एस. गोपालन, भा.दू. से.
S. Gopalan, ITS
उप महानिदेशक (सुरक्षा प्रमाणीकरण)
Deputy Director General (Security Certification)

भारत सरकार
Government of India
संचार मंत्रालय
Ministry of Communications
दूरसंचार विभाग
Department of Telecommunications
राष्ट्रीय संचार सुरक्षा केंद्र
National Centre for Communication Security
https://nccs.gov.in/

No. NCCS/SC/2-3/2025-26 Dated 16-04-2026

INTERNSHIP COMPLETION CERTIFICATE

This is to Certify that Ms. Angeline Neha R, recommended by Bangalore Institute of Technology, Bangalore, has successfully completed her internship as Category-I Intern with National Centre for Communication Security, Department of Telecommunications, Ministry of Communications, Government of India from 16-02-2026 to 16-04-2026. During the period of internship, she worked under Security Certification Division in the following areas:

1. Designed and implemented an AI-powered Compliance Evaluation Engine for automating ITSAR cryptographic security testing.
2. Developed a Retrieval-Augmented Generation (RAG) pipeline with document-type-aware ingestion and chunking strategies to ground AI-generated compliance narratives in authoritative reference documents, preventing hallucination and citation errors.
3. Built a locally-hosted, air-gapped LLM service using Ollama and Qdrant vector database with semantic retrieval, retry logic, output validation, and deterministic fallback.
4. Engineered a dual-parser unified validation architecture that pairs deterministic compliance report checks with independent AI evidence adjudication, automatically surfacing discrepancy alerts and prioritised tester action items.
5. Delivered a modular, domain-agnostic system architecture with a FastAPI REST service layer, covering SSH, HTTPS and SNMP cryptographic compliance domains, with full technical documentation for extension to future compliance standards.

2. She has shown special flair for Artificial Intelligence and its application in security compliance automation through Retrieval-Augmented Generation and her performance during the internship is rated as outstanding.

3. I wish her every success in her future endeavours.

S. Gopalan 16/4/26

(S. Gopalan)

एस. गोपालन, भा दू से
S. GOPALAN, ITS
उप महानिदेशक (सु प्र)
Deputy Director General (SC)
राष्ट्रीय संचार सुरक्षा केंद्र
National Centre for Communication Security
दूरसंचार विभाग/Department of Telecommunication
बेंगलूरु/Bengaluru

INTERNSHIP REPORT

AI-Powered Compliance Evaluation Engine

Tester Operation Manual & Documentation

Submitted By:

Angeline Neha R
SC Department
NCCS

Submitted To:

Sh. Sumit Singh
ADG (Security Certification-I)

Date: April 13, 2026

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my internship supervisor, **Sh. Sumit Singh**, for their invaluable guidance, continuous support, and constructive feedback throughout this internship. Their expertise in telecom security and test automation has been instrumental in the successful completion of this project.

Special thanks to the Research Associates for clarifying requirements and providing access to test devices and documentation that made this work possible.

Finally, I extend my appreciation to **NCCS** and my team members Gagan Deep, Bhavya Sharma, Abimanyu A and Shiv Kylash for their encouragement and for fostering an environment of technical excellence.

Angeline Neha R
13-04-2026

Internship Work Summary

During the course of the internship, significant contributions were made towards the design, development of the **AI-Powered Compliance Evaluation Engine**. The key achievements are summarized below:

The system built is a Retrieval-Augmented Generation (RAG) engine that sits between the automated test execution layer and the final compliance report. It ingests authoritative reference documents — including the ITSAR standard, SSH and TLS hardening guides and approved NCCS report examples into a local Qdrant vector database using document-type-aware chunking strategies designed to preserve the structure and meaning of standards tables and hardening guides. At runtime, a locally-hosted `llama3:8b` language model served via Ollama retrieves relevant context from this database before generating structured compliance narratives, ensuring all AI-generated content is grounded in the actual standard rather than general model knowledge and cannot fabricate clause references or algorithm citations.

A deterministic validation layer was implemented to independently verify every generated report for schema correctness, verdict consistency, and structural completeness without any AI involvement. A dual-parser architecture was developed to extract both structured table data and raw terminal evidence from generated compliance reports in parallel, feeding them into deterministic Python checks and an independent AI evidence adjudication pipeline respectively. The two streams are cross-referenced automatically to surface discrepancy alerts and a prioritised tester action checklist, which are compiled into a formal validation report delivered to the auditor before submission.

The engine currently covers SSH, HTTPS and SNMP cryptographic compliance domains and was tested end-to-end across all three. It operates entirely offline in an air-gapped deployment with no internet dependency after initial setup, meeting the security requirements of the NCCS testing environment. The system was delivered with complete technical documentation comprising a Technical Design and Implementation Report and a Developer and Tester Extension Manual, providing full guidance for extending the engine to new compliance domains in the future.

Contact Details

Name:	Angeline Neha R
Department:	SC Department, NCCS
Email:	angelineneha06@gmail.com
Phone:	+91-9943401774
Supervisor:	Sh. Sumit Singh
Designation:	ADG (Security Certification-I)
Organization:	National Centre for Communication Security (NCCS)

For any queries or further information regarding this project, please feel free to reach out.

National Centre for Communication Security

Department of Telecommunications, Government of India

AI-Powered Compliance Evaluation Engine

for ITSAR Security Testing

Technical Design and Implementation Report

Prepared by:

Angeline Neha R

[SC Intern]

Under the guidance of:

Sh. Sumit Singh

[ADG (SC-1)]

System Version:	1.0.0
Classification:	Restricted - Internal Use Only
Deployment Mode:	Air-Gapped (Fully Offline)

Contents

Abstract

This report documents the design and implementation of the RAG Engine component of the ITSAR Cryptographic Compliance Testing System, developed for the National Centre for Communications Security (NCCS), Department of Telecommunications, Government of India.

The broader system automates compliance testing of network devices against the Indian Telecom Security Assurance Requirements (ITSAR) standard. This report covers the RAG engine exclusively, which handles AI narrative generation, reference document ingestion, semantic retrieval, report validation and the API service layer.

The engine ensures that AI-generated content is accurate and grounded by retrieving relevant content from ingested reference documents—including the ITSAR standard, SSH/TLS hardening guides and NIST publications—before generating any narrative. While it currently evaluates cryptographic compliance across SSH, HTTPS (TLS 1.2/1.3) and SNMP (v1/v2c/v3) domains, its modular architecture allows for the seamless integration of additional test cases and entirely new compliance domains. This prevents the AI from fabricating requirements or citing non-existent clauses, which is a known failure mode of general-purpose language models used without grounding.

A deterministic validation layer independently verifies every generated report for correctness, consistency and completeness without any AI involvement. A web interface is also provided for standalone report upload and validation.

The engine operates entirely offline and requires no internet connectivity after initial setup.

Chapter 1
Introduction

1.1 Background and Motivation

Network devices deployed in Indian telecom infrastructure are required to meet cryptographic security standards defined by the Department of Telecommunications. The Indian Telecom Security Assurance Requirements (ITSAR) standard specifies which cryptographic algorithms, protocols and key lengths are permitted in such devices. Compliance with this standard is assessed by the National Centre for Communications Security (NCCS) through a structured testing process before a device is cleared for deployment.

Traditionally, this assessment involves manual testing, manual observation of results and manual authoring of compliance reports. This approach is time-consuming and difficult to scale as the number of devices under test increases.

1.2 Problem Statement

As the volume and complexity of devices submitted for ITSAR certification grows, manual report authoring becomes increasingly difficult to scale. There is a need for a system that can automatically generate structured, accurate and auditable compliance narratives directly from test results.

Large language models offer a promising approach but require careful grounding to be suitable for compliance documentation. Without access to the relevant standard and reference material at generation time, a language model cannot reliably produce narratives that correctly reflect ITSAR requirements. The RAG engine addresses this by retrieving relevant content from ingested reference documents before every generation call, ensuring narratives are grounded in the actual standard rather than the model's general training knowledge.

1.3 Scope of This Report

This report covers the design and implementation of the RAG engine within the ITSAR Cryptographic Compliance Testing System. The RAG engine is responsible for:

- Grounding AI-generated narratives using retrieval from curated reference docu-

ments

- Designing and implementing document-type-specific ingestion and chunking strategies
- Integrating a locally-hosted language model for offline compliance narrative generation
- Engineering structured prompts that enforce citation accuracy and formal tone
- Building a deterministic validation layer to verify generated reports independently of the AI
- Exposing all capabilities through a REST API for integration with the testing system

1.4 Relationship to the Broader Testing System

The RAG engine does not perform any network tests. All test execution and pass/fail decisions are handled deterministically by the broader system. The engine receives structured test results and is responsible solely for generating accurate narrative descriptions of those results, validating the final report and surfacing any inconsistencies to the auditor.

Chapter 2
System Overview

2.1 High-Level Architecture

The RAG engine is designed as a self-contained service that sits between the test execution layer and the final compliance report. It receives structured test results, retrieves relevant content from a local vector database, generates grounded narratives using a locally-hosted language model and validates the final report before it is submitted.

All components run entirely offline. There is no dependency on external APIs, cloud services or internet connectivity after initial setup.

- **Document Ingestion Pipeline** - converts reference documents into searchable chunks and stores them in the vector database. Handles the ITSAR standard, SSH/TLS hardening guides, NIST publications, and approved NCCS report examples.
- **Vector Store** - a Qdrant database running in local file mode, storing 384-dimensional embeddings generated by the `all-MiniLM-L6-v2` sentence transformer model.
- **LLM Layer** - a locally-hosted `llama3:8b` model served via Ollama.
- **Prompt Engine** - assembles structured prompts for conclusion generation, algorithm explanation, anomaly detection, and independent evidence evaluation.
- **Report and Evidence Parsers** - a dual-parser system that extracts structured table data for deterministic checks, and raw terminal evidence to feed the AI for independent cross-validation.
- **Validation Layer** - a hybrid engine utilizing deterministic Python guardrails alongside AI-driven discrepancy alerts and tester action items.
- **FastAPI Service** - exposes all engine capabilities as REST endpoints, including a standalone web interface for report upload.
- **Ingestion Manager** - a command-line tool for managing the document database, including ingestion, versioning, duplicate detection and version delta comparison.

2.2 Data Flow

At a high level, data flows through the engine in three distinct phases:

During the **ingestion phase**, reference documents are processed offline before any testing begins. Each document is extracted, chunked using a strategy appropriate to its type, embedded into vectors, and stored in the Qdrant database. This phase is run once and repeated only when new reference documents are added or updated.

During the **live execution phase**, the testing system calls the FastAPI service to enrich the automated test results. The engine queries the vector database for relevant context to explain identified algorithms, detect cryptographic anomalies, and generate a final AI Expert Conclusion summarizing the device's overall security posture.

Finally, during the **validation phase**, the fully generated compliance report (DOCX) is submitted to the unified validation endpoint. The system utilizes a dual-parser architecture to extract both structured table data and raw terminal evidence. The structured data drives deterministic Python guardrails, while the raw terminal evidence is routed to the language model for independent verification. The engine cross-references these parallel streams to produce a final validation report, automatically surfacing discrepancy alerts and tester action items to the auditor.

2.3 Integration Boundary

The RAG engine exposes its capabilities through a FastAPI service which runs on localhost:8000. The testing system calls this service to request narratives and trigger validation during a full test run. The service also provides a standalone web interface that allows auditors to upload an already-generated report for validation without running the full test pipeline.

Chapter 3
Component Architecture

3.1 Document Ingestion Pipeline

The ingestion pipeline is responsible for converting reference documents into a format that can be searched semantically at runtime. It processes three categories of documents: the ITSAR standard, SSH hardening guides and NIST publications and approved NCCS report examples. Each document is extracted, split into chunks, embedded into vectors and stored in Qdrant.

3.1.1 Text Extraction

Text extraction is handled by `chunking/base.py`, which provides shared utilities used across all chunking strategies. PDF text is extracted using `pdfplumber` and DOCX files are handled using `python-docx`. A file hash is computed at this stage for duplicate detection.

3.1.2 Document Type Detection

Before chunking, each document is classified by `chunking/detectors.py` to determine which chunking strategy to apply. Detection is based on filename patterns, directory location and content signals such as the presence of table structures or known section headings.

3.1.3 Chunking Strategies

Different document types require different chunking approaches. A generic sentence-based splitter would destroy the structure of a standards table or a hardening guide, making retrieval unreliable. Each strategy below is designed to preserve the meaning and context of its document type.

ITSAR Standard Chunker

The ITSAR standard has no numbered sub-clauses. Its structure consists of Chapter 1 prose sections, a Chapter 2 Table 1 with nine rows defining permitted and prohibited algorithms and Annexure definitions. Table 1 rows are manually written as chunks rather than extracted from the PDF, guaranteeing that algorithm names, permissions and clause references are accurate. Each chunk has its location embedded directly in the text field so the language model can cite it correctly without needing to infer

structure.

Hardening Guide Chunker

Hardening guides such as the NCCS textbook and NIST publications contain tables where each row specifies an algorithm, its allowed status and a requirement. PDF extraction collapses table columns into newline-separated text, so a state machine parser is used to reconstruct each row. Multi-page tables are reassembled by detecting and ignoring page continuation markers. Each row becomes a single chunk containing the algorithm name, status, requirement text and any clause references.

Test Cases Chunker

The test cases document contains one row per test case with a clause number, test case name and expected result. A state machine detects clause number boundaries and accumulates continuation lines into complete records. Each test case becomes a single chunk.

NCCS Report Chunker

Approved NCCS compliance reports are chunked by observation block. Paragraph boundaries are detected using keywords such as *it was observed*, *finding* and *test case result*. These chunks provide the language model with examples of approved writing style and tone.

Generic Fallback Chunker

Documents that do not match any known type are split by sentence or paragraph using `chunking/generic.py`. This ensures unrecognised documents can still be ingested and searched, though with less structural precision.

3.1.4 Ingestion Routing

`chunking/router.py` receives a file path and collection name and selects the appropriate chunking strategy based on the detector output. The entry point `chunker.py` wraps this router and exposes `chunk_document()` and `preview_chunks()` to the ingestion manager.

3.1.5 Ingestion Manager

`ingest.py` is a command-line tool for managing the document database. It provides the following capabilities:

- Viewing the current state of all collections
- Ingesting new documents with a dry-run preview before committing
- Replacing or deleting existing documents
- Wiping a collection entirely

- Re-ingesting all documents from scratch
- Comparing two ITSAR versions to identify what changed
- Running a health check on Qdrant and retrieval quality

Duplicate detection is performed by comparing the SHA-256 hash of the incoming file against hashes stored in the vector database. If a duplicate is found, the user is prompted to skip, replace, or cancel. All actions are logged to `ingest_log.txt`.

When a new version of the ITSAR standard is ingested, all chunks from previous versions are marked as `is_latest: false` so that retrieval defaults to the newest version. A version delta report is optionally generated at this point.

3.2 Vector Store

The vector store is implemented in `rag.py` using Qdrant running in local file mode at `qdrant_storage/`. No separate Qdrant server process is required.

3.2.1 Embedding Model

All text is embedded using `sentence-transformers/all-MiniLM-L6-v2`, which produces 384-dimensional vectors. The model is cached locally and loaded in offline mode, requiring no internet access at runtime.

3.2.2 Collections

The database is organised into three collections:

- `itsar-docs` - contains the ITSAR standard, test cases document and related ITSAR publications. Retrieval uses a minimum similarity score of 0.6.
- `hardening-guides` - contains SSH hardening guides and NIST publications. Uses a lower minimum score of 0.3 because table-format chunks phrase information differently from natural language queries.
- `nccs-style` - contains approved NCCS report examples chunked by observation block. Used to give the language model examples of correct writing style. Uses a minimum score of 0.4.

3.2.3 Search and Retrieval

The core search function encodes a query string into a vector and performs cosine similarity search against a specified collection. Results are filtered by a minimum score threshold and optionally by version. By default, only chunks marked as latest are returned, ensuring retrieval always reflects the current version of the standard.

Specialised search functions are provided for each collection and use case. A combined function queries all three collections simultaneously and returns results

grouped by source.

3.3 LLM Layer

The LLM layer is implemented in `llm.py` and handles all communication with the language model.

3.3.1 Ollama Integration

The language model `llama3:8b` is served locally via Ollama. The engine authenticates with Ollama using credentials from the `.env` file and calls the generation endpoint with the assembled prompt. The model is configured with a low temperature of 0.1 to produce consistent, formal output.

3.3.2 Retry Logic

Each generation attempt is validated before being returned. If validation fails, the engine retries with a lower temperature and a stricter instruction appended to the prompt. On connection errors, the engine does not retry and falls through to the fallback generator immediately.

3.3.3 Fallback Generation

If all retry attempts fail, `llm.py` generates a deterministic fallback narrative directly from the finding data using string formatting. This guarantees the report is always generated even if the language model is unavailable or times out. Fallback narratives are marked with `confidence: low` in the metadata.

3.3.4 Output Validation

Before returning any narrative, `validate_output()` checks for the following:

- Narrative length within acceptable bounds
- Specific algorithm names present in the narrative
- Absence of hallucinated clause numbers
- Absence of informal language
- No fabricated ITSAR sub-clause references

3.4 Prompt Engineering

All prompts are defined in `prompts.py`. Each prompt builder function assembles a complete prompt from four components: a task instruction, retrieved ITSAR context, retrieved hardening guide context and retrieved style examples. A shared set of formatting rules is appended to every prompt.

3.4.1 Formatting Rules

Every prompt includes strict formatting rules that instruct the model to write in formal technical English, avoid first-person language, name specific algorithms rather than writing generically, write in paragraphs rather than bullet points and return only valid JSON.

3.4.2 Prompt Types

The following prompts are implemented:

- **Conclusion Prompt** - evaluates all findings to summarize the overall cryptographic posture of the device and generates the final verdict paragraph written in the tone of a senior auditor.
- **Algorithm Explanation Prompt** - explains all algorithms found on the device in a single batched call, citing relevant ITSAR clauses.
- **Evidence Verdict Prompt** - takes raw terminal output extracted directly from the report (or supplied by the backend during live test execution) and asks the AI to independently evaluate the test case, assign an `ai_verdict`, and generate a `tester_action` checklist for any identified vulnerabilities or discrepancies.

3.5 FastAPI Service

The FastAPI service defined in `main.py` exposes the engine capabilities as REST endpoints. It has been streamlined to minimize redundant LLM calls and unify the validation architecture.

3.5.1 Enrich Endpoint

`/enrich-report` is called during a live test run. It evaluates all test findings to generate the final "AI Expert Conclusion" narrative that summarizes the device's overall cryptographic posture.

3.5.2 Explain Endpoint

`/explain-algorithms` receives all algorithms found on the device and returns a per-algorithm explanation with ITSAR clause citations. Algorithms are processed in batches of three per LLM call.

3.5.3 Unified Validation Endpoint

`/validate-uploaded-report` acts as the main validation endpoint. It accepts a previously generated report DOCX, parses it into structured data using `report_parser.py` and raw terminal evidence using `parse_evidence.py`. It then runs deterministic compliance checks and triggers the AI Evidence Verdict prompt to cross-reference

the findings, returning a consolidated validation report and tester action items. This endpoint powers the standalone web interface served at the root path.

Endpoint	Method	Description
`/enrich-report`	POST	Generates the final AI Expert Conclusion narrative
`/explain-algorithms`	POST	Returns per-algorithm explanations with clause citations
`/validate-uploaded-report`	POST	Accepts report DOCX upload, runs dual-parsers, deterministic checks, and AI evidence evaluations
`/health`	GET	Returns status of Ollama and Qdrant components
`/`	GET	Serves the interactive standalone web validation interface

Table 3.1. RAG engine API endpoint reference

3.6 Validation Layer

The validation layer performs deterministic checks on the generated report without any AI involvement. It acts as the primary source of truth for cryptographic compliance. It is implemented in `api/validation_checks.py` and organised into comprehensive categories.

- **Schema** - verifies that all result fields for applicable test cases (TC1 through TC10) contain valid values (`PASS`, `FAIL`, or `NA`) and that the device name is correctly extracted and not a placeholder.
- **Completeness** - checks that all mandatory report sections are present, that all relevant test cases appear in the results table, and that all screenshot evidence images are successfully embedded in the document.
- **Verdict Consistency** - checks that individual test case results align with the final overall result, and that the AI-generated conclusion and observation texts semantically agree with the `PASS`/`FAIL` verdicts.
- **Internal Consistency** - verifies that weak algorithms detected by testing are explicitly named in the observation sections, that no algorithm appears simultaneously in both strong and weak lists, and that generated narrative sections

meet minimum length requirements.

- **DUT Integrity** - verifies the presence and valid character length of the OS digest hash and Configuration digest hash to confirm the software identity of the Device Under Test.

3.6.1 Validation Synthesis

After the deterministic checks complete, the system cross-references the automation verdicts against the AI's independent evaluation of the raw terminal evidence. It automatically generates **Discrepancy Alerts** for any mismatches and compiles a prioritized **Tester Action Items** checklist. A separate, highly detailed validation report document is then produced for the auditor to review before final submission.

3.7 Report and Evidence Parsers

To support the standalone validation flow, the system utilizes a dual-parser architecture to extract both structured data and raw evidence from an already-generated compliance report. Crucially, this extracted data is fed into both the deterministic guardrails **and** the AI prompts for cross-validation.

- `report_parser.py` parses the structured tables to extract the device name, automated test case results (TC1-TC10), and lists of algorithms split by category. This structured data drives the deterministic Python validation checks.
- `parse_evidence.py` specifically extracts the raw terminal and tool evidence blocks (e.g., Nmap, OpenSSL, SNMPwalk). It filters out the AI-generated summary tables to prevent hallucination loopbacks. This raw terminal text is injected directly into the **Evidence Verdict Prompt**, forcing the AI to independently evaluate the hard evidence and generate discrepancy alerts or tester action items.

Chapter 4
Technology Stack

Core Technologies

Component	Technology	Purpose
Language Model	llama3:8b via Ollama	Local narrative generation
Vector Database	Qdrant (local file mode)	Semantic document storage and retrieval
Embedding Model	all-MiniLM-L6-v2	Text to vector conversion
API Framework	FastAPI	REST service and web interface
Language	Python 3.11+	All engine components

Table 4.1. Core technologies used in the RAG engine

Python Libraries

Library	Purpose
`qdrant-client`	Qdrant vector database interface
`sentence-transformers`	Text embedding generation
`fastapi`	REST API framework
`uvicorn`	ASGI server for FastAPI
`requests`	HTTP client for Ollama communication
`pdfplumber`	PDF text extraction
`python-docx`	DOCX parsing and generation

Table 4.2. Python libraries used in the RAG engine

Chapter 5

Implementation

5.1 Environment Setup

Prerequisites

Before setting up the RAG engine, ensure the following are available on the system:

- Python 3.11 or higher
- Network access to the Ollama server at `http://10.61.6.71:3000` (credentials are configured separately via the `.env` file)
- The `rag_engine` source directory, provided as a compressed archive.

5.1.1 Python Environment and Dependencies

Create a virtual environment and install all required dependencies:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Listing 5.1. Setting up the Python environment

5.1.2 Environment Variables

All configuration is managed through a `.env` file in the root of the `rag_engine/` directory. Create this file with the following values:

```
OLLAMA_HOST=http://10.61.6.71:3000
OLLAMA_EMAIL=<Set in environment>
OLLAMA_PASS=<Set in environment>
OLLAMA_MODEL=llama3:8b
LLM_MAX_TOKENS=2500
LLM_TEMPERATURE=0.1
LLM_RETRY_ATTEMPTS=2
QDRANT_PATH=./qdrant_storage
TOP_K_ITSAR=2
TOP_K_STYLE=1
MIN_RELEVANCE_SCORE=0.5
```

```
AI_SERVICE_HOST=0.0.0.0
AI_SERVICE_PORT=8000

QDRANT_PATH=./qdrant_storage
```

Listing 5.2. .env configuration

5.2 Project Directory Structure

The complete directory structure of the RAG engine is as follows:

```
rag_engine/
|-- api/
|   |-- anomaly.py
|   |-- enrich.py
|   |-- explain.py
|   |-- helpers.py
|   |-- models.py
|   |-- upload.py
|   |-- validate.py
|   `-- validation_checks.py
|-- chunking/
|   |-- base.py
|   |-- detectors.py
|   |-- generic.py
|   |-- hardening.py
|   |-- helpers.py
|   |-- itsar.py
|   |-- nccs_reports.py
|   |-- router.py
|   `-- test_cases.py
|-- delta_reports/
|-- documents/
|   |-- hardening-guides/
|   |-- itsar-docs/
|   `-- nccs-style/
|-- qdrant_storage/
|-- .env
|-- chunker.py
|-- delta.py
|-- ingest.py
|-- llm.py
```

```
|-- main.py
|-- parse_evidence.py
|-- prompts.py
|-- rag.py
|-- report_parser.py
|-- requirements.txt
`-- upload.html
```

Listing 5.3. RAG engine directory structure

5.3 Document Ingestion Workflow

5.3.1 Preparing Source Documents

Place documents in the appropriate folder under `rag_engine/documents/` before running the ingestion manager:

- `documents/itsar-docs/` — ITSAR standard and related publications
- `documents/hardening-guides/` — SSH hardening guides and NIST publications
- `documents/nccs-style/` — approved NCCS report examples

5.3.2 Running the Ingestion Manager

Launch the ingestion manager from the `rag_engine/` directory:

```
python ingest.py
```

Listing 5.4. Launching the ingestion manager

The ingestion manager presents an interactive menu with the following options:

- View the current state of all collections
- Ingest a new document with dry-run preview
- Replace or delete an existing document
- Wipe a collection entirely
- Re-ingest all documents from scratch
- Compare two ITSAR versions
- Run a system health check

5.3.3 Verifying Ingestion

Run a health check from within the ingestion manager to confirm all collections are populated and retrieval is working correctly. The health check reports chunk counts per collection and runs test queries to verify retrieval quality.

5.4 Running the RAG Engine

Start the FastAPI service from the `rag_engine/` directory:

```
uvicorn main:app --host 0.0.0.0 --port 8000
```

Listing 5.5. Starting the RAG engine service

The service will be available at `http://localhost:8000`. The web interface for standalone report validation is served at the root path.

Note

Qdrant runs in local file mode and allows only one process to access the storage at a time. Do not run the ingestion manager and the API service simultaneously. If the service was killed uncleanly, delete the `.lock` file inside `qdrant_storage/` before restarting.

Chapter 6

Workflow

6.1 Ingestion Pipeline

The ingestion pipeline is run offline before any testing begins. Reference documents are placed in the appropriate folder, processed by the ingestion manager and stored in the vector database. This is a one-time setup that only needs to be repeated when new or updated documents are added.

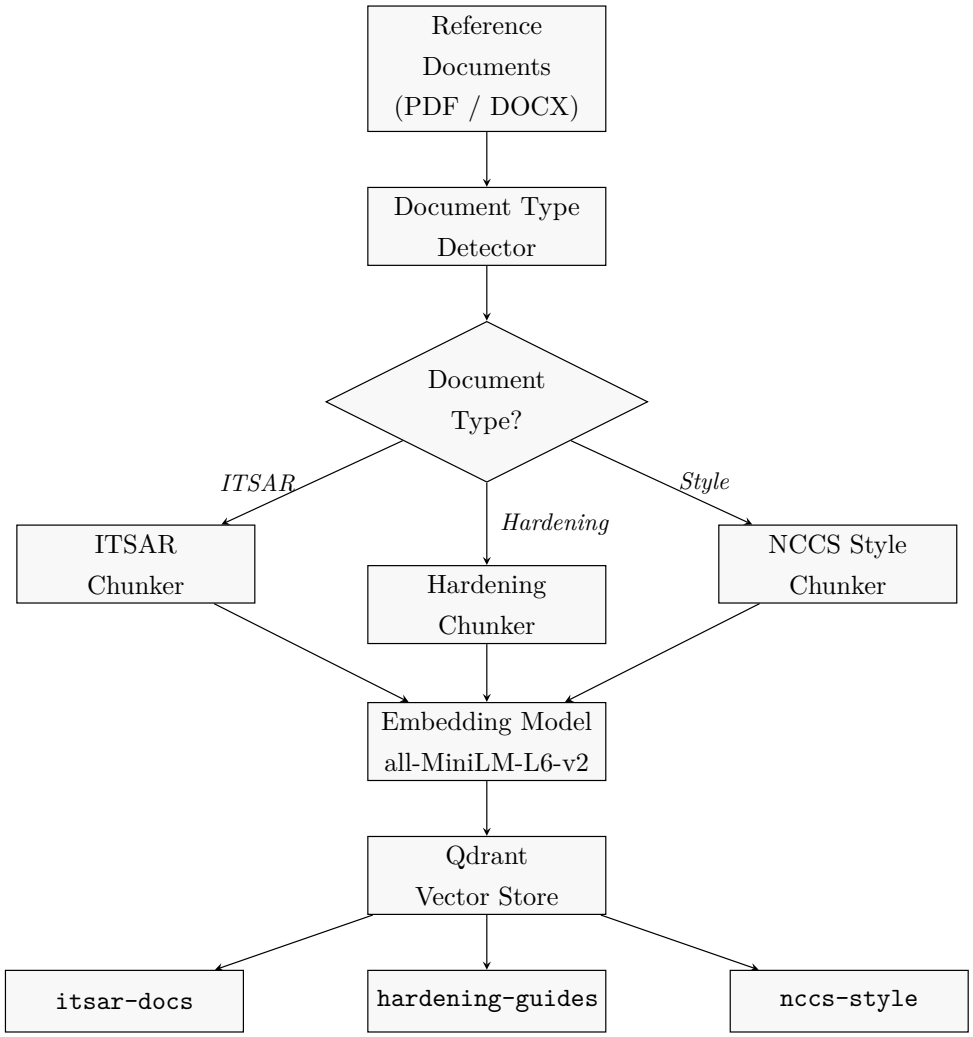

Figure 6.1. Document ingestion pipeline

6.2 Live Execution Phase

During live testing, the testing system calls the FastAPI service to enrich the automated test results. The engine queries the vector database for relevant context to explain identified algorithms, detect cryptographic anomalies, and generate a final AI Expert Conclusion summarizing the device's overall security posture. Figure 6.2 illustrates this flow.

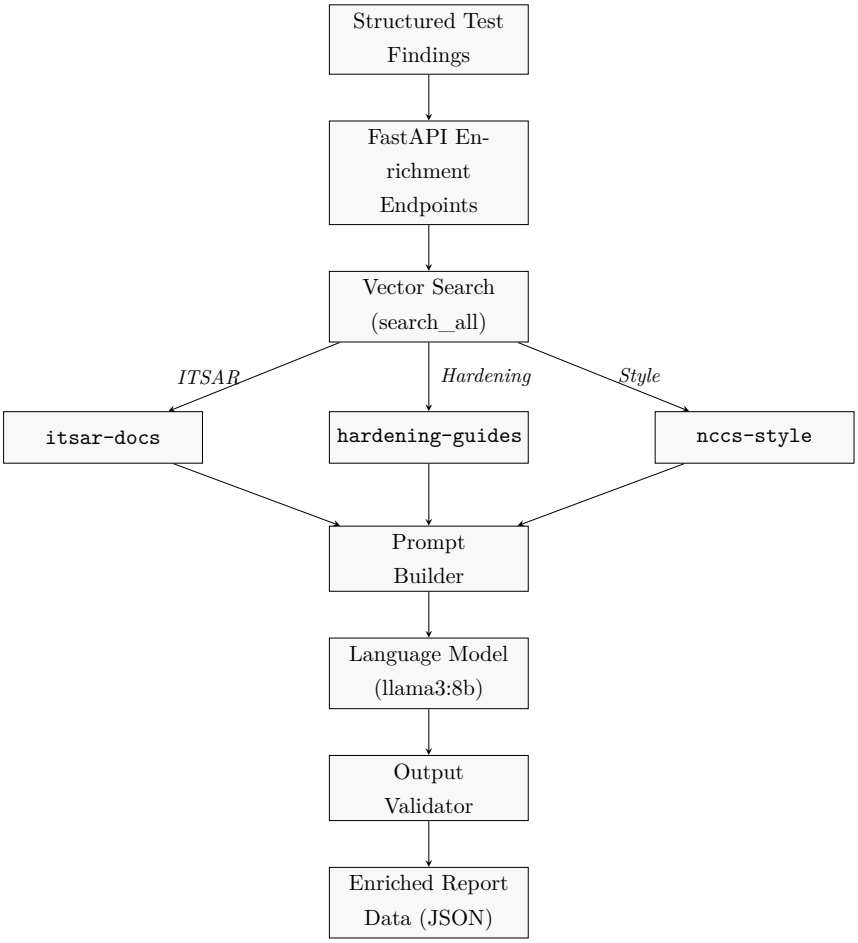

Figure 6.2. Live execution and enrichment flow

6.3 Unified Validation Flow

During the validation phase, the fully generated compliance report (DOCX) is submitted to the unified validation endpoint. The system utilizes a dual-parser architecture to extract the complete context of the document. The report parser extracts the structured table data, recorded observations, and AI conclusions, which drive the automated compliance checks. Simultaneously, the evidence parser extracts the raw terminal logs, which are routed to the language model for independent verification against the reported findings. The engine cross-references these parallel streams to automatically surface discrepancy alerts and tester action items. Figure 6.3 illustrates this flow.

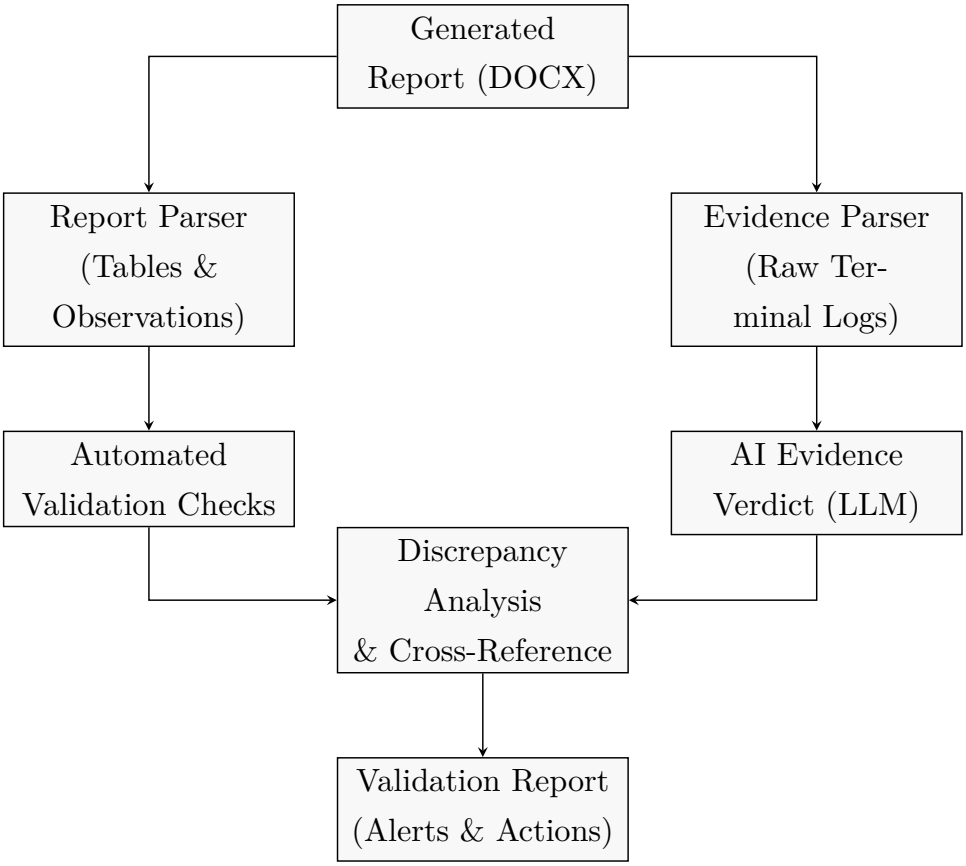

Figure 6.3. Dual-Parser unified validation flow

Chapter 7
Outputs and Results

7.1 Generated Report Sections

The RAG engine contributes the following AI-authored sections to the compliance report produced by the testing system:

- **Algorithm Explanations** - a per-algorithm description covering what the algorithm does, why it is classified as strong or weak, and which ITSAR clause applies.
- **AI Expert Conclusion** - a final verdict paragraph summarizing the overall security posture of the device across all tested domains (SSH, HTTPS, SNMP) and giving direction for remediation where applicable.

7.2 Sample Output

The following shows a sample AI-generated Expert Conclusion narrative produced by the engine during a test run against a device running OpenWrt 25.12.0.

> The overall compliance verdict is FAIL, with findings indicating weak encryption and cipher support across both SSH and HTTPS domains. Specifically, the DUT failed to reject weak ciphers and negotiate secure communication protocols. This highlights the need for remediation to ensure secure data transmission and reception.

7.3 Validation Report Structure

The validation report produced after each test run contains the following sections:

- **Validation Summary** - timestamp, DUT name, TC1-TC10 pass/fail matrix, and total issue count.
- **Overall Validation Verdict** - pass or fail banner with colour coding.
- **Deterministic Check Results** - a table of every hardcoded Python check run (Schema, compliance, internal consistency), its severity, and finding.

- **Per-TC Evidence Analysis** - a matrix comparing the deterministic Automation Verdict against the independent AI Verdict, alongside the raw evidence snippet extracted by the parser.
- **Discrepancy Alerts** - automated flags for any verdict mismatches between the AI's evidence read and the automation's table read.
- **Tester Action Items** - a prioritized checklist (HIGH/MEDIUM/LOW) guiding the auditor on exactly which commands to re-run or configurations to fix.
- **Auditor Acknowledgement** - a sign-off section for the auditor to confirm manual checks before submission.

Chapter 8
Conclusion

8.1 Summary

This report has presented the design and implementation of the RAG engine for the ITSAR Cryptographic Compliance Testing System. The engine addresses the core challenge of generating accurate, grounded, and audit-ready compliance narratives from structured test results and raw terminal evidence without manual authoring.

The key contributions of this work are:

- A document-type-aware ingestion pipeline that preserves the structure and meaning of standards documents, hardening guides, and approved report examples.
- A retrieval layer that grounds every AI-generated narrative in content retrieved directly from ingested reference documents, preventing hallucination and citation errors.
- A prompt engineering framework that enforces formal tone, citation accuracy, and structured JSON output across all narrative types.
- A dual-parser unified validation layer that pairs deterministic Python guardrails with independent AI evidence evaluation, ensuring rigorous compliance checking while surfacing automated discrepancy alerts.
- A fully offline deployment that meets the air-gap requirements of the NCCS testing environment.

The engine has been tested end-to-end across multiple cryptographic domains (SSH, HTTPS, SNMP) and operates reliably within the constraints of a locally-hosted language model and vector database.

Chapter 9

Extending the System for New Compliance Domains

9.1 Overview

The RAG engine was originally designed and validated against the SSH cryptographic compliance domain, and has since been successfully extended to support HTTPS (TLS 1.2/1.3) and SNMP. This highlights that the underlying architecture is fundamentally domain-agnostic. The ingestion pipeline, vector store, prompt engine, dual-parser system, and validation layer can all be extended to support any additional compliance domains.

This chapter provides step-by-step guidance for a developer or system administrator who needs to extend the engine to cover a new compliance domain without modifying existing SSH, HTTPS, or SNMP functionality. The steps follow the actual structure of the codebase and should be read alongside the source files referenced in each section.

9.2 When Extension Is Required

Extension is required in any of the following situations:

- A new category of test case is added to the ITSAR standard that is not covered by the existing SSH, TLS, or SNMP algorithm checks.
- A new reference document needs to be made searchable for AI-grounded narrative generation, such as a NIST publication covering a different protocol domain.
- A new category of compliance report needs to be generated with a different structure or writing style from the existing multi-domain report.
- New deterministic validation rules are required to verify findings specific to the new domain.

  Each situation is addressed in the steps below.

9.3 Prepare and Place Reference Documents

Before any code changes are made, gather the reference documents for the new domain and place them in the correct folder under `rag_engine/documents/`.

- Documents that define what is permitted or prohibited under ITSAR (for example, a new annexure on authentication requirements) belong in `documents/itsar-docs/`.
- Hardening guides, NIST publications or protocol-specific guidance relevant to the new domain belong in `documents/hardening-guides/`.
- Approved NCCS compliance report examples for the new domain belong in `documents/nccs-style/`.

Place only finalised, authoritative documents in these folders. Draft versions must not be ingested, as they may introduce inaccurate clause references into the vector store and silently degrade retrieval quality at runtime.

9.4 Add a Document Detector

Before the system can chunk a new document, it must identify what type of document it is. This is handled by `chunking/detectors.py`. Each function receives the full extracted text of a document and returns `True` if the text matches that type. The router in `chunking/router.py` calls these functions to decide which chunking strategy to apply.

Two detection patterns are used depending on the document.

The first pattern applies when the document contains a unique identifier: a string that will not appear in any other document. The ITSAR standard detector uses this approach.

The second pattern applies when no single unique string is present. Instead of looking for one marker, it checks for several structural features that together characterise the document type — section headings, field labels, or terminology and requires a minimum number of them to match. The detector for real NCCS certification reports uses this approach.

To support a new document type, add a new function to `chunking/detectors.py` using whichever pattern fits. Choose signals that are specific enough that they would not appear in documents already handled by the existing detectors. Once written, import the new function in `chunking/router.py` alongside the existing detector imports so the router can call it.

9.5 Write a Chunking Strategy

The most important extension point is the chunking layer. A generic paragraph-based chunker produces poor retrieval results for structured documents such as standards tables or protocol requirement matrices. A dedicated chunker must be written for each new document type.

9.5.1 Create a New Chunker Module

Create a new file in the `chunking/` directory named after the document type, for example `chunking/authentication.py`. The chunker function must return a `list[dict]` where each dict contains at minimum a `text` field and an identifier field. Use `"clause"` as the identifier key for standards documents and `"section"` for guides and reports. These values must match the `id_field` used by `router.py` when it assembles the final metadata record for storage.

9.5.2 Embedding Clause References in Chunk Text

The most critical design rule is that every chunk must be self-contained. The language model receives only the chunk text as context at generation time and it does not see metadata fields such as `clause` or `source_file` as structured data. Those fields are used by the retrieval system to filter and rank results in Qdrant, but they are never passed to the model.

This means that if a clause reference exists only in the `clause` metadata field and not in the `text` field, the model has no way to cite it. Clause references must therefore be embedded directly inside the `text` field at ingestion time, so that when a chunk is retrieved and passed to the model, the reference is already present in the text the model reads.

The hardening guide chunker in `chunking/hardening.py` demonstrates this pattern. The table ID, row number, protocol name, allowed status and requirement are all assembled into the `text` field during chunking:

```
text = (
    f"Hardening Guide, Table {current_table_id} "
    f"({current_table_name}), Row {sl}\n"
    f"Protocol/Algorithm: {item}\n"
    f"Allowed: {allowed}\n"
    f"Hardening Requirement: {req}\n"
)
```

Listing 9.1. Clause reference embedded in chunk text from hardening.py

When this chunk is retrieved at runtime, the model reads the table reference and requirement directly from the text and can cite them correctly in the generated

narrative.

When writing a chunker for a new domain, follow the same pattern. Whatever reference information the model will need to cite such as clause numbers, table references, section headings and requirement identifiers must be included as part of the `text` string during chunking, not stored separately in metadata.

9.5.3 Choosing a Chunking Strategy

Select a strategy appropriate to the document structure.

Table-structured documents such as a matrix of authentication methods with permitted or prohibited status should use a state machine parser, as implemented in `chunking/hardening.py`. Multi-page tables are reassembled by detecting and discarding continuation markers using `CONTINUED_PATTERN` from `chunking/base.py`. Each table row becomes a single chunk containing the protocol or algorithm name, its allowed status, the hardening requirement text and any clause references.

Prose-structured documents with numbered clauses should split by clause boundary. Each chunk should contain one clause or sub-clause with its full requirement text. The `CLAUSE_PATTERN` and `SECTION_PATTERN` regular expressions in `chunking/base.py` are available for this purpose.

Report-style documents containing test case blocks should follow the pattern in `chunking/nccs_reports.py`. The `split_real_nccs_report` function locates test case block boundaries with a regular expression, extracts the Test Observations paragraph as the primary content per block. Each chunk is prefixed with a result tag such as `[RESULT:PASS]` or `[RESULT:FAIL]` and the DUT name. This tagging is consumed by `_format_style()` in `prompts.py`, which labels each style example with its outcome so the language model has context when generating narratives. New report-style chunkers for other domains must follow the same tagging convention.

9.6 Register the Chunker in the Router

All routing logic is centralised in `chunking/router.py` inside the `chunk_document()` function. The router dispatches to chunkers based on **collection name first**, then detector result within that collection. A new chunker must therefore be registered under the correct collection branch.

The three existing collection branches are `itsar-docs`, `hardening-guides` and `ncss-style`. If the new document belongs to an existing collection, add a new `elif` detector branch within it. If the new domain requires a separate collection, add a new top-level `elif collection == "..."` branch before the generic fallback.

The following example adds an authentication document type to the `itsar-docs` collection:

```
# Add to imports at the top of router.py:
from chunking.detectors import is_authentication_doc
from chunking.authentication import split_authentication_doc

# Inside chunk_document(), within the itsar-docs branch:
elif collection == "itsar-docs":
    if is_itsar001962411(full_text):
        # ... existing ...
    elif is_test_cases_doc(full_text):
        # ... existing ...
    elif is_authentication_doc(full_text):          # new branch
        print("  Chunking strategy: authentication doc")
        pages      = extract_pages_from_pdf(filepath) \
                     if filepath.endswith(".pdf") else [full_text]
        raw_chunks = split_authentication_doc(pages, filename)
        id_field   = "clause"
    else:
        # ... generic fallback ...
```

Listing 9.2. Adding a new detector branch in router.py

The `id_field` value must match the key used in the dicts returned by the new chunker. The router automatically appends all remaining metadata fields (`source_file`, `file_hash`, `version`, `is_latest`, `chunk_index`, `ingested_at`, `char_count`) to every chunk before storing it. The chunker itself only needs to return `text` and the identifier field.

9.7 Ingest and Verify

Once the detector and chunker are registered, ingest the new documents using the ingestion manager:

```
python ingest.py
```

Listing 9.3. Launching the ingestion manager

Select the option to ingest a new document and use the dry-run preview before committing. Inspect the preview output and verify the following:

- The correct chunking strategy name is printed to the console, confirming the detector matched as expected.
- Clause references are correctly embedded in each chunk text field, not only in metadata.
- No table rows are missing, merged or split incorrectly across page boundaries.

- The chunk count is consistent with the number of rows or clauses in the source document.

If the preview reveals structural errors, correct the chunker and re-run the preview before committing. Do not commit a malformed ingestion since incorrect chunks degrade retrieval quality silently without producing any runtime errors.

After committing, run the health check from within the ingestion manager to confirm the collection is populated and test queries return relevant results.

9.8 Add Prompt Templates

All prompt builder functions are defined in `prompts.py`. Each builder assembles a complete prompt from a task instruction, retrieved context blocks and the shared `FORMATTING_RULES` constant. Add a new builder function for each narrative type required by the new domain.

9.8.1 The FORMATTING_RULES Constant

`FORMATTING_RULES` is defined once at the top of `prompts.py` and must be appended to every prompt via `{FORMATTING_RULES}` in the f-string. It enforces formal technical English, prohibition of first-person language, named algorithm citations, paragraph-only output and strict JSON-only responses. Do not duplicate or modify it for new prompt types. These rules apply universally across all domains.

If a new domain introduces additional valid citation forms, update rule 4 in `FORMATTING_RULES` accordingly.

9.8.2 Prompt Builder Structure

Each builder function accepts a `finding_data` dict containing the structured test results and retrieved chunk lists from the vector store. Context is formatted using two helper functions already defined in `prompts.py`: `_format_chunks()` for ITSAR and hardening guide chunks, and `_format_style()` for NCCS style example chunks.

The JSON response schema must be consistent across all prompt types:

```
{
  "narrative": "the generated paragraph",
  "sources_cited": ["list of source document names referenced"],
  "confidence": "high/medium/low"
}
```

Listing 9.4. Required JSON response schema for all prompt builders

Note

The **Evidence Verdict Prompt** is a specialized exception to this schema, returning `ai_verdict` and `tester_action` fields instead of a narrative. `validate_output()` is configured to handle both structural variants.

This schema is validated by `validate_output()` in `llm.py` before any narrative is returned to the caller. If a new prompt builder requires a different response structure, for example, additional fields specific to the new domain then `validate_output()` in `llm.py` must also be updated to expect and accept the new schema. A new prompt builder that returns a different schema without updating `llm.py` will cause validation to fail on every call and fall through to the fallback generator, which produces a deterministic string-formatted narrative marked with `confidence:low` in the metadata.

9.9 Add API Endpoints

Create a new router file under `api/` for the new domain. Define FastAPI route handlers that accept structured test findings, retrieve context from the vector store using the search functions in `rag.py`, call the new prompt builder and pass the assembled prompt to `llm.py`.

Define request and response models for all new endpoints in `api/models.py`, consistent with the existing Pydantic model structure. Document the new endpoints in a table following the format of Table 3.1 in this report.

9.10 Extend the Validation Layer

Open `api/validation_checks.py` and add a new category of deterministic checks for the new domain. Each check must return a structured result containing the check name, category, severity, and finding text, consistent with the format of the existing checks.

Assign severity levels consistently: `ERROR` for issues that invalidate the compliance verdict (e.g., a missing section), `WARNING` for inconsistencies that require auditor review, and `INFO` for harmless observations. If the new domain includes rules that can be mathematically verified (e.g., an exact list of approved configurations), implement a **Deterministic Override** to automatically assign a `PASS` and safely bypass the AI evaluation.

Register all new checks in the main `run_all_checks()` function. Because the validation architecture is unified, these new checks will automatically execute when `/validate-uploaded-report` is called, and will seamlessly be cross-referenced against the AI Evidence Verdicts to generate discrepancy alerts.

9.11 Update the Report and Evidence Parsers

Because the system relies on a dual-parser architecture, extending to a new domain requires updating both extraction layers:

- **`report_parser.py`**: If the new domain introduces new structured tables in the DOCX (e.g., a matrix of supported authentication protocols), add extraction logic here. This data exclusively feeds the deterministic Python checks.
- **`parse_evidence.py`**: If the new domain relies on new terminal tools or evidence formats (e.g., a custom CLI output), update the regex boundaries to accurately slice this raw text. This ensures the AI Evidence Verdict prompt receives clean, uncorrupted logs for its independent evaluation.

If new fields or terminal blocks are not added to these parsers, the upload validation path will be blind to them and safely default to `INCONCLUSIVE`.

9.12 Update ai_client.py

`ai_client.py` is the interface layer between the testing system and the RAG engine's FastAPI service. It is not part of the RAG engine itself; it lives on the testing system side and makes HTTP calls to the engine endpoints.

Each domain-specific capability in the RAG engine has a corresponding function in `ai_client.py`. The existing functions reflect the unified architecture:

- `get_ai_content()` — calls `/enrich-report` to retrieve the final AI expert conclusion.
- `get_algorithm_explanations()` — calls `/explain-algorithms` for weak algorithm explanations.
- `validate_uploaded_report_docx()` — calls `/validate-uploaded-report` to upload the generated DOCX, triggering the dual-parsers, deterministic checks, and AI evidence verdicts in a single unified flow.

When a new domain is added and new API endpoints are created in the RAG engine, corresponding client functions must be added to `ai_client.py`. Each function must follow the same pattern as the existing ones:

- Check service availability or handle the failure gracefully.
- Wrap the request in a `try/except` block and a general `Exception` catch.
- Return a safe, well-defined fallback value on any failure.
- Print a status line using the `[AI Client]` prefix so failures are visible in the console log.

Also add domain-specific helper functions for safe response extraction. These helpers allow the testing system to extract individual fields from the response without needing to know the complex JSON structure or handle missing keys directly.

The `AI_SERVER` base URL and `TIMEOUT` values at the top of the file apply to all requests. The timeout is set to 600 seconds because CPU-based inference is slow. If a new domain involves longer generation tasks, increase the per-request timeout in the new client function rather than changing the global value.

9.13 Update validation_report.py

`validation_report.py` generates the internal NCCS validation DOCX. It is produced by `generate_validation_report()` after the compliance report is evaluated. Rather than calling multiple disjointed endpoints, it now takes the consolidated outputs from the unified validation pipeline: the deterministic Python checks, the extracted evidence, and the independent AI Evidence Verdicts.

The document currently contains seven sections:

1. Validation Summary: timestamp, DUT name, TC1-TC10 results, issue count.
2. Overall Validation Verdict: pass or fail with colour coding.
3. Deterministic Check Results: a table of every hardcoded Python check run.
4. Per-TC Evidence Analysis: a matrix comparing Automation vs. AI Verdicts.
5. Discrepancy Alerts: automated flags for any verdict mismatches.
6. Tester Action Items: AI-generated checklist of remediation steps.
7. Auditor Acknowledgement: sign-off section.

When a new domain is added, update `validation_report.py` in the following ways:

- **Validation Summary table**: if the new domain introduces additional test cases beyond TC1 through TC10, add the new result fields to the `summary_data` list in Section 1.
- **Deterministic Check Results table**: no changes are needed here. This table dynamically iterates over the full issues list returned by `validation_checks.py`, so new domain checks will appear automatically.
- **Discrepancy Alerts & Tester Action Items**: These sections are dynamically populated based on mismatches between the parsers and the AI. No structural changes to the DOCX generator are needed unless the new domain requires highly custom formatting for its alerts.

The colour scheme and formatting helpers are reusable for any new sections added to the report. Use them directly rather than introducing new formatting patterns.

9.14 Verification Checklist

Before deploying an extension to the production environment, verify the following:

1. The new detector function correctly identifies the new document type and does not produce false positives against existing document types. Confirmed by running the ingestion manager against the new document and verifying that the correct chunking strategy name is printed.
2. The ingestion dry-run preview shows correctly structured chunks with clause references embedded in the `text` field.
3. The ingestion health check confirms the collection is populated and test queries return relevant results.
4. The new prompt builder injects retrieved context correctly, includes `FORMATTING_RULES`, and returns the standard JSON schema.
5. The language model output passes `validate_output()` in `llm.py` without triggering retry or fallback behaviour for representative inputs.
6. All new deterministic validation checks execute correctly and return the expected severity for both passing and failing report examples.
7. **Both the report parser and evidence parser** correctly extract all new structured fields and raw terminal logs from a sample generated report.
8. The unified validation flow correctly surfaces **Discrepancy Alerts** if the new domain's structured tables are intentionally manipulated to contradict the raw terminal evidence.

9.15 Important Constraints

The following constraints apply to all extensions.

- **Offline operation must be preserved.** No extension may introduce a dependency on an external API, cloud service, or internet connectivity at runtime.
- **Qdrant file lock.** Qdrant in local file mode allows only one process to access `qdrant_storage/` at a time. Do not run the ingestion manager and the API service simultaneously. If the service was killed uncleanly, delete the `.lock` file inside `qdrant_storage/` before restarting.

- **Version tracking.** When a new version of a standards document is ingested, use the ingestion manager's replace workflow. Do not ingest a new version as a separate document with a different name, as both versions will then appear in retrieval results simultaneously.

  For ITSAR standard documents, the ingestion manager also triggers `delta.py` automatically. It compares the old and new versions clause by clause, classifies each change as `stricter`, `relaxed`, `modified`, or `unchanged`, and saves a structured JSON report to the `delta_reports/` folder. This gives the auditor a clear record of exactly what changed between versions before any new test runs begin. If a new domain has its own versioned standards document, a domain-specific delta comparison can be added to `delta.py` following the same pattern.

- **Manual chunks for critical tables.** If the core rules table of a new standard is structurally fragile under PDF extraction due to merged cells, footnotes, or complex column layouts, write the chunks manually as done in `chunking/itsar.py`.

National Centre for Communication Security

Department of Telecommunications, Government of India

Developer and Tester Extension Manual

AI-Powered Compliance Evaluation Engine
for ITSAR Security Testing

System Version:	1.0.0
Classification:	Restricted — Internal Use Only
Deployment Mode:	Air-Gapped (Fully Offline)
Prepared by:	Angeline Neha R
	[SC Intern]
Under the guidance of:	Sh. Sumit Singh
	[ADG (SC-1)]

Contents

Chapter 1
Introduction and Purpose

1.1 About This Manual

This manual is the primary reference for any developer or tester who needs to extend, maintain, or operate the RAG Engine component of the ITSAR Cryptographic Compliance Testing System. While the Technical Design and Implementation Report documents what the system does and how it was built, this manual focuses exclusively on what a developer needs to know to extend the system safely and correctly without breaking existing functionality.

1.2 What This Manual Covers

This manual covers the following:

- Network access requirements and how to verify connectivity to the AI server before starting work.
- Complete environment setup, including exact dependency versions and configuration.
- How to run the system and confirm it is operating correctly.
- A step-by-step guide to extending the system for new compliance domains or test cases.
- Detailed guidance on the two parser components, chunking strategies, prompt engineering, and the validation layer.
- A verification checklist to confirm an extension is correct before deployment.

Chapter 2
System Architecture Quick Reference

2.1 Component Summary

The engine consists of the following components. Each is described in detail in later chapters. This table serves as a quick orientation reference.

Component	Primary File(s)	Responsibility
Document Ingestion Pipeline	`ingest.py`, `chunker.py`, `chunking/`	Converts reference documents into searchable chunks and stores them in the vector database.
Vector Store	`rag.py`	Qdrant database running in local file mode. Stores and retrieves 384-dimensional embeddings.
LLM Layer	`llm.py`	Communicates with the locally-hosted `llama3:8b` model via Ollama. Handles retry logic, fallback generation, and output validation.
Prompt Engine	`prompts.py`	Assembles structured prompts for each narrative type from retrieved context and shared formatting rules.

Component	Primary File(s)	Responsibility
Report Parser	`report_parser.py`	Extracts structured table data from a generated compliance report DOCX. Feeds the deterministic validation checks exclusively.
Evidence Parser	`parse_evidence.py`	Extracts raw terminal and tool output from the same DOCX. Feeds the AI Evidence Verdict prompt exclusively.
Validation Layer	`api/validation_checks.py`	Deterministic Python checks for schema, completeness, verdict consistency, and internal consistency.
FastAPI Service	`main.py`, `api/`	Exposes all engine capabilities as REST endpoints and serves the standalone web validation interface.
Ingestion Manager	`ingest.py`	Command-line tool for managing the document database including ingestion, versioning, and health checks.

Table 2.1. RAG engine component summary

2.2 Data Flow Summary

Data moves through the engine in three distinct phases. Understanding this flow is essential before making any extension, because a change to one phase can affect the inputs available to the next.

2.2.1 Ingestion Phase

Reference documents are processed offline before any testing begins. Each document is extracted, chunked using a strategy matched to its document type, embedded into

vectors by the `all-MiniLM-L6-v2` sentence transformer model, and stored in the Qdrant database. This phase is run once and repeated only when new or updated documents are added. The ingestion manager and the API service must never run simultaneously due to the Qdrant file lock (see Chapter **??**).

2.2.2 Live Execution Phase

During a live test run, the testing system calls the FastAPI service with structured test findings. The engine queries the vector database for relevant context, assembles a structured prompt, calls the language model, validates the output, and returns an enriched JSON payload containing the AI-generated narratives.

2.2.3 Validation Phase

After the compliance report DOCX has been generated, it is submitted to the unified validation endpoint. The dual-parser architecture extracts structured table data and raw terminal evidence from the same document in parallel. The structured data drives the deterministic Python checks. The raw terminal evidence is routed to the language model for independent evaluation. The engine cross-references both streams to produce a validation report containing discrepancy alerts and a prioritised tester action checklist.

2.3 API Endpoint Reference

The FastAPI service runs on `localhost:8000`. The following endpoints are available in the current system.

Endpoint	Method	Description
`/enrich-report`	POST	Generates the final AI Expert Conclusion narrative from structured test findings.
`/explain-algorithms`	POST	Returns per-algorithm explanations with ITSAR clause citations, processed in batches of three.
`/validate-uploaded-report`	POST	Accepts a report DOCX upload, runs the dual-parsers, deterministic checks, and AI evidence evaluations in a single unified flow.
`/health`	GET	Returns the status of the Ollama and Qdrant components.

Endpoint	Method	Description
/	GET	Serves the interactive standalone web validation interface.

Table 2.2. RAG engine API endpoint reference

2.4 Key Design Principles

The following principles govern the entire system. Every extension must respect them.

- **Offline-first.** No component may introduce a runtime dependency on an external API or internet connectivity. The only permitted network dependency is the internal Ollama server.
- **Self-contained chunks.** Every chunk stored in the vector database must contain all information the language model will need to cite it correctly, including references, table identifiers, and requirement text. Metadata fields are not passed to the model and cannot be relied upon for citation.
- **Separation of parsers.** The report and the evidence parser extract from the same document but feed entirely different downstream systems. Their outputs must never be mixed. Feeding structured table data to the AI prompt or feeding raw terminal logs to the deterministic checks will produce incorrect results.

Chapter 3

Network and Access Requirements

3.1 Overview

The RAG engine depends on a locally-hosted language model served via Ollama. This server is hosted on the internal NCCS office network and is not accessible from outside it. If the developer's machine is not connected to the correct network, all language model calls will fail.

Critical Prerequisite

The developer's machine must be connected to the internal NCCS office Wi-Fi before starting the RAG engine. The Ollama AI server is hosted at `http://10.61.6.71:3000` and is reachable only from within the internal office network. All LLM calls will silently fall back to deterministic string-formatted narratives if this server is unreachable. These fallback narratives are marked `confidence: low` but no runtime error is raised. Do not begin development or testing on a machine that is not connected to the office network.

3.2 The Ollama AI Server

This server is managed separately from the RAG engine and must be running before the engine is started. The RAG engine does not start or manage the Ollama process. If the server is down, contact the system administrator responsible for the Ollama host machine.

Authentication credentials for the Ollama server are supplied exclusively through the `.env` configuration file, which is described in Section 4.6. The `.env` file must be present in the root of the `rag_engine/` directory before the engine is started.

3.3 Verifying Network Connectivity

Before running the engine for the first time on any machine, verify that the Ollama server is reachable. Run the following command from a terminal:

```
curl -I http://10.61.6.71:3000
```

Listing 3.1. Verifying reachability of the Ollama server

A reachable server will return an HTTP response with a status code. A connection refused error or a timeout confirms that the machine is not on the correct network or the server is down.

Connectivity can also be verified after the engine has started by calling the health endpoint:

```
curl http://localhost:8000/health
```

Listing 3.2. Checking engine health via the API

The health endpoint returns the status of both the Ollama connection and the Qdrant database. A healthy response will confirm that both components are reachable. If the Ollama status field reports an error, verify network connectivity before proceeding.

Note

The health endpoint only confirms that the Ollama server responded to a ping-style request. It does not confirm that the credentials in the `.env` file are correct. A full end-to-end LLM call must be made to confirm that authenticated generation is working correctly.

3.4 Credentials

The Ollama server requires authentication. Credentials are configured through the `.env` file only. These credentials must never be committed to version control or shared.

Security Notice

The `.env` file contains authentication credentials. It must not be committed to any version control system. Verify that `.env` is listed in `.gitignore` before making any commits to the repository.

Chapter 4

Environment Setup and Running the System

4.1 Overview

This chapter describes how to set up the RAG engine from scratch on a new machine and how to start it correctly.

4.2 Prerequisites

The following must be available on the machine before setup begins:

- Python 3.11 or higher. The engine uses syntax and standard library features that are not available in earlier versions.
- The `rag_engine/` source directory, provided as a compressed archive.
- Active connection to the internal NCCS office network (see Chapter 3).
- Ollama server credentials, obtained from the .env file.

4.3 Python Virtual Environment

All dependencies must be installed inside a dedicated virtual environment.

Create and activate the virtual environment from the `rag_engine/` directory:

```
python3 -m venv venv
source venv/bin/activate
```

Listing 4.1. Creating and activating the virtual environment

On Windows, replace the second command with:

```
venv\Scripts\activate
```

Listing 4.2. Activating the virtual environment on Windows

4.4 Installing Dependencies

Install all required dependencies using the pinned `requirements.txt` file:

```
pip install -r requirements.txt
```

Listing 4.3. Installing all dependencies

Important — Use Exact Versions

All packages in `requirements.txt` are pinned to specific versions. Do not upgrade any package without testing the full system end-to-end, as version drift in `qdrant-client`, `sentence-transformers`, or `fastapi` can silently break embedding dimensions, API contracts, or serialisation behaviour. The pinned versions listed in Section 4.5 are the only versions that have been validated against the full system.

4.5 Dependency Reference

The following table lists every package in `requirements.txt` with its pinned version and the reason it is present. This table exists so that a developer understands what breaks if any version is changed.

Package	Version	Purpose
`fastapi`	0.111.0	REST API framework. The endpoint contracts and Pydantic model integration are validated against this version.
`uvicorn`	0.29.0	ASGI server that serves the FastAPI application.
`qdrant-client`	1.9.1	Interface to the Qdrant vector database. The local file mode API and collection schema are specific to this version. Upgrading may change the storage format and corrupt the existing `qdrant_storage/` directory.
`sentence-transformers`	2.7.0	Loads and runs the `all-MiniLM-L6-v2` embedding model. The 384-dimensional vector output is fixed by this version.
`pymupdf`	1.24.3	PDF text and page extraction used by the chunking pipeline and the evidence parser.

Package	Version	Purpose
python-docx	1.1.2	DOCX parsing and generation used by the report parser, evidence parser, and validation report generator.
langchain	0.2.1	Provides text splitting utilities used in the generic fallback chunker.
langchain-community	0.2.1	Community extensions for LangChain. Must match the langchain version exactly to avoid import errors.
requests	2.32.2	HTTP client used by llm.py to communicate with the Ollama server.
python-dotenv	1.0.1	Loads the .env configuration file into environment variables at startup.
rich	13.7.1	Terminal formatting used by the ingestion manager's interactive menu and health check output.
typer	0.12.3	CLI framework for the ingestion manager. Must match the rich version as Typer depends on it for output.
tqdm	4.66.4	Progress bars during document ingestion and embedding generation.
numpy	1.26.4	Numerical operations used internally by sentence-transformers. Upgrading to NumPy 2.x is not supported and will break the embedding pipeline.

Table 4.1. Pinned dependency reference for requirements.txt

4.6 Configuring the Environment File

All runtime configuration is managed through a .env file placed in the root of the rag_engine/ directory.

```
# Ollama AI Server
OLLAMA_HOST=http://10.61.6.71:3000
OLLAMA_EMAIL=croom.nccs-dot@gmail.com
```

```
OLLAMA_PASS=Temp$123
OLLAMA_MODEL=llama3:8b

# LLM Generation Parameters
LLM_MAX_TOKENS=2500
LLM_TEMPERATURE=0.1
LLM_RETRY_ATTEMPTS=2

# Qdrant Vector Store
QDRANT_PATH=./qdrant_storage

# Retrieval Parameters
TOP_K_ITSAR=2
TOP_K_STYLE=1
MIN_RELEVANCE_SCORE=0.5

# FastAPI Service
AI_SERVICE_HOST=0.0.0.0
AI_SERVICE_PORT=8000
```

Listing 4.4. Complete `.env` configuration

The following notes apply to specific fields:

- `LLM_TEMPERATURE` is set to `0.1` to produce consistent, formal output. Do not increase this value in a production environment, as higher temperatures produce less deterministic narratives that are harder to validate.
- `LLM_RETRY_ATTEMPTS` controls how many times the engine retries a failed generation before falling back to the deterministic narrative generator.
- `MIN_RELEVANCE_SCORE` sets the minimum cosine similarity threshold for vector search results. Results below this score are discarded. Lowering this value will increase retrieval recall but may introduce irrelevant context into prompts.

4.7 Document Ingestion

A pre-populated `qdrant_storage/` directory is included in the source archive for standard deployments. This means document ingestion does not need to be repeated on a fresh machine unless the reference documents have changed or the storage directory has been corrupted.

If ingestion must be performed when adding new reference documents or setting up on a machine where the storage directory was not provided, launch the ingestion manager from the `rag_engine/` directory:

```
python ingest.py
```

Listing 4.5. Launching the ingestion manager

The ingestion manager presents an interactive menu. On a fresh machine, select the option to re-ingest all documents from scratch. Use the dry-run preview option before committing any ingestion to verify that documents are being chunked correctly.

Critical - File Lock Constraint

Qdrant runs in local file mode and permits only one process to access `qdrant_storage/` at a time. Never run `ingest.py` and the API service simultaneously. If the API service is running, stop it before launching the ingestion manager. If the service was terminated uncleanly and the ingestion manager reports a lock error, delete the `.lock` file inside `qdrant_storage/` and retry.

4.8 Starting the RAG Engine

Once the virtual environment is active, the `.env` file is in place, and the Qdrant storage is populated, start the FastAPI service from the `rag_engine/` directory:

```
uvicorn main:app --host 0.0.0.0 --port 8000
```

Listing 4.6. Starting the RAG engine service

The service will start and print startup log lines to the terminal. Once the line `Application startup complete` appears, the engine is ready to accept requests.

The web interface for standalone report validation is served at the root path and can be accessed at:

`http://localhost:8000`

To confirm the engine is fully operational, call the health endpoint:

```
curl http://localhost:8000/health
```

Listing 4.7. Confirming the engine is operational

A fully operational engine will return a JSON response confirming that both Ollama and Qdrant are reachable. If either component reports an error, refer to Section 3.3 for Ollama connectivity checks, and verify that `qdrant_storage/` is present and not locked for Qdrant issues.

Note

The engine binds to `0.0.0.0`, which makes it accessible from other machines on the same network. In the NCCS testing environment this is intentional, as the

broader testing system calls the engine over the local network. If the engine is being run on a personal development machine, replace `0.0.0.0` with `127.0.0.1` to restrict access to localhost only.

4.9 Startup Sequence Summary

The correct startup sequence for a session is summarised below. Follow this order every time the system is started.

1. Connect the machine to the internal NCCS office network.
2. Verify Ollama server reachability with `curl -I http://10.61.6.71:3000`.
3. Navigate to the `rag_engine/` directory.
4. Activate the virtual environment with `source venv/bin/activate`.
5. Confirm the `.env` file is present and contains valid credentials.
6. Start the FastAPI service with `uvicorn main:app --host 0.0.0.0 --port 8000`.
7. Confirm startup with `curl http://localhost:8000/health`.

Important - Do Not Run Ingestion and the API Simultaneously

If document ingestion is required in the same session, stop the API service first, run `python ingest.py`, wait for the health check to pass, and then restart the API service. Running both processes at the same time will corrupt the Qdrant storage.

Chapter 5
Codebase Walkthrough

5.1 Overview

This chapter describes the responsibilities of every key file in the `rag_engine/` directory.

5.2 Top-Level Files

File	Responsibility
`main.py`	FastAPI application entry point. Mounts all API routers, serves the web UI at `/`, and exposes the `/health` endpoint.
`rag.py`	All Qdrant interactions: collection management, embedding generation, chunk storage, deletion, and all search functions. The single interface between the rest of the codebase and the vector database.
`llm.py`	All Ollama interactions: authentication, generation, JSON extraction, output validation, retry logic, and fallback narrative generation.
`prompts.py`	All prompt builder functions. Each function assembles a complete prompt from retrieved context and finding data. Contains the shared `FORMATTING_RULES` constant.
`chunker.py`	Public entry point for document chunking.
`ingest.py`	Interactive CLI for managing.
`report_parser.py`	Extracts structured table data (device name, TC results, algorithm lists) from a generated compliance report DOCX. Feeds the deterministic validation checks exclusively.

File	Responsibility
parse_evidence.py	Extracts raw terminal and tool output blocks from the same DOCX. Feeds the AI Evidence Verdict prompt exclusively.
delta.py	Compares two ingested ITSAR versions clause by clause and writes a structured JSON delta report to delta_reports/.

Table 5.1. Top-level file responsibilities

5.3 The api/ Package

Each file in api/ is a FastAPI router that handles one category of endpoint. They are mounted in main.py.

File	Responsibility
models.py	All Pydantic request and response models. Every new endpoint must define its input and output models here.
helpers.py	Shared utility functions used across routers.
enrich.py	/enrich-report, /conclusion, /weak-cipher and related enrichment endpoints.
explain.py	/explain-algorithms endpoint. Processes algorithms in batches of three per LLM call.
validate.py	/validate-report and /validation-synthesis endpoints.
upload.py	/validate-uploaded-report endpoint. Accepts the DOCX upload, invokes both parsers, runs all checks, and returns the consolidated validation report.
validation_checks.py	All deterministic validation check functions organised by category. The primary source of truth for compliance verdicts.

Table 5.2. api/ package file responsibilities

5.4 The chunking/ Package

File	Responsibility
base.py	Shared utilities: text extraction from PDF/-DOCX/TXT, file hashing, version detection, page filtering, and all shared regex patterns.
detectors.py	Document type detection functions.
router.py	Routing logic. Receives a file path and collection name, calls detectors, and dispatches to the correct chunker.
itsar.py	Structure-aware chunker for the ITSAR001962411 standard.
hardening.py	Table-aware chunker for hardening guides. Uses a state machine to reconstruct multi-page tables row by row.
test_cases.py	Row-per-TC chunker for the test cases document.
nccs_reports.py	Observation-block chunker for approved NCCS report examples.
generic.py	Fallback chunkers for unrecognised documents.

Table 5.3. chunking/ package file responsibilities

5.5 How main.py Wires Everything Together

main.py is intentionally thin. It imports four routers and mounts them, then defines /health and /download. The health endpoint calls check_ollama() from llm.py and health_check() from rag.py and merges the results.

```
from api.enrich  import router as enrich_router
from api.explain import router as explain_router
from api.validate import router as validate_router
from api.upload  import router as upload_router
app.include_router(enrich_router)
app.include_router(explain_router)
app.include_router(validate_router)
app.include_router(upload_router)
```

Listing 5.1. Router mounting in main.py

When a new compliance domain is added, its router is imported and mounted here in exactly the same way. Nothing else in `main.py` changes.

5.6 How `rag.py` Manages the Vector Store

`rag.py` uses lazy initialisation for both the Qdrant client and the embedding model. Neither is loaded until the first call that needs them. The embedding model is loaded in offline mode.

This guarantees the model never attempts a network download at runtime. The model must be cached locally before first use. If the model is missing, the engine will fail to start.

5.7 How `llm.py` Handles Generation

Every LLM call goes through `generate()`, which performs authentication, calls Ollama, extracts JSON from the response, validates it using `validate_output()` and retries on failure.

`validate_output()` checks narrative length, verifies that algorithm names in the narrative are present in the finding data, and rejects any invented ITSAR clause numbers. If all retry attempts fail, `_build_fallback()` generates a deterministic narrative from the finding data using string formatting and marks it `confidence: low`.

Note

Three separate generation functions exist for different response schemas. `generate()` is for narrative endpoints. `generate_batch()` is for algorithm explanations which return an `explanations` dict. `generate_evidence_verdicts()` is for per-TC adjudication which returns an `ai_tc_analysis` dict.

Chapter 6

Implementing the Dual Parsers

6.1 Overview

The dual-parser architecture is the most critical component to understand before extending the system. Both parsers operate on the same generated compliance report DOCX, but they extract entirely different content for entirely different purposes.

- `report_parser.py` extracts **structured table data** and feeds the **Python checks**.
- `parse_evidence.py` extracts **raw terminal and tool output** and feeds the **AI Evidence Verdict prompt**.

6.2 `report_parser.py` — Structured Table Extraction

6.2.1 How It Works

The parser iterates over all tables in the DOCX using `python-docx`. It identifies each table by inspecting its header row for known column names. Once a table is identified, it reads the relevant cells row by row and stores the values in a structured dict.

6.2.2 Adding Extraction for a New Domain

When a new domain produces a report with a different structure or a new table format, the correct approach is to create a dedicated parser module rather than modifying `report_parser.py` directly. Editing the existing parser risks breaking extraction for the SSH, HTTPS, and SNMP domains that already depend on it.

Create a new file in the `rag_engine/` directory. Once the new parser module is written, register it in `api/upload.py` alongside the existing parser. The upload handler should detect which parser to invoke based on the report type, which can be determined from a field in the request payload or from a document type signal in the DOCX itself. The same principle applies to `parse_evidence.py`. If the new domain's evidence blocks use a different heading format or come from different tools, create `parse_evidence_auth.py` with its own boundary patterns and register it in `api/upload.py` the same way. Never modify the existing evidence parser's boundary patterns to accommodate a new domain, as boundary changes affect the slice points for all existing test cases simultaneously.

6.3 `parse_evidence.py` — Raw Terminal Extraction

6.3.1 What It Extracts

`parse_evidence.py` locates and extracts the raw terminal and tool output blocks embedded in the compliance report DOCX. Each tool's output is extracted separately and stored under a key corresponding to its test case.

Never Bypass the Anti-Hallucination Filter

The parser explicitly filters out the AI-generated summary tables and observation paragraphs that appear in the same DOCX. Do not modify `parse_evidence.py` in a way that allows AI-generated summary tables or observation text to be included in the evidence output. Any such content passed to the Evidence Verdict prompt will cause the model to loop on its own prior narratives, producing verdicts that appear confident but are entirely self-referential.

6.3.2 How Boundaries Are Defined

The parser uses regex boundary patterns to slice the raw text of the DOCX into evidence blocks. The parser reads between two consecutive boundaries to extract the raw text for one test case.

The boundary patterns must match the exact heading text that the testing system inserts before each evidence block in the DOCX.

6.3.3 Adding Boundaries for a New Domain

When a new domain introduces a new terminal tool or a new evidence section heading, add its boundary pattern and after adding the boundary, verify extraction on a real sample report and inspecting the output. The extracted text should contain only the raw terminal output with no AI-generated content.

Chapter 7
Prompt Engineering Guidelines

7.1 Overview

All prompt builder functions live in `prompts.py`. This chapter explains the rules that govern every prompt in the system.

7.2 The `FORMATTING_RULES` Constant

`FORMATTING_RULES` is defined once at the top of `prompts.py` and must be appended to every prompt via `{FORMATTING_RULES}` in the f-string.

1. Formal technical English only. No casual language.
2. No first-person language: `I`, `we`, `our`, `you` are prohibited.
3. Algorithm names cited in the narrative must come from the `FINDING DATA` section of the prompt only.
4. No bullet points. Paragraph form only.
5. Response must be valid JSON only. No text before or after the JSON object.

Never Duplicate or Modify `FORMATTING_RULES`

Do not copy `FORMATTING_RULES` into a new prompt builder with modifications. All changes to formatting rules must be made in the single definition at the top of `prompts.py` and will apply to every prompt automatically.

7.3 The Required JSON Response Schema

Every narrative prompt must request the following JSON structure:

```
{
  "narrative":      "the generated paragraph",
  "sources_cited":  ["list of source document names
   referenced"],
  "confidence":     "high/medium/low"
}
```

Listing 7.1. Standard JSON response schema for narrative prompts

7.4 Context Assembly Functions

Three helper functions in `prompts.py` format retrieved chunks for prompt injection. Use the correct one for each context type:

- `_format_chunks(chunks, label)` — formats ITSAR or hardening guide chunks. Use for all domains unless algorithm hallucination is a risk in that prompt.
- `_format_chunks_scrubbed(chunks, label, legitimate_algos)` — same as above but redacts algorithm names not in the finding data. Use for conclusion and similar domain-summary prompts.
- `_format_style(chunks)` — formats NCCS style example chunks. Automatically detects and labels `[RESULT:PASS]` and `[RESULT:FAIL]` tags so the model has outcome context when generating narratives.

7.5 Output Validation in `llm.py`

`validate_output()` runs automatically after every `generate()` call. It checks:

- Narrative word count between 15 and 200 words.
- Algorithm names in the narrative are present in the finding data (not hallucinated from context).
- No invented ITSAR clause numbers.
- No informal language markers.

If validation fails, the engine retries up to `LLM_RETRY_ATTEMPTS` times with progressively lower temperature and a stricter instruction appended to the prompt. If all attempts fail, the fallback generator produces a deterministic string-formatted narrative marked `confidence: low`.

Chapter 8
Validation Layer Extension

8.1 Overview

This chapter describes the design of the existing checks, the rules for adding new ones, and the interaction between the deterministic layer and the AI layer in `api/upload.py`.

8.2 Design Principle

The validation layer was deliberately narrowed after an earlier refactor. It now handles only what the AI cannot: structured data mathematics, file-system verification, hash presence checks, and table cross-referencing. Narrative quality, observation completeness, and tone consistency are delegated to the AI evidence evaluation.

8.3 Existing Check Categories

The current checks are organised into six categories, each implemented as a standalone function that receives `body`, `issues`, and `checks` as arguments and appends to them directly:

Function	What It Checks
`check_dut_integrity`	OS and configuration digest hash presence and minimum length (32 characters).
`check_nmap_analysis`	Cross-references Nmap port scan output against TC applicability flags. Port 22, 443, and 161 open/-closed vs TC marked applicable/NA.
`check_completeness`	Mandatory section headings, results table values vs computed TC results, screenshot embedding, and conclusion narrative length.
`check_protocol_version`	SSHv2 (protocol 2.0) confirmed from parser output or Nmap text.

Table 8.1. Existing deterministic check functions

8.4 Severity Levels

Every issue appended to the `issues` list must include a `severity` field. Use the following levels consistently:

- `HIGH` — the issue invalidates or contradicts the compliance verdict.
- `MEDIUM` — inconsistency that requires auditor review but does not automatically invalidate the report.
- `INFO` — harmless observation.

Passing checks are appended to the `checks` list with no severity field. The validation report renders issues and passing checks in separate tables.

8.5 Adding Checks for a New Domain

Add a new function to `api/validation_checks.py` following the exact signature pattern of the existing functions. Register the new function in `run_all_checks()` at the bottom of the file:

```
def run_all_checks(body: dict) -> tuple[list, list]:
    issues: list = []
    checks: list = []

    check_dut_integrity(body, issues, checks)
    check_nmap_analysis(body, issues, checks)
    check_completeness(body, issues, checks)
    check_verdict_consistency(body, issues, checks)
    check_protocol_version(body, issues, checks)
    check_tls_compliance(body, issues, checks)
    check_snmp_compliance(body, issues, checks)
    check_auth_compliance(body, issues, checks)  # new domain

    return issues, checks
```

Listing 8.1. Registering a new check in `run_all_checks()`

Chapter 9
Integration with the Broader Testing System

9.1 Overview

The RAG engine does not exist in isolation. It is called by the broader ITSAR testing system through two files that live on the testing system side: `ai_client.py` and `validation_report.py`. These files are not part of the `rag_engine/` directory. When a new domain is added to the RAG engine and new API endpoints are created, corresponding changes must be made to both files on the testing system side before end-to-end operation is possible.

9.2 `ai_client.py`

`ai_client.py` is the HTTP interface between the testing system and the RAG engine. It contains one function per RAG engine endpoint. The three existing functions map to the three main endpoints:

- `get_ai_content()` — calls `/enrich-report` to retrieve the final AI Expert Conclusion narrative.
- `get_algorithm_explanations()` — calls `/explain-algorithms` for per-algorithm explanations.
- `validate_uploaded_report_docx()` — calls `/validate-uploaded-report` to upload the generated DOCX and trigger the full unified validation pipeline.

When a new domain adds new endpoints to the RAG engine, add a corresponding client function to `ai_client.py`. Every client function must follow this exact pattern — no exceptions:

```
AI_SERVER  = "http://localhost:8000"
TIMEOUT    = 900

def get_auth_narratives(finding_data: dict) -> dict:
    """
    Calls /auth-narratives on the RAG engine.
    Returns the AI-generated authentication domain narratives,
```

```
    or a safe fallback dict on any failure.
    """
    try:
        response = requests.post(
            f"{AI_SERVER}/auth-narratives",
            json=finding_data,
            timeout=TIMEOUT
        )
        response.raise_for_status()
        print("[AI Client] /auth-narratives: OK")
        return response.json()

    except requests.RequestException as e:
        print(f"[AI Client] /auth-narratives failed: {e}")
        return {"narrative": "", "confidence": "low"}

    except Exception as e:
        print(f"[AI Client] /auth-narratives unexpected error: {e}")
        return {"narrative": "", "confidence": "low"}
```

Listing 9.1. Required pattern for new client functions in `ai_client.py`

Four rules apply to every client function without exception:

1. Check service availability or handle failure gracefully via the `except` block. Never let an unhandled exception from the AI client propagate into the testing system.
2. Wrap the request in a `try/except` block with a general `Exception` catch as the outermost handler.
3. Return a safe, well-defined fallback value on any failure. The fallback must be structurally identical to a successful response so that the testing system can proceed without branching on failure.
4. Print a status line using the `[AI Client]` prefix. This prefix is what makes AI client failures visible in the console log during a full test run.

Timeout Guidance

The global `TIMEOUT` is set to 900 seconds because CPU-based inference is slow. Do not lower this value globally. If a new domain involves longer generation tasks such as large multi-TC evidence evaluation, increase the timeout in the specific new client function using a local override rather than changing the global constant.

9.3 `validation_report.py`

`validation_report.py` generates the internal NCCS validation DOCX that is delivered to the auditor after every test run. It is called by `generate_validation_report()` and takes the consolidated outputs of the entire validation pipeline as arguments: the deterministic check results, extracted evidence, AI TC analysis, discrepancies, and tester action items.

The current validation report contains seven sections:

1. **Validation Summary** — timestamp, DUT name, TC1–TC10 pass/fail matrix, total issue count.
2. **Overall Validation Verdict** — pass or fail banner with colour coding.
3. **Deterministic Check Results** — table of every check run by `run_all_checks()`, its severity, and finding. This table is dynamically generated from the `issues` and `checks` lists and will automatically include new domain checks without any code change.
4. **Per-TC Evidence Analysis** — matrix comparing the Automation Verdict against the independent AI Verdict with the raw evidence snippet.
5. **Discrepancy Alerts** — automated flags for any verdict mismatches.
6. **Tester Action Items** — prioritised checklist (HIGH / MEDIUM / LOW) from `_build_tester_action_items()`.
7. **Auditor Acknowledgement** — sign-off section for manual confirmation before submission.

When a new domain is added, the only section that requires a code change is Section 1 if the new domain introduces test cases beyond TC1–TC10. In that case, add the new TC result fields to the `summary_data` list in the Validation Summary section:

```
summary_data = [
    # ... existing TC1-TC10 entries ...
    ("TC11 (Auth Method Check)", kwargs.get("tc11", "NA")),
    ("TC12 (Auth Rejection Test)", kwargs.get("tc12", "NA")),
]
```

Listing 9.2. Adding new TC fields to the validation summary table

Chapter 10
Testing and Verification Checklist

10.1 Overview

This checklist must be completed in full before any extension is deployed to the production environment. Each item is a mandatory verification step, not an optional quality check. Items that cannot be confirmed should be treated as failures and the corresponding extension step revisited.

10.2 Ingestion Verification

1. Run `python ingest.py` and use the dry-run preview on the new document. Confirm the correct chunking strategy name is printed to the console, confirming the detector matched and no existing detector was triggered by mistake.
2. Inspect the dry-run chunk output line by line. Confirm that clause references, table identifiers, and requirement text are embedded inside the `text` field of each chunk — not only in the metadata.
3. Confirm the chunk count is consistent with the number of rows or clauses in the source document. A significantly lower count indicates rows were merged; a higher count indicates rows were split.
4. Commit the ingestion and run the health check from within the ingestion manager. Confirm the new collection is populated and that test queries return relevant results with scores above the minimum threshold.

10.3 Prompt and LLM Verification

5. Call the new API endpoint with a representative `finding_data` payload. Confirm the response contains a `narrative` field (or the appropriate schema for the endpoint type) and that `confidence` is not `low`.
6. Confirm the response does not contain the `_meta.fallback: true` field. Its presence means all retry attempts failed and the response is a deterministic string, not an AI-generated narrative.
7. Confirm that algorithm names in the narrative are present in the `finding_data`

payload sent in the request and not copied from the ITSAR context. Inject a payload with a single known algorithm and verify that only that algorithm appears in the narrative.

8. Confirm no invented clause numbers appear in the narrative. Patterns such as `clause 2.6.1`, `clause 10.208`, or any bracketed clause reference not matching `ITSAR001962411, Chapter 2, Table 1` indicate hallucination.

10.4 Validation Layer Verification

9. Run `/validate-uploaded-report` with a known-good sample report for the new domain. Confirm all new deterministic checks appear in the Deterministic Check Results section of the validation report with the expected severity and finding text.

10. Run the same endpoint with a deliberately failing report — one where a **PASS** result is manually changed to **FAIL** in the structured table. Confirm a `HIGH` severity issue is raised in the correct check category.

11. Confirm that checks for the new domain respect the applicability flag — if the domain is marked not applicable in the report, the checks should append a passing note and return without raising issues.

10.5 Parser Verification

12. Call the new domain's report parser module directly on a sample report and inspect the output dict. Confirm all expected fields are present and correctly populated.

13. Call the new domain's evidence parser module directly and inspect the output. Confirm raw terminal text is present under the correct TC key and contains no AI-generated summary text.

14. Intentionally corrupt the structured table in the sample report (change a **PASS** to **FAIL** by editing the DOCX directly). Submit to `/validate-uploaded-report` and confirm a Discrepancy Alert fires in the validation report.

15. Remove an evidence block from the sample report and resubmit. Confirm the corresponding AI Verdict returns `INCONCLUSIVE` rather than a fabricated **PASS** or **FAIL**.

10.6 End-to-End Verification

16. Run a full test cycle for the new domain using the testing system with the RAG engine active. Confirm the generated compliance report contains the AI-

generated narrative sections for the new domain and that they are not fallback placeholder text.

17. Submit the generated report to `/validate-uploaded-report` via the web interface. Confirm the validation report downloads successfully and all seven sections are present.

18. Confirm that running the extension end-to-end does not affect the SSH, HTTPS, or SNMP validation results for an existing reference report. Re-submit a known-good existing domain report and verify its validation output is identical to the pre-extension baseline.

Chapter 11

Common Mistakes and Troubleshooting

11.1 Overview

This chapter documents the failure modes most commonly encountered when extending or operating the system. Most of these failures produce no runtime error and are therefore easy to miss.

11.2 LLM Calls Failing Silently

Symptom: Reports are generated but all narrative sections contain fallback placeholder text.

Cause: The machine is not connected to the internal NCCS office network. The Ollama server at `http://10.61.6.71:3000` is unreachable.

Fix: Connect to the internal office Wi-Fi and verify reachability with:

```
curl -I http://10.61.6.71:3000
```

Listing 11.1. Verifying Ollama server reachability

Any HTTP response confirms the network path is open. A timeout or connection refused means the machine is on the wrong network.

11.3 Qdrant Lock File Error on Startup

Symptom: The API service fails to start with an error referencing a lock file or `qdrant_storage/` being in use.

Cause: The previous process was terminated uncleanly, leaving the Qdrant file lock in place. This also occurs if the ingestion manager was running when the API service was started.

Fix: Stop all Python processes, delete the lock file, and restart:

```
rm qdrant_storage/.lock
uvicorn main:app --host 0.0.0.0 --port 8000
```

Listing 11.2. Removing the Qdrant lock file

11.4 Hallucinated Algorithm Citations in Narratives

Symptom: Narratives reference algorithm names that were not found on the device under test. The `validate_output()` function logs a RAG hallucination warning and the output is marked `confidence: low` after retries.

Cause: Clause references were not embedded in the chunk `text` field during ingestion.

Fix: Re-inspect the chunker for the new document type. Confirm that every chunk's `text` field contains the algorithm name, compliance status, and clause reference inline. Re-ingest the document after fixing the chunker.

11.5 Schema Mismatch Breaking `validate_output()`

Symptom: Every call to a new endpoint returns `confidence: low`. The console log shows validation issues about missing keys.

Cause: The new prompt builder returns a JSON schema that differs from what `validate_output()` in `llm.py` expects.

Fix: Confirm the prompt's requested JSON schema matches exactly. If the new prompt requires a genuinely different schema, update `validate_output()` in `llm.py` to accept the new structure before deploying.

11.6 Evidence Parser Returning Empty Blocks

Symptom: The Per-TC Evidence Analysis section of the validation report shows `(no evidence extracted)` for one or more test cases. The corresponding AI Verdict is `INCONCLUSIVE`.

Cause A: The heading format in the generated report does not match the boundary pattern in `parse_evidence.py` (or the new domain's evidence parser). The parser could not locate the start of the evidence block.

Fix A: Open a sample report, find the exact heading text before the evidence block, and update the boundary regex to match it. Run the parser directly on the sample report and print the output to confirm the block is now extracted.

Cause B: The evidence block exists in the report but was filtered out by the anti-hallucination filter, which incorrectly identified the terminal output as AI-generated content.

Fix B: Inspect the filter regex in the evidence parser. Tighten the filter so it targets only the AI-generated summary table headings rather than broad patterns that also match terminal output headings.

AUTOMATED REPORT VALIDATION

INTERNAL NCCS QUALITY CHECK — NOT FOR VENDOR DISTRIBUTION

1. Validation Summary

Validation Timestamp	2026-04-13 05:54:34
DUT Name	OpenWrt
Report Validated	/tmp/tmpo3qqe1hd.docx
SSH (TC1/TC2/TC3/TC4)	**FAIL / PASS / FAIL / PASS**
HTTPS (TC5/TC6/TC7/TC8)	**PASS / PASS / FAIL / PASS**
SNMP (TC9/TC10)	**FAIL / FAIL**
Final Test Result	**FAIL**
Validation Issues Found	**3**

2. Overall Validation Verdict

REPORT REQUIRES CORRECTION BEFORE SUBMISSION

3. Deterministic Check Results

	Category	Severity	Finding
✗	**SNMP COMPLIANCE**	**HIGH**	SNMPv1 is enabled. It uses plaintext community strings with no authentication or encryption and must be disabled per ITSAR001962411.
✗	**SNMP COMPLIANCE**	**HIGH**	SNMPv2c is enabled. It relies on community strings with no encryption and must be disabled per ITSAR001962411.
✗	**SNMP COMPLIANCE**	**HIGH**	SNMPv3 authNoPriv mode was accepted by the DUT. This provides no encryption and must be rejected per ITSAR001962411.
✓	**DUT INTEGRITY**		OS digest hash present (65 chars) — DUT software identity can be confirmed.
✓	**DUT INTEGRITY**		Configuration digest hash present (65 chars) — DUT configuration identity can be confirmed.

✓	**NMAP ANALYSIS**		Port 22 confirmed open — consistent with SSH TC applicability.
✓	**NMAP ANALYSIS**		Port 443 confirmed open — consistent with HTTPS TC applicability.
✓	**NMAP ANALYSIS**		Port 161 confirmed open — consistent with SNMP TC applicability.
✓	**COMPLETENESS**		All mandatory section headings present.
✓	**COMPLETENESS**		Conclusion section present with generated narrative (37 words).
✓	**COMPLETENESS**		26 screenshot image(s) detected as embedded in report.
✓	**COMPLETENESS**		Results table verified for 10 test case(s).
✓	**VERDICT CONSISTENCY**		Final result 'FAIL' is mathematically consistent with individual TC results.
✓	**VERDICT CONSISTENCY**		TC3 result is consistent with SSH weak cipher negotiation findings.
✓	**VERDICT CONSISTENCY**		TC7 result is consistent with TLS weak cipher negotiation findings.
✓	**PROTOCOL VERSION**		SSH protocol version 2.0 confirmed — compliant (SSHv2 required by ITSAR001962411).
✓	**TLS COMPLIANCE**		TLS protocol version TLSv1.3 — meets minimum TLS 1.2 requirement.
✓	**TLS COMPLIANCE**		No weak TLSv1.2 cipher suites detected (3 strong).
✓	**TLS COMPLIANCE**		No weak TLSv1.3 cipher suites detected (3 strong).
✓	**TLS COMPLIANCE**		TC5 result 'PASS' is consistent with TLS cipher suite tables.
✓	**TLS COMPLIANCE**		ECDHE-ECDSA-AES128-GCM-SHA256 — APPROVED per ITSAR001962411 Table 1.
✓	**TLS COMPLIANCE**		ECDHE-ECDSA-AES256-GCM-SHA384 — APPROVED per ITSAR001962411 Table 1.
✓	**TLS COMPLIANCE**		ECDHE-ECDSA-CHACHA20-POLY1305 — APPROVED per ITSAR001962411 Table 1.
✓	**TLS COMPLIANCE**		TLS_AES_128_GCM_SHA256 — APPROVED per ITSAR001962411 Table 1.
✓	**TLS COMPLIANCE**		TLS_AES_256_GCM_SHA384 — APPROVED per ITSAR001962411 Table 1.
✓	**TLS COMPLIANCE**		TLS_CHACHA20_POLY1305_SHA256 — APPROVED per ITSAR001962411 Table 1.

✓	**SNMP COMPLIANCE**		SNMPv3 authPriv mode confirmed functional.
✓	**SNMP COMPLIANCE**		SNMPv3 noAuthNoPriv mode correctly rejected.
✓	**SNMP COMPLIANCE**		TC9 result 'FAIL' consistent with SNMP version findings.
✓	**SNMP COMPLIANCE**		TC10 result 'FAIL' consistent with SNMPv3 mode findings.

4. Per-TC Evidence Analysis

TC	Evidence Snippet	AI Verdict	Automation Verdict	Match
TC1	diffie-hellman-group14-sha256	FAIL	FAIL	MATCH
TC2	ssh root@192.168.56.104	PASS	PASS	MATCH
TC3	debug1: kex: algorithm: diffie-hellman-group14-sha256	FAIL	FAIL	MATCH
TC4	command-line line 0: bad ssh2 cipher spec 'none'.	PASS	PASS	MATCH
TC5	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (ecdh_x25519) - A	PASS	PASS	MATCH
TC6	openssl s_client -connect 192.168.56.104:443	PASS	PASS	MATCH
TC7	Cipher is tls_aes_128	FAIL	FAIL	MATCH
TC8	cipher is (none)	PASS	PASS	MATCH
TC9	iso.3.6.1.2.1.1.1.0 = STRING: "Linux OpenWrt 6.12.71 #0 SMP Tue Mar 3 00:14:15 2026 x86_64"	FAIL	FAIL	MATCH
TC10	No response	FAIL	FAIL	MATCH

5. Discrepancy Alerts

No discrepancy alerts. AI and automation verdicts are aligned for all applicable test cases.

6. Tester Action Items

TC	Priority	Action	Detail / Command
TC1	**HIGH**	Update SSH configuration to disallow 'diffie-hellman-group14-sha256' and only allow secure algorithms.	ssh -o KexAlgorithms=+sntrup761x25519-sha512,+curve25519-sha256
TC3	**HIGH**	Update SSH configuration to disallow the use of diffie-	ssh -o KexAlgorithms=+ecdh+sha2 root@192.168.56.104

		hellman-group14-sha256	
TC7	**HIGH**	Disable TLS_AES_128_GCM_SHA256 and only allow approved AEAD suites	openssl -cipher ECDHE-RSA-AES128-SHA
TC9	**HIGH**	Configure the device to only use SNMPv3 with authPriv	snmpwalk -v3 -c <new_password> -A <new_password> -a SHA -x AES-128-CBC 192.168.56.104
TC10	**HIGH**	Configure SNMPv3 with authPriv and restrict access to only allowed hosts.	snmpwalk -v3 -u snmpuser -l authPriv -a SHA -A AuthPass123 -x AES -X PrivPass123 192.168.56.104

7. Auditor Acknowledgement

The auditor acknowledges receipt of this automated validation result. Discrepancy alerts and tester action items must be reviewed prior to report submission.

Auditor Name	
Date	