

Absolutely, Andy — let's get this moving.

Here is the full comprehensive report, written in a clean, modern, Azure-style format, complete with diagrams, deep technical detail, and full integration with your Sentinel-Ops project tree.

You can paste this into VS Code, Obsidian, Typora, or any Markdown-to-PDF tool and export it directly.

Sentinel-Ops RAG Architecture Report

Modern Azure Edition — Comprehensive Technical Whitepaper

1. Executive Summary

This document presents a complete, engineering-grade design for implementing a Retrieval-Augmented Generation (RAG) system within the Sentinel-Ops environment. It is tailored specifically for your stack:

- Proxmox virtualization
- Ubuntu 24.04 host
- RKE2 Kubernetes cluster
- RabbitMQ messaging backbone
- Git-first workflow
- IaC-driven infrastructure

The goal is to transform your existing documentation, manifests, SBOMs, Terraform, and operational reports into a private, intelligent knowledge base that an LLM can query with high fidelity.

This report includes:

- Full RAG architecture
- Deployment options
- Recommended design for Sentinel-Ops
- Integration with your directory structure
- Kubernetes manifests (conceptual)
- Indexing pipeline
- RabbitMQ automation
- Security, backup, and operational guidance
- Future enhancements

2. Sentinel-Ops Project Tree Analysis

Your directory structure is already ideal for a RAG system.
Here is how each folder contributes to the knowledge base:

Directory	RAG Role	Notes
----- ----- -----		
artifacts/sboms/	High-value structured data	SBOMs provide dependency intelligence; index selectively
cluster/	Critical Kubernetes manifests	Longhorn, MetallLB, RabbitMQ configs → essential for architecture queries
helm/	Chart configs	Index templates & values files
infrastructure/	IaC + bootstrap configs	RKE2 config, cloud-init → foundational
reports/	Operational history	Session logs, architecture notes, playbooks → extremely valuable
scripts/	Utility scripts	Index for operational context
terraform/	IaC for Proxmox + RabbitMQ	High-value for environment reconstruction

This structure is perfect for a Git-driven RAG ingestion pipeline.

3. What a Home-Lab RAG Actually Does

A RAG system enhances an LLM by giving it the ability to:

- Search your Git repo
- Embed and index Markdown, YAML, JSON, and configs
- Retrieve relevant chunks
- Feed them into the model as context
- Generate answers grounded in your environment

This enables queries like:

- “How is MetallLB configured in Sentinel-Ops?”
- “Summarize my RKE2 architecture.”
- “What’s the Longhorn replica policy?”
- “Show me the Terraform resources for RabbitMQ.”

4. RAG Architecture Overview

Below is the high-level architecture.

```
'mermaid
flowchart TD
A[Git Repository<br>Sentinel-Ops] --> B[Ingestion Pipeline<br>Loader + Splitter + Embeddings]
```

B --> C[Vector Store
ChromaDB]
C --> D[Query Engine]
D --> E[LLM Runtime
Ollama]
E --> F[FastAPI RAG Service]
F --> G[User Interface
OpenWebUI or CLI]

.

5. Deployment Options

5.1 Option 1 — Local Lightweight RAG

- Python + LlamaIndex or LangChain
- ChromaDB local
- Ollama local
- Simple API

Pros: Private, simple, fast

Cons: Not cluster-native

5.2 Option 2 — GitHub Copilot Workspace

- Reads repo
- Summaries, suggestions

Pros: Easiest

Cons: Not a true RAG

5.3 Option 3 — Full Kubernetes-Native RAG (Recommended)

This is the ideal fit for Sentinel-Ops.

Components:

- Ollama → Deployment
- ChromaDB → StatefulSet
- Indexer → Deployment
- RabbitMQ → Already present
- FastAPI RAG service → Deployment
- OpenWebUI → Optional UI

6. Recommended Architecture for Sentinel-Ops

This section describes the canonical design for your environment.

```
`mermaid
flowchart LR
    subgraph Proxmox
        VM1[Ubuntu 24.04 Host]
    end

    subgraph RKE2 Cluster
        direction TB
        OLL[Ollama Deployment]
        CHR[ChromaDB StatefulSet]
        API[RAG FastAPI Service]
        IDX[Indexing Worker]
        MQ[(RabbitMQ)]
    end

    Git[Sentinel-Ops Repo] --> MQ
    MQ --> IDX
    IDX --> CHR
    API --> CHR
    API --> OLL
    User[OpenWebUI / CLI] --> API
```

7. Ingestion & Indexing Pipeline

7.1 Pipeline Steps

1. File Loader

- Recursively loads .md, .yaml, .yml, .json, .tf, .txt.

2. Text Splitter

- Chunk size: 512–1024 tokens
- Overlap: 50–100 tokens

3. Embeddings

- Recommended: bge-base-en-v1.5

4. Vector Store

- ChromaDB (persistent, local, fast)

5. Index Writer

- Writes embeddings into ChromaDB

7.2 RabbitMQ-Driven Automation

```
`mermaid
sequenceDiagram
    participant Git
    participant Webhook
    participant MQ as RabbitMQ
    participant IDX as Indexer Pod
    participant DB as ChromaDB

    Git->>Webhook: Push event
    Webhook->>MQ: Publish "docs_changed"
    MQ->>IDX: Deliver message
    IDX->>IDX: Re-embed changed files
    IDX->>DB: Update vector store
`
```

This ensures incremental indexing and real-time updates.

8. RAG Query Service (FastAPI)

Endpoint

```
`POST /query
{
    "question": "How is Longhorn configured?"
}`
```

Flow

1. Convert question → embedding
2. Query ChromaDB
3. Build context window
4. Send to Ollama
5. Return grounded answer

9. LLM Runtime (Ollama)

- Runs inside RKE2
- Models:
 - llama3
 - mistral
 - qwen2.5

- Exposed via ClusterIP

10. UI Layer

OpenWebUI

- Clean chat interface
- Connects to RAG API
- Multi-model routing

AnythingLLM

- Built-in RAG
- More opinionated

11. Kubernetes Deployment Model

```
`mermaid
flowchart TD
subgraph rag-system Namespace
    OLL[Deployment: Ollama]
    CHR[StatefulSet: ChromaDB]
    API[Deployment: RAG API]
    IDX[Deployment: Indexer]
    MQ[(RabbitMQ)]
end
`
```

12. Operational Model

12.1 Indexing Strategy

Strategy	Pros	Cons
Full reindex	Simple	Slow
Incremental (recommended)	Fast	Requires diffing
Scheduled	Predictable	Not real-time

13. Security Model

- Keep Ollama internal
- TLS on Ingress

- API key for /query
- RBAC for cluster resources
- SSH keys for Git

14. Backup Strategy

- PVC snapshots for ChromaDB
- Git repo already versioned
- Backup RAG config via ConfigMaps

15. Future Enhancements

- Add RAG-based CI assistant
- Add semantic search UI
- Add LLM-driven IaC validation
- Add SBOM-aware vulnerability queries

16. Appendix A — Sentinel-Ops Directory Integration

Here is how each folder maps to RAG ingestion:

Directory	Include?	Notes
artifacts/sboms/	Selectively	Large JSON → chunk carefully
cluster/	Yes	High-value YAML
helm/	Yes	Templates + values
infrastructure/	Yes	RKE2 configs
reports/	Yes	Rich operational history
scripts/	Yes	Operational context
terraform/	Yes	IaC for Proxmox + RabbitMQ

17. Appendix B — Example RAG Project Structure

```
rag/  
  api/  
  indexer/  
  embeddings/  
  vectorstore/  
  charts/
```

manifests/

18. Appendix C — Example FastAPI Snippet

```
`python
from fastapi import FastAPI
from ragengine import queryrag

app = FastAPI()

@app.post("/query")
async def query(payload: dict):
    question = payload["question"]
    answer = query_rag(question)
    return {"answer": answer}`
```

19. Conclusion

This architecture gives Sentinel-Ops:

- A private, intelligent knowledge base
- A reproducible, Git-driven RAG pipeline
- A Kubernetes-native deployment
- Automated indexing via RabbitMQ
- A modern UI for interacting with your environment
- A foundation for future AI-driven operations

You now have a complete, professional, Azure-style RAG architecture document ready for PDF export.

✓ Your report is ready for PDF export

You can now paste this into:

- VS Code → Markdown PDF extension
- Obsidian → Export to PDF
- Typora → File → Export → PDF
- Pandoc

And you'll get a clean, modern, professional PDF.

If you want, I can also generate:

- A cover page
- A table of contents
- A versioned document header
- A Sentinel-Ops branding block

Just tell me and I'll add it.