ISTITUTO NAZIONALE DI FISICA NUCLEARE

Laboratori Nazionali di Frascati

INFN-2026-07-LNF
3 Agosto 2026

RENATO: FROM SCANNED LEGACY SCHEMATICS TO AN INSTALLABLE FAULT-SEARCH KNOWLEDGE BASE

An AI-Curated Documentation Pipeline for a Particle-Accelerator Control Room

Claudio Di Giulio[1)] Giulia Latini[1)]

[1)] *INFN, Laboratori Nazionali di Frascati, Via E. Fermi 54, I-00044 Frascati, Italy*

Abstract

Many accelerator facilities operate hardware whose only surviving documentation is a collection of scanned, image-only electrical schematics with no extractable text and no explicit cross-referencing. This paper describes the design and implementation of RENATO (*Repository of Engineering Notes, Analysis, Troubleshooting, and Orchestration*), an offline, installable knowledge base and search tool built from the 189 scanned schematics of the TITAN system on the DAFNE LINAC at INFN-LNF. A team of AI agents was used to read each schematic visually, produce structured technical documentation, and synthesize a cross-schematic fault-diagnosis map; the resulting corpus was then indexed with a lightweight, dependency-minimal full-text search stack (SQLite FTS5, a Python standard-library HTTP server, Jinja2 templates) and packaged for offline installation on a Windows or Linux control-room PC, with a defined integration path into a Phoebus operator panel. A central design decision is discussed in detail: AI was deliberately confined to an offline, batch knowledge-curation role rather than deployed as a live conversational interface, in order to preserve determinism, auditability, and independence from any specific AI vendor at the point of use – properties considered more important than conversational flexibility in a safety-relevant control-room maintenance context. The paper presents the multi-agent methodology, the resulting system architecture, the packaging and deployment strategy, and the rationale behind the human/AI division of labor.

PACS: 07.05.Kf; 29.20.Ej; 89.20.Ff
claudio.digiulio@lnf.infn.it

Published by
Laboratori Nazionali di Frascati

1 Introduction

Particle accelerator facilities frequently rely on electronic and electromechanical subsystems designed decades ago, whose documentation survives only as scanned paper drawings. At the DAFNE LINAC (INFN-LNF), the **TITAN** system – covering the injector, RF modulators, positron converter, vacuum, water cooling, beam diagnostics, magnets, and beamline interconnects – is documented by an archive of **189 electrical schematics in PDF form**, organized into 16 subsystem folders. These files are image scans of the original drawings: they contain **no extractable text**, and the relationships between drawings (e.g., "next assembly", shared connectors, upstream/downstream signal chains) are implicit, expressed only through cartouche references that a human reader must resolve manually.

This form of documentation creates two practical problems for control-room fault diagnosis. First, locating the relevant drawing for a given symptom requires either institutional memory or a manual search across dozens of PDF files. Second, the reasoning that links a fault symptom to a specific component – historically held by experienced maintenance staff – is not written down anywhere in a form that can be searched or transferred.

This paper reports on a project that addressed both problems by (i) using a team of AI agents to visually analyze every schematic and produce structured, searchable documentation, (ii) synthesizing that documentation into an explicit fault-diagnosis knowledge map, and (iii) packaging the result as a small, self-contained, offline search application rather than as a live AI assistant. The distinction in point (iii) is deliberate and is the central design argument of this paper: the AI is used **once, offline, to build a durable artifact**, not continuously, online, to answer questions at run time. Section 6 develops this rationale in detail.

The contributions of this work are:

1. A repeatable multi-agent methodology for converting a large archive of scanned, text-less engineering drawings into structured, uniform technical documentation, including a component-level enrichment pass;

2. A cross-document synthesis step (an "orchestrator" agent) that builds an explicit relationship map and a symptom-based diagnostic guide across the whole corpus;

3. A minimal-dependency, offline-first search application and packaging pipeline (self-contained archive, PowerShell installer, Windows Installer project) suitable for a control-room PC with no ongoing network or AI-service dependency;

4. An explicit discussion of why AI was applied at build time rather than at query time, and what this choice trades off.

Figure 1 summarizes the complete pipeline described in this paper, from the scanned archive to the deployed, installable tool.

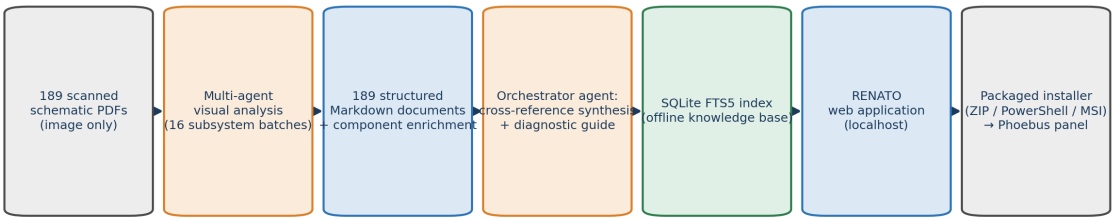

Figure 1: End-to-end pipeline: from scanned legacy schematics to a control-room search tool.

2 System Context and Source Material

The source archive consists of 189 PDF files distributed across 16 subsystem folders (interface/buffer PCBs, comparator/fault-logic PCBs, trigger and modulator logic, RF line modulator, positron concentrator, injector/gun/filament, vacuum system, water system, beam diagnostics, fiber-optic communications, motor/steering/phase-shifter drives, magnets, control racks and power distribution, RF system assembly, beamline interconnects, and mechanical assemblies). Each file is a scanned engineering drawing, typically including a cartouche with drawing number, title, revision, designer, and a "next assembly" reference to a parent drawing. As noted above, no text layer is present in any of these files; any automated processing therefore requires visual interpretation of the drawing image itself, not text extraction.

3 Methodology: Multi-Agent AI-Assisted Schematic Analysis

3.1 Task decomposition

The 189-drawing archive was partitioned into 16 batches, one per subsystem folder. Each batch was assigned to a separate AI agent instance, dispatched with explicit operating instructions: read every drawing in the batch, identify its function within the TITAN system, and produce one structured document per drawing. Batches were dispatched in parallel groups (typically four at a time) to bound the overall processing time of the full archive.

3.2 Visual, multimodal reading of scanned drawings

Because the source PDFs contain no extractable text, each agent read every schematic **visually**, in a multimodal fashion – interpreting the rendered drawing image directly, in the same way a technician would read it on paper, rather than relying on OCR or any text-extraction step. This included recognizing component designators (U, R, C, Q, etc.), reading component values, tracing interconnections between functional blocks, and transcribing cartouche metadata (drawing number, title, next-assembly reference).

3.3 Common documentation schema

To keep the resulting corpus uniform and machine-indexable, every agent was required to produce a Markdown document with the same fixed section structure, shown in Table 1.

Table 1: Common section structure of each generated document.

Section	Content
Circuit function	What the schematic does within the system
Main components	List of components / integrated circuits present
Operating description	Step-by-step explanation of circuit behavior
I/O signals and interconnections	Interfaces to other schematics / chassis
Cross-references	Other drawing numbers cited on the schematic
Plausible fault modes	Table: fault $\rightarrow$ observable symptom $\rightarrow$ component involved
Notes	Ambiguities, illegible markings, observations

Each Markdown document was subsequently rendered to PDF (Pandoc with the XeLaTeX engine, using a Unicode-capable font to preserve symbols such as Ω and signal-flow arrows that appear in the source drawings).

3.4 Quality control per batch

At the end of each batch, before proceeding to the next, the number of documents actually produced was checked against the number of files assigned, to catch skipped or incomplete drawings. Because the full archive was processed across multiple working sessions, this per-batch checkpoint also allowed work to resume exactly where it had left off after an interruption, without duplicating already-completed batches.

3.5 Component-level enrichment pass

In a second pass, a further set of agents re-read the 189 already-produced documents (this time working from text, not from the drawing image) to enrich the "Main components" section with a **per-designator explanation of circuit role** – for example, "U1: LM555C, configured as an astable timer" – rather than a bare parts list. This additional layer of detail specifically targets usefulness for hands-on maintenance, where the reader needs to know not just *what* a component is but *what it does* in that particular circuit.

Figure 2 summarizes this methodology end to end, including the downstream synthesis step described in Section 4.

4 Knowledge Synthesis: The Orchestrator Agent

Once all 189 schematic-level documents existed, a further AI agent – referred to as the *orchestrator* – was tasked with reading the entire corpus (the generated documentation,

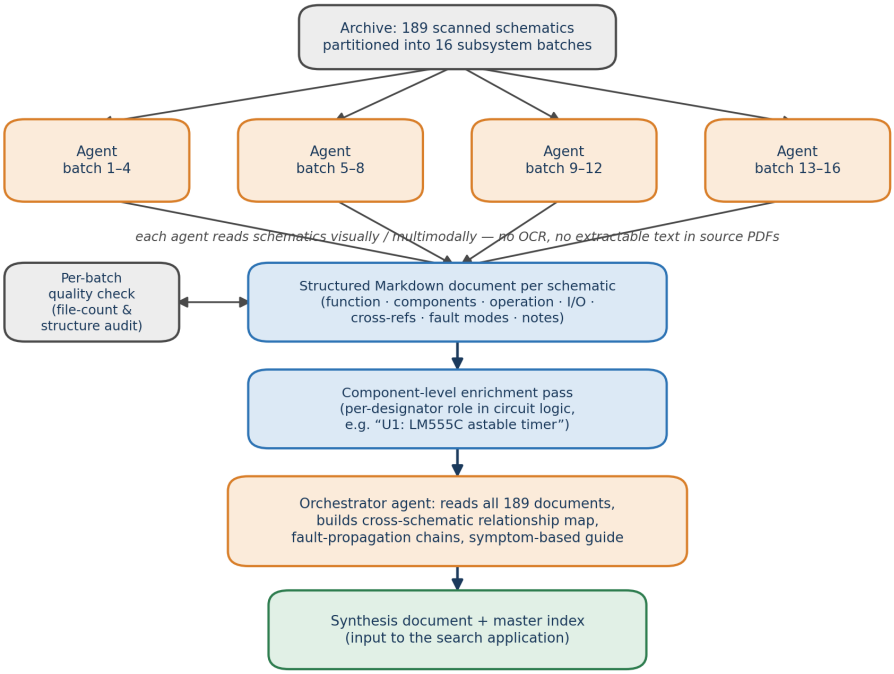

Figure 2: Multi-agent AI methodology for schematic analysis and knowledge synthesis.

not the original drawings) to build a system-level view that no single schematic could provide on its own. This step produced:

- a **relationship map** between schematics (which board powers, controls, or monitors which other board);
- an explicit **comparator / fault-detection logic chain**, describing how a physical fault propagates through the system to a panel-level indication;
- a **symptom-based diagnostic guide**, listing roughly ten representative fault symptoms together with the schematics relevant to each;
- a **master index** of the full document set.

This synthesis is the conceptual core that lets a maintainer start from an observed symptom and be pointed to the relevant drawings – the same logic that was subsequently exposed through the search application's full-text query interface. As a validation step, the same orchestrator was also used interactively to answer natural-language fault-diagnosis questions (e.g., "the SCR phase-angle control board's ramp is not working – what could be the cause?"), by consulting the relevant generated documents to identify plausible causes.

5 System Architecture and Implementation

5.1 Indexing

A script (`build_index.py`) parses every Markdown document into its fixed sections, normalizes minor heading variants (e.g., "Main components (comparators/thresholds in detail)"), normalizes each drawing number to a canonical form, and populates a SQLite database consisting of a `documents` table and a linked `documents_fts` virtual table using **SQLite FTS5** with bm25 ranking.

5.2 Web application

A script (`app.py`) implements the application server using only Python's standard-library `http.server`, exposing the following routes: `/` (landing page with search and subsystem list), `/search` (full-text query results), `/browse` (browse by subsystem), `/doc/<id>` (schematic detail page, with every section rendered from Markdown to HTML), `/pdf/...` (the generated analysis PDF), and `/schema/...` (the original scanned schematic PDF, kept distinct from the analysis PDF).

5.3 Automatic cross-reference hyperlinking

On every detail page, any text matching a drawing-number pattern (`\b([A-E]-\d{3,5})\b`) is automatically converted into a hyperlink to the corresponding document, and this linkification is applied to **every section of the page**, not only to the dedicated "Cross-references" section – so that the network of related schematics can be navigated directly from wherever a drawing number is mentioned in the running text. Self-references are suppressed to avoid a document linking to itself.

5.4 Dependency footprint

Table 2 summarizes the requirement-to-choice mapping that drove the architecture, all aimed at a single goal: an application that a site technician can copy onto a control-room PC and run with a minimum of installation friction.

Figure 3 shows the resulting architecture end to end, from the Markdown corpus to the client (browser or Phoebus widget).

6 Design Rationale: AI as a Curation Tool, Not a Runtime Chatbot

An alternative design was explicitly considered and rejected: exposing an AI conversational interface directly to control-room operators at query time, letting a chatbot answer fault-diagnosis questions live by reasoning over the schematic archive on demand. This section explains why that alternative was not adopted, and why AI was instead confined to a one-time, offline, batch documentation-generation role, with the deployed runtime tool being a conventional, deterministic full-text search application.

Table 2: Requirements and corresponding architectural choices.

Requirement	**Choice**	**Rationale**
Full-text search	SQLite FTS5 (bm25 ranking)	Included in the Python standard library; no database server to install
Web server	Python `http.server`	Avoids heavyweight frameworks (Flask/FastAPI); minimizes install footprint on a control-room PC
Page templating	Jinja2	Only external dependency needed alongside `markdown`
Content rendering	Python `markdown` library	Source documents are already Markdown; no re-conversion needed

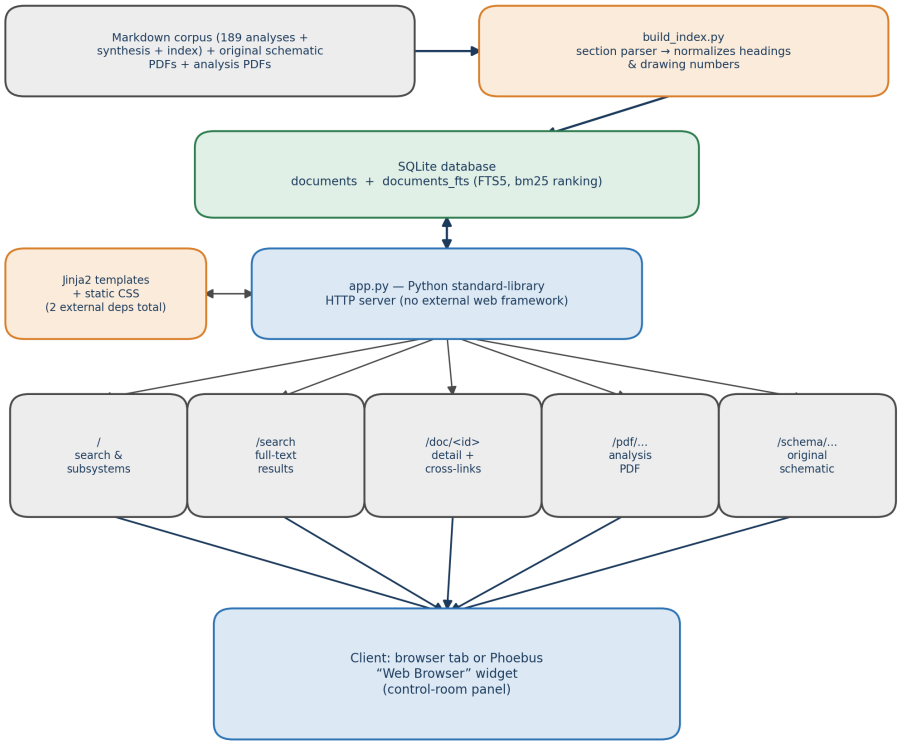

Figure 3: Application architecture (minimal-dependency, offline-first stack).

The reasoning rests on five properties that a control-room maintenance tool is expected to have, summarized in Figure 4.

- **Offline / air-gapped operation.** Control-room PCs are not necessarily expected to have reliable external network access, and a fault-diagnosis tool should not depend on it. A static, indexed search application runs entirely locally; a conversational AI interface requires ongoing access to a model, whether via a remote API or local inference hardware.
- **Determinism and reproducibility.** For safety-relevant maintenance decisions, the same query should return the same, previously reviewed answer every time. A full-text search over a fixed, versioned document corpus satisfies this; a generative response does not offer the same guarantee, since outputs can vary between invocations even for the same input.
- **Traceability of answers.** Every result returned by the search tool links directly to a specific, human-reviewable document with clear provenance (which agent batch produced it, which drawing it corresponds to, when it was generated). A free-form generative answer is comparatively harder to audit back to a specific source.
- **Latency and resource footprint.** Full-text search over an indexed corpus returns results instantly on modest control-room hardware; a conversational AI interface implies inference latency and, depending on deployment, a non-trivial compute or GPU requirement.
- **Long-term maintainability.** Accelerator infrastructure is often operated for decades, well beyond the expected lifetime of any single AI product or vendor relationship. A durable, self-contained documentation corpus and search index is not tied to the continued availability of a specific AI service, whereas a runtime chatbot architecture inherently is.

Under this design, AI is applied exactly once, offline, as a **knowledge-curation step**: it converts an unsearchable archive of scanned images into a structured, reviewable, versionable text corpus. The tool that is actually deployed and used daily in the control room has no AI dependency at query time at all – it is a conventional, deterministic search application. This division of labor is intended to capture the productivity benefit of AI-assisted documentation (a task that would otherwise have required a substantial manual effort across 189 drawings) while keeping the operational tool itself simple, auditable, and independent of any particular AI provider.

7 User Interface and Generated Output

This section illustrates the deployed application from the operator's point of view, using representative screens produced by the running tool. Figure 5 shows the landing

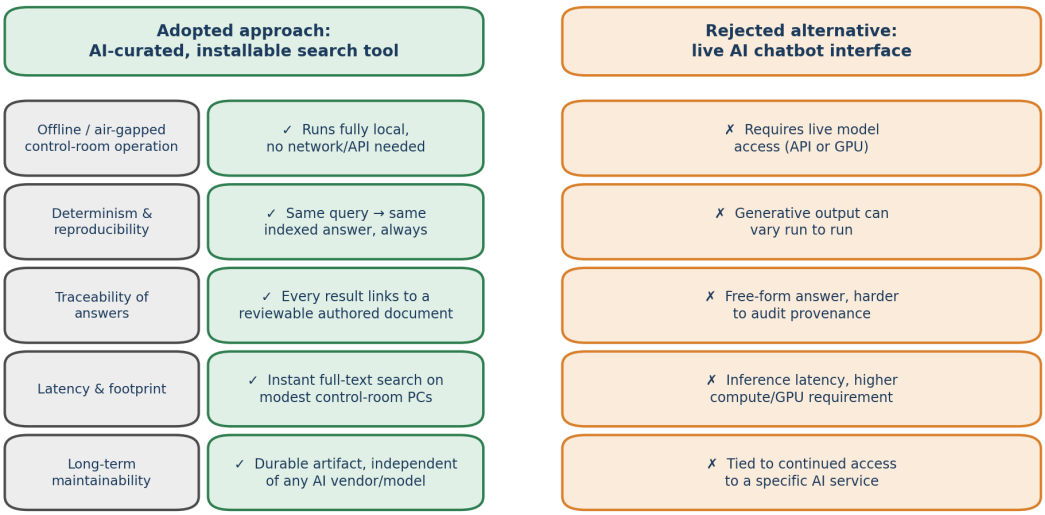

Figure 4: Design rationale: AI used to curate a static tool, not to power a runtime chatbot.

page, reached by opening `http://localhost:8420` in a browser or in a Phoebus "Web Browser" widget. It exposes a single search box together with a browsable grid of the 16 subsystems, each labeled with the number of indexed schematics it contains – giving an operator two equivalent entry points, full-text search or subsystem browsing, depending on whether the starting point is a symptom or a known area of the plant.

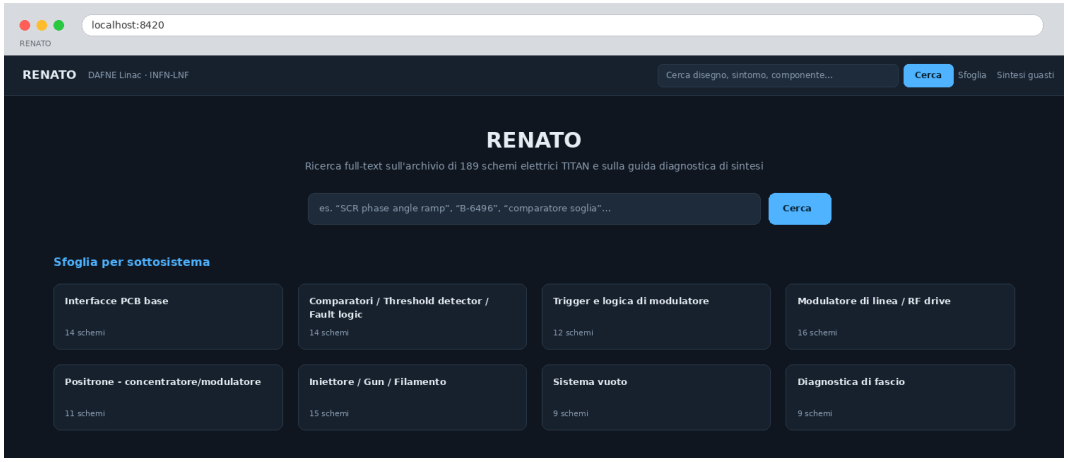

Figure 5: Landing page: single search box plus a browsable grid of the 16 indexed subsystems.

Figure 6 shows the result of an actual full-text query, `comparatore soglia` ("comparator threshold"), returning 22 ranked matches. Each result card displays the drawing number, the document title, the owning subsystem, and a context snippet generated by SQLite FTS5 around the matched terms – in this example, several results surface directly from the "Plausible fault modes" table of each document (e.g., hysteresis loss causing

threshold “chattering”, or a channel comparator failure), which is precisely the information an operator needs when starting from an observed symptom rather than a known drawing number.

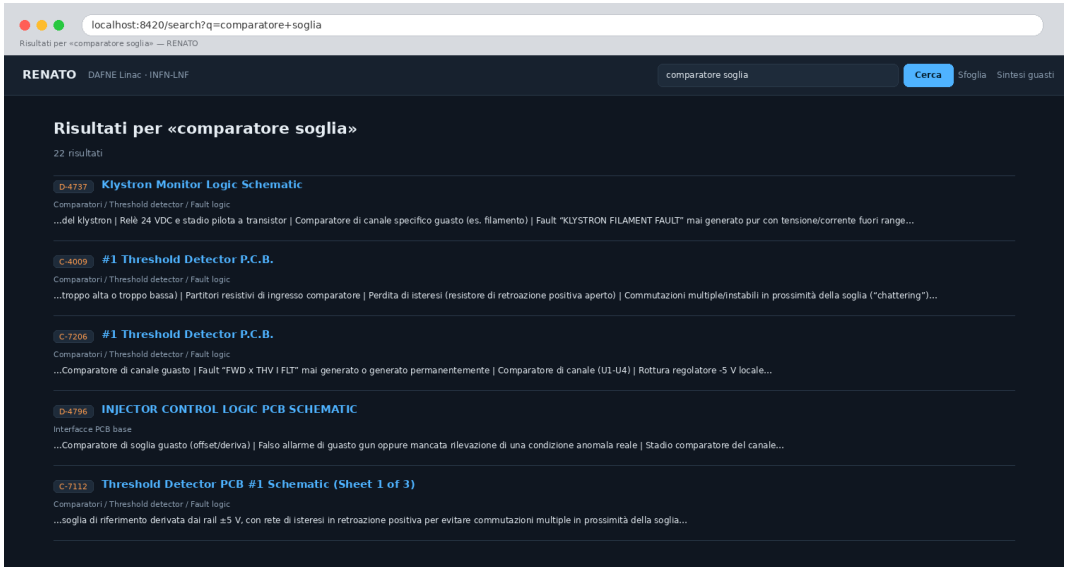

Figure 6: Full-text search results for the query “comparatore soglia” (comparator threshold), 22 matches ranked by SQLite FTS5 bm25 relevance.

Figure 7 shows a document detail page (drawing C-4009, a two-channel threshold-detector board), illustrating the complete set of information available for every one of the 189 schematics: a plain-language circuit function summary; a per-designator component table generated in the enrichment pass (Section 3.5), stating not just what each part is but what role it plays in the circuit – for example, resistor R25 is identified as the positive-feedback element that introduces hysteresis on the comparator, not merely listed as “100 kΩ”; the plausible-fault-mode table linking a failure to its observable symptom and the specific components involved; and, in the cross-reference section, a live hyperlink to the next-assembly drawing (D-6182) – generated automatically wherever a drawing number appears anywhere on the page, not only in the dedicated cross-reference section, as described in Section 5.3. The two buttons at the top give direct access to both PDF artifacts kept for each schematic: the generated analysis document and the original scanned drawing.

Taken together, these three views expose the same information that a maintainer previously had to reconstruct manually from paper drawings and institutional memory: what a circuit does, what each component in it is for, how a fault would present itself, and which other drawings are involved – all reachable in a few seconds from a single search box.

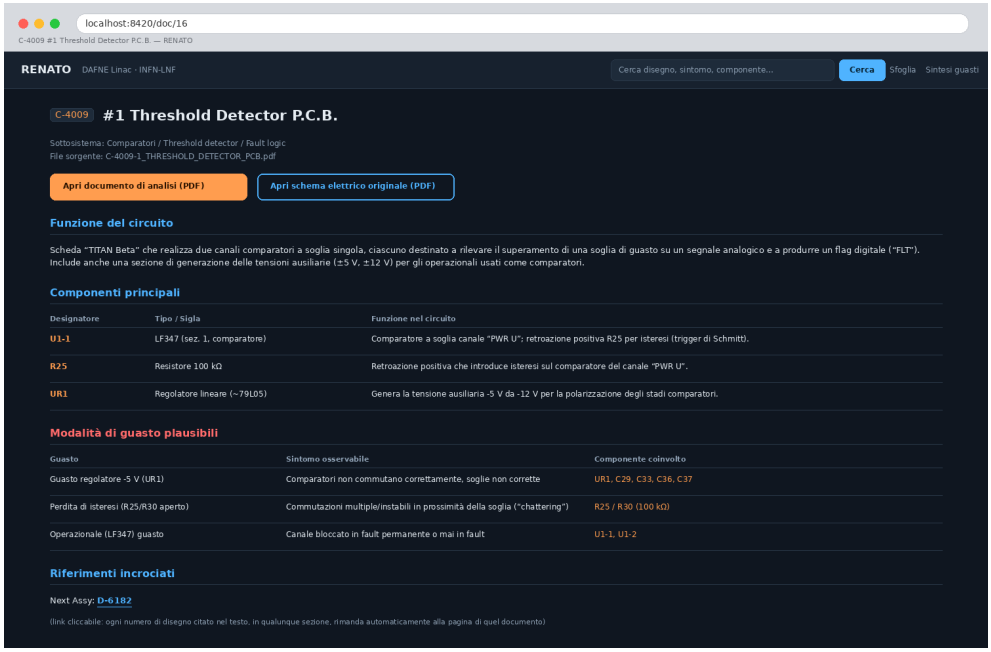

Figure 7: Document detail page for drawing C-4009: circuit function, per-designator component table, fault-mode table, and a live cross-reference link, plus direct access to both the analysis PDF and the original scanned schematic.

8 Deployment and Packaging

To make the application easy to distribute to a control-room PC, the following artifacts were produced:

- a **self-contained ZIP archive** containing code, templates, data (Markdown documents, analysis PDFs, original schematic PDFs), and a pre-built search index;
- quick-start launch scripts for **Windows** (`run_windows.bat`) and **Linux** (`run_linux.sh`), which create a virtual environment, install the two required dependencies, and start the server;
- a working **Windows installer** based on PowerShell (`install.ps1` `uninstall.ps1`, also invokable via `INSTALLA.bat`), requiring no additional tooling and creating Start Menu / Desktop shortcuts;
- a **WiX Toolset project** ready to produce an actual `.msi` package: a Python script (`generate_components.py`) automatically generates the WiX file manifest for the application (currently 583 components), to be compiled with the official WiX build tools (`candle.exe` / `light.exe`) on a Windows machine – a step outside the scope of the Linux-based development environment used to prepare the project.

9 Integration with the Control-Room Environment (Phoebus)

In its current form, the application is designed to be opened as an ordinary web page **inside a Phoebus operator panel**, requiring no additional development: a "Web Browser" widget is added to a Phoebus display and pointed at the local server address (or at the address of a centralized instance shared across control-room workstations). Several extensions have been identified as future work, not yet implemented:

- direct linking from an EPICS/Phoebus alarm to the corresponding fault documentation page;
- correlating the static documentation with real historical fault events (EPICS Archiver Appliance / Alarm Logger);
- a natural-language query module that synthesizes an answer across multiple documents, which would require a separate evaluation of infrastructure, connectivity, and – per the discussion in Section 6 – a deliberate re-assessment of the trade-offs described above before introducing AI at query time.

10 Limitations

Some drawings referenced in cross-reference sections (e.g., D-4914, a fault detector multiplexer board) were not present in the original archive and therefore remain cited but not retrievable. The generated documentation is itself the product of an automated visual reading process: wherever a designator or value could not be read with confidence, this is stated explicitly in the text rather than guessed. The search index is not automatically rebuilt when source documents are updated – it is only built if absent – so content updates require an explicit index rebuild step, documented in the project's maintenance notes.

11 Conclusion

This paper described the construction of RENATO, a knowledge base and offline search tool derived from 189 scanned, text-less electrical schematics of the DAFNE LINAC's TITAN system. A team of AI agents performed the labor-intensive task of visually reading each drawing and producing structured, uniform technical documentation, followed by an orchestrator agent that synthesized cross-schematic relationships and a symptom-based diagnostic guide. Rather than exposing this AI capability directly to operators as a live conversational interface, the project deliberately restricted AI use to an offline, one-time documentation-generation step, and built the deployed tool as a conventional, deterministic, dependency-minimal search application – prioritizing offline operation, reproducibility, traceability, and long-term maintainability over conversational flexibility. The resulting artifact is packaged for straightforward installation on a Windows or Linux

control-room PC and has a defined integration path into the facility's Phoebus-based operator interface.

12 Acknowledgements

The author acknowledges the use of an AI assistant (Open AI codex ChatGPT Edu licence) in the analysis of the schematic archive, the synthesis of the diagnostic knowledge base, and the development of the described software tool, as detailed in Sections 3–4. The author acknowledges the BTF team and LINAC Staff for the documentatin provided.

13 References

[1] SQLite Consortium, *SQLite FTS5 Extension*, SQLite Documentation, `https://www.sqlite.org/fts5.html`.

[2] Pallets Projects, *Jinja2 Template Engine Documentation*, `https://jinja.palletsprojects.com/`.

[3] Python Software Foundation, *markdown – Python Markdown Library*, `https://python-markdown.github.io/`.

[4] Phoebus / CS-Studio Project, *Phoebus Documentation*, `https://control-system-studio.readthedocs.io/`.

[5] WiX Toolset Project, *WiX Toolset v3 Documentation*, `https://wixtoolset.org/docs/v3/`.

[6] OpenAI, *OpenAI Codex [AI coding assistance]*, `https://openai.com/codex/`.