Pulse – Module Extraction AI Agent

By- CHERRY SHARMA (RA2211042010051)

Technical Architecture, Approach, Assumptions & Edge Case Handling

1. Overview

The **Pulse – Module Extraction AI Agent** is an AI-powered application designed to automatically extract **structured product intelligence** from documentation-based help websites.

Modern SaaS products expose functionality through large and often fragmented documentation portals. For Product Managers, understanding the **functional breakdown of a product (modules and submodules)** is time-consuming and error-prone when done manually.

This project addresses that problem by building a **Generative AI–driven system** that:

- **Accepts one or more documentation URLs**
- **Crawls and processes relevant documentation pages**
- **Infers modules (major product areas)**
- **Infers submodules (specific functionalities under each module)**
- **Generates accurate, PM-style descriptions**
- **Outputs results in a structured JSON format suitable for downstream consumption**

The solution is built as a **local full-stack GenAI application**, focusing on reasoning quality, structure, and maintainability rather than UI complexity.

2. Technical Architecture Description

2.1 High-Level Architecture

The system follows a **layered architecture** with clear separation of responsibilities:

User Interface (React / Streamlit-like UI)

|

| REST API Call

v

```
Backend Orchestration Layer (FastAPI)

    |

    |-- URL Validation

    |-- Documentation Crawler

    |-- Content Cleaner & Normalizer

    |-- LLM-based Module Inference Engine

    |

    v

Structured JSON Output
```

This architecture ensures:

- **Modularity**
- **Ease of debugging**
- **Independent evolution of crawling, AI logic, and UI layers**

2.2 Component-Level Breakdown

2.2.1 Input Layer (Frontend / UI)

Responsibilities

- **Accept one or more documentation URLs**
- **Trigger module extraction**
- **Display structured results**

Design Decisions

- **Simple UI with a single responsibility: enable interaction with the AI agent**
- **No authentication or persistence to keep scope aligned with assignment**
- **Output rendered in a human-readable hierarchical format**

2.2.2 Backend API Layer

Technology

- **Python + FastAPI**

Responsibilities

- **Validate incoming URL inputs**
- **Coordinate crawling, processing, and AI inference**
- **Enforce structured output schema**

Key API Contract

POST /extract

Input: { urls: [string] }

Output: { modules: [...] }

FastAPI was chosen for:

- **Strong request/response validation**
- **Clean API semantics**
- **Production-grade performance characteristics**

2.2.3 Documentation Crawler

Responsibilities

- **Recursively crawl documentation pages starting from the input URL(s)**
- **Follow only internal documentation links**
- **Handle redirects, broken links, and unsupported pages gracefully**

Content Filtering
The crawler removes:

- **Headers**
- **Footers**
- **Navigation menus**
- **Scripts and styles**

This ensures that only **meaningful documentation content** reaches the AI layer.

Controls Implemented

- **Maximum page crawl limit**
- **Timeout handling**
- **Duplicate URL avoidance**

These controls prevent runaway crawling and noise accumulation.

2.2.4 Content Processing & Normalization

Responsibilities

- **Convert raw HTML into clean, normalized text**
- **Preserve logical structure (headings, lists) where possible**
- **Truncate content intelligently to respect LLM token limits**

This layer ensures the AI receives **signal-rich input**, improving accuracy and consistency.

2.2.5 Module & Submodule Inference (GenAI Layer)

This is the **core intelligence** of the system.

Responsibilities

- **Infer top-level product modules from documentation sections**
- **Group related features as submodules**
- **Generate concise, accurate descriptions for both levels**

Key Characteristics

- **LLM is used strictly for reasoning and inference**
- **All outputs are derived only from extracted documentation content**
- **A strict JSON schema is enforced to avoid ambiguity**

Why GenAI?
Traditional rule-based or keyword-based approaches fail because:

- **Documentation varies widely in structure**
- **Similar features are described using different terminology**
- **Product hierarchies require semantic understanding**

LLMs excel at:

- **Semantic grouping**
- **Contextual inference**
- **Hierarchical reasoning**

3. Detailed Approach

3.1 End-to-End Flow

1. **Input Acquisition**
   - **User provides one or more documentation URLs**
2. **Validation**
   - **URLs are validated for format and accessibility**
3. **Recursive Crawling**
   - **Relevant internal documentation pages are collected**
4. **Content Cleaning**
   - **Non-informational HTML elements are removed**
5. **Normalization**
   - **Text is flattened and prepared for AI consumption**
6. **LLM-Based Inference**
   - **Modules, submodules, and descriptions are inferred**
7. **Structured Output**
   - **Results returned as structured JSON**

3.2 Mapping to Assignment Requirements

Assignment Requirement	How It Is Addressed
Accept one or more URLs	Supported via API/UI
Recursive crawling	Implemented with link traversal
Content filtering	Headers, footers, nav removed
Hierarchy inference	LLM-based semantic reasoning
Structured JSON output	Strict schema enforced
PM-style descriptions	Prompt-guided generation

4. Assumptions

The implementation is based on the following assumptions:

1. **Documentation URLs are publicly accessible**
2. **Documentation content accurately reflects product features**
3. **HTML structure is reasonably well-formed**
4. **LLM has sufficient context to infer hierarchy**
5. **The goal is product understanding, not legal or contractual accuracy**

These assumptions are realistic for internal product intelligence tools used by PMs.

5. Edge Case Handling

5.1 Broken Links & Redirects

- **Skipped gracefully**
- **Crawling continues for remaining pages**
- **Partial but valid results returned**

5.2 Deeply Nested Documentation

- **Crawl limits prevent infinite traversal**
- **AI inference relies on semantic grouping rather than strict depth**

5.3 Sparse or Poorly Structured Content

- LLM infers structure from semantics instead of headings alone
- Still produces usable modules where possible

5.4 Unsupported or Non-Documentation URLs

- Input validation prevents crashes
- Empty or minimal output returned safely

5.5 LLM Output Formatting Errors

- Strict JSON parsing enforced
- Failures handled gracefully without server crash

6. Best Practices Followed

Engineering Best Practices

- Modular architecture
- Single responsibility per file
- Clean function naming
- Schema validation with Pydantic

GenAI Best Practices

- Low-temperature inference
- Explicit structured prompts
- Token-aware input truncation
- No hallucinated content injection

Reliability & Safety

- Environment variables for secrets
- Graceful failure handling
- Defensive parsing and validation

7. Limitations

- No support for authenticated documentation
- No caching mechanism in current version
- No confidence scoring per module
- No persistence or database layer

These are **intentional MVP trade-offs**, not architectural constraints.

8. Testing & Validation

The system was tested on multiple real-world documentation sources, including:

- https://support.neo.space/hc/en-us
- Instagram
- X. It's what's happening
- https://wordpress.org/documentation/
- https://help.zluri.com/
- https://www.chargebee.com/docs/2.0/

Results demonstrated:

- **Consistent module grouping**
- **Meaningful submodule extraction**
- **High-quality PM-style descriptions**

9. Conclusion

The Module Extraction AI Agent demonstrates:

- **Strong alignment with Product Management workflows**
- **Practical, high-signal use of Generative AI**
- **Clean, extensible full-stack architecture**
- **Ability to deliver a meaningful MVP within tight timelines**

This project closely aligns with Pulse's mission of **revolutionizing product management using AI**, and can serve as a foundation for more advanced product intelligence capabilities.