area	Result	What it means
Interactive response	16 concurrent requests = average TTFT ranged from 2.74 s to 3.67 s across workloads, well within the 5 s target, and TPOT ranged from 102.7 to 104.1 ms/token	The CPU-based platform provides responsive RAG interaction at 16 concurrent users, with TTFT holding comfortably under target
Concurrency	TPOT tracked closely to the 100 ms/token target through 16 concurrent requests, at most 4.1 ms over; degradation became significant only at 24 concurrent requests (126.9 to 128.5 ms/token)	16 concurrent requests is a practical operating point: TTFT is met with margin, and TPOT lands within a few milliseconds of the target, well before the sharp degradation seen at higher concurrency

Vector search	k-NN remained approximately 42-58 ms across the tested load range	Vector retrieval speed remains stable with concurrency but will increase with dataset size.
Keyword search	BM25 remained approximately 30-50 ms across the tested load range	Lexical retrieval remains stable with concurrency but will increase with dataset size
Embedding	Latency increased as concurrency increased, especially at high load	Embedding capacity should be considered when sizing for higher concurrency
Reranking	The cross-encoder reranker was the largest retrieval-stage latency contributor	Reranking is the first retrieval service to consider scaling independently

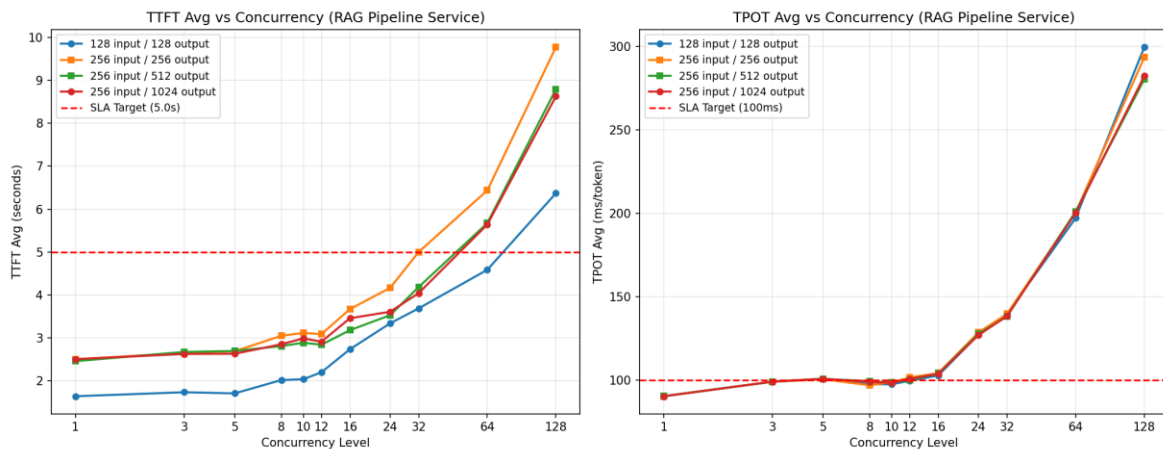


Figure 7. Time to first token (left) and time per output token (right) by concurrency level.

Optimal Concurrency Range (1-16): The system comfortably met both TTFT and TPOT SLAs. At C=1 (128/128), average TTFT was 1.643s (p95 2.093s) and TPOT was 90.2ms. At C=16, average TTFT remained well within bounds at 2.738s to 3.673s across tiers, with TPOT ranging 102.7ms to 104.1ms.

TTFT Breach Threshold (C > 32): The system held under the 5-second SLA through C=32 across all tiers (3.689s to 4.185s), with 256/256 reaching the threshold at 5.001s. The breach occurred at C=64 for the 256-input tiers (5.640s to 6.435s), while the lighter 128/128 configuration held through C=64 (4.587s), only breaching at C=128 (6.371s).

TPOT SLA Limits (C > 16): TPOT remained close to the 100ms/token target through C=16 across all tiers (90.2ms to 104.1ms). Moderate saturation occurred at C=24 (TPOT ~126.9ms to 128.5ms), with severe queueing delay observed at extreme load (C=128, TPOT reaching 280.4ms to 299.7ms).

Takeaway: Response latency remains within the validation targets at moderate concurrency. As concurrent requests increase, CPU contention raises TTFT and TPOT, indicating when additional worker capacity should be added.

RAG Pipeline Stage Latency

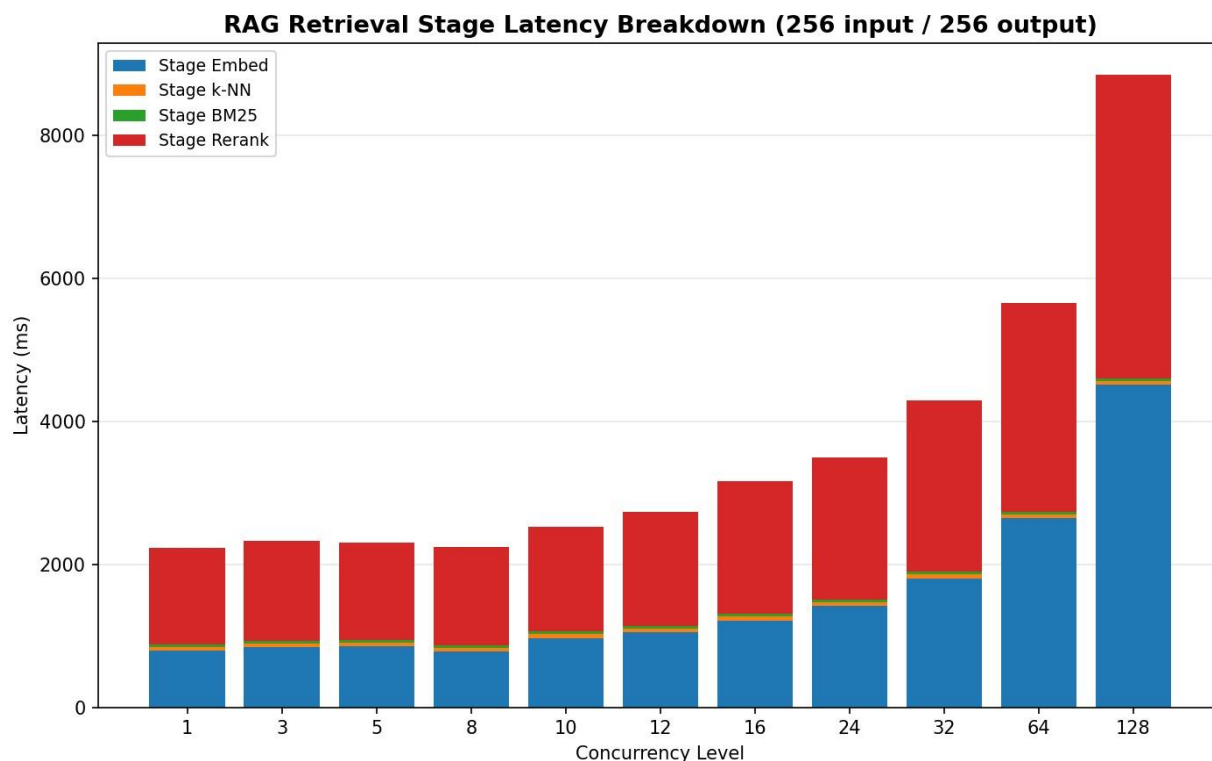


Figure 8. RAG Retrieval Stage Latency by Concurrency Level

Takeaway: Vector and keyword retrieval remain consistently low latency. Most retrieval-stage latency comes from embedding and reranking, making these services the primary optimization and scaling targets.

Sizing and Scale-out Guidance

Recommended operating range: Up to approximately 16 concurrent requests per instance when maintaining the tested TPOT target is required. Higher concurrency can be supported by scaling additional worker capacity.

Scale-out guidance: The validated single-node configuration establishes the performance baseline. For larger deployments, the architecture can scale horizontally by adding Lenovo worker nodes and distributing inference or retrieval services across the cluster.

Embedding and reranking capacity: The validation results support the compute allocation described in the Deployment Overview. Embedding and reranking become the primary retrieval-stage constraints as concurrency increases, which is why OVMS receives a relatively large share of the platform CPU resources.

Validation Outcome

The Lenovo Hybrid AI Platform 201 successfully demonstrated an end-to-end enterprise RAG workflow using Intel Xeon 6 CPU infrastructure. Under the tested configuration, the platform delivered interactive response performance through 16 concurrent requests, with average TTFT well within the 5-second target and TPOT within approximately 4 ms of the 100 ms/token validation target.

Vector and keyword retrieval remained consistently low latency as load increased. Embedding and cross-encoder reranking became the primary retrieval-stage constraints at higher concurrency, providing practical guidance for resource allocation and horizontal scaling. These results demonstrate that CPU-based inference can provide a practical foundation for applicable RAG workloads while allowing organizations to scale additional worker capacity as user demand grows.

Solution Summary

The Lenovo Hybrid AI Platform 201 with Open-Source RAG provides a validated foundation for enterprise AI inference and Retrieval-Augmented Generation on CPU-based Lenovo ThinkSystem infrastructure. The solution combines Intel Xeon 6 processors with Intel Advanced Matrix Extensions, Ubuntu Server, Canonical Kubernetes, and open-source RAG components to deliver an integrated and flexible AI environment.

Its open and modular architecture allows organizations to adopt compatible models, frameworks, and AI tools as requirements evolve without depending on a tightly coupled proprietary AI platform. Canonical Kubernetes provides consistent orchestration for containerized services, while CPU-optimized inference enables applicable RAG workloads to run without dedicated GPU infrastructure.

The validated design provides organizations with a repeatable starting point to:

- Deploy enterprise RAG applications on Lenovo ThinkSystem infrastructure.
- Run applicable AI inference workloads on CPU-based infrastructure.
- Maintain control over enterprise data and deployment location.
- Adapt models, tools, and services through an open architecture.
- Scale resources as workload and concurrency requirements grow.

By integrating the infrastructure, orchestration, retrieval, and inference components into a documented design, Hybrid AI Platform 201 reduces integration complexity and provides a practical path from AI experimentation to operational deployment.

Appendix A: Lenovo Bill of materials (BOM)

SR630 V4

Part number	Product Description	Qty
7DG9CTO1WW	Server: ThinkSystem SR630 V4-3yr Base Warranty	1
C1XE	ThinkSystem 1U V4 10x2.5" Chassis	1
C3JB	ThinkSystem General Computing - Power Efficiency	1
BVGL	Data Center Environment 30 Degree Celsius / 86 Degree Fahrenheit	1
C5QX	Intel Xeon 6737P 32C 270W 2.9GHz Processor	2
C1XJ	ThinkSystem 1U V4 Performance Heatsink	2
C0U9	ThinkSystem 32GB TruDDR5 6400MHz (1Rx4) RDIMM	8
5977	Select Storage devices - no configured RAID required	1
CGP9	ThinkSystem 2.5" U.2 CD9P 1.92TB Read Intensive NVMe PCIe 5.0 x4 HS SSD	1
C2NN	ThinkSystem 1U V4 4x2.5" NVMe Gen5 Backplane	1
CC7H	ThinkSystem M.2 B350i-2i NVMe Enablement Kit	1
C287	ThinkSystem M.2 VA 960GB Read Intensive NVMe NHS SSD	2
B5T1	ThinkSystem Broadcom 5719 1GbE RJ45 4-port OCP Ethernet Adapter	1
BNWM	ThinkSystem Broadcom 57504 10/25GbE SFP28 4-port PCIe Ethernet Adapter	1
BK1J	ThinkSystem Broadcom 57508 100GbE QSFP56 2-Port PCIe 4 Ethernet Adapter	1
C1YH	ThinkSystem SR630 V4 x16/x16 PCIe Gen5 Cable Riser 1	1
C9AR	ThinkSystem SR630 V4 Full Height+Low Profile Riser1 Cage v2	1
CE84	ThinkSystem 2000W 230V Titanium CRPS Premium Hot-Swap Power Supply v2	2
6400	2.8m, 13A/100-250V, C13 to C14 Jumper Cord	2
C1YT	ThinkSystem 1U V4 Performance Fan Module	4
C1YP	ThinkSystem 1U V4 Standard Media Bay	1
C2DH	ThinkSystem Toolless Slide Rail Kit V4	1
BPKR	TPM 2.0	1

Appendix B: Deployment Procedure - Details

1.0 Deployment Procedure

Validation for this guide was performed on a test platform with an equivalent 64-core allocation, carved out of a 172-core system, due to hardware availability at the time of testing. Core-specific values referenced in this section, such as NUMA bindings and thread counts, should be confirmed against the actual target platform's topology before production deployment.

1.1 Install Kubernetes

Install the Canonical Kubernetes distribution, which uses containerd as its runtime.

```
sudo snap install k8s --classic
sudo k8s bootstrap
sudo k8s status --wait-ready
```

Alias kubectl to the bundled, version-matched binary:

```
alias kubectl='sudo k8s kubectl'
```

Create the working namespace for all RAG services:

```
kubectl create namespace rag-poc
```

1.2 Deploy OpenSearch

Deploy OpenSearch as a StatefulSet (not a Deployment) to provide stable storage and network identity. Save the following as opensearch-statefulset.yaml. We recommend setting a strong, unique value for OPENSEARCH_INITIAL_ADMIN_PASSWORD before applying:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: opensearch
  namespace: rag-poc
spec:
```

```

serviceName: opensearch

replicas: 1

selector:
  matchLabels:
    app: opensearch

template:
  metadata:
    labels:
      app: opensearch

spec:
  containers:
    - name: opensearch
      image: opensearchproject/opensearch:2.19.0
      env:
        - name: discovery.type
          value: single-node
        - name: OPENSEARCH_INITIAL_ADMIN_PASSWORD
          value: "<set-a-strong-password>"
        - name: OPENSEARCH_JAVA_OPTS
          value: "-Xms2g -Xmx2g"
      ports:
        - containerPort: 9200
      resources:
        requests: {cpu: "2", memory: "8Gi"}
        limits: {cpu: "2", memory: "8Gi"}
      volumeMounts:
        - name: data
          mountPath: /usr/share/opensearch/data
  volumeClaimTemplates:
    - metadata: {name: data}

```

```

    spec:
      accessModes: ["ReadWriteOnce"]
      resources: {requests: {storage: 20Gi}}
---
apiVersion: v1
kind: Service
metadata:
  name: opensearch
  namespace: rag-poc
spec:
  selector: {app: opensearch}
  ports:
    - port: 9200
      targetPort: 9200

```

Apply the manifest and set the resource allocation:

```

kubectl apply -f opensearch-statefulset.yaml -n rag-poc
kubectl set resources statefulset opensearch -n rag-poc \
  --requests=cpu=2,memory=8Gi --limits=cpu=2,memory=8Gi

```

Verify the pod reaches Running state before proceeding:

```

kubectl get pods -n rag-poc -l app=opensearch -w

```

1.3 Deploy Docling

Deploy Docling for asynchronous, structure-aware document ingestion. Save the following as docling-deployment.yaml:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: docling

```

```

    namespace: rag-poc
spec:
  replicas: 1
  selector:
    matchLabels: {app: docling}
  template:
    metadata:
      labels: {app: docling}
    spec:
      containers:
        - name: docling
          image: quay.io/docling-project/docling-serve-cpu:latest
          ports:
            - containerPort: 5001
          resources:
            requests: {cpu: "4", memory: "8Gi"}
            limits: {cpu: "4", memory: "8Gi"}
---
apiVersion: v1
kind: Service
metadata:
  name: docling
  namespace: rag-poc
spec:
  selector: {app: docling}
  ports:
    - port: 5001
      targetPort: 5001

```

Apply the manifest, set a generous synchronous-wait ceiling since some downstream integrations call the synchronous endpoint, and set the resource allocation:

```
kubectl apply -f docling-deployment.yaml -n rag-poc

kubectl set env deployment/docling -n rag-poc \

    DOCLING_SERVE_MAX_SYNC_WAIT=1800

kubectl set resources deployment docling -n rag-poc \

    --requests=cpu=4,memory=8Gi --limits=cpu=4,memory=8Gi
```

1.4 Deploy OVMS (Embedding and Reranking)

OVMS serves both the embedding and reranking models from a single instance. The image runs as a non-root user (UID 5000), so a permission-fix init container is required before the model volume is writable. Save the following as `ovms-deployment.yaml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ovms-models
  namespace: rag-poc
spec:
  accessModes: ["ReadWriteOnce"]
  resources: {requests: {storage: 10Gi}}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ovms
  namespace: rag-poc
spec:
  replicas: 1
  selector:
    matchLabels: {app: ovms}
  template:
    metadata:
```

```

labels: {app: ovms}

spec:
  initContainers:
    - name: fix-permissions
      image: busybox:1.36
      command: ["sh", "-c", "chown -R 5000:5000 /models"]
      volumeMounts:
        - name: models
          mountPath: /models
      securityContext: {runAsUser: 0}
  containers:
    - name: ovms
      image: openvino/model_server:latest
      command: ["/ovms/bin/ovms"]
      args:
        - "--config_path=/models/config.json"
        - "--rest_port=8000"
        - "--port=9000"
      ports:
        - containerPort: 8000
        - containerPort: 9000
      resources:
        requests: {cpu: "16", memory: "48Gi"}
        limits: {cpu: "16", memory: "48Gi"}
      volumeMounts:
        - name: models
          mountPath: /models
  volumes:
    - name: models
      persistentVolumeClaim: {claimName: ovms-models}

```

```

---
apiVersion: v1
kind: Service
metadata:
  name: ovms
  namespace: rag-poc
spec:
  selector: {app: ovms}
  ports:
    - {name: rest, port: 8000, targetPort: 8000}
    - {name: grpc, port: 9000, targetPort: 9000}

```

Apply the manifest and set the resource allocation:

```

kubectl apply -f ovms-deployment.yaml -n rag-poc
kubectl set resources deployment ovms -n rag-poc \
  --requests=cpu=16,memory=48Gi --limits=cpu=16,memory=48Gi

```

Pull and register the embedding model:

```

kubectl exec -n rag-poc deploy/ovms -- /ovms/bin/ovms --pull \
  --source_model OpenVINO/Qwen3-Embedding-0.6B-int8-ov \
  --model_repository_path /models --task embeddings --pooling LAST

```

Pull and register the reranking model:

```

kubectl exec -n rag-poc deploy/ovms -- /ovms/bin/ovms --pull \
  --source_model OpenVINO/bge-reranker-base-fp16-ov \
  --model_repository_path /models --task rerank

```

Set the stream count on both models to match the allocated core count, then restart the pod to apply:

```

kubectl exec -n rag-poc deploy/ovms -- sed -i \
  's/"NUM_STREAMS": "1"/"NUM_STREAMS": "16"/' \

```

```

/models/OpenVINO/Qwen3-Embedding-0.6B-int8-ov/graph.pbtxt
kubectl exec -n rag-poc deploy/ovms -- sed -i \
's/"NUM_STREAMS": "1"/"NUM_STREAMS": "16"/' \
/models/OpenVINO/bge-reranker-base-fp16-ov/graph.pbtxt
kubectl delete pod -n rag-poc -l app=ovms

```

1.5 Launch vLLM

vLLM runs as a host-pinned process rather than a Kubernetes pod, to allow explicit NUMA-aware core binding across both sockets:

```

export LD_PRELOAD="/usr/lib/x86_64-linux-gnu/libtcmalloc_minimal.so.4:\
/root/vllm-cpu-env/lib/libiomp5.so"
export VLLM_CPU_KVCACHE_SPACE=64
export VLLM_CPU_OMP_THREADS_BIND="0-17|32-49"

numactl --cpunodebind=0,1 --membind=0,1 \
vllm serve /path/to/Ministral-3-14B-Instruct-2512-BF16 \
--tokenizer_mode mistral --config_format mistral \
--load_format mistral --dtype=bfloat16 \
--tensor-parallel-size 2 --max-model-len 32768 \
--port 8000

```

1.6 Deploy Open WebUI

Deploy Open WebUI and configure it to use the OVMS embedding endpoint and OpenSearch as the vector store. Note that in-cluster service DNS names (<service>.<namespace>.svc.cluster.local) are used throughout rather than pod IP addresses, since ClusterIPs are dynamically assigned and will differ on every deployment. Save the following as openwebui-deployment.yaml, replacing <vllm-host-ip> with the IP address of the host running vLLM (Section 3.5) and setting a strong value for OPENSEARCH_PASSWORD matching what was set in Section 3.2:

Note: this configuration uses Open WebUI's built-in retrieval. To deploy the retrieval pipeline used during performance validation, see Appendix C.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: openwebui
  namespace: rag-poc
spec:
  replicas: 1
  selector:
    matchLabels: {app: openwebui}
  template:
    metadata:
      labels: {app: openwebui}
    spec:
      containers:
        - name: openwebui
          image: ghcr.io/open-webui/open-webui:main
          env:
            - name: OPENAI_API_BASE_URL
              value: "http://<vllm-host-ip>:8000/v1"
            - {name: WEBUI_AUTH, value: "true"}
            - {name: VECTOR_DB, value: "opensearch"}
            - name: OPENSEARCH_URI
              value: "http://opensearch.rag-poc.svc.cluster.local:9200"
            - {name: OPENSEARCH_USERNAME, value: "admin"}
            - name: OPENSEARCH_PASSWORD
              value: "<set-matching-strong-password>"
            - {name: RAG_EMBEDDING_ENGINE, value: "openai"}
            - name: RAG_OPENAI_API_BASE_URL
              value: "http://ovms.rag-poc.svc.cluster.local:8000/v1"
            - {name: RAG_EMBEDDING_MODEL, value: "Qwen3-Embedding-0.6B"}

```

```

      - {name: CONTENT_EXTRACTION_ENGINE, value: "docling"}
      - name: DOCLING_SERVER_URL
        value: "http://docling.rag-poc.svc.cluster.local:5001"
      - {name: CHUNK_SIZE, value: "2000"}
      - {name: CHUNK_OVERLAP, value: "200"}
      - {name: RAG_TOP_K, value: "6"}
    ports:
      - containerPort: 8080
    resources:
      requests: {cpu: "1", memory: "2Gi"}
      limits: {cpu: "1", memory: "2Gi"}
---
apiVersion: v1
kind: Service
metadata:
  name: openwebui
  namespace: rag-poc
spec:
  selector: {app: openwebui}
  type: NodePort
  ports:
    - {port: 8080, targetPort: 8080, nodePort: 30080}

```

Apply the manifest:

```
kubectl apply -f openwebui-deployment.yaml -n rag-poc
```

Setting an explicit chunk size is recommended: Open WebUI's default chunk size is small enough to split tables and other structured content across chunk boundaries, which degrades retrieval accuracy on table-heavy documents.

2.0 Post-Deployment Validation

2.1 Service Health Checks

Confirm every service is running before proceeding to functional validation:

```
kubect1 get pods -n rag-poc
```

All pods should report a single restart count of zero and READY 1/1. Confirm vLLM separately, since it runs outside Kubernetes:

```
curl -s http://localhost:8000/v1/completions \  
  -H "Content-Type: application/json" \  
  -d '{"model": "<model-path>", "prompt": "test", "max_tokens": 5}'
```

2.2 Functional Validation

1. Upload a test document containing at least one table through Open WebUI.
2. Confirm ingestion completes and ask a question whose answer is contained in the uploaded table.
3. Confirm the response correctly cites the source document and returns the correct value from the table.

Appendix C: RAG Pipeline Service Reference Implementation

1.0 Purpose and Scope

The performance results in the Performance Validation section were measured through this service. It implements query embedding, hybrid k-NN and BM25 retrieval, Reciprocal Rank Fusion, document-scoped filtering, and cross-encoder reranking before generation, and exposes the pipeline as an OpenAI-compatible chat completions API. Open WebUI connects to this service as its model endpoint, so requests from the user interface pass through the same retrieval logic used during validation. The service also provides document ingestion and management endpoints, so documents are indexed with the same Docling structure-aware chunking and OVMS embedding used at query time.

Performance was validated with the parameter values listed in Section 4.0. Organizations that change these values, particularly RERANK_TOP_N, should re-validate latency in their own environment.

2.0 Service Source: rag_pipeline_service.py

Save the following as rag_pipeline_service.py. All connection details are read from environment variables set in the Kubernetes manifest in Section 4.0.

```
"""
Hybrid AI 201 - RAG Pipeline Service

Exposes the validated retrieval pipeline (embed -> hybrid k-NN + BM25 retrieval ->
Reciprocal Rank Fusion -> optional document filtering -> cross-encoder reranking ->
generation) as an OpenAI-compatible chat completions API.

Open WebUI points its OPENAI_API_BASE_URL at this service instead of at vLLM
directly. From Open WebUI's perspective this is simply a model; all retrieval
happens inside this service before the request is forwarded to vLLM.

Endpoints

GET    /v1/models                model listing (Open WebUI calls this on startup)
POST   /v1/chat/completions      RAG-augmented chat completion (streaming supported)
POST   /ingest                   upload a document: Docling chunk -> embed -> index
```

```

GET    /documents          list indexed documents and chunk counts

DELETE /documents/{source}  remove a document from the index

GET    /health            liveness check
"""

import json
import os
import time
import asyncio
from typing import Optional

import httpx

from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.responses import StreamingResponse, JSONResponse

# -----
# Configuration (all overridable via environment)
# -----

OVMS_URL = os.getenv("OVMS_URL", "http://ovms:8000")
OPENSEARCH_URL = os.getenv("OPENSEARCH_URL", "http://opensearch:9200")
DOCLING_URL = os.getenv("DOCLING_URL", "http://docling:5001")
VLLM_URL = os.getenv("VLLM_URL", "http://localhost:8000")
VLLM_MODEL = os.getenv("VLLM_MODEL", "/path/to/Minstral-3-14B-Instruct-2512-BF16")

EMBED_MODEL = os.getenv("EMBED_MODEL", "Qwen3-Embedding-0.6B")
EMBED_DIM = int(os.getenv("EMBED_DIM", "1024"))
RERANK_MODEL = os.getenv("RERANK_MODEL", "bge-reranker-base")
INDEX_NAME = os.getenv("INDEX_NAME", "rag-docs")

FETCH_K = int(os.getenv("FETCH_K", "20"))          # candidates per retrieval method

```

```

RERANK_TOP_N = int(os.getenv("RERANK_TOP_N", "4")) # chunks passed to the LLM
MAX_CANDIDATES = int(os.getenv("MAX_CANDIDATES", "10")) # fused candidates sent to the
reranker

RRF_K = int(os.getenv("RRF_K", "60")) # standard RRF smoothing constant
DEFAULT_DOC_SOURCE = os.getenv("DEFAULT_DOC_SOURCE", "") # "" = search all documents

EXPOSED_MODEL_ID = os.getenv("EXPOSED_MODEL_ID", "hybrid-ai-201-rag")

OPENSEARCH_AUTH = (
    (os.getenv("OPENSEARCH_USERNAME"), os.getenv("OPENSEARCH_PASSWORD"))
    if os.getenv("OPENSEARCH_USERNAME")
    else None
)

PROMPT_TEMPLATE = """Answer the question using ONLY the context below. If the answer
isn't in the context, say so.

Context:
{context}

Question: {question}

Answer: """

app = FastAPI(title="Hybrid AI 201 RAG Pipeline Service")
# httpx defaults to 100 max connections; raise it so the pool is never the
# concurrency ceiling (all outbound calls to OVMS, OpenSearch and vLLM share it).
http = httpx.AsyncClient(
    timeout=httpx.Timeout(600.0),
    limits=httpx.Limits(max_connections=1024, max_keepalive_connections=256),
)

```

```

# -----
# Index bootstrap
# -----

INDEX_BODY = {
    "settings": {"index": {"knn": True}},
    "mappings": {
        "properties": {
            "text": {"type": "text"},
            "embedding": {"type": "knn_vector", "dimension": EMBED_DIM},
            "source": {"type": "keyword"},
            "chunk_id": {"type": "integer"},
            "headings": {"type": "keyword"},
            "page_numbers": {"type": "integer"},
        }
    },
}

@app.on_event("startup")
async def ensure_index():
    r = await http.head(f"{OPENSEARCH_URL}/{INDEX_NAME}", auth=OPENSEARCH_AUTH)

    if r.status_code == 404:
        r = await http.put(f"{OPENSEARCH_URL}/{INDEX_NAME}", json=INDEX_BODY,
auth=OPENSEARCH_AUTH)

        r.raise_for_status()

# -----
# Pipeline stages (identical logic to the validated benchmark pipeline)
# -----

```

```

async def embed(texts: list[str]) -> list[list[float]]:

    r = await http.post(f"{OVMS_URL}/v3/embeddings", json={"model": EMBED_MODEL, "input":
texts})

    r.raise_for_status()

    return [d["embedding"] for d in r.json()["data"]]


def _scope(doc_source: Optional[str]) -> dict:

    return {"filter": {"term": {"source": doc_source}}} if doc_source else {}


async def knn_search(vector: list[float], k: int, doc_source: Optional[str]) ->
list[dict]:

    knn = {"knn": {"embedding": {"vector": vector, "k": k}}}

    body = {"size": k, "query": {"bool": {"must": knn, **_scope(doc_source)}}}

    r = await http.post(f"{OPENSEARCH_URL}/{INDEX_NAME}/_search", json=body,
auth=OPENSEARCH_AUTH)

    r.raise_for_status()

    return [h["_source"] for h in r.json()["hits"]["hits"]]


async def bm25_search(query: str, k: int, doc_source: Optional[str]) -> list[dict]:

    match = {"match": {"text": query}}

    body = {"size": k, "query": {"bool": {"must": match, **_scope(doc_source)}}}

    r = await http.post(f"{OPENSEARCH_URL}/{INDEX_NAME}/_search", json=body,
auth=OPENSEARCH_AUTH)

    r.raise_for_status()

    return [h["_source"] for h in r.json()["hits"]["hits"]]


def rrf_fuse(*ranked_lists: list[dict]) -> list[dict]:

    """Reciprocal Rank Fusion: score(doc) = sum over lists of 1 / (RRF_K + rank)."""

```

```

scores: dict[str, float] = {}
docs: dict[str, dict] = {}

for ranked in ranked_lists:
    for rank, doc in enumerate(ranked, start=1):
        key = doc["text"]

        scores[key] = scores.get(key, 0.0) + 1.0 / (RRF_K + rank)

        docs.setdefault(key, doc)

return [docs[k] for k in sorted(scores, key=scores.get, reverse=True)]

async def rerank(query: str, candidates: list[dict], top_n: int) -> list[dict]:
    if not candidates:
        return []

    r = await http.post(
        f"{OVMS_URL}/v3/rerank",
        json={
            "model": RERANK_MODEL,
            "query": query[:2000],
            "documents": [c["text"] for c in candidates],
        },
    )

    r.raise_for_status()

    ranked = sorted(r.json()["results"], key=lambda x: x["relevance_score"],
reverse=True)

    return [candidates[x["index"]] for x in ranked[:top_n]]

async def retrieve_context(question: str, doc_source: Optional[str]) -> list[dict]:
    vector = (await embed([question]))[0]

    knn_hits, bm25_hits = await asyncio.gather(
        knn_search(vector, FETCH_K, doc_source),

```

```

        bm25_search(question, FETCH_K, doc_source),
    )

    fused = rrf_fuse(knn_hits, bm25_hits)[:MAX_CANDIDATES]
    return await rerank(question, fused, RERANK_TOP_N)

# -----
# OpenAI-compatible endpoints
# -----

@app.get("/health")
async def health():
    return {"status": "ok"}

@app.get("/v1/models")
async def list_models():
    model = {"id": EXPOSED_MODEL_ID, "object": "model", "owned_by": "hybrid-ai-201"}
    return {"object": "list", "data": [model]}

@app.post("/v1/chat/completions")
async def chat_completions(request: dict):
    messages = request.get("messages", [])
    user_turns = [i for i, m in enumerate(messages) if m.get("role") == "user"]
    if not user_turns:
        raise HTTPException(400, "No user message in request")

    last = user_turns[-1]
    question = messages[last]["content"]
    if isinstance(question, list): # multimodal-style content blocks

```

```

        question = " ".join(p.get("text", "") for p in question if isinstance(p, dict))

    doc_source = request.get("doc_source") or DEFAULT_DOC_SOURCE or None
    chunks = await retrieve_context(question, doc_source)
    context = "\n\n---\n\n".join(c["text"] for c in chunks)

    if not chunks:
        context = "(no relevant context found)"

    augmented = list(messages)

    prompt = PROMPT_TEMPLATE.format(context=context, question=question)
    augmented[last] = {"role": "user", "content": prompt}

    payload = {k: v for k, v in request.items() if k not in ("messages", "model",
"doc_source")}

    payload.update({"model": VLLM_MODEL, "messages": augmented})

    stream = bool(request.get("stream", False))

    if not stream:
        r = await http.post(f"{VLLM_URL}/v1/chat/completions", json=payload)
        r.raise_for_status()
        body = r.json()
        body["model"] = EXPOSED_MODEL_ID
        body["rag_sources"] = [
            {"source": c.get("source"), "chunk_id": c.get("chunk_id")} for c in chunks
        ]
        return JSONResponse(body)

    async def relay():
        async with http.stream("POST", f"{VLLM_URL}/v1/chat/completions", json=payload)
as resp:
            async for line in resp.aiter_lines():

```

```

        if line:

            yield line + "\n\n"

    return StreamingResponse(relay(), media_type="text/event-stream")

# -----
# Ingestion and document management
# -----

async def docling_chunk(

    filename: str, data: bytes, poll_s: int = 5, max_wait_s: int = 3600
) -> list[dict]:

    r = await http.post(

        f"{DOCLING_URL}/v1/chunk/hybrid/file/async", files={"files": (filename, data)}
    )

    r.raise_for_status()

    task_id = r.json()["task_id"]

    waited = 0

    while waited < max_wait_s:

        s = await http.get(f"{DOCLING_URL}/v1/status/poll/{task_id}")

        s.raise_for_status()

        status = s.json()["task_status"]

        if status == "success":

            res = await http.get(f"{DOCLING_URL}/v1/result/{task_id}")

            res.raise_for_status()

            body = res.json()

            return body.get("chunks", body if isinstance(body, list) else [])

        if status == "failure":

            raise HTTPException(502, f"Docling task failed:
{s.json().get('error_message')}")

        await asyncio.sleep(poll_s)

```

```

        waited += poll_s

    raise HTTPException(504, "Docling task timed out")

@app.post("/ingest")
async def ingest(file: UploadFile = File(...)):
    data = await file.read()

    source = file.filename

    chunks = await docling_chunk(source, data)

    if not chunks:
        raise HTTPException(422, "Docling returned no chunks")

    embeddings: list[list[float]] = []

    for i in range(0, len(chunks), 8):
        embeddings.extend(await embed([c["text"] for c in chunks[i:i + 8]]))

    lines = []

    for i, (c, vec) in enumerate(zip(chunks, embeddings)):
        lines.append(json.dumps({"index": {"_index": INDEX_NAME, "_id": f"{source}-{i}"}}))

        lines.append(json.dumps({
            "text": c["text"],
            "embedding": vec,
            "source": source,
            "chunk_id": i,
            "headings": c.get("headings") or [],
            "page_numbers": c.get("page_numbers") or [],
        }))

    r = await http.post(
        f"{OPENSEARCH_URL}/_bulk",
        content="\n".join(lines) + "\n",

```

```

        headers={"Content-Type": "application/x-ndjson"},
        auth=OPENSEARCH_AUTH,
    )
    r.raise_for_status()
    await http.post(f"{OPENSEARCH_URL}/{INDEX_NAME}/_refresh", auth=OPENSEARCH_AUTH)
    return {"source": source, "chunks_indexed": len(chunks)}

@app.get("/documents")
async def list_documents():
    body = {"size": 0, "aggs": {"sources": {"terms": {"field": "source", "size": 1000}}}}
    r = await http.post(f"{OPENSEARCH_URL}/{INDEX_NAME}/_search", json=body,
auth=OPENSEARCH_AUTH)
    r.raise_for_status()
    buckets = r.json()["aggregations"]["sources"]["buckets"]
    return {"documents": [{"source": b["key"], "chunks": b["doc_count"]} for b in
buckets]}

@app.delete("/documents/{source}")
async def delete_document(source: str):
    body = {"query": {"term": {"source": source}}}
    url = f"{OPENSEARCH_URL}/{INDEX_NAME}/_delete_by_query"
    r = await http.post(url, json=body, auth=OPENSEARCH_AUTH)
    r.raise_for_status()
    return {"source": source, "deleted": r.json().get("deleted", 0)}

```

3.0 Container Image

Save the following as Dockerfile in the same directory as rag_pipeline_service.py:

```

FROM python:3.12-slim

WORKDIR /app

RUN pip install --no-cache-dir "fastapi>=0.110" "uvicorn[standard]>=0.29" "httpx>=0.27"
python-multipart

COPY rag_pipeline_service.py .

EXPOSE 8080

CMD ["uvicorn", "rag_pipeline_service:app", "--host", "0.0.0.0", "--port", "8080", "--
workers", "1"]

```

Build the image into the cluster's local image store:

```

sudo nerdctl --namespace k8s.io build -t rag-poc/rag-pipeline:latest ./rag-pipeline-
service/

```

4.0 Kubernetes Manifest

Save the following as `rag-pipeline-deployment.yaml`, replacing `<vllm-host-ip>` with the IP address of the host running vLLM, `<set-matching-strong-password>` with the OpenSearch password set in Appendix B, and the `VLLM_MODEL` value with the model path on the vLLM host:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: rag-pipeline
  namespace: rag-poc
spec:
  replicas: 1
  selector:
    matchLabels: {app: rag-pipeline}
  template:
    metadata:
      labels: {app: rag-pipeline}

```

```

spec:
  containers:
    - name: rag-pipeline
      image: rag-poc/rag-pipeline:latest
      imagePullPolicy: Never
      env:
        - name: OVMS_URL
          value: "http://ovms.rag-poc.svc.cluster.local:8000"
        - name: OPENSEARCH_URL
          value: "http://opensearch.rag-poc.svc.cluster.local:9200"
        - name: OPENSEARCH_USERNAME
          value: "admin"
        - name: OPENSEARCH_PASSWORD
          value: "<set-matching-strong-password>"
        - name: DOCLING_URL
          value: "http://docling.rag-poc.svc.cluster.local:5001"
        - name: VLLM_URL
          value: "http://<vllm-host-ip>:8000"
        - name: VLLM_MODEL
          value: "/path/to/Ministral-3-14B-Instruct-2512-BF16"
        - {name: EMBED_MODEL, value: "Qwen3-Embedding-0.6B"}
        - {name: EMBED_DIM, value: "1024"}
        - {name: RERANK_MODEL, value: "bge-reranker-base"}
        - {name: INDEX_NAME, value: "rag-docs"}
        - {name: FETCH_K, value: "5"}
        - {name: MAX_CANDIDATES, value: "5"}
        - {name: RERANK_TOP_N, value: "1"}
        - {name: EXPOSED_MODEL_ID, value: "hybrid-ai-201-rag"}
      ports:
        - containerPort: 8080

```

```

resources:
  requests: {cpu: "2", memory: "4Gi"}
  limits: {cpu: "2", memory: "4Gi"}
readinessProbe:
  httpGet: {path: /health, port: 8080}
  initialDelaySeconds: 5
  periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: rag-pipeline
  namespace: rag-poc
spec:
  selector: {app: rag-pipeline}
  ports:
    - port: 8080
      targetPort: 8080

```

Apply the manifest and confirm the service is healthy and advertising its model:

```

kubectl apply -f rag-pipeline-deployment.yaml -n rag-poc
kubectl get pods -n rag-poc -l app=rag-pipeline -w

kubectl run -n rag-poc curl-test --rm -it --image=curlimages/curl --restart=Never -- \
  curl -s http://rag-pipeline.rag-poc.svc.cluster.local:8080/v1/models

```

The service allocates 2 CPU cores and 4 GB of memory, as shown in the compute allocation in Table 4.

Variable	Validated value	Purpose
----------	-----------------	---------

FETCH_K	5	Candidates retrieved by each method (k-NN and BM25) before fusion
MAX_CANDIDATES	5	Maximum fused candidates sent to the reranker
RERANK_TOP_N	1	Chunks retained after reranking and passed to the LLM
RRF_K	60	Reciprocal Rank Fusion smoothing constant (standard value)
DEFAULT_DOC_SOURCE	(unset)	When set, restricts every query to one document; leave unset to search all indexed documents (validation used a single-document scope)
EXPOSED_MODEL_ID	hybrid-ai-201-rag	Model name shown in Open WebUI

5.0 Open WebUI Configuration

When the pipeline service is deployed, Open WebUI acts purely as the chat interface. Replace the Open WebUI manifest from Appendix B with the following, which points Open WebUI at the pipeline service as its model endpoint. The RAG-related environment variables from Appendix B are no longer required, since retrieval and indexing are handled by the pipeline service:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: openwebui
  namespace: rag-poc
spec:
  replicas: 1
  selector:
    matchLabels: {app: openwebui}
  template:
    metadata:
      labels: {app: openwebui}
    spec:
      containers:

```

```

- name: openwebui

  image: ghcr.io/open-webui/open-webui:main

  env:
    - name: OPENAI_API_BASE_URL
      value: "http://rag-pipeline.rag-poc.svc.cluster.local:8080/v1"
    - {name: OPENAI_API_KEY, value: "not-required"}
    - {name: WEBUI_AUTH, value: "true"}

  ports:
    - containerPort: 8080

  resources:
    requests: {cpu: "1", memory: "2Gi"}
    limits: {cpu: "1", memory: "2Gi"}
---
apiVersion: v1
kind: Service
metadata:
  name: openwebui
  namespace: rag-poc
spec:
  selector: {app: openwebui}
  type: NodePort
  ports:
    - {port: 8080, targetPort: 8080, nodePort: 30080}

```

Apply the manifest, then select the hybrid-ai-201-rag model in Open WebUI. We recommend ingesting documents through the pipeline service (Section 6.0) rather than attaching files in the chat, so that all documents are indexed with the same chunking and embedding used at query time.

6.0 Document Ingestion and Management

Ingest a document, list indexed documents, or remove a document using the service's management endpoints:

```
curl -s -X POST http://<rag-pipeline-endpoint>:8080/ingest \
  -F "file=@/path/to/document.pdf"

curl -s http://<rag-pipeline-endpoint>:8080/documents

curl -s -X DELETE http://<rag-pipeline-endpoint>:8080/documents/document.pdf
```

7.0 Pipeline Feature Traceability

The following table maps each retrieval feature described in the Solution Overview to its implementation in the reference service and the deployment step that enables it.

Feature	Implementation	Deployment step
Hybrid retrieval (k-NN + BM25)	knn_search and bm25_search, run in parallel	Section 4.0 (pipeline service)
Reciprocal Rank Fusion	rrf_fuse	Section 4.0 (pipeline service)
Document filtering	DEFAULT_DOC_SOURCE, or a doc_source field on the request	Section 4.0 (pipeline service)
Reranking	rerank, calling OVMS bge-reranker-base	Appendix B (OVMS); Section 4.0
Structure-aware chunking	/ingest endpoint, calling Docling hybrid chunking	Appendix B (Docling); Section 4.0
Generation	Forwarded to vLLM after context assembly	Appendix B (vLLM)

Appendix D: Abbreviations

Abbreviation	Meaning
AI	Artificial Intelligence
AMX	Advanced Matrix Extensions
API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
LLM	Large Language Model
MLOps	Machine Learning Operations
NFS	Network File System
RAG	Retrieval-Augmented Generation
RBAC	Role-Based Access Control
S3	Simple Storage Service-compatible object protocol
SLA	Service-Level Agreement
SMB	Server Message Block
TPOT	Time Per Output Token
TTFT	Time To First Token
vLLM	LLM inference and serving framework

Resources

Intel AMX	Intel AMX
SR630 V4	Lenovo ThinkSystem SR630 V4 Server Product Guide > Lenovo Press
SR650 V4	https://lenovopress.lenovo.com/lp2127-thinksystem-sr650-v4-server
SR650a V4	https://lenovopress.lenovo.com/datasheet/ds0195-lenovo-thinksystem-sr650a-v4
Lenovo DM3200F	https://lenovopress.lenovo.com/lp2073-introducing-the-new-lenovo-thinksystem-storage-arrays
Nokia 7220 IXR-D series	Nokia 7220 Interconnect Router for Data Center Fabrics

Document history

Version 1.0 September 2026 ThinkSystem SR630 V4 Open-Source RAG Deployment

Trademarks and special notices

© Copyright Lenovo 2026.

Lenovo and the Lenovo logo are trademarks or registered trademarks of Lenovo in the United States, other countries, or both. A current list of Lenovo trademarks is available on the Web at <https://www.lenovo.com/us/en/legal/copytrade/>.

The following terms are trademarks of Lenovo in the United States, other countries, or both:

Lenovo®
ThinkEdge®
ThinkShield®
ThinkSystem®
XClarity®

The following terms are trademarks of other companies:

AMD and AMD EPYC™ are trademarks of Advanced Micro Devices, Inc.

Intel® and Intel Core® are trademarks of Intel Corporation or its subsidiaries.

Linux® is the trademark of Linus Torvalds in the U.S. and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used Lenovo products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-Lenovo products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by Lenovo. Sources for non-Lenovo list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. Lenovo has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-Lenovo products. Questions on the capability of non-Lenovo products should be addressed to the supplier of those products.

All statements regarding Lenovo future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local Lenovo office or Lenovo authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in Lenovo product announcements. The information is presented here to communicate Lenovo's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard Lenovo benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models. Any references in this information to non-Lenovo websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this Lenovo product and use of those websites are at your own risk.