POLITECNICO DI TORINO

Master degree course in Digital Skills for Sustainable Societal Transitions

Master Degree Thesis

Engineering a Modern Research Website Backend with an AI-Powered Content Generation Pipeline

Relatori
Prof. Gianvito Urgese
Dr. Giuseppe Fanuli

Candidato
Wu Yuxuan

February 2026

Abstract

Web technologies have become essential for the dissemination of scientific research, with laboratories increasingly relying on web platforms to present their academic outcomes. Specifically, this thesis focuses on the backend development of the web portal for the research laboratory of inNuCE. The primary mission of inNuCE website is to share knowledge effectively; however, the technical depth of academic papers often creates a barrier for non-experts.

InNuCE group aims to involve a broader audience with different backgrounds in their researches, ensuring that their findings could reach beyond just a small circle of specialists. To achieve this, simply providing static PDF downloads is insufficient. The need is to transform complex academic content into more accessible formats. By integrating features such as AI-generated podcasts, video summaries, interactive quizzes and so on, it can help users from diverse fields better understand the core value and context of our research.

However, traditional web applications primarily focus on static information display, often referred to as Create, Read, Update and Delete(CRUD) model. This approach limits the ability of users to gain an in-depth understanding of academic papers, as features like generated summaries or audio interpretations are rarely supported. While powerful AI tools such as Google NotebookLM are available, integrating them into web applications currently requires manual intervention. Administrators must manually upload files and maintain links, which results in significant delays in data updates and high management costs.

First, the study established the necessary web infrastructure and content management system. The target website includes core sections such as Publication (a repository of academic papers), Use Case (showcasing practical project examples), and About Us page (introducing the research team). In the back-end, a persistent database was set up alongside an administrator-facing Admin Panel. This structure enables standard CRUD operations, allowing administrators to manage content efficiently without the need to modify the underlying source code.

Second, to resolve the inefficiency of manually uploading files to NotebookLM, this research developed an "Automated Intelligent Generation Pipeline". Built upon the LangChain framework and Prompt Engineering, this pipeline is designed to another alternative option of the manual workflow. The system automatically parses

uploaded documents and directs Large Language Models (LLMs) to generate nine types of multimodal resources, including podcasts, quizzes, and mind maps. These outputs are organized as documents on the server, and resource links are automatically maintained in the database. This architectural upgrade advances the system from manual maintenance to fully automated production, improving the speed and efficiency of content management.

In summary, by combining robust infrastructure construction with AI automation, this research creates a closed-loop platform for scientific dissemination. The system effectively solves the issues of human resource limitations and data latency found in traditional models. It achieves a transition from static display to intelligent interaction, offering a practical reference for the application of generative AI technology in web application.

Contents

List of Figures 8

List of Tables 10

1 Introduction 11

2 Backgroud 15
2.1 Evolution of Web Architecture: From CRUD to AI-Native 15
2.1.1 From Monolithic to Microservices 15
2.1.2 The Shift to Microservices 15
2.1.3 The CRUD Paradigm 16
2.1.4 Limitations of CRUD in the Context of Deep Understanding 16
2.1.5 The Rise of AI-Native Applications 16
2.1.6 From Deterministic to Probabilistic 17
2.2 Large Language Models (LLMs) and Generative AI 17
2.2.1 The Transformer Architecture 17
2.2.2 Foundation Models and In-Context Learning 18
2.2.3 Multimodality and Structured Generation 18
2.3 Orchestration Frameworks for LLM Applications 18
2.3.1 The LangChain Ecosystem 19
2.4 Asynchronous Architecture and Event-Driven Design 20
2.4.1 The Latency Challenge 20
2.4.2 The Producer-Consumer Pattern 20
2.4.3 Time-Based Scheduling and Container Orchestration 21
2.5 Prompt Engineering and Structured Data Adaptation 21
2.5.1 Prompt Engineering as Software Design 21
2.5.2 The Impedance Mismatch in Web Engineering 22
2.5.3 Schema Enforcement and Output Parsing 22
2.6 Context Management Strategies 23
2.6.1 Retrieval-Augmented Generation (RAG) 23
2.6.2 Full-Context Injection Strategy 23
2.7 Human-in-the-Loop (HITL) Systems 24

2.7.1 AI Reliability and Hallucinations . . . 24
2.7.2 The HITL Design Pattern . . . 24
2.8 Analysis of Existing AI-Assisted Research Tools . . . 25
2.8.1 The Benchmark: Google NotebookLM . . . 25
2.8.2 Architectural Limitations for Web Integration . . . 25
2.9 Research Gaps and Problem Formulation . . . 26
2.9.1 The Lack of End-to-End Automation . . . 26
2.9.2 The Structured Data Gap . . . 27
2.9.3 Summary and Proposed Solution . . . 27

3 Material and Methods 29
3.1 System Architecture Overview . . . 30
3.1.1 Architectural Design Principles . . . 30
3.1.2 Dual-Workflow Logic . . . 32
3.2 Technological Stack and Justification . . . 33
3.2.1 Backend Frameworks: Node.js vs. Python Flask . . . 33
3.2.2 Database System: PostgreSQL . . . 36
3.2.3 AI & Orchestration: LangChain & LLMs . . . 36
3.2.4 Asynchronous Task Queue: Redis & Celery . . . 37
3.3 Web Infrastructure Implementation . . . 38
3.3.1 Database Schema Design . . . 39
3.3.2 Public inNuCE Website Backend Service (Node.js) . . . 41
3.3.3 Administrator System Implementation . . . 45
3.3.4 Automated Data Synchronization . . . 48
3.4 Automated Intelligent Generation Pipeline . . . 50
3.4.1 Task Scheduling and Queue Management . . . 50
3.4.2 Document Pre-processing . . . 52
3.4.3 Prompt Engineering Strategy . . . 54
3.4.4 Multimodal Resource Generation . . . 56
3.4.5 Closed-Loop Data Integration . . . 59
3.5 Containerization and Deployment . . . 61
3.5.1 Docker Architecture . . . 61
3.5.2 Volume Management and Persistence . . . 63
3.5.3 Reverse Proxy Configuration . . . 63
3.6 Chapter Summary . . . 64

4 Results and Discussion 67
4.1 Multimodal Content Generation Outcomes . . . 67
4.1.1 Executive Summary Report . . . 67
4.1.2 Mental Map Visualization . . . 67
4.1.3 Interactive Assessment: Quiz . . . 70
4.1.4 Knowledge Reinforcement: Flashcards . . . 70

4.1.5 Data Extraction: Performance Table 72
4.1.6 Visual Insight: Infographic 72
4.2 Discussion 74
4.2.1 System Strengths 74
4.2.2 Engineering Limitations and Challenges 74

5 Conclusions 75

Bibliography 79

List of Figures

3.1 inNuCE Website Technical Architecture 31
3.2 ER Diagram of Infrastructure Schema 39
3.3 ER Diagram of Usecase Schema 40
3.4 ER Diagram of About Us and Publication Schemas 42
3.5 Sequence Diagram of User Request Processing Logic 44
3.6 Admin Dashboard for Use Case Management 46
3.7 Task Scheduling and Asynchronous Execution Workflow 51
3.8 Detailed Workflow of PDF Document Parsing 53
3.9 Systematic Structure of a Prompt Template 55
3.10 Closed-Loop Workflow: From Generation to Frontend Display . . . 60
3.11 Docker Containers Network Topology and Volume Mounts 62

4.1 Generated Executive Summary Report 68
4.2 Generated Mental Map of Research Structure 69
4.3 Interactive Quiz with Explanation 70
4.4 Flashcard for Concept Definition 71
4.5 Extracted Performance Metrics Table 72
4.6 Visual Presentation of Research Highlights 73

Listings

3.1 Dependencies configuration in package.json for inNuCE - website - backend 35
3.2 An Example Code for Celery Worker 37
3.3 Example Output: Audio Script and Quiz Question 58
3.4 Nginx Reverse Proxy Configuration 63

List of Tables

3.1 Summary of Technology Stack 34

Chapter 1

Introduction

Nowadays, web technologies are used everywhere for sharing scientific research. Laboratories and research institutes rely on web platforms to show their academic results and share knowledge. However, the number of scientific papers around the world is growing exponentially [1]. This creates a great challenge for traditional information sharing. Researchers and the public need more than just downloading papers; they need to understand these papers deeply, get quick summaries, and see multi-modal interpretations in a short time. Because of this, putting Artificial Intelligence (AI), especially Large Language Models (LLMs), into web applications has become an efficient way to disseminate academic papers.

But traditional web architectures mainly focus on showing static information (the basic CRUD model: Create, Read, Update, Delete). These cannot meet the user's demand for deep interaction with the content. Although Generative AI such as GPT-4 and Gemini show a great ability to understand text [2], there is a clear "gap" when trying to integrate them. For example, tools like Google NotebookLM are powerful, but they usually exist as standalone SaaS products without open APIs. In real web engineering projects, this forces administrators to rely on labor-intensive manual operations: manually uploading PDF files, waiting for generation, downloading results, and then manually updating the database links. This "manual bottleneck" causes serious delays in data updates and high management costs, which prevents academic results from spreading quickly.

AI Engineering has attracted a lot of attention in both academia and industry. The majority of the current research focuses on improving the model's reasoning ability or increasing the size of the context window[3]. However, there is still a lack of research on how to build these models into a high-availability, automated web backend system.

Specifically, integrating AI into the Web application faces three main challenges:

First, the conflict between automation and efficiency. As mentioned before, the "semi-automated" workflow that relies on moving data by hand cannot handle a large number of documents. We need a backend pipeline that can automatically

arrange tasks.

Second, the conflict between unstructured output and engineering strictness. Web front-end components (such as filters and dynamic charts) depend on strict structured data (JSON/XML). However, LLMs naturally output unstructured language. This "Impedance Mismatch" makes it difficult to use AI-generated content to directly apply in web interactions [4].

Thirdly, the issue of system reliability. Because AI has inherent hallucination problems, a fully automated publishing process is risky. Finding a balance between automation and content accuracy by introducing a "Human-in-the-Loop" mechanism is a problem that must be solved in system design [5].

To fill the gap between "AI Model Capabilities" and "Web Engineering Implementation," this thesis aims to design and evaluate a modern web platform that integrates AI solutions. This study is based on the actual needs of a scientific research lab, covering the whole process from constructing the bottom infrastructure of the inNuCE website to developing the top-level intelligent automation pipeline.

The core contribution of this research is the proposal of a "Dual-Track Generation Strategy." On one hand, the system holds the support for high-quality content from NotebookLM; on the other hand, it developed a custom automated pipeline as well based on the LangChain framework and Prompt Engineering.

This work is built on a solid Web infrastructure, including a user-facing front-end system (Infrastructure, Publication, Use Case) and an administrator content management panel (Admin Panel). To solve the "manual bottleneck," this study uses an asynchronous task queue (Celery) and a message broker (Redis). This makes the process fully automated, from uploading the paper to generating 9 types of multimodal outputs (such as abstracts, quizzes, mind map codes, and etc.). Similarly, the system introduces a structured data adapter to force the model to output JSON data that meets front-end standards. Additionally, a Content Management System (CMS) was designed to ensure that the inNuCE website is flexible and reliable.

The rest of this thesis is organized as follows:

Chapter 2: Background. This chapter reviews related literature and technical background. It covers the evolution of modern Web architectures, the basic principles of Large Language Models, core concepts of orchestration frameworks (LangChain), asynchronous Architecture and Event-Driven Design, the importance of Human-in-the-Loop (HITL) in AI system design and so on. It will also analyze the pros and cons of existing reading tools (like NotebookLM) to set the basis for my technical approach.

Chapter 3: Material and Methods. This chapter details the design and implementation of the system. First, it describes the Web infrastructure construction as the data foundation, including front-end interaction and back-end database design. Then, it focuses on building the "Automated Intelligent Generation Pipeline," including Prompt Engineering strategies, asynchronous task scheduling logic, and

parsing schemes for structured data needed by the Web front-end. It also explains the implementation details of the dual-track generation mechanism and the admin system.

Chapter 4: Results and Discussion. This chapter presents the actual performance of the system. Through a specific Case Study, it explains how the system automatically processes an academic paper and generates multimodal resources. It compares the "Manual Maintenance Mode" with the "Automated Pipeline Mode" in terms of efficiency, response speed, and management costs. I also discuss the engineering challenges encountered during development (such as JSON alignment and long-text handling) and how they were solved.

Chapter 5: Conclusions. This chapter summarizes the main findings of the study, explains the practical significance of the system for smart research dissemination, and points out the limitations of the current system (such as multi-language support). It also suggests future research directions for deeper integration of Web engineering and AI.

Chapter 2

Backgroud

2.1 Evolution of Web Architecture: From CRUD to AI-Native

Web application architecture has changed significantly over the last thirty years. Engineers have had to adapt to increasing complexity and the need for systems that can handle. This led to a shift from rigid, centralized systems to flexible, distributed ones [6]. This evolution is not just about how code is deployed; it represents a fundamental change in how data is processed and delivered. To understand the integration of AI nowadays, it is essential to examine the architectural patterns that both support and constrain the deployment of Large Language Models (LLMs).

2.1.1 From Monolithic to Microservices

Historically, web applications were built using the **Monolithic Architecture**. In this approach, every part of the application—the user interface, the business logic, and the database access—is packaged into a single unit. As Bass et al. (2012) point out, this structure is simple to develop and test in the early stages [7]. However, as the application grows, this design creates problems. Even a small change in one part of the code requires the entire system to be recompiled and redeployed. This creates a "deployment monolith," which makes continuous integration and delivery (CI/CD) difficult and slow [8].

2.1.2 The Shift to Microservices

To solve the problems of monolithic systems, the industry moved towards **Microservices Architecture**. This style breaks an application down into a collection of loosely coupled services. Each service handles a specific business function and

communicates via lightweight protocols like HTTP. Newman (2015) notes that microservices improve maintainability and scalability because each component can be deployed independently [9]. This separation allows teams to use different technologies for different tasks. However, it is important to note that while microservices changed the *structure* of web apps, the *logic* stayed the same: these systems still run on deterministic rules, where a specific input always produces a predictable output based on hardcoded logic.

2.1.3 The CRUD Paradigm

While the structural architecture evolved, the way data is handled has remained largely dominated by the **CRUD (Create, Read, Update, Delete)** model. This model, often implemented via RESTful APIs [10], is the standard for web data management. For scientific platforms like digital libraries, CRUD is very effective for managing static files. It works well for storing PDFs, retrieving metadata like titles and authors, and updating records.

2.1.4 Limitations of CRUD in the Context of Deep Understanding

However, the CRUD model has limitations when meeting modern user needs. It is good at *retrieving* information—finding a document based on a search term—but it cannot help with *understanding* it. Today's users need more than just file access; they need help summarizing long texts or visualizing complex concepts. As O'Reilly (2005) discussed during the rise of Web 2.0, the web became greatly for user-generated content but had not yet used machine intelligence to aid human understanding [11]. A CRUD system is passive: it serves data but does not interpret it, leaving all the analysis work to the user or reader.

2.1.5 The Rise of AI-Native Applications

The recent growth of Generative AI has led to a new type of system: **AI-Native Applications**. Unlike traditional apps where the database is the main source of truth, these architectures use the Large Language Model (LLM) as a reasoning engine. In this setup, the backend does more than just read and write data; it performs a great power in "Inference." This means processing existing data (like a PDF) to create new content—such as summaries, audio scripts, or quizzes—that did not exist in the database before [12].

2.1.6 From Deterministic to Probabilistic

This shift requires a change in architectural thinking: changing from **Deterministic Logic** to **Probabilistic Outputs**. In a standard web request, the result is binary (success or fail) and defined by returning code. In contrast, interactions with LLMs are variable; the same input might give slightly different outputs. Also, the system must handle variable latency and potential errors (hallucinations) [13]. Therefore, the backend architecture must adapt to this uncertainty by adding mechanisms for prompt management, validation, and asynchronous processing. This marks the transition from simple content management to intelligent knowledge services.

2.2 Large Language Models (LLMs) and Generative AI

The recent shift in Artificial Intelligence is characterized by the change from discriminative models, which classify existing data, to generative models that synthesize new data distributions. To understand why LLM models are suitable for automating academic resource generation, we must look at their architectural basis, their ability to learn from context, and their handling of non-textual outputs.

2.2.1 The Transformer Architecture

The technical feasibility of processing full academic papers relies on the Transformer architecture, proposed by Vaswani et al. (2017) [14]. Before Transformers, sequence processing used Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs). These older architectures processed words one by one sequentially, which made them computationally inefficient and prone to forgetting information over long sequences [14].

The Transformer solved this by using the "Self-Attention" mechanism. Instead of reading sequentially, the model calculates the relationship between every word in a sequence simultaneously. This allows the model to process a whole document in the same time (parallelization). For this thesis, this architecture is the primary reason why feed a short page PDF can be fed into the system and expect the model to connect the methodology section with the results section without losing context. It enables the handling of "Long-Range Dependencies" necessary for summarizing complex scientific arguments.

2.2.2 Foundation Models and In-Context Learning

Modern systems utilize Foundation Models, a class of models trained on broad data capable of adapting to diverse downstream tasks, as introduced by the Stanford Institute for Human-Centered AI (HAI) [12].

A critical feature for engineering applications is In-Context Learning or Few-Shot Learning. As demonstrated in the seminal GPT-3 paper by Brown et al. (2020), these models can adapt to specific tasks based on instructions given in the prompt, without updating internal parameters of their model [15]. Furthermore, Wei et al. (2022) highlighted the "Emergent Abilities" of these models, showing that sufficiently large models begin to exhibit reasoning chains that were not explicitly trained [16]. In this project, these capabilities dictate the system design: instead of the resource-intensive process of fine-tuning a model, we utilize Prompt Engineering. By providing the model with a clear schema and examples within the input window, the system directs the general-purpose model to function as a specialized academic tool. A comprehensive survey by Zhao et al. (2023) confirms that this approach is currently the most efficient paradigm for application development [2].

2.2.3 Multimodality and Structured Generation

Early LLMs were limited to text-to-text tasks. However, recent developments have expanded their capabilities to include Multimodal processing and structured output generation. OpenAI's technical report (2023) highlights that newer models (like GPT-4) can understand images and generate code [17].

In the proposed pipeline, this capability is used to convert text into visual formats. The system does not generate pixels; rather, it instructs the LLM to act as a translator from natural language to code. For example, to create a Mind Map, the model generates Mermaid.js syntax. To create an Audio Overview, it generates a structured script. This ability to output strict formats (like JSON or Markdown) rather than just free text is what allows the backend to automate the creation of the 9 distinct resource types.

2.3 Orchestration Frameworks for LLM Applications

While Large Language Models provide the core reasoning engine, using them directly via raw APIs (such as the OpenAI API) presents significant engineering challenges for building complex web applications. The primary issue is that raw LLM endpoints are stateless. The model does not retain memory of previous interactions, nor does it have intrinsic access to external private data, such as the specific PDF papers stored in a laboratory's database. Mialon et al. (2023) define

this limitation clearly: while models have parametric knowledge (learned during training), they lack "source knowledge" (access to current or proprietary files) unless it is explicitly provided in the input context [18].

To build an application that can process a file, remember user preferences, and execute a sequence of tasks, a middleware layer with prompts is required. This layer is responsible for managing the context window, retrieving external documents, and formatting prompts. This architectural pattern is often referred to as "Augmented Language Models" or "LLM Orchestration" [18].

2.3.1 The LangChain Ecosystem

To address these integration challenges, Chase (2022) introduced LangChain, an open-source framework designed to assist developers in building applications powered by LLMs [19]. Unlike simple API wrappers, LangChain focuses on composability. It treats the LLM as one module in a larger software pipeline, allowing developers to connect the model to other sources of data and computation. This study utilizes LangChain as the "glue layer" of the backend architecture, bridging the gap between the static PDF files stored on the server and the inference capabilities of the AI model.

The framework is built around three core concepts relevant to this thesis:

1. Chains: A Chain is a sequence of calls. In a raw API interaction, the developer sends one prompt and gets one answer. However, complex tasks often require breaking the problem down into steps. Wu et al. (2022) describe this as "Prompt Chaining," where the output of one step becomes the input for the next [20]. In this project, the "Automated Pipeline" relies heavily on this concept. For example, the system uses a sequential chain: first, it uses a *Loader* to extract text from the PDF; second, it passes the text to a *Summarizer Chain* to condense the information; finally, it passes the summary to a *Formatter Chain* to ensure the output is valid JSON.

2. Agents: Agents use the LLM as a reasoning engine to determine which actions to take and in what order. As proposed in the "ReAct" framework by Yao et al. (2023), an agent can analyze a request and dynamically decide to use specific tools [21].

Relevance to this study: While Agents offer dynamic flexibility, they introduce unpredictability and unnecessary token consumption. In this project, the goal is to generate a distinct set of 9 specific resources (e.g., Summary, Quiz, Mind Map) for every paper. Merging these instructions into a single Agent prompt would dilute the model's attention and risk context overflow. Therefore, this study **deliberately excludes the Agent pattern** in favor of modular, isolated Chains. This ensures that each resource type uses a specialized Prompt Template optimized for that specific task, maximizing generation quality and token efficiency.

3. Memory: In conversational AI, "Memory" stores interaction history. However, in this system's automated pipeline, the concept is adapted to "Context Window Management." Since the goal is batch processing (generating results from a specific paper), the system ensures that the extracted text from the paper PDF is efficiently loaded and embedded into the prompt context for each generation task, ensuring the AI has full visibility of the source material without needing multi-turn conversation memory.

2.4 Asynchronous Architecture and Event-Driven Design

Integrating Large Language Models into a web environment introduces performance characteristics that differ fundamentally from standard web applications. While the previous section discussed the logic of orchestration, this section addresses the system architecture required to handle the computational costs and timing constraints of LLMs.

2.4.1 The Latency Challenge

In traditional web development, the "Request-Response" cycle is expected to be near-instantaneous. Standard HTTP gateways (such as Nginx or Gunicorn) often have strict timeout limits, typically ranging from 30 to 60 seconds. However, LLM inference is computationally intensive. A comprehensive evaluation by Xu et al. (2024) indicates that the "Time to First Token" (TTFT) and total generation time for long-context tasks (like summarizing a research paper) can span from several seconds to minutes, far exceeding the tolerance of a synchronous web request [13]. Keeping a user's browser connection open for this duration is impractical and leads to poor user experience and potential connection timeouts.

2.4.2 The Producer-Consumer Pattern

To resolve this timing mismatch, software architecture relies on decoupling the interface from the processing logic. Fowler (2002) describes this as the Producer-Consumer pattern within enterprise architectures [22]. In this model, the web server acts as the "Producer," placing a task (e.g., "process this paper") into a queue and immediately returning a confirmation to the user. A separate, background process acts as the "Consumer" (or Worker), picking up tasks and executing them independently. This approach ensures that the web interface remains responsive regardless of how long the background analysis takes. Hohpe and Woolf (2003) further classify this as an asynchronous Message Channel pattern, which guarantees that messages are stored reliably until they can be processed [23].

2.4.3 Time-Based Scheduling and Container Orchestration

To automate the execution of these resource-intensive tasks, this study implements a Time-Based Scheduling strategy utilizing the Linux Cron daemon. As described by Nemeth et al. (2017) in the Unix System Administration Handbook, Cron is the standard utility for scheduling periodic jobs in Unix-like operating systems, offering robust reliability independent of the application layer [24].

In the specific implementation of this project, the application is containerized using Docker, a platform that packages software into standardized units called containers [25]. The scheduling logic operates at the system level: a Cron job is configured to trigger a restart of the backend Docker container at fixed intervals (e.g., hourly).

This design leverages the application's "Initialization Routine." Upon container startup, a bootstrap script automatically scans the database for research papers marked as "pending." If new data is found, the generation pipeline is triggered immediately.

This approach—effectively a periodic batch process—offers distinct advantages for AI-integrated systems. It ensures a "clean state" for memory-intensive LLM operations, mitigating potential memory leaks or resource exhaustion common in long-running Python processes involving heavy inference libraries.

2.5 Prompt Engineering and Structured Data Adaptation

Integrating Large Language Models in a production web environment requires more than simply sending text to an API. A fundamental challenge lies in controlling the stochastic nature of the model to produce deterministic, machine-readable outputs required by the web application. This section discusses the methodologies employed to bridge the gap between natural language generation and software engineering requirements.

2.5.1 Prompt Engineering as Software Design

In the context of this system, "Prompt Engineering" is not merely the act of writing questions but serves as the primary interface definition between the application logic and the model. As categorized by White et al. (2023), prompt engineering has evolved into a discipline with established design patterns similar to software architecture patterns [26].

This study implements specific patterns identified by White et al. to guide the generation process. For instance, the "Persona Pattern" is used to instruct the model to act as a "Senior Academic Researcher" when summarizing papers, ensuring

the tone remains formal and objective. Furthermore, the "Template Pattern" is applied to the generation of quizzes and summaries. Instead of allowing the model open-ended freedom, the prompt provides a rigid skeletal structure that the model must fill. This approach aligns with the findings of Liu et al. (2023), who argue that pre-train, prompt, and predict has become the new paradigm in NLP, replacing the need for task-specific fine-tuning [27].

2.5.2 The Impedance Mismatch in Web Engineering

A critical engineering hurdle addressed in this thesis is the "Impedance Mismatch" between AI capabilities and Web standards. Modern web front-end frameworks (such as the one used in the **inNuCE** platform) rely on strict data structures. Components like the search filter, quiz interactive cards, and mind map renderers require valid JSON (JavaScript Object Notation) or XML formats to function correctly.

However, LLMs are probabilistic engines designed to output unstructured natural language. If a model generates a summary but includes conversational filler (e.g., "Here is the JSON you asked for...") or misses a closing bracket, the web front-end parser will fail, causing the application to crash. Sculley et al. (2015), in their seminal paper on machine learning systems, classify this type of integration issue as "Glue Code" and "Entanglement" technical debt, noting that handling the interface between ML models and surrounding infrastructure is often more complex than the model itself [28].

2.5.3 Schema Enforcement and Output Parsing

To resolve this mismatch, this study moves beyond simple prompting to Schema Enforcement. The methodology draws upon the approach proposed by Josifoski et al. (2023), who demonstrated that injecting structural constraints (such as SQL schemas or JSON definitions) into the prompt context significantly improves the model's ability to generate executable code [4].

In the "Automated Pipeline," this is implemented using LangChain's Output Parsers. The system defines a data model (using the Pydantic library) that specifies exactly what fields are required—for example, a "Quiz" object must contain a list of questions, and each question must have four options and one correct answer index. These constraints are injected into the prompt as a formatting instruction. During post-processing, the Output Parser validates the raw text returned by the LLM against the schema. If the validation fails (e.g., malformed JSON), the system can automatically trigger a "fix" request, asking the model to correct its syntax. This mechanism ensures that the data stored in the database is always strictly structured and ready for front-end rendering.

2.6 Context Management Strategies

A central challenge in building LLM-based applications is managing the "Context Window"—the maximum amount of text the model can process in a single interaction. For scientific resource generation, the system must ensure that the model has access to the relevant information from the uploaded PDF without exceeding token limits or losing critical details. Two primary strategies exist to handle this: Retrieval-Augmented Generation (RAG) and Full-Context Injection.

2.6.1 Retrieval-Augmented Generation (RAG)

The standard approach for handling large document collections or books is **Retrieval-Augmented Generation (RAG)**. As defined by Lewis et al. (2020), RAG involves splitting a document into smaller "chunks," storing them in a vector database, and retrieving only the top-k most relevant chunks based on a user's query [29].

This method is highly efficient for "Open-Domain Question Answering" where specific facts need to be located within massive datasets. Gao et al. (2023) note in their survey that RAG is essential when the input data exceeds the model's context window [30]. However, RAG introduces a **fragmentation problem**: by slicing a text into independent chunks, the logical flow between the introduction, methodology, and conclusion is often severed. This makes it difficult for the model to generate a comprehensive summary or a mind map that requires a holistic view of the document structure.

2.6.2 Full-Context Injection Strategy

In contrast to RAG, **Full-Context Injection** involves placing the entire textual content of the document into the prompt.

Design Decision: This study analyzes the specific characteristics of the target data—research papers uploaded to the laboratory website. The average length of these documents is not large. This is within the effective context window of modern Foundation Models (e.g., GPT-4 supports 128k tokens).

Consequently, this project adopts the **Full-Context Injection** strategy. This approach prioritizes **Global Coherence** over retrieval efficiency. By feeding the complete text to the model, the system ensures that the AI can correlate the hypothesis presented in the abstract with the results discussed in the conclusion, avoiding the information loss associated with chunking.

While Liu et al. (2023) identified the **"Lost in the Middle"** phenomenon—where models struggle to retrieve information located in the middle of very long contexts—their experiments indicated that performance degradation becomes significant primarily at much larger scales or when precise key-value retrieval is required [3]. For the specific scope of this project (short academic papers), the input size

remains in the "safe zone." Therefore, avoiding the structural complexity of RAG in favor of a full-context approach represents the optimal engineering trade-off for generating high-quality summaries.

2.7 Human-in-the-Loop (HITL) Systems

While the previously discussed architectures enable automation, the probabilistic nature of Large Language Models introduces reliability challenges that distinguish them from deterministic software systems. Unlike a standard database query which returns exact matches, an LLM generates text based on statistical likelihood, leading to potential inaccuracies. Consequently, relying solely on full automation for academic dissemination carries the risk of publishing incorrect information. This section outlines the theoretical basis for integrating human oversight into the system architecture.

2.7.1 AI Reliability and Hallucinations

The primary motivation for restricting full automation is the phenomenon of "Hallucination". In the context of Natural Language Generation (NLG), Ji et al. (2023) define hallucination as generated content that is "nonsensical or unfaithful to the provided source content" [31].

Even with advanced techniques like the Full-Context Injection described in Section 2.6, models may still misinterpret specific numerical data from a results table or attribute a quote to the wrong author. As noted by Zhang et al. (2023), these errors are often "fluent and convincing," making them difficult to detect without careful scrutiny [32]. In an academic context, where accuracy is paramount, such unfaithful generation is unacceptable. Therefore, the system design must acknowledge that the AI component is a non-deterministic actor, so each generated contents need to be valided.

2.7.2 The HITL Design Pattern

To mitigate these risks, this study adopts the Human-in-the-Loop (HITL) design pattern. Amershi et al. (2019), in their foundational guidelines for Human-AI Interaction, emphasize the necessity of providing mechanisms for efficient correction and control [5].

Specifically, this aligns with the "Generate-Review-Publish" workflow pattern described by Wang et al. (2021) in data-driven decision systems [33]. In this model, the AI algorithm functions not as the final decision-maker but as a "provisional drafter." The system architecture must explicitly support a state transition where

generated resources are held in a staging area (the Admin Panel) before being exposed to the public frontend.

In the implementation of this thesis, this theoretical framework is translated into a "Gatekeeper" mechanism. The backend pipeline does not write directly to the public Publication table. Instead, it populates a Review queue. The administrator acts as the human loop, verifying the structured JSON output (e.g., checking if the Quiz options are correct) and the synthesized summaries. Only upon a specific API trigger (the "Approve" action) does the content become visible to end-users. This redundancy ensures that the efficiency of automation does not compromise the integrity of the scientific content.

2.8 Analysis of Existing AI-Assisted Research Tools

The integration of AI in academic workflows has led to a proliferation of commercial tools designed to assist researchers. Among these, **Google NotebookLM** currently represents the state-of-the-art benchmark for document understanding. Before detailing the proposed custom pipeline, it is essential to analyze the capabilities of NotebookLM and, crucially, explain why it fails to meet the automation requirements of the proposed web platform.

2.8.1 The Benchmark: Google NotebookLM

Built upon the Gemini 1.5 Pro model with a massive context window (up to 2 million tokens), NotebookLM distinguishes itself from earlier tools like ChatPDF or Semantic Scholar through its multimodal capabilities [34].

Its most significant feature is the **"Audio Overview,"** which generates a conversational podcast between two AI hosts discussing the uploaded source material. Furthermore, it employs a strict "Source Grounding" technique, reducing hallucinations by restricting the model's answers exclusively to the provided documents. In the context of this thesis, NotebookLM serves as the **Quality Baseline**. The "Dual-Track" strategy proposed in Chapter 1 acknowledges that, for certain qualitative tasks (like the audio experience), NotebookLM remains superior to open-source alternatives.

2.8.2 Architectural Limitations for Web Integration

Despite its high performance, NotebookLM operates as a consumer-facing SaaS product rather than a developer platform. This creates three fundamental "blocking issues" for integration into the *inNuCE* laboratory website:

1. Lack of Programmatic Access (The API Gap): The most critical limitation is the absence of a public API. As of early 2025, automating NotebookLM

requires manual interaction via a web browser. There is no endpoint to programmatically upload a PDF or retrieve a generated summary. This forces a **Manual Workflow**: the administrator must manually log in, upload files, wait for generation, and copy-paste results. As discussed by Gargeya et al. (2021), such "Walled Garden" architectures prevent the creation of event-driven workflows, making real-time automation impossible [35].

2. Incompatibility with Structured Web Data: The output from NotebookLM is designed for human reading (Markdown text) or listening (Audio files). It cannot generate the strict **JSON (JavaScript Object Notation)** structures required by modern web front-ends. For example, the *inNuCE* platform features an interactive "Use Case" filter and a "Quiz Card" component. These components require specific data fields (e.g., `"correct_answer_index": 2`). NotebookLM provides no mechanism to enforce this schema. Relying on its textual output would require writing complex and fragile regular expressions (Regex) to parse the data, which is widely considered an anti-pattern in software engineering [28].

3. Data Siloing: Using NotebookLM creates a dependency on an external, closed ecosystem. The generated intelligence (summaries, insights) resides on Google's servers, decoupled from the laboratory's own database. This separation complicates data management, as there is no single "source of truth" for the paper's metadata.

2.9 Research Gaps and Problem Formulation

A review of the current literature and technological landscape reveals a distinct dichotomy. On one side, advanced Large Language Models demonstrate sufficient reasoning capabilities to summarize and interpret scientific texts [2]. On the other side, modern Web Architectures have evolved into robust, event-driven systems capable of handling complex asynchronous tasks [22]. However, a significant gap remains in the engineering integration of these two domains, specifically within the context of academic dissemination platforms.

2.9.1 The Lack of End-to-End Automation

Existing solutions typically fall into two extremes: manual usage of high-level SaaS tools (like NotebookLM) or raw usage of low-level APIs. As identified in Section 2.8, reliance on SaaS tools introduces "SaaS Silos," preventing data interoperability and creating a manual bottleneck for administrators [35]. There is a lack of open-source reference architectures that demonstrate how to build a fully automated pipeline—from PDF upload to multi-modal resource generation—without human intervention.

2.9.2 The Structured Data Gap

While Prompt Engineering is a well-researched field [26], there is limited literature addressing the "Impedance Mismatch" in a production web environment. Most research focuses on the quality of text generation, rather than the interoperability of the output. As noted by Sculley et al. (2015), the "glue code" required to make machine learning outputs usable in real-world systems often represents a larger technical debt than the model itself [28]. A methodology for reliably enforcing strictly structured outputs (JSON) suitable for driving dynamic web components (like the interactive quizzes proposed in this thesis) is under-represented in current applied research.

2.9.3 Summary and Proposed Solution

In a nutshell, there is no existing integrated solution that addresses the end-to-end lifecycle of academic content within a web platform. Specifically, current tools lack a unified architecture that combines:

1. **Asynchronous Automation:** A mechanism to automatically trigger the generation of all nine resource types immediately upon PDF upload, handling the latency of LLM inference in the background without blocking the user interface.

2. **Hybrid Quality Assurance (Dual-Track):** A "Generate-Review-Publish" workflow. The system uses the automated LangChain pipeline as the default engine, but allows administrators to intervene. If an automated result is unsatisfactory, the administrator can seamlessly swap it with a manually generated version from NotebookLM, ensuring high quality before approval.

3. **Controlled Dissemination:** A "Gatekeeper" mechanism where generated resources (validated against a strict JSON schema) remain stored in the database but invisible to public users until explicitly published by the administrator.

This thesis aims to bridge these gaps by designing and implementing this **"Automated Intelligent Generation Pipeline."** This system proves that LLMs can be effectively engineered into a robust backend architecture, transforming them from passive chat interfaces into active, structural, and controllable components of a production-grade web application.

Chapter 3

Material and Methods

With fast development of internet technology, information flows very quickly nowadays. However, websites for academic research are usually different. The resource content and form are often static and single. This situation is not good for non-expert people to understand research, which creates barrier between researchers and normal audience.

As mentioned in Chapter 1, the bottleneck is the text way of dissemination. Papers are full of dense texts. This way is definitely not good for propagation, so people lose interest easily. On the other hand, for some dynamic WEB platforms with too many resource contents, there are other problems. For example, data latency happen often. What's more, the cost of website management and maintenance are high since administrators need to update data manually.

This chapter describe technic design and development of inNuCE website platform. Main goal is to transform static academic papers into diverse interactive digital resources. The discussion covers whole software development life cycle. It starts from architecture decisions, then moves to technology selection, and finally explains specific implementation of core modules.

Specifically, engineering implementation focuses on several key points:

1. Dual-Workflow Access Architecture is designed to separate standard user access (browsing the inNuCE website) from administrator access (managing and updating the information). This design model makes system secure.

2. Robust Web Infrastructure is established. It builds backend services and persistent database which is the foundation, which supports Content Management System and helps to manage publications and use cases efficiently.

3. Automated Intelligent Generation Pipeline is developed. This is the core part of this thesis. It uses **Large Language Models (LLMs)** and Prompt Engineering. The pipeline parse PDF documents automatically, then it generates 9 types of multimodal resources by the output message from the LLMs. For example, it creates podcasts and quizzes. These resources are stored in the server and then update the resource url to database without human work.

4. Micro-services inspired architecture is adopted, which does not use single big code block. What's more, it takes the coding function of Object Oriented Programming(OOP) . Each service is separated from other services. Finally, whole project is deployed by using **Docker**. The application is fully containerized. This make deployment on the same or different server(s) become easier and stabler without considering environmental conflict.

The remainder of this chapter is organized as follows: Section 3.1 provides a high-level overview of the system architecture; Section 3.2 justifies the technological stack and tools employed; Section 3.3 details the implementation of the web infrastructure and data management modules; Section 3.4 offers an in-depth analysis of the automated generation pipeline; Section 3.5 describes the containerization strategy and deployment topology; Section 3.6 summarizes the engineering contributions of this study.

3.1 System Architecture Overview

The architecture of inNuCE platform is designed to be stable and secure, and its main goal is to handle two different tasks: public user access and administrator operations. The system uses a modular design, so different parts can work independently without affecting each other.

Figure 3.1 shows the high-level architecture. As the figure displays, the whole system is divided into several layers. Frm top to bottom, there are Client Layer, Gateway Layer, Service Layer, and Data Layer. In addition, there is a special Pipeline part at the bottom designed for auto AI tasks.

The architecture in Figure 3.1 effectively solves the problem of "coupling". In old systems, user function and admin function are mixed together, so if one part crash, the whole system might crash. In this technic design, these functions are totally separated to ensure stability. The following sections will explain why the design is like this and how data flows in the system.

3.1.1 Architectural Design Principles

The system follows two main principles during the construction. The one is Layered Architecture, and another one is Decoupling.

1. Layered Architecture

The system is organized into four clear layers, and each layer has a specific job. They work together but do not interfere with other layers too much.

- **Client Layer:** This is where interaction happens. For public users, browsers on PC or mobile digital devices are used, while for the administrator, it is a private computer terminal or IDE with server private key to vist.

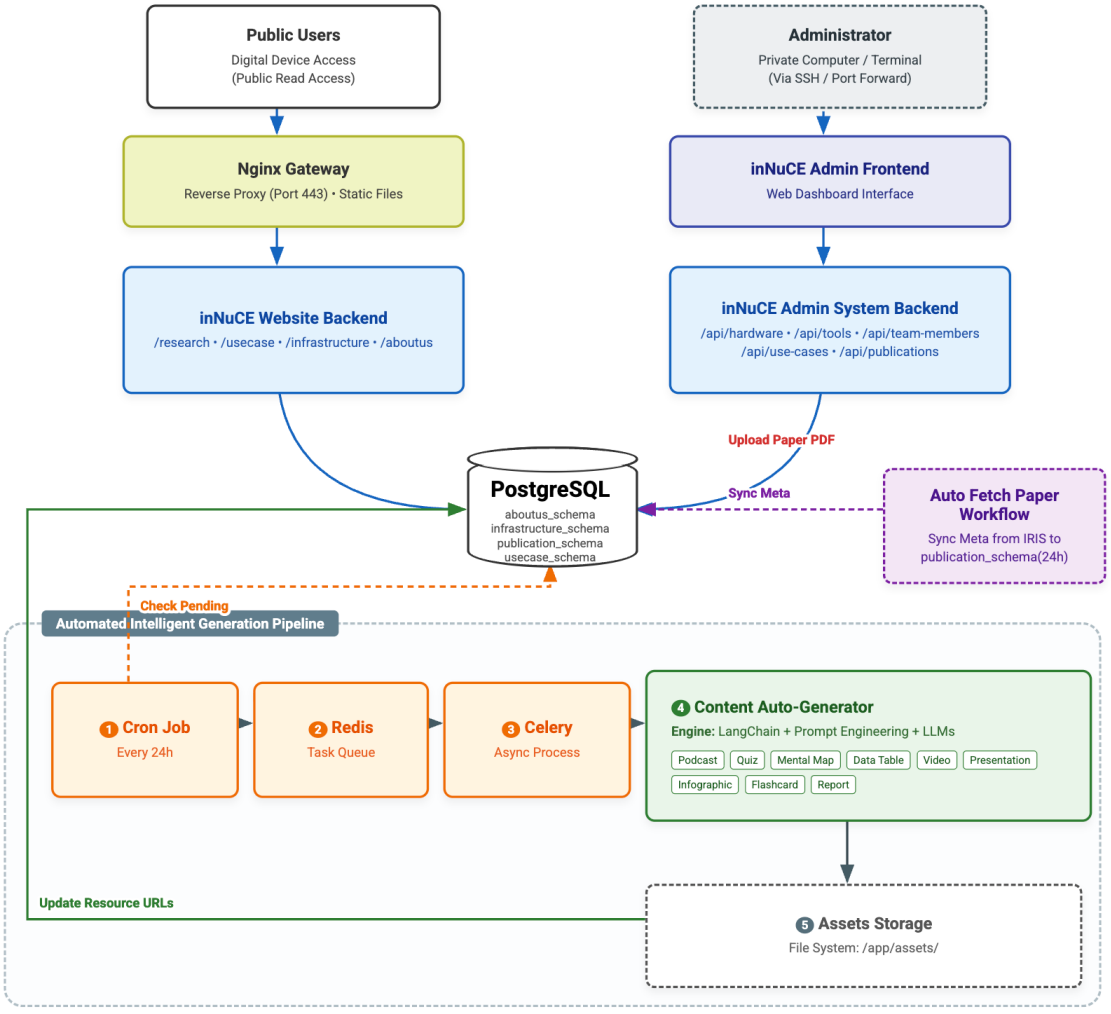


Figure 3.1: inNuCE Website Technical Architecture

- **Gateway Layer:** Nginx is used here to act as an entry point for users. It handles HTTPS requests and load balancing, which also protects backend services from direct attack.
- **Service Layer:** This layer contains logic code. It includes the Node.js Backend for the inNuCE website and Flask Backend for the admin system. They process business logic and send data back to the front end.
- **Data Layer:** This is the bottom layer. It includes the PostgreSQL database, which stores all meta data like paper info and user info.

2. Decoupling Strategy

Decoupling is quite important in this project because it separates Public Access Service (Node.js [36]) from Admin Operations Service (Flask [37]).

- **Security Isolation:** Public users only need to read data, and they do not need to manipulate data into database. So, the Node.js service mostly has "Read" permission. However, the Admin service need "Write" permission. By separating them, the database remains safe from being deleted even if Node.js is hacked.
- **Performance Isolation:** Public traffic is high because many people visit website to obtain resources, but admin traffic is low. Admin tasks (like AI generation) consume many CPU resource. If they are mixed, AI tasks might slow down the public website. Separating them make sure the public website is always fast.

3.1.2 Dual-Workflow Logic

As shown in Figure 3.1, the system has two distinct workflows. The left side is for public users, and the right side is for the administrator. Different paths are used to access the system to ensure security and efficiency.

1. Public User Workflow (The Left Path)

This path is designed for standard web access, and it is optimized for speed.

- **Step 1: Digital Device Access.** Browsers are opened on user devices and the URL is typed to visit the website.
- **Step 2: Nginx Gateway.** The request hits Nginx first. Nginx checks if the request is valid, and it also serves static files (like images, css) directly to make response faster.
- **Step 3: inNuCE Website Backend.** If the request is for API (like getting paper list), Nginx passes it to backend, which is good at handling many TCP requests at the same time.

- **Step 4: Database Read.** Server queries PostgreSQL to get necessary data and then sends it back to the frontend displaying on the screen.

2. Administrator Workflow (The Right Path)

This path is designed for security, so it does not expose the Admin Panel to the public internet directly.

- **Step 1: Private Access.** A private computer is used by the administrator to manage the system. The public domain name is not accessed directly.
- **Step 2: SSH Tunnel / Port Forwarding.** This is a key security feature. A secure SSH tunnel is established to the server, and it maps the local port to the remote server port. This means the Admin Frontend is only visible inside the tunnel, so hackers from outside cannot see the login page.
- **Step 3: Admin Frontend & Backend.** Once connected, the Dashboard Interface can be managed by the admin, and information and files can be updated and uploaded safely.
- **Step 4: Internal API Call.** The Frontend calls the Python Flask Backend. Since both services are inside the internal network (or Docker network), the communication is fast and safe.
- **Step 5: Triggering Pipeline.** When a paper is uploaded, the backend of admin system saves the file and automatical triggers the AI pipeline later. This part will be discussed in Section 3.4.

3.2 Technological Stack and Justification

The selection of technology stack is a critical step in software engineering before starting development. It needs to balance performance, development efficiency, and ecosystem compatibility. In this project, we do not stick to a single language. Instead, a hybrid approach is adopted to solve specific problems in different scenarios. The following sections describe the details of each component and the reason for choosing them.

Table 3.1 gives a summary of the technology stack used in the inNuCE platform.

3.2.1 Backend Frameworks: Node.js vs. Python Flask

The system architecture uses two different backend languages. This decision is based on the specific requirements of the Public Workflow and the Admin Workflow.

Component	Technology	Version	Key Reason
Public Backend	Node.js	v22.19.0	High concurrency, Non-blocking I/O, Fast response for read operations.
Admin Backend	Python	v3.11.9	Native AI ecosystem support (PyTorch, LangChain), Easy integration with Celery.
Database	PostgreSQL	v14-alpine	Strong consistency (ACID), JSONB support for flexible metadata storage.
AI Orchestration	LangChain	>=v0.2.0	Efficient context window management, Prompt chaining capability.
LLM Models	Google	gemini 2.5 pro	Advanced reasoning capability for complex content generation.
Task Queue	Redis	v7-alpine	Low latency in-memory data store, Standard broker for Celery.
Async Process	Celery	v5.2.7	Distributed task scheduling, Robust asynchronous processing.
Containerization	Docker	v29.1.3	Environment isolation, Reproducible deployment.

Table 3.1: Summary of Technology Stack

Public Access Side: Node.js

For the public-facing website, Node.js is selected as the runtime environment. The framework used is Express.js. There are two main reasons for this choice.

First, Node.js uses an event-driven, non-blocking I/O model. Public users mostly perform "Read" operations, such as viewing the paper list or opening a specific case study. These operations involve many database queries but less CPU calculation. Node.js handles these I/O tasks very efficiently, so it can support thousands of concurrent connections with low resource consumption.

Second, the response speed is critical for user experience. Node.js starts up quickly and processes JSON data natively. Since the frontend also uses JavaScript (or data is exchanged in JSON format), utilizing Node.js for the backend reduces the overhead of data conversion.

The following code snippet shows the dependency configuration in `package.json`, which is lightweight, as shown in Listing 3.1:

```
{
  "name": "innuce-website-backend",
  "dependencies": {
    "cors": "^2.8.5",
    "dotenv": "^16.6.1",
    "express": "^4.18.2",
    "helmet": "^7.1.0",
    "morgan": "^1.10.1",
    "pg": "^8.11.3"
  }
}
```

Listing 3.1: Dependencies configuration in package.json for inNuCE - website - backend

Admin and Pipeline Side: Python Flask

For the administrator system and the AI generation pipeline, Python is the best logical choice, especially running server is lightweight.

The primary reason is the dominance of Python in the AI ecosystem. Most cutting-edge AI libraries, such as LangChain, OpenAI SDK, and PyPDF2, are native to Python. If Node.js was used here, we would need to write complex wrappers to call Python scripts, which is unstable and hard to debug. Using Python directly allows seamless integration with these libraries.

Another reason is task integration. Flask is a micro-framework, and it is lightweight and flexible. It integrates perfectly with Celery, which is a distributed task queue written in Python. This combination makes it easy to run heavy AI generation tasks to background workers.

3.2.2 Database System: PostgreSQL

Data storage is the core of any information system. In this project, PostgreSQL is chosen as the primary database. It is an advanced, open-source object-relational database system.

1. Structured Data Requirement

Academic papers have a very strict structure. Every paper has a title, authors, publication date, and abstract. This kind of data fits perfectly into a Relational Database Management System (RDBMS). We need to define clear schemas to ensure data quality. PostgreSQL enforces data types and constraints, so invalid data cannot enter the system.

2. Strong Consistency (ACID)

Consistency is vital for an academic platform. When an administrator updates the meta-data of a paper, public users must see the update immediately. PostgreSQL guarantees ACID (Atomicity, Consistency, Isolation, Durability) properties. This ensures that the data viewed by users is always accurate and up-to-date.

3. JSONB Support

Although the meta-data is structured, the generated AI resources are flexible. For example, a "Quiz" resource might have 10 questions or 13 questions. A "Podcast" resource is just a URL string. PostgreSQL offers a powerful feature called JSONB. It allows us to store these semi-structured data in a JSON column while still supporting efficient indexing and querying. This gives system the flexibility of NoSQL (like MongoDB) within a reliable SQL database.

3.2.3 AI & Orchestration: LangChain & LLMs

The core feature of inNuCE is the automated generation of resources. To achieve this, It leverages Large Language Models (LLMs) and an orchestration tool called LangChain.

Why LangChain?

Directly invoking the OpenAI API is possible, but it is not efficient for complex tasks. LangChain is introduced to solve several engineering challenges, which is totally opening source in the GitHub and has pretty active developing community.

The first challenge is Context Management. Academic papers are usually long, often exceeding 10,000 words. However, AI modal usually have a context window limit (e.g., 8k or 32k tokens). If it sends the whole paper at once, once reaching limitation, the request will be rejected. LangChain provides built-in utilities for "Chunking". It splits a long PDF into smaller text blocks automatically.

The second challenge is Chain of Thought. Generating a high-quality report requires multiple steps. For example, the system first needs to summarize the abstract, then analyze the methodology, and finally generate a conclusion. LangChain

allows us to define a "Sequential Chain". The output of the first step becomes the input of the second step. This makes the logic clear and easy to maintain.

Role of LLM Models

In this project, it utilizes the Google Gemini 2.5 Pro model. Access is managed through the Google Cloud API.

Gemini 2.5 Pro serves as the core intelligence engine. It is chosen for two main reasons. First, it has advanced reasoning capabilities. This is essential for complex tasks like generating podcast scripts or summarizing technical methodologies. Second, it supports a massive context window. Academic papers are often very long, but Gemini can process the entire document content at once without heavy chunking. This ensures the generated content is coherent and accurate.

3.2.4 Asynchronous Task Queue: Redis & Celery

In a standard web application, a request usually finishes within a few seconds. However, AI generation is different.

1. The Necessity of Async Processing

Generating a comprehensive PDF report or a video script takes time. It typically takes from 30 seconds to several minutes, depending on the length of the paper. If the web server (Flask) waits for the completion, the HTTP connection will timeout. The user browser will show an error, and the server thread will be blocked. To avoid this, the best choice is asynchronous processing.

2. Architecture Roles

The system uses a producer-consumer model to handle these heavy tasks.

- **Redis (Message Broker):** Redis acts as a buffer. When the admin upload the full-text paper and Cron(like a clock) will kick the container to run in every specific time period, the container will sends the tasks to Redis queue. Each task contains the Paper ID, the path of document in the server and the requested resource types and so on. Redis stores these tasks in memory.

- **Celery (Task Queue):** Celery is the consumer. It runs as a separate process in the background (or in a separate Docker container). It constantly listens the Redis for new tasks. When a task arrives, Celery picks it up and starts the AI generation pipeline. This design ensures that the web interface remains responsive even when heavy calculations are running in the background.

The following is a simplified code example for Celery Worker:

```
1 from celery import Celery
2 from ContentAutoGenerator import ContentAutoGenerator
```

```
3
4  # Initialize Celery app
5  celery_app = Celery(
6      'content_generation',
7      broker=os.getenv('CELERY_BROKER_URL', 'redis://redis:6379/0'),
8      backend=os.getenv('CELERY_RESULT_BACKEND', 'redis://redis:6379/1')
9  )
10
11 # Configure Celery
12 celery_app.conf.update(
13     task_serializer='json',
14     accept_content=['json'],
15     result_serializer='json',
16     timezone='UTC',
17     enable_utc=True,
18 )
19
20 @celery_app.task(bind=True, name='generate_content')
21 def generate_content_task(
22     self,
23     pdf_path: str,
24     resource_types: list = None,
25     output_dir: str = None,
26     llm_provider: str = None):
27
28     try:
29         # Update task state
30         self.update_state(state='PARSING', meta={'status': 'Parsing PDF...'})
31
32         # Initialize generator with parameters
33         generator = ContentAutoGenerator(
34             pdf_path=pdf_path,
35             resource_types=resource_types,
36             output_dir=output_dir,
37             llm_provider=llm_provider
38         )
39
40     except Exception as e:
41         # Log error and re-raise
42         self.update_state(
43             state='FAILURE',
44             meta={'status': f'Error: {str(e)}'}
45         )
46         raise
```

Listing 3.2: An Example Code for Celery Worker

3.3 Web Infrastructure Implementation

This section describes the design and implementation of the data layer. PostgreSQL is chosen as the core database because it supports strong type constraints and JSONB complex features. The system data model is divided into four independent schemas to achieve logical isolation.

3.3.1 Database Schema Design

The system data is organized into four schemas: `infrastructure_schema`, `usecase_schema`, `aboutus_schema`, and `publication_schema`. Each schema is responsible for a specific business area.

Infrastructure Schema

This schema manages hardware facilities and tools in the laboratory. The structure is simple, and it is mainly used for static display. Figure 3.2 shows the table design.

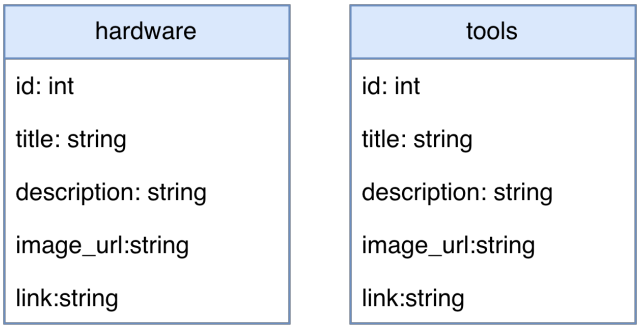


Figure 3.2: ER Diagram of Infrastructure Schema

- **Table: hardware.** It stores information about physical devices owned by the lab. The core fields include `id`, `title`, `description`, and `image_url`.
- **Table: tools.** This table stores information about software tools or small instruments. Its structure is consistent with the `hardware` table.

The design feature is that these two tables are independent. They do not have foreign key associations, which are suitable for static pages.

Usecase Schema

This is the most complex schema, and it manages specific research application cases. It adopts a "Many-to-Many" relationship design to support flexible classification, as shown in Figure 3.3.

1. Core Table: usecase

This table is the aggregation center. All tags point to this table. It contains basic fields like `title`, `description`, and `image_src`.

2. Tagging System

There are four dimension tables: `model`, `framework`, `keyword`, and `hardware`. They connect to the `usecase` table through pivot tables (e.g., `model_usecase`).

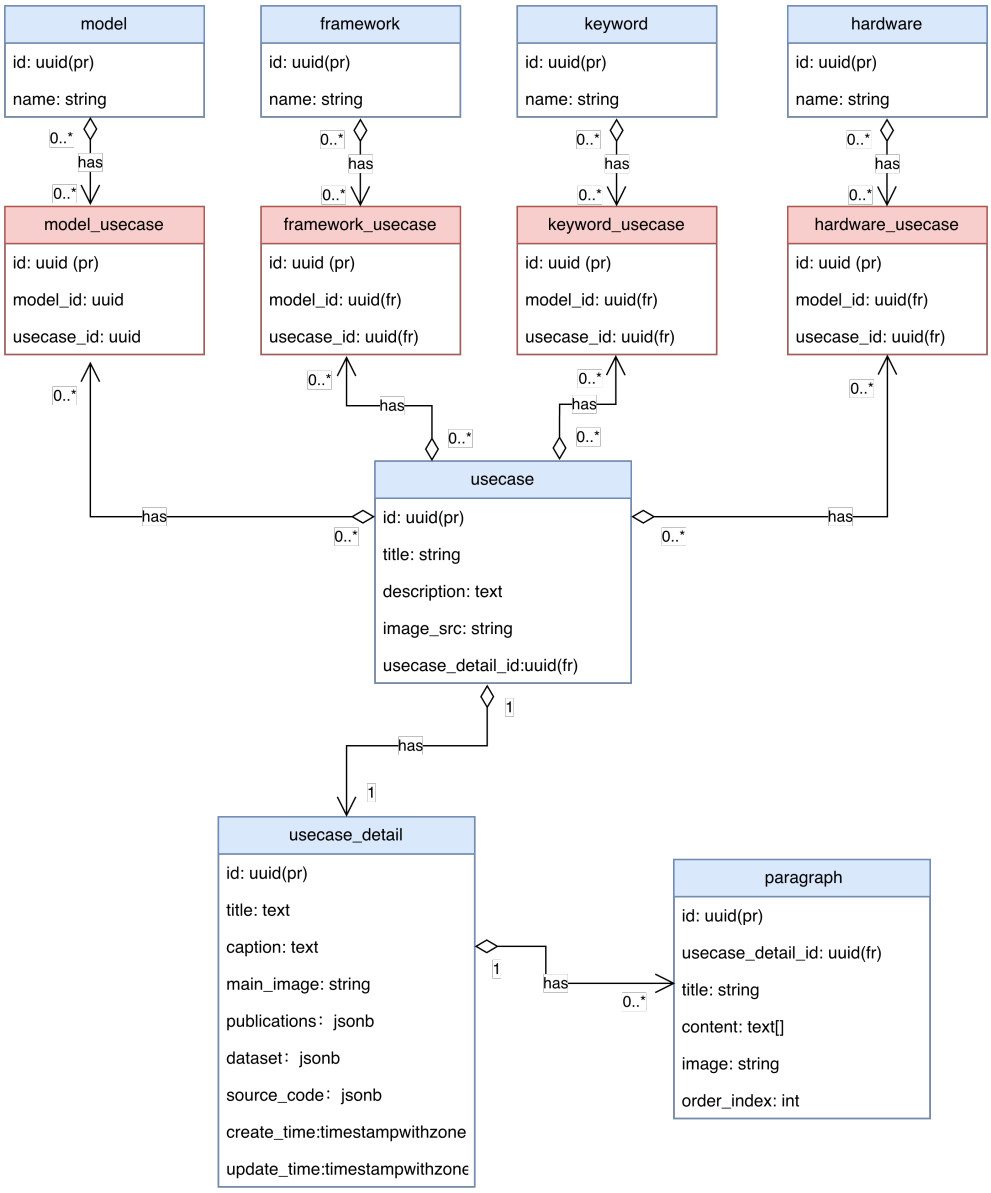


Figure 3.3: ER Diagram of Usecase Schema

The logic is that one case relate multiple models, and one model can be manipulate by multiple cases.

3. Detail Extension: usecase_detail

This table has a 1:1 relationship with the `usecase` table. It stores detailed long text and metadata. Fields like `publications`, `dataset`, and `source_code` use the JSONB type. Using JSONB provides great flexibility for storing unstructured data.

4. Content Chunking: paragraph

This table has a 1:N relationship with `usecase_detail`. It splits a long article into multiple paragraphs to support mixed layout of text and images.

About Us & Publication Schema

These two schemas work together to manage personnel and academic outputs. Figure 3.4 illustrates their relationship.

The **About Us Schema** (upper part) manages lab member information. The main table is `members`. It stores basic info like `name`, `surname`, and `role`. It also contains many profile link fields, such as `linkedin_profile` and `google_scholar_profile`. The `active` field is a boolean value. It controls whether a member is displayed on the frontend.

The **Publication Schema** (lower part) stores academic paper metadata. The data mainly comes from automated synchronization tasks.

- **Table: papers.** It stores information about the paper itself. Core fields include `title`, `keywords`, and `year`. The `state` field controls the publishing lifecycle.
- **Table: papers_authors.** This is an association table. It connects the `papers` table and the `members` table (from About Us Schema). The relationship is Many-to-Many. It has attributes like `author_rank` to record the order of authors.

In summary, this database design has high normalization features. The introduction of JSONB in `usecase_detail` and cross-schema association in `publication_schema` balances flexibility and data integrity. This provides a solid foundation for API development.

3.3.2 Public inNuCE Website Backend Service (Node.js)

The public inNuCE Website backend service is built based on the Node.js environment and Express.js framework. It uses a modular route architecture to provide RESTful APIs for the frontend React application. The core responsibility of this service is to provide read-only and reliable access for public users. It ensures that only reviewed content of the database can be displayed to the users.

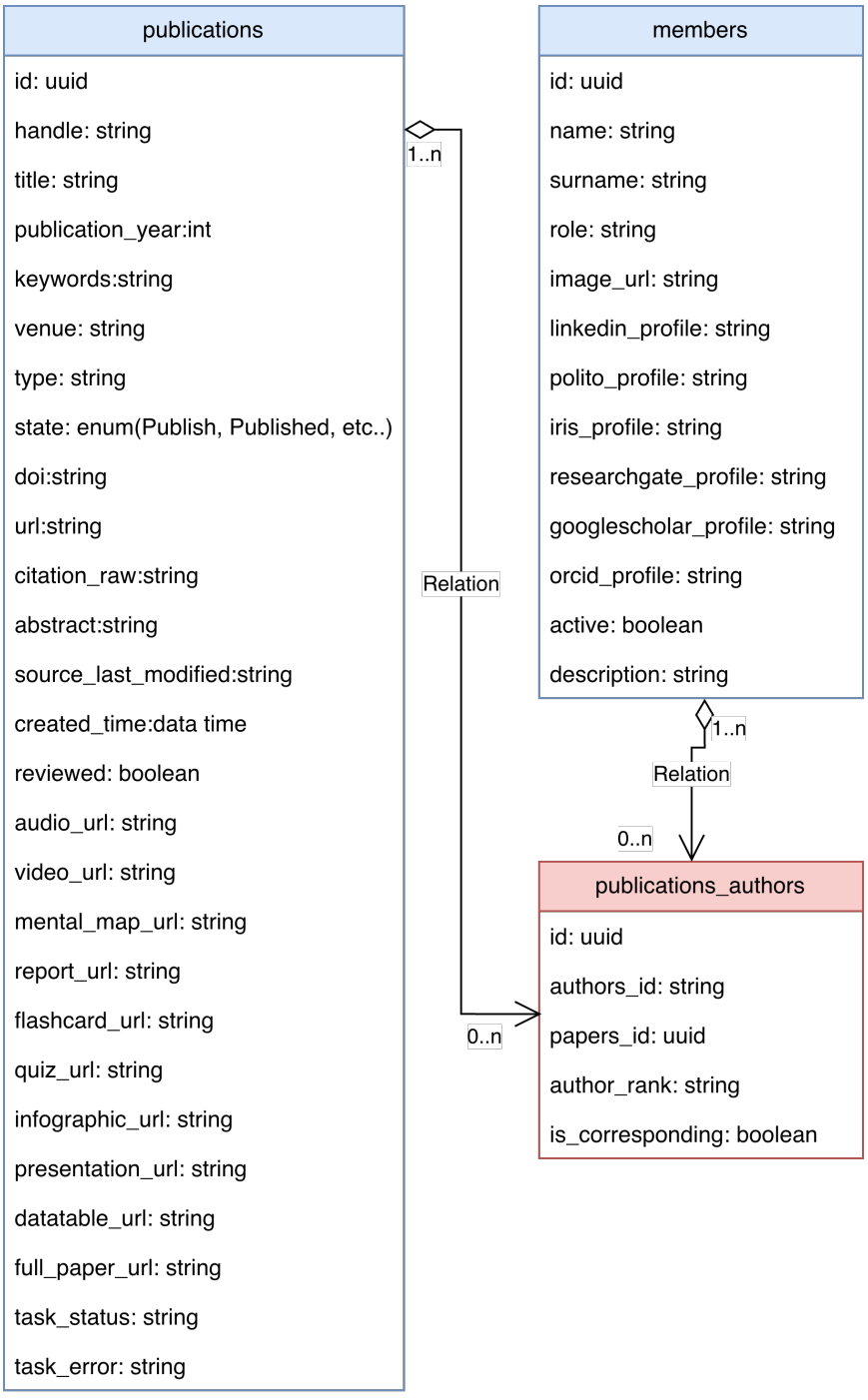


Figure 3.4: ER Diagram of About Us and Publication Schemas

API Route Design

The system implements four core API route modules.

1. Research Publication API (/research)

This route is responsible for returning all reviewed academic publications and their multimodal resources. The query filters data with `reviewed = true` condition, so only approved content is returned from the database, which includes basic paper information, keyword list, and access URLs for 9 types of generated resources and so on.

During data transformation stage, the system converts relative file paths stored in database into complete HTTP access URLs. This makes it convenient for frontend use by returning url link of resources. For the keyword field, the system parses comma-separated strings into array format. Specifically, for the `quiz_url` field, the system implements smart logic. If the URL starts with "http"/"https", it is returned directly. Otherwise, it concatenates the domain path of server.

2. Use Case API (/usecase)

This route provides three endpoints to support multi-dimensional queries.

- `/usecase/cards`: It returns a card-style overview of all cases. This includes title, description, thumbnail, and associated tags. The system uses PostgreSQL JSON aggregation function (`json_agg`) to organize Many-to-Many relationship data at database level. This avoids complex JOIN operations in application layer.

- `/usecase/:slug`: It returns detailed content based on case ID stored in database. Each paragraph contains title, text content, and optional image. The system uses `json_build_object` function to build nested JSON structure directly in database, which ensures correct order of paragraphs through `ORDER BY order_index` as well.

- `/usecase/filter`: This endpoint provides data source for frontend filtering function. It queries options from four dimensions in parallel. `Promise.all` is used to implement concurrent queries, which significantly improves response speed.

3. Other Routes

The `/aboutus` route provides team member information of inNuCE Lab, and `/infrastructure` route returns the lab hardware list. These routes follow the same design pattern to ensure consistency of API interface.

Request Processing Flow

Figure 3.5 illustrates the typical processing flow when a user visits the Research page. The process is divided into seven stages.

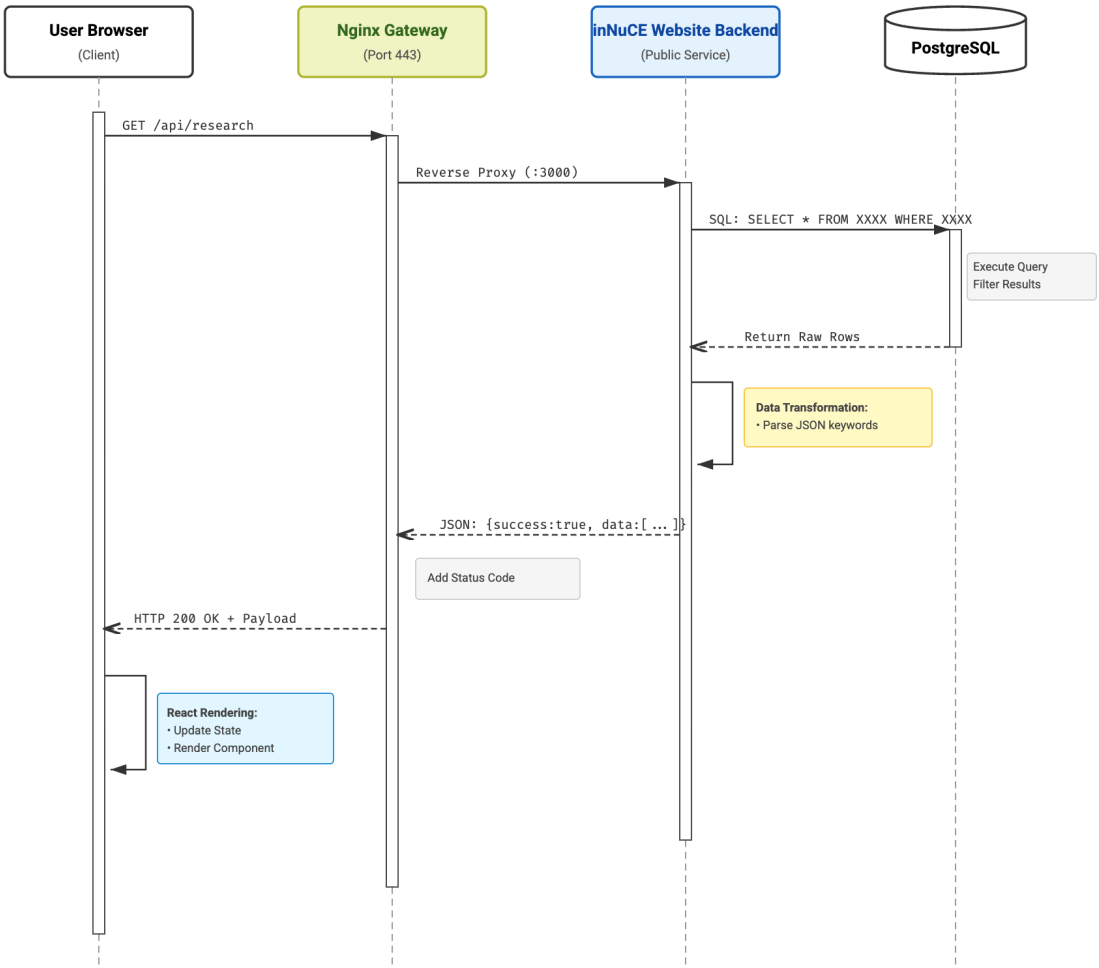


Figure 3.5: Sequence Diagram of User Request Processing Logic

1. **Request Initiation:** User visits Research page in browser. React application sends HTTP GET request to Nginx.
2. **Routing:** Nginx reverse proxies the request to port of inNuCe website backend based on configuration file.
3. **Data Query:** Express route handler sends parameterized query to PostgreSQL via `pg` driver. It uses `reviewed = true` condition to ensure only list of reviewed contents is fetched.
4. **Database Processing:** PostgreSQL executes query and returns result set that matches the condition.
5. **Data Transformation:** Frontend of inNuCE processes raw data. This includes splitting keyword strings, converting relative paths to full URLs, and formatting dates.
6. **Response Construction:** Nginx adds security-related HTTP headers (configured via Helmet middleware) and CORS headers. Then it returns JSON response to browser.
7. **UI Rendering:** React component receives data and renders UI elements. These elements include research paper list, audio player, and PDF preview iframe.

3.3.3 Administrator System Implementation

Administrator system is divided into two parts: Frontend and Backend. They work together to provide efficient management capabilities.

Admin Frontend: Dashboard Design

Frontend dashboard is the control center for administrator system of platform. Design focuses on clarity and efficiency based on the regular manipuliation, CRUD.

1. Layout Architecture

The interface uses a traditional sidebar navigation system. There are five main modules: *Publication*, *Use Cases*, *Team Member*, *Hardware*, and *Tools*. Each module has an icon and explanation for quick recognition. The active item is highlighted to show current window location.

Main content area is on the right. It consists of an action bar and a data table. The action bar contains buttons like "Add New" and "Manage Filters". The data table displays content list. Column widths are optimized based on content type.

2. Use Case Management Interface

Figure 3.6 shows the actual design of Use Case Management page.

As shown in Figure 3.6, the table has four columns.

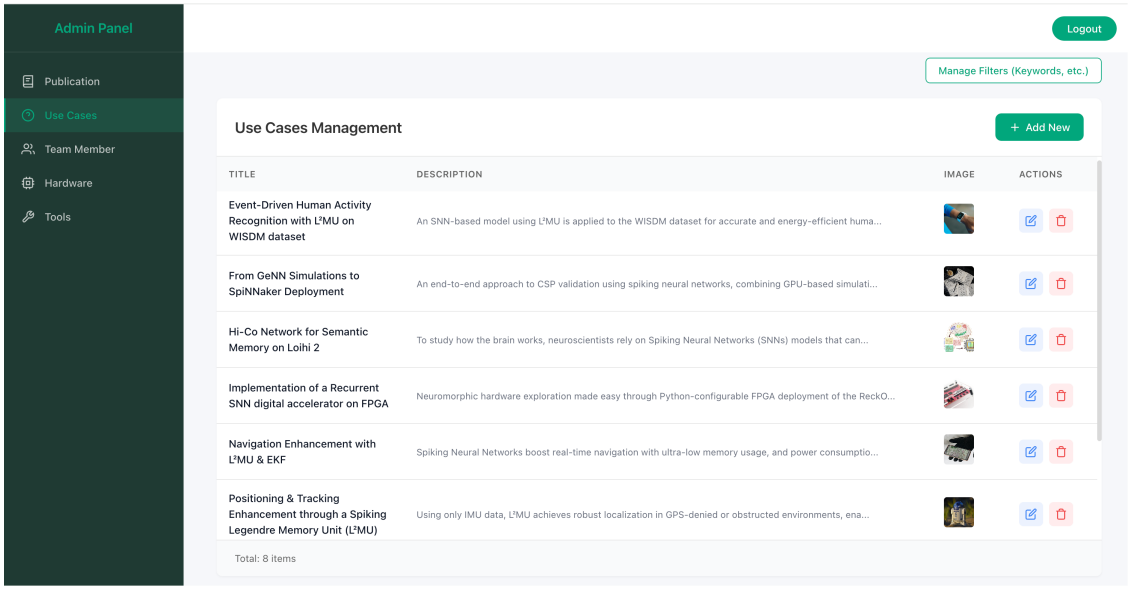


Figure 3.6: Admin Dashboard for Use Case Management

- **TITLE:** Displays full title of the case.
- **DESCRIPTION:** Shows a short summary. Long text is truncated with ellipsis to keep the UI clean.
- **IMAGE:** Displays a thumbnail preview. This helps admins identify cases visually.
- **ACTIONS:** Contains "Edit" (blue pencil) and "Delete" (red trash can) buttons. Color coding reduces the risk of wrong operation.

At the bottom, a statistic "Total: 8 items" is displayed. The "Add New" button is placed at top of right corner with a distinct green color to guide user action.

Admin Backend: CRUD Logic Implementation

The backend of administration system is built with Python, Flask, which provides standardized RESTful APIs for all resource types.

1. Unified API Pattern

All resources follow the same URL pattern: `GET /api/[resource]` for query, `POST` for creation, `PUT` for updates, and `DELETE` for removal. This consistency reduces development effort for frontend.

2. Publication CRUD Workflow

- **Create:** Frontend sends JSON data with paper metadata and 9 resource URLs. Backend sanitizes file paths first. Then it uses parameterized queries to insert data into PostgreSQL. This prevents SQL injection. The ID and timestamps are returned immediately via `RETURNING` clause.
- **Read:** List query supports multi-level sorting (e.g., by review status, then year). Single item query uses `LEFT JOIN` to fetch author information even if the list is empty.
- **Update (Smart Cleanup):** This is a key innovation. Before updating, system checks if any file path has changed. If a new file is uploaded, the old file is physically deleted from disk. This prevents "orphan files" and saves storage space. The whole process runs inside a database transaction to ensure consistency.
- **Delete (Cascading Cleanup):** Deleting a record triggers a full cleanup. System first queries all file paths associated with the record. Once database deletion is confirmed, it batch deletes all physical files.

3. Use Case CRUD Logic

Use Case data is further more complex because it has nested structures. Creating a case involves inserting data into multiple tables: `usecase`, `usecase_detail`, and `paragraph`. For updates, system uses a "Full Replacement" strategy for paragraphs. It deletes old paragraphs and inserts new ones. This simplifies the logic significantly.

Key Function: Upload PDF Interface

PDF upload is the trigger point for the whole automation pipeline.

1. Request Handling

The interface accepts `multipart/form-data` POST requests. Backend extracts file object using Flask `request.files`.

2. Security Validation

System uses a whitelist strategy. Only extensions in `ALLOWED_EXTENSIONS` set (like pdf, docx, png, mp4) are accepted. Others are rejected with HTTP 400 error. File size is also limited to 50MB to save bandwidth.

3. UUID Naming Strategy

Uploaded files are renamed using Python `uuid.uuid4()`. For example, `paper.pdf` becomes `c1b2...c1d4.pdf`. This strategy has three benefits:

1. **Uniqueness:** UUID collision probability is extremely low.

2. **Security:** Attackers cannot guess filenames to access unauthorized files.

3. **Path Safety:** It avoids special characters that might cause path traversal attacks.

4. Storage

Files are saved to `/app/assets` inside the Docker container. This directory is mounted to host machine via Docker Volume, so files persist even if container restarts or is deleted. Finally, the interface returns only the filename to frontend.

3.3.4 Automated Data Synchronization

The purple block in Figure 3.1, labeled "Auto Fetch Paper Workflow", represents an independent background service. Its goal is to synchronize research metadata from the IRIS system (Academic Information Management System of Politecnico di Torino). This ensures that the inNuCE administration system platform always displays the latest official records.

Working Principle

1. Data Source Connection

The IRIS system provides a public JSON API. This interface returns structured metadata for all published papers by specific id. The data includes basic info like title, year, journal name, and DOI. Crucially, it contains a `lastModified` timestamp. This field marks when the record was last updated in IRIS system.

2. Scheduling Mechanism

This synchronization service runs as an independent Docker container named `auto_workflow`. When container starts, it performs a full synchronization immediately. Then, it enters a loop. A Python script puts the process to sleep for 86,400 seconds (24 hours). After waking up, it triggers the task again.

3. Incremental Update Strategy

To avoid performance issues, system implements an intelligent update mechanism based on `source_last_modified`. The logic is as follows:

- When a record is fetched from IRIS, system checks local database using the `handle` field (unique ID).
- If the record does not exist locally, it is inserted as a "New Paper". The status of review is set to false.
- If the record exists, system compares two timestamps: `lastModified` from IRIS and `source_last_modified` from local database.
- An UPDATE operation is executed only if IRIS `lastModified` is newer. If they are same, the record is skipped.

This strategy improves performance significantly. At beginning of running, system processes about 80 records. But in daily updates, usually less than 10 records change. This reduces synchronization time from half minutes to about several seconds.

4. Author Recognition

IRIS API distinguishes between "internal authors" (Politecnico members) and "external authors". The system parses this information automatically, which links internal authors to the `innuce_schema.authors` table and generates links to their IRIS profiles. Also, `author_rank` is maintained to ensure authors are listed in correct order.

5. Consistency Guarantee

The whole process is protected by PostgreSQL transactions. A sync task may involve hundreds of INSERT/UPDATE operations. They are executed within a single transaction. If network error happens, transaction rolls back. This prevents database from having partial dirty data.

Architectural Benefits

This automation brings three main benefits:

- **Timeliness:** New papers entered in IRIS appear on inNuCE platform within 24 hours automatically.
- **Efficiency:** Administrators do not need to type paper titles or abstracts manually. They only need to review resource contents and upload PDF files that need to generate contents by AI.
- **Authority:** Metadata comes directly from official academic system, so information accuracy is guaranteed.

Through this workflow, the platform achieves "self-updating" capability. It keeps content fresh while reducing manual maintenance cost significantly.

3.4 Automated Intelligent Generation Pipeline

Automated Intelligent Generation Pipeline is the core innovation of inNuCE platform. The system converts academic PDF papers into 9 types of multimodal educational resources automatically. The pipeline uses modular design, and it covers task scheduling, document pre-processing, prompt engineering, generation, and data integration.

3.4.1 Task Scheduling and Queue Management

Figure 3.7 illustrates the task scheduling workflow.

1. Cron Job and Detection Logic

The system uses Linux Cron job to restart the container to check pending publications periodically in database. The Cron service runs as an independent Docker container (`cron_scheduler`). It executes the check script every 24 hours.

The script connects to the PostgreSQL database and runs a complex SQL query to filter records. Three conditions are checked: First, `full_paper_url` is not null (PDF uploaded). Second, at least one resource URL is NULL (resources need generation). Third, `task_status` is not 'processing' (avoid duplicate tasks).

For each valid record, the system checks 9 resource fields (e.g., `audio_url`, `video_url`). If a field is NULL, the corresponding type is added into the `types` list. Before pushing to the queue, the database record is updated to 'processing'.

Each task is a JSON object, and it contains `publication_id`, relative `pdf_path`, and `types` array.

2. Redis as Message Broker

Redis acts as a message broker, and it uses List data structure for task queue.

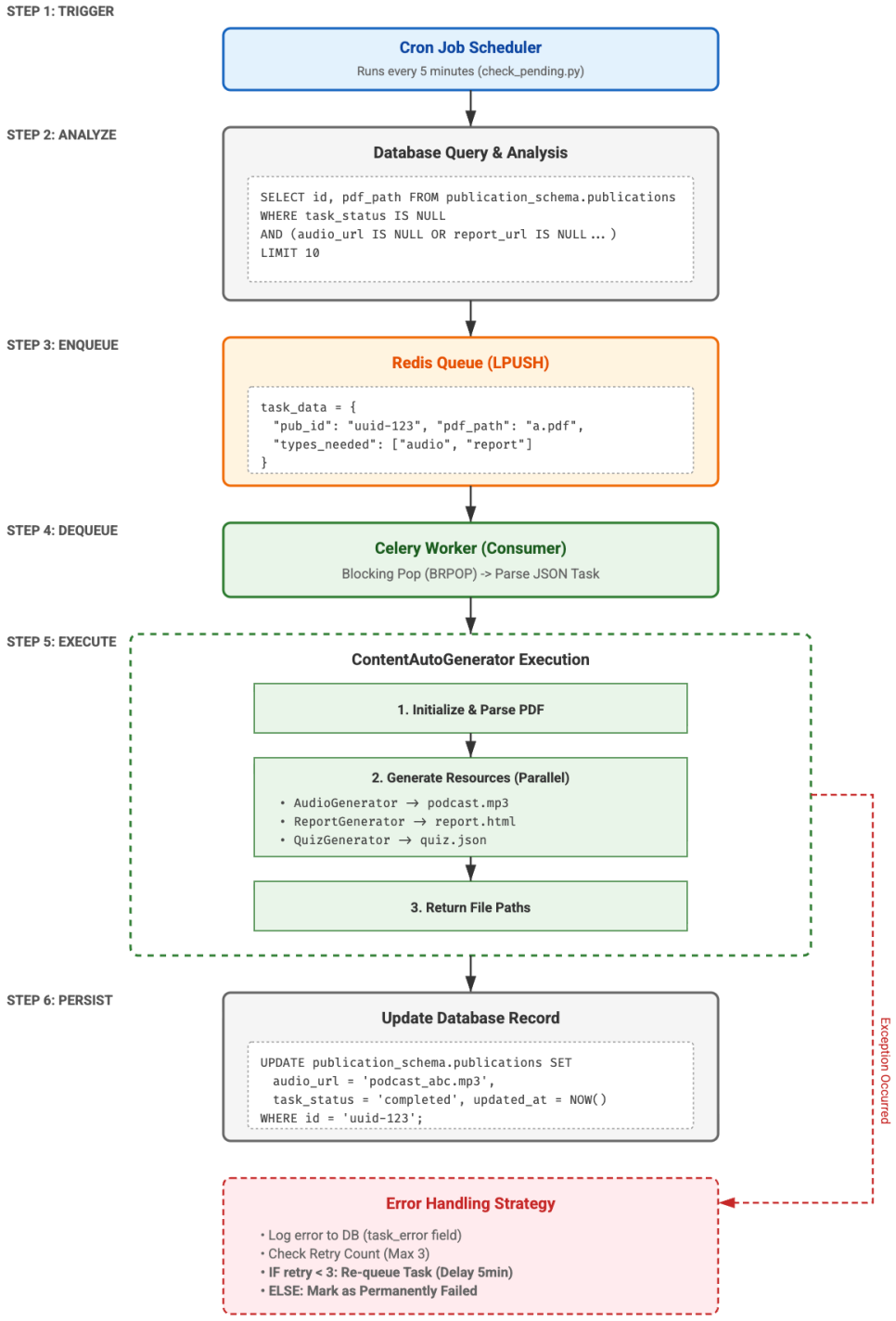


Figure 3.7: Task Scheduling and Asynchronous Execution Workflow

The Cron script pushes task JSON to the head of `content_generation_queue` list using `lpush` command. The Celery Worker pulls task from tail using `brpop` command. This ensures sequential processing. Redis container mounts the catalogue to the local file path. Every write operation is logged to file, so tasks are not lost even if the container restarts.

3. Celery Workers Configuration

Celery is used for executing async tasks. It runs as an individual Docker container (`celery_worker`). Environment variables configure connection to Redis and database.

The worker is started with `-concurrency=2`. This means one container processes 2 tasks simultaneously. It deploys 2 worker replicas in Docker Compose, so total concurrency is 4 tasks. This balance resource usage and speed because high concurrency might cause API rate limit or memory issue.

Worker pulls task, parses JSON, and initializes `ContentAutoGenerator`. Then it calls `generator.run()` to execute generation flow.

Tasks have `max_retries=3`. If execution fails (e.g. API timeout), the system retries for 3 times. Interval is 300 seconds. If all retries fail, task status is marked as 'failed' in database.

3.4.2 Document Pre-processing

Document pre-processing is the first stage of pipeline. It extracts structured text and metadata from PDF for LLM input. Figure 3.8 shows detailed workflow.

1. PDF Parsing Tech Stack

The system uses two Python libraries: `PyPDF2` and `pdfplumber`.

`PyPDF2` is used for metadata extraction. It reads document properties like Title, Author, and Creation Date quickly. It also counts total pages for progress tracking.

`pdfplumber` is used for text extraction. It performs better on complex academic layouts (e.g. two columns). Its `extract_text()` method keeps logical order of text.

2. Core Logic of PDFParser

`PDFParser` class handles the logic, which has three main steps inside `extract_all()` method:

- **Basic Info:** Open PDF with `PyPDF2` to get page count.
- **Text Extraction:** Iterate pages with `pdfplumber`. Extract text and join them with double newlines.
- **Metadata:** Extract metadata dictionary and standardize field names.

3. Text Cleaning

Raw text contains noise, so the system performs cleaning steps: removing extra blank lines, fixing PDF encoding issues (like ligatures), and preserving paragraph structure and sequence.

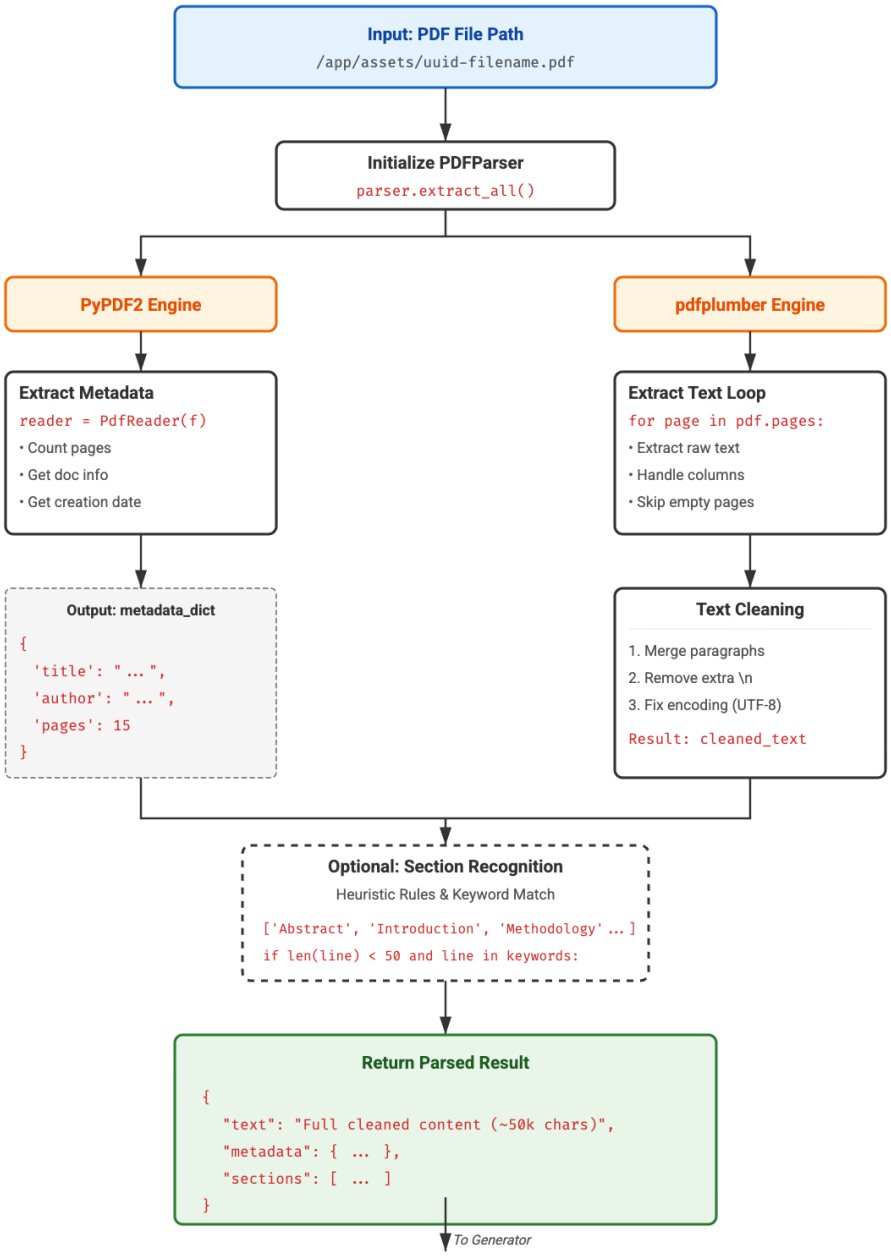


Figure 3.8: Detailed Workflow of PDF Document Parsing

4. Section Recognition (Optional)

`extract_sections()` method identifies chapters based on keywords. It uses a list of common titles like 'Abstract', 'Introduction'. Heuristic rules are applied: line length less than 50 chars, match keywords, and standalone line. This structured info helps LLM focus on specific parts.

3.4.3 Prompt Engineering Strategy

Prompt engineering is the core of high-quality content generation. It ensures that the LLM produces accurate and structured outputs. InNuCE system designs specialized prompt templates for 9 resource types. Each template is optimized iteratively.

1. Universal Structure of Prompt Template

All prompts follow a unified structure. This structure ensures that the LLM receives sufficient context and clear instructions. Figure 3.9 illustrates the five key components of a prompt.

As shown in Figure 3.9, the process starts with defining the role. Then it injects the paper content as a variable. After that, detailed task instructions and constraints are given. Finally, the output format is strictly defined to ensure machine readability.

2. Case Study: Audio (Podcast) Generation

The goal is to create a lively dialogue script. It makes complex academic content easy to understand.

- **Role Design:** It define two characters. The "Host" (Male voice) represents a curious layperson. He asks questions and uses analogies. He often shows emotions like surprise. The "Guest" (Female voice) is an academic expert. She explains concepts patiently and validates the Host's understanding.
- **Structure Instruction:** The dialogue must follow three stages. First is the "Hook", starting with a surprising fact. Second is the "Deep Dive", explaining the core "why" and "how". Third is "Implications", discussing real-world impact.
- **Style Control:** The style is defined as "NPR meets YouTube Explainer". It encourages colloquial expressions like "Wait, hang on..." to make the conversation feel authentic.
- **Format Control:** The output must be a strict JSON array. Each turn contains `speaker` and `text` fields. This is crucial for the subsequent Text-to-Speech (TTS) engine.

3. Case Study: Quiz Generation

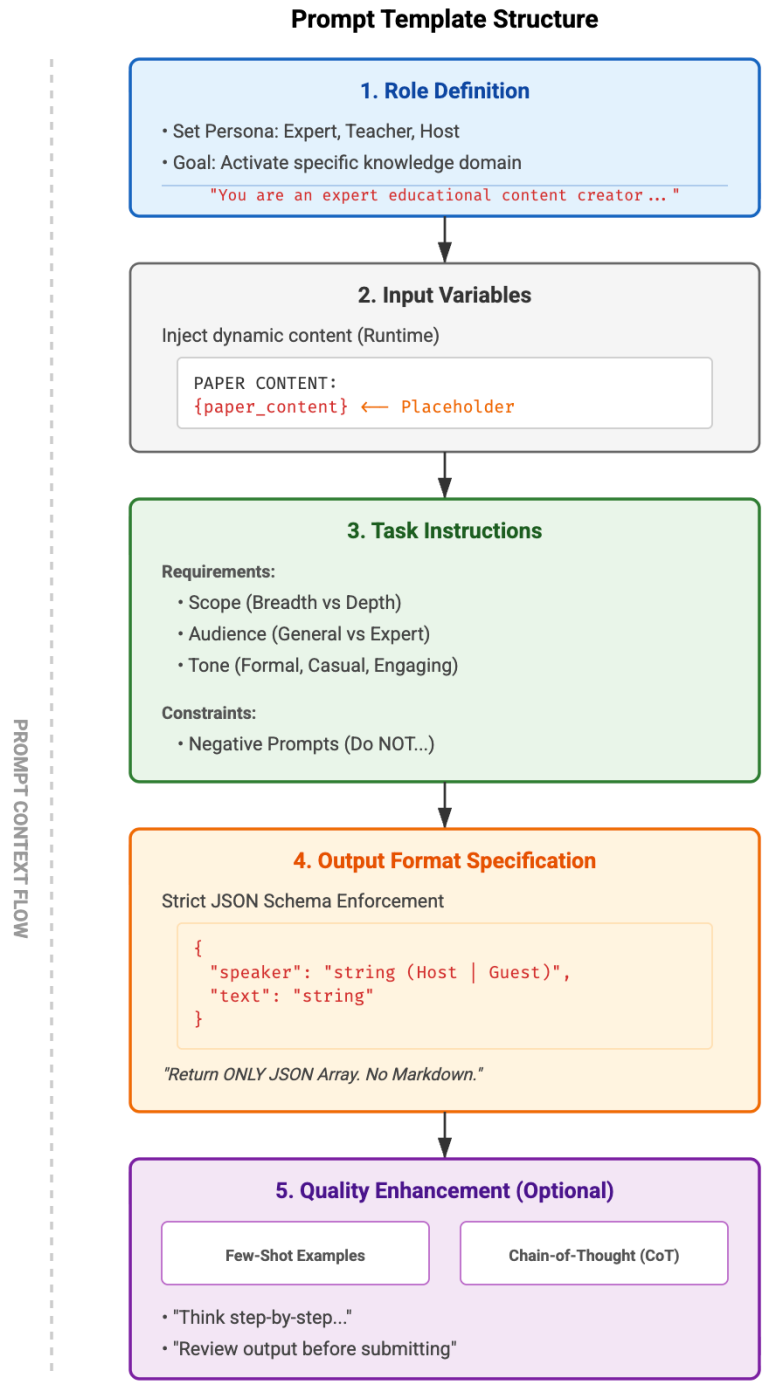


Figure 3.9: Systematic Structure of a Prompt Template

The goal is to create a comprehensive assessment. It covers multiple cognitive levels based on Bloom's Taxonomy.

- **Cognitive Levels:** The prompt requires questions to span from basic "Remember" (40%) to advanced "Evaluate" (5%). This ensures a balanced difficulty curve.
- **Question Types:** Most questions (60-75%) are Multiple Choice. The prompt requires high-quality distractors. These distractors should reflect common misconceptions, not just random errors. True/False questions (15-25%) focus on key facts. Short Answer questions (10-15%) assess deep understanding.
- **Educational Value:** Each question must include an `explanation` field. It explains why the correct answer is right and why others are wrong. A `hint` field is also required to guide thinking without giving away the answer directly.
- **Difficulty Balance:** The distribution is set to 40% Basic, 40% Medium, and 20% Advanced. This structure helps to motivate beginners while challenging advanced learners.

3.4.4 Multimodal Resource Generation

The system implements 9 different types of resource generators. They all inherit from a `BaseGenerator` abstract base class. This ensures a unified interface for LLM interaction while allowing specific implementation for each type.

1. Generator Architecture and Strategy Pattern

The `BaseGenerator` class acts as the "Context" in Strategy Pattern. It provides LLM initialization and a common method for calls.

- `_initialize_llm()`: This method selects LLM implementation based on `LLM_PROVIDER` environment variable. In this project, we primarily use Google Gemini (`gemini-2.5-pro`) because it supports long output up to 8000 tokens.
- `generate_with_prompt()`: This is the standard interface for all subclasses. It fills the prompt template with paper content using Python's string formatting. Then it sends the message to LLM and returns the raw text response.

2. Classification of Generators

Based on complexity and output format, the 9 generators are divided into three categories.

Category A: Text-to-JSON-to-Visual

These generators produce intermediate JSON data first. Then specialized plugins convert JSON into final visual formats.

- **AudioGenerator:** LLM outputs a JSON array with 'speaker' and 'text' fields. The system calls `TextToSpeech` plugin to generate audio clips using Edge TTS API. `MoviePy` library is then used to concatenate these clips into a full MP3 file. Intermediate JSON files are deleted to save space.
- **MentalMapGenerator:** LLM outputs graph definitions in DOT language. `DiagramGenerator` plugin renders it into PNG image using Graphviz engine.
- **PresentationGenerator (PPTX):** LLM outputs slide structure in JSON. `PresentationBuilder` plugin uses `python-pptx` library to create PowerPoint files.

Category B: Text-to-HTML

These generators ask LLM to generate full HTML+CSS code directly. This allows highly customized designs.

- **ReportGenerator:** It produces a self-contained HTML document with embedded CSS. The design focuses on academic style and responsive layout.
- **InfographicGenerator:** It creates creative HTML visualizations. The design is unique every time, using gradients and animations. Data visualization is implemented with CSS flex/grid directly

Category C: Text-to-JSON

The output of these generators is the final product. It is stored as JSON and rendered by frontend React components.

- **QuizGenerator:** It outputs a complete quiz object with questions, options, explanations, and hints.
- **FlashcardGenerator:** It generates an array of flashcards with front/back content and difficulty levels.
- **VideoGenerator:** It produces a video script with scenes, narration, and visual descriptions.
- **DatatableGenerator:** It extracts structured data tables from the paper.

3. Examples of Generated Output

Listing 3.3 shows examples of the generated JSON content.

The first example (Audio Script) demonstrates natural dialogue flow and emotional expression. The second example (Quiz) tests the "Apply" level of understanding. The explanation field provides educational value by clarifying why the answer is correct.

```
// 1. Audio Podcast Script Segment
[
  {
    "speaker": "Host",
    "text": "Welcome back to Deep Dive! Today, we're exploring a groundbreaking paper on neuromorphic computing."
  },
  {
    "speaker": "Guest",
    "text": "It's one of those papers that makes you rethink what's possible with hardware-software co-design."
  }
]

// 2. Quiz Question Object
{
  "id": 3,
  "type": "multiple_choice",
  "difficulty": "intermediate",
  "question": "Which application benefits MOST from the proposed architecture?",
  "options": [
    "Real-time object detection on battery-powered drones",
    "Training large language models in datacenters"
  ],
  "correct_answer": "A",
  "explanation": "The paper emphasizes ultra-low power consumption, making option A ideal."
}

```

Listing 3.3: Example Output: Audio Script and Quiz Question

3.4.5 Closed-Loop Data Integration

This section describes final stage of generation pipeline. It involves saving generated files to disk and updating database records. Figure 3.10 illustrates the complete workflow from result processing to frontend display.

1. File Saving Strategy

A utility class `FileHandler` manages file operations. It provides unified methods like `save_text()` for JSON/HTML and `save_binary()` for MP3/PNG images. All files are stored in `/app/assets` directory inside Docker container. This path is mounted to host machine via Docker Volume, so data persists in server after restart.

Files are organized by resource type (e.g., `/app/assets/audio/`). Filenames use UUID to ensure uniqueness, avoiding conflicts when multiple papers are processed simultaneously.

2. Database Update Mechanism

Before saving to database, system performs path normalization. As shown in "Step 1" of Figure 3.10, absolute paths like `/app/assets/pod.mp3` are converted to relative paths `pod.mp3`. This design decouples database data from server file system structure. Frontend can easily construct full URLs by adding base domain automatically.

3. Error Handling and Fault Tolerance

System handles partial failures robustly. If one resource fails (e.g., Quiz generation error), other successful resources (e.g., Audio) are still saved. The task status is updated to `'partial'` instead of `'failed'`. This is clearly visualized in the "PostgreSQL State" block of Figure 3.10, where Audio is marked green (Success) and Quiz is red (Failed). This ensures users can access whatever content is successfully generated. The most useful way is check the table of the admin system. If the resource generate successfully, corresponding column would be show the link of resources.

By implementing this closed-loop mechanism, inNuCE achieves end-to-end automation from PDF upload to multimodal resource publication.

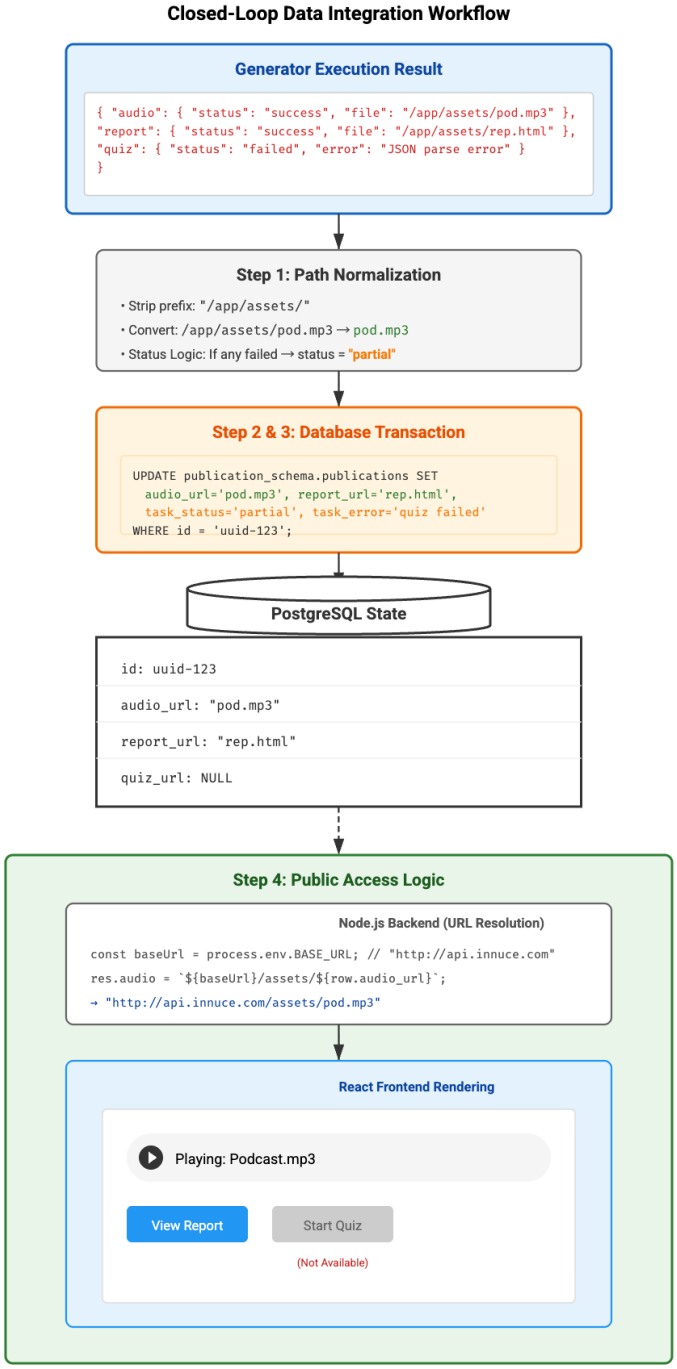


Figure 3.10: Closed-Loop Workflow: From Generation to Frontend Display

3.5 Containerization and Deployment

The inNuCE platform adopts Docker containerization technology to achieve service isolation, portability, and scalable deployment. The entire system architecture is based on micro-services concept. It encapsulates different functional modules into independent Docker containers and uses Docker Compose for unified orchestration.

3.5.1 Docker Architecture

Figure 3.11 illustrates the Docker architecture and volume mapping strategy.

Containerization Design

The system deploys 8 core containers. Each container has specific responsibility and communicates through custom Docker network.

1. Container Roles

- **nginx (Gateway):** Acts as unified entry point. It handles HTTP request routing and static resource service.
- **innuce-website-backend (javascript):** Handles API requests from public users (e.g. `/research`). It exposes port 3000 but only accessible without expose port to public Internet.
- **admin_website_backend (Flask):** Handles CRUD operations for administrators.
- **db (PostgreSQL):** Persists all business data.
- **redis (Queue):** Acts as message broker for Celery.
- **celery_worker:** Consumes tasks from Redis and executes content generation. We configure 3 replicas for parallel processing.
- **cron_scheduler:** Checks pending publications periodically.
- **auto_workflow:** Synchronizes metadata from IRIS system every 24 hours.

2. Orchestration with Docker Compose

`docker-compose.yml` defines the entire multi-container application architecture. It enables "one-click startup".

The `depends_on` field defines startup order. For example, database and redis start first, then backend services, and finally Nginx.

It uses two ways to manage environment variables. `env_file` is used for sensitive config like passwords. `environment` is used for communication config, such as `DB_HOST=db`. This utilizes Docker internal DNS for service discovery.

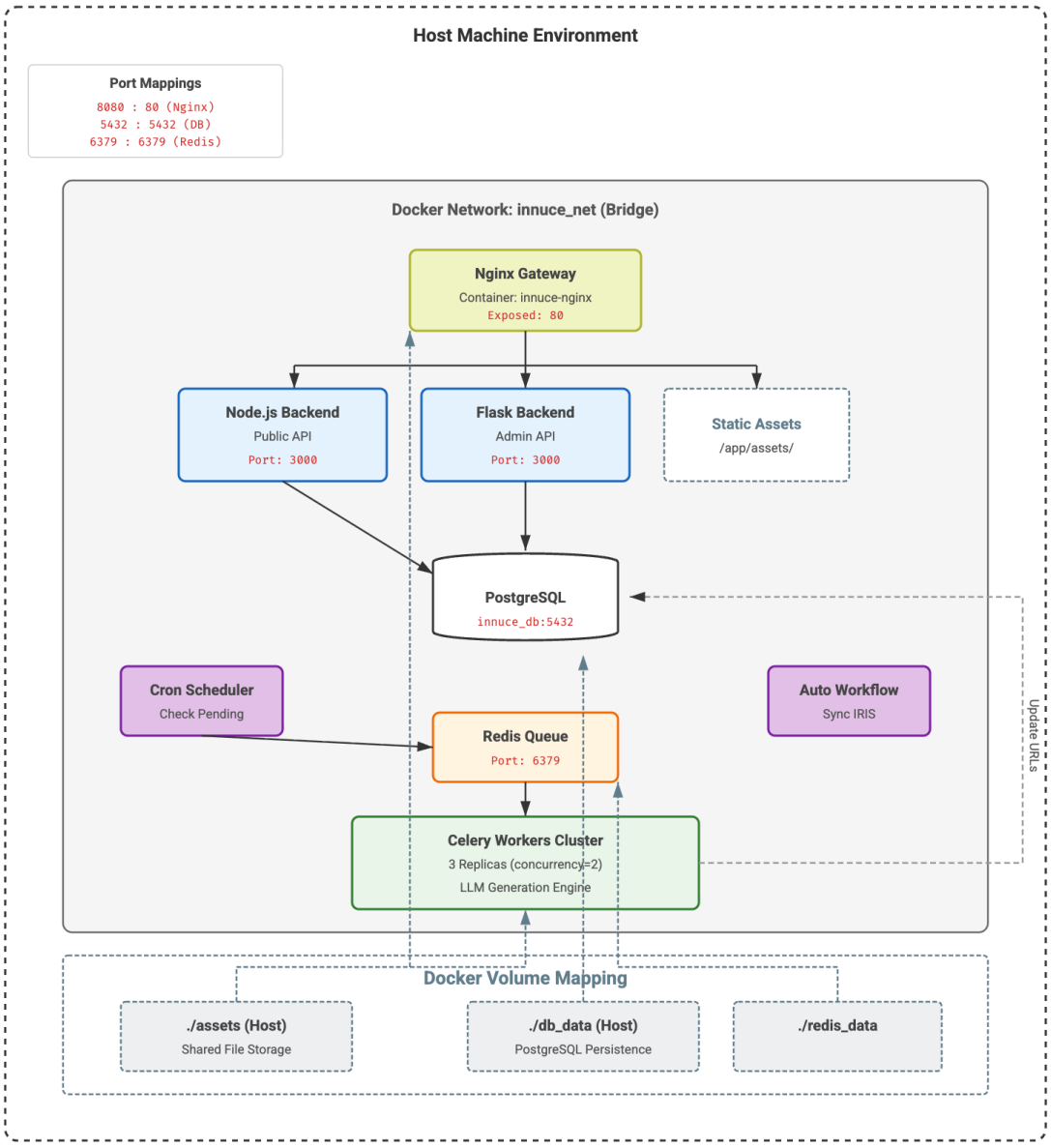


Figure 3.11: Docker Containers Network Topology and Volume Mounts

3.5.2 Volume Management and Persistence

Docker containers are stateless. To ensure data safety, system uses Docker Volume must mount host directories into containers.

Database Persistence Strategy

1. PostgreSQL Volume

It maps host directory `./db/postgresql` to container path `/var/lib/postgresql/data`. All table data and logs are stored here. This ensures data is retained even if container is deleted. Migration is also simple by copying this directory.

2. Redis Persistence

Redis uses AOF (Append Only File) mode. Write operations are logged to `appendonly.aof` file in `/data` directory (mounted from host). This provides better safety than RDB snapshot mode.

Shared Static Assets Mechanism

The `./assets` directory is the most complex mount point. It is accessed by three containers simultaneously, enabling a "Write Once, Read Many" pattern.

- **Flask Backend:** Writes uploaded PDF files to `/app/assets`.
- **Celery Worker:** Writes generated files (MP3, HTML) to `/app/outputs` (which points to same assets).
- **Nginx:** Reads files from `/usr/share/nginx/html/static/` for high-performance delivery.

3.5.3 Reverse Proxy Configuration

Nginx acts as the unified gateway. Listing 3.4 shows the core configuration.

```
1  server {
2      listen 80;
3      server_name localhost;
4      client_max_body_size 50M;
5
6      # 1. Route root requests to Node.js backend
7      location / {
8          include /etc/nginx/snippets/cors.conf;
9
10         proxy_pass http://innuce-website-backend:3000/;
11
12     }
13
14     # 2. Route /admin requests to Flask backend
15     location /admin {
16         include /etc/nginx/snippets/cors.conf;
```

```
17
18          proxy_pass http://admin_website_backend:3000/;
19
20      }
21
22      # 3. Serve static files directly from file system
23      location /assets {
24          include /etc/nginx/snippets/cors.conf;
25
26          # Map URL path to internal container path
27          alias /usr/share/nginx/html/static/;
28      }
29  }
```

Listing 3.4: Nginx Reverse Proxy Configuration

The configuration uses three `location` rules for intelligent routing:

- **Rule 1 (Public API):** Matches root path and proxies to inNuCE website backend. `proxy_pass` uses container name, which is resolved by Docker DNS.
- **Rule 2 (Admin API):** Routes requests starting with `/admin` to admin system backend.
- **Rule 3 (Static Files):** This is key for performance. Requests for `/assets` bypass backend logic and are served directly by Nginx from file system using `alias` directive.

3.6 Chapter Summary

This chapter detailed engineering implementation of inNuCE platform. It successfully built a robust, secure, and highly automated scientific dissemination system.

The design process covered three key pillars. First, dual-workflow architecture separates public access from administrator operations. This ensures security through SSH tunneling while maintaining high performance for public users. Second, robust web infrastructure is established. It used Node.js javascript for high-concurrency reading and Python flask for logic-heavy processing. PostgreSQL database with four schemas provides solid data foundation.

Third, and most importantly, automated intelligent generation pipeline is developed. By integrating Cron, Redis, Celery, and LLMs (Google Gemini), the system can transform static PDF papers into 9 types of multimodal resources automatically. The "Closed-Loop" mechanism ensures that generated content is safely stored and immediately available to admin to check and review the quality of the content.

Finally, entire system is containerized using Docker to develop and deploy. This makes deployment reproducible and scalable, avoding the environment conflict.

In the next chapter (Chapter 4: Results and Discussion), It will demonstrate actual performance of the system, which show screenshots of generated resources, evaluate quality of AI content.

Chapter 4

Results and Discussion

4.1 Multimodal Content Generation Outcomes

The automated pipeline successfully produced diverse digital assets. This section presents the visual and interactive outcomes generated from a sample paper ("An Ensemble Method for Calling and Ranking Somatic SVs"). Each resource type is analyzed in detail below.

4.1.1 Executive Summary Report

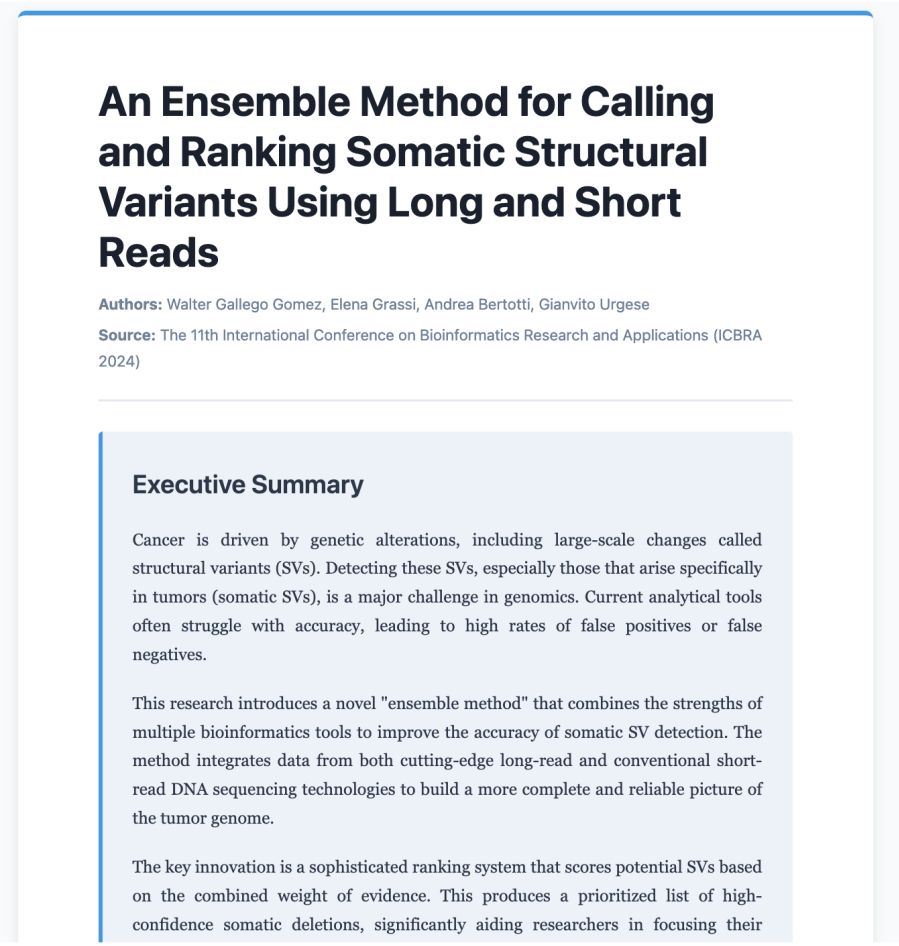
The Report Generator produces a structured HTML document. Figure 4.1 shows the rendered output.

As seen in the figure, the content is not just a plain text summary. It is organized into logical sections like "Executive Summary" and "Key Innovation". This proves that the LLM successfully identified the core contributions of the paper and filtered out redundant background information. The clean typography makes it much easier for readers to read than the original two-column PDF, which make the paper friendly dissemination.

4.1.2 Mental Map Visualization

Understanding the logical flow of a complex algorithm is difficult. Figure 4.2 shows the generated Mental Map.

The graph clearly illustrates the hierarchical relationship from "Problem" to "Methodology" and "Solution". This demonstrates that the system captured the logical hierarchy of the research. It successfully decomposed the complex "Ensemble Method" into visual branches, helping readers grasp the structural picture instantly.



An Ensemble Method for Calling and Ranking Somatic Structural Variants Using Long and Short Reads

Authors: Walter Gallego Gomez, Elena Grassi, Andrea Bertotti, Gianvito Urgese

Source: The 11th International Conference on Bioinformatics Research and Applications (ICBRA 2024)

Executive Summary

Cancer is driven by genetic alterations, including large-scale changes called structural variants (SVs). Detecting these SVs, especially those that arise specifically in tumors (somatic SVs), is a major challenge in genomics. Current analytical tools often struggle with accuracy, leading to high rates of false positives or false negatives.

This research introduces a novel "ensemble method" that combines the strengths of multiple bioinformatics tools to improve the accuracy of somatic SV detection. The method integrates data from both cutting-edge long-read and conventional short-read DNA sequencing technologies to build a more complete and reliable picture of the tumor genome.

The key innovation is a sophisticated ranking system that scores potential SVs based on the combined weight of evidence. This produces a prioritized list of high-confidence somatic deletions, significantly aiding researchers in focusing their

Figure 4.1: Generated Executive Summary Report

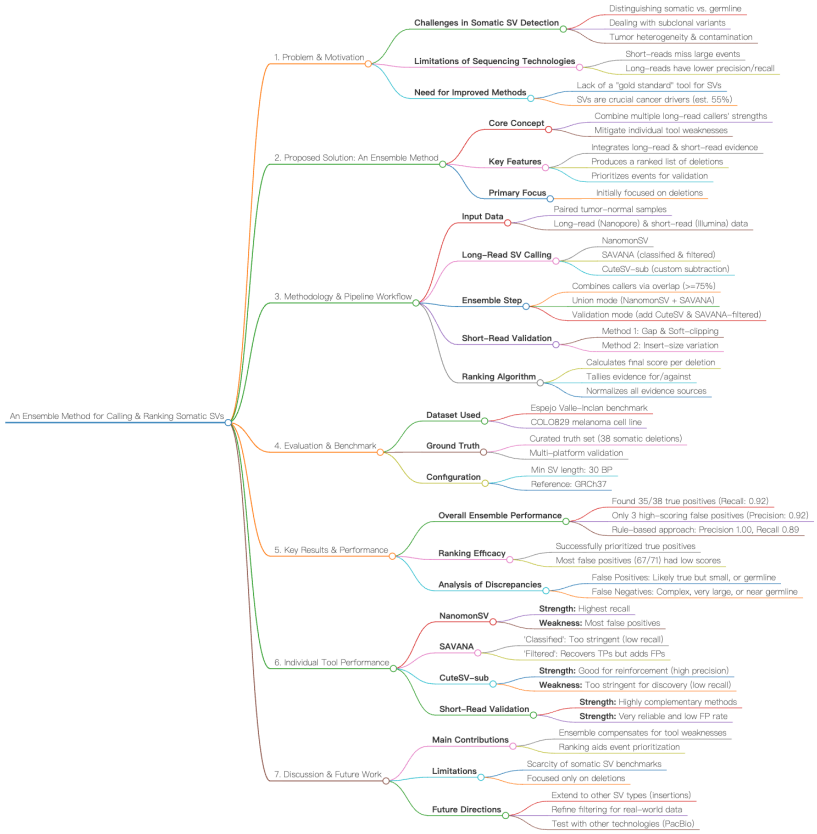


Figure 4.2: Generated Mental Map of Research Structure

4.1.3 Interactive Assessment: Quiz

To test user understanding, the system generates interactive quizzes. Figure 4.3 displays the quiz interface.

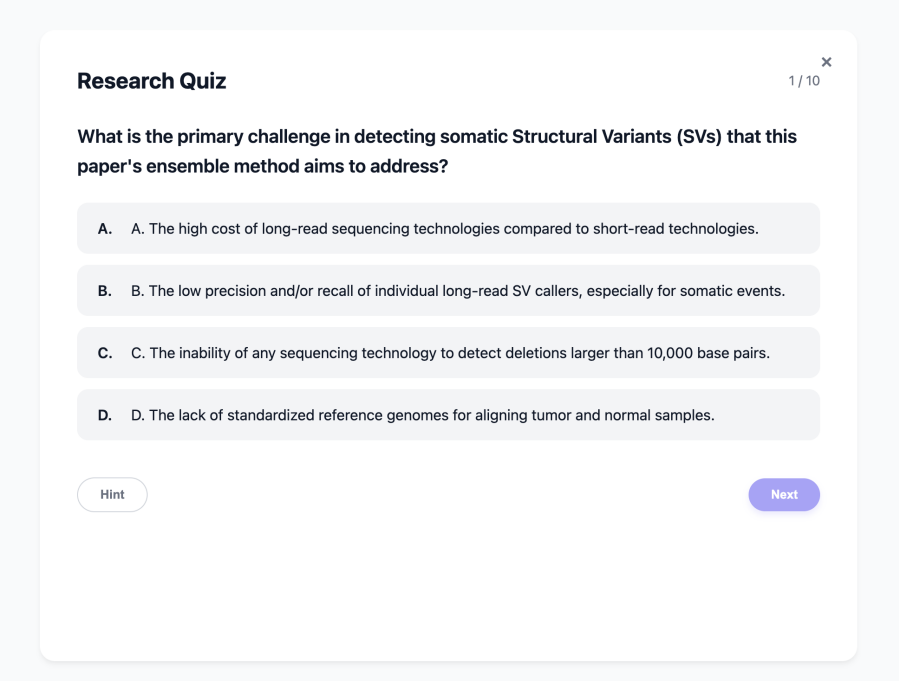


Figure 4.3: Interactive Quiz with Explanation

The highlight of this result is the "Explanation" feedback. When a user selects an option, the system provides a detailed reason derived from the paper's findings. This shows "Pedagogical Intelligence". It does not just judge right or wrong, but also reinforces learning by explaining the underlying logic of Somatic SV detection challenges.

4.1.4 Knowledge Reinforcement: Flashcards

For quick revision of key concepts, Flashcards are generated. Figure 4.4 shows the card interface.

The card front displays a specific term (e.g., "Somatic Structural Variants"), and the back provides its precise definition in the context of cancer genomics. This proves the system's ability to perform precise Information Extraction (IE), identifying key "Concept-Definition" pairs from unstructured text.

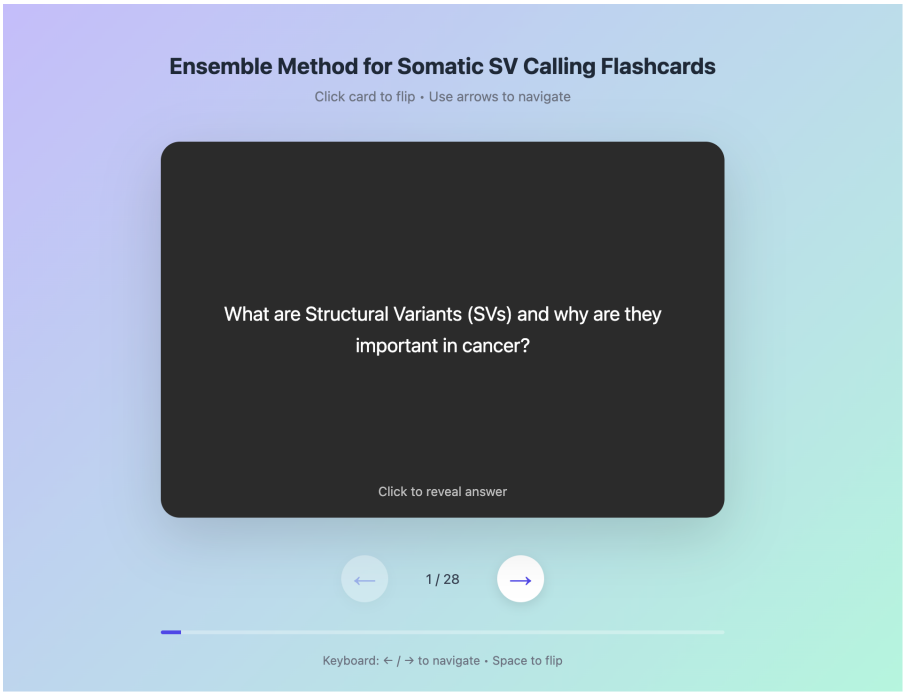
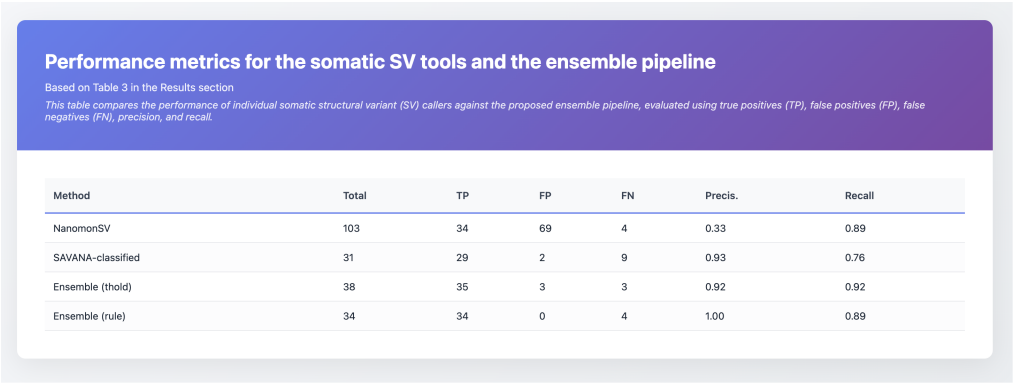


Figure 4.4: Flashcard for Concept Definition

4.1.5 Data Extraction: Performance Table

Extracting data from PDF tables is technically challenging. Figure 4.5 shows the reconstructed data table.



Performance metrics for the somatic SV tools and the ensemble pipeline

Based on Table 3 in the Results section

This table compares the performance of individual somatic structural variant (SV) callers against the proposed ensemble pipeline, evaluated using true positives (TP), false positives (FP), false negatives (FN), precision, and recall.

Method	Total	TP	FP	FN	Precis.	Recall
NanomonSV	103	34	69	4	0.33	0.89
SAVANA-classified	31	29	2	9	0.93	0.76
Ensemble (thold)	38	35	3	3	0.92	0.92
Ensemble (rule)	34	34	0	4	1.00	0.89

Figure 4.5: Extracted Performance Metrics Table

The system successfully extracted the "Precision" and "Recall" metrics for different methods (NanomonSV, SAVANA, etc.). It correctly mapped the numerical values to the corresponding tools. This allows researchers to compare experimental results directly on the web page without searching through the dense PDF document, which prompt the efficiency.

4.1.6 Visual Insight: Infographic

To make data visually appealing, the system generates Infographics. Figure 4.6 shows the visual presentation of key statistics.

The infographic summarizes the "Challenge" and "Solution" using distinct visual blocks. It uses high-contrast colors to highlight critical information like "Low Precision & Recall". This transformation from text to visual layout significantly enhances the engagement level for non-expert audiences.

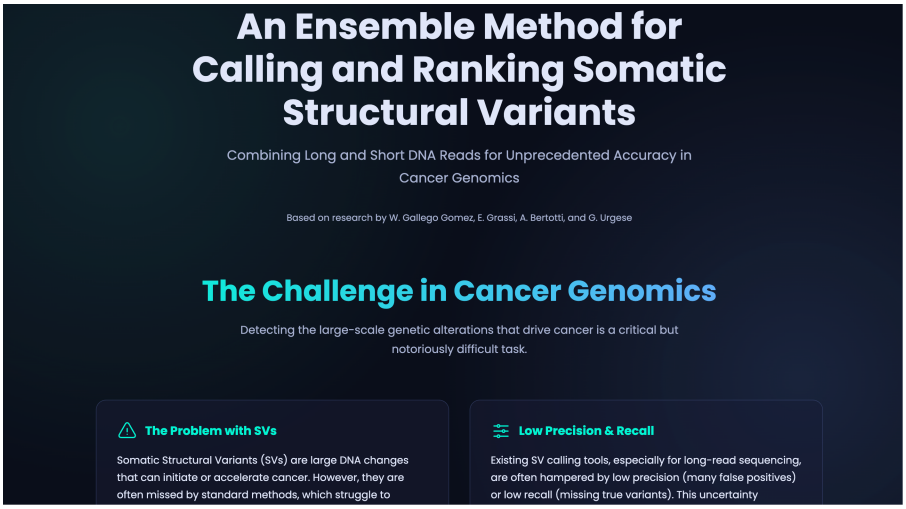


Figure 4.6: Visual Presentation of Research Highlights

4.2 Discussion

This section evaluates and show the performance and quality of the generated resource content by AI. It discusses the engineering strengths we achieved and the technical limitations we faced during development.

4.2.1 System Strengths

The results demonstrate three major engineering strengths:

- **Automation:** The system achieves a true "Upload & Forget" workflow. Once the admin uploads a PDF, the backend handles everything. No human intervention is needed until the final review.
- **Scalability:** The architecture is built on Docker and Celery. This allows the system to process multiple papers at the same time. The message queue ensures that the server does not crash under high load.
- **Accessibility:** By converting static PDFs into audio and quizzes, the platform lowers the barrier for reading. It makes scientific content available to more people.

4.2.2 Engineering Limitations and Challenges

Despite the success, there are technical challenges in the current implementation.

1. Stability of LLM Output Format

The most critical issue for a developer is the unpredictability of LLM. It instructed the model to return strict JSON format. However, it sometimes fails. For example, it might add a trailing comma in the list or include markdown code blocks (like "‘`json`). Although it added regex cleaning code, parsing errors still happen occasionally. This requires the task to be retried.

2. PDF Layout Parsing Constraints

The system uses the `pdfplumber` library. It works well for standard single-column text. However, many academic papers use complex double-column layouts or special fonts. In these cases, the text extraction might be disordered. The generator relies on this text, so if the extraction is terrible, the generated content will be poor.

3. External API Dependency

The entire pipeline relies on Google Gemini API. This introduces external risks. If the network connection to Google is unstable, or if the API rate limit is reached, the generation task will fail. It implemented a retry mechanism in Celery to mitigate this, but the dependency is still a bottleneck for system stability. What's more, the ability to AI reasoning is crucial as well, which determine the quality of the output directly.

Chapter 5

Conclusions

This thesis presented the design and development of inNuCE web platform. The primary objective was to address the specific challenges highlighted in the Abstract: the difficulty for non-expert audiences to understand static academic papers, and the inefficiency of manual website maintenance. Through the construction of a full-stack system with an automated AI pipeline, these goals are largely achieved.

The project successfully delivered a robust infrastructure based on a dual-workflow architecture. By separating the inNuce website backend from the administrative system backend, the system ensures both security and high performance. The core innovation lies in the Automated Intelligent Generation Pipeline. By integrating LangChain, Redis, and Celery, the system transforms a static PDF file into 9 distinct types of multimodal resources. As demonstrated in the Results chapter, these resources range from audio podcasts to interactive quizzes, effectively turning passive reading into an active learning experience.

The implementation results directly respond to the problems of data latency and high management costs. The automation mechanism, including the Cron scheduler and "Auto Fetch" workflow, ensures that the platform content remains synchronized with the official IRIS database. The processing time for a single paper is reduced to approximately one minute. This validates that the "Closed-Loop" data integration strategy is practical and efficient.

However, there are technical limitations and challenges in the current implementation. First, the stability of LLM output is not totally guaranteed. As discussed, the model occasionally generates invalid JSON formats (like trailing commas), which causes parsing errors in the backend. Second, the PDF parsing library depends on standard document layouts. It struggles with complex multi-column structures or non-standard fonts, leading to incomplete text extraction. Finally, the system relies heavily on third-party APIs (Google Gemini). Changes in API rate limits or pricing policies could affect the long-term sustainability of the platform. Future maintenance will need to focus on improving error handling mechanisms to mitigate these risks.

In summary, the inNuCE platform offers a concrete reference for applying Generative AI in web applications. It successfully bridges the gap between complex academic research and the general public, providing a scalable solution for modern scientific dissemination.

Acknowledgments

First, I would like to express my deepest gratitude to my supervisors, **Professor Gianvito Urgese** and **Doctor Giuseppe Fanuli**. Their guidance and patience were very important during this thesis. They helped me solve many technical problems and gave me clear directions when I was confused.

I also want to thank all the professors who taught me during my master's study. The knowledge I learned from their courses built a solid foundation for my research.

I am deeply grateful to my parents. Their unconditional support and love made it possible for me to complete my education. They always encouraged me when I faced difficulties.

Thanks to all my friends who accompanied me during this journey. The time we spent together studying and relaxing will be a cherished memory.

This two-and-a-half-year journey has been an unprecedented period of growth for me. I started with zero coding ability. But now, I can identify needs in real-life scenarios, propose my own solutions, and implement them using modern technologies. This transformation makes me feel proud and confident.

Finally, I hope I can apply what I have learned to solve real-world problems and keep improving as a Product Manager with a coding background. I wish myself a bright future.

Bibliography

[1] Lutz Bornmann and Rüdiger Mutz. «Growth rates of modern science: A bibliometric analysis based on the number of publicationss and cited references». In: *Journal of the Association for Information Science and Technology* 66.11 (2015), pp. 2215–2222. DOI: 10.1002/asi.23329.

[2] Wayne Xin Zhao et al. «A Survey of Large Language Models». In: *arXiv preprint arXiv:2303.18223* (2023). URL: https://arxiv.org/abs/2303.18223.

[3] Nelson F. Liu et al. «Lost in the Middle: How Language Models Use Long Contexts». In: *Transactions of the Association for Computational Linguistics* 12 (2024). (Preprint available as arXiv:2307.03172 in 2023), pp. 157–173.

[4] Nicola Josifoski Martin andity, Maxime Peyrard, and Robert West. «Exploiting Structured Knowledge in Text-to-SQL Generation». In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Singapore: Association for Computational Linguistics, 2023, pp. 1337–1353.

[5] Saleem Amershi et al. «Guidelines for Human-AI Interaction». In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, 2019, pp. 1–13. DOI: 10.1145/3290605.3300233.

[6] Roger S. Pressman and Bruce R. Maxim. *Software Engineering: A Practitioner's Approach*. 8th. New York, NY: McGraw-Hill Education, 2014.

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd. Boston, MA: Addison-Wesley Professional, 2012.

[8] Nicola Dragoni et al. «Microservices: Yesterday, Today, and Tomorrow». In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[9] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 978-1491950357.

[10] Roy Thomas Fielding. «Architectural Styles and the Design of Network-based Software Architectures». PhD thesis. University of California, Irvine, 2000.

[11] Tim O'Reilly. *What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software*. Accessed: 2024-01-01. 2005. URL: https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html.

[12] Rishi Bommasani et al. «On the Opportunities and Risks of Foundation Models». In: *arXiv preprint arXiv:2108.07258* (2021). URL: https://arxiv.org/abs/2108.07258.

[13] X. Xu et al. «A Survey on Efficient Inference for Large Language Models». In: *arXiv preprint arXiv:2402.01153* (2024). URL: https://arxiv.org/abs/2402.01153.

[14] Ashish Vaswani et al. «Attention is All you Need». In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[15] Tom Brown et al. «Language Models are Few-Shot Learners». In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 1877–1901.

[16] Jason Wei et al. «Emergent Abilities of Large Language Models». In: *Transactions on Machine Learning Research* (2022). URL: https://openreview.net/forum?id=yzkSU5zdwD.

[17] OpenAI. «GPT-4 Technical Report». In: *arXiv preprint arXiv:2303.08774* (2023).

[18] Grégoire Mialon et al. «Augmented Language Models: a Survey». In: *arXiv preprint arXiv:2302.07842* (2023).

[19] Harrison Chase. *LangChain*. Open Source Software. 2022. URL: https://github.com/langchain-ai/langchain.

[20] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. «AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts». In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 2022, pp. 1–22.

[21] Shunyu Yao et al. «ReAct: Synergizing Reasoning and Acting in Language Models». In: *International Conference on Learning Representations (ICLR)*. 2023.

[22] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley Professional, 2002. URL: https://martinfowler.com/books/pea.html.

[23] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. URL: https://www.enterpriseintegrationpatterns.com/.

[24] Evi Nemeth et al. *UNIX and Linux System Administration Handbook*. 5th. Addison-Wesley Professional, 2017. URL: https://www.admin.com/.

[25] Dirk Merkel. «Docker: lightweight linux containers for consistent development and deployment». In: *Linux Journal* 2014.239 (2014), p. 2. URL: https://dl.acm.org/doi/10.5555/2600239.2600241.

[26] Jules White et al. «A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT». In: *arXiv preprint arXiv:2302.11382* (2023). URL: https://arxiv.org/abs/2302.11382.

[27] Pengfei Liu et al. «Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing». In: *ACM Computing Surveys* 55.9 (2023), pp. 1–35. URL: https://dl.acm.org/doi/10.1145/3560815.

[28] D. Sculley et al. «Hidden Technical Debt in Machine Learning Systems». In: *Advances in Neural Information Processing Systems*. Vol. 28. 2015. URL: https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf.

[29] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

[30] Yunfan Gao et al. «Retrieval-Augmented Generation for Large Language Models: A Survey». In: *arXiv preprint arXiv:2312.10997* (2023). URL: https://arxiv.org/abs/2312.10997.

[31] Ziwei Ji et al. «Survey of Hallucination in Natural Language Generation». In: *ACM Computing Surveys* 55.12 (2023), pp. 1–38. URL: https://dl.acm.org/doi/10.1145/3571730.

[32] Yue Zhang et al. «Siren's Song in the AI Era: A Survey on Hallucinations in Large Language Models». In: *arXiv preprint arXiv:2309.01219* (2023). URL: https://arxiv.org/abs/2309.01219.

[33] Xinru Wang and Ming Yin. «Are Explanations Helpful? A Comparative Study of the Effects of Explanations in AI-Assisted Decision-Making». In: *Proceedings of the 26th International Conference on Intelligent User Interfaces* (2021), pp. 318–328. URL: https://dl.acm.org/doi/10.1145/3397481.3450650.

[34] Google Gemini Team. «Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context». In: *arXiv preprint arXiv:2403.05530* (2024). URL: https://arxiv.org/abs/2403.05530.

[35] Rishabh Gargeya and Vibhav Rastogi. «Interoperability in Cloud Computing: Issues and Solutions». In: *International Journal of Computer Applications* 183.11 (2021), pp. 25–30. URL: https://www.ijcaonline.org/archives/volume183/number11/30282-2021921434.

[36] *Node.js*. `https://nodejs.org/`. Accessed: 2025-09-25.

[37] *Flask*. `https://flask.palletsprojects.com/`. Accessed: 2025-11-24.